软件源代码

```
--- File: src/memory_manager.py ---
import json
import os
from datetime import datetime, timedelta
from typing import List, Dict
MEMORY_FILE = os.path.join(os.path.dirname(os.path.dirname(os.path.abspath(__file__))), "data", "history.json")
class MemoryManager:
    def __init__(self):
        self.memory_file = MEMORY_FILE
        self._ensure_data_dir()
    def _ensure_data_dir(self):
        """Ensure data directory exists"""
        dirname = os.path.dirname(self.memory_file)
        if not os.path.exists(dirname):
            os.makedirs(dirname)
        if not os.path.exists(self.memory_file):
            with open(self.memory_file, 'w', encoding='utf-8') as f:
                json.dump([], f)
    def load_history(self, days: int = 7) -&gt; List[Dict]:
        """
        Load summary history from the last N days.
        """
        try:
            with open(self.memory_file, 'r', encoding='utf-8') as f:
                history = json.load(f)
            cutoff = datetime.now() - timedelta(days=days)
            recent_history = []
            for entry in history:
                try:
                    entry_date = datetime.strptime(entry['date'], "%Y-%m-%d")
                    if entry_date &gt;= cutoff:
                        recent_history.append(entry)
                except ValueError:
                    continue
            return recent_history
        except Exception as e:
            print(f" Error loading memory: {e}")
            return []
    def save_summary(self, summary_data: Dict):
        """
        Save today's summary to history.
        """
        if not summary_data:
            return
        today = datetime.now().strftime("%Y-%m-%d")
        record = {
            "date": today,
            "intro": summary_data.get("intro", ""),
            "top_stories": [
                {
                    "title": s["title"],
                    "summary": s["summary"]
                } for s in summary_data.get("top_stories", [])
            ]
        }
        try:
            with open(self.memory_file, 'r', encoding='utf-8') as f:
                history = json.load(f)
            history = [h for h in history if h['date'] != today]
            history.append(record)
            if len(history) &gt; 30:
                history = history[-30:]
            with open(self.memory_file, 'w', encoding='utf-8') as f:
                json.dump(history, f, ensure_ascii=False, indent=2)
            print(f" Summary saved to memory ({today})")
        except Exception as e:
            print(f" Failed to save memory: {e}")
```

```
    def get_context_string(self, days: int = 3) -&gt; str:
        """
        Format recent history as a string for LLM context.
        """
        recent = self.load_history(days)
        if not recent:
            return "No recent history available."
        context = "Recent AI News Context:\n"
        for entry in recent:
            context += f"--- {entry['date']} ---\n"
            context += f"Overview: {entry['intro']}\n"
            for story in entry['top_stories']:
                context += f"- {story['title']}: {story['summary']}\n"
            context += "\n"
        return context
if __name__ == "__main__":
    mm = MemoryManager()
    print(mm.get_context_string())


--- File: src/config.py ---
import os
from dotenv import load_dotenv
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
load_dotenv(os.path.join(BASE_DIR, ".env"))
RSS_FEEDS = {
    "OpenAI Blog": "https://openai.com/blog/rss.xml",
    "Google DeepMind": "https://deepmind.google/blog/rss.xml",
    "Microsoft Research": "https://www.microsoft.com/en-us/research/feed/",
    "Hugging Face Blog": "https://huggingface.co/blog/feed.xml",
    "TechCrunch AI": "https://techcrunch.com/category/artificial-intelligence/feed/",
    "The Verge AI": "https://www.theverge.com/rss/ai/index.xml",
    "MIT Technology Review AI": "https://www.technologyreview.com/topic/artificial-intelligence/feed",
    "Reddit Machine Learning": "https://www.reddit.com/r/MachineLearning/.rss",
    "机器之心": "https://www.jiqizhixin.com/rss",
    "量子位": "https://www.qbitai.com/feed",
}
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
OPENAI_BASE_URL = os.getenv("OPENAI_BASE_URL", "https://api.openai.com/v1")
LLM_MODEL = os.getenv("LLM_MODEL", "gpt-4o")
SMTP_SERVER = os.getenv("SMTP_SERVER", "smtp.gmail.com")
SMTP_PORT = int(os.getenv("SMTP_PORT", "587"))
SMTP_USER = os.getenv("SMTP_USER")
SMTP_PASSWORD = os.getenv("SMTP_PASSWORD")
EMAIL_RECIPIENTS = os.getenv("EMAIL_RECIPIENTS", "").split(",")
REPORT_OUTPUT_DIR = os.path.join(BASE_DIR, "output")
TEMPLATE_DIR = os.path.join(BASE_DIR, "templates")
TOKEN_SAVING_MODE = os.getenv("TOKEN_SAVING_MODE", "true").lower() == "true"


--- File: src/reporter.py ---
import os
from jinja2 import Environment, FileSystemLoader
from datetime import datetime
from .config import REPORT_OUTPUT_DIR, TEMPLATE_DIR
class Reporter:
    def __init__(self):
        print(f"DEBUG: TEMPLATE_DIR = {TEMPLATE_DIR}")
        if not os.path.exists(TEMPLATE_DIR):
            print(f"ERROR: Template directory does not exist: {TEMPLATE_DIR}")
        self.env = Environment(loader=FileSystemLoader(TEMPLATE_DIR))
        try:
            self.template = self.env.get_template("report_template.html")
        except Exception as e:
            print(f"ERROR: Could not load template 'report_template.html' from {TEMPLATE_DIR}")
            print(f"Available files: {os.listdir(TEMPLATE_DIR) if os.path.exists(TEMPLATE_DIR) else 'Dir not found'}")
            raise e
        if not os.path.exists(REPORT_OUTPUT_DIR):
            os.makedirs(REPORT_OUTPUT_DIR)
    def generate_report(self, data: dict):
        """
        Generate HTML report from data
```

```
    """
    if not data:
        print("No data to report.")
        return
    current_time = datetime.now()
    now = current_time.strftime("%Y-%m-%d %H:%M:%S")
    try:
        html_content = self.template.render(data=data, generated_at=now)
    except Exception as e:
        print(f"Error rendering template: {e}")
        return
    filename = f"ai_news_report_{current_time.strftime('%Y%m%d_%H%M%S')}.html"
    filepath = os.path.join(REPORT_OUTPUT_DIR, filename)
    with open(filepath, "w", encoding="utf-8") as f:
        f.write(html_content)
    print(f"Report generated successfully: {filepath}")
    return filepath
def generate_wechat_html(self, data: dict):
    """
    Generate WeChat-optimized HTML report
    """
    if not data:
        return None
    try:
        template = self.env.get_template("wechat_template.html")
        context = {
            "date": datetime.now().strftime("%Y-%m-%d"),
            "summary_intro": data.get("intro", ""),
            "hot_news": [],
            "other_news": []
        }
        for item in data.get("top_stories", []):
            context["hot_news"].append({
                "title": item.get("title"),
                "source": item.get("source"),
                "image": item.get("image_url") or "", # Fallback logic in template?
                "one_sentence_summary": item.get("summary"),
                "key_points": item.get("key_points", []),
                "insight": item.get("impact", "")
            })
        for cat, items in data.get("categories", {}).items():
            for item in items:
                context["other_news"].append({
                    "title": item.get("title"),
                    "one_sentence_summary": item.get("summary"),
                    "source": item.get("source")
                })
        html_content = template.render(**context)
        filename = f"wechat_post_{datetime.now().strftime('%Y%m%d_%H%M%S')}.html"
        filepath = os.path.join(REPORT_OUTPUT_DIR, filename)
        with open(filepath, "w", encoding="utf-8") as f:
            f.write(html_content)
        print(f" WeChat HTML ready: {filepath}")
        return filepath
    except Exception as e:
        print(f"Error generating WeChat HTML: {e}")
        return None
def generate_markdown(self, data: dict):
    """
    Generate Markdown report (optional, for simple text output)
    """
    if not data:
        return ""
    md = f"# {data.get('title', 'AI News Report')}\n\n"
    md += f"**{data.get('intro', '')}**\n\n"
    md += "##  Top Stories\n"
    for story in data.get('top_stories', []):
        md += f"### [{story['title']}]({story['link']})\n"
        md += f"**Source:** {story['source']}\n\n"
        md += f"{story['summary']}\n\n"
        if 'impact' in story:
            md += f"&gt;  {story['impact']}\n\n"
```

```
        for category, stories in data.get('categories', {}).items():
            if stories:
                md += f"## {category}\n"
                for story in stories:
                    md += f"- **[{story['title']}]({story['link']})** ({story['source']}): {story['summary']}\n"
                md += "\n"
        md += "\n---\n"
        md += "### 免责声明\n"
        md += "本报告由 AI 自动生成，内容仅供参考。投资者应自行承担风险。本文不构成任何投资建议。\n"
        return md
    def generate_wechat_markdown(self, data: dict):
        """
        Generate WeChat Official Account friendly Markdown
        """
        if not data:
            return ""
        today = datetime.now().strftime("%Y-%m-%d")
        md = f"# AI 每日早报 ({today})\n\n"
        md += f"{data.get('intro', '')}\n\n"
        md += "---\n\n"
        md += "## 今日热点\n\n"
        for i, story in enumerate(data.get('top_stories', []), 1):
            title = story['title']
            md += f"### {i}. {title}\n"
            md += f"**来源**: {story['source']}\n\n"
            md += f"{story['summary']}\n\n"
            if story.get('key_points'):
                md += "**核心要点**:\n"
                for point in story['key_points']:
                    md += f"- {point}\n"
                md += "\n"
            if 'impact' in story:
                md += f"&gt; **深度洞察**: {story['impact']}\n\n"
            md += f" [原文链接]({story['link']})\n\n"
        for category, stories in data.get('categories', {}).items():
            if stories:
                md += f"## {category}\n\n"
                for story in stories:
                    md += f"- **{story['title']}**\n"
                    md += f" {story['summary']} ([{story['source']}]({story['link']}))\n"
                md += "\n"
        md += "---\n"
        md += "*本报告由 AI Agent 自动生成，内容仅供参考。*\n"
        md += "*免责声明：本文不构成任何投资建议，请独立判断。*\n" # Disclaimer
        return md
if __name__ == "__main__":
    mock_data = {
        "title": "AI Daily Test",
        "intro": "Nothing much happened today.",
        "top_stories": [
            {"title": "Test Story 1", "link": "#", "source": "Test", "summary": "This is a test summary.", "impact": "Huge impact."}
        ],
        "categories": {
            "Other": [{"title": "Small Story", "link": "#", "source": "Test", "summary": "Small summary."}]
        }
    }
    reporter = Reporter()
    reporter.generate_report(mock_data)


--- File: src/preferences.py ---
USER_INTERESTS = [
    "Large Language Models (LLM)",
    "Generative AI",
    "AI Agents",
    "OpenAI",
    "DeepMind",
    "Transformer Architecture",
    "RAG (Retrieval-Augmented Generation)",
    "Autonomous Systems",
    "AI Coding Assistants",
    "AI in Healthcare"
```

```
]
USER_DISLIKES = [
    "Crypto",
    "Blockchain",
    "NFT",
    "Metaverse Hype",
    "Web3"
]
LANGUAGE = "Chinese"  # Output language


--- File: src/dashboard.py ---
import streamlit as st
import sys
import os
import json
import threading
from datetime import datetime, timedelta
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from src.main import run_daily_job
from src.memory_manager import MemoryManager
from src.config import RSS_FEEDS, EMAIL_RECIPIENTS
from src.preferences import USER_INTERESTS, USER_DISLIKES
from src.deep_research import DeepResearchFetcher
from src.summarizer import NewsSummarizer
st.set_page_config(
    page_title="AI Daily Intelligence",
    page_icon="",
    layout="wide",
    initial_sidebar_state="expanded"
)
st.markdown("""
&lt;style&gt;
    .stButton&gt;button {
        width: 100%;
        border-radius: 5px;
        height: 3em;
        font-weight: bold;
    }
    .metric-card {
        background-color: #f0f2f6;
        padding: 20px;
        border-radius: 10px;
        text-align: center;
        box-shadow: 0 2px 4px rgba(0,0,0,0.05);
    }
    .report-frame {
        border: 1px solid #e0e0e0;
        border-radius: 8px;
        box-shadow: 0 4px 6px rgba(0,0,0,0.1);
    }
    .premium-badge {
        background-color: #FFD700;
        color: #000;
        padding: 2px 6px;
        border-radius: 4px;
        font-size: 0.8em;
        font-weight: bold;
        margin-left: 5px;
    }
&lt;/style&gt;
""", unsafe_allow_html=True)
def run_agent_async():
    """Run the agent in a separate thread"""
    with st.spinner(" Agent is running... This may take 1-2 minutes."):
        try:
            run_daily_job(hours=24)
            st.success(" Agent finished successfully! Refreshing data...")
            st.rerun()
        except Exception as e:
            st.error(f" Error: {e}")
def generate_deep_dive(topic):
```

```
    """Run deep research and generation"""
    status_text = st.empty()
    progress_bar = st.progress(0)
    try:
        status_text.text(f" Searching web for '{topic}'...")
        researcher = DeepResearchFetcher()
        progress_bar.progress(20)
        data = researcher.research_topic(topic)
        if not data:
            st.error("No data found.")
            return
        status_text.text(f" Reading {len(data)} articles...")
        progress_bar.progress(50)
        status_text.text(" Synthesizing deep report...")
        summarizer = NewsSummarizer()
        report_md = summarizer.generate_deep_report(topic, data)
        progress_bar.progress(90)
        status_text.text(" Done!")
        progress_bar.progress(100)
        return report_md
    except Exception as e:
        st.error(f"Error: {e}")
        return None
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
report_dir = os.path.join(BASE_DIR, "output")
mm = MemoryManager()
history = mm.load_history(days=30)
total_reports = len(history)
total_stories = sum(len(h.get('top_stories', [])) for h in history)
last_run = history[-1]['date'] if history else "N/A"
with st.sidebar:
    st.title("AI Intelligence")
    st.caption(f"v2.3 | Last Run: {last_run}")
    st.markdown("---")
    st.subheader(" Actions")
    if st.button(" Trigger New Run", type="primary"):
        run_agent_async()
    st.markdown("---")
    st.subheader(" Configuration")
    with st.expander(" Data Sources", expanded=False):
        st.write(f"Active Feeds: **{len(RSS_FEEDS)}**")
        for name, url in RSS_FEEDS.items():
            st.text_input(name, value=url, disabled=True)
    with st.expander(" Interest Profile", expanded=False):
        st.write("**Focus Areas:**")
        st.info(", ".join(USER_INTERESTS))
        st.write("**Exclusions:**")
        st.warning(", ".join(USER_DISLIKES))
    with st.expander(" Delivery", expanded=False):
        st.write(f"Recipients: {', '.join(EMAIL_RECIPIENTS)}")
st.title(" Intelligence Dashboard")
col1, col2, col3, col4 = st.columns(4)
with col1:
    st.metric("Total Reports", total_reports, "+1" if total_reports &gt; 0 else "0")
with col2:
    st.metric("Stories Analyzed", total_stories, f"+{len(history[-1].get('top_stories', []))}" if history else "0")
with col3:
    st.metric("Active Feeds", len(RSS_FEEDS))
with col4:
    st.metric("System Status", "Online", delta_color="normal")
st.markdown("---")
tab1, tab2, tab3, tab4 = st.tabs([" Latest Report", " Deep Dive (Beta)", " Archive", " Knowledge Graph"])
with tab1:
    if os.path.exists(report_dir):
        files = sorted([f for f in os.listdir(report_dir) if f.endswith('.html')], reverse=True)
        if files:
            latest_file = files[0]
            file_path = os.path.join(report_dir, latest_file)
            col_a, col_b = st.columns([3, 1])
            with col_a:
                st.subheader(f" {latest_file}")
            with col_b:
```

```
            with open(file_path, "r") as f:
                html_content = f.read()
            st.download_button(
                label=" Download HTML",
                data=html_content,
                file_name=latest_file,
                mime="text/html",
                use_container_width=True
            )
        st.components.v1.html(html_content, height=800, scrolling=True)
    else:
        st.info("No reports found. Click 'Trigger New Run' to generate one.")
else:
    st.warning(f"Output directory not found: {report_dir}")
with tab2:
    st.markdown("###  Deep Dive Research")
    st.markdown("*Generate a comprehensive research report on any AI topic.* &lt;span class='premium-badge'&gt;PRO&lt;/span&g
    is_activated = st.session_state.get("is_pro_activated", False)
    if not is_activated:
        st.info(" This is a Premium feature. Please enter your activation code to unlock.")
        col_key, col_btn = st.columns([3, 1])
        with col_key:
            activation_code = st.text_input("Activation Code", type="password", placeholder="Enter code here...")
        with col_btn:
            st.write("") # Spacer
            st.write("")
            if st.button("Unlock "):
                if activation_code == "VIP888":  # Simple hardcoded check for demo
                    st.session_state["is_pro_activated"] = True
                    st.success(" Premium features unlocked!")
                    st.rerun()
                else:
                    st.error(" Invalid activation code.")
        st.markdown("---")
        st.markdown("####  Why Upgrade?")
        st.markdown("""
        - **Unlimited Deep Research**: Generate 3000+ word reports on any topic.
        - **Multi-Source Synthesis**: Aggregates info from 20+ search engines.
        - **Export to PDF/PPT**: Ready-to-present formats.
        """)
        st.image("https://via.placeholder.com/800x400.png?text=Deep+Dive+Report+Preview", caption="Sample Deep Dive Report")
    else:
        topic = st.text_input("Enter a topic (e.g., 'Agentic Workflow', 'DeepSeek Technical Architecture')", placeholder="What do you wa
        if st.button(" Generate Report", type="primary", disabled=not topic):
            report_md = generate_deep_dive(topic)
            if report_md:
                st.markdown("---")
                with st.expander(" Research Report", expanded=True):
                    st.markdown(report_md)
                    st.download_button(" Download Report (MD)", report_md, file_name=f"{topic}_report.md")
with tab3:
    st.subheader(" Historical Reports")
    if os.path.exists(report_dir):
        files = sorted([f for f in os.listdir(report_dir) if f.endswith('.html')], reverse=True)
        if files:
            col_list, col_preview = st.columns([1, 2])
            with col_list:
                selected_file = st.radio("Select Date", files, format_func=lambda x: x.replace("ai_news_report_", "").replace(".html", ""))
            with col_preview:
                if selected_file:
                    file_path = os.path.join(report_dir, selected_file)
                    with open(file_path, "r") as f:
                        html_content = f.read()
                    st.components.v1.html(html_content, height=600, scrolling=True)
        else:
            st.info("No historical reports found.")
    else:
        st.warning(f"Output directory not found: {report_dir}")
with tab4:
    st.subheader(" Memory Stream")
    if history:
        for entry in reversed(history):
```

```
            with st.expander(f" {entry['date']} - {entry['intro'][:50]}...", expanded=False):
                st.info(f"**Overview:** {entry['intro']}")
                st.markdown("#### Top Stories")
                for story in entry.get('top_stories', []):
                    st.markdown(f"- **{story['title']}**: {story['summary']}")
    else:
        st.info("No memory history available yet.")


--- File: src/content_factory.py ---
import os
import argparse
import time
from datetime import datetime
from src.deep_research import DeepResearchFetcher
from src.summarizer import NewsSummarizer
from src.topic_manager import TopicManager
from src.config import REPORT_OUTPUT_DIR
def generate_content_for_topic(topic: str, output_base_dir: str = REPORT_OUTPUT_DIR):
    """
    Automated Content Factory:
    Topic -&gt; Deep Research -&gt; Report -&gt; Viral PPT -&gt; TikTok Script
    """
    print(f"\n [START] Processing topic: {topic}")
    researcher = DeepResearchFetcher()
    summarizer = NewsSummarizer()
    safe_topic = "".join([c for c in topic if c.isalnum() or c in (' ', '-', '_')]).strip()[:50]
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    topic_dir = os.path.join(output_base_dir, f"{timestamp}_{safe_topic}")
    if not os.path.exists(topic_dir):
        os.makedirs(topic_dir)
    print("   [Phase 1] Deep Researching...")
    research_data = researcher.research_topic(topic)
    if not research_data:
        print(f"    No data found for '{topic}'. Skipping.")
        return
    print("   [Phase 2] Generating Base Report...")
    report_content = summarizer.generate_deep_report(topic, research_data)
    report_path = os.path.join(topic_dir, "1_report.md")
    with open(report_path, "w", encoding="utf-8") as f:
        f.write(report_content)
    print(f"    Report saved.")
    print("   [Phase 3] Generating Viral PPT...")
    ppt_content = summarizer.generate_marp_slides(topic, report_content, style="viral")
    ppt_path = os.path.join(topic_dir, "2_viral_slides.md")
    with open(ppt_path, "w", encoding="utf-8") as f:
        f.write(ppt_content)
    print(f"    PPT Code saved.")
    print("   [Phase 4] Generating TikTok Script...")
    script_content = summarizer.generate_video_script(topic, report_content, platform="tiktok")
    script_path = os.path.join(topic_dir, "3_tiktok_script.txt")
    with open(script_path, "w", encoding="utf-8") as f:
        f.write(script_content)
    print(f"    Video Script saved.")
    print(f" [DONE] Content ready in: {topic_dir}")
def run_daily_batch(limit: int = 10):
    """
    Fetch trending topics and run content factory for each.
    """
    print(f" Starting Daily Batch Run (Limit: {limit})")
    tm = TopicManager()
    topics = tm.get_trending_topics(count=limit)
    print(f" Today's Topics: {topics}")
    for i, topic in enumerate(topics, 1):
        print(f"\n----------------------------------------")
        print(f"Processing {i}/{len(topics)}: {topic}")
        try:
            generate_content_for_topic(topic)
        except Exception as e:
            print(f" Error processing '{topic}': {e}")
        time.sleep(2)
if __name__ == "__main__":
```

```
    parser = argparse.ArgumentParser(description="Viral Content Factory")
    parser.add_argument("--mode", type=str, default="single", choices=["single", "batch"], help="Mode: single or batch")
    parser.add_argument("--topic", type=str, help="Topic for single mode")
    parser.add_argument("--limit", type=int, default=10, help="Limit for batch mode")
    args = parser.parse_args()
    if args.mode == "batch":
        run_daily_batch(limit=args.limit)
    else:
        topic = args.topic if args.topic else "2025年春节消费降级现象与机会"
        generate_content_for_topic(topic)


--- File: src/summarizer.py ---
import json
import concurrent.futures
from typing import List, Dict
from openai import OpenAI
from .config import OPENAI_API_KEY, OPENAI_BASE_URL, LLM_MODEL, TOKEN_SAVING_MODE
from .preferences import USER_INTERESTS, USER_DISLIKES
from .memory_manager import MemoryManager
class NewsSummarizer:
    def __init__(self):
        if not OPENAI_API_KEY:
            raise ValueError("OPENAI_API_KEY not found in environment variables.")
        self.client = OpenAI(
            api_key=OPENAI_API_KEY,
            base_url=OPENAI_BASE_URL
        )
        self.model = LLM_MODEL
        self.memory = MemoryManager()
    def batch_filter_articles(self, news_items: List[Dict]) -&gt; List[Dict]:
        """
        [Filter Step] Use a single cheap LLM call to filter out irrelevant news by title.
        """
        if not news_items:
            return []
        print(f" [Token Saving] Batch filtering {len(news_items)} articles by title...")
        titles_text = ""
        for i, item in enumerate(news_items):
            titles_text += f"{i}. {item['title']} (Source: {item['source']})\n"
        prompt = f"""
请作为一名严格的 AI 新闻编辑，从以下列表中筛选出**真正重要**且**符合用户兴趣**的新闻。
用户兴趣: {", ".join(USER_INTERESTS)}
不感兴趣: {", ".join(USER_DISLIKES)}
筛选标准：
1. 必须是 AI 领域的**重大**进展、新模型发布、重要研究或商业大事件。
2. 剔除：教程类("How to")、过于细分的每日论文、无关的推广、重复的报道。
3. 严格控制数量，只保留最有价值的前 30%-50%。
新闻列表：
{titles_text}
请仅输出保留的新闻编号列表，格式如 JSON：
{{
  "keep_indices": [0, 2, 5, ...]
}}
"""
        try:
            response = self.client.chat.completions.create(
                model=self.model, # Can use a cheaper model here if available
                messages=[
                    {"role": "system", "content": "You are a strict news editor. Output JSON only."},
                    {"role": "user", "content": prompt}
                ],
                temperature=0.1,
                response_format={"type": "json_object"}
            )
            result = json.loads(response.choices[0].message.content)
            keep_indices = set(result.get("keep_indices", []))
            filtered_items = [item for i, item in enumerate(news_items) if i in keep_indices]
            print(f" Filtered down to {len(filtered_items)} items (from {len(news_items)})")
            return filtered_items
        except Exception as e:
            print(f" Filter failed, keeping all items: {e}")
```

```
            return news_items
    def analyze_single_article(self, item: Dict) -&gt; Dict:
        """
        [Map Step] Analyze a single article's full content to extract key insights.
        """
        content = item.get('full_content', item.get('summary', ''))[:3000]
        prompt = f"""
请分析以下 AI 新闻内容，提取关键信息。
用户偏好（请据此调整 importance_score）：
- 重点关注: {', '.join(USER_INTERESTS)}
- 忽略或低分: {', '.join(USER_DISLIKES)}
标题：{item['title']}
来源：{item['source']}
内容：
{content}
请输出 JSON 格式（不要 Markdown 标记）：
{{
  "title_zh": "中文标题",
  "summary_zh": "中文摘要（50字以内）",
  "key_points": ["关键点1", "关键点2", "关键点3"],
  "category": "模型/行业/学术/应用/其他",
  "importance_score": 1-10 (符合用户兴趣的给高分，无关的给低分),
  "impact_analysis": "一句话分析其对行业的影响"
}}
"""
        try:
            response = self.client.chat.completions.create(
                model=self.model,
                messages=[
                    {"role": "system", "content": "You are an AI analyst. Output raw JSON only."},
                    {"role": "user", "content": prompt}
                ],
                temperature=0.3,
                response_format={"type": "json_object"}
            )
            analysis = json.loads(response.choices[0].message.content)
            item.update(analysis)
            return item
        except Exception as e:
            return item
    def summarize(self, news_items: List[Dict]) -&gt; Dict:
        """
        [Reduce Step] Aggregate analyzed items into a final report.
        """
        if not news_items:
            return {}
        if TOKEN_SAVING_MODE and len(news_items) &gt; 5:
            news_items = self.batch_filter_articles(news_items)
        print(f" Analyzing {len(news_items)} articles in depth...")
        analyzed_items = []
        with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
            futures = [executor.submit(self.analyze_single_article, item) for item in news_items]
            for future in concurrent.futures.as_completed(futures):
                try:
                    analyzed_items.append(future.result())
                except Exception:
                    pass
        valid_items = [i for i in analyzed_items if i.get('importance_score', 0) &gt;= 4]
        valid_items.sort(key=lambda x: x.get('importance_score', 0), reverse=True)
        top_stories = []
        categories = {
            "模型与技术": [],
            "行业与商业": [],
            "学术与研究": [],
            "工具与应用": [],
            "其他": []
        }
        for i, item in enumerate(valid_items):
            story_data = {
                "title": item.get('title_zh', item['title']),
                "summary": item.get('summary_zh', item['summary']),
                "source": item['source'],
```

```
                "link": item['link'],
                "image": item.get('image'),
                "impact": item.get('impact_analysis', ''),
                "key_points": item.get('key_points', [])
            }
            if i &lt; 5:
                top_stories.append(story_data)
            else:
                cat = item.get('category', '其他')
                target_cat = "其他"
                if "模型" in cat or "技术" in cat: target_cat = "模型与技术"
                elif "行业" in cat or "商业" in cat: target_cat = "行业与商业"
                elif "学术" in cat or "研究" in cat: target_cat = "学术与研究"
                elif "应用" in cat or "工具" in cat: target_cat = "工具与应用"
                categories[target_cat].append(story_data)
        history_context = self.memory.get_context_string(days=3)
        intro_prompt = f"""
请根据以下今日头条新闻，结合过去几天的历史背景，生成一句简短的今日 AI 行业动态综述。
历史背景（仅作参考，无需强行关联）：
{history_context}
今日头条：
{[t['title'] for t in top_stories]}
要求：简练、专业，突出连续性（如果有）。
"""
        try:
            intro_resp = self.client.chat.completions.create(
                model=self.model,
                messages=[{"role": "user", "content": intro_prompt}]
            )
            intro_text = intro_resp.choices[0].message.content.strip()
        except:
            intro_text = "今日 AI 领域有多项重要更新。"
        final_summary = {
            "title": f"AI Daily Insight ({valid_items[0]['published'][:10] if valid_items else ''})",
            "intro": intro_text,
            "top_stories": top_stories,
            "categories": categories
        }
        self.memory.save_summary(final_summary)
        return final_summary
    def generate_deep_report(self, topic: str, research_data: list) -&gt; str:
        """
        Generate a long-form deep dive report based on research data.
        """
        context = ""
        for i, item in enumerate(research_data):
            context += f"--- Source {i+1}: {item['title']} ---\n"
            context += f"{item['full_content'][:2000]}\n\n"
        prompt = f"""
请根据以下收集到的资料，撰写一份关于 "{topic}" 的深度行业研报。
资料库：
{context}
要求：
1. 结构清晰：包含【背景与现状】、【核心技术/事件解析】、【市场竞争格局】、【未来趋势预测】四个章节。
2. 深度分析：不要简单的堆砌资料，要进行逻辑串联和观点提炼。
3. 数据支撑：引用资料中的关键数据。
4. 篇幅：1500字左右。
5. 格式：Markdown。
"""
        try:
            print(f" Generating deep dive report for '{topic}'...")
            response = self.client.chat.completions.create(
                model=self.model,
                messages=[
                    {"role": "system", "content": "You are a senior AI industry analyst."},
                    {"role": "user", "content": prompt}
                ],
                temperature=0.4
            )
            return response.choices[0].message.content
        except Exception as e:
            return f"Error generating report: {e}"
```

```
    def generate_marp_slides(self, topic: str, report_content: str, style: str = "academic") -&gt; str:
        """
        Convert a deep dive report into a Marp-formatted slide deck.
        Style options: 'academic' (default, strict), 'viral' (for social media/business).
        """
        if style == "viral":
            theme_instruction = "theme: uncover" # A more modern/visual theme
            style_instruction = """
            3. **风格**：
               - 极具视觉冲击力，适合社交媒体传播。
               - 标题要夸张、吸引眼球（Clickbait风格）。
               - 每一页字数要少，重点突出金句。
               - **不要**包含 Methodology 页。
               - **不要**写汇报人名字。
            """
        else:
            theme_instruction = "theme: gaia"
            style_instruction = """
            3. **风格**：专业、学术、极简。
            4. **特殊要求**：
               - 包含 "Methodology" 页（简述研究方法：PICO分析/广度搜索+深度综合）。
               - 封面页汇报人必须写：刘佳兴。
               - 禁止使用占位符，必须描述具体的图表内容。
            """
        prompt = f"""
请将以下深度研报内容转换为 Marp (Markdown Presentation Ecosystem) 格式的 PPT 代码。
研报主题：{topic}
研报内容：
{report_content[:3000]}... (截取部分内容)
要求：
1. **格式**：必须是标准的 Marp Markdown 格式。
  - 头部包含 `marp: true`, `{theme_instruction}`, `paginate: true`。
  - 每页幻灯片用 `---` 分隔。
2. **结构**：
  - 封面页：标题、副标题。
  - 目录页。
  - 正文页：提炼关键点，使用列表。
  - 结束页。
{style_instruction}
5. **语言**：中文。
输出示例：
---
marp: true
{theme_instruction}
paginate: true
---
---
...
"""
        try:
            print(f" Generating Marp slides for '{topic}' (Style: {style})...")
            response = self.client.chat.completions.create(
                model=self.model,
                messages=[
                    {"role": "system", "content": "You are a presentation expert skilled in Marp markdown."},
                    {"role": "user", "content": prompt}
                ],
                temperature=0.5 if style == "viral" else 0.3
            )
            return response.choices[0].message.content
        except Exception as e:
            return f"Error generating slides: {e}"
    def generate_video_script(self, topic: str, report_content: str, platform: str = "tiktok") -&gt; str:
        """
        Generate a short video script based on the report.
        """
        prompt = f"""
请将以下深度研报内容改编为一个适合 {platform} (抖音/TikTok/小红书) 的短视频口播文案。
主题：{topic}
参考内容：
{report_content[:2000]}
要求：
```

1. **黄金前三秒**：开头必须有一个极其抓人的钩子（Hook），引发好奇或焦虑。
2. **口语化**：完全大白话，不要书面语，多用“家人们”、“注意看”、“绝了”等连接词（视平台风格而定）。
3. **分镜描述**：左侧写【画面建议】，右侧写【口播文案】。
4. **时长**：控制在 60-90 秒（约 200-300 字）。
5. **结尾**：引导关注/点赞/评论。
输出格式示例：
【画面：主播震惊脸，背景放相关新闻截图】
文案：天呐，这件事如果真的发生了，我们所有人的钱袋子都要缩水！
【画面：展示数据图表，箭头指向关键下降趋势】
文案：大家看这张图，短短三天...
"""
        try:
            print(f" Generating video script for '{topic}'...")
            response = self.client.chat.completions.create(
                model=self.model,
                messages=[
                    {"role": "system", "content": "You are a viral content creator."},
                    {"role": "user", "content": prompt}
                ],
                temperature=0.7
)
            return response.choices[0].message.content
        except Exception as e:
            return f"Error generating script: {e}"

--- File: src/scheduler.py ---
import schedule
import time
import sys
import os
from datetime import datetime
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from src.main import run_daily_job
def job():
    print(f"\n Scheduled job started at {datetime.now()}")
    try:
        run_daily_job(hours=24)
    except Exception as e:
        print(f" Job failed: {e}")
    print(f" Job finished at {datetime.now()}\n")
if __name__ == "__main__":
    print(" AI News Agent Scheduler is running...")
    print(" Schedule: Daily at 08:00")
    schedule.every().day.at("08:00").do(job)
    while True:
        schedule.run_pending()
        time.sleep(60)

--- File: src/email_sender.py ---
import smtplib
import os
import time
import resend
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from datetime import datetime
from .config import SMTP_SERVER, SMTP_PORT, SMTP_USER, SMTP_PASSWORD, EMAIL_RECIPIENTS, BASE_DIR
class EmailSender:
    def __init__(self):
        self.server = SMTP_SERVER
        self.port = SMTP_PORT
        self.user = SMTP_USER
        self.password = SMTP_PASSWORD
        self.resend_key = os.getenv("RESEND_API_KEY")
        if self.resend_key:
            resend.api_key = self.resend_key
        self.recipients = self._load_subscribers()
    def _load_subscribers(self):
        """Load emails from subscribers.txt if exists, else from .env"""
        subs_file = os.path.join(BASE_DIR, "subscribers.txt")

```
        emails = []
        if os.path.exists(subs_file):
            try:
                with open(subs_file, "r") as f:
                    emails = [line.strip() for line in f if line.strip() and "@" in line and not line.startswith("#")]
            except Exception as e:
                print(f"Error reading subscribers.txt: {e}")
        if not emails:
            emails = [r.strip() for r in EMAIL_RECIPIENTS if r.strip()]
        return list(set(emails)) # Dedup
    def send_report(self, html_content: str, title: str):
        """
        Send the HTML report via Resend (Preferred) or SMTP
        """
        if not self.recipients:
            print(" No email recipients configured. Skipping email.")
            return
        subject = f" {title} - {datetime.now().strftime('%Y-%m-%d')}"
        if self.resend_key:
            try:
                print(f" Sending via Resend API to {len(self.recipients)} recipients...")
                params = {
                    "from": "AI Daily Agent &lt;onboarding@resend.dev&gt;", # Default test domain
                    "to": ["delivered@resend.dev"], # Placeholder for 'To' field
                    "bcc": self.recipients,
                    "subject": subject,
                    "html": html_content
                }
                if self.user and "@" in self.user and "gmail" not in self.user and "qq" not in self.user:
                    params["from"] = f"AI Daily Agent &lt;{self.user}&gt;"
                r = resend.Emails.send(params)
                print(f" Email sent via Resend! ID: {r.get('id')}")
                return
            except Exception as e:
                print(f" Resend failed: {e}. Falling back to SMTP...")
        if not self.user or not self.password:
            print(" SMTP credentials not configured. Skipping email.")
            return
        msg = MIMEMultipart()
        msg['From'] = f"AI Daily Agent &lt;{self.user}&gt;"
        msg['Subject'] = subject
        msg.attach(MIMEText(html_content, 'html'))
        max_retries = 3
        for attempt in range(max_retries):
            try:
                print(f" Sending via SMTP to {len(self.recipients)} recipients (Attempt {attempt+1})...")
                with smtplib.SMTP(self.server, self.port) as server:
                    server.starttls()
                    server.login(self.user, self.password)
                    server.sendmail(self.user, self.recipients, msg.as_string())
                print(" Email sent successfully via SMTP!")
                return
            except Exception as e:
                print(f" Failed to send email: {e}")
                if attempt &lt; max_retries - 1:
                    time.sleep(5) # Wait before retry
                else:
                    print(" Max retries reached. Email sending failed.")
if __name__ == "__main__":
    sender = EmailSender()
    sender.send_report("&lt;h1&gt;Test Report&lt;/h1&gt;&lt;p&gt;This is a test.&lt;/p&gt;", "Test Subject")


--- File: src/topic_manager.py ---
import json
from datetime import datetime
from duckduckgo_search import DDGS
from openai import OpenAI
from .config import OPENAI_API_KEY, OPENAI_BASE_URL, LLM_MODEL
class TopicManager:
    def __init__(self):
        self.ddgs = DDGS()
```

```
        self.client = OpenAI(
            api_key=OPENAI_API_KEY,
            base_url=OPENAI_BASE_URL
        )
        self.model = LLM_MODEL
    def get_trending_topics(self, count: int = 10) -&gt; list:
        """
        Fetch trending topics from search and refine with LLM.
        """
        print(f" Fetching top {count} trending topics...")
        try:
            query = f"today's top news and trending topics in China {datetime.now().strftime('%Y-%m-%d')}"
            results = self.ddgs.text(query, max_results=10)
            if not results:
                print(" Search returned empty, using fallback trends.")
                return self._get_fallback_topics(count)
            raw_text = "\n".join([f"- {r['title']}: {r['body']}" for r in results])
        except Exception as e:
            print(f" Error fetching trends: {e}")
            return self._get_fallback_topics(count)
        prompt = f"""
请根据以下搜索结果，提炼出 {count} 个当下中国互联网最热门、最适合做短视频（有争议、有流量、有新意）的话题。
搜索结果：
{raw_text}
要求：
1. 话题要具体（例如：“iPhone 16 销量暴跌” 而不是 “科技新闻”）。
2. 去重。
3. 返回纯 JSON 数组格式，不要 Markdown。
示例：
["话题1", "话题2", ...]
"""
        try:
            response = self.client.chat.completions.create(
                model=self.model,
                messages=[
                    {"role": "system", "content": "You are a social media trend analyst. Output JSON only."},
                    {"role": "user", "content": prompt}
                ],
                temperature=0.7,
                response_format={"type": "json_object"}
            )
            content = response.choices[0].message.content
            data = json.loads(content)
            if isinstance(data, list):
                topics = data
            elif "topics" in data:
                topics = data["topics"]
            else:
                topics = next((v for v in data.values() if isinstance(v, list)), [])
            if not topics:
                return self._get_fallback_topics(count)
            return topics[:count]
        except Exception as e:
            print(f" Error extracting topics: {e}")
            return self._get_fallback_topics(count)
    def _get_fallback_topics(self, count: int) -&gt; list:
        """
        Fallback topics in case search fails.
        """
        defaults = [
            "2025春节档电影票房预测",
            "OpenAI o3 模型发布影响",
            "年轻人为何不爱换手机了",
            "新能源汽车价格战",
            "职场35岁危机真相",
            "预制菜进校园争议",
            "房地产市场最新政策解读",
            "大学生就业难现状",
            "董宇辉新号带货数据分析",
            "马斯克火星殖民计划更新"
        ]
        return defaults[:count]
```

--- File: src/main.py ---

```
import argparse
import os
import sys
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from src.fetcher import NewsFetcher
from src.full_content_fetcher import FullContentFetcher
from src.summarizer import NewsSummarizer
from src.reporter import Reporter
from src.email_sender import EmailSender
def run_daily_job(hours=24, send_email=False):
    print(" Starting Daily AI News Agent...")
    try:
        fetcher = NewsFetcher()
        news_items = fetcher.fetch_all(hours_back=hours)
        if not news_items:
            print("No news found in the specified time range.")
            return
    except Exception as e:
        print(f"Error fetching news: {e}")
        return
    try:
        content_fetcher = FullContentFetcher()
        news_items = content_fetcher.enrich_news_items(news_items)
    except Exception as e:
        print(f"Error enriching news content: {e}")
    try:
        print(" Analyzing and summarizing news (this may take a moment)...")
        summarizer = NewsSummarizer()
        summary_data = summarizer.summarize(news_items)
        if not summary_data:
            print("Failed to generate summary.")
            return
    except Exception as e:
        print(f"Error summarizing news: {e}")
        return
    try:
        print(" Generating report...")
        reporter = Reporter()
        report_path = reporter.generate_report(summary_data)
        with open(report_path, 'r', encoding='utf-8') as f:
            html_content = f.read()
        if send_email:
            print(" Sending email report...")
            email_sender = EmailSender()
            email_sender.send_report(html_content, summary_data.get('title', 'AI Daily News'))
        md_summary = reporter.generate_markdown(summary_data)
        wechat_md = reporter.generate_wechat_markdown(summary_data)
        wechat_html_path = reporter.generate_wechat_html(summary_data)
        print("\n" + "="*50)
        print(" 微信公众号版本 (可直接复制):")
        print(wechat_md)
        print("="*50 + "\n")
        if report_path:
            print(f" Report ready: {report_path}")
            if wechat_html_path:
                print(f" WeChat HTML ready: {wechat_html_path}")
            if sys.platform == 'darwin' and not send_email: # Don't popup if running in automation
                os.system(f"open '{report_path}'")
    except Exception as e:
        print(f"Error generating report: {e}")
def main():
    parser = argparse.ArgumentParser(description="Daily AI News Agent")
    parser.add_argument("--hours", type=int, default=24, help="Fetch news from the last N hours (default: 24)")
    parser.add_argument("--email", action="store_true", help="Send email report to subscribers")
    args = parser.parse_args()
    run_daily_job(args.hours, send_email=args.email)
if __name__ == "__main__":
    main()
```

--- File: src/fetcher.py ---

```
import feedparser
import time
import requests
import concurrent.futures
from datetime import datetime, timedelta
from typing import List, Dict
from .config import RSS_FEEDS
class NewsFetcher:
    def __init__(self):
        self.feeds = RSS_FEEDS
        self.headers = {
            'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/121.0.0.0 Safa
            'Accept': 'application/rss+xml, application/xml, application/atom+xml, text/xml;q=0.9, */*;q=0.8',
            'Referer': 'https://www.google.com/'
        }
    def parse_time(self, time_struct):
        """Convert time_struct to datetime object"""
        if not time_struct:
            return datetime.now()
        return datetime.fromtimestamp(time.mktime(time_struct))
    def fetch_feed(self, source_info):
        """Fetch single feed with error handling"""
        source_name, feed_url = source_info
        news_items = []
        try:
            response = requests.get(feed_url, headers=self.headers, timeout=15)
            response.encoding = 'utf-8'
            if response.status_code != 200:
                print(f" {source_name}: Status {response.status_code}")
                return []
            feed = feedparser.parse(response.text)
            cutoff_time = datetime.now() - timedelta(hours=self.hours_back)
            for entry in feed.entries:
                published_parsed = getattr(entry, 'published_parsed', None) or getattr(entry, 'updated_parsed', None)
                published_dt = self.parse_time(published_parsed)
                if published_dt &gt;= cutoff_time:
                    news_items.append({
                        "title": entry.title,
                        "link": entry.link,
                        "summary": getattr(entry, 'summary', getattr(entry, 'description', '')),
                        "source": source_name,
                        "published": published_dt.strftime("%Y-%m-%d %H:%M:%S"),
                        "raw_date": published_dt
                    })
            if news_items:
                print(f" {source_name}: Found {len(news_items)} items")
            else:
                print(f" {source_name}: No new items")
            return news_items
        except Exception as e:
            print(f" {source_name}: Error ({str(e)[:50]}...)")
            return []
    def fetch_all(self, hours_back: int = 24) -&gt; List[Dict]:
        """
        Fetch news from all configured RSS feeds in parallel
        """
        self.hours_back = hours_back
        all_news = []
        print(f" Fetching news from {len(self.feeds)} sources (last {hours_back} hours)...")
        with concurrent.futures.ThreadPoolExecutor(max_workers=10) as executor:
            future_to_source = {
                executor.submit(self.fetch_feed, (name, url)): name
                for name, url in self.feeds.items()
            }
            for future in concurrent.futures.as_completed(future_to_source):
                try:
                    items = future.result()
                    all_news.extend(items)
                except Exception as e:
                    print(f"Error processing feed result: {e}")
        all_news.sort(key=lambda x: x['raw_date'], reverse=True)
        for item in all_news:
```

```
            del item['raw_date']
        print(f" Total {len(all_news)} news items collected.")
        return all_news
if __name__ == "__main__":
    fetcher = NewsFetcher()
    news = fetcher.fetch_all(hours_back=48)


--- File: src/deep_research.py ---
from duckduckgo_search import DDGS
from .full_content_fetcher import FullContentFetcher
import concurrent.futures
class DeepResearchFetcher:
    def __init__(self):
        self.ddgs = DDGS()
        self.content_fetcher = FullContentFetcher()
    def search(self, query: str, max_results: int = 5) -&gt; list:
        """
        Search for a topic and return raw results
        """
        print(f" Searching web for: {query}...")
        results = []
        try:
            ddg_results = self.ddgs.text(query, max_results=max_results)
            if ddg_results:
                results.extend(ddg_results)
        except Exception as e:
            print(f"Error searching DDG: {e}")
        if not results:
            print(" Search failed or returned no results. Using Mock Data for Demo.")
            results = [
                {
                    "title": "2025春节消费新趋势：年轻人更爱“平替”",
                    "href": "https://example.com/news1",
                    "body": "今年春节，高端白酒和奢侈品销量下滑，而平价餐饮和周边游火爆。数据显示..."
                },
                {
                    "title": "消费降级下的商机：二手交易平台流量暴增",
                    "href": "https://example.com/news2",
                    "body": "闲鱼等平台发布报告称，春节期间闲置物品交易量同比增长 40%..."
                },
                {
                    "title": "从“买买买”到“体验至上”：2025春节消费心理变迁",
                    "href": "https://example.com/news3",
                    "body": "消费者不再盲目追求大牌，而是更看重情绪价值和实际体验..."
                }
            ]
        return results
    def research_topic(self, topic: str) -&gt; list:
        """
        Full research pipeline: Search -&gt; Fetch Content
        """
        search_results = self.search(topic, max_results=5)
        if not search_results:
            return []
        print(f" Reading {len(search_results)} articles for deep dive...")
        detailed_results = []
        with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
            future_to_result = {
                executor.submit(self.content_fetcher.fetch_details, res['href']): res
                for res in search_results
            }
            for future in concurrent.futures.as_completed(future_to_result):
                original_res = future_to_result[future]
                try:
                    details = future.result()
                    detailed_results.append({
                        "title": original_res['title'],
                        "link": original_res['href'],
                        "source": "Web Search",
                        "summary": original_res['body'], # Initial snippet
                        "full_content": details.get("text", "")[:5000], # Limit text length
```

```
                    "image": details.get("image")
                })
            except Exception:
                continue
    return detailed_results
--- File: src/full_content_fetcher.py ---
from newspaper import Article
from bs4 import BeautifulSoup
import requests
import concurrent.futures
class FullContentFetcher:
    def __init__(self):
        self.headers = {
            'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/121.0.0.0 Safa
        }
    def fetch_details(self, url: str) -&gt; dict:
        """
        Fetch full content and top image using newspaper3k/bs4
        """
        result = {
            "text": "",
            "image": None
        }
        try:
            article = Article(url)
            article.download()
            article.parse()
            if article.text and len(article.text) &gt; 200:
                result["text"] = article.text
            if article.top_image:
                result["image"] = article.top_image
            if not result["text"] or not result["image"]:
                response = requests.get(url, headers=self.headers, timeout=10)
                soup = BeautifulSoup(response.content, 'html.parser')
                if not result["text"]:
                    paragraphs = soup.find_all('p')
                    text = ' '.join([p.get_text() for p in paragraphs])
                    if len(text) &gt; 100:
                        result["text"] = text
                if not result["image"]:
                    og_image = soup.find("meta", property="og:image")
                    if og_image and og_image.get("content"):
                        result["image"] = og_image["content"]
            return result
        except Exception as e:
            return result
    def enrich_news_items(self, news_items: list) -&gt; list:
        """
        Parallel fetch full content and images for all news items
        """
        print(f" Fetching full content &amp;amp; images for {len(news_items)} articles...")
        with concurrent.futures.ThreadPoolExecutor(max_workers=10) as executor:
            future_to_item = {
                executor.submit(self.fetch_details, item['link']): item
                for item in news_items
            }
            for future in concurrent.futures.as_completed(future_to_item):
                item = future_to_item[future]
                try:
                    details = future.result()
                    if details["text"] and len(details["text"]) &gt; 200:
                        item['full_content'] = details["text"]
                    else:
                        item['full_content'] = item['summary'] # Fallback
                    item['image'] = details["image"]
                except Exception:
                    item['full_content'] = item['summary']
                    item['image'] = None
        return news_items
```