ChatGPT

Building a Company Research Assistant AI – Implementation Roadmap

This guide outlines a **step-by-step plan** to build a conversational AI (chat/voice) that researches companies and generates editable account plans. It covers technology choices, key AI concepts and resources, conversation design for varied personas, architecture for agentic behavior and mid-task updates, and deployment/demonstration best practices. Each section cites relevant sources on tools and methods.

1. Technology Stack & Frameworks

- **Core Language & Frameworks:** Use **Python** for AI logic (rich libraries for ML/NLP) and optionally **JavaScript/TypeScript** for frontend UI. Python frameworks like **FastAPI** (for serving the agent) and **Streamlit** or **Gradio** (for quick web UIs) are recommended. For example, FastAPI is a modern, high-performance web framework for building APIs in Python [1], and the Real Python tutorial serves a LangChain agent via FastAPI with a Streamlit chat UI [2].
- **LLM & Orchestration:** Use an open-source LLM (e.g. Meta's Llama 2, Mistral, GPT-J, etc.) via **Hugging Face Transformers** or the **LLama.cpp** interface, to avoid paid APIs. Or consider a free tier of services if allowed. Crucially, use an agent framework like **LangChain** to manage dialogue, tools, and retrieval chains. LangChain is an open-source library with pre-built agent architectures and integrations for any model or tool [3] [4], making it ideal for building adaptive LLM-driven agents.
- **Retrieval/Vector Store:** For *Retrieval-Augmented Generation (RAG)*, use a vector database (e.g. **FAISS**, **Qdrant**, **Chroma**) to store document embeddings. Use libraries like **LangChain** or **ChromaDB** to handle embedding generation and similarity search. Embedding models (e.g. OpenAI/GPT-like or Hugging Face sentence-transformers) will convert text to vectors.
- **Web Scraping & Search:** To gather public data, use web scraping libraries (e.g. **Requests** + **BeautifulSoup**) or search APIs. LangChain's `WebBaseLoader` (using BeautifulSoup) can fetch and parse HTML from URLs into text [5]. For search, open tools like the `googlesearch-python` module or a free search API can fetch links.
- **Conversational/NLU Frameworks:** Optionally use **Rasa** or **Botpress** for NLU/dialog management. (Rasa is open-source and designed for dialogue handling, though it requires separate STT/TTS integration [6].) However, for an LLM-centric agent, you may handle context and intent via prompts and LangChain memory directly, without a separate NLU engine.
- **Speech I/O:** For voice, integrate an **Automatic Speech Recognition (ASR)** library and a **Text-to-Speech (TTS)** engine. For example, the Python **SpeechRecognition** library supports many ASR engines (Google, Azure, CMU Sphinx, **OpenAI Whisper** etc.) [7]. OpenAI's Whisper (or WhisperX) is a free, high-accuracy ASR that works offline. For TTS, use libraries like **gTTS** or **Coqui TTS** (Coqui TTS is an open-source toolkit with many pretrained models). Many voice-agent libraries (e.g. Vocode) explicitly support Whisper for STT and gTTS for TTS [7]. Alternatively, **PyAudio** can capture/stream audio, and gTTS can generate speech audio from text.
- **Persistence & Storage:** Store scraped documents and the generated plan (e.g. in JSON or a simple database). Use lightweight storage (SQLite or JSON files) since scale is small. Optionally containerize

components with Docker for easy deployment. The Real Python demo uses Docker Compose to orchestrate a FastAPI backend and Streamlit UI [2].
- **Other Tools:**
- **Vector Search:** FAISS (open-source) or hosted solutions (if free tier available).
- **Browser automation:** If needed for complex sites, use Selenium or Playwright.
- **Code Repos:** Host code on GitHub with a clear README (setup, design notes) as required.

In summary, a **Python+LangChain** stack with open-source LLMs and voice libraries aligns well with modern agent design and avoids paid APIs [3] [7]. This matches the "open and neutral" approach (no vendor lock-in, many integrations) that LangChain promotes [8].

2. Essential AI & LLM Study Topics

Before implementation, study these key areas:

- **Large Language Models (LLMs):** Understand how Transformers work. The free Hugging Face *LLM Course* is excellent for learning Transformers, tokenization, and fine-tuning basics [9]. It covers Hugging Face libraries (Transformers, Datasets) and is free and comprehensive [9]. Additional resources: Fast.ai's Practical Deep Learning for Coders and Stanford CS224N (if available) for background in deep learning/NLP.
- **Retrieval-Augmented Generation (RAG):** Learn how to augment LLMs with external data. A clear introduction is Agarwal's "Beginner's Guide to RAG" [10], which explains that RAG retrieves relevant documents to ground the LLM and reduce hallucinations. For hands-on practice, the Real Python tutorial "Build an LLM RAG Chatbot with LangChain" walks through indexing docs, creating a vectorstore, and answering queries [11]. These show how retrieved context is combined into prompts. Key takeaway: RAG helps the model fetch up-to-date facts [12] [11].
- **LangChain & AI Agents:** Study the LangChain framework, which provides tools for RAG, memory, and agent behaviors. The LangChain documentation and Academy have courses and guides [13] [3]. LangChain excels at connecting LLMs to tools and databases [4]. Its tutorials (including templates like ReAct or conversation chains) will teach you to structure queries and follow-up actions.
- **Prompt Engineering:** Learn to craft effective prompts for different tasks. Google's Prompt Engineering Guide defines it as "the art and science of designing and optimizing prompts to guide AI models" [14]. Key topics: zero-shot vs few-shot prompting, chain-of-thought prompting, instructing tone/format, and role-playing prompts. Practice by writing and iterating prompts to get the LLM to output structured account-plan sections or clarification questions.
- **Conversational Design:** Study dialogue flow techniques. The Botpress "Conversational AI Design" guide emphasizes blending **user research, natural language, and structured flows** [15]. Learn how to map user journeys, define bot persona and tone, and include recovery paths for off-script inputs [15]. This will help tailor responses to personas (see next section).
- **Speech Recognition & Synthesis:** Explore libraries for ASR and TTS. Tutorials on OpenAI's Whisper (e.g. Whisper's GitHub or blog posts) show how to transcribe audio in Python. For TTS, investigate **Coqui TTS** (GitHub) or **gTTS** (Python module). The Analytics Vidhya article on voice libraries lists options like Whisper for ASR and gTTS for TTS [7]; reviewing it can identify tools for your platform.
- **Agentic AI Patterns:** Read about LLM "agentic" systems (e.g. ReAct, AutoGPT, BabyAGI) to see how LLMs can plan and invoke tools. Though more advanced, understanding how to chain reasoning steps and include human feedback (e.g. through questions) will be valuable.

Resources: - **Courses:** Hugging Face LLM Course [9], LangChain Academy [13]. - **Tutorials:** RealPython RAG chatbot [11], LangChain docs on RAG/agents. - **Blogs/Guides:** Medium, blog posts on RAG/agents, OpenAI's prompt engineering guide [14].

3. Designing the Conversation Flow

Good conversation design ensures *conversational quality*. Follow these practices:

- **User Personas:** Plan for the specified personas. The assignment explicitly lists "Confused" (needs guidance), "Efficient" (wants concise answers), "Chatty" (goes off-topic), and "Edge Case" users [16]. Document strategies for each. For example, with an efficient user, skip small talk and provide bullet lists; with a chatty user, acknowledge their points but gently steer back. The Botpress guide advises blending **user research, natural language, and structured flows** to make bots feel human [15]. Use that advice to adapt tone (e.g. friendly vs formal) and handling (e.g. saying "I'm not sure about that, but I can find out!" for confusion).
- **Flow Structure:** Map out the conversation steps. Typical flow: User requests research on Company X → Agent acknowledges and starts retrieving data → Agent shares progress/questions → (optional user input/clarification) → Agent presents draft plan → User edits sections → Agent updates plan. Use diagrams or bullet lists to plan these turns. Include *fallbacks*: e.g., if input is unclear ("Could you clarify which section?"), or out-of-scope ("I don't have info on that right now").
- **Prompts & Memory:** Use conversation memory so the agent remembers context (company name, earlier answers). For example, the agent's first message might confirm understanding ("Searching for recent news on *Company X*..."). For a confused user, the agent can proactively ask clarifying questions about scope or priorities. For chatty users, you may implement polite interjections ("Let's see..."). Use prompt templates or few-shot examples to set the agent's style and persona.
- **Reactivity:** When new information arrives (e.g. user says "By the way, also consider their EU operations"), the agent should integrate it seamlessly, revising the plan or searching additional sources. Design the dialogue manager to handle mid-conversation updates. Good conversation design includes *recovery paths* when things go off-script [15]. In practice, if the user becomes chatty or stray, the agent could say: "That's interesting! Now, regarding the company's product line...".

In sum, create a detailed dialogue flow that addresses each persona's needs and includes prompts to handle edits and off-topic remarks. Test variations until the conversation feels natural and goal-driven.

4. Agentic Behavior & Intermediate Updates

To satisfy **agentic behavior** and mid-task updates, architect the agent with multi-step reasoning and human-in-the-loop queries:

- **Tool Integration:** Give the LLM *tools* for information gathering. For example, define a **web search tool** or **scraper tool** that takes a query and returns text snippets. Using LangChain, you can register a search function as a `@tool` that the agent can call. This lets the agent decide to search the web as part of its reasoning (a ReAct pattern). See LangChain's tools docs: e.g. a DuckDuckGo or Google Search tool allows the agent to fetch real-time data.
- **Planning & Reasoning:** Use chain-of-thought prompting or LangChain's AgentExecutor so the model can think step-by-step. After receiving the user's request, the agent might outline its approach ("Step 1: find company overview. Step 2: gather latest news. Step 3: compile plan sections."). You can

even extract these reasoning steps from the model's output. For example, the Real Python example captures "intermediate_steps" in its API response model [17], which could include the tools invoked.

- **User Queries & Conflicts:** Implement logic for the agent to communicate mid-task. For instance, if conflicting facts appear ("Source A says X, Source B says not-X"), the agent should inform the user (e.g. "I'm seeing contradictory reports on X; would you like me to dig deeper?"). This can be done by having the agent occasionally output messages like "Intermediate finding: ..." or by pausing to ask the user for guidance. Designing this requires the conversation loop to allow *agent-initiated* messages (not just user prompts).
- **Transparent Outputs:** When serving results, consider returning intermediate reasoning with the answer. The Real Python project's API defines a response schema with an `intermediate_steps` field alongside the final output [17]. This way, you can display the agent's reasoning or actions (e.g. the search queries it ran) if needed. It increases transparency and aligns with the requirement to "provide research updates during research."
- **Memory and Adaptation:** Keep track of what's been done. Use LangChain's memory modules or a simple session store to remember the company name and past findings, so the agent does not repeat work. Also allow edits: if the user says "revise the Strategy section," the agent should recall the existing plan and only regenerate that section.

By structuring the agent as a planning/execution loop (with search and summarization steps), and by coding the system to break in and query the user when needed, you create a truly *agentic* assistant that drives the task forward.

5. Building the Retrieval Pipeline

Core to this agent is gathering company info from public sources. Implement as follows:

- **Web/Data Collection:** For each company query, fetch data from multiple sources:
- **Company Website:** Use Requests + BeautifulSoup (or LangChain's `WebBaseLoader`) to scrape the "About" page, press releases, product pages, etc. LangChain's `WebBaseLoader` can load HTML and extract text using rules [5]. For example, you might target tags like headers or paragraphs.
- **News Articles:** Use a free news API or scrape news sites. You could call RSS feeds or Google News (with a custom scraper). Tools like `newspaper3k` can simplify scraping news articles by URL.
- **Profiles & Social:** Scrape or query public LinkedIn company pages, Crunchbase, or Wikipedia (which are public) for summaries. (Be mindful of robots.txt and usage policies.)

- **Search APIs:** If allowed, a free search API (like Bing Search with limited calls) can find additional URLs. Otherwise, a Python search library (`googlesearch`) can retrieve links to crawl.

- **Document Processing:** Once you have raw text, break it into chunks. Use a text splitter (e.g. LangChain's `RecursiveCharacterTextSplitter`) to divide large documents into ~500-1000 character pieces. This aids retrieval.

- **Embeddings & Indexing:** Convert each chunk into a vector embedding. You can use a SentenceTransformer model (e.g. `all-MiniLM`) or an open embedding model. Store these in your vector database (e.g. FAISS index) along with metadata (source, URL, snippet).

- **Similarity Search:** When the agent queries for info (e.g. “Company X competitors”), embed that query and retrieve the top-N relevant chunks from the vector store. These chunks become context for the LLM to generate answers. This is classic RAG: “retrieve, then generate” [10] [11].
- **Updating the Index:** If during conversation new URLs are discovered (user mentions a link, or new news appears), consider indexing them on-the-fly and re-running searches.

Ensure you handle errors (missing pages, no results) gracefully. Log each retrieval step so you can review what the agent found. Over-engineering beyond a simple index is unnecessary for demo scale, but following the RAG pattern (index-pull-generation) is essential for accuracy [12] [11].

6. Integrating Voice Interaction

To support voice, incorporate **Speech-to-Text (ASR)** and **Text-to-Speech (TTS)** components alongside the chat interface:

- **ASR (Speech Recognition):** Capture user speech via microphone (e.g. PyAudio). Pass the audio to Whisper or a similar model. OpenAI’s Whisper can be installed via `pip install -U openai-whisper`. It transcribes audio to text with high accuracy. Alternatively, the Python `speech_recognition` library can use free engines (like Google’s) [7], but Whisper is fully offline and self-contained.
- **Voice Command Handling:** Once speech is transcribed to text, feed it into the same chat pipeline. You may prepend a system prompt like “The user spoke this sentence:” to maintain context.
- **TTS (Speech Synthesis):** After the agent generates a text response, convert it to speech. Libraries:
- **gTTS:** Google’s free TTS API via Python. It generates an MP3 file for text.
- **Coqui TTS or pyttsx3:** For a fully offline, high-quality voice. Coqui has many pretrained voices but is heavier to set up.
- **Streaming:** For real-time interaction, stream the audio to the user. Even a simple “playback MP3” in a GUI suffices for demo.
- **Integration:** Frameworks like Vocode streamline ASR+TTS+conversation. Vocode, for instance, “makes integration of speech recognition, text-to-speech, and conversation AI easy” and supports Whisper and gTTS [7]. Consider using such a framework to save development time. Otherwise, tie together PyAudio for input, Whisper for transcription, pass text to LangChain, then use gTTS to output audio.
- **Voice vs Chat Mode:** Provide a UI toggle or command to switch between chat and voice (e.g. a “talk” button). Ensure both modes update the same conversation state.

By using open-source ASR/TTS and connecting them to the agent’s text logic, you fulfill the “voice or chat” requirement without paid services. The key is seamless conversion between speech and text.

7. Interactive Output & Editing

The final account plan should be **interactive and editable**:

- **Structured Output:** Organize the account plan into labeled sections (e.g. *Company Overview, Products/Services, Customers, Competitors, Strategy*). Have the agent output it in a structured format (Markdown or JSON) so sections can be easily identified. For example, the LLM could output a Markdown document with headers for each section.

- **User Editing:** Allow the user to request edits by referring to section names. For instance, "Revise the *Products* section to focus on recent updates." The agent can then re-run its generation on just that section. In practice, store the original response's sections and when a user commands an edit, send a follow-up prompt like: "Regenerate the *X* section with this additional context: ...". LangChain makes it easy to reuse output chunks.
- **Mid-Task Updates:** If the user adds info mid-way ("We forgot to include recent acquisition news"), pause the plan generation and retrieve new data to incorporate. You might insert a new retrieval step for the updated query, then merge results.
- **UI Considerations:** If building a chat UI, consider formatting the plan as interactive elements. For example, each section could be in its own chat bubble with "Edit" buttons (if using a custom UI). In a minimal interface (like Streamlit), simply accept text commands referencing sections.
- **Persistence:** Keep the current version of the plan in memory or a temporary store, so edits accumulate. Each time the user updates a section, rebuild the full plan document with the new content.

This design ensures the user can refine any part of the plan and that the agent dynamically updates outputs, demonstrating **adaptability**. It also satisfies "allow users to update select sections" by giving clear hooks to re-run or refine sections on demand.

8. Testing with Personas & Conversation Quality

To meet the evaluation criteria, rigorously test the agent with different user profiles:

- **Confused User:** Test with someone who gives vague requests ("I'm not sure what I need."). The agent should ask clarifying questions and guide them (e.g. "What aspect of the company are you most interested in?"). Verify it doesn't just output garbage but seeks more info.
- **Efficient User:** Test with a user who is terse and impatient ("Just give me the facts."). Ensure the agent skips niceties and responds concisely. The response style could be bullet points or a short summary.
- **Chatty User:** Have a user who goes off-topic or provides lots of irrelevant chatter. The agent should politely acknowledge ("That sounds interesting"), then steer back ("Now, regarding the company's strategy..."). Ensure conversation *flow* remains on target.
- **Edge Cases:** Feed invalid or unexpected inputs (nonsense text, irrelevant requests, or overly broad queries). The agent should handle gracefully: e.g. "I'm sorry, I can't help with that," or try to salvage ("I don't have information on that, but I can tell you about [some related topic]"). The design tip from Botpress: build in **recovery paths** for off-script conversation [15].
- **Conversational Quality:** Evaluate if the dialogue feels natural. Follow Botpress's advice to make the bot feel human by using a friendly tone and connecting logically [15]. Keep utterances clear and varied.
- **Real Users:** Have colleagues role-play these personas. Gather feedback on clarity, helpfulness, and whether intermediate updates (conflict notices, etc.) are handled well. Iterate prompts and flows accordingly.

By specifically testing these scenarios, you align with the assignment's suggestion to present demos of *Confused/Efficient/Chatty/Edge-Case users* [16]. Document these cases in your README or demo script.

9. Deployment & Demo Preparation

Finally, prepare the system for presentation:

- **Codebase & GitHub:** Organize your code with clear structure (e.g. separate `backend/` and `frontend/`, or `agent/`, `api/`, `ui/`). Write a comprehensive README that covers setup (install libraries, download models), architecture diagrams, and design rationale. Include instructions for the tool/scraper to respect robots.txt if needed. Use version control (GitHub) and ensure all code is pushable and runs from scratch.
- **Containerization (Optional):** Package the backend (APIs, agent logic) in a Docker container. If you built a UI, containerize it too. Real Python's example uses Docker Compose with one service for the FastAPI app and another for the Streamlit UI [2]. This makes deployment easy (just `docker-compose up`).
- **Voice/Chat Demo:** Record a demo video (≤10 min) showing both chat and voice interactions. Demonstrate key features: multi-source research, conflict notice ("I found two different figures for revenue, see?", etc.), user editing a section, and persona handling. Use a screen recorder and narrate your actions. Ensure code runs live (avoid slides). A good flow: start with a simple query, then show a confused user scenario, then an edit, etc.
- **Environment:** If using large models, note their requirements. For demo, you might run models locally or via free Hugging Face inference. Ensure the system runs within reasonable latency (use smaller models if needed to avoid long waits).
- **Submission Checklist:** The assignment requires a public GitHub link (with code and README) and a demo video. Triple-check that the repo is accessible and the video covers the agent's capabilities.

By following these steps, you'll have a working agent and a polished demonstration aligning with Eightfold's assignment requirements.

10. Aligning with Evaluation Criteria

Ensure your design addresses **all evaluation dimensions**:

- **Conversational Quality:** Use clear, engaging language and recover from errors. Leverage conversation design best practices [15]. Show that the bot adapts to user style (concise when needed, elaborative when prompted). Use turn-taking and back-channel cues ("Okay... Got it") to feel natural.
- **Agentic Behavior:** Demonstrate autonomy in research. For example, have the agent proactively say "I'm looking up their latest news..." and later "I found conflicting info – how should I proceed?" Include at least one example of the agent taking initiative or asking for feedback. The ability to call tools (search/scraper) illustrates agentic tool use.
- **Technical Implementation:** Write modular, documented code. Use appropriate frameworks (FastAPI, LangChain) as cited above [1] [3]. Follow best practices: asynchronous calls for API (as RealPython suggests), error handling (e.g. retries) [1], and version control. Even though it's a demo, ensure it's robust (won't crash on common inputs).
- **Intelligence & Adaptability:** The RAG pipeline ensures the agent has up-to-date factual info [12]. The agent should refine answers as new data arrives or as the user edits. If you can, include a short test showing the agent learns from a user edit (e.g. user adds a new requirement and the plan updates). This demonstrates the agent isn't static.

- **Documenting Decisions:** In your README, explain why you chose each technology (e.g. “We used LangChain because it simplifies building agents [3], and Whisper for ASR due to its accuracy [7]”). This transparent reasoning aligns with “document design decisions” encouragement in the assignment.

By explicitly linking your implementation back to these criteria – and using the cited best practices – you will tightly satisfy the assignment rubric. Good luck!

Sources: We have referenced authoritative guides and docs for RAG, LangChain, speech libraries, and conversation design [10] [3] [15] [7] [11] [2] [17] [14] [13] [9]. Use them to deepen your implementation knowledge.

---

[1] [2] [11] [17] Build an LLM RAG Chatbot With LangChain – Real Python
https://realpython.com/build-llm-rag-chatbot-with-langchain/

[3] [4] [8] LangChain
https://www.langchain.com/langchain

[5] Build a RAG agent with LangChain - Docs by LangChain
https://docs.langchain.com/oss/python/langchain/rag

[6] [7] Top 10 Open Source Python Libraries for Voice Agents
https://www.analyticsvidhya.com/blog/2025/03/python-libraries-for-building-voice-agents/

[9] Introduction - Hugging Face LLM Course
https://huggingface.co/learn/llm-course/en/chapter1/1

[10] [12] A Beginner’s Guide to Retrieval-Augmented Generation (RAG) | by Nikunj Agarwal | Medium
https://medium.com/@nikunj.agarwal012/a-beginners-guide-to-retrieval-augmented-generation-rag-8d9920c6e6ae

[13] LangChain Academy
https://academy.langchain.com/

[14] Prompt Engineering for AI Guide | Google Cloud
https://cloud.google.com/discover/what-is-prompt-engineering

[15] Conversational AI Design in 2025 (According to Experts)
https://botpress.com/blog/conversation-design

[16] AI Agent Building Assignment - Eightfold.pdf
file://file_00000000cd9472098a2d687b11f8af6d