SOMMARIO

0. What this app does (in one sentence) *3*

1. Prerequisites (One-Time Setup) *3*

2. Project files (Description) *3*

3. Start the app *4*

4. How it Works – The Flow *4*

5. The tools (plain english) *4*

6. Key UI options (what they do) *5*

7. History (how to use it) *5*

8. Common errors & fixes *5*

9. tools.py — Annotated Walkthrough *5*

9. 1 Imports & setup **5**

9. 2 Lightweight LLM (for summaries/translations) **6**

9. 3 Save results to TXT **6**

9. 4 Web search (DuckDuckGo) **6**

9. 5 Wikipedia **7**

9. 6 Translate **7**

9. 7 Summarize **7**

9. 8 Export to PDF (StructuredTool) **8**

9. 9 Scrape **8**

10. main.py — Annotated Walkthrough *9*

10. 1 Imports & Tool Registration **9**

10. 2 Load environment variables **10**

10. 3 Define the structured output schema **10**

10. 4 LLM & Output Parser **10**

10. 5 System Prompt **10**

10. 6 Prompt Template **11**

10. 7 Tools & Agent **11**

10. 8 Memory & Executor **12**

10. 9 Command-Line Interface (CLI) **12**

10. 10 How to run (examples) **13**

11. app.py — Annotated Walkthrough ***13***

11. 1 Imports & Environment **13**

11. 2 Session State (History) **14**

11. 3 UI Helpers **14**

11. 4 Sidebar (Theme, Options, History) **14**

11. 5 Model, Prompt & Parser **15**

11. 6 Form and Agent Builder **16**

11. 7 Execute (main flow) **16**

11. 8 Rendering & Download **16**

11. 9 History & Debug **17**

AI RESEARCH ASSISTANT — BEGINNER'S GUIDE

0. WHAT THIS APP DOES (IN ONE SENTENCE)

You type a topic → the app researches it (Wikipedia + optionally web), writes a clean paragraph, (optionally) translates it, shows sources, and lets you download TXT/PDF. It also keeps a local history.

1. PREREQUISITES (ONE-TIME SETUP)

1. **Python 3.10+** installed
2. **Create a virtual env** (recommended)

```
python -m venv .venv

source .venv/bin/activate      # mac/linux

# .venv\Scripts\activate       # windows powershell
```

3. **Add your OpenAI key**

   Create a file named .env in the project root:

```
OPENAI_API_KEY=sk-...your_key_here...
```

4. **Install dependencies**

```
pip install -r requirements.txt
```

Tip: If DuckDuckGo shows a warning, run `pip install ddgs`.

2. PROJECT FILES (DESCRIPTION)

File / Folder	Purpose
app.py	Streamlit app (UI, history, translations, downloads)
tools.py	All tools the agent can call (search, Wikipedia, translate, summarize, PDF export, scrape)
main.py	CLI version (run in terminal without UI)
style.css	Dark theme & UI styling
requirements.txt	List of dependencies
outputs/	Saved TXT and PDF files
.env	Your OpenAI API key *(not committed)*

3. START THE APP

```
streamlit run app.py
```

Open the link Streamlit prints (usually http://localhost:8501), type a topic, choose options (translate, PDF, etc.), click **Run Research**.

Want CLI instead?

```
python main.py "Tell me about the Renaissance" --translate Italian
```

4. HOW IT WORKS – THE FLOW

1. **User Input** → You type a query in the UI or CLI.
2. **Agent Creation** → The app builds a prompt and creates a LangChain Agent with **only the tools you enabled**.
3. **Information Gathering** →
    - Uses **Wikipedia** (and optionally **DuckDuckGo**)
    - Optionally summarizes with the **summarize tool**
4. **JSON Output** → Agent returns a strict JSON with:
    - topic
    - summary
    - sources
    - tools_used
5. **Post-Processing** → If translation is requested, the app translates the *final summary* (saves tokens & keeps consistency).
6. **Results Display** → Shows result in the app, saves TXT, and optionally exports a PDF.
7. **Session History** → Stores each run so you can reload it later from the sidebar.

5. THE TOOLS (PLAIN ENGLISH)

	What it Does	When to Use	Notes
search_tool	Searches the web using DuckDuckGo.	For latest news, trends, and recent events.	Slightly higher cost due to API calls.
wiki_tool	Gets a short, focused summary from Wikipedia.	For well-established topics.	Very fast & reliable for basics.
summarize_tool	Compresses long text into one clean paragraph.	When sources are lengthy or verbose.	Keeps results concise and readable.
translate_tool	Translates the final summary.	When you need output in another language.	Translation is done **after** research to save cost.
export_to_pdf_tool	Generates a professional PDF report.	For sharing or archiving research.	Saves in outputs/ folder.
scrape_tool	Extracts clean text from a given URL.	For online articles or documents.	Best for simple, non-JavaScript-heavy pages.
save_tool	Appends the research JSON to a TXT file.	If you want a cumulative log.	Writes to research_output.txt.

Tip: In the UI, you can enable/disable tools to control **both behavior and API costs**.

6. KEY UI OPTIONS (WHAT THEY DO)

- **Output language**: the app translates the final paragraph **after** the agent runs (cheap & reliable).
- **Use Wikipedia**: include Wikipedia summaries in research.
- **Use Web Search**: include DuckDuckGo results (newsier, costs a bit more).
- **Summarize**: encourages a tighter paragraph when the agent gathers a lot.
- **Translate**: do the final translation into the selected language.
- **Export PDF**: save a PDF report in outputs/.
- **Show original text**: when translating, show English + translated side by side.
- **Cost saver**: switches to gpt-4o-mini and reduces tool usage to cut costs.

---

7. HISTORY (HOW TO USE IT)

- Every successful run is saved in memory (within the session), and files are saved in the outputs/directory.
- In the **sidebar**, open **Search history**, select an entry, click **Load** → the app shows that result again (no API cost).
- **Clear** resets the in-memory history list (your TXT/PDF files remain on disk).

8. COMMON ERRORS & FIXES

Error Message	Likely Cause	How to Fix
"Missing OPENAI_API_KEY"	.env file missing or wrong key.	Create/update .env with OPENAI_API_KEY=sk-... and restart terminal (or re-activate venv).
"duckduckgo_search renamed" warning	Package rename in recent update.	Run pip install ddgs. The app works even with the warning.
PDF export fails	Missing dependency or no write permissions.	Ensure fpdf is installed (already in requirements.txt) and outputs/ is writable.
Weird JSON parse errors	LLM returned unexpected format.	Click **Run Research** again. If it persists, disable unnecessary tools.

9. TOOLS.PY — ANNOTATED WALKTHROUGH

9. 1 IMPORTS & SETUP

```
from datetime import datetime
from typing import Optional
import json
import requests
from bs4 import BeautifulSoup
from fpdf import FPDF

from dotenv import load_dotenv
from pydantic import BaseModel
```

```
from langchain_community.tools import WikipediaQueryRun, DuckDuckGoSearchRun
from langchain_community.utilities import WikipediaAPIWrapper
from langchain.tools import Tool, StructuredTool
from langchain_openai import ChatOpenAI
```

- datetime, json: used to timestamp and serialize data for saving.
- requests: performs HTTP requests (used by the scraper).
- BeautifulSoup: parses HTML to extract visible text.
- FPDF: generates PDFs from plain text.
- load_dotenv: loads environment variables (e.g., your OPENAI_API_KEY).
- BaseModel (Pydantic): defines structured schemas (used for the PDF tool).
- WikipediaQueryRun, DuckDuckGoSearchRun, WikipediaAPIWrapper: prebuilt helpers.
- Tool, StructuredTool: wrap Python functions so the Agent can call them safely.
- ChatOpenAI: OpenAI chat model client (language model).

```
load_dotenv()
```

- Loads environment variables once so ChatOpenAI can read OPENAI_API_KEY without hardcoding it.

9. 2 LIGHTWEIGHT LLM (FOR SUMMARIES/TRANSLATIONS)

```
llm_tools = ChatOpenAI(model="gpt-4o-mini", temperature=0)
```

- Creates a small/cheap/fast LLM instance dedicated to utility tasks (translate/summarize).
- temperature=0 makes outputs deterministic and stable.

9. 3 SAVE RESULTS TO TXT

```
def save_to_txt(data: dict, filename: str = "research_output.txt") -> str:
    """Append a structured research result as JSON to a text file."""
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    json_text = json.dumps(data, indent=2, ensure_ascii=False)
    formatted_text = f"--- Research Output ---\nTimestamp:
{timestamp}\n\n{json_text}\n\n"
    with open(filename, "a", encoding="utf-8") as f:
        f.write(formatted_text)
    return f"Data successfully saved to {filename}"

save_tool = Tool(
    name="save_text_to_file",
    func=save_to_txt,
    description="Append the complete structured research result (topic, summary,
sources, tools_used) to a local text file."
)
```

- save_to_txt: turns the result dict into pretty JSON and appends it to a .txt file.
- save_tool: exposes the function to the Agent as a callable tool.

9. 4 WEB SEARCH (DUCKDUCKGO)

```
search_tool = Tool(
    name="search",
    func=DuckDuckGoSearchRun().run,
    description="Search the web for recent or general information. Input is a
search query."
)
```

- Wraps DuckDuckGo search as a tool; input is a plain query string.

9. 5 WIKIPEDIA

```
wiki_tool = WikipediaQueryRun(
    api_wrapper=WikipediaAPIWrapper(top_k_results=1, doc_content_chars_max=1500)
)
```

- Fetches a concise Wikipedia page summary (limited length to keep costs low and responses crisp).

9. 6 TRANSLATE

```
def translate_text(text: str, target_language: Optional[str] = "Italian") -> str:
    """Translate text to the specified target language."""
    prompt = (
        f"Translate the following text into {target_language}. "
        "Return ONLY the translation, with no preface or quotes.\n\n"
        f"Text:\n{text}"
    )
    try:
        resp = llm_tools.invoke(prompt)
        return resp.content.strip()
    except Exception as e:
        return f"(Translation failed: {e}) {text}"

translate_tool = Tool(
    name="translate_tool",
    func=translate_text,
    description="Translate a given text into another language. Input: text and
optional target_language."
)
```

- Asks the small model to translate and return only the translation (no extra fluff).
- On error, it returns the original text with a small error note so the Agent can continue.

9. 7 SUMMARIZE

```
def summarize_text(text: str) -> str:
    """Summarize text into one concise paragraph."""
    prompt = (
        "Summarize the following text into a single concise, fluent paragraph. "
        "Avoid bullet points and keep essential facts.\n\n"
        f"Text:\n{text}"
    )
    try:
        resp = llm_tools.invoke(prompt)
```

```
        return resp.content.strip()
    except Exception as e:
        return f"(Summary failed: {e}) {text[:300]}..."

summarize_tool = Tool(
    name="summarize_tool",
    func=summarize_text,
    description="Generate a short, fluent paragraph summary of a longer text."
)
```

- Prompts the lightweight LLM for a single-paragraph, no-bullets summary.
- On failure, it returns a truncated snippet so your app still has content.

9. 8 EXPORT TO PDF (STRUCTUREDTOOL)

```
def export_to_pdf(text: str, filename: str = "output.pdf") -> str:
    """Generate a PDF file from provided text."""
    pdf = FPDF()
    pdf.add_page()
    pdf.set_font("Arial", size=12)
    pdf.multi_cell(0, 8, text)
    pdf.output(filename)
    return f"PDF saved as {filename}"

class PdfArgs(BaseModel):
    text: str
    filename: str = "output.pdf"

export_to_pdf_tool = StructuredTool.from_function(
    func=export_to_pdf,
    name="export_to_pdf_tool",
    description="Generate a PDF from provided text and save it to the given
filename.",
    args_schema=PdfArgs,
)
```

- export_to_pdf: builds a simple, readable PDF via FPDF.
- StructuredTool: defines a strict schema (PdfArgs) so the Agent passes both 'text' and 'filename' correctly.

9. 9 SCRAPE

```
def scrape_page(url: str) -> str:
    """Scrape visible text from a public webpage."""
    try:
        headers = {"User-Agent": "Mozilla/5.0 (compatible; ResearchAgent/1.0)"}
        resp = requests.get(url, headers=headers, timeout=15)
        resp.raise_for_status()
        soup = BeautifulSoup(resp.text, "html.parser")
        for tag in soup(["script", "style", "noscript"]):
            tag.decompose()
        text = " ".join(soup.get_text(separator=" ").split())
        return text[:2000]
```

```
    except Exception as e:
        return f"(Scraping failed: {str(e)})"

scrape_tool = Tool(
    name="scrape_tool",
    func=scrape_page,
    description="Scrape visible text content from a public URL (must start with
https://)."
)
```

- Adds a friendly User-Agent and a timeout to be robust.
- Removes script/style/noscript content; normalizes whitespace.
- Returns only the first ~2000 chars to keep output manageable.

10. MAIN.PY — ANNOTATED WALKTHROUGH

This document explains the CLI entrypoint that wires your LangChain agent together. **It's written for beginners: each section shows a code block followed by plain-English notes.**

10. 1 IMPORTS & TOOL REGISTRATION

```
 mport os
import sys
import argparse
from typing import List

from dotenv import load_dotenv
from pydantic import BaseModel, ValidationError

from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import PydanticOutputParser
from langchain.agents import create_tool_calling_agent, AgentExecutor
from langchain.memory import ConversationBufferMemory

from tools import (
    search_tool,
    wiki_tool,
    save_tool,
    translate_tool,
    summarize_tool,
    export_to_pdf_tool,
    scrape_tool,
)
```

- dotenv: loads secrets from a .env file (OPENAI_API_KEY).
- pydantic: defines/validates the JSON shape the model must return.
- langchain_openai / core / agents / memory: core LangChain pieces to build a tool-calling agent.
- tools: your custom tool objects (web search, wikipedia, translate, etc.).

10. 2 LOAD ENVIRONMENT VARIABLES

```
load_dotenv()

OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
if not OPENAI_API_KEY:
    print("❌ Missing OPENAI_API_KEY in environment. Create a .env with
OPENAI_API_KEY=...")
    sys.exit(1)
```

- load_dotenv(): pulls variables from .env into process env.
- Hard-fails early if the key is missing to avoid confusing errors later.

10. 3 DEFINE THE STRUCTURED OUTPUT SCHEMA

```
class ResearchResponse(BaseModel):
    topic: str
    summary: str
    sources: List[str]
    tools_used: List[str]
```

- This is the single source of truth for what the agent must output.
- Pydantic will validate/parse the model's raw text into this object (or raise).

10. 4 LLM & OUTPUT PARSER

```
llm = ChatOpenAI(model="gpt-4o", temperature=0)
parser = PydanticOutputParser(pydantic_object=ResearchResponse)
format_instructions = parser.get_format_instructions()
```

- temperature=0 → deterministic, concise answers.
- PydanticOutputParser generates the JSON schema string the prompt will include.

10. 5 SYSTEM PROMPT

```
SYSTEM_TEXT = f"""
# ROLE:
You are a highly capable AI Research Assistant.
You specialize in generating structured research outputs and can use external tools
when necessary.

# TASK:
- Help the user research a given topic.
- Gather, synthesize, and present the information in a single, fluent paragraph (no
bullet points).
- Ensure clarity, coherence, and conciseness.

# OUTPUT:
Return ONLY a JSON object matching this schema:

{format_instructions}
```

```
# TOOLS AVAILABLE:
- search (DuckDuckGo), wikipedia, summarize, translate, scrape, export to PDF, save to
file.

# RULES:
- Use tools only when necessary.
- Do not fabricate sources; list only sources you actually used.
- If translation is requested, perform it as the final step.

# EXAMPLES:
User: "Research climate change and translate it in Italian"
Assistant: Research → Summarize → Translate → Return JSON.

User: "Tell me about blockchain and summarize it"
Assistant: Research → Summarize → Return JSON.
"""
```

- Clear role, tasks, and hard rules to keep outputs strictly JSON.
- The {format_instructions} placeholder injects the exact JSON schema.

10. 6 PROMPT TEMPLATE

```
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", SYSTEM_TEXT),
        ("placeholder", "{chat_history}"),
        ("human", "{query}"),
        ("placeholder", "{agent_scratchpad}"),
    ]
)
```

- {chat_history}: memory context; {agent_scratchpad}: agent's internal tool-call traces.
- Human slot is just {query}; the CLI will pass your question there.

10. 7 TOOLS & AGENT

```
tools = [
    search_tool,
    wiki_tool,
    save_tool,
    translate_tool,
    summarize_tool,
    export_to_pdf_tool,
    scrape_tool,
]

agent = create_tool_calling_agent(llm=llm, prompt=prompt, tools=tools)
```

- Order doesn't matter; names/descriptions inside each tool guide the LLM.
- create_tool_calling_agent wires the LLM so it can plan and call tools.

10. 8 MEMORY & EXECUTOR

```
memory = ConversationBufferMemory(
    return_messages=True,
    memory_key="chat_history",
)

executor = AgentExecutor(
    agent=agent,
    tools=tools,
    memory=memory,
    verbose=True,
    return_intermediate_steps=False,
    return_only_outputs=True,
)
```

- ConversationBufferMemory stores previous user/AI turns to keep context.
- AgentExecutor runs the loop (think → call tool → observe → respond).
- return_only_outputs=True means executor.invoke(...) returns the final text only.

10. 9 COMMAND-LINE INTERFACE (CLI)

```
def run_cli():
    ap = argparse.ArgumentParser(description="Research Agent (CLI)")
    ap.add_argument("query", nargs="*", help="Your research query")
    ap.add_argument("--translate", "-t", default="", help="Translate the final summary
to this language (e.g., Italian)")
    args = ap.parse_args()

    user_query = " ".join(args.query).strip()
    if not user_query:
        user_query = input("What can I help you research? ").strip()

    if args.translate:
        user_query += f"\n\nPlease translate the final summary into {args.translate}."

    try:
        raw = executor.invoke({"query": user_query})
        output_text = raw.get("output", raw)
        data = parser.parse(output_text)

        print("\n✅ Parsed JSON Output:")
        print(data.model_dump_json(indent=2, ensure_ascii=False))

    except ValidationError as ve:
        print("❌ Pydantic validation error:")
        print(ve); sys.exit(2)

    except Exception as e:
        print("❌ Agent execution error:")
        print(e)
        if isinstance(raw, dict):
```

```
            print("\nRaw output:\n", raw.get("output", raw))
        sys.exit(3)

if __name__ == "__main__":
    run_cli()
```

- You can pass the query directly (e.g., python main.py "Herman Hesse"),
- or run without args and type when prompted.
- --translate/-t appends a natural-language instruction so the agent translates at the end.
- parser.parse(...) guarantees JSON validity against the ResearchResponse schema.

10. 10 HOW TO RUN (EXAMPLES)

- `python main.py "Renaissance art"`
- `python main.py --translate Italian "Quantum computing basics"`
- `python main.py` (then type your query interactively)

11. APP.PY — ANNOTATED WALKTHROUGH

A line-by-line, beginner-friendly guide to the Streamlit research agent. This version uses dark code blocks, clear callouts, and practical tips.

11. 1 IMPORTS & ENVIRONMENT

```
import os
from datetime import datetime
from typing import List

import streamlit as st
from dotenv import load_dotenv
from pydantic import BaseModel

from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import PydanticOutputParser
from langchain_openai import ChatOpenAI
from langchain.agents import create_tool_calling_agent, AgentExecutor
from langchain.memory import ConversationBufferMemory

from tools import (
    search_tool, wiki_tool, save_tool, translate_tool,
    summarize_tool, export_to_pdf_tool, scrape_tool,
)
```

• Keep standard libs on top, third-party libs next, and local imports last.
• We import Streamlit for the UI, LangChain for agent + tools, and our custom tools.

```
load_dotenv()
st.set_page_config(page_title='AI Research Assistant', layout='wide')
```

• load_dotenv(): loads OPENAI_API_KEY and any other secrets from .env.
• set_page_config(): sets the browser tab title and a wide layout.

11. 2 SESSION STATE (HISTORY)

```
if 'history' not in st.session_state:
    st.session_state.history = []  # list of dicts
if 'selected_history' not in st.session_state:
    st.session_state.selected_history = None
```

- Session state survives reruns in Streamlit.
- We store past runs and which run (if any) the user wants to reload.

11. 3 UI HELPERS

```
def tool_chip(name: str) -> str:
    n = name.lower()
    icon = ('🔎' if 'search' in n else '📖' if 'wiki' in n else '📝' if 'summarize'
in n
            else '🌐' if 'translate' in n else '📄' if 'pdf' in n else '💾' if
'save' in n else '🧩')
    return f"""<span class='tool-badge'>{icon} {name}</span>"""
```

- Returns small HTML badges (styled in style.css).

```
def linkify_sources(sources):
    out = []
    for s in sources:
        s = s.strip()
        url = s if s.startswith('http') else None
        label = s if not url else (s.split('//', 1)[-1][:60] + ('…' if len(s) > 60
else ''))
        out.append(f"<li><a href='{url or '#'}' target='_blank' rel='noopener
noreferrer'>{label}</a></li>"
                   if url else f"<li>{label}</li>")
    return '<ul>' + ''.join(out) + '</ul>' if out else '—'
```

- Converts a list of raw strings/URLs to a clean clickable list.

```
def show_loader():
    placeholder = st.empty()
    html = """<div style='display:flex;align-items:center;gap:14px;margin:20px 0'>
      <div class='loader-dots'></div>
      <div style='font-weight:600;font-size:16px;'>The AI is gathering knowledge for
you...</div>
    </div>"""
    placeholder.markdown(html, unsafe_allow_html=True)
    return placeholder
```

- Creates a temporary area for the animated loader. Call .empty() to hide it.

11. 4 SIDEBAR (THEME, OPTIONS, HISTORY)

```
theme_choice = st.sidebar.radio('Select Theme:', ['Light', 'Dark'], index=1)
output_language = st.sidebar.selectbox('Output language:',
```

```
['Italian','English','French','Spanish'], index=0)
use_wikipedia = st.sidebar.checkbox('Use Wikipedia', value=True)
use_search = st.sidebar.checkbox('Use Web Search (DuckDuckGo)', value=True)
do_summarize = st.sidebar.checkbox('Summarize', value=True)
do_translate = st.sidebar.checkbox('Translate', value=True)
do_pdf = st.sidebar.checkbox('Export PDF', value=True)
show_original = st.sidebar.checkbox('Show original text', value=True)
```

- These toggles drive which tools get loaded and how the post-processing behaves.

```
# History picker
st.sidebar.markdown('---')
hist_items = list(reversed(st.session_state.history))[:10]
choices = [f"{it.get('topic','—')} · {it.get('timestamp','')} ·
{it.get('language','')}" for it in hist_items]
selected = st.sidebar.selectbox('Previous runs:', ['—'] + choices, index=0)
c1, c2 = st.sidebar.columns(2)
if c1.button('Load') and selected != '—':
    st.session_state.selected_history = hist_items[choices.index(selected)]
if c2.button('Clear'):
    st.session_state.history.clear(); st.session_state.selected_history = None
```

11. 5 MODEL, PROMPT & PARSER

```
model_name = 'gpt-4o-mini'  # or 'gpt-4o' for higher quality
llm = ChatOpenAI(model=model_name, temperature=0)

class ResearchOutput(BaseModel):
    topic: str
    summary: str
    sources: List[str]
    tools_used: List[str]

parser = PydanticOutputParser(pydantic_object=ResearchOutput)
format_instructions = parser.get_format_instructions()

system_message = '''
# ROLE:
You are a highly capable AI Research Assistant...
# OUTPUT FORMAT:
{format_instructions}
# CONSTRAINTS:
- Output ONLY valid JSON...
'''.strip()

prompt = ChatPromptTemplate.from_messages([
    ('system', system_message),
    ('placeholder', '{chat_history}'),
    ('human', '{input}'),
    ('placeholder', '{agent_scratchpad}'),
]).partial(format_instructions=format_instructions)
```

- Pydantic enforces a strict JSON schema for the agent's output.
- ChatPromptTemplate wires system + user + tool-scratchpad messages.

11. 6 FORM AND AGENT BUILDER

```
st.title('AI Research Assistant')
with st.form('research_form'):
    query = st.text_input('What do you want to research?')
    run = st.form_submit_button('Run Research')

def build_agent(selected_tools):
    agent = create_tool_calling_agent(llm=llm, prompt=prompt, tools=selected_tools)
    memory = ConversationBufferMemory(memory_key='chat_history',
return_messages=True)
    return AgentExecutor(agent=agent, tools=selected_tools, memory=memory,
                         verbose=True, return_intermediate_steps=True,
return_only_outputs=True)
```

11. 7 EXECUTE (MAIN FLOW)

```
if run and query:
    user_intent = query.strip()
    if do_summarize:  # optional extra instruction
        user_intent += '\n\nPlease summarize the findings in one fluent paragraph.'

    selected_tools = []
    if use_search: selected_tools.append(search_tool)
    if use_wikipedia: selected_tools.append(wiki_tool)
    if do_summarize: selected_tools.append(summarize_tool)
    selected_tools.append(save_tool)
    if do_pdf: selected_tools.append(export_to_pdf_tool)

    executor = build_agent(selected_tools)
    loader = show_loader()
    raw = executor.invoke({'input': user_intent})
    loader.empty()

    data = parser.parse(raw['output'])
    original_summary = data.summary

    if do_translate and output_language.lower() != 'english':
        data.summary = translate_tool.func(data.summary, output_language).strip()
        data.tools_used.append(f'post_translate({output_language})')
```

- We pass a single input string to the agent; the model decides tool calling.
- Translation is a post-processing step to save money and keep behavior predictable.

11. 8 RENDERING & DOWNLOAD

```
# Title + badge
badge = f"<span class='lang-badge'>Translated to {output_language}</span>" \
    if (do_translate and output_language.lower() != 'english') else ''
st.markdown(f"<h2>📌 {data.topic} {badge}</h2>", unsafe_allow_html=True)

# Summary (optionally side-by-side original/translated)
# ...
```

```
# Sources & tools
st.markdown(' '.join(tool_chip(t) for t in data.tools_used), unsafe_allow_html=True)
st.markdown(linkify_sources(data.sources), unsafe_allow_html=True)

# Save to files
os.makedirs('outputs', exist_ok=True)
timestamp = datetime.now().strftime('%Y-%m-%d_%H-%M-%S')
safe_topic = data.topic.replace(' ', '_')
txt_path = f'outputs/research_{safe_topic}_{timestamp}.txt'
with open(txt_path, 'w', encoding='utf-8') as f: ...
st.download_button('⬇ Download TXT', data=open(txt_path,'rb'),
file_name=os.path.basename(txt_path))

if do_pdf:
    pdf_path = f'outputs/report_{safe_topic}_{timestamp}.pdf'
    export_to_pdf_tool.run({'text': data.summary, 'filename': pdf_path})
    st.download_button('⬇ Download PDF', data=open(pdf_path,'rb'),
file_name=os.path.basename(pdf_path))
```

11. 9 HISTORY & DEBUG

```
# Save run for quick reloads
st.session_state.history.append({
    'topic': data.topic,
    'summary': data.summary,
    'original_summary': original_summary,
    'language': output_language if do_translate else 'English',
    'sources': data.sources,
    'tools_used': data.tools_used,
    'timestamp': datetime.now().strftime('%Y-%m-%d %H:%M'),
    'txt_path': txt_path,
    'pdf_path': pdf_path if do_pdf else None,
})

with st.expander('🔍 Debug / Technical details'):
    steps = st.session_state.get('last_steps', [])
    if steps:
        for i, (action, observation) in enumerate(steps, start=1):
            st.markdown(f"**Step {i} – Tool:** `{getattr(action, 'tool',
'unknown')}`")
            st.code(str(getattr(action, 'tool_input', ''))[:1000])
            st.write(observation if isinstance(observation, str) else
str(observation))
```