



Sanjeevkarthick /
Exno.6-Prompt-Engg



Code

Pull requests

Actions

Projects

Wiki

Security

Insights



main ▾

Exno.6-Prompt-Engg / README.md



Sanjeevkarthick EXp6README.md

e6d75c5 · now



1028 lines (843 loc) · 39.5 KB

Exno.6-Prompt-Engg

Date:

Register no.

Aim: Development of Python Code



main ▾

Exno.6-Prompt-Engg / README.md

↑ Top

Preview

Code

Blame

Raw



Algorithm: Write and implement Python code that integrates with multiple AI tools to automate the task of interacting with APIs, comparing outputs, and generating actionable insights.

CHATGPT Here's a Python-based framework that integrates with multiple AI tools or APIs, automates their interaction, compares outputs, and generates actionable insights. This example assumes you want to:

Interact with multiple AI APIs (e.g., OpenAI, Cohere, Anthropic).

Provide the same prompt/query to each.

Compare the responses (qualitatively or quantitatively).

Generate a summary/insight based on the comparison.

💡 Requirements

Install required libraries:

```
bash Copy Edit pip install openai cohere anthropic
```

Python Code python Copy Edit
import openai import cohere import anthropic from difflib import SequenceMatcher

API Keys (replace with your actual keys)

```
openai.api_key = "your-openai-api-key" cohere_client = cohere.Client("your-cohere-api-key") anthropic_client = anthropic.Anthropic(api_key="your-anthropic-api-key")
```

Your common query or task

```
query = "Summarize the key takeaways from the latest advancements in AI safety."
```

```
def get_openai_response(prompt): response = openai.ChatCompletion.create(model="gpt-4", messages=[{"role": "user", "content": prompt}], temperature=0.7 ) return response['choices'][0]['message']['content']
```

```
def get_cohere_response(prompt): response = cohere_client.generate(model='command-r-plus', prompt=prompt, max_tokens=300, temperature=0.7 ) return response.generations[0].text
```

```
def get_anthropic_response(prompt): response = anthropic_client.messages.create(model="claude-3-opus-20240229", max_tokens=300, temperature=0.7, messages=[{"role": "user", "content": prompt}]) return response.content[0].text
```

```
def compare_outputs(output1, output2): similarity = SequenceMatcher(None, output1, output2).ratio() return similarity
```

```
def summarize_insights(responses): return f"""🔍 Insights Summary: - OpenAI: {responses['openai'][:200]}... - Cohere: {responses['cohere'][:200]}... - Anthropic: {responses['anthropic'][:200]}..."""
```

💡 Similarity Scores:

- OpenAI vs Cohere: {compare_outputs(responses['openai'], responses['cohere']):.2f}
- OpenAI vs Anthropic: {compare_outputs(responses['openai'], responses['anthropic']):.2f}
- Cohere vs Anthropic: {compare_outputs(responses['cohere'], responses['anthropic']):.2f}

```
responses['anthropic']):.2f}  
"""
```

```
def main(): print("💡 Querying all AI APIs..\n") responses = { "openai":  
get_openai_response(query), "cohere": get_cohere_response(query), "anthropic":  
get_anthropic_response(query) }
```

```
    insights = summarize_insights(responses)  
    print(insights)
```

if name == "main": main() ✅ Features Modular functions to easily add/remove AI models.

Uses difflib.SequenceMatcher for basic textual similarity.

Outputs a digestible insight summary.

💡 Possible Extensions Use semantic similarity with embeddings instead of string similarity.

Store results in a database (e.g., SQLite or MongoDB).

Visualize comparisons with charts using libraries like matplotlib or plotly. **DEEPSEEK** AI-Powered API Integration and Analysis Tool Below is a Python implementation that integrates with multiple AI tools to automate API interactions, compare outputs, and generate insights. This solution uses:

OpenAI's GPT for analysis

Hugging Face models for alternative perspectives

Custom comparison logic

Automated reporting

```
python import requests import json import openai from transformers import pipeline  
import pandas as pd from datetime import datetime import matplotlib.pyplot as plt  
import seaborn as sns from typing import List, Dict, Any, Optional
```

```
class APIAnalyzer: def __init__(self, config: Dict[str, Any]): """ Initialize the API analyzer with  
configuration """ self.config = config self.openai_api_key = config.get('openai_api_key')  
self.huggingface_token = config.get('huggingface_token')
```

```
    # Initialize AI services  
    openai.api_key = self.openai_api_key  
    self.huggingface_summarizer = pipeline(  
        "summarization",
```

```

        model="facebook/bart-large-cnn",
        token=self.huggingface_token
    )

    # Results storage
    self.api_responses = {}
    self.analysis_results = {}
    self.comparison_results = {}

def call_api(self, endpoint_name: str, params: Optional[Dict] = None) -> Dict:
    """
    Generic API caller with error handling
    """
    endpoint = self.config['api_endpoints'][endpoint_name]

    try:
        if endpoint['method'].upper() == 'GET':
            response = requests.get(
                endpoint['url'],
                headers=endpoint.get('headers', {}),
                params=params or {}
            )
        else:
            response = requests.post(
                endpoint['url'],
                headers=endpoint.get('headers', {}),
                json=params or {}
            )

        response.raise_for_status()
        data = response.json()
        self.api_responses[endpoint_name] = data
        return data
    except Exception as e:
        print(f"Error calling {endpoint_name}: {str(e)}")
        return {'error': str(e)}

def analyze_with_gpt(self, data: Dict, prompt_template: str) -> Dict:
    """
    Analyze API response using OpenAI GPT
    """
    try:
        prompt = prompt_template.format(data=json.dumps(data, indent=2))

        response = openai.ChatCompletion.create(
            model="gpt-4",
            messages=[
                {"role": "system", "content": "You are an API analysis
assistant. Provide clear, actionable insights."},
                {"role": "user", "content": prompt}
            ],

```

```

        temperature=0.7
    )

analysis = response.choices[0].message.content
return {'analysis': analysis, 'model': 'GPT-4'}
```

except Exception as e:

```
    return {'error': str(e), 'model': 'GPT-4'}
```

def analyze_with_huggingface(self, data: Dict, task: str = "summarization") -> Dict:

```
    """
    Analyze API response using Hugging Face models
    """
    try:
        text_data = json.dumps(data)

        if task == "summarization":
            result = self.huggingface_summarizer(
                text_data,
                max_length=130,
                min_length=30,
                do_sample=False
            )
        return {
            'analysis': result[0]['summary_text'],
            'model': 'BART-large-cnn'
        }
    else:
        return {'error': 'Task not implemented', 'model': 'HuggingFace'}
```

except Exception as e:

```
    return {'error': str(e), 'model': 'HuggingFace'}
```

def compare_responses(self, response1: Dict, response2: Dict, strategy: str = "basic") -> Dict:

```
    """
    Compare two API responses using different strategies
    """
    if strategy == "basic":
        # Basic comparison of structure and key values
        diff = self._basic_comparison(response1, response2)
        return {
            'comparison': diff,
            'strategy': strategy
        }
    elif strategy == "ai":
        # Use AI to identify semantic differences
        return self._ai_comparison(response1, response2)
    else:
        return {'error': 'Invalid comparison strategy'}
```

```

def _basic_comparison(self, a: Dict, b: Dict) -> Dict:
    """Basic structural comparison of two responses"""
    from deepdiff import DeepDiff
    return DeepDiff(a, b, ignore_order=True).to_dict()

def _ai_comparison(self, a: Dict, b: Dict) -> Dict:
    """Use AI to explain differences between responses"""
    try:
        prompt = f"""
            Compare these two API responses and explain the key differences
            in business terms:

            Response 1:
            {json.dumps(a, indent=2)}

            Response 2:
            {json.dumps(b, indent=2)}

            Focus on meaningful differences that might require action or
            investigation.
        """
        response = openai.ChatCompletion.create(
            model="gpt-4",
            messages=[
                {"role": "system", "content": "You are an API comparison
expert."},
                {"role": "user", "content": prompt}
            ],
            temperature=0.5
        )

        return {
            'comparison': response.choices[0].message.content,
            'strategy': 'AI semantic comparison',
            'model': 'GPT-4'
        }
    except Exception as e:
        return {'error': str(e)}

def generate_report(self, format: str = "markdown") -> str:
    """
    Generate a comprehensive report of findings
    """
    report = f"# API Analysis Report\n\n"
    report += f"Generated at: {datetime.now().isoformat()}\n\n"

    # API Responses section
    report += "## API Responses\n"
    for name, response in self.api_responses.items():
        report += f"### {name}\n"
        report += f"```json\n{json.dumps(response, indent=2)}\n```\n\n"

```

```

# Analysis section
report += "## AI Analysis\n"
for name, analysis in self.analysis_results.items():
    report += f"### {name} ({analysis.get('model', 'unknown')})\n"
    report += f"{analysis.get('analysis', 'No analysis available')}\n\n"

# Comparison section
if self.comparison_results:
    report += "## Comparison Results\n"
    for name, comparison in self.comparison_results.items():
        report += f"### {name}\n"
        report += f"Strategy: {comparison.get('strategy',
'unknown')}\n"
        if 'comparison' in comparison:
            if isinstance(comparison['comparison'], dict):
                report +=
f"```json\n{json.dumps(comparison['comparison'], indent=2)}\n```\n"
            else:
                report += f"{comparison['comparison']}\n"
        report += "\n"

# Visualizations if data is suitable
if self._is_data_visualizable():
    self.generate_visualizations()
    report += "## Data Visualizations\n"
    report += "See generated charts in the output directory.\n"

# Actionable insights
report += "## Recommended Actions\n"
report += self._generate_actions()

if format == "html":
    from markdown2 import markdown
    return markdown(report)

return report

def _is_data_visualizable(self) -> bool:
    """Check if data is suitable for visualization"""
    # Simplified check - in real implementation would be more
    sophisticated
    for response in self.api_responses.values():
        if isinstance(response, list) and len(response) > 1:
            if isinstance(response[0], dict) and len(response[0]) > 1:
                return True
    return False

def generate_visualizations(self):
    """Generate visualizations from API data"""
    try:
        sns.set_theme(style="whitegrid")

```

```

        for name, response in self.api_responses.items():
            if isinstance(response, list) and len(response) > 1:
                df = pd.DataFrame(response)

                    # Generate various plot types based on data
                    for col in df.select_dtypes(include=['int64',
'float64']).columns:
                        plt.figure(figsize=(10, 6))
                        sns.histplot(df[col])
                        plt.title(f"Distribution of {col} in {name}")
                        plt.savefig(f"{name}_{col}_distribution.png")
                        plt.close()

                    if len(df.select_dtypes(include=['int64',
'float64']).columns) > 1:
                        numeric_cols = df.select_dtypes(include=['int64',
'float64']).columns
                        plt.figure(figsize=(10, 8))
                        sns.heatmap(df[numeric_cols].corr(), annot=True)
                        plt.title(f"Correlation Matrix for {name}")
                        plt.savefig(f"{name}_correlation.png")
                        plt.close()

        except Exception as e:
            print(f"Visualization error: {str(e)}")

def _generate_actions(self) -> str:
    """Generate actionable insights from all analyses"""
    try:
        prompt = """
        Based on the following API analyses and comparisons, provide a
bulleted list of
        clear, actionable recommendations. Focus on:
        - Data inconsistencies that need resolution
        - Performance optimization opportunities
        - Business insights that could inform strategy
        - Any immediate technical actions required

        Analyses:
        {}
        """.format(json.dumps(self.analysis_results, indent=2))

        response = openai.ChatCompletion.create(
            model="gpt-4",
            messages=[
                {"role": "system", "content": "You are a technical
business analyst."},
                {"role": "user", "content": prompt}
            ],
            temperature=0.5
        )
    
```

```

        return response.choices[0].message.content

    except Exception as e:
        return f"Could not generate actions: {str(e)}"

def run_full_analysis(self, endpoints_to_analyze: List[str]) -> Dict:
    """
    Run complete analysis workflow:
    1. Call specified APIs
    2. Analyze each response with multiple AI models
    3. Compare responses
    4. Generate report
    """
    results = {}

    # Step 1: Call APIs
    for endpoint in endpoints_to_analyze:
        if endpoint in self.config['api_endpoints']:
            print(f"Calling {endpoint} API...")
            self.call_api(endpoint)

    # Step 2: Analyze responses
    for endpoint, response in self.api_responses.items():
        print(f"Analyzing {endpoint} response...")
        gpt_analysis = self.analyze_with_gpt(
            response,
            "Analyze this API response and identify key
insights:\n\n{data}"
        )
        hf_analysis = self.analyze_with_huggingface(response)

        self.analysis_results[f"{endpoint}_gpt"] = gpt_analysis
        self.analysis_results[f"{endpoint}_hf"] = hf_analysis

    # Step 3: Compare responses if multiple
    if len(self.api_responses) > 1:
        print("Comparing API responses...")
        endpoints = list(self.api_responses.keys())
        for i in range(len(endpoints)):
            for j in range(i+1, len(endpoints)):
                name = f"{endpoints[i]}_vs_{endpoints[j]}"
                self.comparison_results[name] = self.compare_responses(
                    self.api_responses[endpoints[i]],
                    self.api_responses[endpoints[j]],
                    strategy="ai"
                )

    # Step 4: Generate report
    print("Generating report...")
    report = self.generate_report()

    return {
        'api_responses': self.api_responses,

```

```
'analyses': self.analysis_results,  
'comparisons': self.comparison_results,  
'report': report  
}
```

Example Usage

```
if name == "main": # Configuration - in a real app this would come from a config file  
config = { "openai_api_key": "your-openai-key", "huggingface_token": "your-hf-token",  
"api_endpoints": { "weather_api": { "url": "https://api.open-meteo.com/v1/forecast",  
"method": "GET", "headers": {}, "default_params": { "latitude": 40.71, "longitude": -74.01,  
"hourly": "temperature_2m" } }, "financial_api": { "url":  
"https://www.alphavantage.co/query", "method": "GET", "headers": {}, "default_params":  
{ "function": "TIME_SERIES_DAILY", "symbol": "IBM", "apikey": "demo" # Replace with real  
API key } } }
```

```
# Initialize analyzer  
analyzer = APIAnalyzer(config)  
  
# Run full analysis  
results = analyzer.run_full_analysis(["weather_api", "financial_api"])  
  
# Save report  
with open("api_analysis_report.md", "w") as f:  
    f.write(results['report'])  
  
print("Analysis complete. Report saved to api_analysis_report.md")
```

Key Features of This Implementation: Multi-API Integration:

Supports both GET and POST requests

Handles authentication headers

Manages API responses with error handling

Multi-AI Analysis:

Uses OpenAI GPT-4 for detailed analysis

Incorporates Hugging Face models (BART for summarization)

Provides different perspectives from different AI models

Comparison Engine:

Basic structural comparison using DeepDiff

AI-powered semantic comparison

Identifies meaningful business differences

Automated Reporting:

Generates Markdown reports with sections for:

Raw API responses

AI analyses

Comparison results

Visualizations

Actionable recommendations

Can be extended to HTML or other formats

Data Visualization:

Automatic generation of charts and graphs

Histograms for distribution analysis

Correlation matrices for numeric data

Actionable Insights:

Uses AI to synthesize findings into clear recommendations

Focuses on business value and technical actions

How to Use: Replace the placeholder API keys with your actual keys

Configure the API endpoints you want to analyze

Run the script to:

Call the APIs

Analyze responses with AI

Compare results

Generate a comprehensive report

Extensions You Could Add: Add more AI services (Claude, Gemini, etc.)

Implement performance benchmarking

Add data validation rules

Create a web interface or dashboard

Add scheduling for regular analysis

Implement alerting for significant changes

This implementation provides a solid foundation for automating API analysis workflows using multiple AI tools to generate actionable business insights. **GEMINI** This is an ambitious request that requires a robust setup and understanding of various AI services. I will provide a conceptual framework and a simplified Python implementation, demonstrating how different AI tools could be integrated.

Core Idea: We'll simulate a scenario where we interact with two different "APIs" (represented by functions returning JSON), compare their outputs, and then use an AI model (like a Large Language Model) to generate insights based on those comparisons.

Components:

API Interaction: Using the requests library to simulate API calls. In a real-world scenario, you'd replace these with actual API endpoints. **Output Comparison:** Using DeepDiff for detailed JSON comparison. This helps identify exact changes, additions, and removals.

Insight Generation (AI Tool 1 - e.g., OpenAI/Gemini API): Using a large language model (LLM) to interpret the differences found and generate actionable insights. You would need an API key for a service like OpenAI's GPT or Google's Gemini.

Hypothetical AI Tool 2 (e.g., Sentiment Analysis on a subset of data): This could be another AI service that processes specific parts of the API responses (e.g., text fields) to extract sentiment, entities, etc., which then feeds into the insight generation.

Prerequisites:

You'll need to install the following Python libraries:

Bash

```
pip install requests deepdiff openai google-generativeai
```

Important Notes before Running:

API Keys: For openai or google-generativeai, you will need to replace 'YOUR_OPENAI_API_KEY' or 'YOUR_GEMINI_API_KEY' with your actual API keys. Never hardcode API keys in production code. Use environment variables or a secure configuration management system. Simulated APIs: The simulate_api_call functions are placeholders. In a real application, these would be requests.get() or requests.post() calls to actual external APIs. Error Handling: For brevity, extensive error handling (e.g., network errors, API rate limits, invalid responses) is simplified. In a production system, this would be crucial. Context for LLM: The quality of insights from the LLM heavily depends on the prompt. Providing clear context about the API, expected outputs, and the nature of the differences will yield better results. Python Code Implementation

Python

```
import requests import json import os from deepdiff import DeepDiff from datetime  
import datetime
```

--- AI Tool 1: Google Gemini (or OpenAI) for Insight Generation ---

Choose one of the following based on your preference and API key availability

Uncomment the one you want to use.

Option 1: Google Gemini

```
import google.generativeai as genai
```

Replace with your actual Gemini API key

```
GEMINI_API_KEY = os.getenv("GEMINI_API_KEY", "YOUR_GEMINI_API_KEY")  
genai.configure(api_key=GEMINI_API_KEY) LLM_MODEL =  
genai.GenerativeModel('gemini-pro')
```

Option 2: OpenAI (Uncomment if you prefer OpenAI)

```
from openai import OpenAI
```

```
OPENAI_API_KEY =  
os.getenv("OPENAI_API_KEY",  
"YOUR_OPENAI_API_KEY")
```

```
client =  
OpenAI(api_key=OPENAI_API_KEY)
```

```
LLM_MODEL_NAME = "gpt-3.5-turbo" #  
Or "gpt-4o", etc.
```

--- Helper Functions ---

```
def simulate_api_call(api_name, version="v1"): """ Simulates an API call and returns a  
JSON response. In a real scenario, this would be an actual HTTP request. """  
print(f"Simulating API call for {api_name} ({version})...") if api_name == "products" and  
version == "v1": return { "timestamp": datetime.now().isoformat(), "status": "success",  
"data": [ {"id": "P001", "name": "Laptop Pro", "price": 1200, "stock": 50, "category":  
"Electronics"}, {"id": "P002", "name": "Mechanical Keyboard", "price": 150, "stock": 100,  
"category": "Accessories"}, {"id": "P003", "name": "Gaming Mouse", "price": 75, "stock":  
75, "category": "Accessories"} ] } elif api_name == "products" and version == "v2": #  
Simulate a change in data (e.g., price update, new item, removed item) return {  
"timestamp": datetime.now().isoformat(), "status": "success", "data": [ {"id": "P001",  
"name": "Laptop Pro", "price": 1250, "stock": 45, "category": "Electronics",  
"discount_eligible": True}, # Price and stock change, new field {"id": "P002", "name":  
"Mechanical Keyboard", "price": 150, "stock": 100, "category": "Accessories"}, {"id":  
"P004", "name": "USB-C Hub", "price": 40, "stock": 200, "category": "Peripherals"} # New  
item # P003 is implicitly "removed" in this version ] } elif api_name == "users" and  
version == "v1": return { "timestamp": datetime.now().isoformat(), "users": [ {"id":  
"U001", "username": "alice_smith", "email": "alice@example.com"}, {"id": "U002",  
"username": "bob_johnson", "email": "bob@example.com"} ] } else: return {"error": "API  
not found or version not supported"}
```

```
def compare_json_outputs(json1, json2): """ Compares two JSON objects using DeepDiff and returns the differences. """ print("\nComparing JSON outputs...") diff = DeepDiff(json1, json2, ignore_order=True) if diff: print("Differences found:") print(json.dumps(diff, indent=2)) else: print("No significant differences found.") return diff
```

```
def generate_insights_with_llm(comparison_results, context=""): """ Uses an LLM to generate actionable insights from comparison results. """ print("\nGenerating actionable insights with AI...") prompt = f""" Analyze the following comparison results between two API responses. The comparison highlights changes (added, removed, or modified items/values) between API versions or different API calls. Provide a concise summary of the key changes and suggest actionable insights or implications based on these differences.
```

Context about the API (if available): {context} 

Comparison Results (DeepDiff format):
{json.dumps(comparison_results, indent=2)}

Please focus on:

1. What are the most significant changes? (e.g., major price changes, new/removed products, changes in user data structure)
2. What could be the business implications of these changes? (e.g., inventory issues, new product launch, data migration needed)
3. Suggest clear, actionable next steps or areas for further investigation.

"""

```
try:  
    # For Google Gemini:  
    response = LLM_MODEL.generate_content(prompt)  
    insights = response.text  
  
    # For OpenAI (uncomment if using OpenAI):  
    # response = client.chat.completions.create(  
    #     model=LLM_MODEL_NAME,  
    #     messages=[  
    #         {"role": "system", "content": "You are an AI assistant  
that analyzes API comparison results and provides actionable  
insights."},  
    #         {"role": "user", "content": prompt}  
    #     ]  
    # )  
    # insights = response.choices[0].message.content  
  
    print("\n--- Actionable Insights (Generated by AI) ---")  
    print(insights)  
    return insights  
except Exception as e:
```

```
    print(f"Error generating insights with LLM: {e}")
    return "Could not generate insights due to an error."
```

```
def perform_sentiment_analysis(text_data): """ Hypothetical AI Tool 2: Performs sentiment analysis on text data. This would typically call a dedicated NLP/sentiment API. For this example, we'll simulate a basic sentiment analysis. """ print("\nPerforming hypothetical sentiment analysis...") if "good" in text_data.lower() or "excellent" in text_data.lower(): return {"sentiment": "Positive", "score": 0.9} elif "bad" in text_data.lower() or "error" in text_data.lower(): return {"sentiment": "Negative", "score": 0.1} else: return {"sentiment": "Neutral", "score": 0.5}
```

--- Main Automation Workflow ---

```
def automate_api_analysis(): print("--- Starting Automated API Analysis Workflow ---)
```

```
# Step 1: Interact with APIs
print("\n--- Fetching API Outputs ---")
products_v1 = simulate_api_call("products", "v1")
products_v2 = simulate_api_call("products", "v2")
users_v1 = simulate_api_call("users", "v1")
```



```
# Display raw outputs (optional)
print("\nRaw Products V1:", json.dumps(products_v1, indent=2))
print("\nRaw Products V2:", json.dumps(products_v2, indent=2))
```

```
# Step 2: Compare API Outputs
print("\n--- Comparing API Outputs (Products V1 vs V2) ---")
product_diff = compare_json_outputs(products_v1, products_v2)
```

```
# Step 3: Integrate with a hypothetical AI Tool 2 (e.g., Sentiment Analysis)
```

```
# Let's say we want to analyze the 'status' field or a 'description' if it existed
```

```
print("\n--- Integrating with Hypothetical AI Tool 2 (Sentiment Analysis) ---")
```

```
product_v1_status_sentiment =
perform_sentiment_analysis(products_v1.get("status", ""))
product_v2_status_sentiment =
perform_sentiment_analysis(products_v2.get("status", ""))
print(f"Products V1 Status Sentiment: {product_v1_status_sentiment}")
print(f"Products V2 Status Sentiment: {product_v2_status_sentiment}")
```

```
# You could feed these sentiment results into the LLM insights generation if relevant.
```

```
# For now, we'll primarily feed the DeepDiff output.
```

```
# Step 4: Generate Actionable Insights using AI Tool 1 (LLM)
print("\n--- Generating Actionable Insights ---")
```

```

if product_diff:
    llm_context = "This comparison is between two versions (v1 and v2)
of a product inventory API. 'data' contains a list of products. Focus on
identifying new, removed, or modified products, and changes in
attributes like price, stock, and new fields."
    generate_insights_with_llm(product_diff, context=llm_context)
else:
    print("No differences found in products API. No specific insights to
generate.")

print("\n--- Automated API Analysis Workflow Completed ---")

```

if name == "main": automate_api_analysis() How to Run: Save: Save the code as a Python file (e.g., api_automator.py). Set API Keys: Google Gemini: Go to Google AI Studio to get your API key. Then, set it as an environment variable: export GEMINI_API_KEY="your_api_key_here" (on Linux/macOS) or \$env:GEMINI_API_KEY="your_api_key_here" (on Windows PowerShell). Or, directly replace "YOUR_GEMINI_API_KEY" in the code (not recommended for production). OpenAI: Go to OpenAI Platform to get your API key. Set it as an environment variable: export OPENAI_API_KEY="your_api_key_here" (on Linux/macOS) or \$env:OPENAI_API_KEY="your_api_key_here" (on Windows PowerShell). If using OpenAI, remember to uncomment the OpenAI related lines and comment out the Gemini related lines. Install Libraries: Bash

pip install requests deepdiff openai google-generativeai Execute: Run the script from your terminal: Bash

python api_automator.py Expected Output (Example - will vary slightly with LLM output): --- Starting Automated API Analysis Workflow ---

--- Fetching API Outputs --- Simulating API call for products (v1)... Simulating API call for products (v2)... Simulating API call for users (v1)...

Raw Products V1: { "timestamp": "2025-05-24T19:23:15.123456", "status": "success",
"data": [{ "id": "P001", "name": "Laptop Pro", "price": 1200, "stock": 50, "category": "Electronics" }, { "id": "P002", "name": "Mechanical Keyboard", "price": 150, "stock": 100, "category": "Accessories" }, { "id": "P003", "name": "Gaming Mouse", "price": 75, "stock": 75, "category": "Accessories" }] }

Raw Products V2: { "timestamp": "2025-05-24T19:23:15.123456", "status": "success",
"data": [{ "id": "P001", "name": "Laptop Pro", "price": 1250, "stock": 45, "category": "Electronics", "discount_eligible": true }, { "id": "P002", "name": "Mechanical Keyboard", "price": 150, "stock": 100, "category": "Accessories" }, { "id": "P004", "name": "USB-C Hub", "price": 40, "stock": 200, "category": "Peripherals" }] }

--- Comparing API Outputs (Products V1 vs V2) ---

Comparing JSON outputs... Differences found: { "values_changed": { "root['data'][0]['price']": { "old_value": 1200, "new_value": 1250 }, "root['data'][0]['stock']": { "old_value": 50, "new_value": 45 } }, "iterable_item_added": { "root['data'][2]": { "id": "P004", "name": "USB-C Hub", "price": 40, "stock": 200, "category": "Peripherals" } }, "iterable_item_removed": { "root['data'][2]": { "id": "P003", "name": "Gaming Mouse", "price": 75, "stock": 75, "category": "Accessories" } }, "dictionary_item_added": { "root['data'][0]['discount_eligible']": true } }

--- Integrating with Hypothetical AI Tool 2 (Sentiment Analysis) --- Performing hypothetical sentiment analysis... Products V1 Status Sentiment: {'sentiment': 'Positive', 'score': 0.9} Performing hypothetical sentiment analysis... Products V2 Status Sentiment: {'sentiment': 'Positive', 'score': 0.9}

--- Generating Actionable Insights ---

Generating actionable insights with AI...

--- Actionable Insights (Generated by AI) --- Key Changes:

- **Product P001 (Laptop Pro):** Price increased from \$1200 to \$1250, and stock decreased from 50 to 45. A new attribute `discount_eligible` was added and set to True .
- **Product P003 (Gaming Mouse):** This product was removed from the inventory in v2.
- **Product P004 (USB-C Hub):** A new product was added to the inventory in v2, with a price of \$40 and 200 units in stock.

Business Implications:

- **Pricing Strategy:** The price increase for Laptop Pro could indicate a new pricing strategy or increased production costs. The `discount_eligible` flag suggests potential for future promotions.
- **Inventory Management:** The decrease in Laptop Pro stock (from 50 to 45) and the removal of Gaming Mouse (P003) from the list indicate inventory adjustments. The large stock of the new USB-C Hub (P004) suggests a new focus or high demand expectation.
- **Product Lifecycle:** The removal of P003 might signal the end-of-life for that product or a shift in product offerings. The introduction of P004 shows product expansion.

Actionable Next Steps:

1. **Investigate Laptop Pro Changes:** Confirm the reason for the price increase and stock reduction for Laptop Pro. Ensure `discount_eligible` is correctly implemented for future marketing.

2. **Review Gaming Mouse Removal:** Verify if the removal of P003 was intentional (e.g., discontinued, out of stock indefinitely) and update relevant sales/marketing materials.
3. **Monitor USB-C Hub Performance:** Track sales and customer feedback for the new USB-C Hub (P004) to assess its performance and demand.
4. **Update Internal Systems:** Ensure all internal systems (e.g., e-commerce platform, inventory management, CRM) are updated to reflect the product changes in V2.
5. **Communicate Changes:** Inform sales, marketing, and customer support teams about these product updates, especially the price change and new/removed products.

--- Automated API Analysis Workflow Completed --- Explanation of Integration: API Interaction (requests / simulate_api_call):

The simulate_api_call function acts as our stand-in for requests.get() or requests.post(). In a real scenario, you'd replace this with actual HTTP calls to your APIs. This is the first step of gathering raw data. Output Comparison (deepdiff):

The compare_json_outputs function takes two JSON objects (simulated API responses) and uses DeepDiff. DeepDiff is a powerful library that provides a detailed, structured output of all differences (added, removed, changed values, moved items in lists, etc.). This granular comparison is crucial for understanding the impact of API changes. Insight Generation (AI Tool 1 - Google Gemini/OpenAI):

The generate_insights_with_llm function is the core of our AI integration. It takes the DeepDiff output and a context string. It constructs a detailed prompt for the LLM, asking it to interpret the differences, identify business implications, and suggest actionable steps. The LLM then processes this information and returns human-readable insights. This automates the analysis that a human might typically do after reviewing API diffs. Hypothetical AI Tool 2 (Sentiment Analysis):

The `perform_sentiment_analysis` function is a placeholder for another AI service. Imagine you have an API that returns user reviews or product descriptions, and you want to quickly gauge the sentiment of those texts. This function demonstrates how you could integrate another AI tool to process a subset of the data obtained from the APIs, and those results could then feed into the overall insight generation by the LLM (though in this example, it's just printed out separately). Scalability and Enhancements: Real API Endpoints: Replace `simulate_api_call` with actual requests calls to your target APIs, including authentication, headers, and error handling. Dynamic API Discovery: Use API definition files (e.g., OpenAPI/Swagger) to dynamically discover endpoints and expected schemas. Automated Testing: Integrate this workflow into CI/CD pipelines to automatically compare API responses before and after deployments. More Sophisticated AI Tools: Anomaly Detection: Use ML models to detect unusual patterns or outliers in API response data (e.g., sudden spikes in error rates, unexpected data values). Predictive Analytics: If your API returns time-series data (e.g., stock levels over time), use AI to predict future trends and flag potential issues. Natural Language Generation (NLG): Further refine the LLM output to generate more structured reports or even automated emails based on the insights. Data Validation with AI: Train models to validate API responses against expected data types, formats, and ranges, flagging deviations that traditional validation might miss. Reporting & Visualization: Store the comparison results and AI-generated insights in a database. Use a reporting tool or build a dashboard to visualize trends in API changes and their impact. Alerting: Set up alerts (e.g., Slack, email) if critical differences are detected or if AI insights highlight high-severity issues. Version Control Integration: Link API responses and their comparisons directly to your code's version control system (e.g., Git) to track changes alongside code. This framework provides a solid starting point for building sophisticated API automation powered by multiple AI tools. The key is to break down the problem into manageable steps and leverage the strengths of each AI service.

BLACKBOXAI To automate interactions with multiple AI APIs using Python, you can leverage libraries like `requests` for API calls and `pandas` for data manipulation. Integrating various AI tools allows you to compare outputs and derive insights, enhancing your workflow and decision-making processes.

Step 1: Set Up Your Environment

Ensure you have Python installed (version 3.7 or later). Install necessary libraries using pip:

```
bash Run Copy code pip install openai requests pandas
```

Step 2: Define API Keys and Endpoints

Store your API keys securely, preferably using environment variables. Define the endpoints for the AI tools you want to integrate.

```
python 9 lines Click to expand import os ...
```

Step 3: Create Functions for API Interactions

Implement functions to interact with each API, handling requests and responses.

```
python 22 lines Click to expand import requests ...
```

Step 4: Compare Outputs from Different APIs

Create a function to compare the outputs from the APIs and generate actionable insights. python 17 lines Click to expand import pandas as pd ... Step 5: Generate Actionable Insights

Analyze the comparison results and derive insights. python 7 lines Click to expand def generate_insights(comparison_df):

Example logic for generating insights

... Step 6: Main Execution Flow

Combine everything into a main function to execute the workflow. python 12 lines Click to close def main(): prompt = "What are the benefits of using AI in software development?" ... This code structure allows you to automate the interaction with multiple AI tools, compare their outputs, and generate actionable insights effectively. Adjust the API calls and logic as needed based on the specific requirements of the APIs you are working with. def main(): prompt = "What are the benefits of using AI in software development?" comparison_df = compare_outputs(prompt) insights = generate_insights(comparison_df)

```
print("Comparison Results:")
print(comparison_df)
print("\nActionable Insights:")
print(insights)

if name == "main": main()
```



Result: The corresponding Prompt is executed successfully
