Technical Design for Phase One Assistant

Retrieval-Augmented Generation and Log Analysis

Retrieval-Augmented Generation (RAG) will be a core mechanism to enhance the assistant's answers with up-to-date and domain-specific information. A RAG pipeline combines two steps: **Retrieval** of relevant data from external sources (documents, databases, logs, APIs) and **Generation** of an answer using that data[1]. In our case, when a user asks a question, the agent will automatically **collect needed information** before answering. This means searching internal knowledge bases, documentation, or even parsing system logs if relevant. For example, if the query involves troubleshooting, the agent can scan log files for error entries and pull those details into context. By integrating log data into the retrieval step, the assistant can analyze logs to explain errors or find root causes. All retrieved content (documents snippets, log lines, etc.) will be fed into the LLM so that the final answer is **grounded in real data** rather than just the model's memory. This approach ensures context-specific, up-to-date responses instead of relying only on static pre-trained knowledge[2][1].

To implement RAG efficiently, we will use an **embedding-based search** (vector database) or a specialized API (such as *DeepSeek* if it provides semantic search). Relevant documents and logs can be indexed as vectors so the agent can semantically find the most relevant pieces when a question is asked. The orchestrator component will handle this retrieval step: upon receiving a query, it issues a search (to the vector store or via DeepSeek's interface) to get top relevant text chunks. This retrieved context is then appended to the prompt for the LLM. By **automating data collection**, the agent proactively gathers the information needed to answer the question without requiring the user to provide it explicitly. In summary, the Phase One assistant will leverage RAG to pull in data (including log entries) on the fly, enabling it to perform basic **log analysis** and knowledge lookups as part of its normal question-answering workflow.

Unified Tool Invocation Mechanism

To give the assistant robust capabilities, we will design a **unified tool-calling interface**. This means every external function or API the agent can use (whether it's a calculator, a web search, or a file reader) will follow a consistent **call protocol**. In practice, we'll maintain a registry of tools, each defined with a name, a description of its purpose, and the parameters it accepts. This structured approach is in line with how modern AI agents incorporate tools. For example, OpenAI's and Google's latest APIs allow developers to define functions (tools) with schemas, which the model can choose to call when needed[3]. By providing the model with a list of available tool "function" definitions, the agent effectively bridges natural language to real-world actions: the model will

decide if a tool is needed and return a structured call with the tool name and arguments, instead of a direct answer[4]. Our system will parse such outputs and execute the corresponding function, then return the result to the model for final answer formulation.

If the model or API supports native function calling (as GPT-4 and Google Gemini do), we will leverage that. In those cases, we supply the tool definitions (name, params, docstring) to the model upfront, and the model's response may include a function call JSON. The application (our orchestrator) checks for this and executes the tool accordingly[5][6]. This approach provides a safe and unified interface for all tools. If a model doesn't natively support this, we can fall back to a prompt-based convention – for instance, instructing the model to respond with a special format like `"Action: <ToolName>\nInput: <Parameters>"` – which our code can regex-match and interpret. Many agent frameworks (like LangChain) use a similar pattern where the model's output explicitly indicates a tool name and input. In either case, **each tool will have a clear natural-language description (for the model's benefit) and a deterministic invocation format**. This registration system ensures the agent knows what tools are available and how to call them. It also allows easy extension: adding a new tool is as simple as registering a new function with its metadata, and the model will be made aware of it in its prompt or function schema list[7]. This unified tool API not only organizes the tools but also simplifies permission control and logging – we can centrally manage how tools are invoked and keep track of their usage.

Key Tools and Capabilities to Include

In Phase One, the assistant should have a mix of **basic and advanced tools** to cover common personal assistant needs. Based on your preferences, we will include as many useful tools as possible (excluding direct system control for now). Below is a list of tool categories we plan to support:

- **Python Code Executor**: A sandboxed Python interpreter tool to perform calculations, run algorithms, or manipulate data. This covers everything from math calculations (acting as a calculator) to more complex data processing or accessing local data files. For example, if the user asks to plot a graph or perform analysis on provided data, the agent can write and execute a Python snippet to do that, then return the result (or even an image plot). This tool is extremely flexible and essentially gives the assistant "programmatic" abilities (similar to OpenAI's Code Interpreter concept[8]).

- **Date/Time Utility**: A tool that provides the current date and time, or does date arithmetic. This is straightforward but useful for scheduling questions, deadlines,

or any query where knowing “today’s date” or differences between dates is needed.

- **Web Search Engine**: Integration with a search API (Google, Bing, or any web search via DeepSeek if it offers web queries) to fetch live information from the internet. This allows the assistant to answer general knowledge questions, get latest news or facts, and retrieve any information that is not in its local knowledge. It will simply take a query string and return search results or a summary. (In the OpenAI ecosystem, this corresponds to their WebSearchTool[9]).

- **Document & File Search**: Tools to find and retrieve information from local files or knowledge bases. This can be twofold:

- *Semantic vector search* in a knowledge base (documents, wikis, etc.), which we covered under RAG. The agent can use an embedding-based **knowledge search tool** (possibly implemented via DeepSeek or a local vector DB query) to get relevant text passages.
- *Keyword or regex search* in files (like searching within log files or code repositories). For instance, a “grep_search” tool that given a regex or keyword returns matching lines from files[10]. This is useful for quickly scanning logs for errors or searching code/config files for a setting.
- *File reader*: Once we know a specific file (or log) is relevant, a tool to read the file (or a portion of it) and return contents is important. We will implement a `read_file` tool that can fetch file contents given a path (with safeguards on size). The agent can combine this with the search tools: first find candidate files or lines, then read the file to get full context if needed. (Similarly, a `list_files` or `list_dir` tool can help the agent navigate directories if we allow it.)

- **Email Access**: An email querying tool to read emails (and possibly send emails or draft responses). This would involve using an email API or IMAP client. For instance, the user could ask “Do I have any new emails from John yesterday?” and the agent could use this tool to fetch that information. Sending emails on user’s behalf could also be supported with proper confirmation. This turns the assistant into a productivity aide that can manage communications.

- **Web Scraper**: A tool to fetch the content of a webpage or API given a URL. While the web search tool gives high-level results, sometimes the task may require extracting specific data from a known webpage (e.g. “scrape the latest stock price from XYZ site”). A simple HTTP GET tool that returns page text (possibly with some filtering) can enable this. We should handle it carefully (avoid

unauthorized scraping), but it's a useful addition for real-time data retrieval beyond search results.

- **Data Visualization**: The ability to generate charts or graphs from data. We can achieve this via the Python tool (using libraries like matplotlib or others to create a chart image). The agent could then return the image to the user. For example, if asked to "plot a bar chart of quarterly sales", the agent could create the chart with Python and provide the image. This makes the assistant's responses more **multimodal** and useful for data analysis tasks. In Phase One, this would be facilitated through Python; we might not need a separate tool if the Python environment has the necessary libraries.

- **Other Advanced Tools**: We will design the system to be extensible, so additional tools can be plugged in as needed. For instance, in the future we could add **image analysis** (using an AI vision model to describe or interpret images) or **image generation** (to create images from prompts), since personal assistants sometimes handle such requests. In fact, the Cursor coding assistant example includes an image analysis tool (`query_images`) to describe images using an LLM with vision[11]. Another possible category is database queries or a natural language to SQL tool if the assistant needs to fetch structured data. While these are not immediate priorities, our unified framework will allow adding them easily later on. The only category we will **exclude in Phase One is direct system operations** (like executing arbitrary shell commands or controlling the OS) for safety and because it's not needed for your current goals. Every other capability that makes the assistant more helpful is on the table.

Each tool will be implemented with real functionality (where feasible) and carefully sandboxed for safety. The agent will choose when to invoke these tools based on the query. By equipping it with this rich toolset, we essentially create a **personal assistant that can not only answer questions with information (via search and RAG)** but also **take actions or perform tasks on behalf of the user** (via Python, email, etc.). This aligns with the direction of modern AI assistants, which go beyond Q&A to actually get things done[4]. All tools will be accessible through the unified interface described above, ensuring consistency in how the agent invokes them.

Integrating Gemini and DeepSeek for Generation

In Phase One, we will not deploy a local LLM; instead, we'll leverage external AI services – specifically **Gemini and DeepSeek** – to handle the natural language generation. You have API access to these models, and we will integrate them as the brains of the assistant. The idea is to use them for crafting the final answers (and

possibly intermediate reasoning steps), while our orchestrator manages the tools and retrieval around them.

Gemini is Google's advanced LLM, known for strong reasoning and multimodal capabilities, and it supports features like long context and even function calling as per Google's API. **DeepSeek** appears to be another LLM (from the context, a strong alternative for coding and general queries[12]). We will treat both as pluggable LLM backends. The system can be designed such that it's easy to switch between Gemini or DeepSeek for responses, or even use them in tandem (for example, one as a fallback or one specialized in certain domains).

The integration will be done through their respective APIs/SDKs. For Gemini, we can use Google's AI API (or Vertex AI if applicable) to send the conversation (system prompt, user prompt, retrieved context, etc.) and get the model's reply. For DeepSeek, we will use its provided API endpoint or SDK similarly. We will create an abstraction layer for the LLM calls – e.g., a `LLMClient` interface with methods like `generate_response(prompt, functions=None)` where the implementation can call either Gemini or DeepSeek under the hood depending on configuration. This way, if we want to experiment with both or switch over entirely, it's a one-line config change rather than a code overhaul. It also means the rest of the agent (tool use, retrieval logic) can remain the same regardless of which model is answering.

Notably, if Gemini supports the **function calling** interface and tool usage directly, we will take advantage of that by passing the function definitions (our tool schemas) to it. This would allow Gemini to decide on tool calls within a single API call cycle. If DeepSeek has similar functionality or a different format, we'll handle that in its integration. We should be prepared that the two models may have slightly different API behaviors (e.g., different ways to specify system instructions or functions). Our orchestrator can normalize this. For instance, LangChain's framework already abstracts many model APIs including OpenAI, Anthropic, and Google's – it even lists Google Gemini among supported models[13]. We might draw inspiration from such frameworks or even utilize them if appropriate, to manage these connections.

Security and cost will be considered: we'll ensure API keys for Gemini/DeepSeek are stored securely and calls are made efficiently (avoiding too many tokens by summarizing or truncating retrieval results as needed). Logging will be implemented around these calls for debugging. The use of two different LLM services also gives us flexibility: we can compare outputs, and in Phase Two when we transition to a local model, we can use the same interface to plug the new model in.

Phase One Workflow Overview

Bringing it all together, here is how the system will work in Phase One:

1. **User Query Processing**: The user asks a question or gives a task. The orchestrator (our agent controller) receives this input along with the conversation context so far (if any).

2. **Determine Retrieval Needs**: Before invoking the LLM, the orchestrator analyzes the query to see if external information is likely needed. For example, if the question is asking for some data ("What is the latest price of X stock?") or references specific documents/logs ("Check the error in yesterday's server log"), the agent will activate the retrieval tools. Using the RAG setup, it will query the relevant knowledge base or logs via the appropriate tool (this could be a vector search through DeepSeek or an internal index). It may also do a quick web search if the info is not in the internal data. The result of this step is a collection of **context snippets** (e.g., relevant paragraphs from docs, or log lines surrounding an error timestamp).

3. **Compose LLM Prompt**: The agent then prepares a prompt for the LLM. This typically includes a predefined system instruction (to set the assistant's role and style), the conversation history (if multi-turn), the user's latest question, and any retrieved context data (embedded either as part of the prompt or via the function calling mechanism). The prompt will also include the **tool function definitions** if we're using function calling. Essentially, at this point the LLM (Gemini or DeepSeek) is given everything it might need: the question and the extra information and tools to produce a good answer.

4. **LLM Reasoning and Tool Use**: The LLM processes the prompt. If the model can solve it with the given information directly, it will proceed to generate an answer. If it determines that an external action is needed (perhaps a calculation or an additional lookup), it will invoke a tool. In the case of function calling, the model's response will include a function name and arguments in JSON (e.g., it might decide to call the `python` tool with a certain code to run, or call the `web_search` tool for a follow-up query). If we are using a prompt-based approach (for a model without native function call support), the model might output a special formatted request like `"Action: Python\nInput: <code>"` – which our parser will recognize.

5. **Tool Execution**: When a tool invocation is received, the orchestrator executes the specified tool function. For example, if it was a Python code, we run that code in the sandbox and capture the output; if it was a web search action, we perform

the search and get results; if it was a file read, we retrieve the file content, etc. We then take the output from the tool and **feed it back to the LLM**. This is usually done by appending a new assistant message in the conversation with the tool result, or directly via the function-calling mechanism (where the result is passed as the function's return value). The LLM now has the outcome of the tool usage.

6. **Final Answer Generation**: The LLM continues the reasoning (it may iterate steps 4-5 if multiple tool calls are needed). Once it has all necessary information, it produces the final answer in natural language for the user. This answer will incorporate any retrieved facts or computed results from previous steps, ensuring accuracy and completeness. The orchestrator returns this answer to the user.

7. **Logging and Learning**: Throughout this process, we log the interactions (for debugging and for potential fine-tuning later). The retrieved documents or log snippets used can also be stored or indexed along with the conversation (creating a short-term memory of what information was pulled in). This could help avoid repeating searches in a multi-turn conversation. Moreover, this lays groundwork for Phase Two: we could use these interaction logs to further train a local model or update our knowledge base.

In summary, Phase One's design is a **modular pipeline**: user input -> (optional retrieval) -> LLM (with tool capabilities) -> tool execution -> LLM -> answer. The use of Gemini/DeepSeek ensures we have powerful language understanding and generation, while the tool and retrieval framework ensures factual accuracy, ability to take actions, and utilization of up-to-date data (like logs and web info). This approach will result in a very capable personal AI assistant that can answer questions with evidence, perform tasks (like calculations, file manipulations, sending info), and continuously improve.

Finally, this design is built with the future in mind. When we move to **Phase Two** with a locally deployed model (and possibly personalized training using accumulated data and hidden states), we can reuse the same architecture. The local model will be hooked into the same interface that we used for Gemini/DeepSeek. The tool registry and retrieval components will remain the same, simply the model endpoint changes. Because we've emphasized a unified tool interface and modular design, the transition to a fine-tuned local model in Phase Two should be smooth. We'll also be able to inject the personalized data (embeddings from user-specific content, refined log analysis patterns, etc.) into the RAG component to further improve answers.

This comprehensive Phase One plan (RAG + tool use + external LLMs) sets a strong foundation for a powerful AI assistant, and it can be iteratively enhanced as we proceed to Phase Two and beyond. The system will be ready to **answer questions, analyze**

your logs, gather information automatically, and call tools to accomplish tasks – all using the best of current AI services and a solid software architecture to orchestrate them.[7][4]

---

[1] [2] RAG Pipeline Diagram: How to Augment LLMs With Your Data

https://www.multimodal.dev/post/rag-pipeline-diagram

[3] [4] [5] [6] Function calling with the Gemini API | Google AI for Developers

https://ai.google.dev/gemini-api/docs/function-calling

[7] [8] [9] Tools - OpenAI Agents SDK

https://openai.github.io/openai-agents-python/tools/

[10] [11] GitHub - civai-technologies/cursor-agent: Cursor Agent Tools - A Python-based AI agent that replicates Cursor's coding assistant capabilities, enabling function calling, code generation, and intelligent coding assistance with Claude, OpenAI, and locally hosted Ollama models.

https://github.com/civai-technologies/cursor-agent

[12] tools.md

https://github.com/gauravmeena0708/IITD-Machine-Learning/blob/7f18a123ccb1da94f4d9d5dd76da074fc02ea2e1/tools.md

[13] Build an Agent | LangChain

https://python.langchain.com/docs/tutorials/agents/