What are LLMs?

Selecting transcript lines in this section will navigate to timestamp in the video

Let's get started by exploring what large language models are and how they came about. In this video, we'll explore the genesis of these transformative technologies, how they first came about, and how they evolved through groundbreaking innovations as well as through wide-ranging applications. We'll also examine the challenges and limitations inherent in these models, providing you with a comprehensive understanding of both the potential and the pitfalls of LLMs. The journey of LLMs began with a significant breakthrough in AI research. The development of the transformer model introduced in the pivotal paper, Attention is All You Need by Vaswani et al in 2017. This paper marked a massive leap forward from previous sequence learning models like recurrent neural networks or RNNs, and long short-term memory networks, or LSTMs, which process data sequentially, but they often struggled with long-range dependencies in text. What does that mean? Well, to understand text, one often has to understand words that are far away from the desired one. For example, consider this sentence, in Ireland, I attended high school, and I had to learn how to speak < blank >. Now, as a human, you probably would have said Irish, which would be good, or Gaelic which would be better. But how did you get that answer? The clues were in the words. Speak indicates that it's language which is right beside the blank, and Ireland, which is the major clue, indicating which language. But in reality, there could be many answers to the blank. There's no reason why it couldn't be English, for example, or even a word like politely, which would also fit. In your mind, you would stack rank these in order of priority, and because you paid attention to the word, Ireland, when deciding which one to use, you put Irish or Gaelic at the top of your stack rank. That's exactly what a large language model does, but it's the attention in the transformer architecture that really broke this technology into the mainstream. Without attention, you likely would have ignored the word Ireland in the sentence and you would have ended up with lower-quality results. This attention mechanism allows the model to process all parts of the input data simultaneously, and to attend differently to different parts of the sequence. This means that the model can weigh the relevance of all other words in the sequence when understanding each specific word, and thereby capturing context more effectively and efficiently. Indeed, one of the original uses of transformers was in language translation, a domain that greatly benefited from their ability to understand and translate the context and meaning of full sentences, rather than just doing word by word. Let's consider another example. This is a sentence in Gaelic. Do you know what it means? It literally translates into the raven loves its chicks. And you might wonder, well, what does that mean? Well, if you see a picture of a raven's chick, it has a face that, well, only its mother could love. And this brings us to one of those English phrases, beauty is in the eye of the beholder. Now that's a much better translation of the meaning of the words, as opposed to just a straight word-for-word mapping of one language to another. So when a transformer was applied, a model could learn the correct mappings from one sentence to another and thereby become a much better translator. This capability led to more fluent and accurate translations, setting new benchmarks in machine translation and significantly impacting global communication. So next, we're going to take a deeper look into what transformers are.

What are transformers?

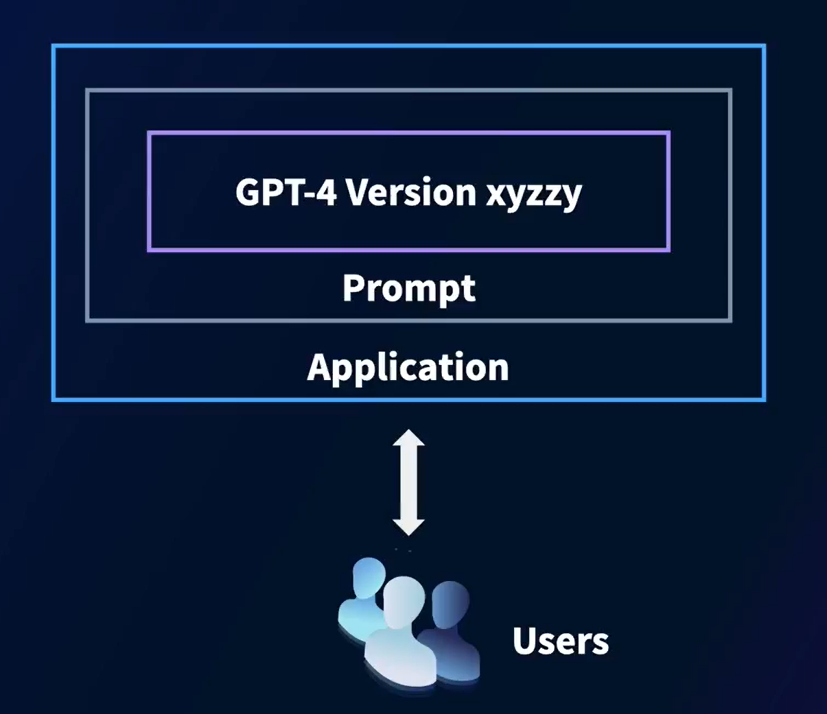
Selecting transcript lines in this section will navigate to timestamp in the video

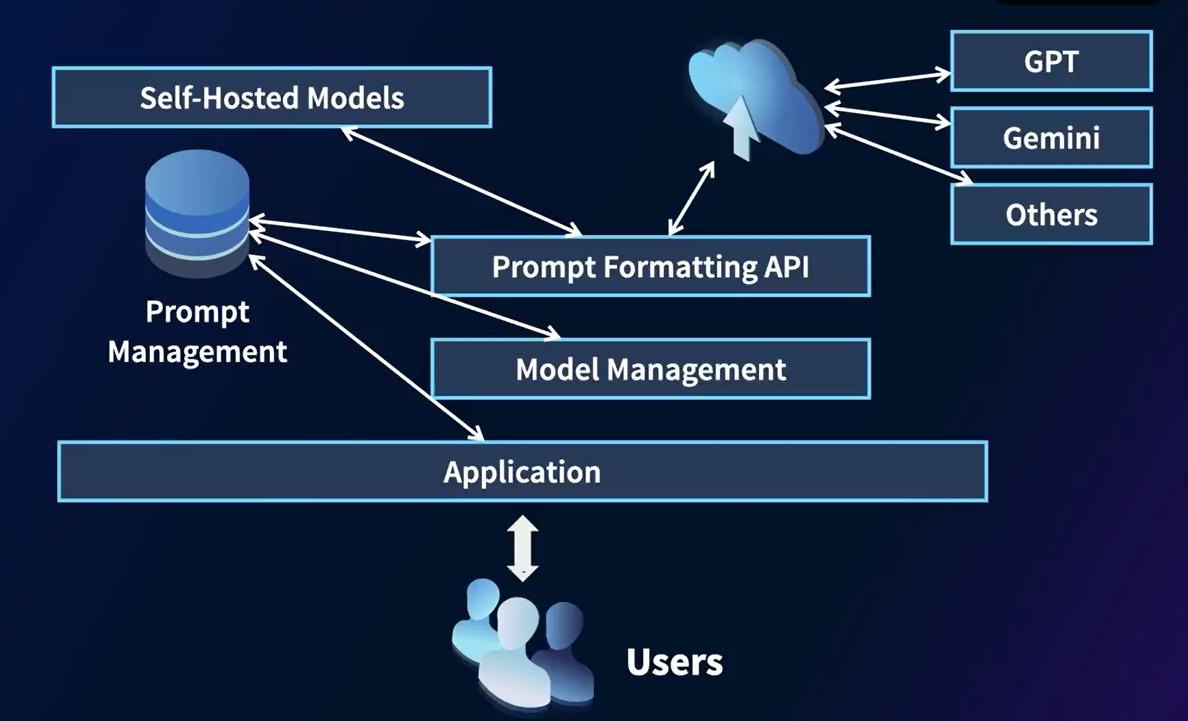
Capabilities of transformers were further expanded with the introduction of models like OpenAI's GPT series. These models use the transformer architecture to create generative, pre-trained transformers trained on diverse and extensive internet text. This gave them the ability to perform a wide array of text-based tasks. GPT-3, with its 175 billion parameters, became a revolutionary example, showcasing an ability to generate text that is contextually relevant, coherent, and strikingly human-like. In addition to generating texts, the GPT series of large language models also launched a whole new area of science, one that I like to call artificial understanding. With their attention-based mechanisms, they could artificially understand sentiment in texts and use that to help you shape thoughts or ideas. I personally find that aspect indispensable and far more valuable than just generating content. Despite their potential, LLMs are not without significant challenges. The risk of perpetuating biases present in their training data can be a critical concern. These models can inadvertently produce biased or offensive content if not carefully monitored and managed. Additionally, the environmental impact of training such large language models is considerable. It involves substantial energy consumption and associated carbon emissions. Indeed, a massive bias challenge comes when one carefully monitors and manages a model. In this case, there are well-documented examples of where filters applied by the model owner to restrict the model from behaving in a way that biases against one culture only caused it to be biased against another. It can be a difficult problem to solve. One other thing, and it's an easy habit to fall into. LLMs and their associated apps like ChatGPT, are artificially intelligent applications trained using machine learning and other techniques. But as a society, we have conflated AI with LLM and we've used the terms interchangeably. They're not. If you want to be a trusted guide to your community, it is really important to remember this. Addressing these challenges involves a combination of technological innovations and ethical guidelines. Techniques like differential privacy, federated learning, and bias audits are being explored to mitigate bias and enhance privacy. Moreover, improving the energy efficiency of these models and using more sustainable energy sources are critical steps towards reducing the ecological footprint incurred by large AI models such as LLMs, generative image, generative video, and so on. In this course, we're going to explore strategies for effectively using LLMs while responsibly addressing their limitations. You're going to learn how to deploy applications that use these models, refine their outputs, and you'll explore integrating ethical considerations into your development processes. Stay tuned as we move deeper into each aspect of LLMs, from their technical underpinnings to practical applications and strategic management. This series will equip you with the knowledge and tools to harness the power of large language models in making informed, ethical, and impactful real-world applications. So let's continue this journey of building and maintaining applications that run on an LLM infrastructure.

What is LLMOps?

Selecting transcript lines in this section will navigate to timestamp in the video

Okay. So to this point, we've explored the ideas behind large language models, including a little bit of history of them. Next, we're going to take a look at how to build and maintain applications on top of them. There's lots of material out there around prompting and how to craft prompts to get the best out of models, but we won't be going deeply into that. I do want to look at the full lifecycle of an application that you're going to build in this way, and this is called LLMOps, Large Language Model Operations, an essential discipline for managing and optimizing applications that use large language models, or LLMs, as their backbone. Throughout this course, we're going to explore why continuous monitoring, updates, and strategic adjustments are imperative for maintaining the efficacy and reliability of applications built using them. Our discussion will not only broaden your understanding of LLMOps, but also equip you with advanced strategies to manage LLMs as dynamic components within your applications. I also want to emphasize the importance of integrating LLMs as seamless background components that enhance functionality, without overshadowing the core business logic and user experience. A poorly architected application is one that has tight dependencies on a particular LLM, and which brings it close to your users with little constraints, whereas a well-architected application has the LLM fade into the background and is loosely coupled from the rest of your application. This approach allows us to leverage their capabilities to understand, reason, and generate text, while keeping our primary business objectives and needs at the forefront. It also reduces brittleness in your application as the underlying LLM changes and evolves over time. And given that LLMs and other models are being released at a breakneck pace, it helps you to be more ready for a future where the LLM you're currently using could suddenly be less desirable, maybe for price, functionality, or a host of other reasons. So more formally, let's understand the reasoning behind this architecture. And I see four major reasons here. Number one, **streamlining the user experience.** The old Google adage was focus on the user and all else will follow. And this is great advice. Users will interact with your applications, seeking solutions and experiences and not the technology per se. Integrating LLMs as underlying engines ensures that the technology remains invisible, providing a smooth, intuitive user experience that's focused on outcomes rather than processes. Next, there's **enhancing business agility**. If you decouple your core business logic from the complexities of LLM operations, you can more easily adapt to market changes and user needs without re-engineering the underlying AI models. Indeed, with a well-engineered system, hot swapping of the back-end LLM should be relatively straightforward. This flexibility is crucial for maintaining competitiveness and responsiveness as the market moves quickly. Then there's **mitigating risks**. Using LLMs as black boxes reduces the dependency on specific model behaviors, and they can change with updates or retraining from the model vendor. I've worked with and I've advised many startups that took a close dependency on GPT, for example, but the prompts that work today may not work tomorrow, and they have to take on a lot of regression testing. While a separated architecture like this won't prevent that from happening, it does make the application a bit less brittle, and it opens the possibility for a company like that to instead switch to an instance of an open-source model so that they can fully control it without large scale changes to their applications. This approach protects the application from fluctuations in model performance, and it ensures consistent functionality. You can focus on core competencies, and of course, with a well-architected system that lets you concentrate on developing and enhancing your unique offerings and value propositions, rather than diverting resources to manage AI complexities. Okay. Now that we understand the advantages of a well-defined, separated architecture, let's also explore how we can continually manage and monitor that over time to make more effective long-term application. I generally consider this in five steps. **Number one**, **define clear interfaces**. If you establish these clear interfaces between the LLMs and your application layer, they should specify how the models will receive input and deliver output without exposing the internal workings of your models. This encapsulation allows large language models to be updated or swapped without disrupting your application functionality. With LLMs, this will often mean you're going to call the back end with a prompt and a bunch of hyperparameters. **Secondly**, you can **automate model management.** And where you can do this, it's great to automate systems for monitoring, updating, and maintaining LLMs. Of course, if you're using a third-party back-end LLM like GPT, this is going to be harder. But if you do reach the point where you're using your own, it's good to have a level of automation in managing them. Do you have enough resources to run them based on your traffic demands? Or maybe you're thinking about using specialized chips like GPUs for inference, but is your fleet ready for spikes and traffic? Automation in LLMOps ensures that models are efficiently managed behind the scenes, reducing the need for manual intervention and allowing teams to focus on business-critical tasks. With **continuous performance monitoring**, regardless of whether you host your own models or use third-party ones, continuous monitoring is essential to ensure they perform as expected without directly involving your end users in their complexities. What happens if the infrastructure supporting the model goes down, or how do you mitigate issues there, for example? Or is it maybe performing very slowly? So what should you do? Indeed, even if there's no impact on your user today, there may be future impact that you want to avoid. So good performance monitoring can help you avoid those future problems. Indeed, involving your user lightly in this, by maybe having them rate the output of a model can help you choose the optimal application characteristics to continue to give them a great experience. If you **develop modular architectures**, as we've discussed a lot, these modular architectures where LLMs can be plugged in as interchangeable components is essential. This modularity allows for flexibility in using different models or versions as needed based on performance, cost, or other business criteria without redesigning the entire system. Indeed, as the industry grows towards agentic-style applications, modular architectures like this, I believe, will become commonplace, as there's no reason why a workflow shouldn't span multiple different models, choosing per-task experts as needed. Next up, **data security and compliance**. You should ensure that data security and compliance are built into the interfaces and operations of LLMs. As these models process potentially sensitive information, integrating robust security measures and compliance checks into their operations is critical, even if they're not at the forefront of the application. And, to be honest, a well-designed architecture with separation of concerns is much easier to audit and much easier to demonstrate compliance on than one that's complete spaghetti code. Okay. So we've now gotten a bit of a grounding in effectively integrating LLMs into business applications as background components. As we go forward, our focus will be on practical strategies to harness their capabilities while keeping your business logic and your user's needs front and center. By treating LLMs as powerful but subordinate tools, you can enhance your application's functionality, agility, and user experience without getting bogged down by the complexities of AI technologies. We're going to get hands-on, and we'll build a simple application that uses the power of an LLM to help you be a better public speaker. It will be much lighter and much simpler than most real-world apps, but hopefully, it will help demonstrate many of the things that we believe you're going to need to consider when building real-world applications.





Prompting

Selecting transcript lines in this section will navigate to timestamp in the video

Previously, you learned about large language models and began to understand how they are strong at completing text. Naturally, the next thing to do is to give them text in order for them to complete it. However, modern large language models can do much more than just a basic autocompletion of texts, and getting them to behave the way that you want becomes an art and a science. We'll explore that in the next few videos, with the goal being, whether you're an absolute beginner or an experienced pro, will help you get the most out of your interactions with AI. I'm going to cover the basics of prompting and how it works. So let's dive in. First things first. What exactly is prompting? In the context of interacting with LLMs, prompting is the art of providing an input or a set of instructions to the model to generate a response. Think of it as asking a question or giving a command to a very knowledgeable assistant. For example, if you prompt with, tell me a joke, the model processes your request and generates a joke in response. It's really that simple. But as we'll see, prompting can be much more powerful and nuanced. So why is prompting so important? Well, the quality of the response depends heavily on the quality of your prompt. A clear, well-structured prompt can yield a precise and useful answer, while a vague or poorly worded prompt might not get you the information that you need. So let's break down the components of a good prompt. You should **be clear**, and you should use straightforward language to avoid confusion. You should **be specific**; provide as much detail as is necessary. You should **be contextual**; give the model some context to understand your request better. You should **be goal-oriented** and be clear about what you want to achieve with the response. So here's an example. Consider this vague prompt, help me write an essay. The model might not know where to start or what aspect of history you're interested in writing about, and you'll get a response like this. But how about a more specific prompt? Help me write a history essay about the important events in the Industrial Revolution. This prompt gives the model clear instructions and context, making it more likely to provide a useful response. And you can see when I tried this, I got a very detailed response, and it's much more than I can even fit on this slide. So now let's take a look at some **basic prompting techniques**. You can use simple queries, instructions, or questions to interact with a model. A **simple query** is a direct request for information or actions. An **instruction** is a command for the model to perform specific tasks. A **question** is an inquiry that requires a detailed response. Pause the video for a moment and then try a simple query, an instruction, and a question in your favorite chatbot. So for a question, and this is a true story, one day I was out running with a friend in Tokyo, and this animal came out of the underbrush near a park. We had no idea what it was, and to me, it actually looked like a red panda, which looks like this. But it couldn't be a red panda unless it had escaped from a zoo. But I could ask my favorite chatbot a question like, what is the animal that looks like a red panda but can be found in cities like Tokyo? And I got the answer of a creature called a tanuki. And here's a picture of one. Now, maybe that's what I saw, but I could swear the one that we encountered was a little bit more red in color. ChatGPT's description said that Japanese folklore mentions that these creatures have shapeshifting abilities, so maybe. Anyway, what did you try? Please share your experiences with us. Regardless of whether you're inquiring, instructing, or questioning, here's some good tips. **Be explicit.** Clearly state what you want the model to do. Instead of saying something like, tell me something interesting, you could say, tell me an interesting fact about space exploration. In my case, I was as explicit as I could be describing the animal reminding me of a red panda. Also, you should **use complete sentences** when you can and this does really help reduce ambiguity. Don't be afraid to **iterate and refine**. If the model's response isn't quite what you wanted, refine your prompt and just try again. Iteration is the key to improving the quality of responses. You should **set constraints**. Sometimes adding constraints can help you get more accurate answers. For example, list three benefits of exercise for mental health is much more focused than just benefits of exercise. Look at the case that I shared. An important constraint was that the mysterious creature I saw was in Tokyo. And of course, don't be afraid to ask follow-up questions. If you need more information, just use follow-up prompts. For instance, after getting a summary of an article, you might ask, can you explain the main argument in a bit more detail? Let's walk through an example of iterating and refining a prompt. Suppose I ask, explain climate change. The response is very broad and it might not contain the details that I need. So if I refine my prompt to explain the main causes of climate change in simple terms, this refined prompt guides the model to provide a more targeted and understandable answer. Remember, the model is artificially intelligent. It simulates intelligence by artificially understanding information and repeating things back to you. You cannot assume that it understands what you're asking, and although it's artificially very strong at getting the sentiment from your prompts, it's still artificial. It will make mistakes. So being super clear can help avoid this. And in that way, it's a bit like coding. And just like coding, it's important to be aware of common mistakes in prompting to avoid them. **Don't be vague.** Avoid prompts that are too broad or unclear. **Don't overload** by including too many requests in a single prompt. Break it down into multiple prompts if necessary. **Challenge your assumptions** and **don't assume** the model has the same context or knowledge as you do. Provide necessary details in your prompt. **Don't neglect context**. Provide as much detail as you can. For example, instead of a vague prompt like tell me about science, a more effective prompt would be, tell me about the scientific method and its importance in research. This provides the AI with a clear topic and the context needed to generate a useful response. Pause the video and try these out for yourself to see the difference in output. To recap, prompting is a powerful tool for interacting with artificially intelligent models like Gemini or GPT. By crafting clear, specific, contextual, and goal-oriented prompts, you can significantly improve the quality of your responses. In our next video, we're going to explore how to use system prompts to specify a role for these purposes.

Advanced prompting

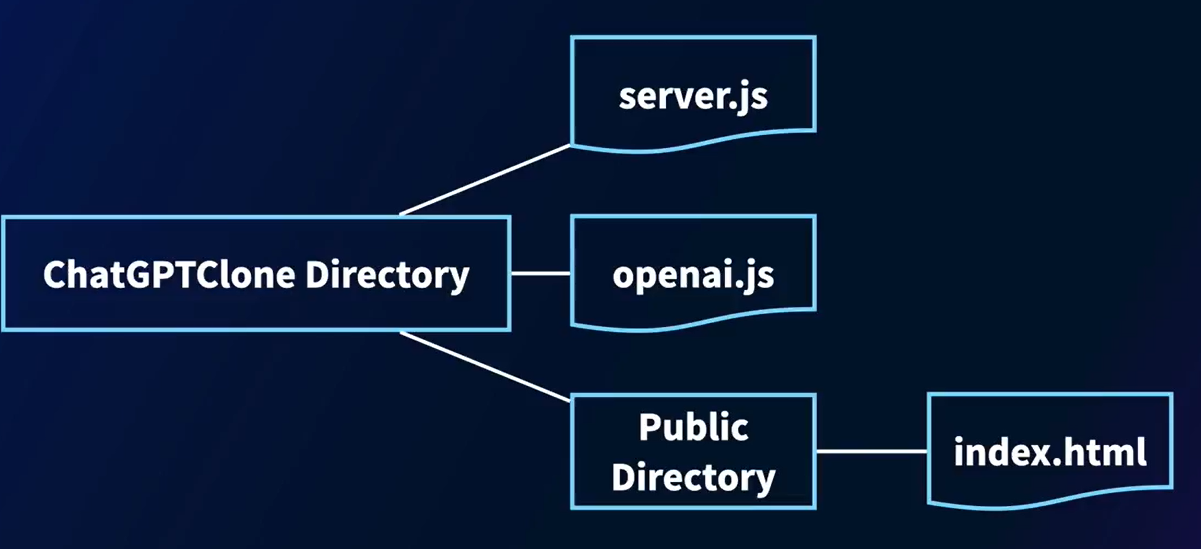
Selecting transcript lines in this section will navigate to timestamp in the video

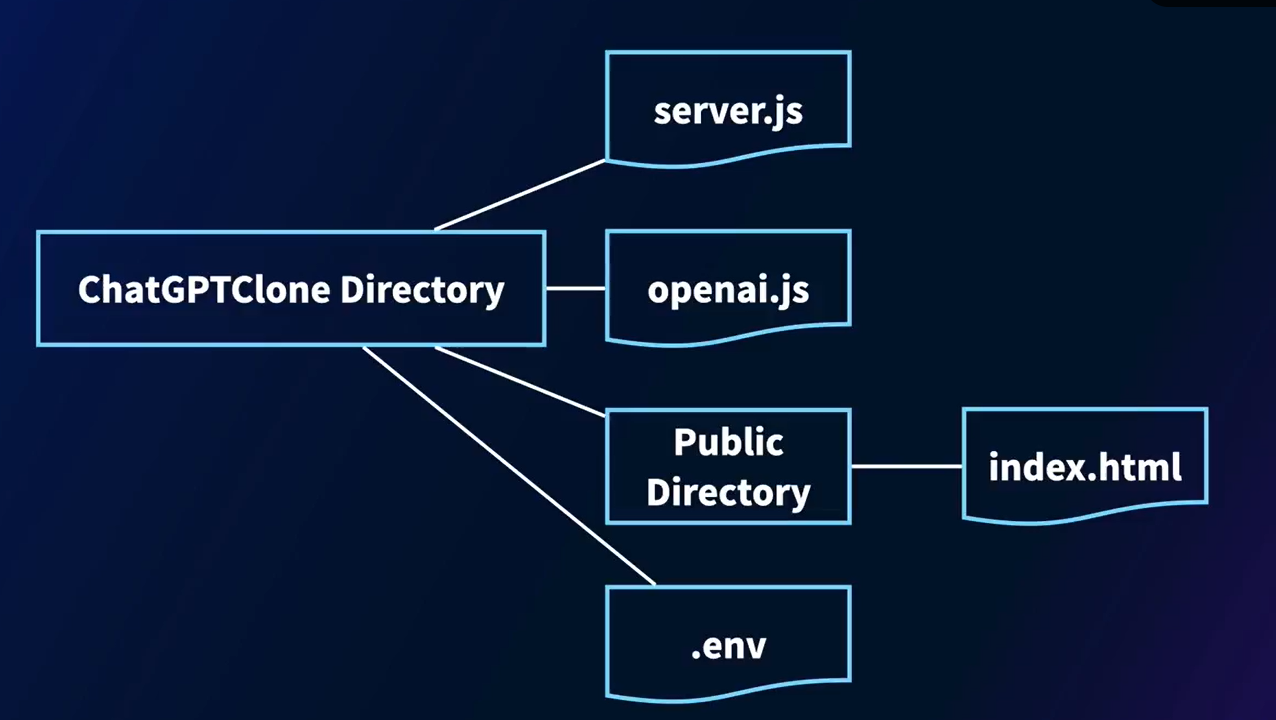
Let's now move beyond the basics to master some more advanced prompting techniques such as system prompts and memory. These capabilities are incredibly useful for anyone looking to build smarter applications that use the reasoning power of large language models. So buckle up as we explore these fascinating aspects of LLM technology. In the previous video, we explored basic prompting techniques, how to ask direct questions or command simple tasks, and how to work on constructing them clearly. In this video, we're going to expand on that foundation by introducing three advanced components: roles, system prompts, and memory. Each plays a vital part in transforming how we interact with LLMs and helps us to really bring the most out of them. Before we dive in, let's briefly recap why these advanced techniques are so important. As LLMs become more sophisticated, our ability to guide and shape their outputs becomes increasingly crucial. They're getting bigger, smarter, and more complex. So in order to really hone any applications that may use them, we really need the full set of tools for working with them at our disposal. And while simple prompting is very powerful, isn't it much better to get more out of the LLM if we can? So let's start with the idea of giving the model a role to play. Now these are basically very simple system prompts where we tell the model to behave in a particular way. And this involves more than just asking it to perform a task. It means setting a persona for the AI that aligns with specific expertise or behavior patterns. This technique enhances the relevance and appropriateness of the model's responses. For example, imagine you're developing a virtual assistant for tech support, by prompting the model with, as a tech support agent, guide a user through troubleshooting their internet connectivity issues. The AI will adopt the persona of a support agent using technical yet accessible language to assist the user effectively. And here you can see my interaction with ChatGPT, where I gave it that type of command. I start with the prompt, you are a friendly and informative tech support agent. You're an expert in internet connectivity. Your goal is to help me get reconnected. Start by asking me how you can help. This gets ChatGPT to act in that way and it will stay on character helping me through my issues. So for example, I'll type. I can't access the internet. The agent will go into its first set of steps to help me. It stays in the friendly and informative character, pleasantly asking me to check on various things. So I respond no Ethernet, and vaguely it looks like I'm on Wi-Fi. I try the steps that it gave me, but I give more specific information back. For example, I can't see google.com. Unfazed, the friendly and helpful assistant doesn't berate me for not telling them that earlier and instead goes through some more steps. One of them is to try the IP address instead of the domain name, and it did give me actually the right IP address for google.com. So no hallucinations here. After informing it that the IP address worked, It now stayed helpful and it showed me how to change my DNS settings. It also gave very specific instructions, such as how to set my DNS, flush my DNS cache, and stuff like that. It did what I had asked it to do and it really helped me out. Now, if there was one issue I had with its performance, it was that it didn't ask me about my operating system first, but that was just a very simple prompt I gave it. I could have been more explicit. Imagine other scenarios you could use with roles like this, such as for creative writing. You could say, as a novelist, describe a serene landscape that could serve as the setting for fantasy story. The LLM will then shift its style to be more descriptive and imaginative, crafting detailed and vivid imagery that could spark creative thought in the author's mind. The AI might produce a response like picture a tranquil valley nestled between towering, mist-shrouded mountains. Crystal-clear streams meander through lush meadows dotted with iridescent flowers that seem to glow in the soft twilight. Ancient, gnarled trees with simmering silver leaves lined the edges of the valley, their branches reaching towards the sky painted with impossible colors. Now that's much nicer than I could write alone. One more example. What if you prompted GPT with something like, as a financial advisor, provide guidance on creating a diverse investment portfolio for a risk-averse client. It should adopt a more analytical and cautious tone. So go ahead, pause the video now and try it out for yourself. Explore the language that it uses as you interact with it, and then come back when you're done. When I tried it, the response included explanations of different asset classes, the importance of diversification, and suggestions for low-risk investments like government bonds or blue chip stocks. The language was professional and reassuring, aimed at building trust and educating me as its client. These prompts had very simple descriptions, maybe only a sentence or two, but we got some really strong output from the LLM. Because we spent a lot of time specifying a role that the LLM will follow, the terminology can get a little confusing. Try to think about these as system prompts, which is a special prompt that starts the conversation and remains in context throughout. Often that prompt, as in the examples we've shown here, gets the LLM to perform in a particular role. But the confusion can come when you start using APIs instead of chatting directly with the LLM via something like ChatGPT. For example, here's a simple call to the OpenAI API, and you can see that it uses the term role to denote either the LLM, which it calls the system, or the person accessing it, which it calls a user by role. And the system prompt that gets the system to act in a particular way, such as a customer service AI assistant, which we often say is its role, is the contents of the system role. The final idea here is memory, and incorporating memory into LLM interactions is perhaps one of the most significant advancements in making AI interactions more human-like. Memory allows LLMs to recall previous parts of the conversation or even previous interactions, depending on the system's capabilities. Think back to the example that I showed earlier where I was interacting with a customer support agent. Notice how I was able to continue the conversation naturally without having to retell it my details each time. That was powered by memory. When creating your own apps, you could implement memory, but LLMs like ChatGPT have already implemented it for you. However, do be careful that the longer the chat goes, the more diluted the memory can become. So enough theory, let's get hands-on next, and we'll look at building our first application that will play a role, and that is of someone helping you to be a better public speaker.

Hosting an app

Selecting transcript lines in this section will navigate to timestamp in the video

So we've seen what LLMs are all about, and we went a little deeper into prompting, seeing how roles and system prompts are used in particular. We also discussed memory a bit, and we saw how models like GPT work well, because they have memory of the conversation and they can act on it. Next, we're going to build our own ChatGPT-style application that's a specialist in helping people with public speaking and putting together talks. The application will look like this. It introduces itself as being available to help you create an engaging and powerful talk, and it asks you about what your topic or idea will be. I started with something silly, the migration patterns of wombats. After thinking about it for a moment, the LLM comes back with clarifying questions. Who's the audience? What's the purpose of the talk and what are the key points I want to cover? Stuff like that. I'll answer this question and note that it's now using memory. I'm not mentioning wombats at all anymore. I'll tell it that it's for a general audience and the purpose is to inform and inspire. But I do need help with the key points. Now it comes back with a very detailed answer. Now this app is a little bit primitive, so there's no output formatting. And it might be nice to render this markdown. Anyway, it does break my talk down into a number of bits. And bit number one is an interesting introduction to these creatures, so I'll ask for it to elaborate on this. And given that LLMs are experts, even my misspelled please is recognized and understood when I ask for a 200-word script. After thinking about it for a bit, I start to get a really interesting script to help me introduce an audience to wombats. I never realized that wombats have pouches like kangaroos, but they face backwards. Note also that because of its role as an aid in preparing talks, the bot will continue the conversation using memory for everything that's happened so far. It will ask if this text captures my desired tone and information depth, and I think that's pretty cool. So then the next question is, well, how will I build that? And I'm glad you asked. And that's what we're going to cover. First, be sure to have Node.js installed on your machine. If you don't have it, you can find details at nodejs.org. Make sure it's installed before continuing. If you have Node installed, make a directory on your machine(mkdir chatgpt-clone), change to it(cd chatgpt-clone), and then install npm like this(npm init -y). After that, you're going to need to install some dependencies, so please use the following (npm install express dotenv openai socket.io). In the repo, I've provided three files: server.js, index.html, and app.js. Now please make sure they're put in the directory properly, as you can see here. Then add a .env file to your project root and add your OpenAI API key there as an environment variable. You can get an API key from OpenAI. Just visit platform.api.com/api-keys. It should look like this, and you can create a new secret key from there. Once you've done this, you can run node server.js from the root directory and your chatbot should be working. Just simply visit localhost:3000 to play with it. Next, we'll take a look at how to create this application from scratch.





Create a chatbot

Selecting transcript lines in this section will navigate to timestamp in the video

Okay. So let's get started with building the chatbot that's going to be the foundation of everything that you do in this course. So I'm going to be going through a lot of code here and a lot of all of the foundations of setting up that chatbot. So all of the code is available in the repo for this course if you want it. And you can just say git clone. And here's the URL of the repo. Once you've done that, you can go and you can take a look in the directory that gets cloned. And you'll see these are the directories that are done for you. Chapters one, two, three, and four, API key, README details, a license, and a README.txt file. So the first thing that you're going to need to be able to continue, of course, is an API key for the back end. And the back end that we're going to be using is OpenAI's ChatGPT back end. If you go to openai.com, and on there go to products, you'll see the API login. If you select that one, it will take you through a login process. If you're logged in already, it will look like this. When you're on here, go up to the upper right-hand corner, and you can see my little face up here, you select that, go to your profile, and then go to user API keys. Once you're in user API keys, you can scroll down. And once you scroll down, you'll see this button to create a new secret key. You give it a name, say, For LLM Course Details, something like that. It's really up to you. Hit Create Secret Key and you'll get an API key. If you hit the copy button, it'll add it into your clipboard. Now that you've done that, you need to create an environment variable on your system so that Node.js and Python can read that as they're actually executing. Now, in the README file, there's some details on how to do this. And it's different in different systems. Look up the help for your particular system on how to create an environment variable. But on my current shell, on my Mac, I'm just going to say nano ~/.zshrc, and we'll see. This is the configuration file for my shell. And in here, I'm exporting the open API key. Make sure you use lowercase exports and all uppercase OPENAI\_API\_KEY, and then paste the API key that you got from OpenAI in quotes into this one. Save that out and to make sure that it works, you should just be able to say echo $OPENAI\_API\_KEY like this and it will be written out for you. So now you know your system can read the API key. So any code that you write in Node.js or a little bit later in Python is able to access that. And you don't need to put the API key into your source code itself. Okay. So now let's take a look at building the chatbot that you're going to be using throughout this course. And once you've cloned the repo, you're going to have these two directories within chapter one, first app, the start, and end. End will have the completed code for you if you want to jump straight to that, but start will have the foundations of the code for you. If you've already installed Node or Node dependencies, you'll see this Node modules. If you've just cloned this from scratch, it won't be there. Don't worry about that. Like I said, as you install dependencies for Node, that directory will get written for you. Okay. So what are we going to build in here. So we're going to work on three different files. The first is openai.js, and that's where we're going to put the stuff in there so that Node can read that environment variable that we just created, so it's able to access the back end from ChatGPT. Server.js is going to be our back end running on our Node server that handles communication with our front end, as well as calling the API on the back end, as we saw on the slides that does all of the chat completions. And within public is index. html, and that's going to contain all of our front-end code that gives us the chat interface and then handles taking what the user has said, sending it to the back end, and taking what the back end has completed and sending it back to the user. I'm going to start by showing what you need to do in openai.js. I'll open that in Sublime, and that's going to be an empty file. So if I bring this empty file over, we can see there's nothing in there right now. And I'm going to start adding the code to that that we're going to want to use. So let's start by just using the requires that Node.js needs to be able to access the OpenAI APIs. And that's just a line of code like this, const OpenAI = require( 'openai' );. Now we want to read the API key from our environment. And we do that with code like this, where we're just going to say process.env.OPENAI\_API\_KEY. If you remember, that was the name of the environment variable. And we're just going to say const OpenAI equals that. So now we have the API key. And we're able to access that API key within our back-end code that I'll show in a moment. But in order to do that, we need to export this file. And we're just going to say module. exports is OpenAI is this const that we've just defined. So now we have this file and I'm going to save that out. And what this file is going to do is when we include it in our Node.js backend, it's going to give us access to OpenAI's API key. Okay. So then the next thing we want to do is take a look at our back end. We can see our back end is nice and empty. So I'm going to add stuff to this piece by piece so we can see everything that we need in order to be able to create our server that will give us the heart of our chatbot. So I'm going to start this as it's a Node.js server app. It uses Express, if you're familiar with Node.js. And these are all the includes that I'm going to bring in to give me the functionality that I need to be able to make the app to work. So it's going to use Express, it's going to import HTTP, so I can use it in the browser. It's going to have socket.io. Socket.io is going to manage communication, real-time communication between the front end and the and the back end so that we can be sending messages to each other. Path is just going to read from the file system in order to give us our HTML files. And here we're just going to specify that we're going to create a server that runs on Port 3000. A moment ago, we saw that we created the API key for OpenAI and we exported that. So I'm just going to say here we're going to have a const openai, which is the import of that openai.js that we just created. And then now if we want to just set up the last part of our server, there's a couple of things that we need to do. First of all, remember we imported path here. We're going to import from the public folder and our web server static files are going to be in there. And when the user types the route http localhost 3000/ with nothing after it, this route is going to say okay, send index.html in that case. And in the moment, I'm going to show you the contents of index.html, which will be the front end to the chatbot. Now let's start looking at the actual chatbot functionality. And the first part is, if you remember in the slides, we were talking about giving the chatbot a role. And that role would be it's an expert in public speaking, and it's going to help you be a better public speaker. Of course, you can use whatever system prompt you want here, but this is a pretty long and descriptive system prompt that we've defined about how to create engaging, powerful talks, et cetera. So that's just set up as a const. So now what we start to do is in the chatbot itself, the first thing is like as the user is typing messages to the bot, they're going to be sent over a socket to this back end. So this back end has to connect to that socket in order to be able to communicate with it. And it does that using the io namespace. So what we're going to look at now is when the user first connects. And we're going to capture that in the io.on connection code. And that's going to have a socket. But what we're just going to say is we're going to log the fact that a new client is connected. And over that particular socket, we're going to emit something that we call a chat response. And remember that when we look at the client, we'll see that we're catching chat responses and doing something with them and what that chat response is going to be is the first message that says, I'm here to help you create an engaging and powerful talk. Once the client connects to this server, it's going to send back this chat response. Now remember, multiple clients could be connected to the server. That's why I'm just going to log new client connected here so we can see how many clients are connected. Now the important thing as we looked at when we were going through the architecture of an application like this, is this idea of a conversation history. So the conversation history is the entire conversation between you and the chatbot during this session. It's going to start with something called a system role. And that system role is the system prompt that we just set up. After that, then all of the messages are either going to be user or assistant. When the user types in something, that's going to be added to this conversation history as a new entry with the role user and the content being whatever they typed, and as we call the API to get it to respond, that's going to be added to the conversation history with the role assistant and the content being whatever came back. So keep an eye on that and remember that as we're working through this. So let's now take a look at what happens when we receive a chat message. And this is a long chunk of code, but I'm going to go through it little by little so that you can understand it. And you know, this is going to be the longest piece of code that we go through as we're creating this application and through this course, because this is, like I mentioned, giving us the foundation on what we're going to work on, and then we're going to be adding little bits to this as we go along. So it's good to understand what's actually happening here throughout. Okay. So when we receive a chat message on our socket, what do we want to do? Well, the user experience is going to be the user type something, they hit enter. That means the message is then going to be sent to this back end. But on the front end, we want a nice user experience for the user. So we don't want them to be paused, waiting, wondering what's going on. So we're going to send back status to say we are thinking. And I'm sending back this thing called thinking and setting that to true that will be caught on the front end, and the front end is going to render like a little gray bar that says thinking until we change this from thinking to false, so that the user just gets that feeling that something is happening. And if you've used ChatGPT, you've probably seen something like that, that you type something, and sometimes it's like little animated dots and stuff like that. That just means we're doing that thinking status while the process is going on. Now, a moment ago, I explained the conversation history. Now what has happened here is the user has sent us something. We've caught that, but we want to add that to the conversation history. So you can see we're pushing the role as the user, the message that they sent to us. We've received the message and we're pushing that to the conversation history. Now, in order to get ChatGPT or the OpenAI API to do its thing, what we're going to do is we're going to send that entire conversation history to the OpenAI chat completions API. That's going to read that and it's going to generate new content. So you can see we're sending the entire conversation history. And we've chosen we're just going to use model GPT-4o. You can change that to whatever you like, of course. It's going to await that. So we're working asynchronously here. And once that gets called back, we're going to get this response. And again, if you remember from using ChatGPT, sometimes when you chat with it, it gives you back multiple answers and you can pick between them. So what I wanted to do here is if in that case, I'm always just going to pick the first answer, you could update your user experience here if you wanted to, to allow them to choose between answers. But for simplicity, I'm just going to take the first answer. So once we get that, I'm going to call that response. And now that response gets added to the conversation history. So again we're going to push to the conversation history. This came back from the OpenAI API. So I'm going to call the role assistant and put the response in there. So as we chat, you'll see that this conversation history continues to get updated. First the system role, then the user, then the assistant, then the user, then the assistant, and so on. We've received our response now, so we don't need to be thinking anymore. So we're just going to admit that the thinking status is false. So the user interface will update. And again, we'll see that in a moment to show that we're no longer thinking. And instead, here's the response that we're sending back to the front end. And the front end will then display that response on the screen. If an error happens, we just want to catch that and let the user know. So the other piece of code that we need to write for socket management, of course, is what happens when the socket disconnects. So we just want to say, if the socket has disconnected, we want to console. log that that has happened and make sure that everything is good. So we're just going to say here, look, our client is disconnected. We're good to go. And then finally, our server is going to listen on a particular port. Let me just tab that properly over here so we can just have nice clean code and console.log it like that. So now we're just going to disconnect and log out that the client is disconnected. And then finally, the piece of code that we need to start the whole ball rolling is really just to get our server started. And we do that with server.listen. Remember, we said our port was 3000. We're just going to log that the server is running on that port. So now that's our entire server that we've created in just 70 lines of code. So I'm just going to save that out so we can run it in a moment. Okay. If I go back and remember what we have, we had our openai.js that we've just created. We had our server.js that we've just created. Now finally, we need to create our front end, which is in index.html. And we'll see that right now this is nice and empty. We have nothing in there. So let me just move that over and we're going to get working on that file to create the front end to our application. Okay. So now let's start working on our front end. I'm going to start by pasting in a big blob of code, which is just the styles that I'm going to use in here. You can use my styles if you like, you can change them to your own. But all I'm going to do here is just within the head. I have some CSS that defines things that like what the chat container looks like to make it pretty, what the user input looks like, all of those kind of things. We have a send button. You know, I just want to give it some color, give it a border or stuff like that. These are just the pure styles for the user interface. It's up to you to define them however you like. And, of course, you can use mine. The second thing I'm going to show, again, I'm just going to paste in is the user interface itself. So the user interface for chat, we're just going to have a chat container. We're going to have a scrolling div with all of the chat messages, both what we say and what we get back. We're going to have the div that shows when we're thinking. So again, as I was saying earlier, when the user types in something, sends it to the back end, the back end needs a little bit of time to take that and send that to the GPT back end to get a message from the GPT back end and send it back to us. And instead of nothing happening on the screen, I'm just going to show a thinking div here That could be like the animated three dots that you see in ChatGPT, for example. But in this case, I'm just going to show the word thinking. And then finally, the user input. There's going to be a text box where they can type in what they want to say. And there's going to be a send button that they use to send it. So that's just a very, very basic user interface. I like to separate the styling from the declaration itself. So you can see why we have the styling and declaration there. Now one of the things I've mentioned throughout is that we're using socket communication in order to manage the comms between our front end in the browser and our back end in Node.js. And this is available in this beautiful script called socket.io.js. So I'm just going to include that like this. And we'll get all of that function automatically. So now I'm going to write the code that executes on the front end to actually be the chatbot. And I'm going to start by just pasting in this script with a bunch of consts that we're going to be working with to begin with. Our first one is going to be the socket that just allows us to have that io, and you've probably seen things like io.on, messages, and stuff like that. That's what's in here. And then there's things to be able to look at my user interface and read or write to my user interface for things like the chat messages, the input button, the send button, and the thinking indicator. So I'm just going to create a const to refer to them. Let me move my code up a little bit so we can read it better. Okay. So the next piece of code that I want to show is what happens for when we want to add a message to the screen. Now this is going to be called in two different ways. The first one is when the user types a message, you would expect the experience to be that that message will appear in the chat window right away, even while we're thinking of the response. So we're going to call it then. And then, of course, the second one is when we get the response back from the OpenAI API, with the completion, we also want to add that. So I've created a single addMessage function here to be able to handle that. So what that's going to do is it's going to create a new message element, which is going to be a div that contains the texts and styles it nicely for us, and then our chat messages, which is the scrolling history of chat messages, we're just going to append that to it, and then we're going to scroll the chat messages to that so that we'll always see that. So very, very simple. Just add the message to the user interface. Where the fun happens, at least to begin with, is what happens when the user types something in and hits the send button. If you recall, there is a text box where they type what they want to type, and then there's a button that they'll press or they'll hit enter in order to send that to the back end. Well, how does that work? Well, first of all, we're just going to read what the user has typed. It's messageInput.value.trim. Trim will get rid of like leading or trailing spaces. And then we're going to add that message with the function that we've just declared up here. So that message renders right away. And then on socket, we're going to emit chat message. And if you remember, earlier, we were catching chat message in the Node.js back end and taking that message and sending that to the OpenAI API in order to get the completion. Here's where we emit it to that back end. Once we've done that, we're going to clear the inputs and we're going to disable the send button. This is a little bit strange. Why would you disable the send button? Well, the idea here is so that the user doesn't keep hitting send, send, send while they're waiting for answers to come back. We'll re-enable it once the answer comes back, and then the user can type something new. And then just to make the user interface a little bit more user-friendly, one of the things is like on the text box itself where they're typing the message. If they hit enter, we'll also send the message and we're just saying, hey, look, if they hit enter, it's the same as clicking the send button. So instead of them typing something and then needing to move to their mouse and push to go to the button and push the send button, we'll also support hitting enter here. Okay. So now the next part that we want to look at is what happens when the back-end server that we created in Node has received a response from OpenAI with a completion to our chats. And then that was sending us a message that it called a chat response. So here, we're just going to say, hey, on getting a chat response back from that, let's read the response. And then let's call the addMessage that we wrote a moment ago with that response. So it adds it to the user interface. And now the button is no longer disabled. So we see that flow. We've typed something in. We use socket.emit to send that to the server. The server does its thing. It emits back to us a chat response. And then we use socket. on to catch that chat response. And now we've updated our user interface with that. But one of the things we also wanted to show was that, again, for just a better user experience was what to happen while it's thinking. So here there was a status on the server about thinking. Remember, when it received the message it sent back to us to say, I'm thinking while it was waiting for the message to come back from OpenAI. Once it received that, it was no longer thinking. So again, we're catching socket.on thinking. This is a Boolean true or false. So if the thinking is true, we're going to display that we're thinking, we're going to disable the button. And once the thinking is no longer true and it's no longer thinking, and we've gotten a message, we just want to make sure that our messages scroll to the right place. So we see what has come back from the server. That's the entire thing. And then I'm just going to close out here, just close out my script tag and my body tag and my HTML tag. And we have the entire client and server have now been created. All right. So let the fun begin. If I go back now, and I'm in the directory that I was working in, you should be in the first app directory. And within there, we were in start. We've now added and edited all the code in start. And I'm going to say node server.js. If you get an error here about Express not being installed, you could just say npm install express, and you should receive a message like this one. So that Express is already installed. If you git cloned it raw, Express may not be installed on your system. So just say npm install express if you get an error trying to run it. And now I'm just going to say node server.js. And you can see here the server is running on Port 3000 and a new client is connected. So let's go and take a look at what this would look like. So I'm going to refresh here. And now we can see our chatbot is active. Hello, I'm here to create an engaging and powerful talk. You remember that was the first message that we sent over. There was system role was set, but that's not displaying. And now I can actually chat with it. So like maybe help me with a presentation about the migration patterns of wombats. It's thinking, as we've described, it's getting the stuff from the back end. It now has sent it back to the front end. The front end has rendered it. The scrollbar has scrolled to the appropriate position. And here's what it actually gave back to us. And because we gave it that role about being a helpful assistant in presentations, it's giving us all these bullets, which is pretty cool. And it may be also asking questions like, am I planning to use visual stories or data? Once we have a clearer picture, we can delve deeper into forming the content. How do you see each of these points fitting into your talk? It's having that conversation with me. Let's say, I'd say, I would like this to be for a young audience, to inspire and inform them, and include some humor. Something like that. And now again, it's thinking. The thinking status is set because it's working with the back end to do that chat completion. Once that chat completion is done, the thinking goes away and it scrolls back. Now to a really nice breakdown for what my talk should be, for example, introduction, meet the wombats. Some silly facts about wombats. Like for example, they have cube-shaped poop. I had no idea of that before I started working on this. And to kind of talk through things about wombats. Now this is just markdown that it's given me. I'm not formatting the markdown or anything like that. That could be an optional update that you could do to this. But we now have our chatbots, our focus chatbot on helping create presentations. This is the foundation for everything that we're going to be doing in the course. So work on this code, get the chatbot to work. If you do get a little bit stuck, I've created that end folder with all of the working code for you, so you can take a look and maybe fix some bugs and just have some fun with it. And in the next video, we're going to take a look at logging and how we can start adding logging to this application. Thank you.