Stock Market Insights With AI Agent using Google GenAI & ChromaDB

code Breakdown and Workflow Explanation

Kanukollu Goutham (804130) AND Bishweshwar Sukla(804133)

Objective

Goal: Build an AI-powered stock market agent using Google Generative AI, LangChain, and ChromaDB. Our motivation behind it is the working professional who work tirelessly during the and don't get time to look into market, after coming back home it is essential for them to spend time with their family,so not to compromise with family time and investment we built something so they can quickly know what happen in the market

Capabilities: Fetch, store, and retrieve market data; process user questions; provide relevant financial insights.

Links

Github Link - https://github.com/bishweashwarsukla/project_agent

Docker Image Link - https://hub.docker.com/r/kanukollugvt/flasktest-app

Sample output for our UI

Deploy

You can now ask questions now.

or

you can update knowledge data base using below button 👇

Update Knowledge Base

Enter your question about stocks, finance, or cryptocurrency:

is adani green bearish?

Submit

Answer:

Based on the provided data, Adani Green shows a bearish trend. At 11:13 AM, its share price was down 0.92%, and by 11:39 AM, the decline increased to 0.76%. While these are relatively small percentage changes, both updates indicate a downward movement in the share price, suggesting bearish sentiment in the short term.

Main Libraries Used

Google Generative AI: For embeddings and language model interactions.

LangChain & LangGraph: To create state-based workflows and manage prompt handling.

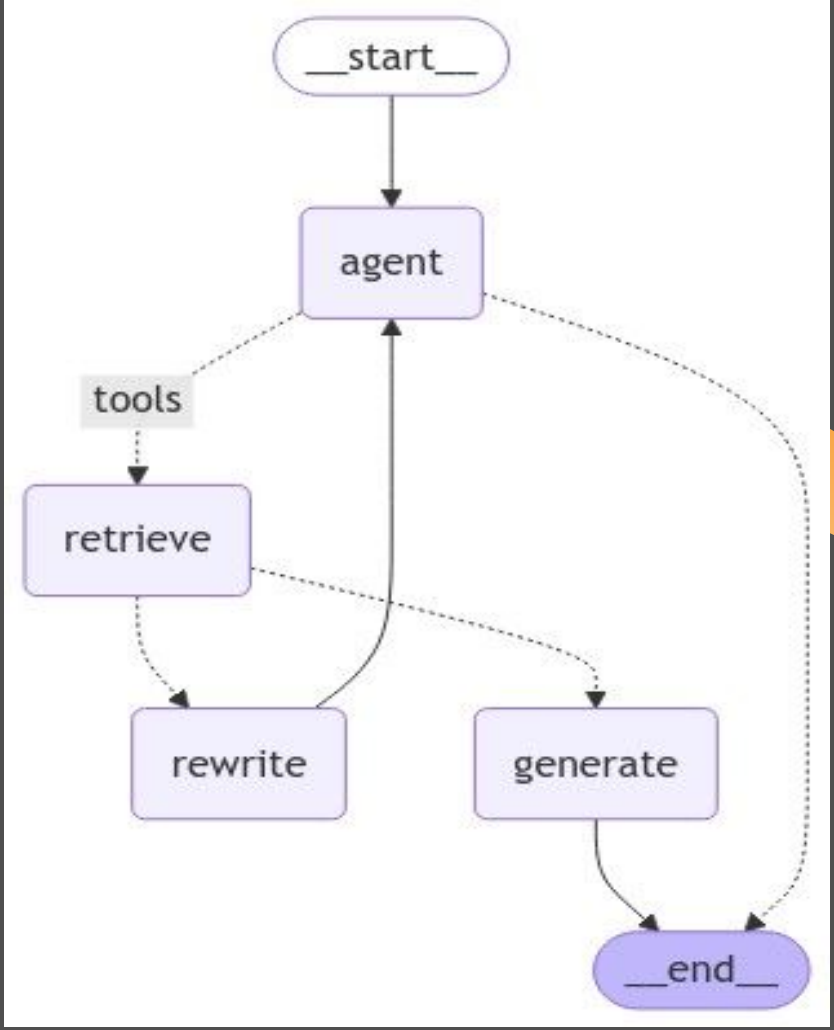
ChromaDB: Manages vector storage and retrieval.

Environment Configuration: dotenv for loading URLs and API keys from environment files.

Streamlit

Github Actions

Agent overview



Workflow Overview

High-Level Flow:

- Data Source & Loading: Load documents from finance news sites.
- Vector Database Setup: Embed documents and store them in ChromaDB.
- Retrieval & Processing: Query vector database for insights.
- Response Generation: Leverage LLM to respond based on query results.


INPUT
STREAMLIT APP
AGENT
DON'T KNOW
KNOW
MONEY CONTROL/ YAHOO FINANCE/ ECONOMICC TIMES
LANGCHAN WEB BASED CODER
GOOGLE GEMINI EMBEDDINGS
CHROMA DB
RELEVANCY CHECKER (GRADE DOCUMENTS)
YES
NO
GENERATE CHAT GOOGLE GEN AI
REWRITE GOOGLE GEN AI
END

Module 1: Vector Database Loading & Setup

Function: load_vector_db()

- Loads a prebuilt Chroma vector store for document retrieval.
- Creates a retriever tool to answer queries related to stock and finance topics.

Function: build_vector_db()

- Builds a new Chroma vector store from scratch.
- Embeds documents using GoogleGenerativeAIEmbeddings.

Purpose: Store and retrieve financial news articles and analysis efficiently.

```python
def load_vector_db():
    load_dotenv()
    embeddings = GoogleGenerativeAIEmbeddings(model="models/embedding-001")

    VECTOR_DB_PATH = "./agentDB"

    vectorstore = Chroma(
        collection_name="rag-chroma",
        persist_directory=VECTOR_DB_PATH,
        embedding_function=embeddings,
    )
    retriever = vectorstore.as_retriever()

    retriever_tool = create_retriever_tool(
        retriever,
        "retriever_stock_market_updates",
        "Use the given documents to provide insights on stocks, finance, interest rates, bitcoin, real estate, news, bullish and bearish trends, etc.",
    )

    tools = [retriever_tool]
    results = vectorstore.similarity_search(
        "bullish stocks",
        k=3,
    )
    for res in results:
        print(f"* {res.page_content} [{res.metadata}]")

    return retriever_tool, retriever, tools, vectorstore
```

```python
def build_vector_db():
    docs = [WebBaseLoader(url).load() for url in urls]
    docs_list = [item for sublist in docs for item in sublist]

    VECTOR_DB_PATH = "./agentDB"

    text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
        chunk_size=100, chunk_overlap=20
    )
    doc_splits = text_splitter.split_documents(docs_list)

    # Pass this client to the Chroma store
    vectorstore = Chroma.from_documents(
        documents=doc_splits,
        collection_name="rag-chroma",
        embedding=embeddings,
        persist_directory=VECTOR_DB_PATH,
    )

    retriever = vectorstore.as_retriever()
    retriever_tool = create_retriever_tool(
        retriever,
        "retriever_stock_market_updates",
        "Use the given documents to provide insights on stocks, finance, interest rates, bitcoin, real estate, news, bullish and bearish trends, etc.",
    )
    tools = [retriever_tool]

    return retriever_tool, retriever, tools, vectorstore
```

Module 2: Text Processing & Chunking

- Text Splitter:
  - RecursiveCharacterTextSplitter: Chunks documents into smaller pieces to fit within token limits for embedding.
  - Chunk Size: Set to 100 characters with a 20-character overlap to improve context consistency across chunks

```
text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=100, chunk_overlap=20
)
doc_splits = text_splitter.split_documents(docs_list)

# Pass this client to the Chroma store
```

Module 3: Process User Input

- **Function**: `process_user_input()`
  - Central function to handle user questions.
  - Loads embeddings and model; processes user input with the `agent`, `retrieve`, `rewrite`, and `generate` functions.
  - Uses `ChatGoogleGenerativeAI` model `gemini-1.5-pro` for question interpretation and response generation.

```python
def process_user_input(input_text, retriever_tool, retriever, tools, vectorstore):

    load_dotenv()

    embeddings = GoogleGenerativeAIEmbeddings(model="models/embedding-001")

    llm = ChatGoogleGenerativeAI(
        model="gemini-1.5-pro",
        temperature=0,
        max_tokens=None,
        timeout=None,
        max_retries=2,
    )

    class AgentState(TypedDict):
        messages: Annotated[Sequence[BaseMessage], add_messages]

    def grade_documents(state) -> Literal["generate", "rewrite"]: ...
        return "generate" if scored_result.binary_score == "yes" else "rewrite"

    def agent(state):
        model = ChatGoogleGenerativeAI(model="gemini-1.5-pro", temperature=0)
        model = model.bind_tools(tools)
        return {"messages": [model.invoke(state["messages"])]}

    def rewrite(state): ...
        return {"messages": [model.invoke([prompt_msg])]}

    def generate(state): ...
        return {"messages": [response]}
```

Core Agent Functions (Within `process_user_input`)

Agent Function:

- **Primary Logic**: Manages interactions between the user input, vector database, and LLM.
- Binds tools to interact with LangGraph workflow.

Document Grading (`grade_documents`):

- Evaluates relevance of retrieved documents to the user's question, generating or rewriting the response if needed.

Rewrite Function:

- **Purpose**: Reformulates the user question if initial query fails to retrieve relevant data.

Generate Function:

- Uses retrieved documents to generate a comprehensive answer for the user based on the relevant insights retrieved.

```python
def grade_documents(state) -> Literal["generate", "rewrite"]:
    class Grade(BaseModel):
        binary_score: str = Field(description="Relevance score 'yes' or 'no'")

    model = ChatGoogleGenerativeAI(model="gemini-1.5-pro", temperature=0)
    llm_with_tool = model.with_structured_output(Grade)

    prompt = PromptTemplate(
        template="Evaluate relevance of retrieved document to user question. Document:\n\n {context}\n\nUser question: {question}\nGive a binary score 'yes' or 'no' for relevance.",
        input_variables=["context", "question"],
    )

    chain = prompt | llm_with_tool
    question = state["messages"][0].content
    docs = state["messages"][-1].content

    scored_result = chain.invoke({"question": question, "context": docs})
    return "generate" if scored_result.binary_score == "yes" else "rewrite"
```

```python
def agent(state):
    model = ChatGoogleGenerativeAI(model="gemini-1.5-pro", temperature=0)
    model = model.bind_tools(tools)
    return {"messages": [model.invoke(state["messages"])]}

def rewrite(state):
    question = state["messages"][0].content
    prompt_msg = HumanMessage(
        content=f"Improve the question based on semantic intent:\n\n{question}\n\nReformulate as an improved question."
    )
    model = ChatGoogleGenerativeAI(model="gemini-1.5-pro", temperature=0)
    return {"messages": [model.invoke([prompt_msg])]}
```

```python
def generate(state):
    docs = state["messages"][-1].content
    question = state["messages"][0].content
    # prompt = hub.pull("rlm/rag-prompt")
    prompt = PromptTemplate(
        template=(
            "Use the following document information to answer the user's question as accurately as possible. "
            "Respond with insights related to stock market updates, stocks, bitcoin, finance, interest rates, "
            "cryptocurrency, news, bullish or bearish trends, and investments.\n\n"
            "\n\nContext: {context}\n\nUser question: {question}\nAnswer:"
        ),
        input_variables=["context", "question"],
    )
    llm = ChatGoogleGenerativeAI(model="gemini-1.5-pro", temperature=0)
    rag_chain = prompt | llm | StrOutputParser()
    response = rag_chain.invoke({"context": docs, "question": question})
    return {"messages": [response]}
```

Workflow Management (Using LangGraph)

- **StateGraph Setup**:
  - **Node Definitions**: Defines the `agent`, `retrieve`, `rewrite`, and `generate` nodes.
  - **Conditional Transitions**: Directs the workflow based on whether documents retrieved are relevant.
  - **Edges**: `START`, `END`, and conditional edges enable smooth transition through each step in the workflow.

```python
workflow = StateGraph(AgentState)
workflow.add_node("agent", agent)
workflow.add_node("retrieve", ToolNode([retriever_tool]))
workflow.add_node("rewrite", rewrite)
workflow.add_node("generate", generate)
workflow.add_edge(START, "agent")
workflow.add_conditional_edges(
    "agent", tools_condition, {"tools": "retrieve", END: END}
)
workflow.add_conditional_edges("retrieve", grade_documents)
workflow.add_edge("generate", END)
workflow.add_edge("rewrite", "agent")
graph = workflow.compile()

inputs = {"messages": [("user", input_text)]}
final_output = ""
```

Environment Setup

dotenv Integration:

- **Purpose**: Loads environment variables (like API keys and URLs).
- Key environment variables include API access for Google GenAI and URLs for financial data sources.

```
""""

import os

from dotenv import load_dotenv

load_dotenv()

""""
```

Extensions & Improvements

Additional Features:

- Add more financial data sources.
- Implement error handling and retry mechanisms.

Performance Optimizations:

- Explore model tuning options for `ChatGoogleGenerativeAI`.
- Optimize document chunk sizes based on actual user input patterns.

Summary & Conclusion

Recap of Workflow: End-to-end process from document retrieval to LLM response generation.

Key Points:

- Clear modularization allows for easy scaling.
- Integrates LangGraph workflows to streamline and organize code structure

Web app


INPUT
STREAMLIT APP
AGENT
DON'T KNOW
KNOW
MONEY CONTROL/ YAHOO FINANCE/ ECONOMICC TIMES
LANGCHAN WEB BASED CODER
GOOGLE GEMINI EMBEDDINGS
CHROMA DB
RELEVANCY CHECKER (GRADE DOCUMENTS)
YES
NO
GENERATE CHAT GOOGLE GEN AI
REWRITE GOOGLE GEN AI
END

Streamlit Interface Overview

Purpose: Provides a user-friendly web interface to interact with the Stock & Finance Query Agent.

Key Components:

- Interface to load or update the knowledge base.
- Text input for user questions on stocks, finance, or cryptocurrency.
- Real-time output display for AI-generated insights.

Streamlit Code Structure

Session State Initialization:

- Uses `st.session_state` to store the state of vector database creation and ensure consistency across user actions.

Knowledge Base Loading:

- Attempts to load an existing vector database using `load_vector_db()`.
- Provides user feedback on success or prompts to create a knowledge base if loading fails.

```python
if "vector_db_created" not in st.session_state:
    st.session_state.vector_db_created = False

st.title("Stock & Finance Query Assistant")


try:
    retriever_tool, retriever, tools, vectorstore = load_vector_db()
    st.session_state.retriever_tool = retriever_tool
    st.session_state.retriever = retriever
    st.session_state.tools = tools
    st.session_state.vectorstore = vectorstore
    st.session_state.vector_db_created = True
    st.success(
        """
        knowledge base loaded succesfully
        \n
        You can now ask questions now.
        \n
        or
        \n
        you can update knowledge data base using below button 👇
        \n
        """
    )
except Exception as e:
    st.error(f"Please create a Knowledge base before you start: {e}")
```

Knowledge Base Update Button

Button Interaction:

- "Update Knowledge Base" button triggers `build_vector_db()`.
- On successful creation, stores the updated vector database in `st.session_state`.
- Provides success/failure messages to inform the user.

```
btn = st.button("Update Knowledge Base")
if btn:
    try:
        retriever_tool, retriever, tools, vectorstore = build_vector_db()
        st.session_state.retriever_tool = retriever_tool
        st.session_state.retriever = retriever
        st.session_state.tools = tools
        st.session_state.vectorstore = vectorstore
        st.session_state.vector_db_created = True
        st.success("Knowledge Base created successfully! You can now ask questions.")
    except Exception as e:
        st.error(f"Error creating knowledge base: {e}")
```

Question Input & Answer Display

User Query Input:

- Text input field for the user to ask questions.
- **Submit Button**: Processes the input only if the knowledge base is loaded and displays the result.

Result Display:

- Shows AI-generated answers directly under the input field.
- Catches empty input to prevent processing without a question.

```python
if st.session_state.vector_db_created:
    user_input = st.text_input(
        "Enter your question about stocks, finance, or cryptocurrency:",
        key="question_input",
    )
    if st.button("Submit"):
        if user_input.strip():
            retriever_tool = st.session_state.retriever_tool
            retriever = st.session_state.retriever
            tools = st.session_state.tools
            vectorstore = st.session_state.vectorstore

            result = process_user_input(
                user_input, retriever_tool, retriever, tools, vectorstore
            )
            st.subheader("Answer:")
            st.write(result)
        else:
            st.warning("Please enter a question before submitting.")
else:
    st.write(
        "Please create the knowledge base first by clicking 'Create Knowledge Base'."
    )
```

Error Handling & User Feedback

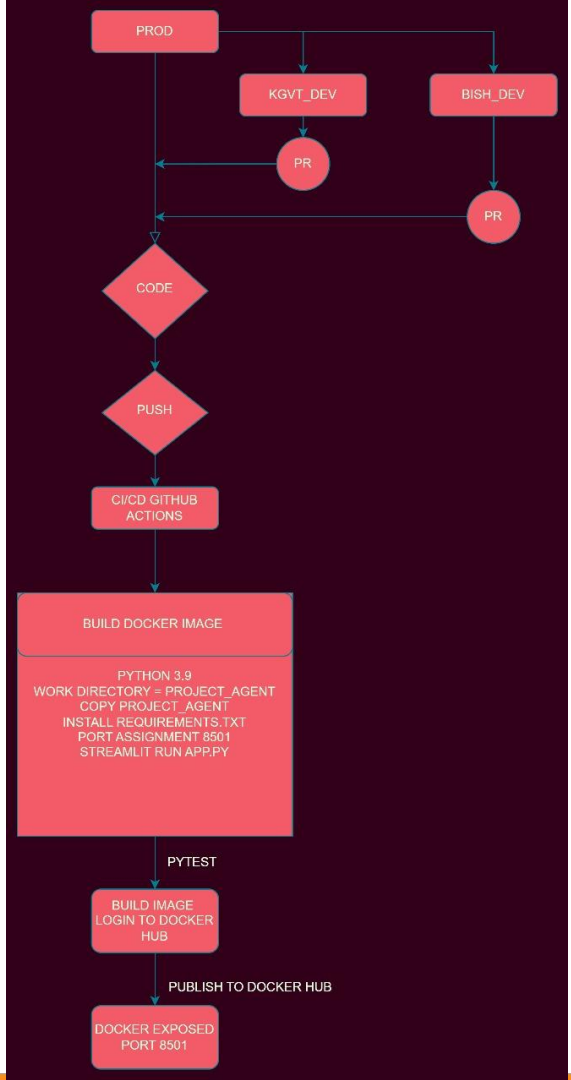
Error Messages:

- Guides users if the knowledge base is not created or if no input is detected.
- Displays success/failure messages after actions like database loading and updating.

Benefits:

- Clear guidance for users unfamiliar with database creation and ensures smoother user interaction.

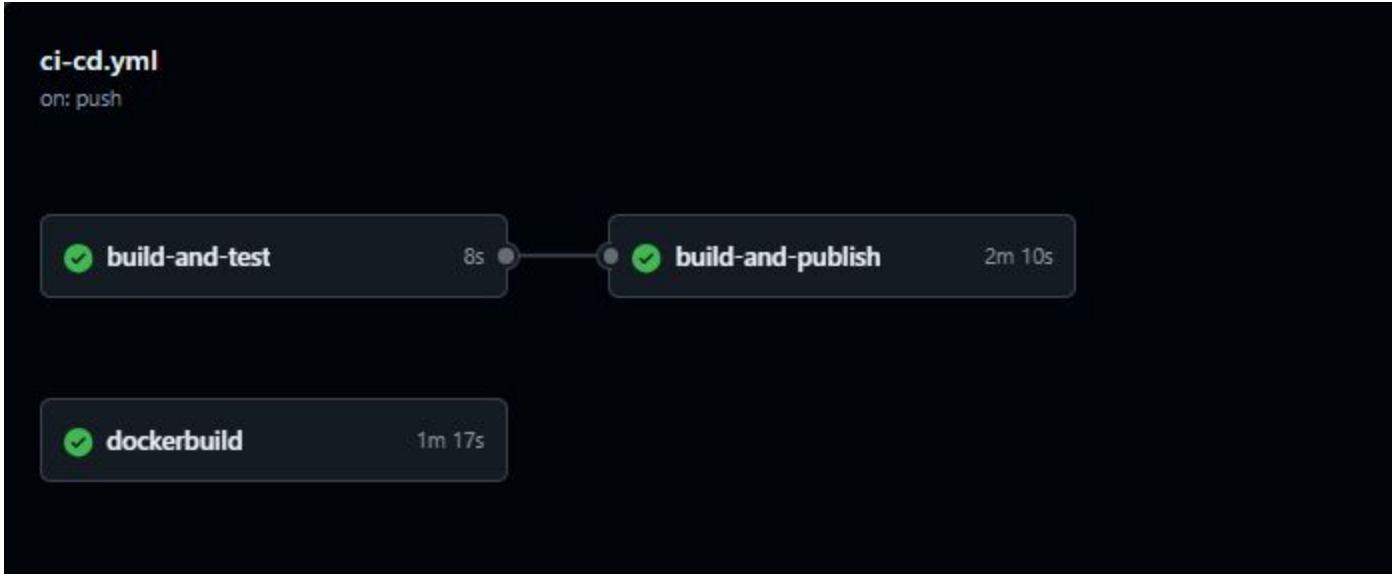
Mlops CI-CD


PROD
KGVT_DEV
BISH_DEV
PR
PR
CODE
PUSH
CI/CD GITHUB ACTIONS
BUILD DOCKER IMAGE
PYTHON 3.9
WORK DIRECTORY = PROJECT_AGENT
COPY PROJECT_AGENT
INSTALL REQUIREMENTS.TXT
PORT ASSIGNMENT 8501
STREAMLIT RUN APP.PY
PYTEST
BUILD IMAGE LOGIN TO DOCKER HUB
PUBLISH TO DOCKER HUB
DOCKER EXPOSED PORT 8501

Overview of GitHub Actions CI/CD Workflow

Purpose: Automates the build, test, and deployment steps for the Dockerized application associated with the Streamlit app and vector database.
Key Components:

- **Trigger Conditions**: Runs on pushes or pull requests to the `prod`, `bish_dev`, and `kgvt_dev` branches.
- **Job Stages**:
  - **Docker Build**: Creates the Docker image.
  - **Build and Test**: Tests Python code to ensure stability.
  - **Build and Publish**: Publishes the Docker image to DockerHub on successful test completion.


ci-cd.yml
on: push
build-and-test
8s
build-and-publish
2m 10s
dockerbuild
1m 17s

Workflow Structure: Jobs and Triggers

Docker Build Job:

- Runs on `ubuntu-latest`.
- Pulls the latest code and builds a Docker image with a timestamp tag.

Build and Test Job:

- Sets up Python 3.9 environment, installs dependencies (`pytest`, `python-dotenv`), and runs tests using `pytest` to ensure functionality.

```yaml
name: CI/CD for Dockerized App

on:
  push:
    branches: [ prod , bish_dev , kgvt_dev]
  pull_request:
    branches: [ prod ]

jobs:
  dockerbuild:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4
    - name: Build The Docker Image
      run: docker build . --file DockerFile --tag workflow-test:$(date +%s)
  build-and-test:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: '3.9'

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install pytest
        pip install python-dotenv

    - name: Run tests
      run: |
        pytest
```

prod

project_agent / test_app.py

Goutham-TA minor fix

Code Blame 14 lines (10 loc) · 243 Bytes Code 55% faster with GitHub Copilot

```python
import os
from dotenv import load_dotenv


def test_app():
    def _set_env():
        load_dotenv()

    _set_env()

    assert (
        os.environ["moneycontrol"]
        == "https://www.moneycontrol.com/stocks/marketstats/index.php"
    )
```

Docker Image Publish Job

Build and Publish Job:

- **Pre-requisite**: Dependent on successful test completion (`needs: build-and-test`).
- **DockerHub Login**: Uses DockerHub credentials from GitHub Secrets for secure login.
- **Image Build & Push**: Builds and pushes the Docker image tagged `latest` to DockerHub.
- **Image Digest**: Outputs the digest for traceability.

```yaml
  build-and-publish:
    needs: build-and-test
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v2

    - name: Login to DockerHub
      uses: docker/login-action@v2
      with:
        username: ${{ secrets.DOCKER_USERNAME }}
        password: ${{ secrets.DOCKER_PASSWORD }}

    - name: Build and push Docker image
      uses: docker/build-push-action@v4
      with:
        context: .
        file: ./DockerFile
        push: true
        tags: ${{ secrets.DOCKER_USERNAME }}/flasktest-app:latest

    - name: Image digest
      run: echo ${{ steps.build-and-publish.outputs.digest }}
```

Code and automation

- Branches
- Tags
- Rules
- Actions
- Webhooks
- Environments
- Codespaces
- Pages

Security

- Code security
- Deploy keys
- Secrets and variables
  - Actions
  - Codespaces
  - Dependabot

workflows that are triggered by a pull request from a fork.

Secrets | Variables

Environment secrets

This environment has no secrets.

Manage environment secrets

Repository secrets

New repository secret

Name	Last updated
DOCKER_PASSWORD	last week
DOCKER_USERNAME	last week

Integrating CI/CD with Streamlit App

Continuous Integration: Automates testing on every code change to maintain app stability.
Continuous Deployment: Automatically publishes Docker image, ensuring the latest version is readily available.
Benefits:

- Eliminates manual deployment steps.
- Guarantees consistency between local and deployed environments.
- Provides quick feedback on code changes via automated tests.

Complete Workflow Overview

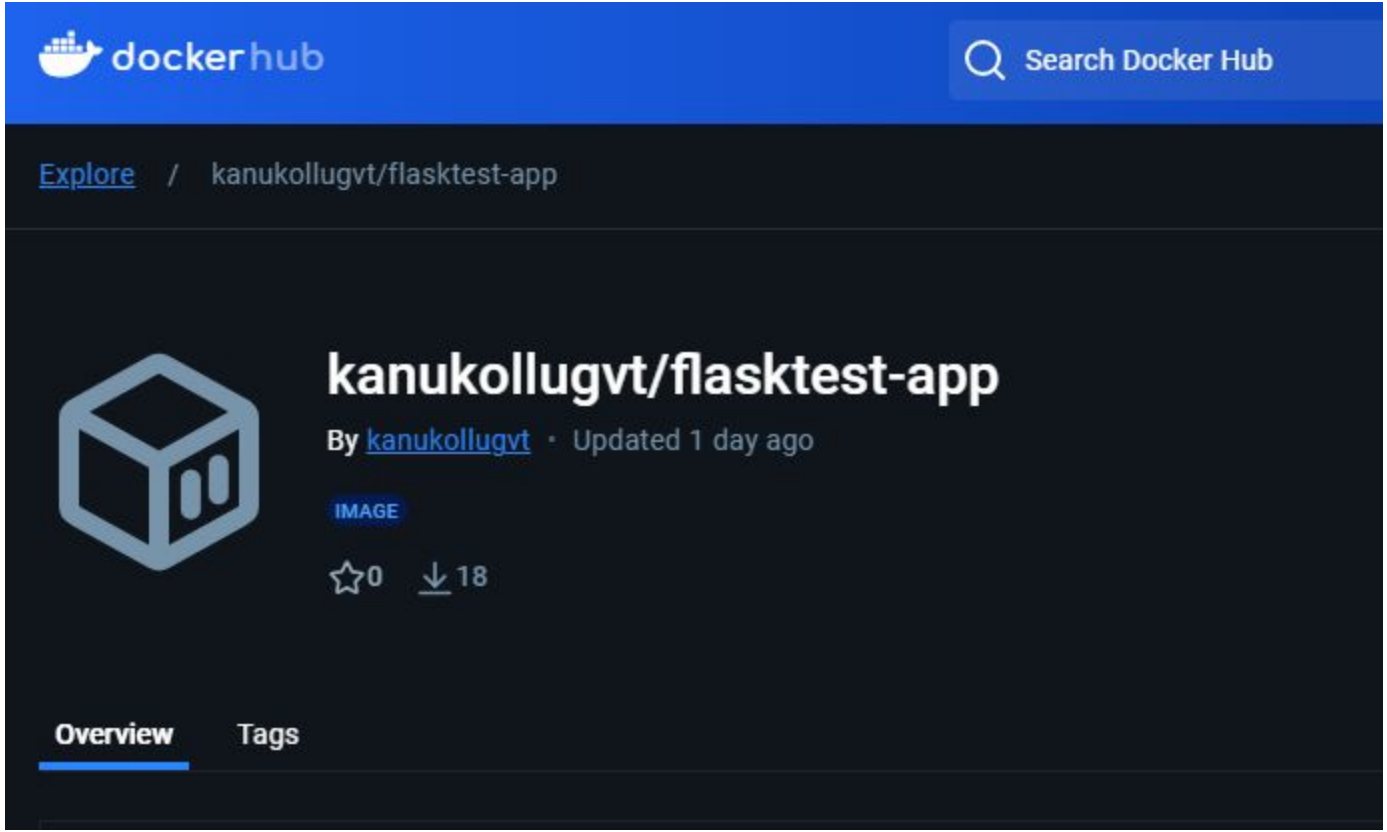
Combined Workflow:

- Vector Database Processing and Streamlit App serve as the application.
- CI/CD Pipeline ensures a stable and up-to-date Docker image is available for deployment.

Demo Flow:

- Push or Pull Request triggers -> Docker Image Build -> Code Testing -> Docker Image Deployment.

Docker


docker hub
Search Docker Hub
Explore / kanukollugvt/flasktest-app
kanukollugvt/flasktest-app
By kanukollugvt • Updated 1 day ago
IMAGE
0
18
Overview
Tags

Dockerfile for Streamlit Application

Purpose: Defines the environment and dependencies for the Dockerized Streamlit app.

Key Points:

- Uses an official, minimal Python image.
- Installs dependencies and configures the container to run the app on startup.

Dockerfile Structure

Base Image:

- `FROM python:3.9-slim`: Uses the official Python 3.9 slim image for efficiency.

Working Directory:

- `WORKDIR /project_agent`: Sets `/project_agent` as the working directory inside the container.

Copy Project Files:

- `COPY . /project_agent`: Copies all files from the current directory on the host to the container's working directory.

Dependency Installation & Port Exposure

Install Requirements:

- `RUN pip3 install -r requirements.txt`: Installs dependencies specified in `requirements.txt`.

Port Exposure:

- `EXPOSE 8501`: Makes port 8501 available for the Streamlit app to serve requests.

Container Startup Command

Run Command:

- `CMD ["streamlit", "run", "app.py"]`: Starts the Streamlit app when the container launches.
- **Explanation**:
  - `streamlit run app.py` command initiates the app, allowing it to respond on the exposed port 8501.

Benefits:

- This setup ensures consistency across deployments, as the same environment is recreated for each instance.

```
# Use the official Python image from the Docker Hub
FROM python:3.9-slim

# Set the working directory
WORKDIR /project_agent

# Copy the current directory contents into the container at /project_agent
COPY . /project_agent

# Install any needed packages specified in requirements.txt
RUN pip3 install -r requirements.txt

# Make port 8501 available to the world outside this container
EXPOSE 8501

# Run app.py when the container launches
CMD ["streamlit", "run", "app.py"]
```

End-to-End Dockerized Workflow

From Development to Deployment:

- The Dockerfile creates a standardized environment for running the Streamlit app.
- GitHub Actions CI/CD workflow builds, tests, and deploys this Docker image.

Integrated Workflow:

- Code -> Docker Image -> CI/CD Workflow -> Streamlit App Deployment

Advantages:

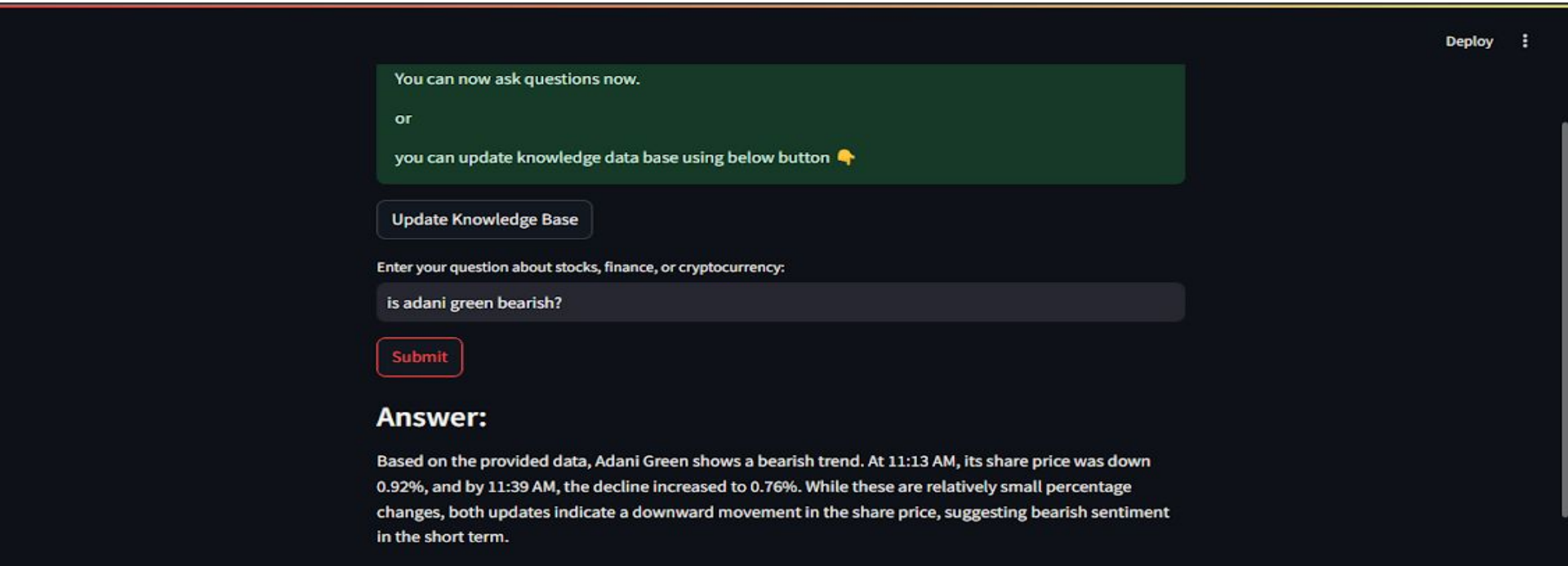
- Streamlined deployment, environment consistency, and reduced setup overhead.

```
(.venv) (base) bishweashwarsukla@BSUKLA-OMEN:~/project/project_agent$ docker run -p 8501:8501 kanukollugvt/flasktest-app

Collecting usage statistics. To deactivate, set browser.gatherUsageStats to false.


  You can now view your Streamlit app in your browser.

  Local URL: http://localhost:8501
  Network URL: http://172.18.0.2:8501
  External URL: http://122.161.72.254:8501

USER_AGENT environment variable not set, consider setting it to identify your requests.
```

```
(.venv) (base) bishweashwarsukla@BSUKLA-OMEN:~$ cd project/
(.venv) (base) bishweashwarsukla@BSUKLA-OMEN:~/project$ cd project_agent/
(.venv) (base) bishweashwarsukla@BSUKLA-OMEN:~/project/project_agent$ docker ps
CONTAINER ID   IMAGE                       COMMAND                  CREATED         STATUS         PORTS
    NAMES
e82d6c7b8cf6   kanukollugvt/flasktest-app   "streamlit run app.py"   3 minutes ago   Up 3 minutes   0.0.0.0:8501->8501/tcp, :::8501->8501/tc
p   strange_chandrasekhar
(.venv) (base) bishweashwarsukla@BSUKLA-OMEN:~/project/project_agent$
```

How to run

- Pull docker image using **docker pull kanukollugvt/flasktest-app**
- Run command **streamlit run app.py** (if running in local)
- Run command **docker run -p 8501:8501 kanukollugvt/flasktest-app**



Tiger Analytics

Thank you

www.tigeranalytics.com

Kanukollu Goutham (804130) AND Bishweshwar Sukla(804133)