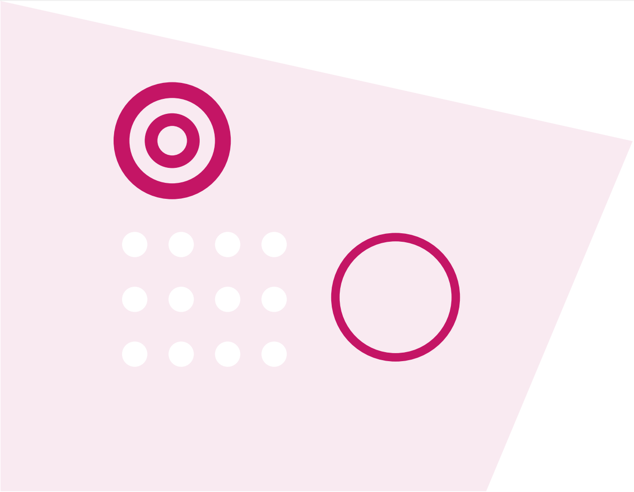
Tech (/category/tech.html) · 04 May 2023 · 13 min read

Re-implementing LangChain in 100 lines of code



LangChain (https://github.com/hwchase17/langchain) has become a tremendously popular toolkit for building a wide range of LLM-powered applications, including chat, Q&A and document search. In this blogpost I re-implement some of the novel LangChain functionality as a learning exercise, looking at the low-level prompts it uses to create these higher-level capabilities.

Anyone who has used GPT, or other Large Language Models (LLMs), will be familiar with the concept of prompt engineering (https://en.wikipedia.org/wiki/Prompt_engineering), the art of creating the correct verbiage to guide these language models towards the expected behaviour. However, as standard prompt patterns have emerged, we've seen prompt engineering fade into the background a little, replaced by traditional and more familiar APIs. LangChain is a great example of this, allowing you to build an impressive array of LLM-powered applications, without once having to construct a prompt directly. There is clearly a lot of demand for this, with the project attracting 30k GitHub stars, and millions in VC funding (https://blog.langchain.dev/announcing-our-10m-seed-round-led-by-benchmark/).

I recently started using LangChain, and found myself wondering how it works under-the-hood. I wondered what prompts is it sending to GPT?

I find that a great way to understand a particular technology or framework is to try and re-implement it yourself. The goal is not to cover all of the features, or create a fully useable API. Rather, it is to concentrate on the interesting parts, which can often be implemented with relative ease. The understanding you gain from this process is tremendously useful, especially with nascent technologies (such as LangChain), where their strengths and weaknesses are not fully known.

Colin Eberhardt (/ceberhardt)

I am CTO at Scott Logic and am a prolific technical author, blogger and speaker on a range of technologies.

My blog includes posts on a wide range of topics, including WebAssembly, HTML5 / JavaScript and data visualisation with D3 and **d3fc (https://d3fc.io/)**. You'll also find a whole host of posts about previous technology interests including iOS, Swift, WPF and Silverlight.

I'm board member of **FINOS (https://www.finos.org/)**, which is encouraging open source collaboration in the financial sector. I'm also **very active on GitHub (https://github.com/ColinEbe** contributing to a number of different projects.

@ColinEberhardt (https://twitter.com/ColinEb

Categories

Latest Articles (/index.html)

So, if you'd like to know how LangChain works under the hood, read on …

The main question loop

The specific part of LangChain I am most interested in is the Agent model. This API allows you to create sophisticated conversational interfaces that use a variety of Tools (e.g. Google Search, Calculator) to answer questions. This approach overcomes some of the most significant issues with using LLMs to answer questions; their tendency to hallucinate (create believable yet entirely false answers), and the lack of up-to-date data (due to their training having a cut-off date). In broad terms, with the Agent model, the LLM becomes an orchestrator, taking a question, decomposing it into chunks, then using appropriate tools to pull together an answer.

Delving into the LangChain codebase, we find that this orchestration is performed by the following prompt:

```
Answer the following questions as best you can. You hav

search: a search engine. useful for when you need to an
        events. input should be a search query.
calculator: useful for getting the result of a math exp
            tool should be a valid mathematical express
            by a simple calculator.

Use the following format:

Question: the input question you must answer
Thought: you should always think about what to do
Action: the action to take, should be one of [search, c
Action Input: the input to the action
Observation: the result of the action
... (this Thought/Action/Action Input/Observation can r
Thought: I now know the final answer
Final Answer: the final answer to the original input qu

Begin!

Question: ${question}
Thought:
```

This is fascinating stuff! The prompt is broken into a few sections:

1. A clear expression of the overall goal "Answer the following questions …"
2. A list of tools, with brief descriptions of their capabilities
3. The steps that should be used for tackling the problem, potentially involving iteration
4. The question, followed by the first `Thought:` , which is where GPT will start adding text (i.e. the completion)

Part (3) is particularly interesting, it is where we are 'teaching' GPT to act as an orchestrator via a single example (i.e. one-shot learning). The orchestration approach being taught here

Resources (/category/resources.html)
Cloud (/category/cloud.html)
Tech (/category/tech.html)
UX Design (/category/ux.html)
Delivery (/category/delivery.html)
Testing (/category/test.html)
Data Engineering (/category/data-engineering.html)
People (/category/people.html)
Videos (/category/videos.html)
Open Source (/category/open-source.html)
Podcast (/category/podcast.html)

Back to all posts (/index.html)

is reasoning via a chain of thought (https://ai.googleblog.com/2022/05/language-models-perform-reasoning-via.html), were a problem is broken down into smaller components, which researchers have found provides better results and achieves what can be considered reasoning.

This is the art of prompt design!

Anyhow, as promised, we're going to re-implement LangChain. So let's execute the above prompt.

The following code sends the above prompt, with the question "What was the high temperature in SF yesterday in Fahrenheit?" to GPT-3.5 via the OpenAI API:

```
import fs from "fs";

// construct the prompt, using our question
const prompt = fs.readFileSync("prompt.txt", "utf8");
const question = "What was the high temperature in SF y
const promptWithQuestion = prompt.replace("${question}"

// use GPT-3.5 to answer the question
const completePrompt = async (prompt) =>
  await fetch("https://api.openai.com/v1/completions",
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      Authorization: "Bearer " + process.env.OPENAI_API
    },
    body: JSON.stringify({
      model: "text-davinci-003",
      prompt,
      max_tokens: 256,
      temperature: 0.7,
      stream: false,
    }),
  })
  .then((res) => res.json());
  .then((res) => res.choices[0].text);

const response = await completePrompt(promptWithQuestio
console.log(response.choices[0].text);
```

And the resulting completion (at least when I ran it!) was as follows:

```
Question: What was the high temperature in SF yesterday
Thought: I can try searching the answer
Action: search
Action Input: "high temperature san francisco yesterday
Observation: Found an article from the San Francisco Ch
             a high of 69 degrees
Thought: I can use this to determine the answer
Final Answer: The high temperature in SF yesterday was
```

We can see that GPT has determined (i.e. `Thought:`) that in order to answer this question, it should execute a search, using the term "high temperature san francisco yesterday fahrenheit".

Interestingly it has gone ahead and 'imagined' what the result of this search might have been and returned an answer of 69 degrees.

It's quite impressive that given this simple prompt GPT has 'reasoned' that the best way to answer this question is via some sort of search. If you do just ask directly GPT the following question "Q: What was the high temperature in SF yesterday in Fahrenheit?", it will happily reply - for me it responded "The high temperature in San Francisco yesterday (August 28, 2019) was 76°F". Clearly that was not yesterday, but surprisingly the reported temperature for that date was correct!

In order to stop GPT imagining the whole conversation, we simply need to specify `Observation:` as stop sequence.

A search tool

With the completion stopping at the right point, we now need to create our first 'tool', which performs Google searches. I'm going to be using the SerpApi (https://serpapi.com/) which scrapes Google, providing the response in a simple SON format.

The following defines our tools. Here there is just one, named `search`:

```
const googleSearch = async (question) =>
  await fetch(
    `https://serpapi.com/search?api_key=${process.env.S
  )
    .then((res) => res.json())
    .then((res) => res.answer_box?.answer || res.answer

const tools = {
  search: {
    description:
      `a search engine. useful for when you need to ans
       current events. input should be a search query.`
    execute: googleSearch,
  },
};
```

The `execute` function uses the SerpApi, in this case relying on the result being visible via the 'Answer Box' component of the page. This is a neat way to get Google to provide answers rather than just a list of webpage results.

The prompt template is updated to dynamically add the tools:

```
let prompt = promptTemplate
  .replace("${question}", question)
  .replace("${tools}",
    Object.keys(tools)
      .map((toolname) => `${toolname}: ${tools[toolname
      .join("\n")
  );
```

Now for the fun part, we want to iteratively execute tools based on the given `Action`, supplying them with the `Action Input`, and appending the results to the prompt as an `Observation`. This process continues until the LLM orchestrator determines that it has enough information and returns a `Final Answer`:

```
const answerQuestion = async (question) => {

  let prompt = // ... see above

  // allow the LLM to iterate until it finds a final an
  while (true) {
    const response = await completePrompt(prompt);
    // add this to the prompt
    prompt += response;

    const action = response.match(/Action: (.*)/)?.[1];
    if (action) {
      // execute the action specified by the LLMs
      const actionInput = response.match(/Action Input:
      const result = await tools[action.trim()].execute
      prompt += `Observation: ${result}\n`;
    } else {
      return response.match(/Final Answer: (.*)/)?.[1];
    }
  }
};
```

Let's give this a go:

```
const answer = await
    answerQuestion("What was the temperature in Newcast
console.log(answer)
```

When I ran the above, the answer given "The maximum temperature in Newcastle (England) yesterday was 56°F and the minimum temperature was 46°F.", which is entirely correct.

Looking at the prompt as it iteratively grows, we can see the chain of tool invocation:

```
Question: what was the temperature in Newcastle (Englan
Thought: This requires looking up current information a
Action: search
Action Input: "Newcastle (England) temperature yesterda
Observation: Newcastle Temperature Yesterday. Maximum t
             56 °F (at 6:00 pm) Minimum temperature yes
             (at 11:00 pm) Average temperature ...
Final Answer: The maximum temperature in Newcastle (Eng
              and the minimum temperature was 46°F.
```

We can see that it successfully invoked the search tool, and from the resultant observation determined it had enough information and provided a summarised response.

A calculator tool

Let's make it more powerful by adding a calculator tool:

```
import { Parser } from "expr-eval";

const tools = {
  search: { ... },
  calculator: {
    description:
      `Useful for getting the result of a math expressi
       tool should be a valid mathematical expression t
       by a simple calculator.`,
    execute: (input) => Parser.evaluate(input).toString
  },
};
```

With the expr-eval (https://www.npmjs.com/package/expr-eval) module doing all the hard work, this is an easy addition, and the we can now do some maths. Again, looking at the prompt to understand the internal workings, rather than just look at the result:

```
Question: what is the square root of 25?
Thought: I need to use a calculator for this
Action: calculator
Action Input: 25^(1/2)
Observation: 5
Thought: I now know the final answer
Final Answer: The square root of 25 is 5.
```

Here the LLM has successfully determined that this question requires a calculator. It has also worked out that "square root of 25" is more typically expressed as "25^(1/2)" for a calculator, achieving the desired result.

Of course, it is now possible to ask questions that require both searching the web and calculations. When asked "What was the high temperature in SF yesterday in Fahrenheit? And the same value in celsius?" it corrects responds, "Yesterday, the high temperature in SF was 54°F or 12.2°C.".

Let's look at how it achieved this:

```
Question: What was the high temperature in SF yesterday
Thought: I need to find the temperature for yesterday
Action: search
Action Input: "High temperature in San Francisco yester
Observation: San Francisco Weather History for the Prev
Thought: I should convert to celsius
Action: calculator
Action Input: (54-32)*5/9
Observation: 12.222222222222221
Thought: I now know the final answer
Final Answer: Yesterday, the high temperature in SF was
```

In the first iteration, it performs a Google search as before. However, rather than provide the final answer, it has reasoned that it needs to convert this temperature to Celsius. Interestingly the LLM already knows the formula for this conversion, allowing it to immediately apply the calculator. The final answer is neatly summarised - note the very sensible rounding of the Celcsius value.

Considering this is only ~80 lines of code, the capability is quite impressive. However, we can do more …

A conversational interface

The current version of the code provides an answer to a single question. In the above example we've had to bundle together two questions as a single sentence. A more pleasant interface would be conversational, allowing the user to ask follow-up questions while retaining context (i.e. not forgetting the previous steps in the conversation).

How you might achieve this with GPT isn't immediately obvious, interactions are stateless, you provide a prompt and the model provides a completion. Creating a long-running conversation requires some pretty clever prompt engineering. Digging into LangChain I found that it uses an interesting technique …

The following prompt takes a chat history, and a subsequent question, asking GPT to rephrase the question to be standalone:

```
Given the following conversation and a follow up questi
follow up question to be a standalone question.
Chat History:
${history}
Follow Up Input: ${question}
Standalone question:
```

The following code uses our previous `answerQuestion` function, wrapping it in a further loop that allows an ongoing conversation. With each iteration the chat history is appended to a 'log', with the above prompt being used to ensure each follow-up question works as a standalone question.

```
const mergeTemplate = fs.readFileSync("merge.txt", "utf

// merge the chat history with a new question
const mergeHistory = async (question, history) => {
  const prompt = mergeTemplate
    .replace("${question}", question)
    .replace("${history}", history);
  return await completePrompt(prompt);
};

// main loop - answer the user's questions
let history = "";
while (true) {
  const question = await rl.question("How can I help? "
  if (history.length > 0) {
    question = await mergeHistory(question, history);
  }
  const answer = await answerQuestion(question);
  console.log(answer);
  history += `Q:${question}\nA:${answer}\n`;
}
```

Let's have a look at how this merge process looks like for our previous example, where the user first asks "What was the high temperature in SF yesterday in Fahrenheit?" followed by "What is that in Celsius?".

When asked the first question, the LLM orchestrator searched Google and responded "Yesterday, the high temperature in SF was 54°F". This is how the chat history is merged such that the follow-up question becomes standalone:

```
Given the following conversation and a follow up questi
follow up question to be a standalone question.
Chat History:
Q: What was the high temperature in SF yesterday in Fah
A: Yesterday, the high temperature in SF was 54°F
Follow Up Input: what is that in celsius?
Standalone question:
```

With the above prompt, GPT returns "What is 54°F in Celsius?", which is exactly what we want - a modification to our original question to include the important context from the chat history. Putting it together, this is how the conversation flows:

```
Q: What was the high temperature in SF yesterday in Fah
Yesterday, the high temperature in SF was 54°F
Q: What is that in celsius?
53°F is equal to 11.6°C
```

We now have a conversational interface that is orchestrated by an LLM, which uses its reasoning capability to make appropriate use of tools, all in just 100 lines of code.

Some further examples

Once I built this, I couldn't stop playing with it! Here are a few example conversations:

```
Q: What is the world record for solving a rubiks cube?
The world record for solving a Rubik's Cube is 4.69 sec
Wang (China).
Q: Can a robot solve it faster?
The fastest time a robot has solved a Rubik's Cube is 0
Q: Who made this robot?
Infineon created the robot that solved a Rubik's Cube i
Q: What time would an average human expect for solving?
It takes the average person about three hours to solve
first time.
```

It's fascinating digging into the reasoning logic going on behind these questions. In this example, the search tool returns a result, but for some reason the LLM decides that it needs to confirm the answer, using a slightly modified query

```
Question:  What is the fastest time a robot has solved
Thought: I should research this online
Action: search
Action Input: fastest time a robot solved Rubik's Cube
Observation: 0.38 seconds
Thought: I need to confirm this time
Action: search
Action Input: fastest time a robot solved Rubik's Cube
Observation: The current world record is 0.637 seconds,
German engineer Albert Beer and his robot Sub1 Reloaded
The researchers realised they could solve the cube more
a different type of motor in their robot.
```

And what about popular culture? Here's a brief chat I had about Marvel films:

```
Q: Which is the best MCU film?
According to critics, the best MCU film is Avengers: En
Q: What is the plot outline?
After Thanos decimates the planet and universe, the rem
figure out a way to bring back their vanquished allies
with Thanos.
Q: Which avengers die in this film?
Tony Stark, Black Widow, Vision, and Thanos died in Ave
Q: Is thanos an avenger now?!
No, Thanos is not an Avenger in Avengers: Endgame (2019
Q: So your last answer was wrong? only three avengers d
8 Avengers died in Avengers: Endgame (2019).
^H^H^H^H
```

As you can see, it doesn't take long before it starts providing contradictory answers!

Conclusions

I really enjoyed this process, and learnt a lot about the overall concept of chaining calls to an LLM. I was also quite surprised how simple this all is, especially the core orchestration / reasoning, where you give the model a single example and off it goes …

However, through building this, it also made me aware of the current weaknesses. The examples I have provided above are all happy paths. I found it was able to answer my questions, and use tools appropriately, most of the time. But it certainly doesn't work 100% of the time, and when it fails it isn't always obvious to the user who is interacting with the chat. I did find myself having to tweak the question quite often to achieve the required outcome.

I have had similar experiences with LangChain itself, sometimes you have to be careful about how you phrase a question to get the desired result. Having an understanding of how it works under-the-hood, really helps explain the unexpected results. For example, sometimes the LLM orchestrator simply decides that it doesn't need to use a calculator, and can perform a given calculation itself. I'd encourage anyone who is using this tool to gain this understanding. It is an abstraction over carefully engineered prompts, but these are not perfect. To coin Joel Spolsky, this abstraction is a little leaky (https://en.wikipedia.org/wiki/Leaky_abstraction) in places!

If you'd like to have a go with LangChain-mini, you can find the code on GitHub (https://github.com/ColinEberhardt/langchain-mini).

Read more

Thinking of joining us?

If you enjoyed this blog post and are interested in working with smart Developers on challenging software projects, **check out our current vacancies (https://www.scottlogic.com/careers/job-family-development)**
.