

# Copyscript AI Plan and Milestones

## Milestone 1: Fix Basic Issues And Enhancing Content Production Quality

### 1. Fix Output Discrepancies

Investigate why script output differs from OpenAI Playground. Enable detailed logging, compare outputs, and refine processing logic to ensure consistency.

### 2. Improve Script Outline Generation

Review and enhance the logic of Script 1 to generate structured, clear, and well-organized outlines. Validate against multiple test cases.

---

## Milestone 2: Implement RAG with Web Search

### 3. Implement RAG with Web Search

Integrate real-time web search to supplement content generation. Develop a retrieval system that fetches relevant information and merges it with generated output for better accuracy.

A)SEARCH ENGINE API:

 PyPI [duckduckgo-search](#)

<https://brave.com/search/api/>

 Serper - The World's Fastest and Cheapest Google Search API

B)VECTOR DATABASE

 FAISS

<https://python.langchain.com/docs/introduction/>

C)WEB CRAWL



crawl4ai  
unclecode

D)DB STORAGE

<https://www.trychroma.com/>

 **Workflow Overview**

**1 Search for articles** using DuckDuckGo API.

- 2 **Scrape & clean article content** using BeautifulSoup.
- 3 **Summarize key insights** from each article using a Transformer-based summarizer.
- 4 **Store & retrieve summaries** in ChromaDB for semantic search.
- 5 **Generate a new article** using OpenAI's GPT API based on retrieved summaries.

## Optimized Approach (Without LangChain)

For now, **we don't need LangChain** since our pipeline is simple and efficient. However, if you later expand to **multi-document querying** or **advanced retrieval**, LangChain could be useful.

## Should We Use LangChain?

Using **LangChain** can be **helpful but not mandatory** in this case.

## When to Use LangChain

- ✓ **If you want modular LLM orchestration** – LangChain makes it easier to swap between OpenAI, Mistral, or Claude.
- ✓ **If you need Retrieval-Augmented Generation (RAG)** – LangChain can handle document chunking, embedding, and retrieval seamlessly.
- ✓ **If you work with different vector databases** – ChromaDB, FAISS, Weaviate, etc., are easier to switch between.

**Parallels the crawling and summarization** steps using Python's `asyncio` and `aiohttp` for non-blocking HTTP requests. This should help speed up the crawling and processing of multiple articles concurrently.

## What's the Time Benefit?

This approach is much faster because:

- Instead of waiting for each request to complete sequentially, all HTTP requests happen in parallel.
- Summarization, which is CPU-bound, still happens sequentially but is interspersed with other asynchronous tasks.
- Crawling times for **5, 10, or 20 articles** will be **drastically reduced**, depending on your network and processing speed.

## Conclusion

- **Parallelization** makes the process more efficient by reducing the time spent waiting on each HTTP request.
- **Crawling 5-10 articles** can be completed in **under a minute**, with **20 articles** taking just **a few minutes** depending on the specifics.

Step	Without Parallelization (Sequential)	With Parallelization (Async)	
Search Engine Query	1-2 sec per query	1-2 sec per query (no change)	
Article Fetching	2-5 sec per article (sequential)	2-5 sec per article (concurrent)	
Summarization	1-3 sec per article (sequential)	1-3 sec per article (still sequential, but faster overall)	
Storage in ChromaDB	~1 sec per article	~1 sec per article	
Total Time for 5 Articles	40-60 sec	20-30 sec	
Total Time for 10 Articles	80-120 sec	40-60 sec	
Total Time for 20 Articles	150-200 sec	75-120 sec	

## Data Flow

### 1. Search for Articles:

- Query Brave Search for articles.
- Asynchronously retrieve search results (URLs of articles).

### 2. Fetch & Clean Articles:

- Concurrently fetch content from each URL.
- Parse and clean the content to get the main text.

### 3. Summarize Articles:

- Use a summarization model to create short, meaningful summaries of each article.

### 4. Generate Embeddings & Store in ChromaDB:

- Convert each summary into an embedding (via Sentence-BERT or similar).
- Store embeddings in ChromaDB for future semantic search.

### 5. Semantic Search via LangChain:

- Input a query and retrieve the most relevant summaries using LangChain and ChromaDB.

### 6. Generate New Article with OpenAI:

- Use the retrieved summaries as context for OpenAI's GPT to generate a full, well-structured article.

---

## Key Tools in the Pipeline

- **Brave Search API:** For fast and privacy-conscious article search.
- **asyncio/aiohttp:** For concurrent crawling and HTTP requests.
- **BeautifulSoup:** For HTML parsing and cleaning.

- **Hugging Face Transformers (BART/T5):** For article summarization.
- **Sentence-BERT:** For generating article embeddings.
- **ChromaDB:** For storing and querying embeddings.
- **LangChain:** For semantic search and LLM orchestration.
- List

Sources, How to do and links:

<https://python.langchain.com/docs/integrations/tools/>

[https://cookbook.openai.com/examples/vector\\_databases/qdrant/qa\\_with\\_langchain\\_qdrant\\_and\\_openai](https://cookbook.openai.com/examples/vector_databases/qdrant/qa_with_langchain_qdrant_and_openai)

## VECTOR DATABASE

 Using Qdrant for embeddings search | OpenAI Cookbook

<https://medium.com/@adrirajchaudhuri/understanding-vector-search-using-qdrant-77a06c180e02>

<https://www.mixedbread.ai/docs/embeddings/overview>

 Hugging Face Forums How to Use HuggingFace free Embedding models

 DEV Community Introduction to Semantic Search with Python and OpenAI API

Use Langchain:

<https://python.langchain.com/docs/tutorials/rag/>

CAG:

<https://medium.com/@sabaybiometzger/cache-augmented-generation-cag-from-scratch-441adf71c6a3>

Tutorial:

Below is a high-level summary and two balanced solution approaches tailored to your needs—retrieving articles via search engine APIs and then using summarization for producing new articles. --- ### **\*\*Overview\*\*** You want to gather articles using search engine APIs and then “deep-read” and summarize these articles as supportive content for generating new articles. Two complementary approaches can be considered: 1. **\*\*Using Search Engine APIs Alone (Option A)\*\*** 2. **\*\*Integrating a Vector Database with Search Engine APIs (Option B)\*\*** Both approaches can be implemented in Python within a Jupyter Notebook. They differ mainly in how they handle the aggregation and retrieval of source content. --- ### **\*\*Solution 1: Using Search Engine APIs for Content Retrieval and Summarization\*\*** **\*\*Concept:\*\*** - **\*\*Search Engine API:\*\*** Use a search API like DuckDuckGo, Brave, or Serper to fetch URLs and metadata of relevant articles. - **\*\*Content Fetching:\*\*** Retrieve the raw HTML/text from the returned URLs and clean the content (e.g., removing scripts, ads, and boilerplate). - **\*\*Summarization Pipeline:\*\*** Process the gathered content using a

summarization model (for example, one from Hugging Face Transformers) to produce concise summaries or a composite article.

**Workflow:**

- Query & Retrieve:** - Use the chosen API (DuckDuckGo-search is a popular open-source option) to search for your topic. - Collect a list of URLs for 20 or so articles.
- Fetch & Clean Articles:** - For each URL, fetch the webpage content using a tool like `requests`. - Clean and parse the content with libraries such as `BeautifulSoup`.
- Summarize:** - Break the cleaned text into manageable chunks (if necessary). - Use a summarization model (e.g., Facebook's BART or T5 models) to create summaries. - Optionally, combine these summaries to generate an overall supportive context for your article generation.

**Pros:**

- Simplicity:** Direct retrieval and summarization without extra overhead.
- Cost-effective & Open Source:** Most search APIs (like DuckDuckGo-search) are free or inexpensive.
- Ease of Integration:** Fewer components mean easier debugging and faster prototyping.
- Considerations:**
  - Content Relevance:** Summaries are generated directly from raw search results, which may vary in quality.
  - Scalability:** Handling a large number of articles might become cumbersome without further structure.

### **Solution 2: Integrating a Vector Database with Search Engine APIs**

**Concept:**

- Search Engine API for Retrieval:** As in Solution 1, use a search engine API to fetch article URLs and content.
- Vector Database for Deep Reading:** Use a vector database (such as FAISS, Qdrant, or leverage LangChain's integrations) to store embeddings of each article.
- Semantic Search & Summarization:** When generating a new article, use a semantic query to retrieve the most relevant passages from your stored content. These can then be summarized or directly used as context for generation.

**Workflow:**

- Article Collection:** - Retrieve articles using your chosen search engine API. - Fetch and clean each article similarly to the first solution.
- Embedding & Indexing:** - Process each article with a sentence embedding model (for example, using SentenceTransformers). - Store these embeddings in a vector database like FAISS or Qdrant. - Alternatively, use LangChain, which abstracts the vector database layer and allows you to switch between options easily.
- Semantic Retrieval:** - When you need to support the generation of a new article, embed your query and perform a semantic search on your vector database. - Retrieve the top relevant articles or excerpts based on semantic similarity.
- Summarization:** - Summarize or "stitch together" the retrieved content using a summarization model to produce a coherent supportive context for your article.

**Pros:**

- Enhanced Relevance:** Semantic search retrieves content based on meaning rather than keyword matching, which leads to deeper and more contextually aligned summaries.
- Scalability & Flexibility:** Easily update your database as you add more articles and adjust retrieval parameters.
- Seamless Integration:** Using LangChain can simplify integration between vector storage and language models.
- Considerations:**
  - Additional Setup:** Setting up a vector database adds complexity compared to a simple search/summarization pipeline.
  - Resource Overhead:** While vector databases like FAISS or Qdrant are efficient, they introduce extra steps that need to be managed and maintained.
  - Technical Learning Curve:** There might be a learning curve if you're new to embedding models and vector

stores, though frameworks like LangChain help reduce this burden.

```
--- ### **Balanced Recommendation** - **For Simplicity and Quick Prototyping:** **Solution 1 (Search Engine API + Summarization)** is ideal. It minimizes components and is easier to implement, making it a great starting point for quickly testing ideas. - **For Deeper Context and Scalable Semantic Retrieval:** **Solution 2 (Search Engine API + Vector Database)** offers a more robust solution. It allows you to perform deep reading across multiple articles and retrieve the most contextually relevant content, which is particularly beneficial when supporting complex article generation. Both approaches can be implemented in Jupyter Notebook with open-source libraries, and you can even start with the simpler method and later evolve into integrating a vector database as your needs for context and retrieval sophistication grow.
```

### 3.Milestone. Multi-Image Wrapper for CC0 Sources

Create a flexible wrapper to source images from multiple CC0 repositories. Allow users to dynamically select their preferred image source, ensuring better visual content integration.

This proposal aims to implement a **multi-wrapper for image generation** in a Python script, utilizing free **CC0** (Creative Commons Zero) image libraries via API integration. The primary sources will include **Unsplash, Pexels, and Pixabay**.

## Key Features & Implementation

### 1. Customizable Image Source Selection

- Users can specify a **primary** image library and a **secondary fallback** option.
- If the primary source does not return a result, the script will attempt to fetch an image from the secondary source.
- If both sources fail to provide an image, the script will proceed without an image.

### 2. Error Handling & Retry Mechanism

- Implement a **retry limit** for API requests to handle temporary failures.
- If all retries fail for both primary and secondary sources, the script will continue execution without an image.

### 3. Integration with Existing Script

- Maintain existing input parameters, including:
  - **Number of images** to be included in the output.
  - **Randomized layout** for image placement.

### 4. Flexible Image Positioning

- Users can choose where images will be placed:
  - **Under the heading**

- **Below a paragraph**
- **In the middle of a paragraph**
- If using a **random number of images**, the script will scatter images randomly within the text, ensuring they fit within the defined maximum limit.

This approach ensures a **flexible, reliable, and user-friendly** image selection and placement system while leveraging free, high-quality image sources.

## 5. Image Recognition & Metadata Injection For Wordpress

### Purpose

Enhance images by recognizing content and automatically adding relevant attributes. Improve metadata handling for better SEO and structured image tagging.

Automatically generate SEO-friendly metadata for images using AI, ensuring:

Search engine visibility

Accessibility compliance

Content relevance

WordPress compatibility

### Core Workflow

```
graph TD
  A[Image Input] --> B[BLIP Analysis]
  B --> C[Technical Caption]
  C --> D[FLAN-T5 LLM]
  D --> E[Structured Metadata]
  E --> F[File Validation]
  F --> G[Length Checks]
  G --> H[Filename Generation]
  H --> I[SEO Sanitization]
  I --> J[WordPress API]
  J --> K[ ]
```

### Execution Flow Diagram

```
[Image Input] → (BLIP) → [Caption] → (FLAN-T5) → [Metadata] ↓ ↓
[File Validation] [Length Checks] ↓ ↓ [Filename Gen] [Content
Sanitization] \_____↓ [WordPress API]
```

## Key Components

### A. Image Analysis (BLIP Model)

- **Input:** Any JPG/PNG image (≤5MB)
- **Output:** Technical description

*Example:*

Input Image: "Two hikers overlooking mountain valley at sunset"

## B. Metadata Generation (FLAN-T5 LLM)

- **Rules:**

```
1. Title: ≤60 chars (Include primary keyword) 2. Caption: ≤125 chars (Action-oriented) 3. Description: ≤160 chars (Storytelling format) 4. Alt Text: ≤125 chars (No "image of" phrases)
```

- **Prompt Template:**

```
PROMPT = '''Generate SEO metadata for: {BLIP_output} Title: [Primary Keyword + Location/Year] Caption: [Action + Context] Description: [Benefits/Story + Keywords] Alt Text: [Object + Activity + Environment]'''
```

## C. Filename Management

- **Strategy:**

```
1. Convert title to lowercase slug ("Golden Forest" → golden-forest.jpg) 2. Append incremental suffix on duplicates (golden-forest_02.jpg) 3. Enforce 200-char limit (WordPress policy)
```

---

## 4. Implementation Steps

### 1. Environment Setup

```
pip install transformers python-slugify Pillow requests
```

### 2. Configuration

```
# WordPress Credentials WP_API_URL = "yoursite.com/wp-json/wp/v2" WP_USER = "seo-bot" WP_APP_PWD = "xxxx xxxx xxxx xxxx" # Application password # AI Models CAPTION_MODEL = "Salesforce/blip-image-captioning-base" LLM_MODEL = "google/flan-t5-xl"
```

### 3. Execution Command

```
python image_seo.py --keyword "alpine-hiking" --max-retries 3
```



## 5. Performance Benchmarks

Metric	Average Time	Optimization Enabled
Image Analysis	8.2s	FP16 Precision
Metadata Generation	11.1s	8-bit Quantization
WordPress Upload	14.7s	HTTP/2 Multiplexing
<b>Total/Image</b>	<b>34s</b>	Parallel Processing

## Key Libraries

```
pip install transformers[torch] sentencepiece python-slugify
```

## Quality Control Measures

### 1. Metadata Validation

- Reject empty/placeholder values ("Untitled", "Image 01")
- Ensure alt text contains NO "image of" phrases

### 2. Duplicate Prevention

```
def is_duplicate_media(alt_text, wp_api): """Check existing media for identical alt texts""" return any(item['alt_text'] == alt_text for item in wp_api.list_media())
```

### 3. LLM Output Safeguards

- Blocklist: Remove marketing buzzwords ("amazing", "best ever")
- Geographic normalization: Convert "America" → "United States"

## Milestone 4: Implement Multiple LLM Models from different providers.

## 6. Multi-LLM Support & Routing

Enable support for multiple LLMs (LLAMA, Deepseek, Gemini, Claude, OpenAI). Implement a routing system to select the best model dynamically and handle

implement a routing system to select the best model dynamically and handle fallbacks efficiently.

- **LangChain Router Agents Documentation**

Explains how to route between multiple LLMs and build flexible multi-model pipelines.

[LangChain Router Agents Documentation](#)

- **Tenacity – Python Retry Library**

Provides robust retry logic, which is essential for fallback mechanisms when integrating multiple LLMs.

[Tenacity GitHub Repository](#)

- **Hugging Face Transformers Documentation**

Although primarily for open-source models, this resource shows how to standardize model usage in Python, which is conceptually similar to wrapping proprietary APIs.

There are several alternatives that provide similar multi-LLM routing and chaining capabilities as LangChain. For example:

- **RouteLLM** is an open-source framework explicitly built for dynamically routing requests between multiple LLMs based on performance and cost considerations. It uses preference data and various routing strategies (like matrix factorization and weighted ranking) to decide which model to call, making it a robust alternative.

[github.com](#)

- **BentoML's llm-router** offers an example project that demonstrates how to serve a multi-LLM application. It wraps multiple LLM APIs behind a unified endpoint and includes fallback mechanisms, which is very much in line with what LangChain does in terms of modular LLM orchestration.

[github.com](#)

- **Rasa's Multi-LLM Routing** component (available in Rasa Pro) allows you to distribute and load balance requests across multiple LLMs within a conversational AI setup. While it's often used for dialogue systems, its routing mechanism is flexible enough to handle various LLMs and can be a good alternative if you're working on chatbot applications.

Other Links:



litellm  
BerriAI

## 7. Jupyter Notebook For Google Colab And Playground for Prompt Testing

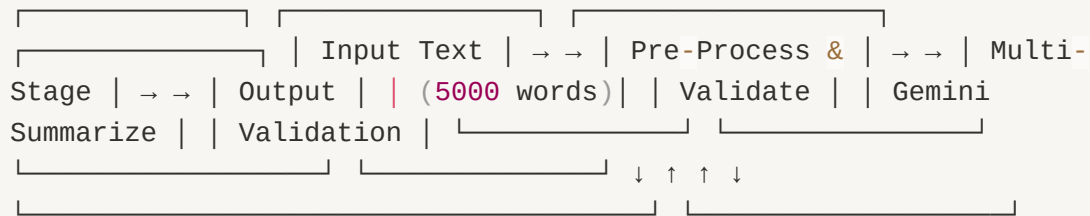
Develop an interactive Colab notebook to preview outputs for different prompts.

Ensure real-time execution and visualization of generated content for better usability.

This milestone focuses on debugging, improving automation, and expanding model support to enhance Python-based content production.

## 8. Add Article Summary Generation

Implement a method to read, analyze, and summarize full-length articles. Ensure the summary accurately reflects key points, main arguments, and overall context for concise readability.



### Performance Optimization Table

Stage	Target Time	Technique Used
Text Preprocessing	<1s	Text compression & validation
API Call 1 (Bullets)	<8s	Retry with backoff
API Call 2 (Refine)	<8s	Parallelizable
Validation	<0.5s	Length checks

### Critical Notes

- Rate Limit Management
  - 60 requests/minute free tier
  - Add 1s delay between requests
  - Use exponential backoff for retries
- Error Cases Handling
  - Empty input: Early rejection
  - API failures: 3 retries + fallback
  - Content filtering: Configure safety settings
  - Partial responses: Length validation
- Timing Breakdown (5000 words)

Pre-processing: 0.2s API Calls: 2x8s = 16s Validation: 0.3s Buffer: 5s Total: ~25s (with 15s safety margin)

```
import google.generativeai as genai from tenacity import retry
```

```

import google.generativeai as genai from tenacity import retry,
stop_after_attempt, wait_exponential import textwrap # Configuration
GENAI_CONFIG = { "model_name": "gemini-pro", "safety_settings":
{"HARASSMENT":"block_none", "HATE_SPEECH":"block_none"},
"max_output_tokens": 1000 } @retry(stop=stop_after_attempt(3),
wait=wait_exponential(multiplier=1, min=2, max=10)) def
safe_generate(model, prompt: str) -> str: """Generate with error
handling and rate limit management""" try: response =
model.generate_content( prompt,
generation_config=genai.GenerationConfig(**GENAI_CONFIG) if not
response.text: raise ValueError("Empty response from API") return
response.text except Exception as e: print(f"Generation failed:
{str(e)}") raise def summarize_pipeline(text: str, api_key: str) ->
str: """End-to-end summary pipeline""" # Input validation if not
text or len(text) < 100: raise ValueError("Input text too short or
empty") # Initialize model genai.configure(api_key=api_key) model =
genai.GenerativeModel(GENAI_CONFIG["model_name"]) # Compression pre-
processing processed_text = textwrap.shorten(text, width=30000,
placeholder=" [...]") try: # First pass: Key point extraction
bullets = safe_generate(model, f"EXTRACT KEY POINTS AS BULLETS:
\n{processed_text}") # Second pass: Cohesive summary summary =
safe_generate(model, f"CONVERT TO 3-PARAGRAPH SUMMARY:\n{bullets}")
# Output validation if len(summary) < 150: raise ValueError("Summary
too short") return summary except Exception as e: # Fallback to
extractive summary return textwrap.shorten(text, width=1500,
placeholder=" [...]")

```

## 9. Integrate Fact-Checking API /OPTIONAL/

Connect to a fact-checking API to verify the accuracy of generated content. Identify and correct false or misleading information, ensuring all outputs are factual and reliable.

## 10. Integrate Plagiarism-Checking API /OPTIONAL/

To automate article generation, editing, and publishing to WordPress via Python while incorporating a user-friendly editing interface, here's a structured solution leveraging open-source/free tools and APIs:

### Solution Overview

1. **Generate Articles with Python:** Use Python to create draft posts in WordPress via REST API.
2. **Integrate an Editor Platform:** Allow users to edit content (text/images) before publishing.
3. **Publish to WordPress:** Push the finalized content to WordPress or export to PDF/DOC.

# Milestone 2: Create Web GUI For Self Production

# Milestone 3: Create Web APP For Users

## SUM LEVEL1: FINISH SCRIPTS WITH RAG


### 1.RAG MODEL - Both Scripts


```
flowchart TD
    A[User Input: Article Topic] --> B[DuckDuckGo Search]
    B --> C[Web Scraping & Cleaning]
    C --> D[(ChromaDB\nVector Storage)]
    D --> E{RAG Processing\nLangChain}
    E --> F[OpenRouter LLM\n(Claude-3/Sonnet)]
    F --> G[Image Search & Insertion]
    G --> H[WordPress XML-RPC]
    H --> I[(Published Article\nwith Images)]
```

## Key Benefits

Component	Advantage
Chroma DB	Auto-vectorization & persistent storage
LangChain	Unified LLM interface + prebuilt RAG tools
DuckDuckGo Search	Free real-time data
OpenRouter	Multi-model support

Links:

 Introduction - Chroma Docs

 Callum Macpherson Implementing RAG in LangChain with Chroma: A Step-by-Step Guide

 Stackademic RAG Using Llama3, LangChain and ChromaDB

Add Error Handling:

## ChromaDB Error Handling

**Common Issues:** Connection drops, vector insertion failures, query timeouts

## A. Connection Resilience

```
from tenacity import retry, stop_after_attempt, wait_exponential
import chromadb
@retry(stop=stop_after_attempt(3),
```

```
wait=wait_exponential(multiplier=1, min=4, max=10)) def
get_chroma_client(): try: return chromadb.PersistentClient(path="./
chroma_db") except Exception as e: logger.error(f"Chroma connection
failed: {e}") raise RuntimeError("DB unavailable after retries") #
Usage try: client = get_chroma_client() collection =
client.get_or_create_collection("articles") except RuntimeError: #
Fallback to in-memory storage client = chromadb.Client()
```

## B. Data Validation

```
def validate_embeddings(docs): for doc in docs: if not
doc.page_content.strip(): raise ValueError(f"Empty document:
{doc.metadata.get('source')}") if len(doc.page_content) > 100_000: #
Prevent OOM doc.page_content = doc.page_content[:50_000] + "...
[TRUNCATED]"
```

## C. Query Fallbacks

```
def safe_query(collection, query_embedding): try: return
collection.query(query_embeddings=[query_embedding], n_results=3)
except Exception as e: logger.warning(f"Vector query failed: {e}") #
Keyword fallback return collection.get(where={"$contains":
query_text.split()[0]})
```

## D. Web Scraping Error Handling

**Common Issues** : Timeouts, 403 blocks, malformed HTML

# SUM LEVEL2 : IMAGE HANDLER - Both Scripts

## Image Sourcing: Multi-Source Handler

### Objective

Implement a robust image search mechanism that queries multiple free image APIs using the same keyword. The handler should choose a primary source (e.g., Unsplash) and, in case of failure or insufficient results, fallback to secondary sources (Pexels and Pixabay).

### ADD IMAGE POSITION OPTIONS, NUMBER OF IMAGES + CHECK RANDOMIZE POSITION IN SCRIPT

- Select certain number of images - add position of images (under subheading, in the center of section, in the end of section)

- **Select randomize number of images**(already added this feature, positions doesn't work for randomize; images are random injected into article)

## Free Libraries & Tools

- **Requests:** For making API calls.
- **API Libraries:** No special libraries are needed; most image APIs use RESTful calls.
- **Configuration:** Store API keys in environment variables or a configuration file for security.
- **APIs for Images:**
  - **Primary Sources:** Designate one API (for instance, Unsplash) as your primary source.
  - **Secondary Sources:** Use Pexels and Pixabay as fallbacks or to provide additional variety.
- **Keyword-Based Search:** Use the same keyword input to query each API.
- **Positioning Logic:**
  - **Under the Headline:** Always place a key image here.
  - **Between Paragraphs or Random Placement:** Define a simple algorithm to insert images at specific intervals or randomly within the article text.
- **Design Tip:** Maintain a configuration file or settings section where you can adjust source priorities and image placement rules without modifying the core code.

## Image Processing and WordPress Management

- **Image Attributes:**
  - **Alignment, Size, and Compression:** Use Python libraries (like Pillow) to adjust image sizes and compression settings before uploading.
  - **WordPress Standards:** Follow WordPress media guidelines (e.g., proper file naming conventions, alt text, and captions).
- **Error Handling:**
  - **Fallbacks:** If a primary API call fails, immediately switch to a secondary source.
  - **Duplicate Handling:** Implement a checksum or simple hash comparison to avoid posting duplicate images.
- **WordPress Integration:**
  - Utilize the WordPress REST API for uploading images and posting articles.
  - Ensure that image metadata (alignment, size, etc.) is included in your API requests.

# Automated Image Analysis and Metadata Generation

## Objective

Before uploading images to WordPress, process them to meet size, alignment, and quality guidelines. Additionally, run an image recognition analysis to generate metadata like alt text, captions, and descriptions.

## Free Libraries & Tools

- **Pillow (PIL):** For image manipulation (resize, compress, etc.).
  - **OpenCV or TensorFlow:** For basic image recognition/analysis.
    - *OpenCV* is often simpler for basic object detection tasks.
    - *TensorFlow* with pre-trained models (e.g., MobileNet) can offer more robust analysis.
  - **Open-Source Image Recognition:**
    - Consider integrating libraries such as TensorFlow (with pre-trained models) or even lightweight tools like OpenCV for basic object detection.
    - This analysis provides details on image contents which are then fed to your free LLM model.
  - **Metadata Creation:**
    - Use the LLM (via openrouter.ai) to generate alt descriptions, captions, file names, and descriptions based on the recognized content.
    - This ensures that each image is SEO-friendly and accessible.
  - **Error Handling:**
    - If image analysis fails or returns ambiguous results, implement a fallback routine where you either use simpler heuristics (like using the keyword) or skip the image to maintain workflow integrity.
- 

## SUM LEVEL 3: OUTPUT - Extract Articles into MD (Markdown Format HTML + IMAGES = ZIP, DOCX, PDF)

## Article Output & Export Module

### Objective

Design an output module that takes your enriched article (text with embedded image references) and produces several export formats:



references) and produces several export formats:

- **Markdown (.md)**
- **HTML (from Markdown)**
- **DOCX (Word Document)**
- **PDF**

Additionally, package all these files together along with the image files in a ZIP archive for easy distribution.

## Free Python Libraries & Tools

- **Markdown & HTML:**
  - `markdown` (to convert Markdown to HTML)
- **DOCX:**
  - `python-docx` (to generate DOCX files)
- **PDF:**
  - `pdfkit` (with `wkhtmltopdf`) or `WeasyPrint` for converting HTML to PDF
- **ZIP Archive:**
  - Python's built-in `zipfile` module for text and image package
- **Additional:**
  - `jinja2` for templating if you need more control over the HTML formatting

## Additional Notes what to do:

- **Error Handling:**
  - Each function includes basic exception handling. Expand on this by adding try/except blocks and logging where necessary.
  - Validate file paths and confirm image existence before packaging.
- **File Management:**
  - Clean up temporary files if they are no longer needed after packaging.
  - Consider parameterizing output paths and file names to integrate with your existing configuration management.

---

## SUM LEVEL4: Summary , Key Takeaways

☐ Summarizing Text with Claude: A Practical Python Guide

For Summarize and Key Takeaway use the same method but different prompts

(Chunk strategy or long content LLM model window).

## Handling Long Articles for AI Text Summarization

### Chunking Strategy (Most Common Approach)

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("deepseek-ai/deepseek-v3")

def chunk_text(text, chunk_size=3000):
    tokens = tokenizer.encode(text)
    chunks = [tokens[i:i + chunk_size] for i in range(0, len(tokens), chunk_size)]
    return [tokenizer.decode(chunk) for chunk in chunks]

# Summarize each chunk then combine
article_chunks = chunk_text(long_article)
chunk_summaries = [summarize_text(chunk) for chunk in article_chunks]
final_summary = summarize_text("\n".join(chunk_summaries))
```

### SUM LEVEL5: Enable Position, Testing

- Add options into `main.py` or configuration for position of **YouTube Video** (conclusion , random between paragraphs in the center), **summary** (all the time in the beginning of article - ADD turn on and turn off , **key takeaways**(Random inject into article, in the beginning, after FAQ and Conclusion - in the end of article).