Codebase to Tutorial Open Source ★ 11.1K

Post-creation

Siddharthakhandelwal/Post-creation

english

gemini-2.5-flash

c3b5daa

Aug 3, 2025

Chapters

Show Raw Markdown

Tutorial: Post-creation

This project acts as a **smart content creation assistant** for social media. It *automatically collects trending information* like news and hashtags from the internet. Then, it uses **advanced Artificial Intelligence (AI) models** to generate engaging and relevant posts tailored for doctors or the general public. Crucially, it *securely manages all the secret access keys* needed to connect with these AI services.

Visual Overview

```
flowchart TD
    A0["Data Source Scrapers
"]
```

```
    A1["Large Language Model (LLM) Connectors
"]
    A2["Prompt Builder
"]
    A3["Main Post Generation Workflow
"]
    A4["API Key Management
"]
    A3 -- "Gathers data from" --> A0
    A3 -- "Provides data to" --> A2
    A2 -- "Constructs prompts for" --> A1
    A3 -- "Calls" --> A1
    A1 -- "Retrieves keys from" --> A4
```

Chapters

Generated by AI Codebase Knowledge Builder.

Post-creation

Siddharthakhandelwal/Post-creation

english

gemini-2.5-flash

c3b5daa

Aug 3, 2025

Chapters

Show Raw Markdown

Chapter 1: API Key Management

Welcome to the `Post-creation` project! In this first chapter, we're going to talk about something super important for any program that connects to online services: **API Key Management**. Think of it like handling your secret passwords in a very safe way.

Why is this important? (The Secure Vault Analogy)

Imagine you have a super-secret diary that contains all your private thoughts. Would you write your bank password or your house key code directly on the first page, visible to everyone? Probably not!

In the world of programming, connecting to online services (like AI models such as Perplexity, or Groq) often requires a special "password" called an **API Key**. This key t service, "Hey, I'm allowed to use your features!" If your API key falls into the wrong hands, someone else could use your access, potentially costing you money or misusing your account.

The problem is: how do we use these keys in our code without accidentally showing them to the world, especially if we share our code online? This is where **API Key Management** comes in. It acts as a "secure vault" for these sensitive keys. Instead of writing them directly into our code (which is like writing your password on the diary's first page), we store them in a special, hidden file. Our program then knows how to safely open this vault and get the key when it needs it, without exposing it to others.

Let's look at a concrete use case: Our project needs to talk to AI models like Google Gemini, Perplexity, and Groq to generate social media posts. Each of these models requires its own API key. We need a way to use these keys securely.

What is an API Key?

An **API Key** is like a unique identification number or a secret token that software programs use to identify themselves when talking to an online service. When your program sends a request to, say, Google Gemini, it also sends your Gemini API key. This key helps Google know it's *your* program making the request and that you have permission.

Why Not Put Keys Directly in Code?

Imagine if your `agent.py` file looked like this:

```python
# DON'T DO THIS! This is BAD practice!
GEMINI_API_KEY = "AIzaSyBkI50xIaE0zhL-S-61RexARSkhIHxPBRU" # Your actual key here!

def post_gemini(prompt):
    # ... use GEMINI_API_KEY directly ...

```

If you ever share this code online (like on GitHub), your API key would be visible to everyone! Anyone could then copy it and use your access to the Gemini service. This is a huge security risk.

The Solution: The `.env` File and `python-dotenv`

To keep our keys secret, we use two main things:

1. **The `.env` file:** This is a simple text file (named `.env`) that lives in the same fol
Python project. It's like our "secure vault". Inside, we store our API keys (and other sensitive

information) in a simple `KEY_NAME=VALUE` format. For example:

```
 GEMINI_API_KEY=your_actual_gemini_key_here
Perplexity=your_actual_perplexity_key_here
GROQ_API=your_actual_groq_key_here
```

Crucially, we tell tools like Git (which programmers use to share code) to **ignore** this `.env` file, so it never accidentally gets uploaded online!

2. **The `python-dotenv` library:** This is a Python tool that acts like the "key master" for our vault. When our program starts, we tell `python-dotenv` to read the `.env` file. It then takes all the keys from that file and loads them into a special place in our computer's memory called "environment variables." Our Python code can then safely ask for these keys by name.

How to Use API Key Management in Our Project

Let's see how our project uses this system.

First, you need to make sure the `python-dotenv` library is installed. This is handled by our `requirements.txt` file.

```
 # From requirements.txt
requests
beautifulsoup4
python-dotenv  # This is the "key master" library!
google-generativeai
google-genai
```

You'll install all these using `pip install -r requirements.txt`.

Next, you need to create the `.env` file yourself in the main folder of your project (where `agent.py` is located). Fill it with your actual API keys:

```
 # In your .env file (DO NOT share this file!)
GEMINI_API_KEY=AIzaSyBkI50xIaE0zhL-S-61RexARSkhIHxPBRU
Perplexity=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
GROQ_API=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
```

Important: Replace the example values with your *actual* API keys obtained from Google AI Studio, Perplexity, and Groq. These are just examples and won't work.

Now, let's look at `agent.py` to see how it "opens the vault" and gets the keys:

```python
# From agent.py
import os
from dotenv import load_dotenv # Import the "key master" tool

load_dotenv() # Tell the "key master" to open the .env vault!

def post_gemini(prompt):
    try:
        # Ask the operating system for the key named "GEMINI_API_KEY"
        api_key = os.getenv("GEMINI_API_KEY")
        # ... rest of the code that uses api_key ...

```

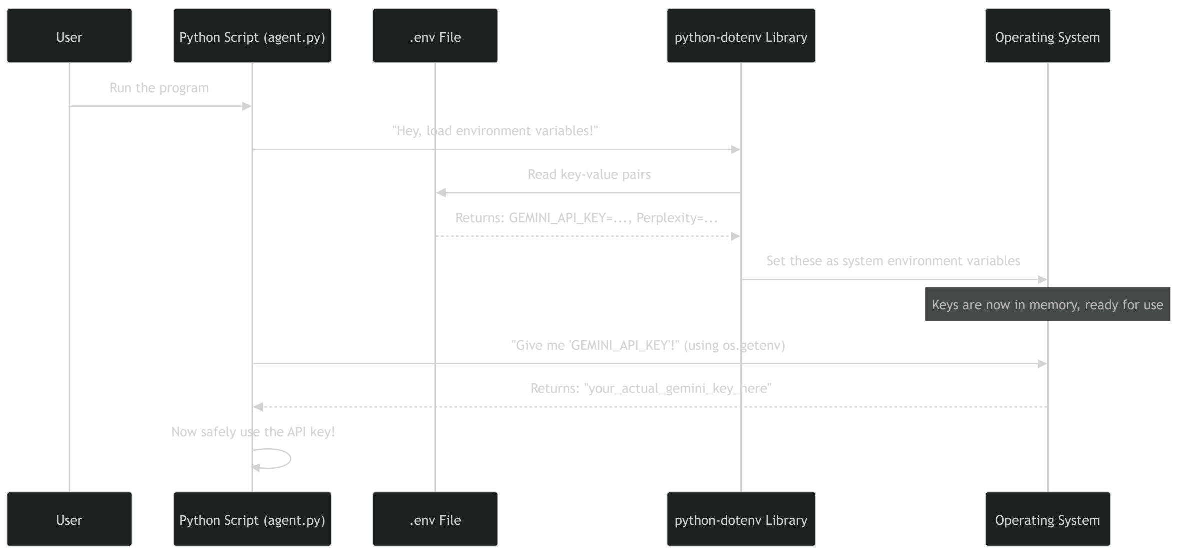
Explanation:

1. `from dotenv import load_dotenv` : This line imports the necessary function from the `python-dotenv` library.
2. `load_dotenv()` : This important line tells the program, "Go find my `.env` file and load all the key-value pairs from it into my computer's environment variables." This happens once, usually at the very beginning of your main script.
3. `api_key = os.getenv("GEMINI_API_KEY")` : After `load_dotenv()` has done its job, you can use `os.getenv()` . This is a standard Python way to ask your computer's operating system, "Hey, do you have an environment variable named `GEMINI_API_KEY` ? If so, please give me its value." The operating system then securely provides the key's value, which is stored in the `api_key` variable.

This way, your actual secret keys are never written directly in the Python code itself. They stay hidden in the `.env` file, and your program only accesses them temporarily in memory when needed.

Under the Hood: How it Works Step-by-Step

Let's visualize the process:


User
Python Script (agent.py)
.env File
python-dotenv Library
Operating System
Run the program
"Hey, load environment variables!"
Read key-value pairs
Returns: GEMINI_API_KEY=..., Perplexity=...
Set these as system environment variables
Keys are now in memory, ready for use
"Give me 'GEMINI_API_KEY'!" (using os.getenv)
Returns: "your_actual_gemini_key_here"
Now safely use the API key!

This sequence shows that your Python script doesn't directly touch the `.env` file *after* `load_dotenv()` has done its job. Instead, it asks the Operating System for the keys, which were previously loaded by `python-dotenv`. This keeps things secure and organized.

Conclusion

In this chapter, we've learned the crucial concept of **API Key Management**. We understood why it's important to keep sensitive information like API keys out of our main code files to prevent security risks. We also learned how to use a `.env` file as a "secure vault" and the `python-dotenv` library as our "key master" to safely load and access these keys when our program needs them. This sets a strong foundation for building secure and maintainable applications.

Next, we'll dive into how our project gathers the information needed to generate posts. Let's explore the Data Source Scrapers!

Generated by AI Codebase Knowledge Builder. **References**: [1], [2], [3], [4], [5]

Post-creation

Siddharthakhandelwal/Post-creation

english

gemini-2.5-flash

c3b5daa

Aug 3, 2025

Chapters

Show Raw Markdown

Chapter 2: Data Source Scrapers

Welcome back to the `Post-creation` project! In our previous chapter, we learned how to securely handle secret keys (API keys) so our program can safely talk to online services like AI models. Now, we need to get the "ingredients" for our social media posts: fresh, trending information!

Why Do We Need "Data Source Scrapers"? (Our Web Reporters)

Imagine you're a chef, and you need fresh vegetables for your delicious meal. You w t guess what's in season or rely on old ingredients, right? You'd go to the market ar freshest produce.

Our `Post-creation` project is like that chef. To create interesting and timely social media posts for doctors, it needs *fresh* information from the internet. This includes things like:

- **Trending Hashtags:** What are people talking about right now? (`#health` , `#wellness` , `#doctor` , etc.)
- **Latest News Headlines:** What are the most important health stories happening today?

We can't just type these in manually every time. We need a way for our program to *automatically* go to the "market" (the internet) and "collect" this information.

This is where **Data Source Scrapers** come in! Think of them as our dedicated "web reporters." Their job is to visit specific public websites and carefully gather trending information. If they ever run into trouble getting the latest news, they have a backup list of common topics ready, so our project always has something to work with.

Our main goal in this chapter is to understand how our project automatically gathers this information, just like a news reporter collects facts for a story.

What is Web Scraping? (The Smart Reader)

At its core, **web scraping** is a technique where a computer program "reads" the content of a website, much like you read a newspaper, but it's much faster and specifically looks for certain pieces of information.

Imagine a website is a big book filled with text and pictures. When you visit a website with your web browser (like Chrome or Firefox), your browser downloads this "book" (the website's code, usually HTML) and then shows you the pretty version.

A web scraper does something similar, but instead of showing it to you, it automatically scans the "book" (the HTML code) for specific patterns or words to pull out *only* the information it needs, like all the headlines or all the hashtags.

For our project, we use two main tools for web scraping:

1. **`requests`** : This is like the postal service for our program. It helps our program "visit" a website by sending a request to the website's server and getting back the website's raw content (the "book" or HTML code).
2. **`BeautifulSoup`** : Once we have the raw website content, `BeautifulSoup` steps in. It's like a super-smart indexer for our "book." It helps us easily navigate through the website's content and find specific pieces of information, like all the headlines, paragraphs, or links, even i[illegible]k" is very long and complex.

How Our Project Uses Data Source Scrapers

Our project has a special file called `data_sources.py`. This file is where all our "web reporters" (the scraper functions) live. When our main `agent.py` script needs fresh data, it simply asks `data_sources.py` to do the work.

Let's look at how `agent.py` asks for trending news and hashtags:

```
# From agent.py

from data_sources import (
    scrape_trending_hashtags,
    scrape_trending_news,
    # ... other imports ...
)

# ... (API key loading and other functions) ...

if __name__ == "__main__":
    print("Scraping trending hashtags...")
    hashtags = scrape_trending_hashtags()
    print("Scraping trending news...")
    news = scrape_trending_news()

    print("--- Collected Data ---")
    print(f"Hashtags: {hashtags}")
    print(f"News Headlines: {news}")

```

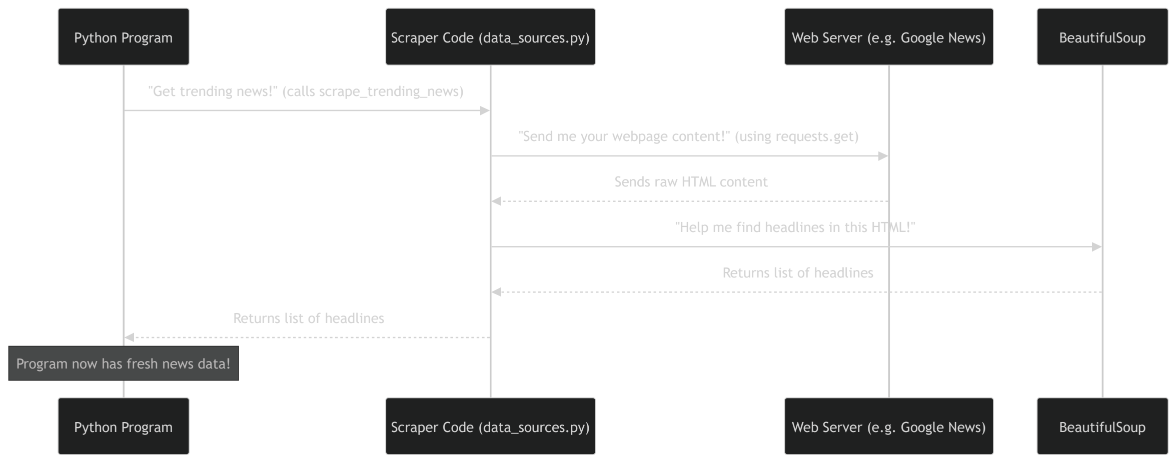
Explanation:

1. `from data_sources import ...` : This line tells our `agent.py` script, "Hey, I need to use some functions from the `data_sources.py` file, specifically `scrape_trending_hashtags` and `scrape_trending_news`."
2. `hashtags = scrape_trending_hashtags()` : This line calls the "web reporter" function that goes out and finds trending hashtags. It then stores the list of found hashtags in a variable called `hashtags`.
3. `news = scrape_trending_news()` : Similarly, this calls the function that finds trending news headlines and stores them in the `news` variable.

When you run this part of the code, you'll see messages like "Scraping trending ha then the actual lists of hashtags and news headlines that were collected from the i

Under the Hood: How a Scraper Works Step-by-Step

Let's peek behind the curtain and see what happens when a scraper function like `scrape_trending_news()` is called.


Python Program
Scraper Code (data_sources.py)
Web Server (e.g. Google News)
BeautifulSoup
"Get trending news!" (calls scrape_trending_news)
"Send me your webpage content!" (using requests.get)
Sends raw HTML content
"Help me find headlines in this HTML!"
Returns list of headlines
Returns list of headlines
Program now has fresh news data!
Python Program
Scraper Code (data_sources.py)
Web Server (e.g. Google News)
BeautifulSoup

Explanation of the steps:

1. **Request the Page:** The scraper code uses `requests` to tell the website's server (like Google News), "Please send me the content of your page!"
2. **Get the HTML:** The web server responds by sending back the raw HTML code of the webpage. This HTML is like the blueprint or source code of the website.
3. **Parse with BeautifulSoup:** The scraper then hands this raw HTML over to `BeautifulSoup`. `BeautifulSoup` carefully reads through the HTML code, understanding its structure.
4. **Extract Information:** Using instructions from our scraper code (like "find all elements that look like headlines"), `BeautifulSoup` extracts just the text we need.
5. **Return Data:** Finally, the scraper function returns the collected information (e.g., a list of news headlines) back to the main program.

Diving Deeper: The `data_sources.py` Code

Let's look at simplified versions of the functions inside `data_sources.py` to see how `requests` and `BeautifulSoup` are used.

First, the necessary tools are imported:

```python
# From data_sources.py
import requests
```

```
from bs4 import BeautifulSoup # Imports the BeautifulSoup tool

```

Explanation: We import `requests` to grab the web page and `BeautifulSoup` (often shortened to `bs4`) to help us parse it.

Now, let's look at a simplified `scrape_trending_hashtags` function:

```
# From data_sources.py (simplified)
def scrape_trending_hashtags():
    try:
        url = 'https://best-hashtags.com/hashtag/health/'
        resp = requests.get(url, timeout=5) # 1. Get the webpage content
        soup = BeautifulSoup(resp.text, 'html.parser') # 2. Prepare for reading

        hashtags = []
        # Find a specific part of the page where hashtags are
        tag_block = soup.find('div', {'class': 'tag-box'})
        if tag_block:
            # Look for words starting with '#' inside that part
            hashtags = [tag.strip() for tag in tag_block.text.split() if tag.star

        return hashtags[:10] # Return top 10
    except Exception:
        # If anything goes wrong, return a backup list
        return ['#health', '#wellness', '#doctor', '#fitness', '#mentalhealth']

```

Explanation:

1. `url = '...'` : This is the address of the website our "reporter" will visit.
2. `resp = requests.get(url, timeout=5)` : This line uses `requests` to go to the `url` and fetch its content. `timeout=5` means it will wait at most 5 seconds for a response.
3. `soup = BeautifulSoup(resp.text, 'html.parser')` : We give `BeautifulSoup` the raw text from the website (`resp.text`) and tell it to read it as HTML. Now, `soup` is like our "smart reader" that can help us find things.
4. `tag_block = soup.find(...)` : We tell `BeautifulSoup` to `find` a specific se webpage. Websites are organized into sections, and we often look for sections wi "class" names or "IDs" to pinpoint where our data lives.

5. `hashtags = [tag.strip() for tag in tag_block.text.split() if tag.startswith('#')]` : Once we have the `tag_block` (the specific section), we extract all its text (`.text`), split it into individual words, and then filter to keep only those words that start with a `#` .
6. `return hashtags[:10]` : We return the first 10 hashtags we found.
7. `except Exception:` : This is a crucial part! Websites can change or be offline. If *anything* goes wrong (like the website not responding or its layout changing), our program won't crash. Instead, it will use a pre-defined **fallback list** of common hashtags. This ensures our project always has data, even if scraping fails!

The `scrape_trending_news` function works very similarly:

```python
# From data_sources.py (simplified)
def scrape_trending_news():
    try:
        url = 'https://news.google.com/topics/CAAqJggKIiBDQkFTRWdvSUwyMHZNRFZ4Y0dNU
        resp = requests.get(url, timeout=5)
        soup = BeautifulSoup(resp.text, 'html.parser')

        # Find all 'a' (link) tags with a specific class, which are usually headlin
        headlines = [a.text for a in soup.find_all('a', {'class': 'DY5T1d'})]

        return headlines[:10]
    except Exception:
        # If scraping fails, provide backup news headlines
        return ["Global health update: Stay safe!", "New research on heart health r

```

Explanation:

- This function targets a Google News URL.
- `headlines = [a.text for a in soup.find_all('a', {'class': 'DY5T1d'})]` : Instead of looking for hashtags, it now looks for all `<a>` (link) tags that have a specific class (`'DY5T1d'`). On many news websites, headlines are often links with unique class names. We then extract the visible text (`.text`) from these links.
- Just like with hashtags, if anything fails, a helpful backup list of generic news headli[illegible]s used.

These "web reporters" are super helpful because they allow our project to be dyna[illegible] generate posts based on what's currently happening, not just old, static information!

Conclusion

In this chapter, we explored **Data Source Scrapers**, our project's "web reporters." We learned how they use tools like `requests` to visit websites and `BeautifulSoup` to "read" and extract specific information like trending hashtags and news headlines. We also saw the importance of having fallback data, so our project can always work, even if a website isn't available.

Having this fresh, real-world data is the first big step in creating relevant social media posts. Next, we'll see how this raw data is transformed into a clear set of instructions for our AI models using the Prompt Builder!

Generated by AI Codebase Knowledge Builder. **References**: [1], [2], [3]

Post-creation

Siddharthakhandelwal/Post-creation

english

gemini-2.5-flash

c3b5daa

Aug 3, 2025

Chapters

Show Raw Markdown

Chapter 3: Prompt Builder

Welcome back! In our previous chapter, we became master chefs of data, learning how our project uses **Data Source Scrapers** to gather fresh "ingredients" like trending news and hashtags from the internet. Now, we have all this raw, valuable information. But how do we tell our super-smart AI what to *do* with it?

Why Do We Need a "Prompt Builder"? (The AI's Scriptwriter)

Imagine you have a brilliant artist. You give them a pile of newspapers and a list of popular topics. They're ready to create, but they need clear instructions: "Draw a happy scen n the news about animals, and make sure to include the color blue." Without these p directions, the artist might draw anything!

Our project's AI models (which we'll meet in the next chapter!) are like that brilliant artist. They're powerful, but they need very specific instructions to create the social media posts we want. They don't just "know" to combine trending news with a doctor's audience, use certain keywords, and stay within a word limit.

This is where the **Prompt Builder** comes in! Think of it as the AI's "scriptwriter." Its job is to meticulously craft the exact instructions, called **"prompts,"** that are sent to the AI. It takes all the gathered information – like scraped news, trending hashtags, desired keywords, and even the required word limit – and weaves them into a clear, concise set of directions.

Our main goal in this chapter is to understand how our project creates these precise "scripts" for the AI, ensuring it understands exactly what kind of social media post to create, tailored for the intended audience (doctors) and content.

What is a "Prompt"?

At its simplest, a **Prompt** is just the text message or question you send to an AI to get it to do something. It's like sending a text message to a friend asking them to pick up milk, bread, and eggs – those are your "prompt" for the friend.

For AI, a good prompt acts like a detailed recipe. It tells the AI:

- **The Context:** "Here's some news about health..."
- **The Task:** "Create a social media post..."
- **The Style/Tone:** "Make it friendly, for a doctor's audience, use emojis..."
- **The Constraints:** "Keep it under 100 words, include '#health', use the word 'wellness' five times..."

Why Do We Need a "Builder" for Prompts?

If we always wanted the exact same social media post, we could just write one static prompt. But our project is dynamic! It needs to:

1. **Use Fresh Data:** The news and hashtags change daily, so the prompt must include the *latest* scraped information.
2. **Respond to User Choices:** You, the user, can decide the word limit, specific keywords, or which AI model to use. The prompt needs to reflect these choices.

A **Prompt Builder** isn't a separate tool you install. It's a *process* within our `agent.py` that intelligently combines all these different pieces of information into a single, cohesi for the AI.

How Our Project Builds a Prompt

Let's look at how our `agent.py` script acts as the "Prompt Builder." After the Data Source Scrapers have done their job and you've provided your preferences (like keyword and word limit), `agent.py` combines everything into a string of text.

Here's a simplified look at how `agent.py` prepares the data for the prompt:

```
# From agent.py (simplified)

# ... (Scraping happens, user inputs are gathered) ...

# 1. Prepare scraped data for the prompt
hashtags_str = ', '.join(hashtags) # Turns a list of hashtags into a single string
news_str = '\n'.join(news)         # Turns a list of news headlines into a single s

# 2. Gather user inputs
word_limit = int(input("Enter the word limit: ")) # Example: 50
keyword = input("Enter the Key word you want: ")  # Example: "wellness"
keyword_count = int(input("Keyword count: "))     # Example: 3

# Now, the Prompt Builder creates the message!
my_social_media_prompt = f'''
Create a post for a doctor's audience.
Based on this news: {news_str}
Use these hashtags: {hashtags_str}
Include the keyword "{keyword}" {keyword_count} times.
Keep the word limit to {word_limit} words max.
'''

print(my_social_media_prompt) # This is the final prompt sent to the AI!

```

Explanation:

1. **`hashtags_str = ', '.join(hashtags)`**: The `scrape_trending_hashtags()` function returns a Python list (e.g., `['#health', '#doctor', '#wellness']`). We use `', '.join()` to convert this list into a single, neat string like `"#health, #doctor, #wellness"`, which is easier for the AI to read in the prompt.
2. **`news_str = '\n'.join(news)`**: Similarly, the `scrape_trending_news()` func list of news headlines. We join them with a newline character (`\n`) so each headli a new line in the prompt, making it easy for the AI to distinguish them.

3. **User Inputs:** Lines like `word_limit = int(input(...))` gather specific preferences directly from you when you run the program.
4. **`my_social_media_prompt = f'''...''':`** This is the core of the **Prompt Builder**. It uses an f-string (notice the `f` before the triple quotes `'''`). F-strings are a super convenient way in Python to embed variables directly into a string. Any variable name inside curly braces `{}` will be replaced with its actual value.

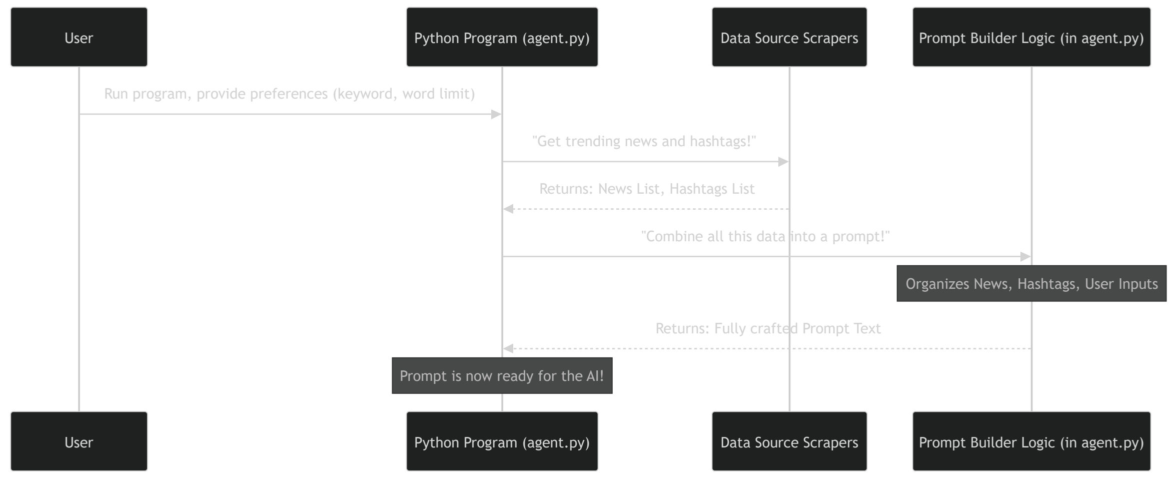
When you run the code, `my_social_media_prompt` will look something like this (if `news` was `["New vaccine research", "Flu season update"]`, `hashtags` was `["#health", "#vaccine"]`, `keyword` was "care", `keyword_count` was 2, and `word_limit` was 50):

```
 Create a post for a doctor's audience.
Based on this news: New vaccine research
Flu season update
Use these hashtags: #health, #vaccine
Include the keyword "care" 2 times.
Keep the word limit to 50 words max.
```

This neatly formatted text is the "script" that our AI will read and follow to create your social media post!

Under the Hood: How the Prompt is Built Step-by-Step

Let's visualize the process of the Prompt Builder combining all the pieces:


User
Python Program (agent.py)
Data Source Scrapers
Prompt Builder Logic (in agent.py)
Run program, provide preferences (keyword, word limit)
"Get trending news and hashtags!"
Returns: News List, Hashtags List
"Combine all this data into a prompt!"
Organizes News, Hashtags, User Inputs
Returns: Fully crafted Prompt Text
Prompt is now ready for the AI!
User
Python Program (agent.py)
Data Source Scrapers
Prompt Builder Logic (in agent.py)

Explanation of the steps:

1. **User Input:** You provide specific details like the `keyword` and `word_limit` whe program asks.

2. **Data Collection:** `agent.py` calls the functions in `data_sources.py` (our "web reporters") to gather the latest `news` and `hashtags`.
3. **Data Preparation:** Inside `agent.py`, the lists of news and hashtags are converted into single strings (`news_str`, `hashtags_str`) that can be easily inserted into the prompt.
4. **Prompt Assembly:** This is where the Prompt Builder logic (still within `agent.py`) takes over. It uses an f-string template. It inserts `news_str`, `hashtags_str`, `keyword`, `keyword_count`, and `word_limit` into their designated spots in the template.
5. **Final Prompt:** The result is one complete, clear text message (the "prompt") that contains all the instructions and context the AI needs. This message is then ready to be sent to the AI model.

Our project actually has two different prompt "recipes" (`prompt_old` and `prompt_new` in `agent.py`) that you can choose from. They ask the AI to do similar things but with slightly different emphasis or wording. This shows how flexible prompt building can be!

```python
# From agent.py (simplified Prompt Builder section)

# ... (data prepared, user inputs gathered) ...

# Prompt "Recipe" 1: prompt_old
prompt_old = f'''
Create an interactive post for a Doctor's audience based on news:\n{news_str}\n
Use hashtags: {hashtags_str}.
Keep the word limit to {word_limit} max.
Use keyword {keyword} at least {keyword_count} times.
'''

# Prompt "Recipe" 2: prompt_new
prompt_new = f'''
You're a doctor. Create a social media post for patients.
Content must be based on: {news_str}
Use relevant hashtags from: {hashtags_str}
Integrate "{keyword}" at least {keyword_count} times.
Keep total word count within {word_limit} words.
'''

# Based on user choice, one of these will be selected and sent to the AI
if "d" in prompt_input: # If user chose 'old'
    final_prompt = prompt_old
elif "w" in prompt_input: # If user chose 'new'
    final_prompt = prompt_new

# The final_prompt is then passed to a function like post_gemini()
# For example: post_gemini(final_prompt)

```

Explanation:

- You can see how both `prompt_old` and `prompt_new` dynamically pull in the `news_str`, `hashtags_str`, `word_limit`, `keyword`, and `keyword_count` using the `{}` f-string syntax.
- The only difference is the wording and emphasis of the instructions. This flexibility is a key benefit of a "Prompt Builder" approach!

Conclusion

In this chapter, we explored the crucial role of the **Prompt Builder**. We learned that it acts as the AI's "scriptwriter," meticulously combining scraped data and user preferences into clear, detailed instructions called "prompts." We saw how `agent.py` uses f-strings to dynamically create these prompts, ensuring the AI gets precisely the information it needs to generate the desired social media posts.

Having crafted our perfect "script," the next logical step is to deliver it to our "actor" – the powerful AI model itself! In the next chapter, we'll dive into how our project connects with these incredible Large Language Model (LLM) Connectors.

Post-creation

Siddharthakhandelwal/Post-creation

english

gemini-2.5-flash

c3b5daa

Aug 3, 2025

Chapters

Show Raw Markdown

Chapter 4: Large Language Model (LLM) Connectors

Welcome back! In our previous chapter, we learned how our project meticulously crafts specific instructions for AI models, transforming raw data and user preferences into clear, detailed "prompts" using the **Prompt Builder**. We've prepared the perfect "script" for our AI. Now, it's time to send that script to the actual "actors" – the powerful AI models themselves!

Why Do We Need "Large Language Model (LLM) Connectors"? (Our AI Translators)

Imagine you have a beautifully written script, but you need to send it to three diffe who speak different languages (like English, Spanish, and French). You wouldn't learn languages yourself just to send the script, would you? Instead, you'd hire a specialized translator for each language. You hand your English script to the English translator, the Spanish translator, or the French translator, and they handle all the complex details of communicating with their respective actors.

In the world of our `Post-creation` project, our "actors" are **Large Language Models (LLMs)** like Google's Gemini, Perplexity, or Groq. These are sophisticated AI brains that can understand human language and generate creative text. While they all "speak" the language of prompts, each one has its own unique technical way of receiving those prompts and sending back responses.

This is exactly why we need **Large Language Model (LLM) Connectors**! Think of these as specialized "translators" or "messengers" for different AI brains. Each connector knows precisely how to communicate with a specific AI service (like Gemini, Perplexity, or Groq), sending it a request (our "prompt") and understanding its response. Instead of our main program (`agent.py`) learning all the different ways to talk to each AI, it just hands the message to the right connector, which then handles all the technical details of the conversation.

Our main goal in this chapter is to understand how our project uses these "translators" to send our carefully built prompts to the correct AI model and receive the generated social media posts back.

What is a Large Language Model (LLM)? (The AI Brain)

Before we talk about connectors, let's briefly understand the "brains" they connect to. A **Large Language Model (LLM)** is a type of artificial intelligence trained on a massive amount of text data. This training allows them to:

- Understand and generate human-like text.
- Answer questions.
- Summarize information.
- Translate languages.
- And, most importantly for us, create original content like social media posts based on instructions (our prompts!).

What is an LLM Connector? (The Communication Specialist)

An **LLM Connector** is essentially a piece of code designed to speak the "language" of a specific LLM service. It handles all the nitty-gritty details like:

- **API Authentication:** Using the API keys we securely stored.
- **Request Formatting:** Packaging our prompt into the specific format the AI serv s (e.g., JSON).

- **Sending Requests:** Using internet communication tools (like `requests` for Perplexity/Groq, or a special library for Gemini).
- **Response Parsing:** Understanding the AI's reply and extracting just the generated text we need.

Our project uses three different LLM connectors, one for each AI model:

- `post_gemini()` : Connects to Google's Gemini AI.
- `post_perplexity()` : Connects to Perplexity AI.
- `post_groq()` : Connects to Groq AI.

How Our Project Uses LLM Connectors

After our **Prompt Builder** has prepared the perfect prompt, our `agent.py` script asks you which AI model you want to use. Based on your choice, it then calls the corresponding LLM connector function (`post_gemini` , `post_perplexity` , or `post_groq`) and passes the prompt to it.

Here's a simplified look at how `agent.py` decides which connector to use:

```python
# From agent.py (simplified)

# ... (Scraping and Prompt Building happens) ...

# 1. User chooses the AI model
model_input = input("Enter model name ( Gemini , perplexity , Groq ):")

# 2. User chooses the prompt style (old or new)
prompt_input = input("Which prompt ( Old or New ): ")

# 3. Select the final prompt based on user choice
if "d" in prompt_input:
    final_prompt = prompt_old # Using the 'old' prompt style
elif "w" in prompt_input:
    final_prompt = prompt_new # Using the 'new' prompt style

# 4. Call the correct LLM Connector based on user's model choice
if "Ge" in model_input:
    print("\nGenerating post using Gemini...")
    post_gemini(final_prompt) # Call the Gemini connector
elif "p" in model_input:
    print("\nGenerating post using perplexity...")
    post_perplexity(final_prompt) # Call the Perplexity connector
```

```python
elif "Gr" in model_input:
    print("\nGenerating post using Groq...")
    post_groq(final_prompt) # Call the Groq connector

```

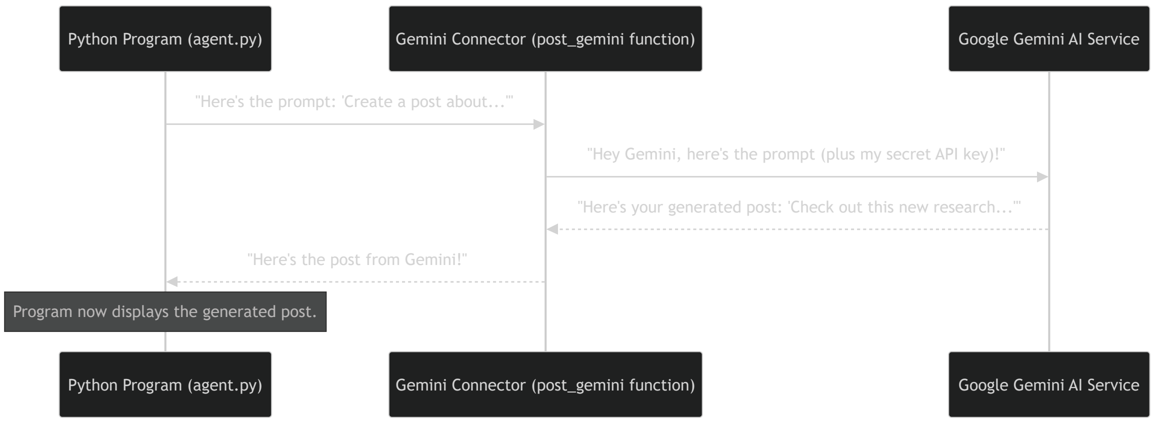
Explanation:

1. `model_input = input( ... )` : This line asks you to type which AI model you prefer.
2. `prompt_input = input( ... )` : This asks you to choose between the "old" or "new" prompt style (as discussed in Chapter 3: Prompt Builder).
3. `final_prompt = ...` : One of the prepared prompts (`prompt_old` or `prompt_new`) is selected.
4. `if "Ge" in model_input:` (and similar `elif` statements): This checks your `model_input` . If you typed something like "Gemini" (so "Ge" is in it), it calls the `post_gemini` function, passing our `final_prompt` as its instruction. The same logic applies to "p" for Perplexity and "Gr" for Groq.

When one of these functions is called, it sends the `final_prompt` to the chosen AI model, waits for the AI to generate a post, and then prints that post right to your screen!

Under the Hood: How an LLM Connector Works Step-by-Step

Let's visualize what happens when our `agent.py` calls an LLM connector, using Gemini as an example:


Python Program (agent.py)
Gemini Connector (post_gemini function)
Google Gemini AI Service
"Here's the prompt: 'Create a post about...'"
"Hey Gemini, here's the prompt (plus my secret API key)!"
"Here's your generated post: 'Check out this new research...'"
"Here's the post from Gemini!"
Program now displays the generated post.
Python Program (agent.py)
Gemini Connector (post_gemini function)
Google Gemini AI Service

Explanation of the steps:

1. **Program Calls Connector:** Our `agent.py` script calls the `post_gemini()` func ng it the ready-to-use prompt.

2. **Connector Configures:** The `post_gemini()` function first sets itself up to talk to Gemini. This includes getting your `GEMINI_API_KEY` (securely fetched from the `.env` file, thanks to Chapter 1: API Key Management!) and configuring the Gemini library.
3. **Connector Sends Prompt:** It then sends the prompt to the actual Google Gemini AI service over the internet.
4. **AI Generates Response:** The Google Gemini AI service processes the prompt and generates a social media post.
5. **AI Sends Response Back:** The generated post is sent back to our `post_gemini()` function.
6. **Connector Parses & Returns:** The `post_gemini()` function extracts just the text of the generated post from the AI's technical response and hands it back to `agent.py`.
7. **Program Displays Post:** Finally, `agent.py` prints the generated post to your console.

The process is very similar for Perplexity and Groq, just with slightly different technical steps for how the connector communicates with *their* specific services.

Diving Deeper: The Connector Code in `agent.py`

Let's look at how these connector functions (`post_gemini`, `post_perplexity`, `post_groq`) are implemented in `agent.py`. Remember, these functions rely on the API keys securely loaded by `load_dotenv()` from Chapter 1: API Key Management.

1. Gemini Connector (`post_gemini`)

The Gemini connector uses a special Python library (`google-generativeai`) to make communication easier.

```python
# From agent.py (simplified - Part 1: Setup)
import os
from dotenv import load_dotenv
import google.generativeai as genai

load_dotenv() # Load API keys from .env

def post_gemini(prompt):
    try:
        # 1. Get the Gemini API Key
        api_key = os.getenv("GEMINI_API_KEY")
        # 2. Tell the Gemini library to use our API key
        genai.configure(api_key=api_key)
        # 3. Choose which Gemini model we want to use
        model = genai.GenerativeModel("gemini-2.5-pro")
        # ... (rest of the function) ...

```

Explanation:

- `import google.generativeai as genai` : This line brings in the special tools for talking to Gemini.
- `api_key = os.getenv("GEMINI_API_KEY")` : This securely retrieves your Gemini API key from your computer's environment (where `python-dotenv` put it).
- `genai.configure(api_key=api_key)` : This tells the `google-generativeai` library, "Use *this* API key for all our conversations with Gemini."
- `model = genai.GenerativeModel("gemini-2.5-pro")` : This line selects a specific version of the Gemini AI model to use, in this case, "gemini-2.5-pro."

Now, let's see how the prompt is sent and the response received:

```
# From agent.py (simplified - Part 2: Send & Receive)
# ... (previous setup code for post_gemini) ...

def post_gemini(prompt):
    try:
        # ... (API key setup and model selection) ...

        # 4. Send our prompt to the Gemini model!
        response = model.generate_content(prompt)

        # 5. Get the generated text from the AI's response
        print(response.text)
        return response.text
    except Exception as e:
        # If anything goes wrong (e.g., internet issue, bad key)
        print(f"Error generating post: {e}")
        return f"[Error generating post: {e}]"

```

Explanation:

- `response = model.generate_content(prompt)` : This is the core line! It send to the `model` we configured. The `model` then talks to the actual Gemini AI servic
- `print(response.text)` : The `generate_content` call returns a `response` objec ccess its `text` property to get the actual generated social media post.

- `except Exception as e:` : This is error handling. If there's a problem (like no internet, or a wrong API key), our program won't crash. Instead, it will print an error message.

2. Perplexity Connector (`post_perplexity`)

The Perplexity and Groq connectors use the standard `requests` library, which is like a general-purpose tool for sending and receiving information over the internet (similar to how we used it in Chapter 2: Data Source Scrapers).

```python
# From agent.py (simplified - Part 1: Setup)
import requests # Needed for internet requests
import os
from dotenv import load_dotenv

# ... (load_dotenv from .env) ...

def post_perplexity(prompt):
    try:
        # 1. Get the Perplexity API Key
        api_key = os.getenv("Perplexity")
        # 2. Perplexity's special internet address for AI chat
        url = "https://api.perplexity.ai/chat/completions"
        # 3. "ID card" and "envelope type" for our request
        headers = {
            "Authorization": f"Bearer {api_key}", # Shows our API key
            "Content-Type": "application/json"   # Tells them we're sending JSON da
        }
        # ... (rest of the function) ...

```

Explanation:

- `import requests` : We bring in the `requests` library.
- `api_key = os.getenv("Perplexity")` : Get the Perplexity API key.
- `url = "https://api.perplexity.ai/chat/completions"` : This is the specific web address (API endpoint) where Perplexity's AI chat service listens for requests.
- `headers = { ... }` : These are like special notes on the "envelope" of our messa ell Perplexity who is sending the request (`Authorization` with our API key) and wha content we're sending (`Content-Type: application/json`).

Now, let's look at sending the prompt and getting the reply:

```python
# From agent.py (simplified - Part 2: Send & Receive)
# ... (previous setup code for post_perplexity) ...

def post_perplexity(prompt):
    try:
        # ... (API key, url, headers defined) ...

        # 4. Prepare the "message content" for Perplexity
        payload = {
            "model": "sonar-pro", # The specific Perplexity AI model to use
            "messages": [
                {"role": "user", "content": prompt} # Our prompt here!
            ],
            "stream": False # We want the full answer at once, not in parts
        }

        # 5. Send the request (our message with headers and payload)
        response = requests.post(url, headers=headers, json=payload)

        # 6. Unpack the AI's reply (it comes as JSON data)
        response_json = response.json()
        content = response_json["choices"][0]["message"]["content"]

        print(content) # Show the generated post
        return content
    except Exception as e:
        print(f"Error generating post: {e}")
        return f"[Error generating post: {e}]"

```

Explanation:

- `payload = { ... }` : This is the actual "body" of our message. It specifies which AI `model` to use on Perplexity's side and, most importantly, includes our `prompt` as a "user message."
- `response = requests.post(url, headers=headers, json=payload)` : This line uses `requests` to send our `payload` to the `url` with the specified `headers`. `post` means we are sending data *to* the server.
- `response_json = response.json()` : The AI's reply comes back as structured We convert it into a Python dictionary.
- `content = response_json["choices"][0]["message"]["content"]` : This is li ng through a nested box to find the actual generated text. AI responses often contain extra

information, so we need to extract just the `content` of the `message` from the first `choice`.

3. Groq Connector (`post_groq`)

The Groq connector works almost identically to the Perplexity connector, as both use a similar type of API structure.

```python
# From agent.py (simplified - Groq Connector)
import requests
import os
from dotenv import load_dotenv

# ... (load_dotenv from .env) ...

def post_groq(prompt):
    try:
        api_key = os.getenv("GROQ_API") # Get Groq API Key
        url = "https://api.groq.com/openai/v1/chat/completions" # Groq's address
        headers = {
            "Authorization": f"Bearer {api_key}",
            "Content-Type": "application/json"
        }
        payload = {
            "model": "llama-3.3-70b-versatile", # Groq model
            "messages": [{"role": "user", "content": prompt}] # Our prompt
        }
        response = requests.post(url, headers=headers, json=payload)

        response_json = response.json()
        content = response_json["choices"][0]["message"]["content"] # Extract pos
        print(content)
        return content
    except Exception as e:
        print(f"Error generating post: {e}")
        return f"[Error generating post: {e}]"

```

Explanation:

As you can see, the structure for `post_groq` is very similar to `post_perplexity`. differences are:

- `api_key = os.getenv("GROQ_API")` : It fetches the `GROQ_API` key.
- `url = "https://api.groq.com/openai/v1/chat/completions"` : It uses Groq's specific API endpoint.
- `"model": "llama-3.3-70b-versatile"` : It specifies a different AI model offered by Groq.

This shows the power of **LLM Connectors**: even though the underlying AI services are different, our project can interact with them using a similar pattern thanks to these specialized functions, making it easy to swap or add new AI models in the future!

Conclusion

In this chapter, we explored the vital role of **Large Language Model (LLM) Connectors**. We learned that these are specialized "translators" or "messengers" that allow our `Post-creation` project to communicate seamlessly with different AI models like Gemini, Perplexity, and Groq. We saw how `agent.py` selects the right connector based on user input and how each connector handles the technical details of sending prompts and receiving generated posts, always securely using our API keys.

With our ability to get fresh data, build perfect prompts, and now connect to powerful AI models, we have all the pieces in place! In the final chapter, we'll put everything together and see the entire Main Post Generation Workflow in action.

Post-creation

Siddharthakhandelwal/Post-creation

english

gemini-2.5-flash

c3b5daa

Aug 3, 2025

Chapters

Show Raw Markdown

Chapter 5: Main Post Generation Workflow

Welcome to the grand finale of our `Post-creation` project tutorial! In our previous chapters, we've carefully built all the necessary pieces:

- In Chapter 1: API Key Management, we learned to securely handle our secret API keys.
- In Chapter 2: Data Source Scrapers, we saw how our "web reporters" gather fresh news and hashtags.
- In Chapter 3: Prompt Builder, we became "scriptwriters" for the AI, crafting detai instructions.
- And in Chapter 4: Large Language Model (LLM) Connectors, we discovered how "talks" to powerful AI models.

Now, it's time to put all these fantastic components together!

Why Do We Need a "Main Post Generation Workflow"? (The Project Manager)

Imagine you're building a fancy custom car. You have an engine, wheels, seats, a steering wheel, and a painting station. Each part is perfect on its own, but they don't magically become a car. You need a detailed plan, a schedule, and a "project manager" to make sure:

1. The engine is installed first.
2. Then the wheels.
3. Then the seats, and so on.
4. Finally, it goes to the painting station.

Without this "project manager," chaos! Parts everywhere, no finished car.

Our `Post-creation` project is like building that car. We have separate, powerful components (scrapers, prompt builder, AI connectors). The **Main Post Generation Workflow** is our "project manager." It's not a single piece of code or a specific file; instead, it's the *orchestration* – the way our main `agent.py` script smartly calls each component in the correct order, ensuring everything runs smoothly from start to finish to create that perfect social media post.

Our Goal for this chapter: To understand how `agent.py` acts as this "project manager," directing all the steps to generate a social media post for a doctor's audience.

What is the "Main Post Generation Workflow"?

The "Main Post Generation Workflow" is the entire sequence of operations that happens from the moment you run the program until you see the final social media post. It handles:

1. **User Preferences:** Asking you what kind of post you want (which AI model, keywords, word limit).
2. **Information Gathering:** Directing the Data Source Scrapers to get fresh news and hashtags.
3. **Instruction Building:** Telling the Prompt Builder to create the perfect "script" for the AI.
4. **AI Communication:** Choosing the correct LLM Connector to send the "script" to the AI and receive the generated post.
5. **Display:** Showing you the final result.

It ensures that every step is performed in the right order and that the output of on
becomes the input for the next.

How Our Project Uses the Workflow (From Your View)

From your perspective, using the "Main Post Generation Workflow" is super simple. You just run the `agent.py` file, and it guides you through the process:

1. It tells you it's scraping data.
2. It asks you a few questions (e.g., "Which AI model?", "What keyword do you want?").
3. It then tells you it's generating the post.
4. Finally, it prints the completed social media post to your screen!

Let's look at the core of `agent.py` to see this in action:

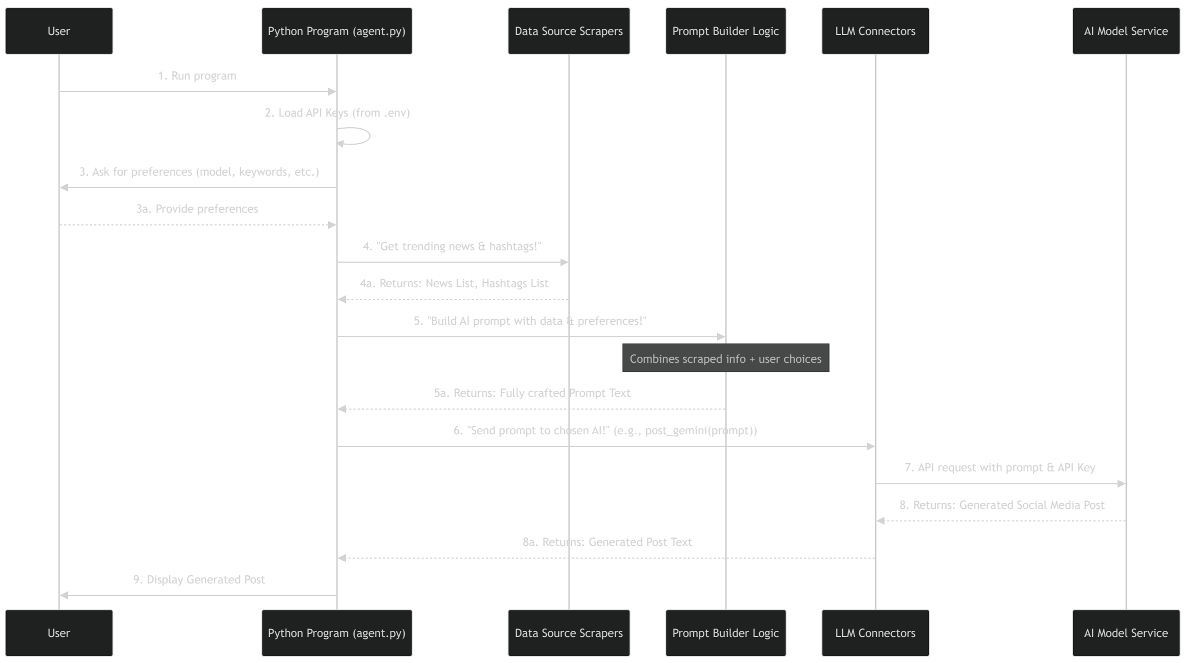
```python
# From agent.py (Simplified Main Workflow)
from data_sources import scrape_trending_hashtags, scrape_trending_news
# ... (Other imports like requests, os, dotenv, genai) ...

# 1. Load API keys securely (Chapter 1)
load_dotenv()

if __name__ == "__main__":
    # 2. Information Gathering (Chapter 2)
    print("Scraping trending hashtags...")
    hashtags = scrape_trending_hashtags()
    print("Scraping trending news...")
    news = scrape_trending_news()

    # Prepare data for prompt
    hashtags_str = ', '.join(hashtags)
    news_str = '\n'.join(news)

    # 3. User Preferences (Input)
    model_input = input("Enter model name (Gemini, perplexity, Groq): ")
    prompt_input = input("Which prompt (Old or New): ")
    word_limit = int(input("Enter the word limit: "))
    keyword = input("Enter the Key word you want: ")
    keyword_count = int(input("Keyword count: "))

    # 4. Instruction Building (Prompt Builder - Chapter 3)
    # Based on user choices, define final_prompt (either prompt_old or prompt_new
    if "d" in prompt_input:
        final_prompt = f'''... (template for prompt_old with {news_
    elif "w" in prompt_input:
        final_prompt = f'''... (template for prompt_new with {news_st

    # 5. AI Communication (LLM Connectors - Chapter 4)
    if "Ge" in model_input:
        print("\nGenerating post using Gemini...")
        post_gemini(final_prompt) # Uses Gemini connector
    elif "p" in model_input:
        print("\nGenerating post using perplexity...")
        post_perplexity(final_prompt) # Uses Perplexity connector
    elif "Gr" in model_input :
        print("\nGenerating post using Groq...")
        post_groq(final_prompt) # Uses Groq connector

```

Explanation:

- This simplified code shows the `if __name__ == "__main__":` block, which is where the main "project manager" logic resides in `agent.py`.
- You can see how it starts by loading environment variables, then calls the scraping functions, takes user inputs, builds the prompt using f-strings, and finally calls the chosen AI connector function.
- Each `print` statement lets you know what the "project manager" is doing at that moment!

Under the Hood: The Workflow Step-by-Step

Let's visualize the entire journey of creating a post, showing how the "Main Post Generation Workflow" orchestrates everything:


User
Python Program (agent.py)
Data Source Scrapers
Prompt Builder Logic
LLM Connectors
AI Model Service
1. Run program
2. Load API Keys (from .env)
3. Ask for preferences (model, keywords, etc.)
3a. Provide preferences
4. "Get trending news & hashtags!"
4a. Returns: News List, Hashtags List
5. "Build AI prompt with data & preferences!"
Combines scraped info + user choices
5a. Returns: Fully crafted Prompt Text
6. "Send prompt to chosen AI!" (e.g., post_gemini(prompt))
7. API request with prompt & API Key
8. Returns: Generated Social Media Post
8a. Returns: Generated Post Text
9. Display Generated Post

Explanation of the Workflow Steps:

1. **User Initiates:** You start the program by running `python agent.py`.
2. **Secure Setup:** `agent.py` first uses `load_dotenv()` to securely load all your API keys from your `.env` file into memory, as learned in Chapter 1: API Key Management.
3. **Gathering Preferences:** The program interacts with you, asking for details like which AI model to use (Gemini, Perplexity, Groq), the desired word limit, and specific keywords to include.
4. **Data Collection:** `agent.py` then calls the functions within our Data Source Scrapers (`scrape_trending_hashtags()`, `scrape_trending_news()`). These functions go out to the internet, gather the latest information, and return it as lists to `agent.py`.
5. **Prompt Construction:** With the scraped data and your preferences in hand, the `agent.py` script acts as the Prompt Builder. It takes all these pieces and cleverly combines them into one detailed text message (our "prompt") for the AI. It chooses between `prompt_old` or `prompt_new` based on your selection.
6. **AI Communication:** Based on your chosen AI model, `agent.py` selects the correct LLM Connector function (`post_gemini()`, `post_perplexity()`, or `post_groq()`). It then passes the perfectly crafted prompt to this connector.
7. **AI Generation:** The chosen LLM connector then handles all the technical details of securely sending the prompt to the actual AI service (like Google Gemini, Perplexity, or Groq) e internet. The AI processes the request and generates the social media post.
8. **Receiving the Post:** The AI sends its generated post back to the LLM connector connector extracts just the text of the post and returns it to `agent.py`.

9. **Displaying the Result:** Finally, `agent.py` prints the complete social media post right there in your console!

The `agent.py` "Project Manager" in Detail

Let's look at how the `if __name__ == "__main__":` block in `agent.py` (which runs when you execute the script directly) coordinates everything:

```
# Part 1: Setup and Data Collection
# ... (imports for data_sources, requests, os, dotenv, genai) ...

load_dotenv() # Load API keys - Step 2

if __name__ == "__main__":
    print("Scraping trending hashtags...") # User feedback
    hashtags = scrape_trending_hashtags() # Call Scraper - Step 4
    print("Scraping trending news...") # User feedback
    news = scrape_trending_news() # Call Scraper - Step 4

    hashtags_str = ', '.join(hashtags) # Prepare data for prompt
    news_str = '\n'.join(news) # Prepare data for prompt

```

Explanation: This first part handles the initial setup (loading API keys) and then immediately kicks off the data collection process by calling the scraper functions. It also formats the scraped data into neat strings, ready for the prompt.

```
# Part 2: User Input and Prompt Building
# ... (continues from Part 1) ...

    # Gather user inputs - Step 3
    model_input = input("Enter model name ( Gemini , perplexity , Groq ):")
    prompt_input = input("Which prompt ( Old or New ): ")
    word_limit = int(input("Enter the word limit: "))
    keyword = input("Enter the Key word you want: ")
    keyword_count = int(input("Keyword count in the whole post: "))
    keyword_count_line = int(input("Keyword count per Line: "))

    # Define prompt templates (simplified for brevity)
    prompt_old = f'''Create an interactive post for a Doctor's audienc
        Based on news:\n{news_str}\nUse hashtags: {hashtags_str}.
        Keep word limit to {word_limit} max. Use keyword {keyword} {keyword_count}
    '''
    prompt_new = f'''You're a doctor. Create a social media post for patients...
        Content must be based on: {news_str}\nUse relevant hashtags from: {hashtags
        Integrate "{keyword}" at least {keyword_count} times.
        Keep total word count within {word_limit} words.
    '''

    # Select the final prompt based on user choice - Step 5
    if "d" in prompt_input:
        final_prompt = prompt_old
    elif "w" in prompt_input:
        final_prompt = prompt_new

```

Explanation: Here, the "project manager" pauses to ask you for your preferences. Then, it uses these preferences along with the previously scraped data to build the full, detailed prompt using the f-string magic we discussed in Chapter 3: Prompt Builder. It chooses between the "old" or "new" prompt template based on your input.

```
# Part 3: AI Communication and Output
# ... (continues from Part 2) ...

    # Call the correct LLM Connector - Step 6 & 7 & 8
    if "Ge" in model_input:
        print("\nGenerating post using Gemini...") # User feedback
        post_gemini(final_prompt) # Calls Gemini Connector
    elif "p" in model_input:
        print("\nGenerating post using perplexity...") # User feedback
        post_perplexity(final_prompt) # Calls Perplexity Connector
    elif "Gr" in model_input :
        print("\nGenerating post using Groq...") # User feedback
        post_groq(final_prompt) # Calls Groq Connector

```

Explanation: This final part is where the "project manager" executes the core task. Ba your model choice, it calls the appropriate LLM connector function (Chapter 4: Large La el (LLM) Connectors), passing the meticulously built `final_prompt`. Once the AI gen st and the connector returns it, the connector function itself (`post_gemini`, `post_perp` , or `post_groq`) handles printing the final result to your screen (Step 9), completing the workflow!

Conclusion

Congratulations! You've now completed the `Post-creation` project tutorial. In this final chapter, we pulled back the curtain to reveal the **Main Post Generation Workflow**. We learned that this workflow, primarily managed by the `agent.py` script, acts as the "project manager" that orchestrates all the individual components we've learned about: secure API key management, data source scraping, prompt building, and large language model connection.

By understanding this workflow, you now see how these seemingly separate parts come together to form a cohesive and powerful application capable of generating relevant and engaging social media posts on demand. You've built a solid foundation for understanding how AI-powered applications gather information, process instructions, and deliver results!