Generative AI

Trainer: Nilesh Ghule

Introduction to Selenium for Web Scraping

- Selenium is a powerful tool for automating web browsers. It allows you to interact with web pages just like a real user .
- While commonly used for testing web applications, Selenium is also a key tool for web scraping, especially on modern, JavaScript-heavy websites .
- The main advantage of using Selenium is its ability to execute JavaScript, click buttons, fill forms, and wait for elements to load, which is essential for scraping data that isn't available in the initial page HTML .
- The core component that drives the browser is called the Selenium WebDriver .

Initial Setup and Installation

- Install the Selenium package for Python using pip:
  - cmd> pip install selenium
- WebDriver Management (The Good News!): Historically, you had to manually download and manage a browser-specific driver (e.g., ChromeDriver for Chrome).
- Since version 4.6.0, Selenium includes the Selenium Manager, which automatically downloads and manages the correct driver for you . No more manual setup is required for Chrome, Firefox, or Edge!
- You must still have the browser itself (e.g., Google Chrome or Mozilla Firefox) installed on your computer .

Your First Selenium Script: The 8 Basic Components

- Every Selenium script follows a similar pattern. Here are the essential steps :
  - Start the Session: Initialize the WebDriver for your chosen browser (e.g., Chrome).
  - Take Action on Browser: Navigate to a specific URL using `driver.get()`.
  - Request Browser Information: Get data like the page title or current URL.
  - Establish a Waiting Strategy: Crucial for dynamic pages. Instruct Selenium to wait for elements to load before interacting with them (more on this later).
  - Find an Element: Locate a specific element on the page (e.g., a search box) using various strategies like ID, name, or CSS selector.
  - Take Action on Element: Interact with the element (e.g., type text, click).
  - Request Element Information: Extract data from the element (e.g., its text, attributes).
  - End the Session: Close the browser window and end the driver process with `driver.quit()`.

Hands-On: A Complete Working Example

```python
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys

driver = webdriver.Chrome() # 1. Start the Session - Selenium Manager handles driver

driver.get("https://duckduckgo.com") # 2. Take Action on Browser
print("Page title is:", driver.title) # 3. Request Browser Information

driver.implicitly_wait(5)  # 4. Wait strategy: Wait up to 5 seconds for elements to appear

search_box = driver.find_element(By.NAME, "q")  # 5. Find element search box by its name

# 6. Take Action on Element
search_box.clear()  # Clear any pre-filled text
search_box.send_keys("Python for beginners")  # Type the query
search_box.send_keys(Keys.RETURN)  # Press Enter

[redacted]
print("New page title is:", driver.title) # 7. Request Element Information

# Optional: Wait a moment to see the result time.sleep(10)
driver.quit() # 8. End the Session
```

Finding Elements: The Key to Scraping

- Before you can scrape data or click a button, you must locate the corresponding HTML element on the page.
- The `find_element` (for one element) and `find_elements` (for a list) methods are used with locator strategies as follows.
  - `By.ID`: Find by the unique `id` attribute (fastest and most reliable).
    - button = driver.find_element(By.ID, "main-button")
  - `By.NAME`: Find by the `name` attribute (as in our search box example).
  - `By.CLASS_NAME`: Find by the CSS class name.
  - `By.TAG_NAME`: Find by the HTML tag name (e.g., `"a"` for links, `"p"` for paragraphs).
    - all_paragraphs = driver.find_elements(By.TAG_NAME, "p")
  - `By.LINK_TEXT`: Find a link by its exact visible text.
  - `By.CSS_SELECTOR`: Find using CSS selector syntax (very powerful and flexible).
    - user_info = driver.find_element(By.CSS_SELECTOR, "div.user-profile > h2.name")
  - `By.XPATH`: Find using XML Path Language expressions (extremely powerful for complex navigation).
    - cell = driver.find_element(By.XPATH, "//td[text()='Target Value']")

Waiting for Elements: Critical for Reliability

- Web pages load dynamically. Your script might try to find an element before it appears on the page, causing a `NoSuchElementException`.
- Synchronizing your code with the page's state is one of the most important aspects of using Selenium reliably .
- Types of Waits :
  - Implicit Wait: A global timer. Tells the WebDriver to poll the DOM for a certain amount of time (e.g., 10 seconds) when trying to find any element. Set once per session. Simple but not always precise .
  - Explicit Wait: A more intelligent, conditional wait. You tell WebDriver to wait until a specific condition is met (e.g., element is clickable, visible) for a specific element. This is the preferred method for robust scripts.
- The `time.sleep(seconds)` function from Python's standard library is a hard, unconditional wait. Use it sparingly for debugging, as it slows down your script unnecessarily.

Headless Browsing & Scraping Data to a DataFrame

- Headless Mode: Running the browser without a graphical user interface (GUI). This is faster, uses less resources, and is ideal for servers and automated scripts .
- Putting It All Together: A real-world scraping task involves find exact elements and looping over elements to extract the data.
- Enable headless mode for browser

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options

chrome_options = Options() # --- Setup Headless Browser ---
chrome_options.add_argument("--headless")  # Run in background
chrome_options.add_argument("--no-sandbox") # Run browser in normal mode (not isolated-
sandbox mode)
driver = webdriver.Chrome(options=chrome_options)
```

Scraping using Selenium

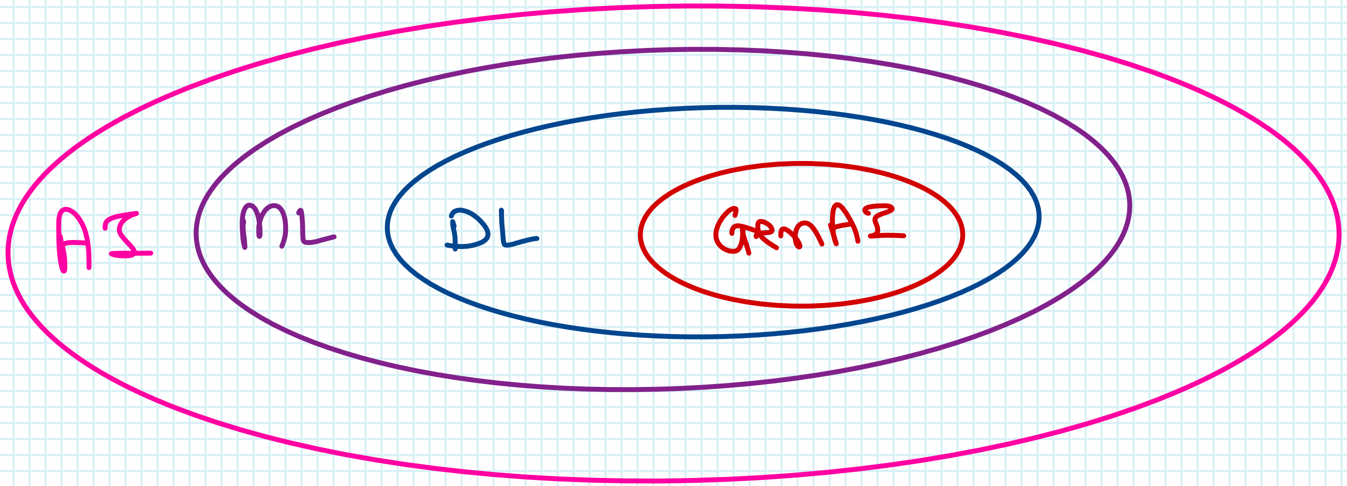
```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.common.by import By

chrome_options = Options() # --- Setup Headless Browser ---
chrome_options.add_argument("--headless")  # Run in background
chrome_options.add_argument("--no-sandbox")
driver = webdriver.Chrome(options=chrome_options)
driver.get(" https://nilesh-g.github.io/learn-web/HTML/demo08.html")

# Assume each item is in an <li> tag with class 'item-name'
item_elements = driver.find_elements(By.TAG_NAME, "li")

# Extract text from each element into a list
scraped_data = []
for item in item_elements:
    scraped_data.append(item.text)  # .text gets the visible text inside the element

print(scraped_data)

driver.quit()
```

Best Practices and Summary

- Summary of Key Concepts:
  - Use Selenium for dynamic websites where content depends on JavaScript.
  - Master `find_element`/`find_elements` with `By` locators.
  - Use explicit waits (`WebDriverWait`) for reliable scripts.
  - Run in headless mode for automation and efficiency.
- Best Practices:
  - Be a Good Net Citizen: Don't overwhelm servers; add delays (`time.sleep`) between requests.
  - Clean Up: Always call `driver.quit()` in a `finally:` block or use a context manager to ensure resources are freed even if your script crashes.
  - Start Simple: Test your selectors and workflow with a visible browser first, then switch to headless.

The AI Landscape - Overview

- Artificial Intelligence (AI): Systems that perform tasks requiring human-like intelligence.
- Machine Learning (ML): Algorithms that learn patterns from data to make predictions.
- Deep Learning (DL): Multi-layer neural networks for complex data such as images and text.
- Generative AI (GenAI): Models that can create new content like text, images, and code.



Nilesh Ghule

Traditional vs Generative AI

- Traditional (Discriminative) AI: Learns to classify or predict labels from existing data.
  - Examples: Email spam detection, fraud detection, image classification.
- Generative AI: Learns data distribution and generates new samples similar to training data.
  - Examples: Chatbots, image generators, code assistants.

What is Generative AI?

- Definition: AI systems that generate new, original content rather than just classifying inputs.
- Core idea: Learn underlying structure of data so the model can sample new examples from it.
- Examples: Generate a new paragraph, design a logo from text, or write a small program.

History of Generative Models

- Early models: n-gram and Markov chain text generators with limited coherence.
- GANs (2014): Generator vs discriminator training loop for realistic image synthesis.
- Transformers (2017): 'Attention Is All You Need' introduced self-attention for sequences.
- Rise of LLMs (2018+): Large Transformer-based models trained on internet-scale text.

Types of Generative Models

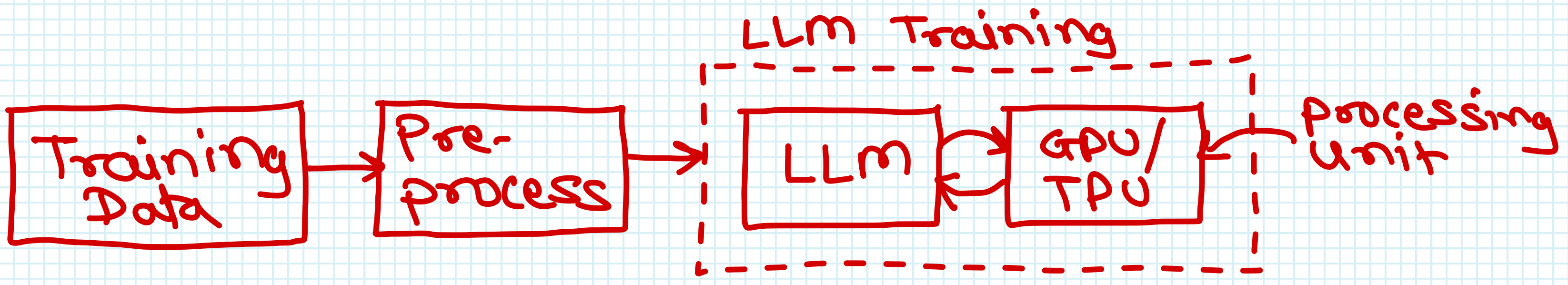
- Text-to-Text (LLMs): Input text, output text for tasks like Q&A, summarization, translation. $\hookrightarrow$ Large Language model.
- Text-to-Image: Input prompt, output image (for example artwork or design).
- Text-to-Code: Input description, output working code snippets or functions.
- Text-to-Music: Input prompt, output audio (e.g. music or speech)
- Text-to-Video: Input prompt, output video

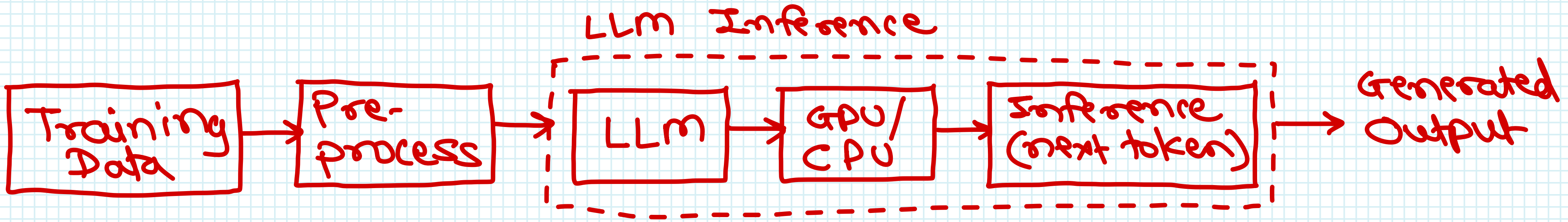
Real-World Applications

- Customer support chatbots that answer FAQs using documents and knowledge bases.
- Content creation for emails, blogs, social media, and marketing copy.
- Developer assistance for code generation, explanation, and refactoring.
- Education, healthcare documentation, legal drafting, and creative arts.

What is a Large Language Model (LLM)?

- Generative model trained to predict next token in text given previous tokens
  - **Heart: Transformer**
- Has millions to billions of parameters learned from large text corpora.
  - **Parameters: weights and biases (neural network)**
- Can perform many tasks without retraining just by changing the prompt.

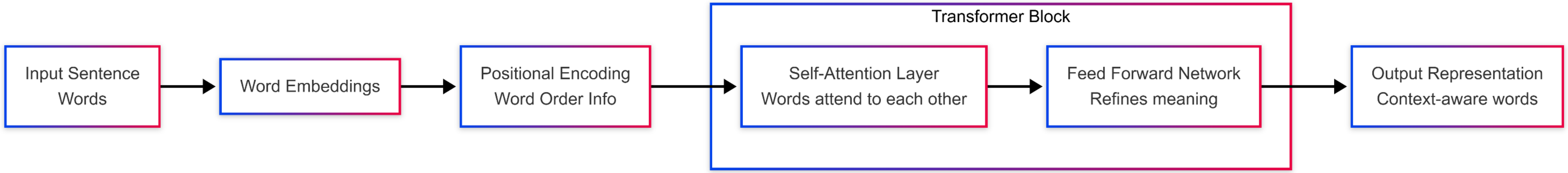




Nilesh Ghule

Transformer Architecture - Intuition

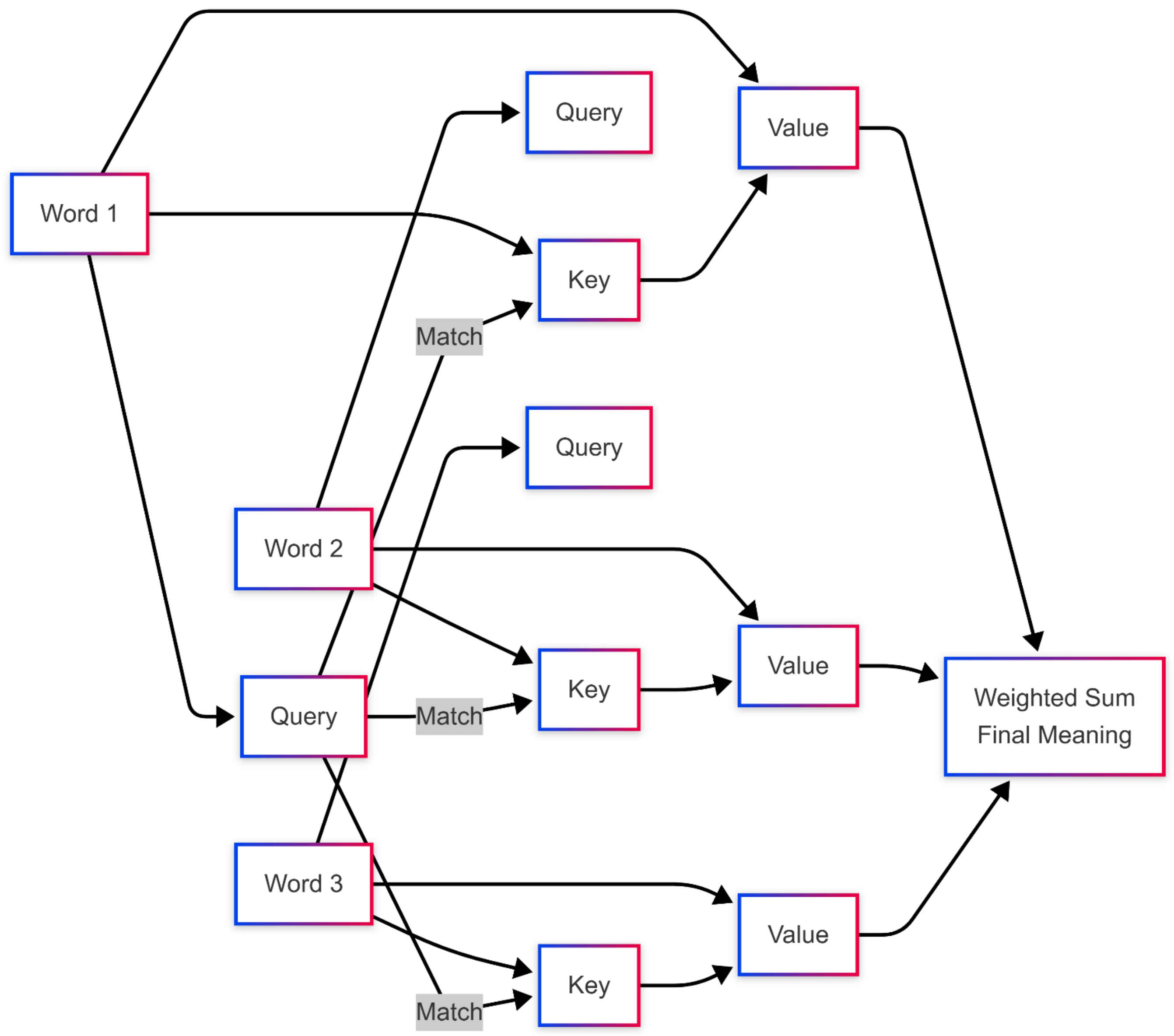
- A Transformer is a neural architecture that understands sequences by letting words talk to each other directly using self-attention.
- Embeddings: Map tokens to numerical vectors capturing meaning.
- Positional encoding: Add information about word order to embeddings.
- Self-attention: Each token attends to all others to gather contextual information.



Nilesh Ghule

Self-Attention Example

- Problem: Words can have different meanings depending on context (for example 'bank').
- Self-attention computes relevance scores between all pairs of tokens.
- Result: Token representations capture meaning conditioned on the entire sentence.



Nilesh Ghule

Demo - Web LLM

- Interact with a state-of-the-art LLM via a web interface (e.g. Gemini, GPT).
- Try prompts
  - Summarization
  - Creative writing
  - Simple code generation.
- Observe fluency, reasoning, and limitations in the responses.

Why Local and Free Tools?

- No paid APIs: everything in this course uses free tiers or open-source tools.
- Local models (via LM Studio) give privacy and control over data.
- APIs and local models will be combined later using LangChain and RAG.

What is an API and How it works with LLM?

- API = Application Programming Interface, a contract that lets software systems talk to each other.
- Python app sends a request to an API and receives a structured response.
- Real-life example: A Restaurant
  - Customer = your Python script (client application).
  - Waiter = API that carries your request and response.
  - Kitchen = LLM provider servers that prepare the result.
- How LLM API works?
  - Client sends an HTTP POST request to an endpoint URL with JSON payload.
  - Server authenticates request using your API key & processes the prompt with an LLM.
  - Server returns a JSON response containing the generated text and metadata.

Why Use LLM APIs?

- Access powerful models like Gemini without owning expensive GPUs or clusters.
- Providers handle scaling, updates, and reliability while you focus on your application.
- Same API patterns work for CLI tools, back-end services, and web/mobile apps.

Free-tier Providers Overview

- Google AI Studio: Gemini models with generous free quotas and strong reasoning.
- Groq: Extremely fast inference on Llama 3 and other open-source models.
- OpenRouter: Aggregator that exposes many models via one unified API.
- Limits:
  - APIs restrict how many requests you can make per minute or per day.
  - Free tiers often have stricter limits, so code should avoid unnecessary loops and retries.

Code: Raw API Call with curl (Groq)

- Example curl command to show the structure of an HTTP POST request.

```
curl -X POST https://api.groq.com/openai/v1/chat/completions \
  -H "Authorization: Bearer YOUR_GROQ_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "model": "llama3-8b-8192",
    "messages": [{"role": "user", "content": "Explain the importance of APIs."}]
  }'
```

Code: Gemini API with Python

```
import os
import json
import requests
from dotenv import load_dotenv

load_dotenv()

api_key = os.getenv("GOOGLE_API_KEY")

url = f"https://generativelanguage.googleapis.com/v1beta/models/gemini-1.5-
flash:generateContent?key={api_key}"
headers = {"Content-Type": "application/json"}

user_prompt = input("Enter your prompt for Gemini: ")

data = { "contents": [{ "parts": [{"text": user_prompt}] }] }

response = requests.post(url, headers=headers, data=json.dumps(data))
print(response.json())
```

Code: Groq Chat Completions with Python

```python
import os
import json
import time
import requests
from dotenv import load_dotenv

load_dotenv()
api_key = os.getenv("GROQ_API_KEY")

url = "https://api.groq.com/openai/v1/chat/completions"
headers = { "Authorization": f"Bearer {api_key}", "Content-Type": "application/json" }

user_prompt = input("Enter your prompt for Groq: ")

data = { "model": "llama3-8b-8192", "messages": [{"role": "user", "content": user_prompt}] }

start = time.time()
response = requests.post(url, headers=headers, data=json.dumps(data))
end = time.time()

print(f"Status: {response.status_code}, Time: {end - start:.2f}s")
print(response.json())
```

Thank You!

Nilesh Ghule

<nilesh@sunbeaminfo.com>