

Project Approach: Autonomous Research Assistant

This document outlines the technical architecture and strategy for building **HelpMe**, an autonomous agent system designed to answer complex user queries by researching, synthesizing, and citing information from online sources.

Objective: To build a system that can take a query, such as *"What is the RICE scoring model for prioritization, and how is it different from the Kano model?"*, and produce a single, reliable, and well-cited answer.

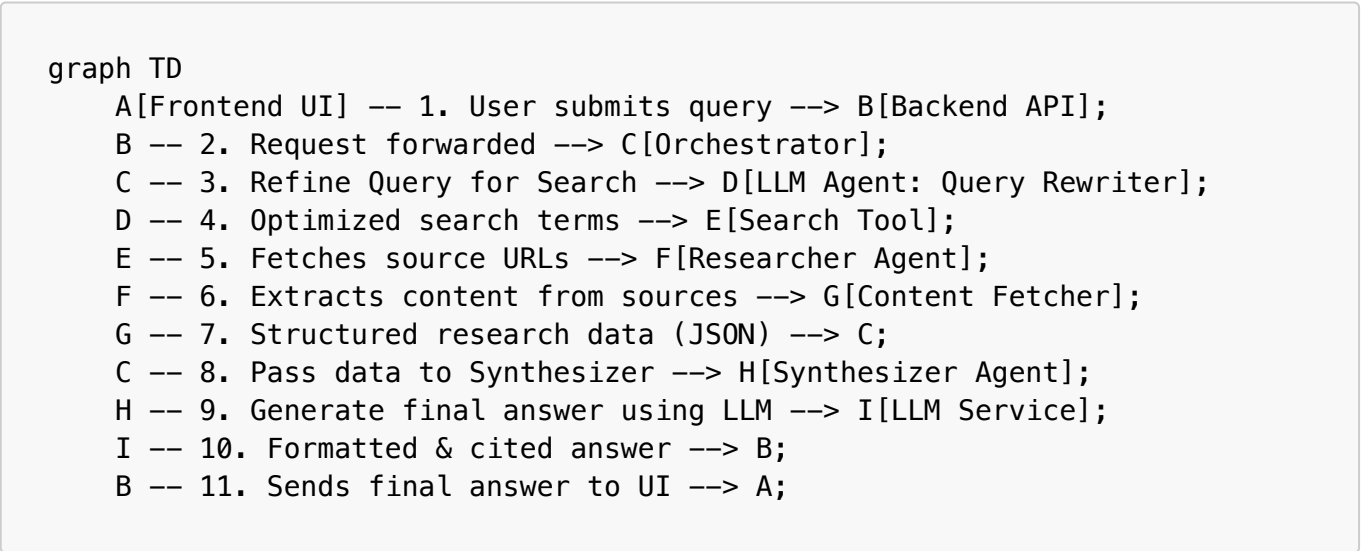
1. High-Level Architecture

The system is built on a decoupled client-server model to ensure a clean separation of concerns, scalability, and maintainability.

- **Frontend:** A **React** Single-Page Application (SPA) provides a clean, responsive user interface for a student or user to submit their question.
- **Backend:** A high-performance **Python API server using FastAPI** orchestrates the multi-agent research and synthesis workflow.
- **External Services:** The backend integrates with third-party APIs for Large Language Models (LLMs) and web search.

Core Workflow: From Question to Answer

The system follows a structured, multi-step process to handle a user's query, mimicking the workflow of a human research assistant.



2. The Autonomous Agent System

The core of the backend is a multi-agent system where each agent has a specialized role. This modular design makes the logic easy to follow, debug, and extend.

1. **Orchestrator:** The central controller that manages the entire workflow, passing the query and data between the other agents and tools.
 2. **Researcher Agent:** This agent's primary responsibility is to find and process reliable information.
 - **Query Refinement:** It first uses an LLM with a specialized prompt (`query_rewriter_system.txt`) to transform the user's natural language question into a set of concise, keyword-driven search queries. This is a critical step for improving the relevance of search results.
 - **Source Aggregation:** It uses a search tool (defaulting to DuckDuckGo) to find relevant articles and URLs.
 - **Content Extraction:** For each URL, it uses the `newspaper3k` library to intelligently extract the core article content, filtering out ads and boilerplate. To avoid being blocked, the fetcher identifies itself with a standard browser `User-Agent` header. All sources are fetched concurrently to minimize latency.
 3. **Synthesizer Agent:** This agent's role is to transform the structured JSON data from the Researcher into a high-quality, human-readable answer.
 - **Prompt Engineering:** The agent uses a carefully crafted system prompt (`synthesizer_system.txt`) that instructs the LLM to act as a professional research analyst.
 - **Core Directives:** The prompt enforces strict rules:
 1. Base the answer **only** on the provided source data to prevent hallucinations.
 2. Meticulously cite sources using a clear format (e.g., `[1]`, `[2]`).
 3. Follow strict formatting rules (Markdown) to ensure the frontend can correctly render the answer and link citations.
-

3. Key Technical Decisions

The technology stack was chosen to prioritize performance, user experience, and maintainability.

- **Backend (FastAPI & Python):** FastAPI was chosen for its high performance, asynchronous capabilities, and automatic API documentation. Python's extensive ecosystem of data processing and AI libraries makes it the ideal choice for the backend logic.
 - **Frontend (React & Vite):** React allows for the creation of a dynamic and component-based UI. Vite provides a fast and efficient development experience. The application logic is separated into custom React Hooks (`useQueryHistory`, `useResearch`) to keep the UI components clean and maintainable.
 - **LLM Agnostic Design:** The system is designed to be flexible in its use of LLMs.
 - **Default (OpenRouter):** It defaults to OpenRouter, an aggregator that provides access to a wide variety of models, including free, open-source options. This ensures the application can be run without requiring paid API keys.
 - **Alternative (Google Gemini):** Native support for Google's Gemini models is also included for users who require higher-quality synthesis.
 - The user can switch between any configured provider and model directly from the UI.
-

4. Setup and Run Instructions

Note: These instructions are also available in the main [README.md](#) file.

HelpMe

Your smart tool for deep-dive research.

What is the RICE scoring model for prioritization, and how is it different from the Kano model?

Research

openrouter

deepseek/deepseek-chat-v3.1:free

Recent Queries

Answer

CopyDownload

The RICE scoring model and Kano model are both feature prioritization frameworks used in product management, but they employ fundamentally different approaches to decision-making ^[1].

RICE Scoring Model

The RICE model provides a quantitative scoring system based on four components:

- Reach:** Estimates how many users or customers a feature will affect within a specific timeframe
- Impact:** Measures the effect the feature will have on individual users
- Confidence:** Assesses how certain the team is about their estimates
- Effort:** Estimates the resources required to implement the feature^[2]^[3]^[5]

The RICE score is calculated by multiplying Reach, Impact, and Confidence, then dividing by Effort, resulting in a numerical value that allows for direct comparison between features ^[3]. However, some product managers note that Reach and Confidence components can be subjective, and some practitioners focus more on the Impact vs Effort ratio rather than using the full RICE formula ^[6].

Kano Model

The Kano model takes a categorical approach, classifying features based on their impact on customer satisfaction. Features are categorized into types such as:

- Basic:** Expected features that cause dissatisfaction when absent but don't necessarily increase satisfaction when present
- Performance:** Features where satisfaction increases linearly with better performance
- Excitement:** Unexpected features that delight customers but don't cause dissatisfaction if missing
- Indifferent:** Features that don't significantly affect satisfaction
- Reverse:** Features that actually decrease satisfaction when implemented^[4]

Unlike RICE's numerical scoring, Kano provides qualitative classification that helps understand how different feature types contribute to overall customer experience ^[1] ^[4].

Key Differences and Applications

The primary distinction lies in their methodological approach: RICE offers quantitative scoring for direct comparison, while Kano provides categorical classification based on customer satisfaction impact ^[1] ^[4]. RICE is particularly useful for comparing features with measurable outcomes and resource requirements, while Kano helps understand the emotional and psychological impact of features on users ^[1].

Both frameworks help transform subjective opinions into more objective decisions and balance customer value against business impact [1]. However, no single framework is perfect, and product managers should avoid becoming overly dependent on any single model [2]. The choice between RICE and Kano depends on the specific context, product stage, and the type of insights needed for decision-making [4] [5].

Sources

 nextsprints.com

Feature Prioritization Frameworks: A Deep Dive into RICE, MoSCoW, Kano & More

 www.prodpad.com

Which Prioritization Model Is Best for My Product?

 blog.clivelancaster.com

The Art of Prioritization: Comparing Popular Prioritization Frameworks

 medium.com

Prioritisation Frameworks (MoSCoW, KANO, RICE)

 www.linkedin.com

How do you prioritize features using the Kano model or the RICE framework?

>_ Raw Research JSON

Prerequisites

- Python 3.10+ and **pip**
- Node.js 18+ and **npm**

Installation & Setup

1. **Clone the repository and navigate into it.**
2. **Backend:** Create a virtual environment, activate it, and run **pip install -r requirements.txt**.
3. **Frontend:** Navigate to the **frontend** directory and run **npm install**.
4. **API Keys:** Create a **.env** file from **.env.example** and add your API keys for Gemini and/or OpenRouter.

Running the Application

1. **Run Backend Server:** **uvicorn backend.main:app --reload**
2. **Run Frontend Server:** **cd frontend && npm run dev**

The application will be available at **http://localhost:5173**.