In this video we'll learn how to split complex tasks into a series of simpler subtasks by chaining multiple prompts together. You might be wondering why would you split up a task into multiple prompts when you could achieve it with one prompt and chain of thought reasoning like we learned in the previous video. We've already shown that language models are very good at following complex instructions, especially the more advanced models like GPT-4. So let me explain why we would do this with two analogies comparing chain of thought reasoning and chaining multiple prompts. The first analogy to compare the two is the difference between cooking a complex meal in one go versus cooking it in stages. Using one long complicated instruction can be like trying to cook a complex meal all at once where you have to manage multiple ingredients, cooking techniques, and timings simultaneously. It can be challenging to keep track of everything and ensure that each component is cooked perfectly. Chaining prompts on the other hand is like cooking the meal in stages where you focus on one component at a time ensuring that each part is cooked correctly before moving on to the next. This approach breaks down the complexity of the task making it easier to manage and reducing the likelihood of errors. However this approach might be unnecessary and over complicated for a very simple recipe. A slightly better analogy for the same thing is the difference between reading spaghetti code with everything in one long file and a simple modular program. The thing that can make spaghetti code bad and difficult to debug is ambiguity and complex dependencies between different parts of the logic. The same can be true of a complex single step task submitted to a language model. Chaining prompts is a powerful strategy when you have a workflow where you can maintain the state of the system at any given point and take different actions depending on the current state. And so an example of the current state would be after you've classified an incoming customer query the state would be the classification so it's an account question or it's a product question. And then based on the state you might do something different. Each subtask contains only the instructions required for a single state of the task which makes the system easier to manage, makes sure the model has all the information it needs to carry out a task and reduces the likelihood of errors as I mentioned. This approach can also reduce in lower costs since longer prompts with more tokens cost more to run and outlining all steps might be unnecessary in some cases. Another benefit of this approach is that it is also easier to test which steps might be failing more often or to have a human in the loop at a specific step. So to summarize because this was a long explanation, instead of describing a whole complex workflow in dozens of bullet points or several paragraphs in one prompt like in the previous video, it might be better to keep track of the state externally and then inject relevant instructions as needed. And what makes a problem complex? I think in general a problem is complex if there are many different instructions and potentially all of them could apply to any given situation as these are the cases where it could become hard for the model to reason about what to do. And as you build with and interact with these models more you'll gain an intuition for when to use this strategy versus the previous. And one additional benefit that I didn't mention yet is that it also allows the model to use external tools at certain points of the workflow if necessary. For example it might decide to look something up in a product catalog or call an API or search a knowledge base, something that could not be achieved with a single prompt. So with that let's dive into an example. So we're going to use the same example as in the previous video where we want to answer a customer's question about a specific product, but this time we'll use more products and also break the steps down into a number of different prompts. So we'll use the same delimiter that we've been using in the previous videos. And let's read through our system message. "You will be provided with customer service queries. The customer service query will be delimited with four hashtag characters." "Output a Python list of objects where each object has the following format.". Category, which is one of these predefined fields, or products. And this is a list of products that must be found in the allowed products below. "Where the categories and products must be found in the customer service query. If a product is mentioned, it must be associated with the correct category in the allowed products list below. If no products or categories are found, output an empty list.". And so now we have our allowed list of products. So we have the categories and then the products within those categories. And our final instruction is, "Only output the list of objects with nothing else." So next we have our user message. And so this message is, "tell me about the smarts pro phone and the fotosnap camera, the DSLR one.". Also "tell me about your TVs.". So we're asking about two specific products and also this general category of televisions. And both of these products are mentioned in the allowed products list. And then we have a television section as well. Then we format the system message and user message into the messages array. And then we get the completion from the model. So as you can see, for our output, we have a list of objects and each object has category and products. So we have the "SmartX ProPhone" and the "Fotosnap DSLR Camera". And then in the final object, we actually only have a category because we didn't mention any specific TVs. And so the benefit of outputting this structured response is that we can then read it into a list in Python, which is very nice. And so let's try another example. So our second user message is, "my router isn't working". And if you notice in the list, we don't actually have any routers. And then, let's format this correctly and get the completion. And so as you can see, in this case, the output is an empty list. And so now that we have this step to identify the category and products, if we find any products and categories we want to load some information about those requested products and categories into the prompt so that we can better answer the customer question. And so in our workflow the state now after this prompt has run is either products have been listed or they haven't been listed and in that case we wouldn't try to look anything up because there's nothing to look up. Also if I were to actually build this into a system I might use category names, maybe something like computers and laptops or something to avoid any weirdness with spaces and special characters but this should work for now. So now we want to look up some information about the products that the user mentioned, so about this phone, this camera and about the TVs in general. And so we need to have some kind of product catalogue to look up this information from. So here we have our product information that I just pasted in. So as you can see we have a large number of products available at our store and all of these products are fake and were actually generated by GPT-4. And so for each product we have a couple of different fields, we have name, category, brand, warranty and so on. And so the products is just a dictionary from product name to this object that contains the information about the product. Notice that each product has a category. So remember we want to look up information about the products that the user asks about, so we need to define some helper functions to allow us to look up product information by product name. So let's create a function, "get\_product\_by\_name". We input the name and then we're going to return product dictionary and we're going to get the value for the item with the name as the key and then our fallback is just going to be none. And so we also want to define another helper function to get all of the products for a certain category. For example when the user is asking about the TVs we have we'd want to load all of the information about all of the different TVs. So "get\_products\_by\_category" input the category name string. And to do this we want to loop through all of the products in the products dictionary and check each one to see if the category is equal to the input category and if so we want to return that. So we'll do this as follows. So first we want to loop through each product and we have to get the values because we need to actually access the category which is in the value. And then, we'll return this product if the product category is equal to our input category. So let's do an example for each of these helper functions. So first we have a product called the "TechPro Ultrabook". So let's get the product information by name. So here you can see we've just fetched all of the product information. And let's do an example to get all of the products for a category. So let's get all of the products in the computers and laptops category. So here you see we fetched all of the products with this category. So let's continue our example and just to remember where we are let's print the user message. So the user message was tell me about the SmartX ProPhone and the camera and the TVs. And then the initial output from the model from the first step was this. And so what we also need to do is read this output from the model which is a string. We need to pass that into a list so that we can use it as input to the helper functions that we just wrote. So let's write a helper function to do this. So we're going to use the Python JSON module and we're going to write a function called "read\_string \_to\_list", a very descriptive title input string. And so first we'll just check if the input string is none. In case something in a previous step failed we're just going to return nothing. And now we're going to have a try except block to make sure that we catch any errors. And so, first we replace any single quotes with double quotes in the input string to make sure we can pass the JSON, and then we're going to use the JSON loads function to read the input string into the array, or the list, and then we're going to return this, and if there's a decode error, we're going to print the error and then return none. So, let's try this with our example. So, we're going to get our category and product list using the "read\_string\_to\_list" helper function, and apply it to this response from the model, and then we're going to print this list. So, it should look the same, let me run this first. And so, as you can see, it's just the same thing, except now the type of this variable is actually a list instead of a string. So, the whole point of what we're doing is to get the product information into a list that we can add to the next instruction to the model, which is going to be the instruction where we ask it to answer the user question. And so, to do this, we need to put the product information into a nice string format that we can add to the prompt. And so, let's also create a helper function to do this. So, we're going to call it "generate\_output\_string", and it's going to take in the list of data that we just created, so this. And then I'm going to copy in some code, and then we'll walk through what it's doing. So, now I'm going to paste in some code and show you an example, and then we'll talk about what this function is doing. So, we're going to get the product information from our first user message, and so we're going to use this helper function "generate\_output\_string" on our category and product list, which, if we remember, was this. And so, here we have all of the product information for the products that were mentioned in the user message. So, we have the phone that they mentioned, we have the camera that they mentioned, and then we have all of the product information for all of our TVs. And this is information that will be helpful for the model to be able to answer the user's initial question. And if you're interested in how this function works, I'll give a brief overview but you can feel free to pause the video and read it more thoroughly. So it basically just loops through all of the objects in this list and first checks if there are products. If so, it gets the information for each product and then it checks if there's a category, if there weren't any products. So that would be for this object for example and then it gets all of the product information for the products in that category. And it just adds them to this string and then that's what it returns. So at this point we've found the relevant product information to answer the user question. Now it's time for the model to actually answer the question. So let's have our system message. So this is the instruction. You're a customer service assistant for a large electronics store. Respond in a friendly and helpful tone with, let's say, with very concise answers. Make sure to ask the user relevant follow up questions. So we want this to be an interactive experience for the user. And so just as a reminder, this was our initial user message. I'll just add it again. And so now we're going to have our messages array. And this is the input to the model. So let's go through this. We have our first message, which is the system message as usual. We have the user message. And then we have this additional assistant message. And this is the message that contains all of the product information that we just looked up. And so we're saying relevant product information, new line, and then this product information that we just found. And so now the model has the relevant context it needs to be able to answer this user's question. So let's get the final response and print it. And we're hoping that the model is going to use relevant information from the product information in order to answer the user in a helpful way. So first it tells the user about the SmartX ProPhone. It tells the user about the Fotosnap camera and then talks about the different televisions that we have in stock and then asks a follow-up question. So as you can see, by breaking this up into a series of steps, we were able to load information relevant to the user query to give the model the relevant context it needed to answer the question effectively. So you might be wondering why are we selectively loading product descriptions into the prompt instead of including all of them and letting the model use the information it needs. And so what I mean by this is, why didn't we just include all of this product information in the prompt and we wouldn't have to bother with all of those intermediate steps to actually look up the product information. There's a couple of reasons for this. Firstly, including all of the product descriptions might make the context more confusing for the model just as it would for a person trying to process a large amount of information at once. I will say this is a lot less relevant for more advanced models like GPT-4, especially when the context is well structured like it is in this example and the model is smart enough just to ignore the information that clearly isn't relevant. The next reasons are more compelling. So the second reason is that language models have context limitations, i.e. a fixed number of tokens allowed as input and output. So if you have a large number of products, imagine you had a huge product catalog, you wouldn't even be able to fit all of the descriptions into the context window. And the final reason is that including all of the product descriptions could be expensive as you pay per token when using language models. So by selectively loading information, you can reduce the cost of generating responses. In general, determining when to dynamically load information into the model's context and allowing the model to decide when it needs more information is one of the best ways to augment the capabilities of these models. And to reiterate, you should think of a language model as a reasoning agent that requires the necessary context to draw useful conclusions and perform useful tasks. And so in this case, we had to give the model the product information and then it was able to reason about that product information to create a useful answer for the user. And in this example, we only added a call to a specific function or functions to get the product description by product name or to get the category products by category name. But the models are actually good at deciding when to use a variety of different tools and can use them properly with instructions. And this is the idea behind ChatGPT plugins. We tell the model what tools it has access to and what they do, and it chooses to use them when it needs information from a specific source or wants to take some other appropriate action. In our example, we can only look up information by exact product and category name match, but there are also more advanced techniques for information retrieval. One of the most effective ways to retrieve information is using text embeddings. And embeddings can be used to implement efficient knowledge retrieval over a large corpus to find information related to a given query. One of the key advantages of using text embeddings is that they enable fuzzy or semantic search, which allows you to find relevant information without using the exact keywords. So in our example, we wouldn't necessarily need the exact name of the product, but we could do a more a search with a more general query like a mobile phone. We're planning to create a comprehensive course on how to use embeddings for various applications soon, so stay tuned. And with that, let's move on to the next video where we're going to talk about how to evaluate the outputs from the language model.