AI-Powered Developer Performance Analytics Dashboard

Project Structure

```
dev_performance_dashboard/
├── data_collection/
│   ├── github_api.py
│   └── data_storage.py
├── metrics/
│   └── calculator.py
├── visualization/
│   ├── charts.py
│   └── dashboard.py
├── query_interface/
│   ├── nlp_processor.py
│   └── rag_model.py
├── app.py
├── requirements.txt
└── README.md
```

data_collection/ github_api.py

```python
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

```python
#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()
```

```python
            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')['resolve
d_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.
```

```python
        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"]
```

```python
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)).owne
r,
            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}")
```