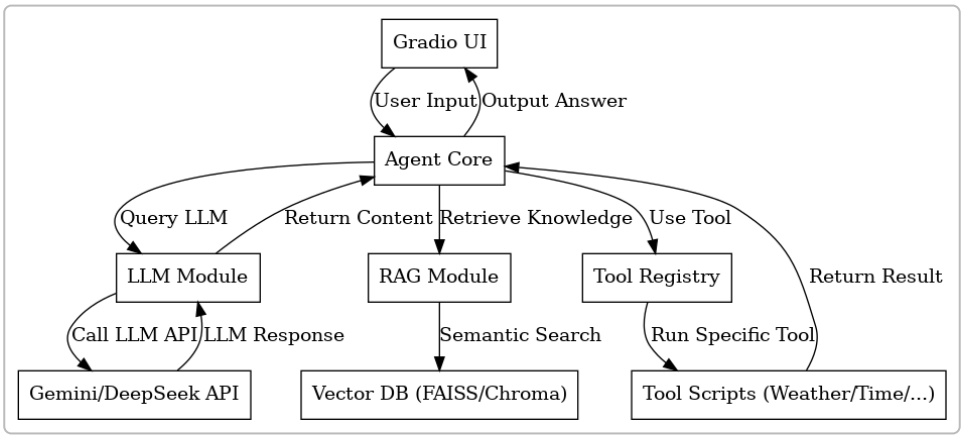
ChatGPT

AI个人助理 Agent MVP 脚本化模块化方案

模块划分与架构设计

该 AI 个人助理项目采用模块化设计，将各项功能拆分为独立的脚本模块，方便热插拔和后续扩展。主要模块包括：

- **LLM 接口模块**：对接 Gemini 和 DeepSeek 等大语言模型 API，负责发送用户请求并获取模型回复。只封装模型调用逻辑，不涉及业务流程。
- **RAG 检索模块**：基于本地向量数据库（如 FAISS/Chroma）实现检索增强生成。使用 JSON 格式的历史聊天记录作为语料，提供语义检索以增强模型上下文。
- **工具注册模块**：统一管理可用工具（天气、时间、计算器、日程、邮件、网页搜索、文件读取等）的元数据和调用接口。负责根据 LLM 请求选择并调用相应工具，将结果返回给主代理。
- **工具功能模块**：每个工具一个独立的 Python 脚本，实现各自最基础的功能。例如天气查询、获取当前日期时间、数学计算、读取日程文件/日历事件、通过 IMAP 获取邮件、调用 SerpAPI 进行网络搜索、读取本地文件内容等。
- **UI 界面模块**：提供用户交互界面，默认使用 Gradio 搭建简易网页，对话窗口；必要时可扩展为 Telegram Bot 接口。该模块负责将用户输入传递给 Agent主流程，并显示模型回复。
- **Agent 主流程模块**：作为中枢协调上述模块，控制对话流程。接收 UI 输入后，调用 LLM 接口获取回复或函数调用指令，必要时通过工具注册模块调用工具或通过 RAG 模块检索知识，然后将最终答案返回给 UI。



图：AI 助理 Agent MVP 架构模块关系图，各模块（LLM 接口、RAG、工具注册、工具脚本、UI 界面）协同工作示意。用户通过 Gradio UI 提问，经由 Agent 主流程判断直接调用 LLM 或使用工具/RAG 辅助，然后由 LLM 生成答案反馈给用户。

上述架构中，**LLM 接口模块**、**RAG 模块**、**工具注册模块**、**工具脚本模块**和 **UI 模块**各司其职，Agent 主流程串联它们共同完成对话。下面分别说明各模块的功能边界和最简实现方式。

模块最简实现逻辑

1. LLM 接口模块（模型 API 封装）

功能边界：负责与远程大语言模型交互。基于配置决定使用 Gemini 或 DeepSeek（假设它们提供 OpenAI Chat API 接口或类似REST API），不处理业务逻辑。

实现逻辑：利用官方 SDK 或 HTTP 请求直接调用模型生成回复。推荐使用 OpenAI 提供的 Python SDK（openai 库）或相应的模型SDK，以简化调用流程。例如，对于 OpenAI 接口，可以直接使用 `openai.ChatCompletion.create(model=..., messages=...)` 调用聊天模型并获得回复。Gemini/DeepSeek 若兼容 ChatGPT API，可采取类似方法。模型函数调用（function calling）机制如被支持，可在发起请求时传入可用工具的函数描述列表，让模型决策是否调用工具 [1]。需要注意，ChatCompletion API 本身不会真实执行函数，而是返回一个函数名和参数的 JSON，需由代理代码解析后再实际调用相应函数 [1]。如果模型不支持原生函数调用，可改用提示词协议：例如约定当需要调用工具时，让模型以特定格式（如JSON或特殊前缀）输出请求，由主流程解析后调用工具。

使用工具包：建议使用**OpenAI 官方 SDK**或模型提供的API库，以便一两行代码即可完成调用。例如 OpenAI SDK 调用格式如下（伪代码）：

```
import openai
openai.api_key = "YOUR_KEY"
response = openai.ChatCompletion.create(
    model="gpt-4",
    messages=[{"role": "user", "content": prompt}],
    functions=[...]  # 如支持函数调用，可在此传入工具描述列表
)
content = response.choices[0].message['content']
```

对于 Gemini/DeepSeek，如果无官方SDK，可使用 `requests` 库直接对其 REST API 接口 POST JSON 数据。模块应将函数封装为简单接口，如 `reply = llm_api.query(prompt, context)`，返回模型文本或函数调用信息，供主流程处理。

2. RAG 检索模块（本地向量数据库）

功能边界：负责加载本地知识库（JSON 格式聊天记录），建立向量索引，并根据查询语句检索相关内容供LLM参考。此模块不直接参与模型对话，只在需要时提供检索结果。

实现逻辑：采用小型嵌入向量数据库（如 **ChromaDB** 或 FAISS）存储语料向量。流程包括：初始化向量库客户端，预先将 JSON 聊天记录按段落或 QA 对切分，使用预训练嵌入模型将文本转为向量并插入数据库；在对话过程中收到查询请求时，对用户提问生成查询向量，在向量库中进行最近邻搜索，返回若干相似的聊天记录片段。然后主流程可将这些片段作为上下文提供给 LLM，提高回答准确性。

使用工具包：推荐 **ChromaDB** 作为向量库，它支持内嵌轻量存储且易于使用 [2]。ChromaDB 可直接在内存中运行（使用 `chromadb.Client()` 创建非持久实例，无需外部服务）[2]。同时，其内置了 SentenceTransformer 等嵌入模型接口，可快速对文本生成向量插入库中 [3]。例如：

```
import chromadb
from chromadb.utils import embedding_functions
```

```
client = chromadb.Client()  # 内存数据库 [2]
embedding_func = embedding_functions.SentenceTransformerEmbeddingFunction(model_name="all-MiniLM-L6-v2")
collection = client.create_collection(name="chat_logs", embedding_function=embedding_func)
# 创建集合并指定嵌入模型 [3]

# 添加文档到向量库
collection.add(documents=[text1, text2, ...], ids=[id1, id2, ...])

# 执行相似检索
results = collection.query(query_texts=[user_query], n_results=3)
retrieved_texts = results.get("documents", [])
```

以上代码演示了使用 Chroma 将文本嵌入存储并查询的基本过程：实例化客户端、创建集合时指定预训练模型生成文本嵌入，然后批量 `add` 文档，最后用 `query` 语义搜索得到相关文本 [3] [4]。若不使用 Chroma，也可选择 **FAISS**：例如借助 `faiss.IndexFlatL2` 建立索引，将 NumPy 向量添加并通过 `index.search()` 检索。但 FAISS 需要手动管理向量与元数据映射，相对复杂。综上，RAG 模块应尽量简化：利用现成库的一行调用完成向量检索，将得到的文本列表返回供 LLM 拼接上下文。

3. 工具注册与调用模块

功能边界：统一管理**工具列表**及其调用逻辑。它维护每个工具的标识、功能描述以及实际执行函数的对应关系。主流程通过该模块查询有哪些工具可用，并在需要时调用指定工具获取结果。工具注册模块不直接包含具体工具实现，只负责管理和调度。

实现逻辑：采用脚本扫描或手动注册的方式收集所有工具脚本。例如，将所有 `tool_*.py` 文件（或特定工具目录下的模块）导入，读取其中预定义的工具名、描述和执行函数，并注册到一个字典或列表中。可以规定每个工具脚本对外提供统一的接口，例如定义一个 `TOOL_INFO` 字典包含工具名称、描述、参数说明和一个 `run(**kwargs)` 函数用于执行。当 LLM 接口模块返回需要调用某工具（函数调用信息或解析出的指令）时，工具注册模块据此匹配找到对应函数并执行，并将执行结果返回给主代理。

若使用 OpenAI 的函数调用特性，可在初始化对话时将所有工具的函数签名（名称、参数JSON模式等）通过 API 提供给模型，使 LLM 能够直接输出 `function_call` 内容指示调用何种工具 [1]。例如：

```
functions = [{
    "name": "get_weather",
    "description": "获取指定城市的天气",
    "parameters": {
        "type": "object",
        "properties": {
            "city": {"type": "string", "description": "城市名称"}
        },
        "required": ["city"]
    }
}, ...]  # 列出所有工具的规范
response = openai.ChatCompletion.create(..., functions=functions)
if response.choices[0].finish_reason == "function_call":
    func_name = response.choices[0].message["function_call"]["name"]
```

```
    args = json.loads(response.choices[0].message["function_call"]["arguments"])
    result = TOOL_REGISTRY[func_name](**args)
```

如上，模型可能返回一个 `function_call` 请求（例如 `get_weather`），代理代码用注册模块找到对应函数并执行。对于不支持直接函数调用的模型，可采用解析约定，例如让模型输出 `[TOOL: 工具名 参数]` 格式字符串，注册模块用正则/字符串解析提取出工具名和参数再执行。

使用工具包：可以考虑使用 **LangChain** 这类框架快速集成工具，但为保持MVP简洁也可自行实现。LangChain 提供标准化的工具接口以及代理Agent，可减少样板代码 [5]；例如 LangChain 内置了 `SerpAPIWrapper` 等工具类，一行代码即可添加搜索功能 [5]。在本方案中，为演示最小原型，推荐直接以 Python 原生方式注册调用工具，尽量避免引入过多抽象层。注册模块实现可简单如：

```
# 简易工具注册示例
TOOL_REGISTRY = {}
for tool_module in discover_tool_modules('tools'):  # 遍历工具脚本
    tool_info = tool_module.TOOL_INFO  # 假定每个工具脚本定义此变量
    TOOL_REGISTRY[tool_info['name']] = tool_module.run
```

以上逻辑通过动态导入收集工具，并使用字典将工具名映射到执行函数，供后续调用。这样增加新工具脚本时无需修改主流程代码，实现**热插拔**和易测试。

4. 工具功能模块（独立脚本）

功能边界：每个工具脚本专注实现单一功能，不与其他模块直接交互。输入输出通过函数参数和返回值定义，避免全局状态。各工具只实现基础能力（MVP级别），确保脚本短小。

实现逻辑：按照需求实现以下基本工具，每个用一个 `.py` 脚本表示：

- **天气查询工具**（如 `tool_weather.py`）：调用天气API获取指定城市当前天气。可使用公开天气API（如 OpenWeatherMap）并用 `requests` 请求获取 JSON，解析后返回简短结果。若无 API，则可模拟返回固定的示例天气数据。
- **日期时间工具**（`tool_datetime.py`）：返回当前日期或时间。直接使用Python标准库 `datetime` 获取系统当前日期时间，并格式化输出。支持根据参数返回不同格式（如日期或时间）。
- **计算器工具**（`tool_calculator.py`）：对简单算术表达式求值。可采用内置 `eval()`（注意安全性，需限制可用运算符）或使用 `ast` 模块解析算术表达式计算结果。
- **日程查询工具**（`tool_calendar.py`）：查询本地日程安排。如有本地日历文件（例如ICS格式）或JSON日程数据，可读取并返回指定日期的事件。MVP 实现可简化为读取本地预先存在的日程JSON，按关键词或日期过滤返回结果。
- **邮件读取工具**（`tool_email.py`）：通过 IMAP 获取收件箱邮件。使用Python内置 `imaplib` 登录邮箱服务器（例如 Gmail IMAP），检索未读邮件列表，返回邮件标题或简要内容。只需实现只读操作（避免复杂的发信操作），确保提供邮箱账号和密码配置。
- **网页搜索工具**（`tool_websearch.py`）：调用第三方搜索API返回网页摘要。建议使用 **SerpAPI** 提供的搜索接口，以规避直接爬取的复杂性。SerpAPI 有官方 Python 包 google-search-results 可用，其用法简单明了：[6] 例如:

```
from serpapi import GoogleSearch
search = GoogleSearch({"q": "查询关键词", "api_key": SERPAPI_KEY})
results = search.get_dict()  # 获取结果JSON [6]
```

上述代码利用 SerpAPI 接管实际搜索，并返回结构化的 JSON 结果，可提取`results["organic_results"]`中的前几条摘要作为答复内容。若无 SerpAPI，也可考虑简单使用 `requests` 请求一个公开的搜索 API 或RSS源。

- **文件读取工具**（`tool_file_reader.py`）：读取本地文件内容并返回摘要。支持文本文件直接打开读取全文，对于 PDF、Word 等可借助 **unstructured** 等解析库。unstructured 提供统一的 `partition` 接口来解析多种文档格式 [7]。例如:

```
from unstructured.partition.auto import partition
elements = partition(filename="example.pdf")
text = "\n".join(str(elem) for elem in elements)
```

如上，调用 `partition` 会自动识别文件类型并提取出文字元素列表，然后拼接为纯文本 [7]。MVP阶段可仅支持TXT/Markdown等简单文本文件，复杂格式在有需要时再扩展。

每个工具脚本对外只需提供一个 `run()` 函数及必要的元数据，不应在模块加载时就执行任何操作，以免影响其它部分的加载和测试。通过保持工具实现的**高内聚、低耦合**，可以随时独立运行单个脚本进行调试或改进，而不用影响整个Agent系统。

5. UI 界面模块（Gradio/Web界面）

功能边界：提供用户与 Agent 交互的界面，接收用户输入的问题，将其传递给 Agent 主流程获得回复，并将回复显示给用户。默认使用 Gradio 搭建网页界面，模块本身不包含业务逻辑。可选支持 Telegram Bot 接口以便在移动端使用。

实现逻辑：利用 Gradio 的简易接口构建对话界面。可使用 `gr.ChatInterface` 或 `gradio.Interface` 实现一个聊天对话框UI。MVP实现可以非常简单，例如：

```
import gradio as gr

def agent_respond(user_input, chat_history):
    response = agent_core.process(user_input)
    # agent_core.process 内部封装调用 LLM、RAG、Tools 的主流程
    return response

with gr.Blocks() as demo:
    chatbot = gr.Chatbot()
    msg = gr.Textbox()
    clear = gr.Button("Clear")
    msg.submit(agent_respond, [msg, chatbot], [chatbot])
    clear.click(lambda: None, None, chatbot, queue=False)
demo.launch()
```

上述伪代码通过 `Textbox.submit` 将用户每次发送的消息交给 `agent_respond` 处理，并将 Agent 回复追加到 Chatbot 对话框中。`agent_core.process` 即主流程接口，其内部会调用 LLM 接口模块获取答案或工具结果。Gradio 封装了大部分前端工作，只需关注 `agent_respond` 函数如何调用 Agent 核心并返回字符串即可。

若扩展 Telegram Bot，则可使用 **python-telegram-bot** 库：编写一个简单的 bot.py 脚本，监听消息事件，将消息转发给同一 `agent_core.process` 获得结果，再调用 Telegram API 将结果发送回用户聊天。这部分可选功能可独立于 Gradio UI 模块实现，不影响整体架构。

6. Agent 主流程模块（对话管理与调度）

功能边界：作为智能体的大脑，负责协调调用 LLM、RAG、工具各模块完成一次完整的用户问答。它实现具体的对话策略（例如 ReAct 或工具优先等）：当用户输入的问题需要外部数据或计算时，决定调用相应工具；当需要知识查询时，通过RAG模块获取上下文；默认情况下直接向LLM请求回答。该模块输出最终给用户的回复文本。

实现逻辑：主流程通常以**循环**的形式等待输入->处理->输出，每次用户提问进入处理流程：

1. **意图判断**（可选，MVP可省略复杂意图分类）：简单判断用户问句中是否包含如“天气”“计算”“日历”“邮件”“搜索”“文件”等关键词，或通过LLM模型先询问它需不需要工具协助。
2. **检索增强**：如有启用RAG且问句涉及聊天记录相关的问题，调用 RAG 模块检索 JSON 聊天记录中相关内容，获取一个简短内容列表（可为空）。将这些内容作为额外上下文附加在提示(prompt)中，例如：“相关记录：... 用户问：...”。
3. **工具调用决策**：如果模型支持函数调用，则直接将工具函数列表传给 LLM 接口，让其决定。如果不支持，则在提示中明示可用工具格式。如用户问“明天上海天气如何？”，主流程可匹配到天气意图，先调用天气工具获取结果，再将结果插入回答；或者让模型先输出一个 `[TOOL:weather {"city": "上海", "date": "明天"}]` 的标记，再由代理实际调用工具获取数据。
4. **LLM回答生成**：准备好上下文和工具结果后，调用 LLM 接口模块生成最后的回答。如果前一步有工具结果，则将结果作为额外信息写入提示，或者在函数调用模式下把工具返回值传回模型让其继续生成答案。
5. **回复输出**：将LLM生成的答案返回给 UI 界面模块，显示给用户。主流程同时可以记录此次对话到内部日志（用于多轮对话上下文或进一步训练数据）。

上述流程力求简单明了，在MVP阶段可采用顺序执行：**尽量避免复杂的任务规划**（如不引入多步推理，除非必要）。如果决定使用 ReAct 等Agent策略，可以让模型在一次对话中多轮思考工具用法，但MVP更建议一问一答、一工具的形式。

使用工具包：如果不使用LangChain自带Agent，可完全手工实现上述逻辑。但如果考虑日后扩展，多工具调度和上下文管理，LangChain提供的Agent框架可以简化这一过程，其原理与上述类似，即根据LLM输出决定 Action 或 Final Answer。LangChain 已内置对**工具调用**、**记忆管理**、**上下文缓存**等的支持 [5]。不过，出于最小原型的考量，直接基于提示工程和if/else逻辑实现也未尝不可。关键是确保模块交互清晰：主流程模块要易于调用 LLM模块函数、RAG检索函数和工具注册模块，无循环依赖。

开源工具包选型及最简用法

为降低开发难度和实现成本，可利用以下开源库，它们经过验证且使用简单：

- **OpenAI API SDK**：适用于调用 OpenAI 提供的模型接口。如果 Gemini/DeepSeek 与 OpenAI API 接口格式兼容，可直接使用其 Python SDK 调用模型，自动处理请求和响应格式。用法是安装 `openai` 库后，一行代码调用模型并获取结果，如前述示例所示。对于函数调用场景，OpenAI SDK 会直接解析返回 JSON，简化了手工正则的麻烦 [1]。若 Gemini/DeepSeek 有自有 SDK，则优先使用官方库确保兼容性。

- **LangChain**：强大的对话代理框架。虽然后端依赖较多，但其封装的**工具接口**非常实用 [5]。例如，通过 LangChain 可以快速注册 SerpAPI、Calculator 等工具，无需关心底层实现。此外还提供 Memory

（对话记忆）和 Chain（链式调用）组件，便于将检索和模型调用串联。不过MVP中应谨慎使用，以防范不必要的复杂度。建议在基本功能跑通后，再考虑逐步融入 LangChain 做增强。

- **Chromadb**：开源向量数据库，号称开箱即用且**本地运行** [2]。Chroma 提供简单的 Python 接口，无需手动安装后端服务。通过 `chromadb.Client()` 即可创建数据库实例（可选持久化到磁盘） [2]，使用内置的 SentenceTransformer 模型将文本转为嵌入向量存储和查询 [3]。对比 FAISS，Chroma 封装更完善，特别适合快速构建检索功能。安装 `pip install chromadb` 即可使用。

- **unstructured**：文件解析库，支持多种文档格式的**自动解析** [7]。使用它可以大大简化文件读取工具的实现，不用分别针对 PDF、DOCX 等编写解析代码。只需调用统一的 `partition` 接口并传入文件路径，即可得到文档的分段文本列表 [7]。但需注意，该库对环境有一些依赖（如需安装 Poppler、Tesseract 等用于PDF和OCR），MVP阶段可根据需要精简安装，例如只处理文本文件则安装最小依赖。

- **SerpAPI Python 客户端**：用于实现网页搜索工具的官方库。通过 `pip install google-search-results` 安装后，可以使用 `GoogleSearch` 类直接调用各大搜索引擎，无需处理 HTML 页面解析 [6]。只需提供查询词和 API Key，调用 `get_dict()` 就能得到结构化结果 JSON [6]。这极大降低了获取搜索结果的门槛，非常适合作为Agent的“网络搜索”能力来源。

- **imaplib/IMAPClient**：用于邮件收取的Python标准库/轻量封装库。imaplib 是 Python 内置库，可直接使用但稍显底层；IMAPClient 是基于 imaplib 的更友好封装（需另行安装）。考虑到MVP追求轻量，使用 imaplib 无需安装依赖。通过 `imaplib.IMAP4_SSL` 连接邮件服务器后，使用 `.login(user,pass)` 登录，再用 `.select('INBOX')` 和 `.search()/fetch()` 检索邮件即可。建议只获取邮件的发件人、主题等基本信息，避免解析复杂邮件内容。

- **Gradio**：快速搭建网络界面的利器。对于本项目，它的**低代码**特性可以让我们在几行内定义交互界面，无需处理前端细节。Gradio 的 Interface/Blocks API 都很直观，适合 MVP 阶段使用。安装 `pip install gradio` 后，可在本地启动一个 Web 服务供测试。注意部署时要考虑密钥安全（例如不要在界面输出敏感的 API keys）。

- **python-telegram-bot** (可选)：如果需要 Telegram 集成，可使用此库简化 Bot 开发。通过注册 Bot Token 后，使用其 API 设置消息处理函数，将用户消息对接 Agent 主流程返回结果即可。该库对 asyncio 也有支持，可以流式地与 LLM交互。不过在 MVP 中，Telegram 接口不是必需部分，只作为未来拓展选项。

以上工具包各司其职，能极大加速开发。同时保持每个模块**独立运行**：例如，可以单独运行 RAG 模块构建索引，或单独调试某个工具模块函数，这都归功于脚本化的模块设计。整个 MVP 系统不需要引入重型 Web 框架或数据库，充分利用轻量级库即可实现目标功能。

模块关系与目录结构

综合以上设计，项目源码可按功能组织为扁平结构，每个模块一个文件（或文件夹）。推荐的代码仓库目录结构如下：

```
project_root/
├── agent_core.py           # Agent主流程：协调LLM调用、RAG检索、工具调度
├── llm_interface.py        # LLM接口模块：封装对 Gemini/DeepSeek API 的调用
├── rag_vector_store.py     # RAG检索模块：构建向量数据库并提供查询接口
├── tool_registry.py        # 工具注册模块：动态加载工具脚本，管理工具列表
```

```
├── tools/                  # 工具脚本目录（每个工具一个独立文件）
│   ├── tool_weather.py      # 天气查询工具
│   ├── tool_datetime.py     # 日期时间工具
│   ├── tool_calculator.py   # 计算器工具
│   ├── tool_calendar.py     # 日程查询工具
│   ├── tool_email.py        # 邮件读取工具
│   ├── tool_websearch.py    # 网页搜索工具
│   └── tool_file_reader.py  # 文件读取工具
├── ui_gradio.py            # UI界面模块：Gradio前端交互脚本
└── bot_telegram.py         # （可选）Telegram Bot 接口脚本
```

各脚本可独立运行和测试。例如运行 `ui_gradio.py` 即启动本地Web界面，`bot_telegram.py` 可单独作为bot服务运行。工具脚本放置在 `tools/` 文件夹是为了便于 `tool_registry.py` 扫描加载（也可以不建子目录，直接以命名约定筛选）。整个仓库无复杂的多层包结构，以脚本方式组织确保了**MVP的简洁性**。

在此结构下，模块间的关系依然清晰：`agent_core.py` 导入 `llm_interface.py`、`rag_vector_store.py` 和 `tool_registry.py`，在对话流程中调用它们提供的接口；`tool_registry.py` 再去加载 `tools/` 下各脚本，将它们的函数注册为可调用工具；UI 脚本（Gradio/Telegram）则调用 `agent_core.py` 对外提供服务。通过这种模块化设计，实现了 AI 助理 Agent 的基本功能，并为今后功能拓展（替换更强模型、增加新工具、改进对话策略）打下良好基础。 [5] [2]

---

[1] Practical Examples of OpenAI Function Calling | by Cobus Greyling
https://cobusgreyling.medium.com/practical-examples-of-openai-function-calling-a6419dc38775

[2] [3] [4] Embeddings and Vector Databases With ChromaDB – Real Python
https://realpython.com/chromadb-vector-database/

[5] 〖大模型LLM〗 Agent 架构图解_llm agent架构-CSDN博客
https://blog.csdn.net/shuizhudan223/article/details/147403401

[6] SerpApi: Python Integration
https://serpapi.com/integrations/python

[7] GitHub - Unstructured-IO/unstructured: Convert documents to structured data effortlessly. Unstructured is open-source ETL solution for transforming complex documents into clean, structured formats for language models. Visit our website to learn more about our enterprise grade Platform product for production grade workflows, partitioning, enrichments, chunking and embedding.
https://github.com/Unstructured-IO/unstructured