Hybrid Approach Architecture for Likenings 14-07-2024

The architecture for Likenings using the hybrid approach will involve several components that work together to provide real-time scraping with caching for improved performance and reliability. Here's a detailed breakdown:

1. Client Interface

- **Web Application**: Users interact with the web application to enter their product queries.
- **Frontend Framework**: Use a modern frontend framework like React, Vue.js, or Angular for the user interface.

2. API Server

- **Flask Server**: Acts as the backend server handling user queries.
- **Endpoints**: Define endpoints for searching products and returning results.
- **Logic**: Handles incoming requests, checks the cache, initiates scraping if necessary, and returns results to the frontend.

3. Caching Layer

- **Redis**: Used to cache the results of recent queries.
- **Cache Management**: Store and retrieve cached data, and set expiration policies to keep the cache updated.

4. Scraping Services

- **Individual Scrapers**: Separate scraping scripts or services for each e-commerce platform (e.g., Amazon, Flipkart, Croma, etc.).
- **Concurrency**: Use asyncio or similar libraries to run scraping tasks concurrently for faster data retrieval.
- **Error Handling**: Implement robust error handling to manage issues like product not found, out of stock, or site access problems.

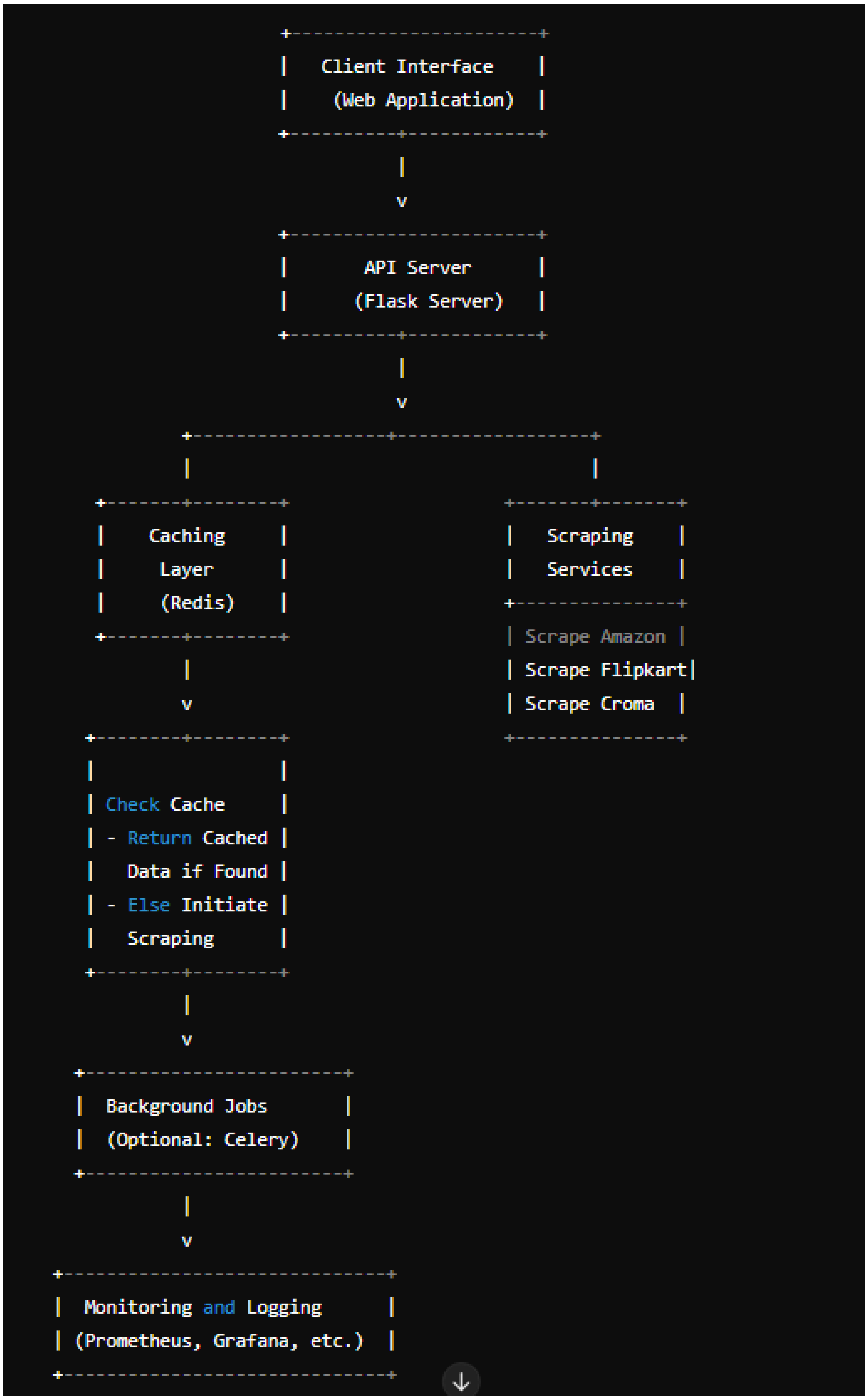
5. Background Jobs

- **Job Scheduler (Optional)**: Use Celery or a similar tool to pre-scrape popular products at regular intervals, if needed.

6. Monitoring and Logging

- **Monitoring Tools**: Implement tools like Prometheus and Grafana for monitoring the performance and health of your services.
- **Logging**: Use logging frameworks to capture and analyze errors and performance metrics.

Detailed Architecture Diagram



```
                     +-----------------------+
                     |   Client Interface    |
                     |    (Web Application)  |
                     +----------+------------+
                                |
                                v
                     +-----------------------+
                     |       API Server      |
                     |      (Flask Server)   |
                     +----------+------------+
                                |
                                v
            +------------------+------------------+
            |                                     |
    +-------+--------+                    +-------+-------+
    |    Caching     |                    |   Scraping    |
    |     Layer      |                    |   Services    |
    |     (Redis)    |                    +---------------+
    +-------+--------+                    | Scrape Amazon |
            |                             | Scrape Flipkart|
            v                             | Scrape Croma  |
   +--------+--------+                    +---------------+
   |                 |
   | Check Cache     |
   | - Return Cached |
   |   Data if Found |
   | - Else Initiate |
   |   Scraping      |
   +--------+--------+
            |
            v
  +------------------------+
  |  Background Jobs       |
  |  (Optional: Celery)    |
  +------------------------+
            |
            v
+------------------------------+
|  Monitoring and Logging      |
| (Prometheus, Grafana, etc.)  |
+------------------------------+
```

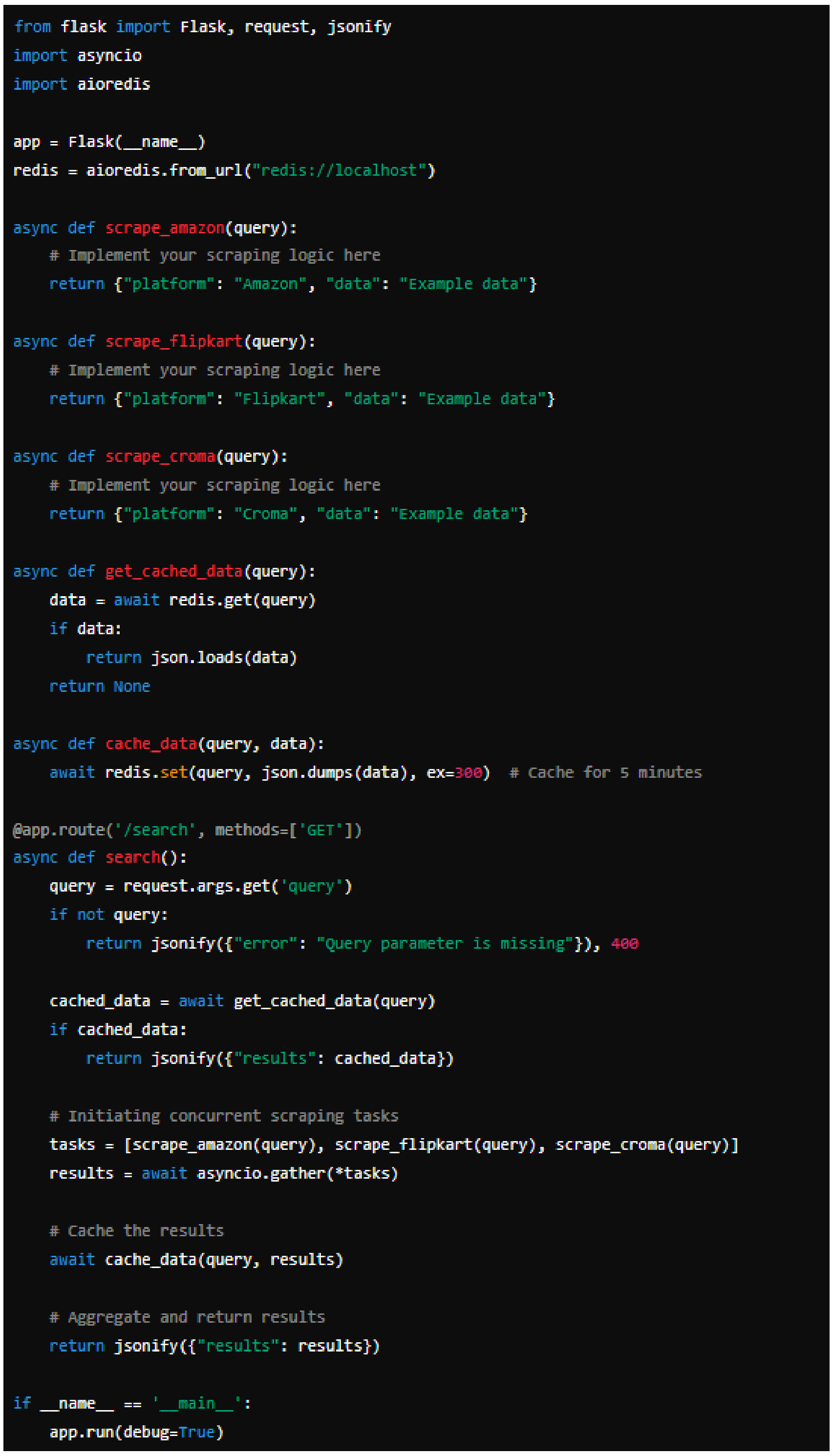
Component Breakdown

1. Client Interface

- **Framework**: React, Vue.js, Angular, or similar.
- **Functionality**: Input fields for user queries, display of results, and user interactions.

2. API Server

- **Flask Server**: Main server to handle incoming requests.
- **Endpoints**:
    - /search: Accepts user queries and returns product data.



```python
from flask import Flask, request, jsonify
import asyncio
import aioredis

app = Flask(__name__)
redis = aioredis.from_url("redis://localhost")

async def scrape_amazon(query):
    # Implement your scraping logic here
    return {"platform": "Amazon", "data": "Example data"}

async def scrape_flipkart(query):
    # Implement your scraping logic here
    return {"platform": "Flipkart", "data": "Example data"}

async def scrape_croma(query):
    # Implement your scraping logic here
    return {"platform": "Croma", "data": "Example data"}

async def get_cached_data(query):
    data = await redis.get(query)
    if data:
        return json.loads(data)
    return None

async def cache_data(query, data):
    await redis.set(query, json.dumps(data), ex=300)  # Cache for 5 minutes

@app.route('/search', methods=['GET'])
async def search():
    query = request.args.get('query')
    if not query:
        return jsonify({"error": "Query parameter is missing"}), 400

    cached_data = await get_cached_data(query)
    if cached_data:
        return jsonify({"results": cached_data})

    # Initiating concurrent scraping tasks
    tasks = [scrape_amazon(query), scrape_flipkart(query), scrape_croma(query)]
    results = await asyncio.gather(*tasks)

    # Cache the results
    await cache_data(query, results)

    # Aggregate and return results
    return jsonify({"results": results})

if __name__ == '__main__':
    app.run(debug=True)
```

3. Caching Layer

- **Redis**: To cache the results temporarily.
- **Configuration**: Set up Redis with an appropriate expiration policy to balance freshness and performance.

4. Scraping Services

- **Separate Scripts**: Each e-commerce platform has its own scraping script or microservice.
- **Concurrency**: Use asyncio for concurrent scraping to improve performance.
- **Error Handling**: Implement detailed error handling to manage exceptions and retry logic.

5. Background Jobs (Optional)

- **Job Scheduler**: Celery or similar for regular scraping of popular products.
- **Pre-Scraping**: Regularly update the database with popular product data.

6. Monitoring and Logging

- **Monitoring**: Set up Prometheus and Grafana for real-time monitoring.
- **Logging**: Use a logging framework to capture errors and performance metrics.

Summary

This architecture ensures that your application can handle real-time scraping efficiently while maintaining performance through caching. It balances the need for fresh data with the practicality of reducing the load on e-commerce sites and ensuring fast response times for users.