RAG (Retrieval-Augmented Generation) application with multi-format outputs and agentic behavior, we can break it into modular Python workflows. Below is a detailed **step-by-step implementation plan**:

Step 1: Set Up the Environment

Install Required Libraries

Ensure the necessary libraries for APIs, text processing, image generation, video creation, and database management are installed:

```
pip install openai pinecone requests flask fastapi pillow transformers synthetize pytorch
```

Step 2: Code Implementation

1. User Input (Prompt)

Create an input handler that takes the user's content type, tone, and prompt.

```
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/generate', methods=['POST'])
def handle_prompt():
    user_input = request.json
    prompt = user_input['prompt']
    tone = user_input.get('tone', 'formal')
    content_type = user_input.get('content_type', 'text')
    platform = user_input.get('platform', 'LinkedIn')
```

```
    return jsonify({
        "message": "Processing...",
        "prompt": prompt,
        "tone": tone,
        "content_type": content_type,
        "platform": platform
    })
```

2. Document Retrieval (RAG Core)

Retrieve the most relevant documents using news APIs like Bing or Google News and apply filters.

```
import requests

def retrieve_articles(query, date_filter=None):
    api_key = "YOUR_BING_NEWS_API_KEY"
    endpoint = f"https://api.bing.microsoft.com/v7.0/news/search?q={query}"
    headers = {"Ocp-Apim-Subscription-Key": api_key}

    response = requests.get(endpoint, headers=headers).json()
    articles = []

    for article in response.get('value', []):
        if date_filter and article['datePublished'] < date_filter:
            continue
        articles.append({
```

```
            "title": article['name'],
            "url": article['url'],
            "description": article['description']
        })

    return articles
```

3. Summarization & Tone Adjustment

Use GPT-4 or similar LLMs to summarize retrieved articles and adjust the tone.

```
import openai

openai.api_key = "YOUR_OPENAI_API_KEY"

def summarize_and_adjust_tone(articles, tone):
    summaries = []
    for article in articles:
        content = f"Summarize the following in a {tone} tone:\n{article['description']}"
        response = openai.ChatCompletion.create(
            model="gpt-4",
            messages=[{"role": "user", "content": content}]
        )
        summaries.append({
            "summary": response.choices[0].message['content'],
            "source": article['url']
```

```
    })
  return summaries
```

4. Multi-Format Post Generation

Generate text, images, memes, and videos dynamically.

Text Generation

```
def generate_text(summary, tone, platform):
  prompt = f"Create a {tone} {platform}-friendly post: {summary}"
  response = openai.ChatCompletion.create(
    model="gpt-4",
    messages=[{"role": "user", "content": prompt}]
  )
  return response.choices[0].message['content']
```

Image Generation

```
def generate_image(description):
  response = openai.Image.create(
    prompt=description,
    n=1,
    size="1024x1024"
  )
  return response['data'][0]['url']
```

Meme Generation

```
from PIL import Image, ImageDraw, ImageFont

def generate_meme(template_path, top_text, bottom_text):
    img = Image.open(template_path)
    draw = ImageDraw.Draw(img)
    font = ImageFont.truetype("arial.ttf", 36)

    # Add top text
    draw.text((50, 50), top_text, font=font, fill="white")
    # Add bottom text
    draw.text((50, img.height - 100), bottom_text, font=font, fill="white")

    img.save("meme_output.jpg")
    return "meme_output.jpg"
```

Video Generation

Integrate video synthesis tools like Synthesia or Pictory via their API.

5. Citation & Transparency

```
def add_citations(summaries):
    return [
        f"{summary['summary']}\nSource: {summary['source']}"
```

```
        for summary in summaries
    ]
```

6. Dynamic Display

Use a **Flask/React.js** landing page to preview outputs.

```
@app.route('/preview', methods=['GET'])
def preview_content():
    content = {
        "text": "Generated text goes here",
        "image_url": "Generated image URL goes here",
        "meme_path": "meme_output.jpg",
        "video_url": "Generated video URL goes here"
    }
    return jsonify(content)
```

7. Interactive Refinements

Allow users to modify the generated content dynamically.

Example Endpoint for Refinements

```
@app.route('/refine', methods=['POST'])
def refine_content():
    user_input = request.json
    content = user_input['content']
```

```
    refinement = user_input['refinement']

    prompt = f"Refine the following content with this instruction: {refinement}\n\n{content}"
    response = openai.ChatCompletion.create(
        model="gpt-4",
        messages=[{"role": "user", "content": prompt}]
    )
    return jsonify({"refined_content": response.choices[0].message['content']})
```

Key Features of the System

1. **Autonomous Prioritization**: The RAG system dynamically adjusts sources and tone based on user preferences.
2. **Interactive Refinements**: Users can iteratively refine results for better quality.
3. **Multi-Format Outputs**: Supports text, images, memes, and video content tailored for different platforms.

Steps to Make the RAG Application Production-Ready

1. Refine the Code for Robustness

Error Handling

Add error handling for every API call, file operation, and database query. For example:

```
try:
    response = requests.get(endpoint, headers=headers)
    response.raise_for_status()
```

```
except requests.exceptions.RequestException as e:
    return {"error": f"API request failed: {e}"}
```

Validation

Validate user input to prevent invalid data or malicious commands:

```
from flask import abort
if not prompt or not isinstance(prompt, str):
    abort(400, "Invalid prompt provided.")
```

2. Security Enhancements

API Key Management

- Use environment variables to store sensitive API keys.
- Avoid hardcoding secrets in your code.

```
import os
api_key = os.getenv("BING_NEWS_API_KEY")
```

Rate Limiting

Implement rate limiting to prevent abuse of your endpoints using tools like Flask-Limiter:

```
pip install flask-limiter
```

```
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address
```

```
limiter = Limiter(get_remote_address, app=app, default_limits=["200 per day", "50 per hour"])
```

Prevent Injection Attacks

Sanitize all inputs to avoid SQL injection, prompt injection, and XSS attacks.

3. Scalability

Use a Production-Ready Server

Deploy the Flask app using **Gunicorn** or **Uvicorn** with a reverse proxy (e.g., Nginx).

```
gunicorn -w 4 -b 0.0.0.0:8000 app:app
```

Asynchronous Processing

For tasks like retrieving articles, generating summaries, and creating images, use an asynchronous task queue (e.g., **Celery** with **Redis**).

4. Optimize Performance

Batch Processing

Fetch articles and process summaries in batches to reduce latency.

Caching

Cache frequent queries and results using **Redis** or **Memcached** to reduce API calls.

5. Database Integration

Use a database for storing user inputs, generated content, and logs. For a production app:

- Use **PostgreSQL** or **MongoDB**.
- Add schemas for structured data storage.

Example with SQLAlchemy:

```
from flask_sqlalchemy import SQLAlchemy

app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://user:password@localhost/dbname'
db = SQLAlchemy(app)

class GeneratedContent(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_prompt = db.Column(db.String(500))
    content_type = db.Column(db.String(50))
    generated_text = db.Column(db.Text)
    image_url = db.Column(db.String(200))
    created_at = db.Column(db.DateTime, default=datetime.utcnow)

db.create_all()
```

Note:

vector databases can be used to store proprietary or private data (such as internal documents, research, product information, or any structured/unstructured text data). This stored data can then be used as **context** for answering user queries with the help of a **Large Language Model (LLM)**.

6. Logging and Monitoring

Logging

Log important events and errors using Python's logging library:

```
import logging

logging.basicConfig(level=logging.INFO)
logging.info("Application started.")
logging.error("Failed to fetch articles.")
```

Monitoring

Use monitoring tools like **Prometheus**, **Grafana**, or **New Relic** to track system performance.

7. **Deployment**

Cloud Hosting

Deploy the app on cloud platforms like **AWS**, **Google Cloud Platform (GCP)**, or **Azure**.

Containerization

Use Docker to containerize the application for portability and easier deployment:

Dockerfile:

```
FROM python:3.9-slim

WORKDIR /app
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
```

```
COPY . .

CMD ["gunicorn", "-w", "4", "-b", "0.0.0.0:8000", "app:app"]
```

CI/CD Pipeline

Set up CI/CD pipelines using tools like **GitHub Actions**, **Jenkins**, or **GitLab CI/CD**.

8. Proactive Enhancements

Multi-Turn Interaction

Enable iterative refinements by storing user sessions using Flask-Session or Redis.

Proactive Content Suggestions

Incorporate trending topics from platforms like Twitter Trends API or Google Trends.

Testing

Add comprehensive tests (unit, integration, and end-to-end) using **pytest**.

```
import pytest

def test_prompt_processing():
    response = app.test_client().post('/generate', json={"prompt": "Test", "tone": "funny"})
    assert response.status_code == 200
    assert "Processing" in response.json['message']
```

Key Aspects:

Production readiness requires these additional measures:

1. **Scalability (Asynchronous Tasks, Caching)**
2. **Security (Key Management, Validation)**
3. **Robustness (Error Handling, Logging)**
4. **Deployment (Cloud Hosting, CI/CD, Monitoring)**

Once these optimizations are in place, the application will be **reliable, scalable, and secure for production**.

RAG Workflow: