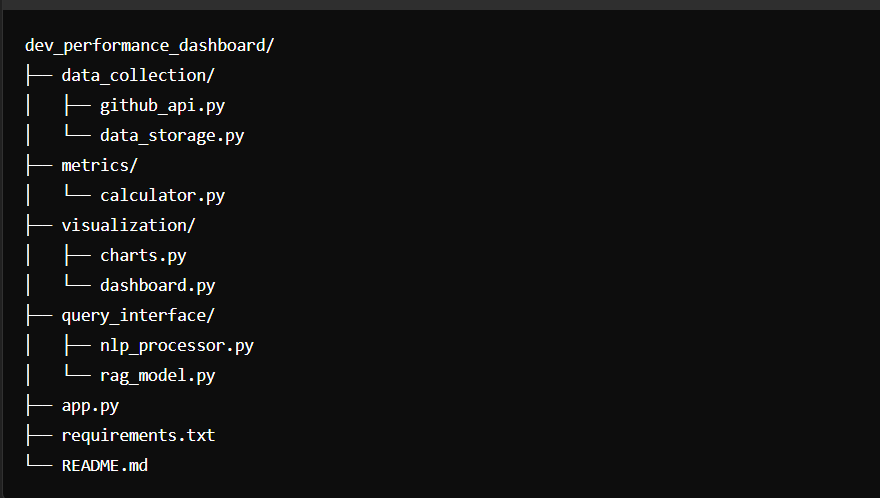
**AI-Powered Developer Performance Analytics Dashboard**

**Project Structure**



**data\_collection/ github\_api.py**

from github import Github

from datetime import datetime

import pytz

import pandas as pd

class GitHubDataCollector:

    def \_\_init\_\_(self, token):

        self.github = Github(token)

    def \_extract\_repo\_name(self, repo\_url):

        # Extract repo name from URL (assuming format https://github.com/owner/repo)

        parts = repo\_url.rstrip('/').split('/')

        return f"{parts[-2]}/{parts[-1]}"

    def \_convert\_to\_utc(self, dt):

        """Convert a timezone-aware datetime to UTC."""

        if dt.tzinfo is not None:

            return dt.astimezone(pytz.utc)

        return dt

    def get\_forks\_data(self, repo\_url):

        repo\_name = self.\_extract\_repo\_name(repo\_url)

        repo = self.github.get\_repo(repo\_name)

        forks = repo.get\_forks()

        forks\_data = []

        for fork in forks:

            forks\_data.append({

                "username": fork.owner.login,

                "date": fork.created\_at,

                "profile\_image": fork.owner.avatar\_url if fork.owner.avatar\_url else None

            })

        return forks\_data

    def get\_repo\_data(self, repo\_url):

        repo\_name = self.\_extract\_repo\_name(repo\_url)

        repo = self.github.get\_repo(repo\_name)

        return {

            "name": repo.name,

            "full\_name": repo.full\_name,

            "description": repo.description,

            "language": repo.language,

            "created\_at": self.\_convert\_to\_utc(repo.created\_at).isoformat(),

            "updated\_at": self.\_convert\_to\_utc(repo.updated\_at).isoformat(),

            "stargazers\_count": repo.stargazers\_count,

            "forks\_count": repo.forks\_count,

            "open\_issues\_count": repo.open\_issues\_count

        }

    def get\_commits\_data(self, repo\_url):

        repo\_name = self.\_extract\_repo\_name(repo\_url)

        repo = self.github.get\_repo(repo\_name)

        commits = repo.get\_commits()

        commits\_data = []

        for commit in commits:

            commits\_data.append({

                "sha": commit.sha,

                "author": commit.commit.author.name,

                "date": self.\_convert\_to\_utc(commit.commit.author.date).isoformat(),

                "message": commit.commit.message

            })

        return commits\_data

    def get\_issues\_data(self, repo\_url):

        repo\_name = self.\_extract\_repo\_name(repo\_url)

        repo = self.github.get\_repo(repo\_name)

        issues = repo.get\_issues(state='all')

        issues\_data = []

        for issue in issues:

            issues\_data.append({

                "id": issue.id,

                "title": issue.title,

                "state": issue.state,

                "created\_at": self.\_convert\_to\_utc(issue.created\_at).isoformat(),

                "closed\_at": self.\_convert\_to\_utc(issue.closed\_at).isoformat() if issue.closed\_at else "Not Closed"

            })

        return issues\_data

    def get\_pull\_requests\_data(self, repo\_url):

        repo\_name = self.\_extract\_repo\_name(repo\_url)

        repo = self.github.get\_repo(repo\_name)

        pull\_requests\_data = []

        for pr in repo.get\_pulls(state='all'):

            pull\_requests\_data.append({

                "id": pr.id,

                "title": pr.title,

                "created\_at": pr.created\_at.isoformat(),

                "merged\_at": pr.merged\_at.isoformat() if pr.merged\_at else "Not Merged",

                "user": pr.user.login

            })

        return pull\_requests\_data

    def get\_code\_reviews\_data(self, repo\_url):

        repo\_name = self.\_extract\_repo\_name(repo\_url)

        repo = self.github.get\_repo(repo\_name)

        pull\_requests = repo.get\_pulls(state='all')

        reviews\_data = []

        for pr in pull\_requests:

            reviews = pr.get\_reviews()

            for review in reviews:

                reviews\_data.append({

                    "pr\_id": pr.id,

                    "reviewer": review.user.login,

                    "submitted\_at": self.\_convert\_to\_utc(review.submitted\_at).isoformat(),

                    "body": review.body

                })

        return reviews\_data

    # Fetch PR data

    def fetch\_pr\_data(self, repo\_url):

        repo\_name = self.\_extract\_repo\_name(repo\_url)

        repo = self.github.get\_repo(repo\_name)

        pulls = repo.get\_pulls(state="all")

        pr\_data = []

        for pr in pulls:

            pr\_data.append({

                'number': pr.number,

                'state': pr.state,

                'merged': pr.merged,

                'created\_at': pr.created\_at,

                'closed\_at': pr.closed\_at if pr.closed\_at else pd.NaT,  # Handle missing closed\_at

                'merged\_at': pr.merged\_at if pr.merged\_at else pd.NaT  # Handle missing merged\_at

            })

        return pd.DataFrame(pr\_data)

    # Fetch issue data

    def fetch\_issue\_data(self, repo\_url):

        repo\_name = self.\_extract\_repo\_name(repo\_url)

        repo = self.github.get\_repo(repo\_name)

        issues = repo.get\_issues(state="closed")

        issue\_data = []

        for issue in issues:

            if not issue.pull\_request:  # Exclude PRs labeled as issues

                issue\_data.append({

                    'number': issue.number,

                    'created\_at': issue.created\_at,

                    'closed\_at': issue.closed\_at,

                    'resolution\_time': (issue.closed\_at - issue.created\_at).total\_seconds() / 3600 if issue.closed\_at else pd.NA  # Handle missing closed\_at

                })

        return pd.DataFrame(issue\_data)

**#data\_Storage.py**

#data\_storage.py

import os

import pandas as pd

class DataStorage:

    def \_\_init\_\_(self, storage\_dir="data"):

        self.storage\_dir = storage\_dir

        if not os.path.exists(self.storage\_dir):

            os.makedirs(self.storage\_dir)

    def save\_data\_to\_csv(self, data, filename):

        """Save data to a CSV file."""

        file\_path = os.path.join(self.storage\_dir, filename)

        # Convert data to DataFrame and save as CSV

        if isinstance(data, list):

            df = pd.DataFrame(data)

        else:

            df = pd.DataFrame([data])  # Convert single dictionary to DataFrame

        df.to\_csv(file\_path, index=False)

        print(f"Data saved to {file\_path}")

        return file\_path

    def load\_data\_from\_csv(self, filename):

        """Load data from a CSV file."""

        file\_path = os.path.join(self.storage\_dir, filename)

        if os.path.exists(file\_path):

            df = pd.read\_csv(file\_path)

            return df.to\_dict(orient="records")

        else:

            raise FileNotFoundError(f"{filename} does not exist")

**metrics/ calculator.py**

import pandas as pd

class MetricsCalculator:

    def \_\_init\_\_(self, data):

        self.data = data

        self.commits\_data = pd.DataFrame(data.get('commits', []))

        self.issues\_data = pd.DataFrame(data.get('issues', []))

        self.repo\_data = data.get('repo', {})

    def check\_repo\_data\_validity(self):

        """Check if repository has sufficient data (stars, forks, open issues, commits)."""

        stars\_count = self.repo\_data.get('stargazers\_count', 0)

        forks\_count = self.repo\_data.get('forks\_count', 0)

        open\_issues\_count = self.repo\_data.get('open\_issues\_count', 0)

        commit\_count = len(self.commits\_data)

        if stars\_count == 0 and forks\_count == 0 and open\_issues\_count == 0 and commit\_count == 0:

            return "Need more information. The repository has no stars, forks, open issues, or commits."

        return None

    def calculate\_commit\_frequency(self):

        """Calculate commit frequency by month."""

        try:

            validity\_message = self.check\_repo\_data\_validity()

            if validity\_message:

                return validity\_message

            if self.commits\_data.empty:

                return "No commit data available."

            self.commits\_data['date'] = pd.to\_datetime(self.commits\_data['date'], utc=True)

            commit\_frequency = self.commits\_data.groupby(self.commits\_data['date'].dt.to\_period('M')).size().reset\_index(name='count')

            commit\_frequency['date'] = commit\_frequency['date'].dt.to\_timestamp()  # Convert to timestamp for Plotly

            return commit\_frequency

        except Exception as e:

            print(f"Error calculating commit frequency: {e}")

            return pd.DataFrame()

    def calculate\_issue\_resolution\_time(self):

        """Calculate average issue resolution time in days."""

        try:

            validity\_message = self.check\_repo\_data\_validity()

            if validity\_message:

                return validity\_message

            if self.issues\_data.empty:

                return "No issue data available."

            self.issues\_data['created\_at'] = pd.to\_datetime(self.issues\_data['created\_at'])

            self.issues\_data['closed\_at'] = pd.to\_datetime(self.issues\_data['closed\_at'], errors='coerce')

            self.issues\_data['resolution\_time'] = (self.issues\_data['closed\_at'] - self.issues\_data['created\_at']).dt.days

            resolution\_time = self.issues\_data['resolution\_time'].dropna().mean()

            return resolution\_time

        except Exception as e:

            print(f"Error calculating issue resolution time: {e}")

            return float('nan')

    def calculate\_issue\_counts\_by\_month(self):

        """Calculate issue counts and resolved/unresolved issues by month."""

        try:

            validity\_message = self.check\_repo\_data\_validity()

            if validity\_message:

                return validity\_message

            if self.issues\_data.empty:

                return "No issue data available."

            self.issues\_data['created\_at'] = pd.to\_datetime(self.issues\_data['created\_at'])

            issue\_counts = self.issues\_data.groupby(self.issues\_data['created\_at'].dt.to\_period('M')).size().reset\_index(name='count')

            issue\_counts['resolved\_issues'] = self.issues\_data.groupby(self.issues\_data['created\_at'].dt.to\_period('M'))['closed\_at'].count().reset\_index(name='resolved\_issues')['resolved\_issues']

            issue\_counts['unresolved\_issues'] = issue\_counts['count'] - issue\_counts['resolved\_issues']

            issue\_counts['date'] = issue\_counts['created\_at'].dt.to\_timestamp()

            return issue\_counts

        except Exception as e:

            print(f"Error calculating issue counts by month: {e}")

            return pd.DataFrame()

    def calculate\_issue\_pie\_chart\_data(self):

        """Calculate data for pie chart showing resolved vs unresolved issues."""

        try:

            validity\_message = self.check\_repo\_data\_validity()

            if validity\_message:

                return validity\_message

            if self.issues\_data.empty:

                return "No issue data available."

            total\_issues = len(self.issues\_data)

            unresolved\_issues = self.issues\_data['closed\_at'].isna().sum()

            resolved\_issues = total\_issues - unresolved\_issues

            return pd.DataFrame({

                'Issue Status': ['Resolved', 'Unresolved'],

                'Count': [resolved\_issues, unresolved\_issues]

            })

        except Exception as e:

            print(f"Error calculating pie chart data: {e}")

            return pd.DataFrame()

    def calculate\_pr\_merge\_rate(self, pull\_requests\_data):

        """Calculate the average time to merge pull requests."""

        try:

            validity\_message = self.check\_repo\_data\_validity()

            if validity\_message:

                return validity\_message

            pr\_df = pd.DataFrame(pull\_requests\_data)

            if pr\_df.empty:

                return "No pull request data available."

            pr\_df['created\_at'] = pd.to\_datetime(pr\_df['created\_at'], utc=True)

            pr\_df['merged\_at'] = pd.to\_datetime(pr\_df['merged\_at'], errors='coerce', utc=True)

            pr\_df['time\_to\_merge'] = (pr\_df['merged\_at'] - pr\_df['created\_at']).dt.days

            merge\_rate = pr\_df['time\_to\_merge'].dropna().mean()

            return merge\_rate

        except Exception as e:

            print(f"Error calculating PR merge rate: {e}")

            return float('nan')

    def calculate\_code\_review\_metrics(self, reviews\_data):

        """Calculate average number of comments per pull request."""

        try:

            validity\_message = self.check\_repo\_data\_validity()

            if validity\_message:

                return validity\_message

            reviews\_df = pd.DataFrame(reviews\_data)

            if 'pr\_id' not in reviews\_df.columns:

                raise ValueError("Missing 'pr\_id' column in reviews data")

            if reviews\_df.empty:

                return "No code review data available."

            comments\_per\_pr = reviews\_df.groupby('pr\_id').size()

            avg\_comments\_per\_pr = comments\_per\_pr.mean()

            return avg\_comments\_per\_pr

        except Exception as e:

            print(f"Error calculating code review metrics: {e}")

            return float('nan')

**queryinterface/nlp\_processor.py**

import re

import ollama

class NLPProcessor:

    def \_\_init\_\_(self):

        # Define query patterns for common metrics

        self.query\_patterns = {

            'commit\_frequency': re.compile(r'\bcommit frequency\b', re.IGNORECASE),

            'issue\_resolution': re.compile(r'\bissue resolution time\b', re.IGNORECASE),

            'pr\_merge\_rate': re.compile(r'\bpull request merge rate\b', re.IGNORECASE),

            'code\_review\_metrics': re.compile(r'\bcode review metrics\b', re.IGNORECASE)

        }

    def process\_query(self, query):

        """

        Process the user's natural language query and return the appropriate result.

        If the query matches predefined patterns, return the corresponding metric type.

        Otherwise, send the query to the Ollama LLM model for further processing.

        """

        query = query.strip().lower()

        # Check if the query matches any predefined patterns

        for key, pattern in self.query\_patterns.items():

            if pattern.search(query):

                return key

        # If no predefined patterns match, use Ollama LLM for query processing

        try:

            desired\_model = 'llama3.1:8b'

            # Send query to Ollama model

            response = ollama.chat(model=desired\_model, messages=[

                {

                    'role': 'user',

                    'content': query,

                },

            ])

            # Extract the response message from Ollama's output

            llm\_response = response['message']['content']

            return llm\_response

        except Exception as e:

            # Handle exceptions related to the LLM model

            return f"Error processing query: {str(e)}"

**queryinterface/rag\_model.py**

import json

from sentence\_transformers import SentenceTransformer

from sklearn.metrics.pairwise import cosine\_similarity

from ollama import Ollama

# Step 1: Load the JSON dataset

def load\_json\_data(file\_path):

    with open(file\_path, "r") as f:

        return json.load(f)

# Step 2: Initialize the embedding model for document retrieval

def initialize\_embedding\_model():

    return SentenceTransformer("thenlper/gte-large")

# Step 3: Compute embeddings for all documents in the dataset

def compute\_embeddings(json\_data, embed\_model):

    return [embed\_model.encode(str(item)) for item in json\_data]

# Step 4: Retrieve the most relevant document based on a query

def retrieve\_relevant\_docs(query, json\_data, embeddings, embed\_model):

    query\_embedding = embed\_model.encode(query)

    similarities = cosine\_similarity([query\_embedding], embeddings)

    best\_match\_idx = similarities.argmax()

    return json\_data[best\_match\_idx]

# Step 5: Query the LLM (Mistral-7B via Ollama) with the retrieved document and user's question

def generate\_answer\_with\_ollama(relevant\_doc, query, model\_name="ollama-3b"):

    # Initialize Ollama LLM

    llm = Ollama(model=model\_name)

    # Prepare context and question for the LLM

    context = f"Document: {relevant\_doc}"

    prompt = f"{context}\n\nQuestion: {query}"

    # Query the LLM and return the response

    response = llm.query(prompt)

    return response

**Virtualization/charts.py**

import plotly.express as px

import plotly.graph\_objects as go

import pandas as pd

class ChartBuilder:

    def plot\_commit\_frequency(self, commit\_frequency\_df):

        """Plot commit frequency using Plotly."""

        if 'count' not in commit\_frequency\_df.columns:

            raise ValueError("DataFrame must contain a 'count' column for commit frequency.")

        fig = px.line(commit\_frequency\_df, x='date', y='count', title='Commit Frequency Over Time')

        fig.update\_layout(xaxis\_title='Date', yaxis\_title='Number of Commits')

        return fig

    def plot\_issue\_resolution(self, resolution\_time):

        """Plot issue resolution time using Plotly."""

        fig = px.bar(x=["Issue Resolution Time"], y=[resolution\_time], labels={"x": "Metric", "y": "Days"})

        fig.update\_layout(title="Average Issue Resolution Time", yaxis\_title="Days")

        return fig

    def plot\_fork\_count\_by\_month(self, forks\_monthly\_count\_df):

        """Plot fork count by month and year with styled lines and markers using Plotly."""

        if 'count' not in forks\_monthly\_count\_df.columns or 'month\_year' not in forks\_monthly\_count\_df.columns:

            raise ValueError("DataFrame must contain 'month\_year' and 'count' columns for fork count.")

        # Plot line chart with markers

        fig = px.line(forks\_monthly\_count\_df, x='month\_year', y='count',

                  title='Fork Count by Month and Year',

                  labels={'month\_year': 'Month-Year', 'count': 'Fork Count'},

                  markers=True)

        # Styling the chart

        fig.update\_traces(line=dict(color='royalblue', width=3),  # Line color and thickness

                      marker=dict(size=10, symbol='circle', color='darkorange'),  # Marker style

                      mode='lines+markers')  # Display both lines and markers

        # Customize layout

        fig.update\_layout(

        xaxis\_title='Month-Year',

        yaxis\_title='Fork Count',

        xaxis=dict(showgrid=False, tickangle=45),  # Remove grid lines from X-axis and angle ticks

        yaxis=dict(showgrid=True, gridwidth=1, gridcolor='lightgray'),  # Customize Y-axis grid

        plot\_bgcolor='white',  # Set plot background color

        title\_font=dict(size=24, family='Arial', color='darkblue'),  # Title font style

        xaxis\_tickfont=dict(size=12, family='Arial', color='black'),  # X-axis ticks font style

        yaxis\_tickfont=dict(size=12, family='Arial', color='black')   # Y-axis ticks font style

    )

        return fig

    def plot\_issue\_count\_by\_month(self, issue\_counts\_df):

        """Plot stacked bar chart of issue counts and resolved issues by month and year."""

        try:

            if 'count' not in issue\_counts\_df.columns or 'resolved\_issues' not in issue\_counts\_df.columns:

                raise ValueError("DataFrame must contain 'count' and 'resolved\_issues' columns.")

            # Melt DataFrame for stacked bar plot

            melted\_df = issue\_counts\_df.melt(id\_vars='date', value\_vars=['resolved\_issues', 'unresolved\_issues'],

                                             var\_name='issue\_type', value\_name='issue\_count')

            fig = px.bar(melted\_df, x='date', y='issue\_count', color='issue\_type',

                         title='Issue Count and Resolution by Month and Year',

                         labels={'date': 'Month-Year', 'issue\_count': 'Issue Count', 'issue\_type': 'Issue Type'},

                         text='issue\_count')

            fig.update\_layout(xaxis\_title='Month-Year', yaxis\_title='Issue Count')

            return fig

        except Exception as e:

            print(f"Error plotting issue count by month: {e}")

            return None

    def plot\_issue\_pie\_chart(self, pie\_chart\_data\_df):

        """Plot pie chart of resolved vs. unresolved issues using Plotly."""

        if 'Issue Status' not in pie\_chart\_data\_df.columns or 'Count' not in pie\_chart\_data\_df.columns:

            raise ValueError("DataFrame must contain 'Issue Status' and 'Count' columns for pie chart.")

        fig = px.pie(pie\_chart\_data\_df, names='Issue Status', values='Count',

                     title='Issue Status Overview')

        fig.update\_layout(legend\_title='Issue Status')

        return fig

    def plot\_pr\_merge\_rate(self, merge\_rate):

        """Plot pull request merge rate."""

        fig = px.bar(x=["PR Merge Rate"], y=[merge\_rate], labels={"x": "Metric", "y": "Days"})

        fig.update\_layout(title="Average Pull Request Merge Rate", yaxis\_title="Days")

        return fig

    def plot\_code\_review\_metrics(self, avg\_comments\_per\_pr):

        """Plot average number of comments per pull request."""

        fig = px.bar(x=["Average Comments per PR"], y=[avg\_comments\_per\_pr], labels={"x": "Metric", "y": "Comments"})

        fig.update\_layout(title="Average Code Review Comments per Pull Request", yaxis\_title="Comments")

        return fig

    def visualize\_metrics(self, pr\_df, issue\_df, period='M'):

        """Visualize PR merge rates and issue resolution times on a dual-axis chart."""

        # Process PR data

        pr\_df['created\_at'] = pd.to\_datetime(pr\_df['created\_at'])

        pr\_df.set\_index('created\_at', inplace=True)

        # Calculate PR merge rate for each period

        pr\_periodic = pr\_df.resample(period).apply(lambda df: pd.Series({'merge\_rate': calculate\_merge\_rate(df)}))

        # Process issue data

        issue\_df['created\_at'] = pd.to\_datetime(issue\_df['created\_at'])

        issue\_df.set\_index('created\_at', inplace=True)

        issue\_periodic = issue\_df.resample(period)['resolution\_time'].mean()  # Average resolution time

        # Create a dual-axis plot

        fig = go.Figure()

        # PR Merge Rate (left axis)

        fig.add\_trace(go.Scatter(

            x=pr\_periodic.index, y=pr\_periodic['merge\_rate'],

            mode='lines+markers',  # Line + marker

            name='PR Merge Rate',

            line=dict(color='blue', dash='solid'),  # Solid blue line

            marker=dict(symbol='circle', color='blue'),  # Circle markers

            yaxis='y1'

        ))

        # Issue Resolution Time (right axis)

        fig.add\_trace(go.Scatter(

            x=issue\_periodic.index, y=issue\_periodic,

            mode='lines+markers',  # Line + marker

            name='Issue Resolution Time (hrs)',

            line=dict(color='red', dash='dash'),  # Dashed red line

            marker=dict(symbol='x', color='red'),  # X markers

            yaxis='y2'

        ))

        # Update layout for dual-axis

        fig.update\_layout(

            title='PR Merge Rate and Issue Resolution Time',

            xaxis=dict(title='Date'),

            yaxis=dict(title='PR Merge Rate (%)', titlefont=dict(color='blue'), tickfont=dict(color='blue')),

            yaxis2=dict(title='Issue Resolution Time (hours)', titlefont=dict(color='red'), tickfont=dict(color='red'),

                        overlaying='y', side='right'),

            legend=dict(x=0.1, y=1.1)

        )

        return fig

def calculate\_merge\_rate(pr\_df):

    """Calculate the PR merge rate."""

    merged\_count = pr\_df['merged\_at'].notna().sum()

    total\_count = len(pr\_df)

    return (merged\_count / total\_count) \* 100 if total\_count > 0 else 0

**virtualiztion/dashboard.py**

import streamlit as st

import pandas as pd

from PIL import Image

import requests

from io import BytesIO

from metrics.calculator import MetricsCalculator

@st.cache\_data

def load\_data(repo\_data, commits\_data, issues\_data, forks\_data, pull\_requests\_data, reviews\_data):

    # Assuming `MetricsCalculator` and `ChartBuilder` are properly implemented

    raw\_data = {

        "commits": commits\_data,

        "issues": issues\_data

    }

    metrics\_calculator = MetricsCalculator(raw\_data)

    commit\_frequency = metrics\_calculator.calculate\_commit\_frequency()

    issue\_resolution = metrics\_calculator.calculate\_issue\_resolution\_time()

    issue\_counts\_by\_month = metrics\_calculator.calculate\_issue\_counts\_by\_month()

    issue\_pie\_chart\_data = metrics\_calculator.calculate\_issue\_pie\_chart\_data()

    pr\_df = pd.DataFrame(pull\_requests\_data)

    issue\_df = pd.DataFrame(issues\_data)

    pr\_merge\_rate = metrics\_calculator.calculate\_pr\_merge\_rate(pr\_df)

    avg\_comments\_per\_pr = metrics\_calculator.calculate\_code\_review\_metrics(reviews\_data)

    avg\_stars = repo\_data['stargazers\_count']

    avg\_star\_rating = min(avg\_stars / 50, 5)

    return {

        "commit\_frequency": commit\_frequency,

        "issue\_resolution": issue\_resolution,

        "issue\_counts\_by\_month": issue\_counts\_by\_month,

        "issue\_pie\_chart\_data": issue\_pie\_chart\_data,

        "pr\_df": pr\_df,

        "issue\_df": issue\_df,

        "pr\_merge\_rate": pr\_merge\_rate,

        "avg\_comments\_per\_pr": avg\_comments\_per\_pr,

        "avg\_star\_rating": avg\_star\_rating

    }

def display\_summary(

    repo\_data, user, avg\_star\_rating, forks\_data, commit\_frequency,

    issue\_resolution, issue\_counts\_by\_month, issue\_pie\_chart\_data,

    pr\_df, issue\_df, pr\_merge\_rate, avg\_comments\_per\_pr, chart\_builder, nlp\_processor

):

   # Process all the data first

    # Summary Report content

    summary\_report\_html = f"""

    <div style="position:relative; border:1px solid #ccc; padding:16px; border-radius:8px;">

        <div style="position:absolute; top:16px; right:16px;">

            <img src="{user.avatar\_url if user.avatar\_url else 'https://via.placeholder.com/100'}" width="100" alt="Profile Image" style="border-radius:50%;"/>

        </div>

        <h2 style="margin:0;">User Information</h2>

        <hr style="border:1px solid #ddd;">

        <ul style="list-style-type:none; padding:0;">

            <li><strong>Name:</strong> {user.name}</li>

            <li><strong>User ID:</strong> {user.id}</li>

            <li><strong>Bio:</strong> {user.bio if user.bio else 'No Bio'}</li>

            <li><strong>Total Repositories:</strong> {user.public\_repos}</li>

            <li><strong>Followers Count:</strong> {user.followers}</li>

            <li><strong>Following Count:</strong> {user.following}</li>

        </ul>

    </div>

    <div style="border:1px solid #ccc; padding:16px; border-radius:8px;">

        <h2 style="margin:0;">Repository Overview</h2>

        <hr style="border:1px solid #ddd;">

        <ul style="list-style-type:none; padding:0;">

            <li><strong>Repository Name:</strong> {repo\_data['name']}</li>

            <li><strong>Description:</strong> {repo\_data['description']}</li>

            <li><strong>Language:</strong> {repo\_data['language']}</li>

            <li><strong>Created At:</strong> {repo\_data['created\_at']}</li>

            <li><strong>Updated At:</strong> {repo\_data['updated\_at']}</li>

            <li><strong>Stars Count:</strong> {repo\_data['stargazers\_count']}</li>

            <li><strong>Forks Count:</strong> {repo\_data['forks\_count']}</li>

            <li><strong>Open Issues Count:</strong> {repo\_data['open\_issues\_count']}</li>

            <li><strong>Average Star Rating:</strong> {'⭐' \* int(avg\_star\_rating)}{''.join(['☆' for \_ in range(5 - int(avg\_star\_rating))])}</li>

        </ul>

    </div>

    """

    # Commit Frequency content

    commit\_frequency\_chart = None

    if (repo\_data['forks\_count'] != 0 or repo\_data['open\_issues\_count'] != 0) and len(commit\_frequency) > 0 and repo\_data['stargazers\_count'] != 0:

        commit\_frequency\_chart = chart\_builder.plot\_commit\_frequency(commit\_frequency)

    # Forks Details content

    forks\_chart = None

    forks\_table\_html = None

    if (repo\_data['forks\_count'] != 0 or repo\_data['open\_issues\_count'] != 0) and len(commit\_frequency) > 0 and repo\_data['stargazers\_count'] != 0:

        forks\_df = pd.DataFrame(forks\_data)

        forks\_df['date'] = pd.to\_datetime(forks\_df['date'])

        forks\_df['month\_year'] = forks\_df['date'].dt.strftime('%Y-%m')

        forks\_monthly\_count = forks\_df.groupby('month\_year').size().reset\_index(name='count')

        forks\_chart = chart\_builder.plot\_fork\_count\_by\_month(forks\_monthly\_count)

        forks\_df\_display = pd.DataFrame({

            'S.No': range(1, len(forks\_df) + 1),

            'Profile Image': [fork['profile\_image'] if fork['profile\_image'] else "https://via.placeholder.com/50" for fork in forks\_data],

            'Username': [fork['username'] for fork in forks\_data],

            'Date': [fork['date'].strftime('%Y-%m-%d') for fork in forks\_data]

        })

        def image\_formatter(image\_url):

            return f'<img src="{image\_url}" width="50"/>'

        forks\_df\_display['Profile Image'] = forks\_df\_display['Profile Image'].apply(image\_formatter)

        forks\_table\_html = forks\_df\_display.to\_html(index=False, escape=False, border=1)

    # Issues Count and Status content

    issues\_chart = None

    issues\_pie\_chart = None

    if (repo\_data['forks\_count'] != 0 or repo\_data['open\_issues\_count'] != 0) and len(commit\_frequency) > 0 and repo\_data['stargazers\_count'] != 0:

        issues\_chart = chart\_builder.plot\_issue\_count\_by\_month(issue\_counts\_by\_month)

        issues\_pie\_chart = chart\_builder.plot\_issue\_pie\_chart(issue\_pie\_chart\_data)

    #side bar

    st.sidebar.title("Navigation")

    nav\_option = st.sidebar.radio(

        "Go to Section:",

        ("Summary Report", "Commit Frequency", "Forks Details", "Issues Count and Status")

    )

    # Display content based on nav\_option

    if nav\_option == "Summary Report":

        st.header("Summary Report based on GitHub URL")

        st.markdown(summary\_report\_html, unsafe\_allow\_html=True)

    elif nav\_option == "Commit Frequency":

        if commit\_frequency\_chart:

            st.header("Commit Frequency")

            st.plotly\_chart(commit\_frequency\_chart)

        else:

            st.write("Need more information to generate metrics.")

    elif nav\_option == "Forks Details":

        if forks\_chart and forks\_table\_html:

            st.header("Forks Count by overtime period")

            st.plotly\_chart(forks\_chart)

            st.header("Forking Project Other People Information")

            st.write(forks\_table\_html, unsafe\_allow\_html=True)

        else:

            st.write("Need more information to generate metrics.")

    elif nav\_option == "Issues Count and Status":

        if issues\_chart and issues\_pie\_chart:

            st.header("Issues Count by Over Time period")

            st.plotly\_chart(issues\_chart)

            st.header("Issue Status Overview")

            st.plotly\_chart(issues\_pie\_chart)

        else:

            st.write("Need more information to generate metrics.")

    # Optional: Natural Language Query section

    st.header("Natural Language Query")

    query = st.text\_input("Ask a question (e.g., 'show commit frequency')", key="nlp\_query\_1")

    if query:

        result = nlp\_processor.process\_query(query)

        if result == 'commit\_frequency':

            st.plotly\_chart(chart\_builder.plot\_commit\_frequency(commit\_frequency))

        elif result == 'issue\_resolution':

            st.write(f"Average issue resolution time: {issue\_resolution:.2f} days")

        elif result == 'pr\_merge\_rate':

            st.write(f"b: {pr\_merge\_rate:.2f}%")

        elif result == 'code\_review\_metrics':

            st.write("Code review metrics not implemented yet.")

        else:

            st.write(result)

**Main root code**

**App.py**

import streamlit as st

from data\_collection.github\_api import GitHubDataCollector

from data\_collection.data\_storage import DataStorage

from metrics.calculator import MetricsCalculator

from visualization.charts import ChartBuilder

from query\_interface.nlp\_processor import NLPProcessor

from visualization.dashboard import display\_summary, load\_data

token = st.secrets["github"]["key"]

st.title("Developer Performance Dashboard")

repo\_url = st.text\_input("Enter GitHub Repository URL")

if repo\_url:

    collector = GitHubDataCollector(token)

    data\_storage = DataStorage()

    nlp\_processor = NLPProcessor()

    chart\_builder = ChartBuilder()

    progress\_bar = st.progress(0)

    try:

        repo\_data = collector.get\_repo\_data(repo\_url)

        commits\_data = collector.get\_commits\_data(repo\_url)

        issues\_data = collector.get\_issues\_data(repo\_url)

        forks\_data = collector.get\_forks\_data(repo\_url)

        pull\_requests\_data = collector.get\_pull\_requests\_data(repo\_url)

        reviews\_data = collector.get\_code\_reviews\_data(repo\_url)

        data\_storage.save\_data\_to\_csv(repo\_data, f"{repo\_data['name']}\_repo.csv")

        data\_storage.save\_data\_to\_csv(commits\_data, f"{repo\_data['name']}\_commits.csv")

        data\_storage.save\_data\_to\_csv(issues\_data, f"{repo\_data['name']}\_issues.csv")

        data\_storage.save\_data\_to\_csv(pull\_requests\_data, f"{repo\_data['name']}\_pull\_requests.csv")

        data\_storage.save\_data\_to\_csv(reviews\_data, f"{repo\_data['name']}\_reviews.csv")

        progress\_bar.progress(100)

        # Cache data and metrics

        cached\_data = load\_data(

            repo\_data, commits\_data, issues\_data, forks\_data, pull\_requests\_data, reviews\_data

        )

        display\_summary(

            repo\_data,

            collector.github.get\_repo(collector.\_extract\_repo\_name(repo\_url)).owner,

            cached\_data["avg\_star\_rating"],

            forks\_data,

            cached\_data["commit\_frequency"],

            cached\_data["issue\_resolution"],

            cached\_data["issue\_counts\_by\_month"],

            cached\_data["issue\_pie\_chart\_data"],

            cached\_data["pr\_df"],

            cached\_data["issue\_df"],

            cached\_data["pr\_merge\_rate"],

            cached\_data["avg\_comments\_per\_pr"],

            chart\_builder,

            nlp\_processor

        )

    except Exception as e:

        st.error(f"Error fetching data: {e}")