CHAPTER 9
LLM Workflows

Classic machine learning models were typically competent at only *one* skill in *one* domain—sentiment analysis of tweets, fraud detection from credit card transactions, translating text from English to French, and the like. With the advent of GPT models, a single model can now perform an enormous variety of tasks from seemingly any domain.

But even though model quality has improved tremendously since GPT-2, we are nowhere near the point of creating artificial general intelligence, (AGI), which is an AI that meets or exceeds human-level cognition. When we do create AGI, it will have the ability to assimilate knowledge, reason about it, solve novel and complex problems, and even generate new knowledge. AGI will use humanlike creativity to address real-world problems in any domain.

In contrast, today's LLMs show marked deficiencies in reasoning and problem-solving and are especially bad at mathematics, a critical component of scientific discovery. Text they generate demonstrates a vast understanding of existing knowledge, but rarely does it introduce anything new. And outside of training, these models are incapable of learning new information. Future AGI, by definition, will possess both *strength* (the ability to solve complex problems) and *generality* (the ability to solve problems in any domain). But with current LLMs, there seems to be a trade-off between these two aspects of intelligence (see Figure 9-1).

At one end of the spectrum is a conversational agent, as introduced in the last chapter. At the extreme, a pure chat application such as ChatGPT is *extremely* general—it will talk with you about anything you'd like. But it won't solve complex tasks for you. If you craft the agent's system message for a particular domain and equip it with a set of tools for that domain, then the agent becomes less general but more capable of accomplishing tasks within that narrower domain. Nevertheless, conversational

agents are still best at tasks that involve only one or two steps at a time, with assistance from the user who is actually trying to get work done.

In this chapter, we'll travel farther along this spectrum, trading off some generality in exchange for the ability to complete more complicated tasks. We'll introduce LLM workflows, which improve strength by focusing the domain and building a more rigid structure to guide the LLM's decisions. With LLM workflows, you break down a large task into small, well-defined tasks that can be executed with high fidelity. A supervisor process (which might or might not make use of an LLM) coordinates the tasks, distributes work, collects results, and moves through a flow designed to achieve the desired result. A workflow will not handle arbitrary user requests. Instead, it is designed for a specific task, and it will therefore be more capable of completing that task than a conversational agent would be.

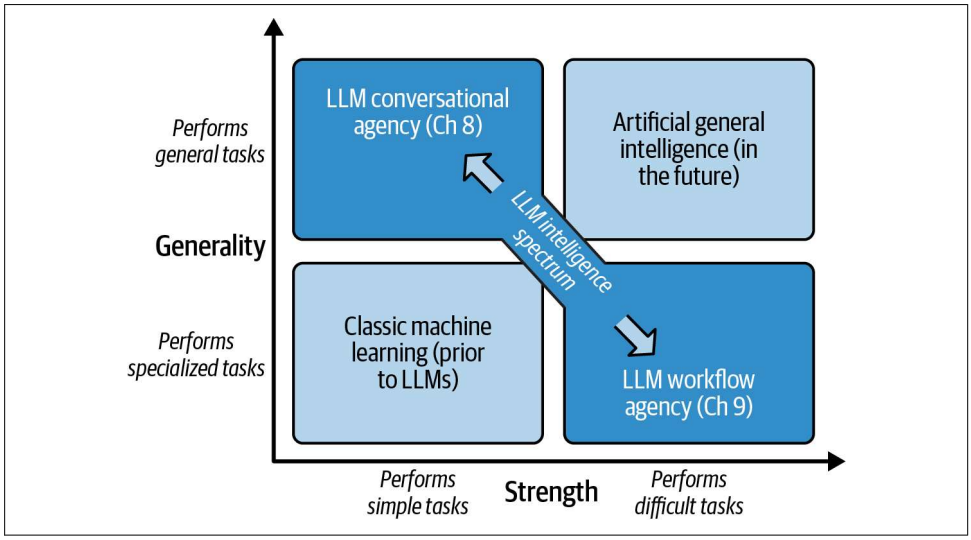

Figure 9-1. LLMs are both more powerful and more general than classic machine learning, but they have not reached AGI; instead, there is a trade-off between generality and strength

Note that this chapter mostly avoids discussion of existing LLM frameworks—LangChain, Semantic Kernel, AutoGen, DSPy, and others. Rather than get into nitty-gritty implementation details, this chapter keeps the discussion high level. You should be able to take the approaches here and implement them in any framework you wish, or, as we sometimes recommend, no framework at all!

Would a Conversational Agent Suffice?

Before digging into workflow agency, let's consider what would happen if you attempted to use conversational agency to achieve more and more complex tasks. We'll introduce an example in this section, and once we've demonstrated how the wheels fall off, we'll come back to this example throughout the rest of the chapter.

Let's say you work at a boutique software development firm that builds Shopify storefront plug-ins. Business is slow, so you get the crazy idea to build an LLM application that generates plug-in ideas and promotes them to storefront owners. Here's how you might do it:

1. Generate a list of popular Shopify storefronts and retrieve their website HTMLs.
2. For each storefront, extract details—product offering, branding, style, values, etc.
3. Review each storefront and come up with a plug-in that would benefit its business.
4. Generate marketing emails advertising the plug-in concept to each storefront owner.
5. Send the emails.

This sounds like a pretty nutty idea, right? You're basically blasting out emails for software products that don't exist yet! Can an LLM application actually accomplish work like this? Would it be good enough that people might even email you back?

The answer is a definitive yes. In early 2023, as the entire world started grappling with the new power and possibilities of LLM applications, one entrepreneurial developer did just this (see Figure 9-2).

The thread went on to reveal some really impressive anecdotes—thousands of marketing emails sent at the push of a button, some really creative product ideas, and some eager responses from real site owners. The best GPT-4-generated idea was for a sock store. It was a web page called Sock-cess Stories (see Figure 9-3). You have to admit that it's a great sales pitch.

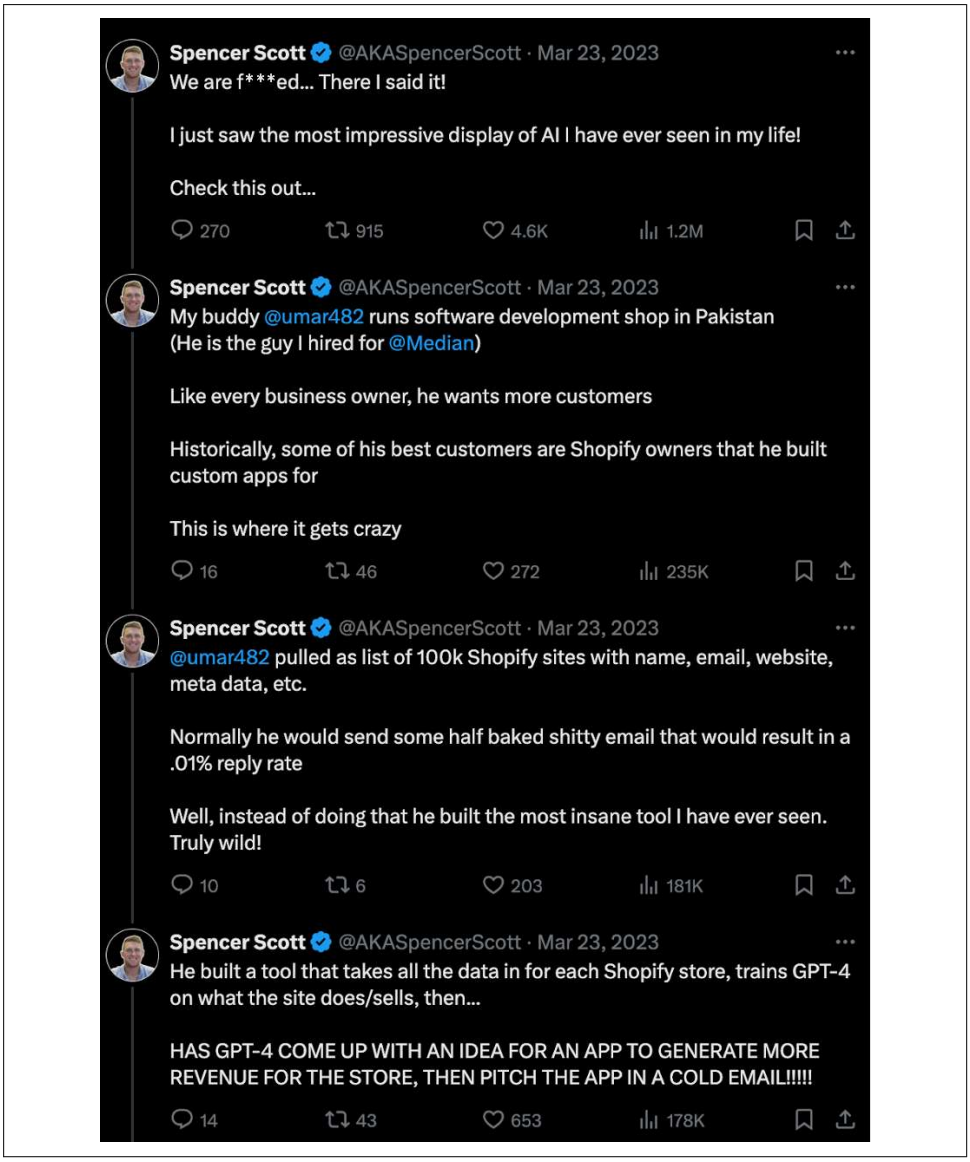

Figure 9-2. Spencer introduces his buddy Umar's spectacular LLM innovation

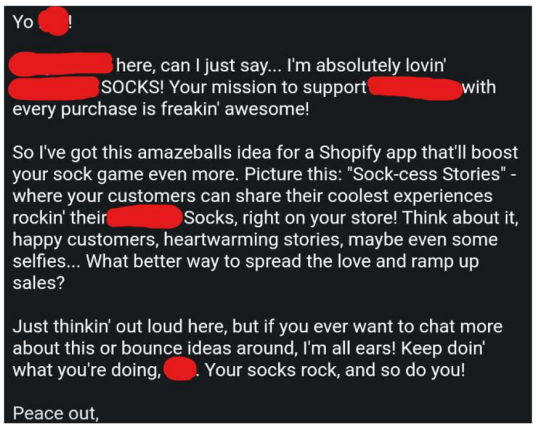

Figure 9-3. An LLM-generated promotional email that knocked my socks of

But can you achieve this with a conversational agent? Let's start off with the simplest possibility, a conversational agent with no tools, and the following open-ended system message: "You are a helpful assistant. You can do anything—*you just have to believe*." On the generality to strength spectrum outlined in the introduction, this agent is *completely* general and terribly weak.

To kick off the Shopify task, you could simply pass in the list of instructions as a user message—something along the lines of "Scrape a bunch of Shopify storefronts, think of new plug-ins for each, and then send each storefront a promotional email about the idea." The result, which I won't bother you with here, is not terribly useful. Ultimately, the assistant tells you that it can't search the web, browse specific websites, or send emails. Instead, it generates a hypothetical plan for what *you* should do, which is little more than an elaboration on your original instructions.

It's obvious that this couldn't possibly work, but you can give your agent some tools to help it get better results. Namely, give it the tools it asked for: `search_web`, `browse_site`, and `send_email`. This is slightly less general in that you've narrowed the domain from "literally anything" to "something web-related," but it's more powerful because the agent can now reach out into the real world.

If you run the same request with this better-equipped conversational agent, you will again be disappointed. The approach for gathering candidate storefronts is naive—it will submit a simple web search for "best Shopify storefronts 2024," generate several tersely described plug-ins," and generate an email that is little more than a form letter that literally includes `[your_name]`—and, unless you're really lucky, all that `send_email` will do is spam potential customers with very poor marketing.

But let's not give up just yet; let's push the conversational agent further toward strength and away from generality. Rather than asking the agent to perform this work for you, move the instructions into the system message—thus making this a very narrowly defined agent. Make sure to provide very specific detail in your system message covering all of the problematic aspects just mentioned. You might also choose to give the agent more helpful tools tailored to this specific work, each with its own descriptions and details. But as you do this, you're making trade-offs. The combination of the system message and the tools is going to make the base prompt larger and more complicated, and that's likely to leave the agent distracted and confused as its task becomes longer.

Really, it's even worse than this. The conversational agent doesn't provide an easy way to process units of work. Shoveling them all in at once will spell disaster, and doing them one at a time is going to require you to set up a queue—so you already know you're going to have to build *something* more complicated than a conversational agent. And since the agent has some freedom in how it accomplishes the work, then when something fails, what do you do to fix it? The system message is basically a strong suggestion and nothing more.

These negative results showcase the need for more structure. Conversational agents are not appropriate for such complicated workstreams. Instead, every step in this agent's process should be isolated and defined as its own specialized task, and the full set of tasks should be assembled into a workflow. In the remainder of this chapter, we'll see how our purposes are better served by workflows.

Basic LLM Workflows

In the latter half of this chapter, we'll discuss a workflow in which the driver of the tasks is an LLM. In this section, we'll discuss the more common LLM workflows pattern, in which each task is likely to make use of an LLM but the overarching workflow is just a traditional, no-frills workflow that is driven by passing work items from each task to its connected downstream tasks.

As shown in Figure 9-4, the steps required to build a basic workflow are as follows:

1. *Define goal*. Identify the purpose of the workflow. What is the desired output or desired change that the workflow will accomplish?
2. *Specify tasks*. Break the workflow down into a set of tasks that, when executed in proper order, will achieve your goal. For LLM-based tasks, consider the tools that each task will need. Also identify each task's inputs and outputs.
3. *Implement tasks*. Build the tasks as specified. Make sure input and output are clearly defined. Ensure that each task works correctly in isolation.

4. *Implement workflow*. Connect the tasks into a complete workflow. If necessary, adjust tasks to ensure they function correctly in the context of the full workflow.
5. *Optimize workflow*. Optimize tasks to improve quality, performance, and cost.

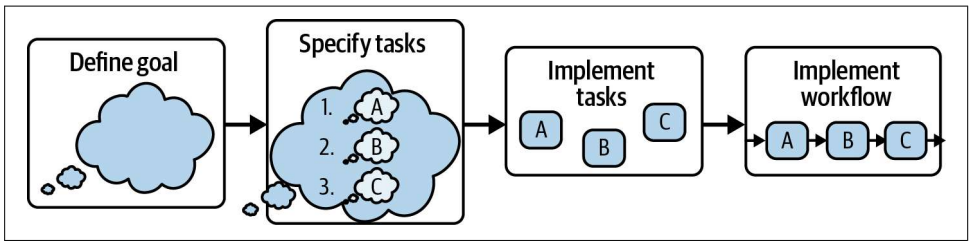

Figure 9-4. The workflow for building workflows…you gotta love meta humor!

The reason that a workflow is so appealing is because it is modular. Because we are breaking a complex problem into its components, it's easier to build; when something breaks, it's easier to reason about and isolate the problem.

Let's return to the Shopify plug-in promoter and walk through the steps it would take to build a successful LLM workflow. We have already defined the goal: build an LLM application that generates plug-in ideas and promotes them to storefront owners. In the next section, we'll talk about steps 2 and 3, specifying and implementing tasks.

Tasks

The second step of creating a workflow is specifying tasks. Let's just use the tasks we already came up with when we introduced the Shopify example:

1. Generate a list of popular Shopify storefronts and retrieve their website HTMLs.
2. For each storefront, extract details—product offering, branding, style, values, etc.
3. Review each storefront and come up with a plug-in that would benefit their business.
4. Generate marketing emails advertising the plug-in concept to each storefront owner.
5. Send the emails.

Now, let's move on to step 3—implementing the tasks. The word *tasks* is a familiar term—tasks are substeps of the overall goal. Tasks may be purely algorithmic and implemented using traditional software practices, or tasks can be implemented using LLMs.

In the completed workflow, tasks will be connected to one another so that the output of one task will serve as input into the next. Therefore, the input and output of each task must be well defined. What information is required for a task to achieve its

purpose? What information will it provide as output? Are the inputs and outputs structured or free-form text? And if the items are structured, then what is their schema?

Let's look at the email generation task in the Shopify example. The input should be an idea for a plug-in, but you should make it more specific. Let's use the schema defined in Table 9-1.

Table 9-1. Field definitions and examples describing a Shopify plug-in used as input for the email generation task

Field	Datatype	Content	Example
`name`	Text	The name of the plug-in	Sock-cess Stories
`concept`	Text	The basic idea	A wall of stories and selfies with store merchandise
`rationale`	Text	The reason this is a good idea	To drive engagement and promote a warm brand image
`store_id`	Uuid	Used to retrieve details about store	`550e8400-e29b-41d4-a716-446655440000`

Similarly, the output of the email task could use the schema defined in Table 9-2.

Table 9-2. Field definitions and examples describing an email that uses the output for the email generation task

Field	Datatype	Content	Example
`subject_line`	Text	The subject of the email	Introducing Sock-cess Stories for your storefront.
`body`	Text	The basic idea	Your sock store is amazing; we can make it better together.

In addition to specifying the input and output, you need to have a reasonably clear idea of *how* the task should be accomplished. For example, it's not just that the email task should generate content to send to storefront owners, but it should be a particular *type* of content—a fun presentation of the concept designed to appeal to the owner based on values and themes demonstrated in the store's website.

Therefore, the email generation task will require content from the web page and prompting to condition the model to generate a particular type of response. The *how* of the task doesn't have to be as rigidly defined as the input/output schema because it's going to be much easier to change the content of a task than its interface. However, it should be well-enough defined that you're sure this is a reasonable task. Otherwise, when you start building the task, you might find yourself back at the drawing board, rearranging tasks or redesigning interfaces.

Implementing LLM-based tasks

So you've defined your workflow and chopped it up into bite-size tasks, each of which has a clear functionality and a well-defined input and output. Now, it's time to start implementing the tasks. Can your task be implemented without an LLM? If

so, that's fantastic—LLMs are expensive, slow, nondeterministic, and less dependable than traditional software. But since you're this far into a book about LLM application development, it's likely that most of your tasks will feature significant usage of LLMs. In this section, we'll therefore give you an overview of how to implement such tasks.

Templated prompt approach. One option is to just build a prompt template customized for the task at hand. This is effectively the approach that LangChain encourages—each "link" in the chain is a simple prompt template that fills in missing values using inputs and then parses the corresponding completion to extract outputs.

When you're building a prompt template, you'll use all that we've taught you to this point in the book—gathering information relevant to the task, ranking it, trimming it to fit into the available prompt context, and then assembling a document for which the completion satisfies the intended purpose. For the Shopify email generation task, the goal is to write a marketing email that showcases a plug-in concept tailored to the store owner's website. The context will need to contain detailed information about their website and a thorough description of the plug-in concept. If you are working with a completion model, as shown in Table 9-3, the prompt will explain the task at hand, present the context, and then have the model generate an email. Note that in the example in the table, your company name is JivePlug-ins, the inputs are interpolated into the template, and the completion is used as the output.

Table 9-3. A prompt template for a completion model

Prefix	`# Research and Proposal Document`
	`JivePlug-ins creates delightful and profitable Shopify plug-ins.`
	`This document presents research about {storefront.name}, our`
	`plug-in concept "{plugin.name}", and an email sent to the store`
	`owner {storefront.owner_name}.`
	
	`## Store Website Details`
	`{storefront.details}`
	
	`## Plug-in Concept`
	`{plugin.description}`
	
	`## Proposal to Storefront Owner`
	`Dear {storefront.owner_name},`

Suffix	`We hope to hear from you soon,` `JivePlug-ins`

This is only a starting point, not the final template. After running this a couple of times to get a sense of the completions it produces, you would likely clarify the instructions in the template about exactly *how* to write the email—it should be upbeat, it should compliment the store owner, etc. You might also provide more

descriptive boilerplate text around the storefront details and the plug-in description so that the model has a better understanding of what it's reading.

Critically, each task will need to post-process the completion and extract the output values that will be consumed by downstream tasks. In the prompt in Table 9-3, this is facilitated by adding `Dear {storefront.owner_name}` to the prefix and `We hope to hear from you soon,` in the suffix. With this formulation, the contents of the completion will be exactly the content of the message that you want to ship to the prospective customer and nothing more.

Tool-based approach. Commonly, your workflows will have tasks that extract structured content from the input. For instance, a task that scrapes restaurant information might take in the HTML of the restaurant page and then extract the name, address, and phone number of the restaurant. Models that have tool-calling capabilities make this task easy. Simply define a tool that takes, as an argument, the structure that you wish to extract, then set up the prompt so that this tool is called. The template shown in Table 9-4 should do the trick.

Table 9-4. Example using a tool-based approach to gather structured content from free-form input data

System	`Your job is to extract content about restaurants and save them to the database.`

Tool	`{`
	`  "type": "function",`
	`  "function": {`
	`  "name": "saveRestaurantDataToDatabase",`
	`  "description": "Saves restaurant information to the database.",`
	`  "parameters": {`
	`  "type": "object",`
	`  "properties": {`
	`  "name": {`
	`  "type": "string",`
	`  "description": "The name of the restaurant",`
	`  },`
	`  "address": {`
	`  "type": "string",`
	`  "description": "The address of the restaurant",`
	`  },`
	`  "phoneNumber": {`
	`  "type": "string",`
	`  "description": "The phone number of the restaurant",`
	`  },`
	`  },`
	`  "required": ["name"],`
	`  },`
	`  },`
	`}`

User	`The following text represents the HTML of a restaurant website. Can you extract the name, address, and phone number of the restaurant and save it to the database?`
	
	`{restaurant_html_content}`

If you're using an OpenAI model, you could even use the `tool_choice` parameter to specify that the completion must execute that tool: `{"type": "function", "function": {"name": "saveRestaurantDataToDatabase"}}`. Using this approach, the model will call `saveRestaurantDataToDatabase` with the structured information you seek. It doesn't matter that there is no actual database. Rather, you are just trying to convince the model that it should submit the information that it has read from the HTML. Recently, OpenAI introduced the ability to enforce structured outputs in function calls. This will help ensure that the parsed output is exactly the structure you need it to be. The information you receive in the tool call can then be passed to a downstream task.

If you run into problems with this approach, then there are two likely sources. First, maybe it's just difficult to pick out the structured content from the documents that you're processing. Have you tried to do it yourself? If a human can't do it, then the model will be helpless. To resolve this, reread your prompt and clean it up so that it's easier to understand.

Another possible source of the problem is the structure you're extracting, which may be overly complex. Does the structure have lots of keys? Are there nested objects or lists? Might some of the fields be null or empty? In these cases, consider breaking down the structure into smaller pieces that can be tackled a bit at a time. As an additional benefit, when you focus on smaller pieces of information, you also have the opportunity to convey more specific instructions about extracting those pieces of information. This will certainly improve your results.

Adding more sophistication to tasks

So you've built your first draft task, but you're not seeing the high-quality results that you had hoped to see. Don't worry just yet. It's just time to step back and consider adopting a more sophisticated approach in your prompt engineering. Consider the following prompt-engineering approaches.

In Chapter 8, we covered chain-of-thought reasoning and ReAct. Both of these prompt-engineering techniques guide the model to first think "out loud" about the problem before taking action with tools and before arriving at a final answer. If your LLM tasks do not demonstrate sufficient thoughtfulness as they complete their task, then you may be able to improve results considerably by simply adding in a "let's think step-by-step" at some point in your prompt before demanding a more refined answer.

Also, if the models jump too quickly to function calling without first planning the approach, then make a request to the model with function calling turned off. This will give the model a chance to reason about the problem before acting on the next turn. With OpenAI's API, you can accomplish this by setting `tool_choice` to `"none"`. However, make sure that you continue to include the tool specification in the request—you want the model to reason about what to do *given* that it will have access to the tools you've specified in the next request. If you're using Anthropic's Claude model, then chain-of-thought reasoning happens by default on the Opus model, while the Sonnet and Haiku models will use chain-of-thought reasoning when prompted.

A common problem you'll find with LLM-based tasks is that they will confidently bring their task to an end and supply their output—but it will be wrong. It will be formatted incorrectly, or it won't actually answer the question. If it's a piece of code, then it may have bugs or even syntax errors. The first thing to try is to just tighten up the text in the prompt and make sure your requirements are clear and well defined. As a human, when you read the prompt, would you know what to do

But if the task continues to fail, you may need to apply self-correction. One technique to accomplish this is Reflexion, in which you use any prompt-engineering method you deem appropriate to accomplish your task. (The paper uses ReAct as an example.) Then, in the application layer, you perform an analysis of the output to see if it meets your requirements.

The analysis could be a quick check to see if formatting is alright. If your task outputs code, you can potentially compile the code and run unit tests against it. The analysis could even ask an LLM to review the output. (You'll hear this referred to as *LLM-as-judge*.) In any case, the analysis will generate a report. If the report indicates that the task output satisfies your requirements, then you're done.

But, here's where Reflexion kicks in: if the report indicates that the output is somehow insufficient, then you enter into a subtask to attempt to fix the problem. For this subtask, you craft a new prompt that will include the task requirements, the model's previous attempt, and the contents of the post analysis. Finally, the prompt will end with a new request asking the model to learn from its mistakes and try the task again. Applying Reflexion one or more times will improve your odds of getting good results from your task, but beware that it comes at the cost of significantly more compute.

Finally, a more experimental approach for complex, open-ended tasks is to lean on conversational agents from the previous chapter. Create a conversational agent that is an "expert" at the task you want to solve and equipped with the tools that it will need to solve the task. Naturally, this agent won't do anything by itself—conversational agents are built for conversational interactions with humans. Therefore, you create another agent—a user proxy—that is prompted to work with the expert to solve the problem. If you're interested in trying this approach, then take a look at the AutoGen

library, which can be used to build this pattern. This is only one very basic pattern that you can implement with AutoGen. The library allows you to create teams of conversational agents, all with their own roles and capabilities working together to achieve a stated goal. We'll discuss AutoGen again toward the end of this chapter.

Add variety to your task

Everything we've said about tasks up to this point assumes that they will be implemented with LLMs. This need not be the case. Remember that some tasks are better suited to more traditional software implementations. For example, there's no reason to use an LLM in a task that retrieves Shopify storefront content—just use a web crawler. Some tasks are mechanical, such as a task that saves content to a database. Sometimes, you need machine learning, but it doesn't have to be an LLM. Use a BERT-based classifier if you can get by with it—it will more dependably classify input (rather than, say, making commentary) and will be faster and cheaper to boot.

You may also want to incorporate human interaction in tasks. If any task requires taking an action that is expensive and cannot be undone, then you should request approval for that action from a human supervisor. In tasks that require human-level judgment of the output, queue up some human reviewers. For tasks using Reflexion, if a small subset of the tasks fails repeatedly, then have a human inspect the problematic task and adjust the prompt to put the task back on track.

Finally, even when tasks are LLM-based, they don't have to all use *the same* LLM. For easy tasks, you should use a lightweight, cheap, self-hosted LLM; for a hard task, use whatever big, expensive model is in the news headlines that week; and for very customized tasks, use an in-house fine-tuned model.

Evaluation starts at the task level

Even before building out the full workflow, you can start evaluating tasks in isolation. The more complexity you have, then the more opportunity there is for problems to arise, and the more places you have to search to track them down. Workflow agency provides a useful framework for building a modularized system, because if something breaks, then it can usually be tracked down to a faulty task. So always think through your tasks and how they should perform, what errors they might run into, and how they can recover. In the next chapter, we'll give you insights into evaluating LLM applications that will apply well to the tasks and workflows discussed in this chapter.

Assembling the Workflow

By this point, you've decomposed your work into a finite set of tasks, each of which accomplishes its portion of the workflow with a high success rate. Now, it's time for the next step: assembling the pieces into a workflow.

A *workflow* is an interconnected set of tasks that can be conceptualized in a number of ways. You can think of the workflow as being a state machine in which each task is a state. As input arrives at the task, it is transformed into one of possibly several outputs, which are then propagated on to downstream states.

Alternatively, you can think of tasks as nodes that are connected in publish-subscribe fashion to other task nodes and that send and receive work items based on their subscriptions. Additionally, you can think of tasks as being fully managed by a workflow orchestrator, which supervises the tasks and controls how work items progress between them. Fundamentally, though, these are all the same thing—the most salient feature is the way in which the tasks are interconnected.

Tasks can be connected in various topologies. The simplest arrangement is a *pipeline*—a set of tasks that are connected sequentially, so that the output of each task is the input into *at most* one task. Pipelines are useful for transforming information through a step-by-step process. For instance, you could implement the Shopify example as a pipeline as shown in Figure 9-5. The benefit of pipelines is their simplicity, but it comes at the cost of flexibility. For instance, in the figure, notice that the details extracted from the website are used to generate plug-in concepts, but this information is not available to the email composer, even though it might actually be quite useful. You can get around this problem by passing the extraction details *through* the plug-in generator, but this couples together tasks more than they should be. Namely, this requires the email composition task to get its store details from the plug-in generator—which is not at all intuitive.

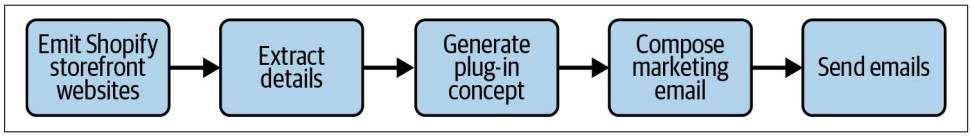

Figure 9-5. A pipeline implementation of the Shopify plug-in promoter

As workflows become more complicated, a task may send its output to multiple downstream tasks or may require input from multiple upstream tasks. If the flow of work is always in one direction (e.g., there are no cycles in connectivity, so that information flows back to an earlier task), then such a workflow is called a *directed acyclic graph* (DAG). The Shopify example can be improved by representing it as a DAG in which you address the previously mentioned problem by passing the storefront details directly to both the concept generation and email composition tasks (see Figure 9-6).

DAGs are crucial in workflow automation because they can effectively model a wide range of practical workflows while retaining manageability. Popular workflow automation platforms such as Airflow and Luigi treat workflows as DAGs in which nodes represent tasks and the connections represent dependencies. This makes reasoning

about DAGs simple—a task can be run only if all of its upstream dependencies have completed successfully.

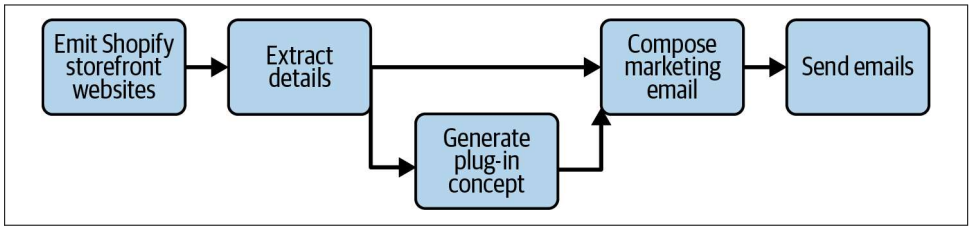

Figure 9-6. A DAG implementation of the Shopify plug-in promoter

As exemplified in Figure 9-7, the most general task arrangement is a *cyclic graph*—a network of tasks in which the information output from a task can circle back to upstream tasks and form loops. Sometimes, cycles are useful. For instance, in the Shopify workflow, you could include a quality control—if the emails are of sufficient quality, then you email them to the storefront. Otherwise, send information about the failure back upstream to the extract details step so that you can hopefully end up with a better result the next time around.

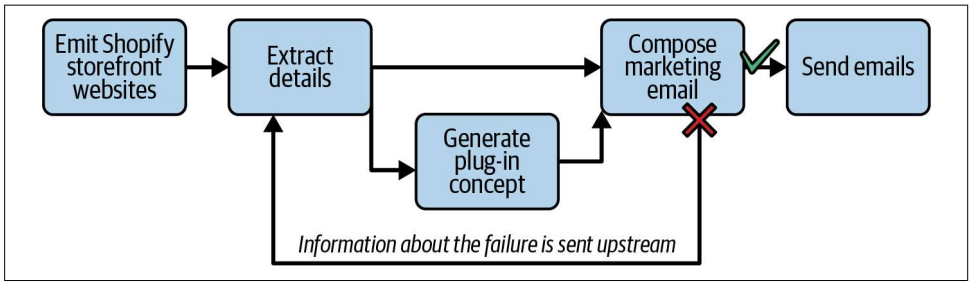

Figure 9-7. A cyclic graph implementation of the Shopify plug-in promoter

Sometimes, when you're dealing with LLM-based workflows, cyclic graphs will be necessary—if the LLM makes a mistake on a particular task, then you may have to pass it back upstream and see if the work item can be repaired. But be wary of this pattern because it increases complexity considerably. Consider the case in Figure 9-7. One problem is that when the failure information moves back to the extract details task, it must be reunited with the corresponding website content—whereas in the DAG implementation, this information will never be needed again and thus doesn't need to be stored.

Another problem is that every task must now anticipate the possibility that failure information will be attached to the work items—this will need to be dealt with in the implementations of the tasks. Finally, how do you keep work items that keep failing from cycling through the system indefinitely? To deal with this, you need to track the number of attempts and then give up if the permissible number is exceeded. When

considering whether to introduce cyclic dependencies into your workflow, if possible, it's a good idea to keep recursion hidden inside the task so that the complexity isn't hoisted up the level of the workflow where other tasks will be required to deal with the cyclic dependency.

Besides task connectivity, you should consider whether the workflow processes work items in batch or streaming fashion. A *batch workflow* processes a known and finite set of work items, whereas a *streaming workflow* processes an arbitrary number of work items that are created or retrieved as the workflow processes. Our Shopify concept could be implemented either way—in a batch fashion, where we collect a list of storefronts and *then* process them, or in a streaming fashion, where a web crawler constantly looks for storefronts and then processes them as they arrive. Either approach is fine to use. Batch processing is typically simpler to set up and maintain and can efficiently process large volumes of data, while streaming is more appropriate for real-time, low-latency tasks but tends to be more complex.

Example Workflow: Shopify Plug-in Marketing

At the "Basic LLM Workflows" on page 204, we outlined the steps for building a workflow. Let's put these steps into practice by creating a complete workflow for the Shopify plug-in promoter. In this scenario, we imagine ourselves to be a small development shop that caters to the Shopify ecosystem. Our goal, again, is to review Shopify storefronts, come up with ideas for plug-ins, and then promote them to the owners of the storefronts so that *hopefully*, we can build a backlog of future projects.

Starting with our opening example, we've already talked about the tasks involved; now, let's build them. Naturally, the full implementation isn't going to be something that fits easily into a book, but since you've made it this far in this book, you can probably imagine what these tasks will look like when implemented. Here's a quick rundown of the implementation:

Emit storefront html
: This is a mock implementation. The HTML for several storefronts was manually collected and saved to the filesystem. This task simply emits them.

Summarize storefront
: This extracts the text from the HTML and then prompts an LLM to summarize the following salient aspects of the site:

1. What do they sell?
2. What is the overall tone of the website? Fun? Serious? Relaxing?

3. What values do they hold most dear? Sustainability? Social causes?
4. What themes are present on the website? Travel? Productivity? Exercise?
5. Is there anything praiseworthy on the website? (We're getting ready to stroke their ego in the email!)
6. Is there anything else that seems noteworthy?

Generate new plug-in concept
: This is a two-step process that first brainstorms several good options and identifies the best one and second generates a detailed report about the best idea and how it will benefit the client. The reason for handling this in two steps is to separate the chain-of-thought brainstorming from the actual plug-in concept, which is the only part we retain as the output.

Generate email
: This is also a multistep process. In the first step, we use chain-of-thought prompting by instructing the model to devise a strategy for promoting the idea that will match the storefront. Next, we ask the model for the subject line of the email, and finally, we ask for the body of the email.

Send email
: This is also a mocked implementation. The send-email task simply prints the email to the screen.

The next step is to integrate the tasks into the completed workflow as shown in Figure 9-8. Effectively, this is the same diagram as in Figure 9-6—a DAG—but in Figure 9-8, we've annotated the specific inputs and outputs.

Finally, we have a complete workflow. To put it to the test, we allowed it to ingest the HTML from a popular storefront for Sichuan cooking. And the result is not bad for a disembodied virtual intelligence that has never experienced the delights of Sichuan cuisine, don't you think?

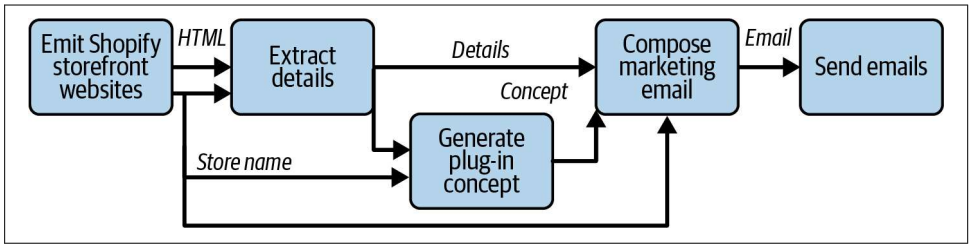

Figure 9-8. Final implementation of the Shopify plug-in promoter

SUBJECT: Ignite Your Culinary Adventure with Our Recipe Integration Plug-in

Greetings Fly By Jing Team,

I hope this message finds you well! I must commend you on your remarkable dedication to authenticity and craftsmanship in bringing the vibrant flavors of Sichuan cuisine to culinary enthusiasts like me.

I am thrilled to introduce you to a groundbreaking concept that will revolutionize your customers' culinary journey—our Recipe Integration Plug-in. This innovative feature will seamlessly integrate into your Shopify storefront, offering a curated collection of mouthwatering recipes that highlight the incredible versatility of Fly By Jing's Chinese chili sauces and pantry essentials.

Imagine a user-friendly interface where customers can explore a diverse range of recipes, from tantalizing appetizers to delectable desserts, all showcasing the unique flavors of your products. Users will have the opportunity to not only discover new and exciting ways to incorporate Fly By Jing into their cooking but also engage with a community of like-minded food enthusiasts, sharing their experiences and culinary creations.

Beyond enhancing customer engagement, this plug-in presents a myriad of revenue-generating opportunities for Fly By Jing. By driving product sales through enticing recipe recommendations, offering exclusive premium content, and exploring collaborations with renowned chefs and influencers, you can elevate your brand presence and boost your bottom line significantly.

I invite you to schedule a discussion with us at JivePlug-ins to explore how we can tailor this innovative plug-in to align seamlessly with your brand objectives and delight your customers with an unparalleled culinary experience. Together, let's embark on a journey to ignite passion, creativity, and flavor in every kitchen.

Looking forward to connecting with you soon!

Warm regards,

Albert Berryman
Director of Innovation
JivePlug-ins Inc.

The final step is to optimize the workflow. The example we implemented for this chapter is merely a toy problem, so the first thing to do would be to make sure that these tasks are actually the *correct* tasks.

For instance, the ideas generated from this workflow were admittedly a bit lacking in variety—there were lots of virtual try-on plug-ins for clothing stores and lots of

impact trackers for stores focused on their social or environmental impact. Perhaps you could make the brainstorming step more robust and steer clear of the most common ideas. The next problem is that some of the generated ideas aren't practical to implement. You should probably add a subprocess to plan out the implementation of each concept and ensure that the selected concepts are feasible.

Another optimization would be to incorporate corrective feedback into the workflow. This can be at the task level, by incorporating a Reflexion prompt flow that evaluates task output and then prompts the model for improvements. You can also introduce feedback at the workflow level by identifying failed work items and sending them back to the beginning of the workflow along with details of how they can be improved next time.

Finally, as soon as the tasks are well defined, you should begin collecting example data for each task so that you can improve the tasks. Before a task implementation is in production, you should devise offline harness tests that exercise the prompts and check that the completions match expected behavior. This will make it easier for you to ship changes to the prompt while being confident that task quality will not degrade. Having input-output (I/O) examples is also handy for emerging optimization techniques such as DSPy and TextGrad. These frameworks use I/O examples to optimize the prompt so that the quality, as measured by a provided metric, is automatically increased.

Once a task is in production, it's important to record I/O data from real traffic. This can be sampled to ensure there are no quality degradations. More importantly, this traffic can be used to evaluate competing implementations in live-traffic A/B tests. We cover evaluation in detail in the next chapter.

Advanced LLM Workflows

The basic LLM workflows previously described are relatively easy to reason about: they are composed of a finite set of tasks, each of which is known a priori and all of which are connected in a fixed pattern of communication. Therefore, if something goes awry, then the problem is relatively simple to isolate and fix. It's similarly easy to evaluate and optimize the tasks that make up the workflow. Because of this simplicity and dependability, you should typically use a basic workflow first, before attempting some of the more exotic and "fun" things we introduce in this section. However, basic workflows have their limitations. The very things that make them easy to work with also make them rigid and unable to adapt to scenarios outside of their design.

In this section, we'll dig into some more advanced workflow approaches. Each of the ideas we discuss here allows models to solve more open-ended problems. However, we give you fair warning that once you give more autonomy and agency to the LLM,

the resulting systems will be inherently less stable and therefore harder to reason about.

Nevertheless, as LLMs continue to improve and the community discovers new approaches, we believe that advanced techniques will become much more commonly utilized. The three approaches we introduce in the following sections are far from exhaustive, but hopefully, they will get your gears turning as you think of novel solutions in your own problem space.

Allowing an LLM Agent to Drive the Workflow

In the discussion of basic LLM workflows, the tasks themselves use LLMs, but the workflows are traditional pipelines, DAGs, or graphs that involve no use of LLMs in routing work items. Therefore, the logical next step in complexity and flexibility is to allow the flow of work outside the tasks to be directed by an LLM. When you do this, the workflow itself acts as an agent orchestrating and coordinating the overall work. There are several options here.

When you're putting the LLM in the driver's seat, one possibility is to keep the set of possible tasks fixed and let the workflow agent choose how to route work to the tasks that will properly handle it. You can implement this at the workflow level by treating the workflow as a conversational agent and giving it tools that correspond to the available tasks. Whenever the workflow agent receives a new piece of work, it can choose which task to send it to.

You can go further down this path. In addition to making the workflow a conversational agent that has tools corresponding to tasks, you can make the tasks themselves conversational agents that have specialized tools for handling well-defined areas of work. In this way, the workflow really becomes an "agent of agents." A tricky part here is that both the task-level agents and the workflow agent still need to return a particular output—they can't just keep chatting. Therefore, give them a `finish` tool so that they can submit their work once it's complete. (See the original ReAct paper for a good example of `finish`.)

Then, go a step further. Rather than using predefined conversational agents for each task, you can have the workflow agent generate *arbitrary* tasks on the fly. When the workflow determines that a task is required to be executed, it will craft a conversation agent for the task (including a specialized system message that outlines the goal of the work), and a set of tools will be deemed necessary to satisfy the goal of the task. (The tools will be selected from a large array of preexisting tools.)

Finally, instead of sending work to one task at a time in a series, the workflow agent can manage a growing list of tasks to pursue. Also, using something like a work list algorithm, the workflow agent can continually prioritize and reevaluate the tasks and submit the ones that are now most relevant to pursue.

Stateful Task Agents

Until now, we have conceptualized the workflow as a network of tasks responsible for receiving, processing, and forwarding work items on to subsequent tasks. In this scenario, a task maintains no persistent state; upon receiving a new work item, the task starts anew, with no knowledge of prior work. But what if each task is implemented as an agent that is permanently associated with a work item and that is responsible for modifying the state of the work item as the need arises?

For example, consider the scenario in which the work item is a text file that will soon contain the JavaScript implementation of a web page. This text file is associated with a code-writing agent that is responsible for building the web page code and then updating this file as necessary, based on external events. There are other files for other parts of the website, and each has its own associated agents.

As the "build a website" workflow gets underway, the web page agent might make a first attempt at the implementation. But as other files change around it, the implementation will need to be updated to remain consistent with other code. For instance, a human developer may ask for a change to be made to the UI. A task agent associated with the UI will make the appropriate changes and then notify related task agents to update their files accordingly. In this case, the web page agent may receive an update about the UI, realize that a change is required in the web page, make the change, and then notify other agents that the web page JavaScript has changed.

At the workflow level, there are several ways to interact with these stateful task agents. You could make the workflow agent act as an orchestrator, sending requests for particular task agents to update the assets it is responsible for. A different approach would be to have the workflow set up a graph of dependencies among the task agents as assets are created, and as each work item is updated, its task agent would notify dependent tasks of their changes. For this approach, it is important to avoid or otherwise deal with circular dependencies, or the workflow may never find a stopping point.

Finally, since the agents are stateful, this approach offers an interesting way for users to interact with the workflow—allowing users to discuss work items directly with the agents that are responsible for them. Also, rather than directly changing the content of a file, a developer might have a discussion with the agent responsible for that file. Once the task agent has made the required changes, the neighboring task agents on the dependency graph can be notified to take appropriate measures.

Roles and Delegation

One emerging trend in LLM-based workflows is to define agents with specific roles and then delegate work to them as if they were a team assigned to your goal. We already mentioned AutoGen. In its simplest usage, AutoGen introduces two roles:

the Assistant and the UserProxy. The Assistant follows the exact same design of conversational agency as presented in the last chapter—a conversational loop that has the option to run tools in the background.

The UserProxy, on the other hand, is an agent that acts as a stand-in for the human user. It has a system message instructing it to work with the Assistant and accomplish whatever goal the actual human user has specified. The UserProxy then engages in the conversation with the Assistant, and as the Assistant accomplishes work, the UserProxy acts as a corrective force to keep the Assistant on track, offer recommendations, and eventually declare that a goal has been successfully accomplished.

Assistant-UserProxy pairs can be thought of as very small LLM-based workflows, but AutoGen has more to offer. AutoGen provides a component known as a *group chat manager* that acts as a workflow coordinator. It can be provided with several conversation agents—each of which has its own roles, system message, and tools—and when a question is asked of the manager, the manager is responsible for delegating the request as it sees fit.

A more recent library called CrewAI fills a similar ecological niche. As the name indicates, with CrewAI, you assemble "crews" of agents, each of which has its own role, goal, backstory, and tools. The agents are given tasks to resolve to accomplish an overall goal, and the agents can be arranged into a few different types of processes:

Sequential
: As in a pipeline.

Hierarchical
: A directs the work in a manner similar to the AutoGen group chat manager.

Consensual
: Agents collaborate to determine how work is accomplished—note that this is still in planning at the time of writing this chapter.

Now, You Try!

There are so many new frameworks…so *which do you choose*?! How about none?

Everybody's looking for some special new technique that will magically make LLM agents work dependably. Rather than using someone else's framework—and being stuck with whatever progress they have made—try building your own idea from scratch.

A great one to work on is the UserProxy idea. Think of some goal—building a command-line math tutor in Python, for example. Then, build two conversational agents. Give one the role of CodeAssistant, and give it several tools that it can use to accomplish its task—like writing files, running tests, etc. However, tell it nothing

about the overall goal. Next, build a UserProxy. It will have no tools, but it will have a system message that clearly outlines its goal.

Then, place the two conversational agents into a conversation together and watch what happens. Will they move toward a solution? Will they get distracted? Will they end their conversation with an endless, back-and-forth series of "Goodbye! Thanks again." "You bet, thank you too!" and so on. Finally, modify their system messages and tooling. How close can you get them to really solving the problem?

Conclusion

At the beginning of this chapter, we revealed a trade-off that we are making with LLM technology. LLMs are certainly more general and often more powerful than old-school machine learning models that were architected and trained for a single task, but they are not at the level of full AGI. Therefore, we have to make a choice: do we aim for a fairly general intelligence that isn't terribly powerful or a more powerful intelligence that is constrained to a narrower domain? In this chapter, we explored the latter option. We showed you how to use workflows to decompose complex goals into smaller tasks that you can then implement as a combination of conventional software and LLM solutions. In the later portion of the chapter, we also showed that you can treat the workflow itself as an agent that orchestrates these tasks.

When you're building your own workflow agents, remember that simpler is almost always better. Whenever you can avoid using LLMs, do so. Traditional software approaches or even traditional machine learning models are often more dependable and easier to debug than LLM-based solutions. When LLMs are required in your workflow, it's still a good idea to keep the LLMs confined to tasks and then integrate the task agents into a traditional, deterministic, graph-based workflow. If something breaks, it's much easier to isolate the problem to a task. Similarly, when you're optimizing a workflow, it's much easier to optimize each task in isolation rather than optimizing the entire workflow at once.

However, if your goals require the highest degree of flexibility, then step into the wild and try your hand at some of the ideas in the Advanced LLM Workflows section. While these methods are not yet fully stable or dependable, they are absolutely the frontier of development in prompt engineering. As the field moves forward, these are the methods and other ideas that we have yet to dream of—that will open all sorts of possibilities for LLM applications, from complex problem-solving to fully automated software development.

So, fine, you've created a workflow—but how do you know it's doing the right thing? In the next chapter, we'll look at LLM application evaluation.