

AI Agent Interaction Log (Modular Architecture):

Me:" I want to start with building an AI agent Prototype that can perform manual task from daily life or university work, can reason, plan, and execute to automate it.

I am thinking of going for an AI-powered 'Study Assistant'. The mandatory core features and optional features of the agent are provided in the file that I have attached with this prompt.The core requirements seem to be automating a task and using at least one fine-tuned model.

Use a modern web framework (like Chainlit). The assistant must be able to perform multiple, academic tasks

- 1) Explain concepts
- 2) Generate quizzes
- 3) Generate Flashcards
- 4) Analyze code.
- 5) Solve maths

The system should be designed using an agentic framework, where a 'master agent' delegates tasks to 'specialist agents'.

I want to build this with a professional, modular architecture. Instead of putting all the prompts in one file, could we create:

1. A agent_config.py file to define all agent prompts, names, and UI elements (like icons and starter messages)?
2. A agent_definitions.py file that builds the agents from that config file?"
- 3.

Gemini: "Hi there. Thanks for sharing the doc. That's a great project, and your "Study Assistant" idea is a perfect fit for automating university work.That's an excellent design choice.

Let's start by creating agent_config.py. We'll use a Python dataclass to hold the configuration for each specialist agent. We'll also define the configuration for the master "Orchestrator" agent.

New File: agent_config.py"

CODE PROVIDED....

Me: "Now, how does the `agent_definitions.py` file use this config to actually create the agent objects?"

Gemini: "Great question. `agent_definitions.py` will act as our "factory." It will import the `SPECIALIST_AGENTS_CONFIG` list, loop through it, and instantiate an Agent object for each configuration. Then, it creates the master agent and gives it the list of specialist agents to handoff to."

New File: `agent_definitions.py` "

CODE PROVIDED...

Me: "the agent architecture is set. Now for the mandatory "fine-tuned model" part. I need to summarize academic lectures. I want to fine-tune a **T5-small model** on an academic paper dataset. What's the general process for doing that?"

Gemini: "That's the perfect way to meet the requirement. Fine-tuning your own T5 model will give you a highly specialized tool. You'd typically do this in a Google Colab notebook."

The high-level process is:

1. Get Data: You need a dataset of documents and their corresponding summaries (e.g., the ArXiv or XSum dataset).
2. Load Model & Tokenizer: Use the transformers library to load `T5ForConditionalGeneration` and `T5Tokenizer` from "t5-small".
3. Preprocess Data: Create a custom Dataset class. Your `__getitem__` function will tokenize the input text (the paper) and the target text (the summary), adding the prefix "summarize: " to the input, which T5 expects.
4. Set Training Arguments: Use `TrainingArguments` to define the output directory, learning rate, number of epochs (e.g., 3-5), batch size, etc.
5. Train: Use the Trainer class, passing it your model, training args, and dataset, then call `trainer.train()`.
6. Save Model: Once training is done, call `model.save_pretrained("./my-t5-summarizer")` to save your new model files.

Here is the basic template to follow :"

CODE PROVIDED...

Me: " Great, I've run the Colab notebook and my fine-tuned model is saved in a folder named .How do I get this onto the **Hugging Face Hub** so I can use it in my project?

Gemini: : Excellent. Uploading it is straightforward.

1. First, install the `huggingface_hub` library: `pip install huggingface_hub`.
2. In your terminal (or Colab), run `huggingface-cli login` and paste in an access token from your Hugging Face account.
3. In your Python script (or notebook cell), you can use the `HfApi` to upload your folder:
Now your model is live on the Hub at `Your-HF-Username/my-t5-summarizer`.

Me: “ my model is now on the Hub. I want to deploy this as a production-ready API. I've heard **Modal** is the best way to do this.t.

Can you give me the `modal_summarizer.py` script that will pull my specific fine-tuned model from the Hub and serve it as an API?”

Gemini: “You're right, Modal is perfect for this. It will pull your model directly from the Hub and serve it on a GPU endpoint.

Here is the complete `modal_summarizer.py` script. I've updated the `model_name` to point to the Hugging Face repo you just created.”

New File: `modal_summarizer.py`

CODE PROVIDED...

When you run `modal deploy modal_summarizer.py`, Modal will give you a public URL.”

Me: “I've deployed it and I have the URL.

Now I need to create the client-side `tools.py` file that `agent_config.py` was importing. How do I make this tool call my new Modal URL?

Gemini: ” This is the client-side integration. We'll create `tools.py` to define `invoke_summarization_service`. Based on your `main.py` file, the best way to handle this is to have the tool read the URL directly from the Chainlit user session.”

You'll need `httpx` for this: `pip install httpx`

New File: `tools.py`”

CODE PROVIDED...

Me: " This is all connected now. agent_config.py defines the agents, agent_definitions.py builds them, modal_summarizer.py hosts my fine-tuned model, and tools.py calls it.

Now, how do I create the main main.py Chainlit app to run this whole system?"

Gemini: " This is the final step. The main.py file will tie everything together. It will:

1. Handle chainlit callbacks.
2. Set up your AI model connection.
3. Call create_master_agent() to build the agent network.
4. Crucially, it will set your modal_url in the session so your tool can find it.
5. Handle file uploads and stream the response.

New File: main.py"

Me: "It works!

For the last part of my assignment, I need to add the "evaluation metrics" to my report. How do I evaluate the quality of my Summarizer?"

Gemini: "This is a critical part of your data science report. Since you fine-tuned your own model, this evaluation is key. For summarization, the standard quantitative metric is **ROUGE**.

Here's the methodology:

1. Test Set: Get 20-30 academic papers that were not in your fine-tuning training set.
2. Ground Truth: Use the actual abstracts of these papers as your "reference" (ground truth) summaries.
3. Generate Candidates: Run each paper's full text through your Chainlit app. Save the exact summary your Summarizer agent produces. These are your "candidate" summaries.
4. Calculate ROUGE: Use the evaluate library from Hugging Face to compare them. pip install evaluate rouge_score

Here's a code snippet you can use in a separate script to get your scores."

CODE PROVIDED...

A Note on Development: This interaction log captures the high-level design, but please be aware that the final code involved a significant debugging and refinement cycle not fully reflected here.

For the UI/UX interface, we have used **Chainlit**. It manages the entire frontend, including the chat profiles, starter prompts, message streaming, and file upload handling.