Here’s a **final large project** that ties together everything you’ve learned in LangChain — and it’s **portfolio-worthy**:

**📦 AI-Powered Multi-Source Research Assistant**

**Description:**  
A web-based AI tool that takes a user’s query and performs **real-time multi-source research** using web search, PDFs, YouTube videos, and structured APIs, then **summarizes, compares, and cites** the results in a well-structured report.

**Core Features**

1. **Multi-Modal Data Retrieval**
   * **Web Search:** Use LangChain’s TavilySearchAPIWrapper or SerpAPI to pull relevant articles.
   * **PDF Parsing:** Allow users to upload PDFs (papers, reports) and index them for semantic search.
   * **YouTube Transcript Extraction:** Auto-fetch transcripts and process them for relevant information.
   * **API Integration:** For structured data (e.g., weather, finance, scientific datasets).
2. **RAG Pipeline**
   * Store all data in a **vector database** (FAISS / Chroma / Pinecone).
   * Use a **Retrieval-Augmented Generation** chain to answer questions with context.
   * Ensure **citations** with chunk sources.
3. **Query Understanding**
   * Classify user queries into **categories** (news, academic research, tutorial, fact-checking) using LangChain’s LLMClassifier.
   * Dynamically choose **which tools to call** based on category.
4. **Smart Summarization**
   * Summarize each source **individually** (source-level summary).
   * Produce a **final synthesized answer** with pros, cons, and references.
5. **Memory**
   * Maintain a **session memory** so the user can refine queries iteratively.
6. **UI**
   * Frontend in **Next.js** with:
     + Query input box
     + File uploader (PDFs)
     + YouTube link field
     + Real-time research progress indicator
   * Backend in **FastAPI** (integrates with LangChain pipeline).

**Tech Stack**

* **Backend:** Python, FastAPI, LangChain, FAISS/Pinecone
* **Frontend:** Next.js, TailwindCSS
* **Integrations:** Tavily Search API, YouTube Transcript API, PyPDF2
* **Model:** OpenAI GPT-4 / Llama 3 / Mistral depending on budget

**Advanced Add-ons**

* **Agent Mode:** Let the AI autonomously search, read, and analyze until a confidence threshold is reached.
* **Report Export:** Export results as PDF/Markdown.
* **Live Updates:** Use WebSockets to stream AI research progress.

**Why It’s Great for a Portfolio**

* Shows **mastery of LangChain tools & agents**.
* Demonstrates **multi-modal data handling**.
* Proves you can integrate **frontend + backend + AI pipeline**.
* Highly **practical** — businesses, students, and researchers would actually use it.

If you want, I can **break this down into an exact step-by-step build plan** so you can finish it in 3–4 weeks while keeping it impressive.

Nice — here’s a compact, **exact step-by-step 4-week build plan** you can follow to deliver the **AI-Powered Multi-Source Research Assistant**. It’s written so you can work day-by-day and end up with a deployable, portfolio-ready product. I’ll keep it focused and practical.

**Overview (what you’ll ship)**

MVP: a FastAPI backend + Next.js frontend where a user submits a query (or uploads PDF / YouTube link), the system:

1. ingests sources (web, PDFs, YouTube transcripts, APIs),
2. embeds and stores chunks in a vector DB,
3. runs a RAG pipeline to produce a cited, structured answer,
4. streams progress to the frontend and allows exporting results.

Estimated total time: **~120–160 hours** (4 weeks, ~30–40 hrs/week).  
Stack: Python, LangChain, FAISS/Chroma/Pinecone, FastAPI, Next.js, embeddings (OpenAI / local), LLM (OpenAI GPT-4 or Llama), YouTube Transcript API, PyPDF2 / pdfplumber.

**Week 1 — Foundations & Ingestion (MVP backbone)**

Goal: repo + ingestion + embeddings + basic retrieval + simple RAG QA.

Day 1 — Repo & env

* Create repo, branch structure, README skeleton.
* Create virtualenv / conda environment; add requirements (langchain, faiss-cpu/chroma, openai, fastapi, uvicorn, PyPDF2, youtube-transcript-api).
* Setup GitHub Actions basic CI (lint, tests).

Day 2 — Pick infra & config

* Choose vector DB (FAISS locally for dev; Pinecone/Chroma for prod).
* Create config file for API keys (env vars): OPENAI\_API\_KEY, PINECONE\_API\_KEY, SERPAPI\_KEY (or preferred search).
* Minimal logging via structlog or python logging.

Day 3 — Document Loaders

* Implement loaders:
  + PDFs (pdfplumber / PyPDF2) → plain text + metadata (filename, page, URL).
  + YouTube transcript loader using youtube-transcript-api.
  + Web search wrapper (SerpAPI or any search API) to fetch top N URLs and their text via simple HTML loader.
* Save raw documents to data/raw/ with metadata JSON.

Day 4 — Text Splitter & Preprocessing

* Implement RecursiveCharacterTextSplitter with tuned chunk\_size (e.g., 800 tokens) and overlap (100–200).
* Add language normalization, remove boilerplate, keep source references in metadata.
* Run ingestion pipeline on a small sample set (10 docs/2 videos).

Day 5 — Embeddings & Vector Store

* Pick embeddings (OpenAI / local). Implement Embeddings wrapper.
* Build ingestion script: loader → splitter → embedding → upsert to vector store with metadata (source, url, page, timestamp).
* Validate: search a sample query and inspect retrieved chunks.

Day 6 — Simple RAG Chain (MVP QA)

* Implement a basic RetrievalQA chain:
  + Retriever (k=4) → LLM (system prompt for citation format) → generate answer + source citations.
* Add a small OutputParser that returns {answer, citations:[{source,score}], raw\_sources}.
* Test with 10 sample queries. Ensure citations reference chunk metadata.

Success criteria (end of week): repo + env + ingestion pipeline + vector store + basic RAG QA working end-to-end on local machine.

**Week 2 — Tools, Agents, & smarter retrieval**

Goal: build tool wrappers, agent orchestration, and improve RAG (citations, query routing).

Day 1 — Tools: wrappers

* Implement tool wrappers as LangChain @tool or small functions:
  + web\_search\_tool(query): calls SerpAPI, returns top URLs & snippets.
  + youtube\_transcript\_tool(url): returns transcript text.
  + pdf\_loader\_tool(file): accepts upload and returns text chunks + metadata.
* Add rate-limit and error handling inside tools.

Day 2 — Agent that orchestrates tools

* Build an Agent (or custom controller) that:
  + Classifies query type (research / quick-fact / tutorial) using a small LLM classifier.
  + Based on classification, chooses tools to call (e.g., heavy web search + YouTube for research).
* Implement a Plan -> Execute -> Synthesize flow where Agent decides which tools to call and how many docs to fetch.

Day 3 — Advanced Retriever logic

* Implement hybrid retrieval: combine dense retriever (vector DB) + keyword search score (BM25 or search API) and merge results by weighted score.
* Implement dynamic k selection: longer/complex queries → larger k.

Day 4 — Improved RAG chain & structured outputs

* Replace MVP chain with a multi-step chain:
  1. Retrieve top chunks.
  2. Summarize each chunk (source-level summary).
  3. Synthesize summaries into final answer with structured fields: {answer, pros, cons, citations, confidence\_score}.
* Use StructuredOutputParser or Pydantic model for structured JSON.

Day 5 — Hallucination mitigation & citation formatting

* Add a verification step: for any factual claim with a number or proper noun, include the chunk that supports it; if unsupported, mark unsupported and lower confidence.
* Adjust prompt templates to force "Answer only using provided sources. If unsupported, say 'Insufficient evidence'."

Day 6 — Tests & demo

* Integration test: user query → agent calls tools → vector store retrieval → structured answer.
* Create Postman / curl sample endpoints for the backend.

Success criteria (end of week): Agent + tools orchestrated, hybrid retrieval implemented, structured RAG responses with source-level summaries.

**Week 3 — Frontend, streaming progress & session memory**

Goal: frontend UI, streaming replies, session memory, user uploads.

Day 1 — Frontend skeleton (Next.js)

* Create Next.js app with Tailwind.
* Pages: Home (query input), Upload (PDF/YouTube), Results (streaming answer), History (session).
* Implement simple form to send queries to FastAPI.

Day 2 — Backend endpoints & streaming

* Add FastAPI endpoints:
  + POST /query starts job, returns job\_id.
  + GET /stream/{job\_id} streams progress (server-sent events or WebSocket).
  + POST /upload for PDFs / YouTube URLs.
* Implement SSE or WebSocket server to stream logs (retrieval started, tool called, summarizing, final answer chunks).

Day 3 — Session memory

* Implement per-session memory store (Redis or simple in-memory for dev) that stores recent queries, selected sources, and vector search context.
* Allow follow-up questions that reuse previous context.

Day 4 — File upload & ingestion UI

* Add file uploader: on upload, backend ingests file, chunks and embeds. Show progress bar.
* Show list of ingested sources and let user pin/unpin sources to prioritize.

Day 5 — UI polishing & export

* Show source-level cards in results with clickable citation to view the original snippet.
* Implement export to Markdown / PDF (server-side render using WeasyPrint or wkhtmltopdf).

Day 6 — User testing & feedback loop

* Run quick usability tests with friends. Collect edge-case feedback.
* Fix obvious UX bugs (long-loading indicator, error handling).

Success criteria (end of week): usable frontend with streaming responses, uploads, memory and export working.

**Week 4 — Productionize, tests, monitoring & advanced features**

Goal: finalize features, deploy, create demo, tests, and documentation.

Day 1 — Authentication & security

* Add simple auth (JWT + NextAuth for frontend).
* Secure file upload (scan file size/type), sanitize inputs, rate-limit endpoints.

Day 2 — Performance & cost tuning

* Add caching layer for embeddings & search results (Redis).
* Add concurrency limits and queueing (Celery / BackgroundTasks) for heavy ingestion.
* Tune LLM parameters (temperature, max tokens) and retrieval k to balance cost & quality.

Day 3 — End-to-end tests & evaluation

* Write tests:
  + Unit tests for loaders, splitters, tools.
  + Integration tests for RAG pipeline.
* Create an evaluation set of queries with expected citations. Measure:
  + Citation coverage (% claims with supporting chunk)
  + Answer quality (manual grading)
  + Latency (median response time)

Day 4 — Deploy

* Backend: containerize (Docker), deploy to Render / Railway / AWS ECS.
* Frontend: deploy to Vercel (Next.js).
* Vector DB: switch to managed Pinecone or Chroma Cloud for production.
* Configure environment variables and secrets.

Day 5 — Monitoring & logs

* Setup Sentry for errors, Prometheus / Grafana for metrics (request latency, error rate), and cost-monitoring for LLM usage.

Day 6 — Final polish & demo

* Create README, architecture diagram, and a 3–5 minute demo video.
* Prepare a short case-study document describing problem, architecture, challenges solved, and results.

Success criteria (end of week): deployed service, tests & monitoring in place, demo assets ready for portfolio.

**Exact Minimal Code/Pseudocode (core RAG flow)**

Keep this short — a template to implement quickly.

# pseudocode

from langchain.document\_loaders import PDFLoader, YoutubeLoader, WebLoader

from langchain.text\_splitter import RecursiveCharacterTextSplitter

from langchain.embeddings import OpenAIEmbeddings

from langchain.vectorstores import FAISS

from langchain.chains import RetrievalQA

from langchain.llms import OpenAI

# 1. Ingest

docs = PDFLoader("file.pdf").load()

chunks = RecursiveCharacterTextSplitter(chunk\_size=800, overlap=150).split\_documents(docs)

emb = OpenAIEmbeddings()

index = FAISS.from\_documents(chunks, emb)

# 2. RAG

retriever = index.as\_retriever(search\_k=6)

llm = OpenAI(temperature=0)

qa = RetrievalQA(llm=llm, retriever=retriever, return\_source\_documents=True)

res = qa.run("Explain X and cite sources")

(Adjust classes to match latest LangChain API.)

**Tests & Acceptance (how you’ll grade success)**

* Ingestion: can ingest 20 documents and search them reliably.
* RAG quality: for 20 test queries, **≥ 80%** of factual claims in answers have supporting citations (manual check).
* Latency: median time for a single small query ≤ 8s (depends on model).
* UX: user can upload a PDF, run a query, see streaming progress, and export a report.

**Stretch features (optional, high-impact)**

* Agent Auto-Research: agent loops calling tools until confidence threshold reached.
* Automatic claim verification using external APIs (fact-check step).
* Multi-LLM ensemble: small model for retrieval summarization + large model for synthesis.
* Browser extension that sends highlighted text to the assistant.
* Scheduler: run periodic knowledge updates (daily RSS ingestion + re-embedding).

**Deliverables to include in your portfolio**

* Public GitHub repo with clear README, architecture diagram, and deploy instructions.
* Live demo link + short demo video (3–5min) walking through features.
* Notebook or report with evaluation metrics and model choices.
* Screenshots of UI and example queries with citations.

If you want, I can now:

* generate the exact requirements.txt, Dockerfile, and docker-compose.yml, **or**
* give you the full FastAPI + LangChain skeleton code for the backend (production-ready), **or**
* convert this plan into a day-by-day Trello/Notion checklist you can tick off.

Which of those would you like next?