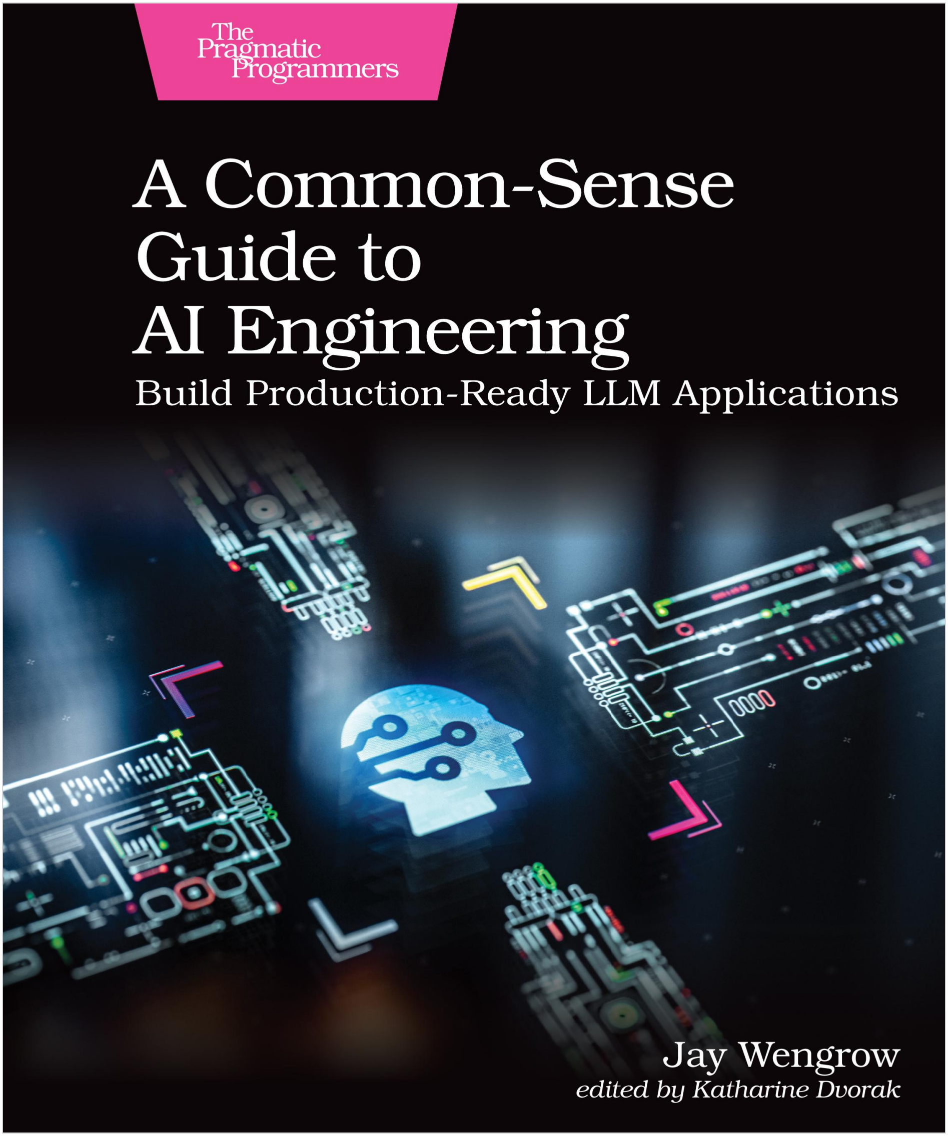


This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

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

You've seen how the agent loop can turn an LLM into a machine that solves complex problems. However, you've also seen that an agent can present an incredible number of failure modes. It may even seem that an agent has a greater chance of failing than being successful! Although it helped to give the model a specific plan, it would be nice if we can make an agent even more dependable. In this chapter, you'll learn a technique that will do just that.

Specifically, you'll learn how to architect *agentic workflows*, which can reduce much of the autonomy that can cause an LLM to go off the rails. We'll apply these workflow techniques to the podcast-producing app we created in the previous chapter, and watch how dependable the app becomes. While no AI app is ever perfect, the agentic workflow approach definitely brings us closer to agentic excellence.

Designing an LLM Assembly Line

With the agent loop from Constructing an Agent Loop, on page ?, we've taken an LLM, given it tools, and told it, "GO!" Assuming that the agent has the right tools to fulfill the user's request, the agent will use the tools in the right sequence to get the job done.

Or not.

Although there may be *some* way to use the tools in concert to accomplish the task, the agent may follow some other, suboptimal approach. After all, the agent is an SNWP, not a brilliant tactician.

You saw in Giving the Agent a Plan, on page ? how instructing the model to follow a detailed plan helped for apps designed to accomplish one specific kind of task, such as producing research-based podcasts. This helped significantly, since the agent no longer has to be a planner. Rather, it can just follow the steps of the plan given to it by a brilliant tactician, *you*.

Even so, an agent can still mess up. Among other things, the LLM may not always remember to follow the plan. However, there's a way to tighten up our agent further to make it more foolproof. And that is to build something called an *agentic workflow*. This term has been popularized by LLM provider Anthropic in a paper[1] on this topic, one well worth reading. Although agentic workflows aren't appropriate where an agent is intended to be multi-purpose, if an agent is supposed to do just one thing (or a small set of very similar things), agentic workflows are often the way to go.

1. https://www.anthropic.com/engineering/building-effective-agents

You can think of an agentic workflow as being a kind of "assembly line" of LLMs. One of the most valuable things about assembly lines in the physical world is that each worker does one thing and does it *really* well.

We can extend this idea to agents as well. An agent that can do lots of things usually doesn't know how to do any of those things particularly well. It's the proverbial "jack of all trades but master of none." But if we can take our task and divide it up among *numerous* LLMs that each do one thing well, our task can "flow" from model to model, with each one doing its special thing just right.

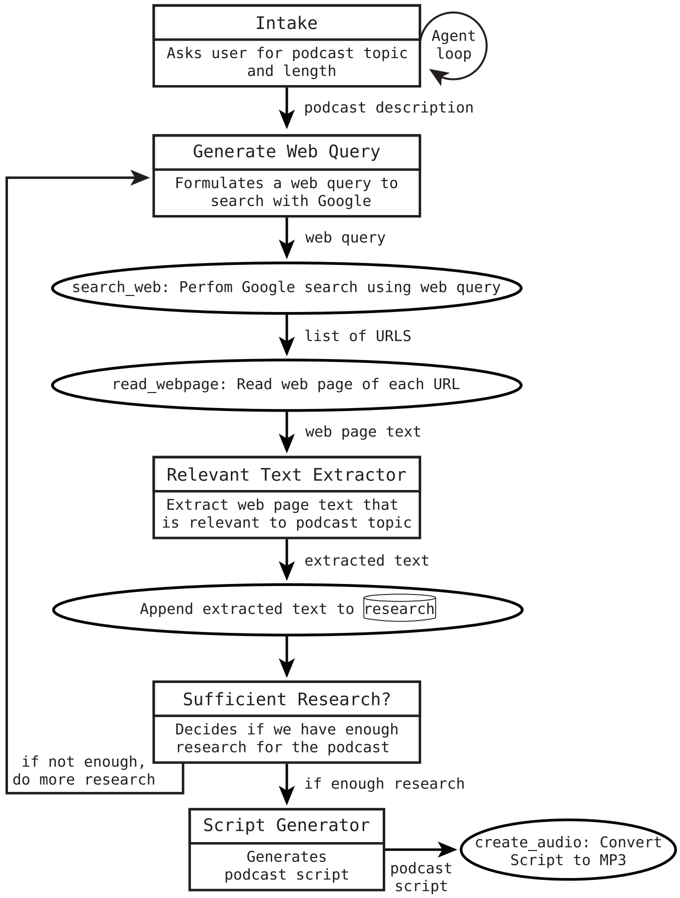
Another great thing about assembly lines is that they follow a very specific sequence, and it's virtually impossible to do things out of order. For example, a car will always have its doors put on before the car reaches the worker tasked with painting the car. Similarly, with an LLM assembly line, we ensure that the LLM charged with creating a podcast script gets the go-ahead to start *only after* another LLM completed all the necessary research.

Let's visualize what an agentic workflow can look like for the podcast-producing app we created in the previous chapter. As a reminder, here's the general plan we want our agent to follow:

1. Use the `search_web` tool to find URLs of web pages with relevant information.
2. Use the `read_webpage` tool to read each URL.
3. If there isn't enough material from these web pages to create a podcast script of the desired length, go back to Step 1.
4. Once there's enough material, use it to write a podcast script.
5. Use the `create_audio` tool to convert the script into an mp3 audio file.

With this plan in place, we can now update our agent's code to follow an assembly-line style workflow. There are many possible ways to structure this, but I'll illustrate one particular approach. Following is an overview of how it works, after which I'll walk through the code.

Let's start with an overview diagram. LLMs are symbolized with rectangles, while regular Python code is symbolized by ovals.



This workflow is an assembly line of LLMs and regular Python functions that put together a beautiful, shiny podcast. This will become clearer once you see the code itself, but it's crucial to understand that the sequence of steps is executed by *regular, deterministic code*. It's just that some of the individual steps happen to involve an LLM.

Here's the gist of what happens at each step:

1. The user interacts with the "Intake Assistant," which also happens to be the LLM of our main conversation loop. This LLM asks the user for their podcast preferences, including the podcast topic and length. The user may enter, as an example, a half-hour podcast explaining the physics and science of rocket engines, explained to me like I'm five years old. This LLM also happens to run an agent loop, but I'll talk about that more in a bit.
2. Once the Intake Assistant feels that it understands what the user wants, it then sends the user's podcast preferences to a second LLM, one that

specializes in generating web queries. This model decides, based on the user's chosen podcast topic, what appropriate query should be used to search the web. For example, this model may generate a query of `how rocket engines work`.

3. Next, our code calls the `search_web` Python function to search the web using the web query provided by the LLM of the previous step. This pulls an array of URLs.
4. Our code calls the `read_webpage` Python function on each URL.
5. Our code sends the text of each web page to an LLM dedicated to extracting whatever text is relevant to the podcast.
6. This extracted text gets appended to a `research` array, which is simply a repository that stores all relevant research we've found so far.
7. After completing the above loop, our code asks another LLM to decide if the `research` array contains enough relevant material to fill a podcast of the user's desired length.
8. If this LLM decides that there is *not* enough research, our code goes back to Step 2, proceeding once again to generate a new web query and perform more web research.
9. Once it's decided that we *do* have enough research, our code asks yet another LLM to write a podcast script based on the material in the `research` array.
10. Finally, our code runs the `create_audio` Python function on the podcast script to create the mp3 file.

If this still seems fuzzy, that's okay. The code that follows should clarify everything.

Implementing an LLM Assembly Line

On that note, here's the code for implementing the workflow for producing podcasts. Beyond the main conversation loop itself, the `initiate_podcast` function is what carries out the entire sequence of assembly line steps. The functions for the individual steps appear first, followed by the `initiate_podcast` function, followed by the main conversation loop:

agentic_systems/chatbot.py

```python
import os
import json
from dotenv import load_dotenv
from openai import OpenAI
from llm_tools import read_webpage, search_web, create_audio
import requests
from bs4 import BeautifulSoup
```

```python
load_dotenv()
llm = OpenAI()
google_api_key = os.getenv("GOOGLE_API_KEY")
search_engine_id = os.getenv("SEARCH_ENGINE_ID")

research = []
previous_web_queries = []

def llm_response(prompt, tools):
    response = llm.responses.create(
        model="gpt-5-mini",
        tools=tools,
        input=prompt
    )
    return response

def generate_web_query(podcast_description):
    web_query = llm.responses.create(
        model="gpt-5-mini",
        input=f"""You are doing research for a podcast, whose details are:
        <details>{podcast_description}</details> Generate a Google web
        search query to help do research for this podcast.

        <instructions>
        Ensure that your web search query is different than any of the past
        queries made: <old_query_list>{previous_web_queries}</old_query_list>.
        The web search query should be specific and succinct, no more than 6
        words. For example, if the podcast topic is about the public
        reception of ChatGPT 5, your query would be simply: public reception
        of ChatGPT 5
        </instructions>
        Generate the web query now:"""
    )
    # Save web query in global previous_web_queries variable so we don't
    # ever repeat the same search again:
    previous_web_queries.append(web_query.output_text)
    return web_query.output_text

def extract_text(urls, podcast_description):
    for url in urls:
        webpage_text = read_webpage(url)
        extracted_text = llm.responses.create(
            model="gpt-5-mini",
            input=f"""You are doing research for a podcast, whose details
            are: <details>{podcast_description}</details>
            Extract whatever relevant information you can from this info
            you've found on the web: <webpage>{webpage_text}</webpage>.
            Just include the extracted text and nothing else in your
            response. Generate the extracted text now:"""
        )
        research.append(extracted_text.output_text)

def has_sufficient_research(podcast_description):
```

```
    sufficient_research = llm.responses.create(
        model="gpt-5-mini",
        input=f"""You are doing research for a podcast, whose details are:
        <details>{podcast_description}</details>. Here is the research you
        have from the web so far: <research>{research}</research>. Do you
        feel that you have enough info to create a fact-based podcast
        based on this research with the information you have so far?
        Keep in mind the desired length of the podcast.
        Respond with either: True/False
        """
    )
    return sufficient_research.output_text

def write_podcast_script(podcast_description):
    script = llm.responses.create(
        model="gpt-5-mini",
        input=f"""You are a podcast scriptwriter, creating scripts for
        news-based and explainer podcasts. The podcast should be based on
        real facts and web research. As such, do not create any fictional
        information for the podcast. Only use what you find based on your
        web research.

        Here are the details of what the podcast should be:
        <details>{podcast_description}</details> Here is the research you
        should use to produce the script: <research>{research}</research>

        The script is read by a single host in a news-like style. Do not
        create music or the like. Only create the words to be spoken by
        the host. Create the script now:"""
    )
    return script.output_text

def initiate_podcast(podcast_description):
    # Generate a web query based on podcast description:
    web_query = generate_web_query(podcast_description)
    # Peform a Google search based on the web query:
    urls = search_web(web_query)

    # Read each url's web page and extract relevant text, storing
    # it in the global variable called research:
    extract_text(urls, podcast_description)

    # Check whether we have enough research to create podcast:
    if has_sufficient_research(podcast_description) == "False":
        # If not enough research, we recursively call initiate_podcast
        # to continue research:
        initiate_podcast(podcast_description)

    # Write podcast script based on research:
    podcast_script = write_podcast_script(podcast_description)
    # Convert podcast script to mp3:
    create_audio(podcast_script)
    return True
```

```
TOOLS = [
    {
        "type": "function",
        "name": "initiate_podcast",
        "description": """Generates an audio podcast as a file
                       called podcast.mp3""",
        "parameters": {
            "type": "object",
            "properties": {
                "podcast_description": {
                    "type": "string",
                    "description": """A description of what type of podcast
                                   the user wishes to create, including
                                   topic and length""",
                }
            },
            "required": ["podcast_description"],
        },
    },
]

TOOL_FUNCTIONS = {"initiate_podcast": initiate_podcast}

print(f"Assistant: What podcast would you like me to create?\n")
user_input = input("User: ")
history = [
    {"role": "developer", "content": """You are a podcast producer, creating
     news-based and explainer podcasts for people on any topic they choose.

    Here is the plan you should follow step by step to create a podcast:
    <plan>
    1. When the user describes the podcast they want, do not proceed to the
    next step until you've obtained the following information:
        * The topic of the podcast.
        * How long the podcast should be. (For example, five minutes long.)
    However, do not ask the user about the podcast style. Assume that the
    podcast style is a single host reporting news and insight.
    2. Next, create a simple summary describing the type of podcast the user
    wants. Ensure that this summary includes the desired time length of the
    podcast. For example, the summary might be: "A 3-minute podcast on the
    latest news, insights, and updates in the field of quantum physics"
    3. You have access to an initiate_podcast tool. Your next step is
    to call the initiate_podcast tool, passing along your summary to it.
    </plan>
    """},
    {"role": "assistant", "content": "How can I help you today?"}
]

while user_input != "exit":  # Main conversation loop
    history += [{"role": "user", "content": user_input}]

    while True:  ## the "agent loop"
        response = llm_response(history, TOOLS)
```

```
        history += response.output
        tool_calls = [obj for obj in response.output \
                      if getattr(obj, "type", None) == "function_call"]

        if not tool_calls:
            break  # exit loop when there are no tool calls

        for tool_call in tool_calls:
            function_name = tool_call.name
            args = json.loads(tool_call.arguments)

            function = TOOL_FUNCTIONS.get(function_name)
            result = {function_name: function(**args)}

            history += [{"type": "function_call_output",
                         "call_id": tool_call.call_id,
                         "output": json.dumps(result)}]

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

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

    user_input = input("User: ")
```

We've got more code here than usual, but it's surprisingly straightforward. Let's start our analysis with the main conversation loop, which is toward the end of the file.

The main conversation loop serves primarily as our Intake Assistant, which asks the user what kind of podcast they wish to create. Specifically, it wants to learn the user's desired podcast topic and length. Once the Intake LLM knows what the user wants, it writes up these preferences as a short summary, and then calls the initiate_podcast function, passing along the summary to it as an argument. This initiate_podcast function, in turn, kicks off an LLM assembly line that puts a podcast together.

Before moving on to inspect the initiate_podcast function itself, note that the Intake Assistant runs a classic agent loop. This might be more than is needed for an LLM that has only one tool that will likely not be called consecutively. (At least, it shouldn't.) However, we kept the classic agent loop to allow this LLM to handle additional tools should we decide to add them at some point. Also note that this is the *only step in our entire workflow in which an LLM uses a tool.*

The initiate_podcast function lays out the LLM assembly line, working step by step to put together all the pieces of a great podcast. It does this by calling other Python functions in a very precise sequence, using good old-fashioned deterministic code.

This is the key to ensuring that our app follows the intended plan. Although the assembly line incorporates LLMs at various points, the overarching plan is driven by standard Python code—and this code is deterministic. This determinism is exactly what gives agentic workflows their reliability. In general, the more we can take nondeterministic pieces of our app and make them deterministic, the better.

Read through the initiate_podcast function to see just how the assembly line works. The assembly line is a sequence of functions, some of which use LLMs, and some that are just plain Python.

I'd like to highlight some noteworthy points about some of these functions.

First, our code handles with two global variables—research and previous_web_queries—which are initialized as empty arrays toward the top of our code. The research array stores all relevant web text we find through our web searches, and the previous_web_queries array stores each Google search query we use to ensure we never use the same search query twice.

The generate_web_query uses an LLM to generate the ideal web search query based on the podcast topic. I wasn't happy with the queries that our agent was generating in our previous chapter's implementation, so I've used some prompt engineering to help the model craft better queries. Note how we ensure that the model doesn't repeat any previously used web query.

This LLM, and those that follow, are all "specialization LLMs" in that they have one single task. As with a real assembly line, this helps ensure that they do their job well. Of course, we can increase the performance of each specialization LLM with good prompt engineering.

The extract_text function also uses a specialization LLM, which extracts podcast-relevant text from a web page's text.

The has_sufficient_research function likewise has a single-minded focus, which is to determine whether our research contains enough relevant material for the podcast based on the user's desired length. Note that the output of this LLM, as with all standard LLM output text, will be a *string* of "True" or "False", rather than an actual boolean object.

Hopping back to initiate_podcast, you'll see that if the has_sufficient_research LLM determines that we do *not* have enough research, we rerun previous steps to perform further research. This is akin to a physical assembly line, where there may be an inspector at some point who checks the product's quality. If the inspector decides that something is lacking, they'll send the product back to some earlier step in the assembly line.

When enough research is finally accumulated, the write_podcast_script uses another specialization LLM to write a great podcast script. And then, the create_audio function turns the script into a beautifully narrated podcast that you can listen to on the go. After that, the intake LLM takes over again in case the user wants to create another podcast.