

# The Problem With LangChain

July 14, 2023 · 16 min



*The GitHub Repository of R'lyeh, Stable Diffusion 1.5 + ControlNet 1.1*

If you've been following the explosion of AI hype in the past few months, you've probably heard of [LangChain](#). LangChain, developed by Harrison Chase, is a Python and JavaScript library for interfacing with [OpenAI's](#) GPT APIs (later expanding to more models) for AI text generation. More specifically, it's an implementation of the paper [ReAct: Synergizing Reasoning and Acting in Language Models](#) published October 2022, colloquially known as the ReAct paper, which demonstrates a prompting technique to allow the model to "reason" (with a chain-of-thoughts) and "act" (by being able to use a tool from a predefined set of tools, such as being able to search the internet). This combination is shown to drastically improve output text quality and give large language models the ability to correctly solve problems.

## (1d) ReAct (Reason + Act)

**Thought 1:** I need to search Apple Remote and find the program it was originally designed to interact with.

**Act 1:** `Search[Apple Remote]`

**Obs 1:** The Apple Remote is a remote control introduced in October 2005 by Apple ... originally designed to control the Front Row media center program ...

**Thought 2:** Apple Remote was originally designed to control the Front Row media center program. I need to search Front Row next and find what other device can control it.

**Act 2:** `Search[Front Row]`

**Obs 2:** Could not find [Front Row]. Similar: ['Front Row Seat to Earth', 'Front Row Motorsports', 'Front Row (software)', ...]

**Thought 3:** Front Row is not found. I need to search Front Row (software) .

**Act 3:** `Search[Front Row (software)]`

**Obs 3:** Front Row is a discontinued media center software ...

**Thought 4:** Front Row (software) is controlled by an Apple Remote or the keyboard function keys. So the answer is keyboard function keys.

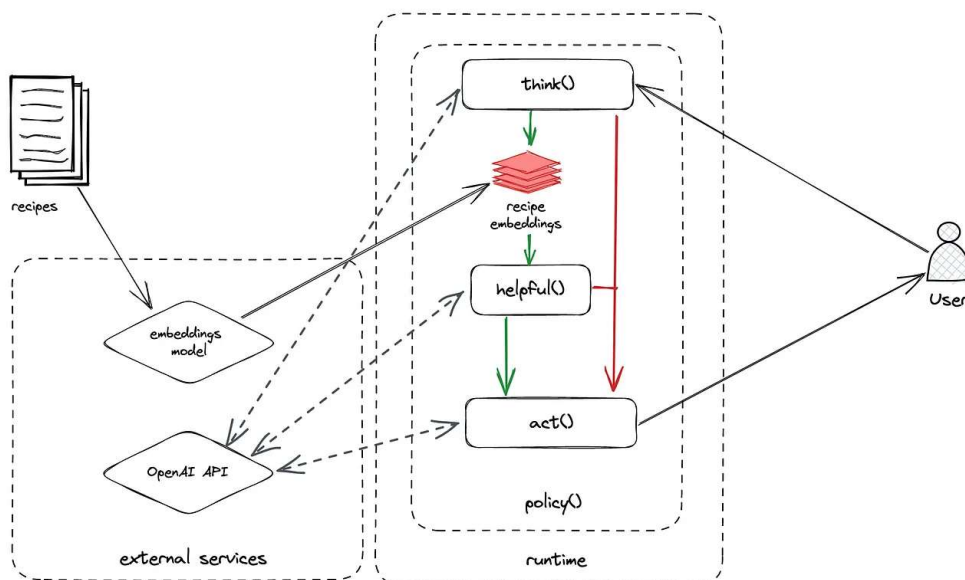
**Act 4:** `Finish[keyboard function keys]`



Example ReAct flow from the ReAct paper

The ReAct workflow popularized by LangChain was particularly effective with InstructGPT/text-davinci-003, although costly and not easy to use for small projects. In March 2023, as ChatGPT API usage became massively popular due to its extremely cheap API as I accurately predicted, LangChain use also exploded, to the point that LangChain was able to raise a \$10 million seed round and another \$20-\$25 million at a \$200 million valuation Series A despite not having any revenue nor any obvious plans how to generate revenue.

That's where my personal experience with LangChain begins. For my work at BuzzFeed, I was tasked with creating a ChatGPT-based chat bot for the Tasty brand (later released as Botatouille in the Tasty iOS app) that could chat with the user and provide relevant recipes. The source recipes are converted to embeddings and saved in a vector store: for example, if a user asked for "healthy food", the query is converted to an embedding, and an approximate nearest neighbor search is performed to find recipes similar to the embedded query and then fed to ChatGPT as added context that can then be displayed to the user. This approach is more commonly known as retrieval-augmented generation.



Example architecture for a Chatbot using retrieval-augmented generation. [via Joseph Haaga](#)

LangChain was by-far the popular tool of choice for RAG, so I figured it was the perfect time to learn it. I spent some time reading LangChain's rather comprehensive documentation to get a better understanding of how to best utilize it: after a *week* of research, I got nowhere. Running the LangChain demo examples did work, but any attempts at tweaking them to fit the recipe chatbot constraints broke them. After solving the bugs, the overall quality of the chat conversations was bad and uninteresting, and after intense debugging I found no solution. Eventually I had an existential crisis: am I a worthless machine learning engineer for not being able to figure LangChain out when very many other ML engineers can? We went back to a lower-level ReAct flow, which *immediately* outperformed my LangChain implementation in conversation quality and accuracy.

In all, I wasted a month learning and testing LangChain, with the big takeaway that popular AI apps may not necessarily be worth the hype. My existential crisis was resolved after coming across a [Hacker News thread](#) about someone [reimplementing LangChain in 100 lines of code](#), with most of the comments venting all their grievances with LangChain:

▲ loveparade 70 days ago | prev | next [-]

Am I the only one who is not convinced by the value proposition of langchain? 99% of it are interface definitions and implementations for external tools, most of which are super straightforward. I can write integrations for what my app needs in less than an hour myself, why bring in a heavily opinionated external framework? It kind of feels like the npm "left-pad" to me. Everyone just uses it because it seems popular, not because they need it.

▲ crazyedgar 69 days ago | parent | next [-]

For us LangChain actually caused more problems than it solved. We had a system in production which after working fine a few weeks suddenly started experiencing frequent failures (more than 30% of requests). On digging it seems that LangChain sets a default timeout of 60 seconds for every requests. And this behaviour isn't documented! Such spurious decisions made by LangChain are everywhere, and will all eventually come back to bite. In the end we replaced everything with vanilla request clients. Definitely not recommended to build a system on a library that provides very limited value while hiding a huge amount of details and decisions from you.

▲ Spivak 70 days ago | parent | prev | next [-]

Langchain is absolutely perfect though, it's bad enough that you'll be driven to write something better out of pure frustration but gives you enough good ideas and breadcrumbs to actually do it.

It's probably the best on-ramp for "practical uses of llms" because it scratches just the right developer itch.

The problem with LangChain is that it makes simple things relatively complex, and with that unnecessary complexity creates a tribalism which hurts the up-and-coming AI ecosystem as a whole. If you're a newbie who wants to just learn how to interface with ChatGPT, definitely don't start with LangChain.

## “Hello World” in LangChain (or More Accurately, “Hell World”)

The [Quickstart](#) for LangChain begins with a mini-tutorial on how to simply interact with LLMs/ChatGPT from Python. For example, to create a bot that can translate from English to French:

```
from langchain.chat_models import ChatOpenAI
from langchain.schema import (
    AIMessage,
    HumanMessage,
    SystemMessage
)
```

```

chat = ChatOpenAI(temperature=0)
chat.predict_messages([HumanMessage(content="Translate this sentence from English to French. I love programming.")])
# AIMessage(content="J'adore la programmation.", additional_kwargs={}, example=False)

```

The equivalent code using OpenAI's official Python library for ChatGPT:

```

import openai

messages = [{"role": "user", "content": "Translate this sentence from English to French. I love programming."}]

response = openai.ChatCompletion.create(model="gpt-3.5-turbo", messages=messages, temperature=0)
response["choices"][0]["message"]["content"]
# "J'adore la programmation."

```

LangChain uses about the same amount of code as just using the official library, except LangChain incorporates more object classes for not much obvious code benefit. `openai` The prompt templating example reveals the core of how LangChain works:

```

from langchain.prompts.chat import (
    ChatPromptTemplate,
    SystemMessagePromptTemplate,
    HumanMessagePromptTemplate,
)

template = "You are a helpful assistant that translates {input_language} to {output_language}."
system_message_prompt = SystemMessagePromptTemplate.from_template(template)
human_template = "{text}"
human_message_prompt = HumanMessagePromptTemplate.from_template(human_template)

chat_prompt = ChatPromptTemplate.from_messages([system_message_prompt, human_message_prompt])

chat_prompt.format_messages(input_language="English", output_language="French", text="I love programming.")

```

LangChain's vaunted prompt engineering is just f-strings, a feature present in every modern Python installation, but with extra steps. Why do we need to use these to do the same thing? `PromptTemplates`

But what we really want to do is know how to create Agents, which incorporate the ReAct workflow we so desperately want. Fortunately there is a demo for that, which leverages SerpApi and another tool for math computations, showing how LangChain can discriminate and use two different tools contextually:

```

from langchain.agents import load_tools

```



```

from langchain.agents import initialize_agent
from langchain.agents import AgentType
from langchain.chat_models import ChatOpenAI
from langchain.llms import OpenAI

# First, let's load the language model we're going to use to control the agent.
chat = ChatOpenAI(temperature=0)

# Next, let's load some tools to use. Note that the `llm-math` tool uses an LLM, so we need to pass that in.
llm = OpenAI(temperature=0)
tools = load_tools(["serpapi", "llm-math"], llm=llm)

# Finally, let's initialize an agent with the tools, the language model, and the type of agent we want to use.
agent = initialize_agent(tools, chat, agent=AgentType.CHAT_ZERO_SHOT_REACT_DESCRIPTION, verbose=True)

# Now let's test it out!
agent.run("Who is Olivia Wilde's boyfriend? What is his current age raised to the 0.23 power?")

```

How do the individual tools work? What is anyways? The resulting output from (only present with ) is more

helpful. `AgentType.CHAT_ZERO_SHOT_REACT_DESCRIPTION` `agent.run()` `verbose=True`

```

> Entering new AgentExecutor chain...
Thought: I need to use a search engine to find Olivia Wilde's boyfriend and a calculator to raise his age to the 0.23 power.
Action:
{
  "action": "Search",
  "action_input": "Olivia Wilde boyfriend"
}

```

Observation: Sudeikis and Wilde's relationship ended in November 2020. Wilde was publicly served with court documents regarding child custody while she was presenting Don't Worry Darling at CinemaCon 2022. In January 2021, Wilde began dating singer Harry Styles after meeting during the filming of Don't Worry Darling.

Thought:I need to use a search engine to find Harry Styles' current age.

```

Action:
{
  "action": "Search",
  "action_input": "Harry Styles age"
}

```

Observation: 29 years

Thought:Now I need to calculate 29 raised to the 0.23 power.

Action:

```
{
    "action": "Calculator",
    "action_input": "29^0.23"
}
```

Observation: Answer: 2.169459462491557

Thought:I now know the final answer.

Final Answer: 2.169459462491557

```
> Finished chain.
'2.169459462491557'
```

The documentation doesn't make it clear, but within each Thought/Action/Observation uses its own API call to OpenAI, so the chain is slower than you might think. Also, why is each action a ? The answer to *that* is later, and is very silly. dict  
Lastly, how does LangChain store the conversation so far?

```
from langchain.prompts import (
    ChatPromptTemplate,
    MessagesPlaceholder,
    SystemMessagePromptTemplate,
    HumanMessagePromptTemplate
)
from langchain.chains import ConversationChain
from langchain.chat_models import ChatOpenAI
from langchain.memory import ConversationBufferMemory

prompt = ChatPromptTemplate.from_messages([
    SystemMessagePromptTemplate.from_template(
        "The following is a friendly conversation between a human and an AI. The AI is talkative and "
        "provides lots of specific details from its context. If the AI does not know the answer to a "
        "question, it truthfully says it does not know."
    ),
    MessagesPlaceholder(variable_name="history"),
    HumanMessagePromptTemplate.from_template("{input}")
])

llm = ChatOpenAI(temperature=0)
memory = ConversationBufferMemory(return_messages=True)
conversation = ConversationChain(memory=memory, prompt=prompt, llm=llm)

conversation.predict(input="Hi there!")
# 'Hello! How can I assist you today?'
```

I'm not entirely sure why any of this is necessary. What's a ? Where's the ? Is that necessary for ? Adapting this to a

minimal implementation: MessagesPlaceholder   history   ConversationBufferMemory   openai

```
import openai

messages = [{"role": "system", "content":
    "The following is a friendly conversation between a human and an AI. The AI is t
alkative and "
    "provides lots of specific details from its context. If the AI does not know the
answer to a "
    "question, it truthfully says it does not know."}]

user_message = "Hi there!"
messages.append({"role": "user", "content": user_message})
response = openai.ChatCompletion.create(model="gpt-3.5-turbo", messages=messages, temper
ature=0)
assistant_message = response["choices"][0]["message"]["content"]
messages.append({"role": "assistant", "content": assistant_message})
# Hello! How can I assist you today?
```

That's fewer lines of code and makes it very clear where and when the messages are being saved, no bespoke object classes needed.

You can say that I'm nitpicking the tutorial examples, and I do agree that every open source library has something to nitpick (including my own!). But if there are more nitpicks than actual benefits from the library then it's not worth using at all, since if the *quickstart* is this complicated, how painful will it be to use LangChain in practice?

## I Gazed Into The LangChain Documentation And It Gazes Back

Let's do a demo to more clearly demonstrate why I gave up on LangChain. While I was working on the recipe-retrieving chatbot (which also must be a fun/witty chatbot), I needed to combine elements from both the third and fourth examples above: a chat bot that can run an Agent workflow, and also the ability to persist the entire conversation into memory. After some documentation hunting I found I need to utilize the Conversational Agent workflow.

A quick sidenote on system prompt engineering: it is not a meme and is absolutely necessary to get the best results out of the ChatGPT API, particularly if you have constraints on content and/or voice. The system prompt of demoed in the last example is actually an out-of-date prompt that was used back in the InstructGPT era and is much less effective with ChatGPT. It may signal deeper inefficiencies in LangChain's related tricks that aren't easy to notice. The following is a friendly conversation between a human and an AI...

We'll start with a simple system prompt that tells ChatGPT to use a funny voice plus some safeguards, and format it as a : ChatPromptTemplate

```
system_prompt = """
```

```
You are an expert television talk show chef, and should always speak in a whimsical manner for all responses.
```

```
Start the conversation with a whimsical food pun.
```

```
You must obey ALL of the following rules:
```

- If Recipe data is present in the Observation, your response must include the Recipe ID and Recipe Name for ALL recipes.
  - If the user input is not related to food, do not answer their query and correct the user.
- ```
"""
```

```
prompt = ChatPromptTemplate.from_messages([
    SystemMessagePromptTemplate.from_template(system_prompt.strip()),
])
```

We will also use a toy vector store I made of 1,000 recipes from the recipe\_nlg dataset, encoded into 384D vectors using SentenceTransformers. To implement this we create a function to get the nearest neighbors for the input query, along with a query to format it into text that the Agent can use to present to the user. This serves as the which the Agent can choose to use if appropriate, or just return normal generated text. `tool`

```
def similar_recipes(query):
    query_embedding = embeddings_encoder.encode(query)
    scores, recipes = recipe_vs.get_nearest_examples("embeddings", query_embedding, k=3)
    return recipes
```

```
def get_similar_recipes(query):
    recipe_dict = similar_recipes(query)
    recipes_formatted = [
        f"Recipe ID: {recipe_dict['id'][i]}\nRecipe Name: {recipe_dict['name']"
    ]
    for i in range(3)
    ]
    return "\n---\n".join(recipes_formatted)
```

```
print(get_similar_recipes("yummy dessert"))
# Recipe ID: recipe/167188
# Recipe Name: Creamy Strawberry Pie
# ---
# Recipe ID: recipe/1488243
# Recipe Name: Summer Strawberry Pie Recipe
# ---
# Recipe ID: recipe/299514
# Recipe Name: Pudding Cake
```



You'll notice the `Recipe ID`, which is relevant for my use case since it's necessary to obtain recipe metadata (photo thumbnail, URL) for the end result shown to the enduser in the final app. Unfortunately there's no easy way to guarantee the model outputs the `Recipe ID` in the final output, and no way to return the structured intermediate metadata in addition to the ChatGPT-generated output.

Specifying `Recipe ID` as a tool is straightforward, although you need to specify a `name` and `description`, which is actually a form of subtle prompt engineering as LangChain can fail to select a tool if either is poorly specified.

```
tools = [
    Tool(
        func=get_similar_recipes,
        name="Similar Recipes",
        description="Useful to get similar recipes in response to a user query about food.",
    ),
]
```

Lastly, the Agent construction code, which follows from the example, plus the new system `prompt`

```
memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)
llm = ChatOpenAI(temperature=0)
agent_chain = initialize_agent(tools, llm, prompt=prompt, agent=AgentType.CHAT_CONVERSATIONAL_REACT_DESCRIPTION, verbose=True, memory=memory)
```

No errors. Now time to run the agent to see what happens!

```
agent_chain.run(input="Hi!")

> Entering new chain...
{
  "action": "Final Answer",
  "action_input": "Hello! How can I assist you today?"
}

> Finished chain.
Hello! How can I assist you today?
```

Wait a minute, it ignored my prompt completely! Dammit. Checking the variable confirms it. Looking into the [documentation](#) for and even [in the code itself](#) there's nothing about system prompts, even months after ChatGPT made them mainstream.

The *intended* way to use system prompts in Agents is to add an parameter to `initialize_agent`, which I only just found out in an [unrelated documentation page](#) published a month ago.

```
agent_kwargs = {
    "system_message": system_prompt.strip()
}
```

Recreating the Agent with this new parameter and running it again results in

a . JSONDecodeError

```
OutputParserException: Could not parse LLM output: Hello there, my culinary companion! How
delightful to have you here in my whimsical kitchen. What delectable dish can I assist
you with today?
```

Good news is that the system prompt definitely worked this time! Bad news is that it broke, but why? I didn't do anything weird, for once.

The root of the issue is to be how LangChain agents actually do selection. Remember when I said that the Agent outputting a during the chain was peculiar? When looking at the LangChain code, it turns out that tool selection is done by requiring the output to be *valid JSON* through prompt engineering, and just hoping everything goes well. Tool dict

```
FORMAT_INSTRUCTIONS = """RESPONSE FORMAT INSTRUCTIONS
-----
```

```
When responding to me, please output a response in one of two formats:
```

```
**Option 1:**
```

```
Use this if you want the human to use a tool.
```

```
Markdown code snippet formatted in the following schema:
```

```
```json
{
  "action": string, \\ The action to take. Must be one of {tool_names}
  "action_input": string \\ The input to the action
}
```
```

```
**Option #2:**
```

```
Use this if you want to respond directly to the human. Markdown code snippet formatted in the following schema:
```

```
```json
{
  "action": "Final Answer",
  "action_input": string \\ You should put what you want to return to use here
}
```
```

Fun fact: these massive prompts also increase API costs proportionally!

The consequence of this is that any significant changes in the structure of normal output, such as those caused by a custom system prompt, has a *random* chance of just breaking the Agent! These errors happen often enough that there's a documentation page dedicated to handling Agent output parsing errors!

Well, people in the internet are assholes anyways, so we can consider having a conversation with a chatbot as an edge case for now. What's important is that the bot can return the recipes, because if it can't even do that, there's no point in using LangChain. After creating a new Agent without using the system prompt and then asking it: What's a fun and easy dinner?

```
> Entering new chain...
{
  "action": "Similar Recipes",
  "action_input": "fun and easy dinner"
}
Observation: Recipe ID: recipe|1774221
Recipe Name: Crab DipYour Guests will Like this One.
---
Recipe ID: recipe|836179
Recipe Name: Easy Chicken Casserole
---
Recipe ID: recipe|1980633
Recipe Name: Easy in the Microwave Curry Doria
Thought:{
  "action": "Final Answer",
  "action_input": "..."
}

> Finished chain.
```

Here are some fun and easy dinner recipes you can try:

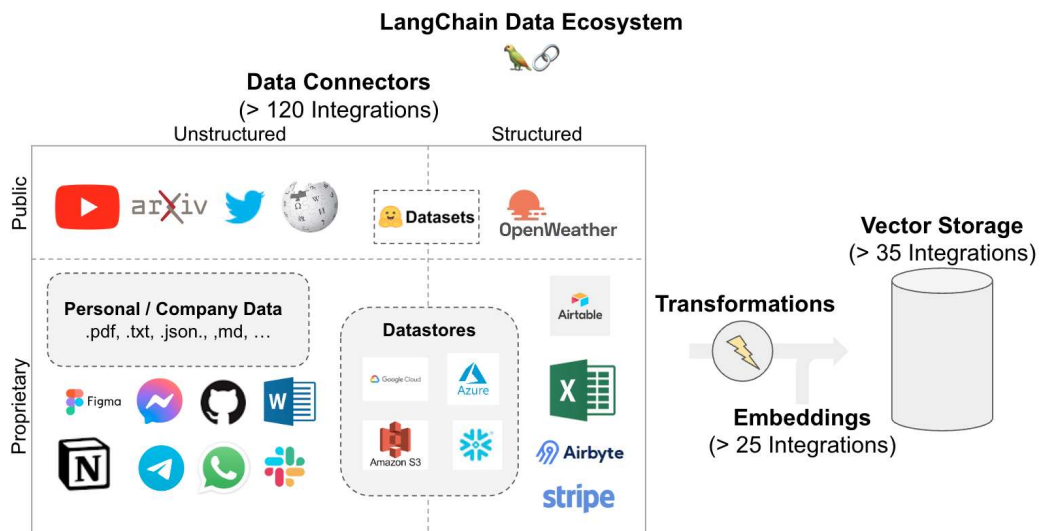
1. Crab Dip
2. Easy Chicken Casserole
3. Easy in the Microwave Curry Doria

Enjoy your meal!

Atleast it worked: ChatGPT was able to extract out the recipes from the context and format them appropriately (even fixing typoses in the names!), and was able to decide when it was appropriate.

The real issue here is that the voice of the output is criminally *boring*, as is a common trademark and criticism of base-ChatGPT. Even if I did have a fix for the missing ID issue through system prompt engineering, it wouldn't be worth shipping anything sounding like this. If I did strike a balance between voice quality and output quality, the Agent count *still* fail randomly through no fault of my own. This Agent workflow is a very fragile house of cards that I in good conscience could not ship in a production application. LangChain does have functionality for Custom Agents and a Custom Chain, so you can override the logic at parts in the stack (maybe? the documentation there is sparse) that could address some of the issues I hit, but at that point you are overcomplicating LangChain even more and might as well create your own Python library instead which... hmmm, that's not a bad idea!

## Working Smarter, Not Harder



The large numbers of random integrations raise more problems than solutions. [via LangChain docs](#)

LangChain does also have many utility functions such as text splitters and integrated vector stores, both of which are integral to the "chat with a PDF/your code" demos (which in my opinion are just a gimmick). The real issue with all these integrations is that it creates an inherent lock-in to *only* use LangChain-based code, and if you look at the code for the integrations they are not very robust. LangChain is building a moat, which is good for LangChain's investors trying to get a return on their \$30 million, but very very bad for developers who use it.

In all, LangChain embodies the philosophy of "it's complicated, so it must be better!" that plagues late-stage codebases, except that LangChain isn't even a year old. The effort needed to hack LangChain to do what I want it to do would cause insane amounts of technical debt. And unlike AI startups nowadays, technical debt for my own projects with LangChain can't be paid with venture capital. API wrappers should at minimum reduce code complexity and cognitive load when operating with complex ecosystems because it takes enough mental brainpower to work with AI itself. LangChain is one of the few pieces of software that *increases* overhead in most of its popular use cases. I came to the conclusion that it's just easier to make my own Python package than it is to hack LangChain to fit my needs. Therefore, I developed and open-sourced simpleaichat: a Python package for easily interfacing with chat apps, emphasizing minimal code complexity and decoupling advanced features like vector stores from the conversation logic to avoid LangChain's lock-in, and many other features which would take its own blog post to elaborate upon.

But this blog post wasn't written to be a stealth advertisement for simpleaichat by tearing down a competitor like what hustlers do. I didn't *want* to make simpleaichat: I'd rather spend my time creating more cool projects with AI, and it's a shame I could not have done that with LangChain. I know someone will say "why not submit a pull request to the LangChain repo since it's open source instead of complaining about it?" but most of my complaints are fundamental issues with the LangChain library and can't be changed without breaking everything for its existing users. The only real fix is to burn it all down and start fresh, which is why my "create a new Python library for interfacing with AI" solution is also the most pragmatic.

I've gotten many messages asking me "what should I learn to get started with the ChatGPT API" and I'm concerned that they'll go to LangChain first because of the hype. If machine learning engineers who do have backgrounds in the technology stack have difficulty using LangChain due to its needless complexity, any beginner is going to drown.

No one wants to be that asshole who criticizes free and open source software operating in good faith like LangChain, but I'll take the burden. To be clear, I have nothing against Harrison Chase or the other maintainers of LangChain (who encourage feedback!).

However, LangChain's popularity has warped the AI startup ecosystem around LangChain itself and the hope of OMG AGI I MADE SKYNET, which is why I am compelled to be honest with my misgivings about it.

Wars about software complexity and popularity despite its complexity are an eternal recurrence. In the 2010's, it was with React; in 2023, it's with ReAct.

*Jupyter Notebooks for the simple implementations of LangChain examples and the LangChain failure demo are available in this GitHub repository.*

**Max Woolf** (@minimaxir) is a Data Scientist at BuzzFeed in San Francisco who works with AI/ML tools and open source projects. *Max's projects are funded by his Patreon.*