

The
Pragmatic
Programmers

A Common-Sense Guide to AI Engineering

Build Production-Ready LLM Applications



Jay Wengrow
edited by Katharine Dvorak

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit <https://www.pragprog.com>.

Copyright © The Pragmatic Programmers, LLC.

Augmenting a Prompt with Knowledge

AI-based customer support representatives have been around for a little while now, but they can be annoying to deal with, to say the least. (Press 9 to hear this entire menu again!) With the advent of powerful LLMs, though, it's possible to build customer support functionality that is actually helpful, perhaps even to the point where it seems like you're interacting with a real human.

For the remainder of this book, we're going to work on building such a customer support bot. While this app provides support to customers of a fictional company, you can absolutely apply these same ideas and techniques to the real world. In this chapter, I first explain the nature of the project and the unique challenges we're up against. I then introduce you to the notion of augmenting prompts with *knowledge*, a central idea in AI engineering.

So, let's roll up our sleeves; we have an app to build.

Building a Chatbot

A primary function of our customer support bot will be to serve as an AI-powered “knowledge chatbot,” one that allows a user to ask for information about an organization's proprietary knowledge base. This is quite a common use case for LLM-based software, and can come in one of two forms: customer-facing or internal-facing.

An example of a *customer-facing* knowledge chatbot is a chatbot created by a company for its customers to use to inquire about or get technical support for the company's products. An *internal-facing* chatbot is a chatbot that is only used by an organization's employees, say, to find information about company projects or policies.

In either case, a knowledge chatbot is one which converses with users and conveys information about an organization's proprietary data. Some like to

call this type of chatbot a “chat-with-your-docs” app. We’ll spend the rest of this book building such an app.

Now, to create a knowledge chatbot, we’ll need a proprietary knowledge base. Given that I don’t actually have a trove of private data (and if I did, I wouldn’t disclose it here), here’s what we’ll do instead.

FLOSS Manuals¹ offers a sweet *corpus* (a jargon term for “collection of data”) of software documentation that covers various Free Software² programs. (FLOSS stands for Free/Libre Open Source Software.) Some of the more well-known programs documented on this site include the Firefox web browser and the WordPress content-management system (CMS). In truth, many of these manuals are out of date, but that’s okay for our purposes.

We’re going to rebrand these manuals under different product names, and pretend that these manuals are for software belonging to a fictional company called GROSS, which stands for Gratuitously Rebranded Obsolete Software Suite. The original manuals are licensed under the Creative Commons CC-BY-SA license,³ which allow for remixing and sharing materials even for commercial purposes.

So, GROSS “offers” (or at least, markets) a web browser, a CMS, and other apps corresponding to each manual on the FLOSS Manuals website. *Our goal is to create a knowledge chatbot that provides tech support for the entire suite of GROSS apps.* That’s the main project that we’ll work on in this quickstart guide. To start, we’ll focus our chatbot on one app: GROSS’s flagship web browser, Flamehamster™.

But here’s the thing. How, exactly, do we build a chatbot that knows about Flamehamster? As you’ve learned, an LLM only knows about the data it’s been trained on. Given that no LLM has ever heard of the Flamehamster web browser, how could it answer a prompt such as How do I install Flamehamster on Linux?

Augmenting with Knowledge

Teaching an LLM about proprietary data is one of the classic problems of AI engineering, and there are a number of solutions to it. Here, I introduce one general and highly effective solution, which is to *augment our prompts with knowledge*.

1. <https://flossmanuals.net>
2. <https://www.fsf.org>
3. <https://creativecommons.org/licenses/by-sa/4.0/deed.en>

Here's what I mean. In the previous chapter, I introduced the technique of prompt augmentation. In particular, I demonstrated how we can augment a prompt with instructions to the LLM on how to behave. However, we can also augment prompts with other things as well. Here, our strategy will be to augment the user prompt with our *proprietary data itself*.

This can be done in numerous ways, but we'll start with the simplest approach. That is, we'll augment the user prompt with *our entire dataset*, which in our case, will be all the contents of the Flamehamster manual. Yes, we'll include the entire manual inside our prompt!

Now, our manual is currently in the form of a PDF, which is difficult to include inside a programmatic prompt. As such, we'll first convert our PDF into some sort of text file, such as raw text or Markdown. Once the manual is in this form, we can then easily load it as a string in our Python code.

Here's a more specific version of what our prompt will look like. Let's say the user input is stored in a variable called `user_input`. We can create a second variable called `documentation`, which is a giant string containing the contents of the Flamehamster manual. We then send the LLM a prompt that looks something like this:

```
history = [
    {"role": "developer", "content": f"""You are an AI customer support
    technician who is knowledgeable about software products created by
    the company called GROSS. One such product is a web browser called
    Flamehamster. You are to answer user queries below solely on
    the following documentation: {documentation}"""},
    {"role": "assistant", "content": "How can I help you today?"},
    {"role": "user", "content": user_input},
]
```

With this prompt, the LLM has all the information it needs to answer the `user_input`. With this knowledge-based prompt augmentation, it's okay that the LLM was never trained on Flamehamster data, since all the Flamehamster documentation is included in the prompt itself.

Knowledge-based prompt augmentation is a brilliant but simple tactic that can work well in many applications. Note that we're augmenting the prompt with both knowledge *and* instructions. That is, in the prompt we're providing knowledge in the form of documentation and also adding instructions to the LLM to use the documentation when answering user queries.

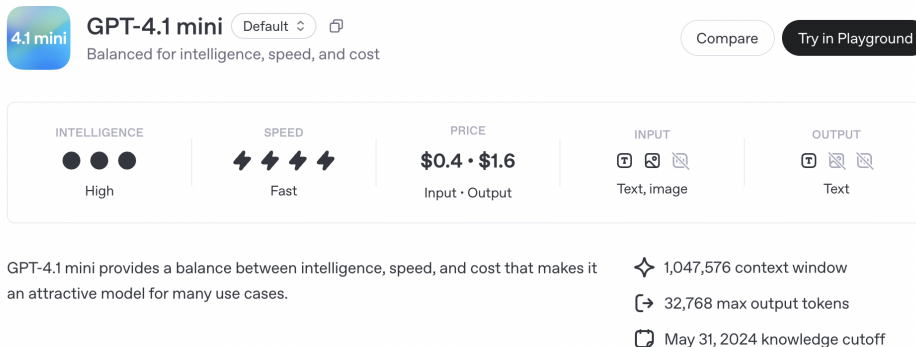
However, there are some downsides to this particular prompt. At a cursory glance, the prompt looks like it's only a few lines long—but in fact, it's *huge*! The `documentation` variable contains all the contents of the Flamehamster

manual, which is well over 3,000 lines long. This will have negative ramifications on several fronts, most of which I'll dig into at the end of the chapter. However, there's one pressing issue we need to deal with straight away.

Avoiding Context Window Limitations

As discussed in [Diving into Tokens, on page ?](#), each LLM has a context window of a limited size, meaning that the model cannot process prompts that exceed a certain number of tokens. So, we need to be absolutely sure that our prompt can fit inside the context window.

The first thing we need to do, then, is check the OpenAI docs to find the context window size of our model. Say, for example, that we're using the GPT-4.1-mini language model. When we look up its specs page (shown in the following figure), we'll see that it has a "1,047,576 context window," which means that our prompt cannot exceed 1,047,576 *tokens*. But how do we determine the number of tokens in our prompt?



One way to do this for OpenAI models is to hop over to OpenAI's Tokenizer web page⁴ and paste our prompt in the text box there. In the next section, I show you how I converted our PDF into a Markdown file called `flamehamster.md`. When I copy the contents of `flamehamster.md` and paste it into the tokenizer text box, the site tells me that `flamehamster.md` contains 44,908 tokens.

However, using the Tokenizer site may become frustrating to use if we'd have multiple files to paste in. (I also don't know if there's some limit to the amount of text we can drop in there.) Fortunately, we can also count tokens programmatically.

4. <https://platform.openai.com/tokenizer>

OpenAI has a Python library called tiktoken,⁵ whose primary use is to convert text into tokens for use in OpenAI models. At the same time, we can utilize tiktoken to count the number of tokens in any text. To do this, we convert our text into tokens, and then count how many tokens there are:

```
import tiktoken

# Choose the encoding model
encoding_model = tiktoken.encoding_for_model("gpt-4")

# Your data
with open("flamehamster.md", "r", encoding="utf-8") as file:
    text = file.read()

# Count tokens
tokenized_text = encoding_model.encode(text)
num_tokens = len(tokenized_text)

print(f"Token count: {num_tokens}")
```

Keep in mind that OpenAI has a different token system for different families of models. Here, we're using the token system of the gpt-4 family of models, which includes GPT-4.1-mini.

When I ran the previous code, the output declared that the flamehamster.md file contains 44,442 tokens. I can't explain the discrepancy between this and the number 44,908 that I was given by the Tokenizer website, but the two numbers are close enough to make me confident that flamehamster.md has *around* 45,000 tokens. Fortunately, 45,000 is way less than GPT-4.1-mini's context window of 1,047,576, so GPT-4.1-mini should certainly be able to process our prompt. We've dodged a bullet—for now.

Armed with our strategy of knowledge-based prompt augmentation, let's get to work building our chatbot.

Preparing the Data

A crucial step in building a knowledge chatbot is ensuring that our proprietary data is in a format the chatbot can effectively use. I'll refer to this general process as *data preparation*. Data preparation can be challenging, and how it's done will vary widely from application to application.

The first thing to consider is how the proprietary data is *currently* stored. Is it in a database? Does it take the form of an online wiki? Is it a folder of PDF files? Data might also be spread out across multiple formats. Depending on

5. <https://github.com/openai/tiktoken>

how your data is currently stored, it can be a complex job to transform them all into one, unified format.

It would be impossible to cover all the different transformation methods in this guide, so we'll just work with what we have for our example project. Luckily, in our case, all of our docs are contained as PDF files, so we only have one format to deal with.

However, PDF isn't the most natural format to deal with programmatically. It may be easy for a human to read a PDF file, but computers have a more difficult time with it. And so, we'll probably do better if we can convert our PDF files into a format more easily read by a Python program.

As to what format we want to convert our data *to*, we have a couple of options. The simplest option would be raw text (think .txt files). Indeed, this can be quite effective since LLMs are great at processing raw text. However, there's an arguably better option, which is *Markdown*. Markdown is very much like raw text but has the added benefit that it organizes data into various sections using things like headings and subheadings. (You may be familiar with Markdown already, but if you're feeling rusty on the details, you can find the syntax docs [here](https://www.markdownguide.org/basic-syntax).⁶) This organized structure can help the LLM find more easily what it's looking for. For example, if a user asks a question about accessibility in Flamehamster and the LLM notices that there's a heading that says `## ACCESSIBILITY FEATURES`, that's a cue to the LLM to look in that section.

So our first goal is to transform the Firefox manual from the FLOSS Manuals from a PDF file to Markdown. Once we do that, we'll replace the word "Firefox" with "Flamehamster" and the organization name "Mozilla" (the creator of Firefox) to "GROSS."

Besides the main FLOSS Manuals webpage, there is also a separate archived site⁷ for manuals older than 2015. The primary Firefox manual can only be found on this archived site. (The newer site only has a secondary "Firefox in a Nutshell" manual, which is too brief for our purposes.)

If you want to cut to the chase, the URL for the manual we'll be using is https://archive.flossmanuals.net/_booki/firefox/firefox.pdf. Read it and you'll feel like you've traveled back in time a couple of decades!

6. <https://www.markdownguide.org/basic-syntax>

7. <https://archive.flossmanuals.net>

In truth, there are many tools available for converting a PDF file into Markdown. Technically, we don't even need to use code; we can use some drag-and-drop software as well.

However, it's important to know that when it comes to data transformation, the quality of tools can vary widely. Also, certain tools may work better on one dataset and worse on another dataset, even if the data is all in the same format. For example, one PDF-to-text tool might work better on PDF files that contain images or charts, while another tool might work better on PDF files that contain hard-to-read text. It turns out that data transformation can be quite a project—and deserves a book unto itself. For now, though, just understand that it can take a lot of careful work to do data transformation right.

One PDF-to-Markdown tool that is generally promising and seemed to work well for the FLOSS PDFs is Docling.⁸ After installing the docling dependency with uv (uv add docling), here's the code I ran to transform the Firefox PDF to Markdown:

```
from docling.document_converter import DocumentConverter

source = "https://archive.flossmanuals.net/_booki/firefox/firefox.pdf"
converter = DocumentConverter()
doc = converter.convert(source).document

with open("flamehamster.md", "w", encoding="utf-8") as f:
    f.write(doc.export_to_markdown())
```

It took some time to run, but eventually produced a Markdown file called flamehamster.md containing all the text from the Firefox PDF. The images were replaced with simple <!-- image --> placeholders, so our LLM will not get to see the contents of those images. However, this manual happened to be written in such a way where the images are mostly supplemental; you can get all the information you need from the actual text.

I then did a simple find/replace to swap out all the instances of the word “Firefox” for the word “Flamehamster.” I also did the same for the word “Mozilla” which I changed to “GROSS.” You can find a copy of flamehamster.md in this book's code repository.⁹

8. <https://github.com/docling-project/docling>

9. <https://www.pragprog.com/titles/jwpaieng>

Implementing the Knowledge Chatbot

Now that we have our data in the desired Markdown format, we're ready to implement our chatbot using the strategy of knowledge-based prompt augmentation. For our first test run, I'm using the GPT-4.1-mini model. Here goes:

```
from dotenv import load_dotenv
from openai import OpenAI

load_dotenv()
llm = OpenAI()

with open("flamehamster.md", "r", encoding="utf-8") as file:
    documentation = file.read()

assistant_message = "How can I help you today?"
print(f"Assistant: {assistant_message}\n")
user_input = input("User: ")
history = [
    {"role": "developer", "content": f""You are an AI customer support
    technician who is knowledgeable about software products created by
    the company called GROSS. One such product is a web browser called
    Flamehamster. You are to answer user queries below solely on
    the following documentation: {documentation}""},
    {"role": "assistant", "content": assistant_message},
    {"role": "user", "content": user_input},
]

while user_input != "exit":
    response = llm.responses.create(
        model="gpt-4.1-mini",
        temperature=0,
        input=history
    )

    print(f"\nAssistant: {response.output_text}")

    user_input = input("\nUser: ")

    history += [
        {"role": "assistant", "content": response.output_text},
        {"role": "user", "content": user_input}
    ]
```

Everything in this code is likely familiar to you. We're using the history variable to manage conversation state, and we're using the prompt I described earlier in [Augmenting with Knowledge, on page 4](#) that includes the entire documentation in the prompt itself.

Test Driving the Flamehamster Chatbot

Okay, let's test drive this thing. I've read over the documentation and picked out some topics that the chatbot should hopefully be able to answer.

My first prompt was: Can I restore a closed tab?

Here's what our chatbot had to say:

Assistant: Yes, you can restore a closed tab in Flamehamster. Here's how:

1. Click the **History** menu at the top of the Flamehamster window.
2. Select **Recently Closed Tabs**.
3. From the list, click the name of the tab you want to reopen.

Alternatively, you can:

- Press **Ctrl+Shift+T** (Cmd+Shift+T on Mac) to reopen the last closed tab. Pressing this shortcut repeatedly will reopen previously closed tabs in reverse order.

You can also choose **Open All in Tabs** from the Recently Closed Tabs menu to restore all recently closed tabs at once.

Let me know if you need help with anything else!

It sounds like this could be right! Let's check the manual to be sure. It seems like the chatbot is getting its information from the section at line 1434. It says there:

RESTORING CLOSED TABS

Flamehamster tracks the tabs that you have recently closed. To restore one or all closed tabs, follow these steps:

1. Click the following menu command: History > Recently Closed Tabs .
2. Do any of the following:
 - * Click the name of the tab that you want to re-open.
 - * Click Open All in Tabs to restore all of the closed tabs.
 - * Press Ctrl+Shift+T to open each tab one by one in reverse order.

Wow, the chatbot really nailed it—sweet!

Let's try one other prompt for now: How do I clear my history?

Here was the response:

Assistant: To clear your browsing history in Flamehamster, follow these steps:

1. Click on the **Tools** menu at the top of the Flamehamster window.
2. Select **Clear Recent History**.
3. In the Clear Recent History dialog, choose the time range to clear from the drop-down menu at the top (e.g., last hour, last two hours, or Everything to clear all history).
4. Click the down arrow next to **Details** to expand the list of data types.

5. Select the types of data you want to clear, such as:
 - Browsing & Download History
 - Form & Search History
 - Cookies
 - Cache
 - Active Logins
 - Offline Website Data
6. Once you have selected the desired items, click ****Clear Now****.

Please note that clearing history cannot be undone.

If you want to prevent Flamehamster from storing history in the future, you can configure privacy settings under Tools > Options (or Preferences) > Privacy panel.

Let me know if you want instructions on that!

This sounds plausible, but is it accurate? In looking at the actual manual, I've found that the LLM must be deriving most of its response from a section that begins on line 2549. Here's what it says there:

CLEARING YOUR HISTORY

Flamehamster allows you to clear some or all of your browsing history. This will delete information from your profile and from disk that could be used to identify your activities online.

<!-- image -->

Clearing Flamehamster History:

1. Click on Tools > Clear Recent History .
2. Flamehamster will display the Clear Recent History dialog, with a drop-down menu for the time period to clear. You can choose between the last hour and all saved history.
3. If you only wish to delete some of your history, such as the information stored for forms or in search history, click Details . This will open a list of items stored by Flamehamster, such as Cookies, Cache, and Form & Search History.
4. Once you have selected the information you wish to delete, click Clear Now. Note that this cannot be undone.

It turns out that the AI seemed to get it basically right. If we look *really* carefully, though, we might spot a couple of potential hallucinations, those “facts” that an LLM completely makes up. For example, the chatbot told us that some of the types of things we can clear from our history include “Active Logins” and “Offline Website Data.” However, it’s not clear from the actual manual if those options actually exist. Perhaps this is a small detail, but one certainly worth noting.

Now, it's your turn. Test out this little chatbot based on your own prompts, and compare the responses with the manual. What do *you* think about the quality of our chatbot? There's no wrong answer to this question.

I'm curious to see if the quality changes when we upgrade our model from gpt-4.1-mini to gpt-4.1. When I do this and enter a prompt, though, I get this surprising response:

```
openai.RateLimitError: Error code: 429 - {'error': {'message': 'Request too
large for gpt-4.1 in organization org-rHItPtE5zH8RH3F7En9Z4Qb0 on
tokens per min (TPM): Limit 30000, Requested 45012. The input or output
tokens must be reduced in order to run successfully. Visit
https://platform.openai.com/account/rate-limits to learn more.',
'type': 'tokens', 'param': None, 'code': 'rate_limit_exceeded'}}
```

Oof. It turns out that there's another hurdle I didn't see coming. If we read the error message carefully, we'll see that it's not about the context window (which was indeed large enough to contain our prompt), but about a separate issue of *rate limiting*. When I visit the link from the error message (<https://platform.openai.com/account/rate-limits>), I see this:

Rate limits

API usage is subject to rate limits applied on tokens per minute (TPM), requests per minute or day (RPM/RPD), and other model-specific limits. Your organization's rate limits are listed below.

Visit our [rate limits guide](#) to learn more about how rate limits work.

📌 **Note:** Limits for specific model versions may vary, expand the table to see all models.

MODEL	TOKEN LIMITS	REQUEST AND OTHER LIMITS	BATCH QUEUE LIMITS
gpt-4.1	30,000 TPM	500 RPM	900,000 TPD
gpt-4.1-mini	200,000 TPM	500 RPM	2,000,000 TPD
gpt-4.1-nano	200,000 TPM	500 RPM	2,000,000 TPD
o4-mini	200,000 TPM	500 RPM	2,000,000 TPD
gpt-4o	30,000 TPM	500 RPM	90,000 TPD
gpt-4o-mini	200,000 TPM	500 RPM 10,000 RPD	2,000,000 TPD

My rate limits displayed here are based on how much money I spend at OpenAI. The limits get increased as one becomes a better paying customer, but at my simple usage level, I'm allowed only 30,000 tokens per minute (TPM) for the gpt-4.1 model. As the error message says, though, my current request contains 45,012 tokens, which exceeds this limit.

Now, gpt-4.1-mini allows for 200,000 TPM, but this doesn't mean that our chatbot will operate well even if we have it use gpt-4.1-mini. If even a single user makes five requests within a minute, our app will be throttled (since

45012 * 5 = 225060). And if *multiple* users interact with our chatbot at once, we'll be very, very throttled. So, we've hit a pretty solid wall.

Switching from OpenAI to a self-hosted model would only partially solve this problem. Although we wouldn't be subject to official rate limits, there's a reason why rate limits exist. When any computer is bombarded with too many tasks at once, it slows down. Likewise, if too many users are hitting your LLM all at once, the LLM inference can become too slow to be effective.

Running into PACKing Problems

Despite this setback, we're on the right track, for knowledge-based prompt augmentation is a viable strategy *in general*. It's just that our *specific implementation* of knowledge-based augmentation is problematic, at least for our use case.

Although we've only discussed one specific form of knowledge-based augmentation so far, I'm going to give it a name to distinguish it from other forms we'll encounter in later chapters. I'm going to call this type of prompt a "PACK"—which stands for Prompt with All Corpus Knowledge. That is, we're creating a prompt that contains our entire corpus of proprietary data. (The verb form of PACK is PACKing, of course.)

We've seen a couple of problems with PACKing, but let me present a more complete list:

1. A PACK may be too large for the context window.
2. PACKing may cause our app to exceed rate limits.
3. PACKing can be expensive, since each prompt has many input tokens.
4. There's increased latency when an LLM has to process a large PACK.
5. The larger the PACK, the more likely the LLM will miss some details buried inside it.

Each of these problems are considerable; let's walk through them briefly.

Although the chatbot we built skirted around the context window issue, our final project won't. Remember, we eventually want our chatbot to cover the entire GROSS suite of apps, and they won't all fit in the context window at once. (I did the math.)

We've already experienced the pain of rate limits firsthand. Even the Flamehamster manual *alone* got throttled.

If you recall from [Checking API Usage, on page ?](#), I made 34 requests that cost a total of 9 cents. With the Flamehamster app, when I have a single

session and ask a total of just *three* questions, I already rack up a bill of 6 cents.

Remember, each time the user takes a turn, the entire conversation history is included in the prompt. And because this history contains all the documentation, this means that the documentation is included at the top of the prompt on each and every turn. It emerges, then, that each user turn is relatively costly. Similarly, because each PACK is large, the LLM responds at a slower clip since it has to process the entire documentation on each conversation turn.

The fifth problem I listed is one I haven't touched on yet, but it's important. There's research¹⁰ that when it comes to long prompts, an LLM may lose sight of some details buried in the middle of the prompt. In our case, this means that an LLM may miss important facts from the manual, but those facts might be needed to correctly answer a particular user query! As a result, PACKing may lead to reduced accuracy in some LLM responses.

Because of all the aforementioned problems, it turns out that while augmenting a prompt with knowledge may be a good idea, doing this through PACKing isn't suitable for the GROSS chatbot. It may work for other apps with a smaller proprietary dataset, but we're going to need some other approach.

Wrapping Up

In this chapter, you explored the fundamental AI engineering technique of knowledge-based prompt augmentation, which allows an LLM to answer queries about an organization's proprietary data. At the same time, our main project is underway! We started to apply knowledge-based augmentation to the GROSS chatbot. So far our app seems promising—assuming there's only a single user who isn't querying the app too many times in succession, and we stick with GPT-4.1-mini—as it gave some decent answers about the Flamehamster web browser. Honestly, that's pretty cool. (Or hot?)

Yet, we encountered some serious hurdles with our specific implementation of PACKing our prompts. This approach may perhaps work for other apps, but the GROSS chatbot will need a more sophisticated approach. In the next chapter, I introduce one such solution. It's called *retrieval-augmented generation (RAG)*, and is a foundational concept of AI engineering.

10. <https://arxiv.org/abs/2307.03172>