International Journal of Core Engineering & Management
Volume-7, Issue-11, 2024 ISSN No: 2348-9510

BUILDING A CUSTOMIZED LLM AGENT AND LESSONS LEARNED

Ashish Bansal
ashishbansalmohan@gmail.com

Abstract

In this paper, we will delve into the world of advanced LLM agents, exploring how a customized agent enables developers to build sophisticated, tailored tools that leverage the capabilities of these cutting-edge models. We would create a custom agent that will help an AI Developer or just a user the understanding of how a Agent is deployed to perform a specific task as well as will also highlights the lessons learned while working on these agents. The rise of Large Language Models (LLMs) has revolutionized the way we interact with technology, opening up new frontiers in natural language processing and artificial intelligence. As these models become more advanced, the need to harness their full potential through customization and integration with various tools becomes paramount. This rise introduced a need of system that can be specific is performing specific actions and are specialized in executing certain task, agent's frameworks were designed to streamline the creation and deployment of custom LLM agents.

In this paper, we will delve into the world of advanced LLM agents, exploring how a customized agent enables developers to build sophisticated, tailored tools that leverage the capabilities of these cutting-edge models. We would create a custom agent that will help an AI Developer or just a user the understanding of how a Agent is deployed to perform a specific task as well as will also highlights the lessons learned while working on these agents.

Keywords: Generative AI, LLM, AI Agents, Agents Framework, Custom Agents

I. INTRODUCTION

Agents are one of the most powerful and fascinating approaches to using Large Language Models (LLMs). The explosion of interest in LLMs has made agents incredibly prevalent in AI-powered use cases. Using agents allows us to give LLMs access to tools. These tools present an infinite number of possibilities. With tools, LLMs can search the web, do math, run code, and more.
Think of agents as enabling "tools" for LLMs. Like how a human would use a calculator for maths or perform a Google search for information — agents allow an LLM to do the same thing. Using agents, an LLM can write and execute Python code. It can search for information and even query a SQL database.

In the recent months, we've have all heard about Agents and Multi-Agent frameworks. These AI agents have become the unsung heroes of automation and decision-making. While pre-built frameworks like AutoGen and CrewAI offer tempting shortcuts, there's an unparalleled thrill and depth of understanding that comes from building your own agent from the ground up.
With that in mind, tools are relatively simple. Fortunately, we can build tools for our agents in no time. By the end of this paper, you'll have a foundational understanding of how AI agents tick, and you'll be well on your way to creating a digital companion that can generate and execute code on demand.

International Journal of Core Engineering & Management
Volume-7, Issue-11, 2024 ISSN No: 2348-9510

II. AGENTS AND TOOLS

To use agents, we require three things:

- A base LLM,
- A tool that we will be interacting with, At their core, tools are objects that consume some input, typically in the format of a string (text), and output some helpful information as a string. In reality, they are little more than a simple function that we'd find in any code. The only difference is that tools take input from an LLM and feed their output to an LLM.
- An agent to control the interaction.

It's like teaching a robot to fish, except instead of fish, it's pulling Python scripts out of the ether! Let's take an example, where we will build an agent that can help find the architecture documentation of software architects and will try to create a architecture diagram and summary
Video of explaining the architecture.

1. The Building Blocks of our Agent

Before we dive into the code, let's outline the key components we'll be constructing:

- **Initialization:** Setting up our agent's main LLM calls that will power the whole agent.
- **Document Parser:** Parser to parse the provided documentation.
- **Architecture Generation**: Teaching our agent to create architecture diagrams.
- **Library Management**: Enabling our agent to install necessary tools or python libraries.
- **Code Execution**: Empowering our agent to run the code it generates.
- **Control plane:** Control Plane to manage all these functions.

Now, let's break down each of these steps and see how they come together to form our AI assistant.

- **Step 1: Initialization** — Agents Powerhouse In the world of AI agents, initialization is a process to provide the powerhouse to the agents. This is where we set up the basic structure of our agent and connect it to its primary source of intelligence — in this case, we will choose Claude API.

- **Step 2: Document Parser**: Parser to parse the provided documentation Now that our agent has a "body," let's give it the ability to think — or in this case, to generate code. This is where things start to get exciting!
  This method will accept the architecture documentation and provides the logical separated explanation. These logical explanations are then fed to a next method in the agent.

- **Step 3: Architecture Generation** — teaching our agent to create architecture diagrams. This method is the crown jewel of our agent's capabilities. It's using the Claude API to generate archi- tecture based on a given prompt.
  Think of it as giving our agent the ability to brainstorm and write steps outlined in documentation. We're also doing some cleanup to ensure we get clean, executable Python code without any markdown formatting or unnecessary comments.
  The parameters we're using (like temperature and top_p) allow us to control the creativity and randomness of the generated explanation. Using the explanation steps it has the code to create an architecture Diagram.

- **Step 4: Library Management** — enabling our agent to install necessary tools or python libraries every good coder knows the importance of having the right libraries at their disposal. Our AI agent is no different. This next method allows Agent to identify and install any necessary Python libraries. This method is like sending our agent on a shopping spree in the Python Package Index. It scans the generated explanation code for any pip install comments, checks if the libraries are already installed, and if not, installs them. It's ensuring our agent always has the right tools for the job, no matter what task we throw at it.

- **Step 5: Code Execution — Bringing the Code to Life**
  Generating code is great, but executing it is where the rubber meets the road. This next method allows our agent to run the code it has generated. The generated code is used to covert that code to the architecture diagram.
  This method is where the magic really happens. It takes the generated code, writes it to a temporary file, executes it, captures the output (or any errors), and then cleans up after itself. It's like giving our agent hands to type out the code and run it, all in the blink of an eye producing the right architecture diagram of that architecture documentation.

- **Step 6: Control plane:** Control Plane to manage all these functions. Finally, we need a way to orchestrate all these amazing capabilities. Enter the run method: This is the Control plane of our AI assistant. It takes a prompt, generates the code, executes it, and reports back with the results or any errors. It's like having a personal assistant who not only understands your requests but carries them out and gives you a full report.

These put together gives us an agent that would convert the big documentation into small video of the architecture diagram. There are many lessons we have learned while developing this agent and would cover all of those in the next section.

2. Lessons Learned

Building AI agents over the past year has been a roller coaster, and we're undoubtedly still early in this new wave of tech. Here's a little overview of what I've learned so far.
In above section we have seen how can we create an agent and what are the main components of an agent. Below are few things I've learned while working on agents.

III. REASONING IS MORE IMPORTANT THAN KNOWLEDGE

Focus less on what your agent "knows", and more on its ability to "think".
For example, let's consider writing SQL Queries. SQL queries fail . . . a lot. In my time as a data scientist, I'm sure I had a lot more queries fail than I ever did succeed. If a complicated SQL query

On real data that you've never used before works the first time you run it, your reaction should be, "Oh crap, something's probably wrong" rather than "wow, I nailed it". Even on a text-to-SQL benchmark evaluating how well models translate a simple question into a query, it caps out at 80% accuracy.

So if you know that your model's capacity for writing accurate SQL translates to a B-minus at best, how can you optimize for reasoning instead? Focus on giving the agent context and letting it "think", instead of hoping it gets the answer right in one try. We make sure to return any SQL

errors, along with all the context we can capture, back to the agent when its query fails. . . which enables the agent to resolve the issue and get the code working a vast majority of the time. We also give our agent a number of tool calls to retrieve context on the data in the database, similar to how a human might study the information schema and profile the data for distributions and missing values before writing a new query.

IV. THE BEST WAY TO IMPROVE PERFORMANCE IS BY ITERATING ON THE AGENT-COMPUTER INTERFACE (ACI)

The term ACI is new (introduced in this research out of Princeton), but the focus on perfecting it has been part of our day-to-day for the last year. The ACI refers to the exact syntax and structure of the agent's tool calls, including both the inputs that the agent generates and the outputs that our API sends back in response. These are the agent's only way to interface with the data it needs to make progress aligned with its directions.

Because the underlying models (gpt-4o, Claude Opus, etc.) each exhibit different behaviour, the ACI that works best for one won't necessarily be right for another. This means a great ACI requires as much art as science. it's more like designing a great user experience than it is writing source code because it constantly evolves and small tweaks cascade like a fender bender turning in to a 30-car pile up. I can't overstate how important the ACI is. we've iterated on ours hundreds of times and seen huge fluctuations in our agent's performance with seemingly small tweaks to the names, quantity, level of abstraction, input formats, and output responses of our tools.

Here's a small, specific example to illustrate how critical and finicky your ACI can be: When testing our agent on gpt-4-turbo shortly after it was released, we noticed an issue where it would completely ignore the existence of specific columns that we were trying to tell it about in a tool call response. We were using a markdown format for this information that was taken directly from the OpenAI docs at the time, and had worked well with gpt-4–32k on the same data. We tried a few adjustments to our markdown structure to help the agent recognize the columns that it was pretending did not exist, even though they were in the response to one of the tool calls it was making. None of the tweaks worked, so we had to start experimenting with entirely different formats for the information... and after an overhaul to start using JSON instead of markdown (only for OpenAI models), everything was working great again. Ironically, the JSON structured response required significantly more tokens because of all the syntax characters, but we found it was necessary and actually critical to help the agent understand the response.

Iterating on your ACI may feel trivial but it's actually one of the best ways to improve the user experience of your agent.

V. AGENTS ARE LIMITED BY THEIR MODEL(S)

The underlying model(s) you use are the brain to your agent's body. If the model sucks at making decisions, then all the good looks in the world aren't going to make your users happy. We saw this limitation first hand when testing our agent simultaneously on gpt-3.5-turbo and gpt-4–32k. On 3.5, we had a number of test cases that went something like this:

1. user provided an objective, for example: "analyze the correlation between starbucks loca-tions and home prices by zip code to understand if they are related"
2. agent would assume that a table existed in the database with a name it hallucinated, like "HOME_PRICES", and columns like "ZIP_CODE" and "PRICE" instead of running a search to find the actual table
3. agent would write a SQL Query to calculate average price by zip code that failed, and get an error message indicating that the table did not exist
4. agent would remember "oh yeah, I can search for actual tables..." and would run a search for "home prices by zip code" to find a real table it could use
5. agent would re-write its query with the correct columns from a real table, and it would work
6. agent would continue on to the Starbucks location data and make the same mistake again

Running the agent on gpt-4 with the same directions was completely different. Instead of leaping into the wrong action immediately and wasting time getting it wrong, the agent would make a plan with the correct sequencing of tool calls, and then follow the plan. As you can imagine, on more complex tasks the gap in performance between the two models grew even larger. As great as the speed of 3.5 was, our users vastly preferred the stronger decision making and analysis capability of gpt-4.

One thing we learned from these tests is to pay very close attention to your agent hallucinates or fails, when it happens. AI agents are lazy (I assume human laziness is well represented in the training data for the underlying models) and will not make tool calls that they don't think they need to. Similarly, when they do make a tool call, if they don't understand the argument instructions well they will often take shortcuts or completely ignore required parameters. There is a lot of signal in these failure modes! The agent is telling you what it wants the ACI to be, and if the situation allows, the easiest way to solve this is to give in and change the ACI to work that way. Of course, there are going to be plenty of times where you have to fight against the agent's instincts via changes to the system prompt or your tool call instructions, but for those times when you can more simply just change the ACI, you'll make your life a lot easier.

VI. FINE-TUNING MODELS TO IMPROVE AGENT PERFORMANCE IS A WASTE OF TIME

Fine-tuning a model is a method to improve the model's performance on a specific application by showing it examples that it can learn from. Current fine-tuning methods are useful for teaching a model how to do a specific task in a specific way, but are not helpful for improving the reasoning ability of the model. In my experience, using a fine-tuned model to power an agent actually results in worse reasoning ability because the agent tends to "cheat" its directions — meaning it will assume that the examples it was fine-tuned on always represent the right approach and sequence of tool calls, instead of reasoning about the problem independently.

Fine-tuning can still be a very useful tool in your Swiss Army pocket knife. For instance, one approach that has worked well is using a fine-tuned model to handle specific tool calls that the agent makes. Imagine you have a model fine-tuned to write SQL queries on your specific data, in your database... your agent (running on a strong reasoning model, without fine-tuning) can use a tool call to indicate that it wants to execute a SQL query, and you can pass that into a standalone task handled by your model that's fine-tuned on SQL queries for your data.

International Journal of Core Engineering & Management
Volume-7, Issue-11, 2024 ISSN No: 2348-9510

VII. IF YOU'RE BUILDING A PRODUCT, AVOID USING ABSTRACTIONS LIKE LANGCHAIN AND LLAMAINDEX

You should fully own each call to a model, including what's going in and out. If you offload this to a 3rd party library, you're going to regret it when it comes time to do any of these with your agent: on-board users, debug an issue, scale to more users, log what the agent is doing, upgrade to a new version, or understand why the agent did something.

If you're in pure prototype mode and just trying to validate that it's possible for an agent to accomplish a task, by all means, pick your favourite abstraction and do it to live.

VIII. YOUR AGENT IS NOT YOUR MOAT

Automating or augmenting human knowledge work with AI agents is a massive opportunity, but building a great agent is not enough. Productionizing an agent for users requires a significant investment in a bunch of non-AI components that allow your agent to actually work . . . this is where you can create competitive differentiation:

- **Security:** AI agents should only run with the access and control of the user directing them. In practice, this means jumping through a hopscotch of OAuth integrations, Single Sign-On providers, cached refresh tokens, and more. Doing this well is absolutely a feature.

- **Data connectors**: AI agents often need live data from source systems to work. This means integrating with APIs and other connection protocols, frequently for both internal and 3rd party systems. These integrations need initial build out and TLC over time.

- **User interface**: Users will not trust an AI agent unless they can follow along and audit its work (typically the first few times a user interacts with an agent, decreasing sharply over time). It's best if each tool call that the agent makes has a dedicated, interactive interface so the user can follow along with the agent's work and even interact with it to help build confidence in its reasoning behaviour (i.e., browse the contents of each element returned in a semantic search result).

- **Long-term memory**: AI agents by default will only remember the current workflow, up to a maximum amount of tokens. Long-term memory across workflows (and sometimes, across users) requires committing information to memory and retrieving it via tool calls or injecting memories into prompts. I've found that agents are not very good at deciding what to commit to memory, and rely on human confirmation that the info should be saved. Depending on your use case, you may be able to get away with letting the agent decide when to save something to memory, a la ChatGPT.

- **Evaluation:** Building a framework to evaluate your AI agent is a frustratingly manual task that is never fully complete. Agents are intentionally nondeterministic, meaning that based on the direction provided, they will look to come up with the best sequence of tool calls available to accomplish their task, reasoning after each step like a baby learning to walk. Evaluating these sequences takes two forms: the overall success of the agent's workflow in accomplishing the task, and the independent accuracy of each tool call (i.e. information re- trieval for search; accuracy for code execution; etc. . .). The best, and only, way I've found to quantify

performance on the overall workflow is to create a set of objective / completion pairs, where the objective is the initial direction provided to the agent, and the completion is the final tool call representing completion of the objective. Capturing the intermediate tool calls and thoughts of the agent is helpful for understanding a failure or just a change in the tool call sequence.

IX. DON'T BET AGAINST MODELS CONTINUING TO IMPROVE

While building your agent, you will constantly be tempted to over-adapt to the primary model you're building it on, and take away some of the reasoning expectations you have for your agent. Resist this temptation! Models will continue to improve, maybe not at the insane pace we are on right now, but definitely with a higher velocity than past technology waves. Customers will want agents that run on the models of their preferred AI provider. And, most importantly, users will expect to leverage the latest and greatest model within your agent. When gpt-4o was released, I had it running in a production account within 15 minutes of it being available in the OpenAI API. Being adaptable across model providers is a competitive advantage!

X. CONCLUSION

The paper highlights what is an agent and a custom agent can be created with less efforts. Even-though the sudden rise in LLM Application a dire need on agent orchestrations is needed that will provide better LLM powered application to realize AI powered systems. Paper also highlights the lessons learned in the way while creating agents for production use and gives us a path to investigate and improve while having them in consideration continuously improving the agent framework optimisations.

REFERENCES

1. Y. Shen, K. Song, et al., HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hug-gingFace (2023)
2. Langchain.io (2019), Wayback Machine
3. Jun-hang Lee, Mother of Language Slides (2018), SlideShare
4. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering arXiv:2405.15793
5. https://www.crewai.com/
6. https://microsoft.github.io/autogen/