CN7050- Intelligent Systems Module

Student ID: 3059676

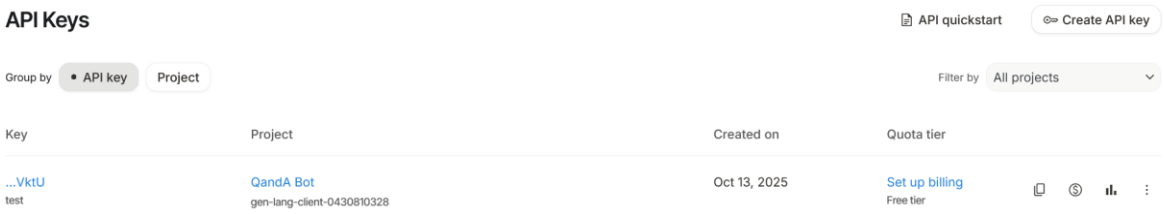
Name: Mann Shrestha

Part I: A Multi Model Interactive Agent with Text and Image Generation capabilities.

Task 1: Account & API key

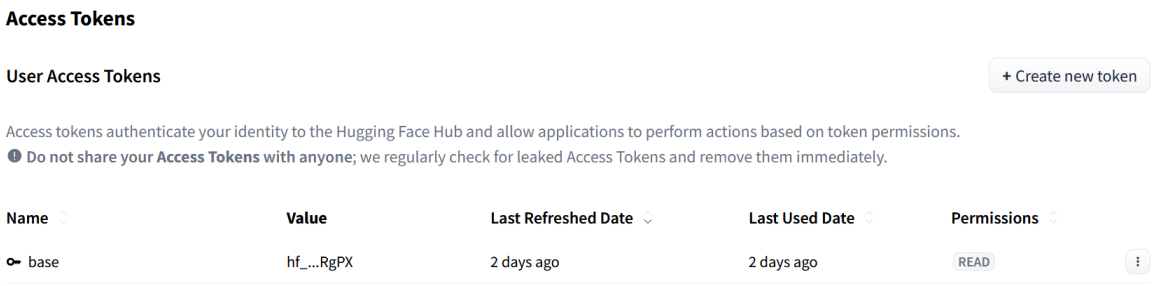
1and 2. Sign in to Google AI Studio (Gemini developer console) and Create an API key (Gemini Developer API)

- I have created a Gemini account and API key from the given instruction in the tutorial 5.



3. Hugging Face API:

- Similarly, I have created a hugging face account to access an API following the instruction in the tutorial 5.



4. Open a new Google Colab notebook

5. Install the Google Gen AI SDK + helper libraries.

```
# installing libraries
!pip install --upgrade google-genai gradio requests pillow
```

- I am using upgradeable version of **google-genai** – provides an interface to integrate Google's generative models into python application.
- **Request:** HTTP library to send HTTP1.1 requests extremely easily. Such as PUT and POST data.

- **Pillow:** The Python Imaging Library adds image processing capabilities to our Python interpreter.

6. Securely provide your API keys in Colab

```
from getpass import getpass
import os

print("Paste keys when prompted:")

GEMINI_KEY = getpass("Gemini API Key: AIzaSyBiYR0ThivMNfBob0HXXbwck050hebVktU")
HF_KEY = getpass("Hugging Face API Key: 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")

os.environ["GEMINI_API_KEY"] = GEMINI_KEY
os.environ["HF_KEY"] = HF_KEY

```

```
Paste keys when prompted:
Gemini API Key: AIzaSyBiYR0ThivMNfBob0HXXbwck050hebVktU·····
Hugging Face API Key: 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·····
```

- We use getpass() module to securely handle the password prompt where programs interact with the users via the terminal.

7. Hugging Face Image Generation function.

```
import requests

def call_hf_image(prompt):
    """
    Hugging Face Stable Diffusion XL example.
    """
    endpoint = "https://api-inference.huggingface.co/models/stabilityai/stable-diffusion-xl-base-1.0"
    headers = {"Authorization": f"Bearer {os.environ['HF_KEY']}"}
    payload = {"inputs": prompt}

    # Make a POST request to a web page, and return the response text:
    response = requests.post(endpoint, headers=headers, json=payload)
    if response.status_code != 200:
        return {"error": response.text}

    image_bytes = response.content
    from PIL import Image
    from io import BytesIO
    img = Image.open(BytesIO(image_bytes))
    return {"pil_image": img}
```

- Create a function to call hugging face API to generate image through prompt.
- I am using hugging face **stable-diffusion-xl-base-1.0 model.** It is developed by Stability AI. It is a diffusion model based on text-to -image generative model that uses two fixed, pretrained text encoders.
  Source: https://huggingface.co/stabilityai/stable-diffusion-xl-base-1.0
- As for the **BytesIO** also called buffered I/O, expects bytes-like objects and produces bytes objects. No encoding, decoding, or newline translation is performed. This category of streams can be used for all kinds of non-text data, and also when manual control over the handling of text data is desired.
  Ref. https://docs.python.org/3/library/io.html

8. Imports + Gemini client initialization.

```python
# imports and Gemini client
import google.generativeai as genai
import requests, json, time, base64, os
from PIL import Image
from io import BytesIO
from IPython.display import HTML, display

import sys
import matplotlib.pyplot as plt

# Initialize Gemini client only if key provided
if os.environ.get('GEMINI_API_KEY'):
    genai.configure(api_key=os.environ['GEMINI_API_KEY'])
    GEMINI_AVAILABLE = True
else:
    GEMINI_AVAILABLE = False
    print("Gemini key not found - text generation disabled until you set GEMINI_KEY.")
```

- Importing necessary libraries and initialize the Gemini client.
- Verify If Gemini key is not found, we will see the error message shown in the last code line.

9. Gemini text generation function.

```python
# Cell 5 - function: generate text with Gemini
def call_gemini(prompt):
    try:
        model = genai.GenerativeModel("gemini-2.5-flash")
        response = model.generate_content(prompt)
        return {"type": "text", "content": response.text}
    except Exception as e:
        return {"type": "text", "content": f"Gemini error: {e}"}
```

- This function to use and generate the text using Gemini LLM model. I had to use Google Gemini Generative model - “gemini-2.5-flash” instead of “gemini-1.5-flash” – this 1.5 version gives me an error i.e. depreciated model. Moreover, 2.5 – flash version model gives me a response in text form, otherwise it will throw an exception.

10. Intent classifier (simple keyword routing)

```python
def classify_intent(user_text):
    t = user_text.lower() # convert in to lower case

    # image triggers
    if "image" in t or "picture" in t or "photo" in t or t.startswith("create me an image"):
        return "image"

    # text/generation/tranform triggers
    if any(k in t for k in ["write", "summarize", "explain", "draft", "describe", "generate text", "summarise"]):
        return "text"

    # fallback: if contains short multimedia instructions mention both -> prefer image
    if "create me" in t and "and" in t and ("video" in t or "image" in t):
        if "image" in t:
            return "image"

    # Default -> text
    return "text"
```

- We created a function **classify_intent()** which takes user text, then convert the text into lowercase. Similarly, this function used to classify image or text or if it contains short multimedia content then return image. Otherwise by default returns text.

11. Orchestrator: route user requests and call right API

```python
def handle_user_request(user_text):
    intent = classify_intent(user_text)
    print(f"[Agent] detected intent: {intent}")

    if intent == "text":
        return {"type": "text", "content": call_gemini(user_text)['content']}
    elif intent == "image":
        return {"type": "image", "content": call_hf_image(user_text)}
    else:
        return {"type": "text", "content": "Sorry, I don't understand."}
```

- handle_user_request() used to handle user text and check for classify intent of the given user text if its intention were text it calls the function call_geming() with this function it uses gemini-2.5-flash model and check for the content either it is for text or image if either not return “Sorry, I don’t understand” text.

12. Display image function

```python
from PIL import Image
from IPython.display import display, HTML
import base64, io, requests, numpy as np

def display_image_response(image_input, filename="/content/generated_image.png"):
    # Extract PIL Image
    if isinstance(image_input, dict):
        if 'pil_image' in image_input:
            img = image_input['pil_image']
        elif 'b64_json' in image_input:
            image_bytes = base64.b64decode(image_input['b64_json'])
            img = Image.open(io.BytesIO(image_bytes))
        elif 'url' in image_input:
            image_bytes = requests.get(image_input['url']).content
            img = Image.open(io.BytesIO(image_bytes))
        else:
            raise TypeError(f"Unsupported dict keys: {image_input.keys()}")
    elif isinstance(image_input, Image.Image):
        img = image_input
    elif isinstance(image_input, str):
        img = Image.open(image_input)
    elif isinstance(image_input, bytes):
        img = Image.open(io.BytesIO(image_input))
    elif isinstance(image_input, np.ndarray):
        img = Image.fromarray(image_input)
    else:
        raise TypeError(f"Unsupported image type: {type(image_input)}")

    # Save image
    img.save(filename)

    # Display thumbnail
    display(img.resize((256, 256)))
    time.sleep(2)
```

- Here, we created a function **display_image_response()** to display with a size of 255x255 and save the image locally.
- In this function, we use base64 - Base64 which is a binary-to-text encoding that uses 64 printable characters to represent each 6-bit segment of a sequence of byte values. As for all binary-to-text encodings, Base64-encoding enables transmitting binary data on a communication channel that only supports text.
- And with the help of base64.b64decode () method, we can decode the binary string into normal form.

13. Function to show the text generated by the Gemini model in the wrapped form.

```
import textwrap

def pretty_print(text, width=80):
    print("\n".join(textwrap.wrap(text, width)))
```

- We use textwrap module for wrapping and formatting of plain text. This module provides formatting of text by adjusting the line breaks in the input paragraph.

14. Code for interactive agent.

```
print("Multimodal Agent (demo). Type 'exit' to quit.")
image_counter = 1 # optional: number images uniquely

while True:
    q = input("You: ")
    if q.strip().lower() in ("exit", "quit"):
        break

    out = handle_user_request(q) # Agent intent

    if out['type'] == 'text': # for text to generate
        print("\n", pretty_print(out['content']))
    elif out['type'] == 'image': # for image to generate
        print("\nAgent (image):")
        # Save with unique filename each time
        filename = f"generated_image_{image_counter}.png"
        display_image_response(out['content'], filename)
        image_counter += 1
    else:
        print("Unknown output type:", out.get('type'))

    print("\n---\n")
```

- This code is used for the demonstration of multi-model Agent which is used for the generation of image or text given by the user input or prompt.

15. Output and result of the code.

- When we run the above code, we will witness a prompt to generate image or text output. If we want to exit the prompt, we can enter exit and press enter. To generate the text, include prompt with keywords like “Image”, “photo” etc. for image generation and “describe”, “summarize” etc., as per step 10.

- **Prompt1:**

```
Multimodal Agent (demo). Type 'exit' to quit.
You: image for a single pink rose.
[Agent] detected intent: image

Agent (image):
```

- **Prompt 2**

```
You: image for multiple vehicles in london
[Agent] detected intent: image

Agent (image):
```

- **Prompt 3**

```
You: write some description about agentic ai.
[Agent] detected intent: text
Agentic AI refers to a class of artificial intelligence systems that exhibit a
high degree of autonomy and goal-oriented behavior. Unlike traditional AI that
primarily reacts to specific prompts or commands, agentic AI proactively
identifies, plans, and executes steps to achieve a predefined objective with
minimal human intervention.  Here are some descriptions highlighting its key
characteristics:  1.  **Goal-Oriented Autonomy:** Agentic AI is designed to
pursue a high-level goal, breaking it down into smaller, manageable sub-tasks.
It doesn't just respond; it *initiates action* to achieve its objective. 2.
**Proactive Planning & Reasoning:** Given a goal, an agentic AI system can
reason about the necessary steps, formulate a multi-step plan, and even generate
alternative strategies if obstacles arise. It anticipates needs and potential
issues. 3.  **Action & Execution:** It's not just about thinking; it's about
doing. Agentic AI can interact with the digital world (e.g., calling APIs,
browsing the web, using software tools, writing code) and potentially the
physical world (via robotics) to execute its plans. 4.  **Self-Correction &
Learning:** These systems often monitor their own progress, evaluate the
outcomes of their actions, and adapt their strategies based on feedback. If a
step fails, they can re-plan or attempt a different approach, demonstrating a
form of iterative learning. 5.  **Tool Use & Integration:** Agentic AI
frequently leverages a wide array of external tools, services, and data sources.
It acts as an orchestrator, intelligently choosing and integrating the right
tools (like a web browser, a code interpreter, a database query, or a text
editor) to accomplish its tasks. 6.  **Persistence & Monitoring:** Once a goal
is set, agentic AI will continue to work towards it, often over extended
periods, monitoring its environment and adjusting its actions until the
objective is met or it determines it cannot proceed. 7.  **Moving Beyond
Reactive AI:** It signifies a shift from AI that merely answers questions or
performs single, isolated tasks to AI that can manage complex, multi-faceted
projects and workflows independently.  **In essence:** Think of Agentic AI not
just as a smart assistant that waits for instructions, but as a digital
colleague that can be given an objective ("Research and summarize the latest
trends in renewable energy, then draft an email to the team with key findings")
and will autonomously plan, execute, monitor, and adapt its way to completing
that complex task. It represents a significant step towards more independent and
capable artificial intelligence.
```

Part II: Learning Objectives

Task 1: Account & API key

1. **Sign in to Google AI Studio (Gemini developer console).**
2. **Create an API key (Gemini Developer API):**
3. **Open a new Google Colab notebook.**
4. **Install the Google Gen AI SDK + helper libraries.**

```
1 !pip install --upgrade google-genai arxiv beautifulsoup4 requests readability-lxml
2
```

- **arXiv** is a free distribution service and an open-access archive for nearly 2.4 million scholarly articles in the fields of physics, mathematics, computer science, quantitative biology, quantitative finance, statistics, electrical engineering and systems science, and economics. Materials on this site are not peer-reviewed by arXiv. https://arxiv.org/

- **Beautiful Soup** is a library that makes it easy to scrape information from web pages. It sits atop an HTML or XML parser, providing Pythonic idioms for iterating, searching, and modifying the parse tree.
  https://pypi.org/project/beautifulsoup4/
- **readability-lxml:** Given an HTML document, extract and clean up the main body text and title.

5. Securely provide your API key in Colab

```
from getpass import getpass
import os

api_key = getpass("Paste your Google Gemini API key here: AIzaSyBiYR0ThivMNfBob0HXXbwck050hebVktU")
os.environ["GEMINI_API_KEY"] = api_key # optional - keep in memory only

```

```
Paste your Google Gemini API key here: AIzaSyBiYR0ThivMNfBob0HXXbwck050hebVktU·····
```

6. Initialize the Gen AI client in Colab.

- To initialize the Gen AI client, use the following code

```
from google import genai

# create client using the key we provided
client = genai.Client(api_key=os.environ["GEMINI_API_KEY"])

```

7. Try a small generate call to confirm connectivity.

```
resp = client.models.generate_content(
    model="gemini-2.5-flash",
    contents="Summarize in one sentence: Why is reproducibility important in research?"
)
print(resp.text)
```

- I will be using “gemini-2.5-flash” model.

8. Search arXiv for scholarly papers.

```
import arxiv

def search_arxiv(query, max_results=5):
    search = arxiv.Search(
        query=query,
        max_results=max_results,
        sort_by=arxiv.SortCriterion.Relevance
    )

    results = []
    for result in search.results():
        results.append({
            "title": result.title,
            "summary": result.summary,
            "authors": [a.name for a in result.authors],
            "pdf_url": result.pdf_url,
            "id": result.get_short_id()
        })
    return results

# quick test
papers = search_arxiv("multimodal transformers", max_results=4)
for p in papers:
    print(p['title'])
```

- **arXiv:** The goal of the API is to allow application developers access to all the arXiv data, search and linking facilities with an easy-to-use programmatic interface. This page provides links to developer documentation and gives instructions for how to join the mailing list and contact other developers and maintainers.

9. Grab web-page content for non-paper websites:

- Use requests + BeautifulSoup to extract main paragraphs (simple heuristic)

```python
import requests
from bs4 import BeautifulSoup

def fetch_page_text(url, max_chars=4000):
    try:
        r = requests.get(url, timeout=10, headers={"User-Agent":"research-agent/1.0"})
        r.raise_for_status()
    except Exception as e:
        return ""

    soup = BeautifulSoup(r.text, "html.parser")
    # simple extraction: join visible <p> text
    paragraphs = [p.get_text(separator=" ", strip=True) for p in soup.find_all("p")]
    content = "\n\n".join(paragraphs)
    return content[:max_chars] # limit length for API
```

- **BeautifulSoup:** https://beautiful-soup-4.readthedocs.io/en/latest/
- **Prepared Requests:** Whenever we receive a Response object from an API call or a Session call, the request attribute is actually the PreparedRequest that was used. In some cases, we may wish to do some extra work to the body or headers (or anything else really) before sending a request.
- We created a fetch_page_text() function with a url and max_char = 4000 as a parameter.

10. Create a safe chunking helper (Gemini has big context but chunking helps reliability):

```python
def chunk_text(text, max_chars=3000):
    chunks = []
    start = 0
    while start < len(text):
        end = min(len(text), start + max_chars)
        chunks.append(text[start:end])
        start = end
    return chunks
```

- We created a function chunk_text () to divide a text with maximum characters of 3000 at a time.

11. Create a function that asks Gemini to summarize a text chunk.

```
SYSTEM_PROMPT = (
    "You are a concise research assistant. For each text provided, return: "
    "1) a one-paragraph summary (3-5 sentences), 2) three bullet key contributions/findings,"
    "and 3) a suggested short title. Be factual and include no hallucinated facts."
)

def summarize_chunk(chunk_text):
    prompt = SYSTEM_PROMPT + "\n\nText to summarize:\n" + chunk_text
    resp = client.models.generate_content(
        model="gemini-2.5-flash", # adjust if unavailable
        contents=prompt
    )
    return resp.text
```

- We keep a short system prompt at the top to set the assistant behaviour.

12. Summarize a long document by summarizing chunks + combining.

```
def summarize_text_long(text):
    chunks = chunk_text(text, max_chars=3000)
    summaries = []
    for c in chunks:
        s = summarize_chunk(c)
        summaries.append(s)

    # Optionally aggregate the chunk summaries into a single final summary:
    if len(summaries) == 1:
        return summaries[0]
    else:
        combined = "\n\n".join(summaries)
        final_prompt = SYSTEM_PROMPT + "\n\nCombine the following chunk summaries into a single concise summary:\n\n" + combined
        final_resp = client.models.generate_content(model="gemini-2.5-flash", contents=final_prompt)
        return final_resp.text

```

- We create summarize_text_long() to summarize long text from the text and chunk_text with max_character= 3000.

13. Putting it together — search and summarize top N arXiv abstracts:

```
query = "multimodal transformers"
# query = "vision transformer"
papers = search_arxiv(query, max_results=5)

for p in papers:
    print("\n---")
    print("Title:", p['title'])
    text_to_summarize = p['summary'] # arXiv abstract (short)
    summary = summarize_text_long(text_to_summarize)
    print("Summary:\n", summary)
    print("PDF:", p['pdf_url'])
```

- Write a manual query for any topic. In our case, “multimodal transformer” – this journal will be search through search_arxiv() with maximum result of 5.

Journal 1:

Title: Multimodal Learning with Transformers: A Survey
Summary:
1) This paper offers a comprehensive survey of Transformer techniques specifically tailored for multimodal data, a burgeoning area in AI research. It begins by establishing a background of multimod

2) Key Contributions:
* Provides a comprehensive survey of Transformer techniques applied to multimodal data.
* Offers a theoretical review of various Transformer architectures from a geometrically topological perspective.
* Discusses applications, common challenges, and future research directions for multimodal Transformers.

3) A Survey of Multimodal Transformer Techniques
PDF: http://arxiv.org/pdf/2206.06488v2

Journal 2:

Title: MANGO: Multimodal Attention-based Normalizing Flow Approach to Fusion Learning
Summary:
1) **Summary:**
Current multimodal fusion methods, relying on implicit Transformer attention, often fail to capture essential features and complex correlations within multimodal inputs. To address this limitation,

2) **Key Contributions/Findings:**
* Introduces MANGO (Multimodal Attention-based Normalizing Flow) for explicit, interpretable, and tractable multimodal fusion learning.
* Proposes an Invertible Cross-Attention (ICA) layer, incorporating three new mechanisms (MMCA, IMCA, LICA), to capture complex multimodal correlations.
* Achieves state-of-the-art performance on diverse multimodal tasks, including semantic segmentation, image-to-image translation, and movie genre classification.

3) **Suggested Short Title:**
MANGO: Explicit Multimodal Fusion via Invertible Cross-Attention
PDF: http://arxiv.org/pdf/2508.10133v1

Journal 3:

Title: Multimodal Transformer With a Low-Computational-Cost Guarantee
Summary:
1) Summary:
Transformer-based models have advanced multimodal understanding tasks like visual question answering, but their multi-head attention mechanism suffers from quadratic complexity, particularly with in

2) Key Contributions/Findings:
* Addresses the quadratic computational complexity of multi-head attention in multimodal Transformers, which limits scalability.
* Introduces LoCoMT, a novel multimodal attention mechanism that reduces computational cost by assigning different attention patterns to individual heads.
* Achieves significant GFLOPs reduction and comparable or superior performance to established models on Audioset and MedVidCL datasets.

3) Suggested Short Title:
LoCoMT: Low-Cost Multimodal Transformers
PDF: http://arxiv.org/pdf/2402.15096v1

Journal 4

Title: Brain encoding models based on multimodal transformers can transfer across language and vision
Summary:
1) This study investigates how the human brain represents concepts across language and vision using encoding models trained with representations from multimodal transformers. While traditional enco

2) Key Contributions/Findings:
* Encoding models trained with multimodal transformer representations can predict fMRI responses across language and vision modalities (stories and movies).
* This cross-modal prediction is prominent in cortical regions that represent conceptual meaning and reveals shared semantic dimensions in language and vision.
* Multimodal transformers learn more aligned concept representations across language and vision than unimodal transformers, aiding understanding of brain's multimodal processing.

3) Suggested Short Title: Multimodal Transformers for Cross-Modal Brain Encoding
PDF: http://arxiv.org/pdf/2305.12248v1