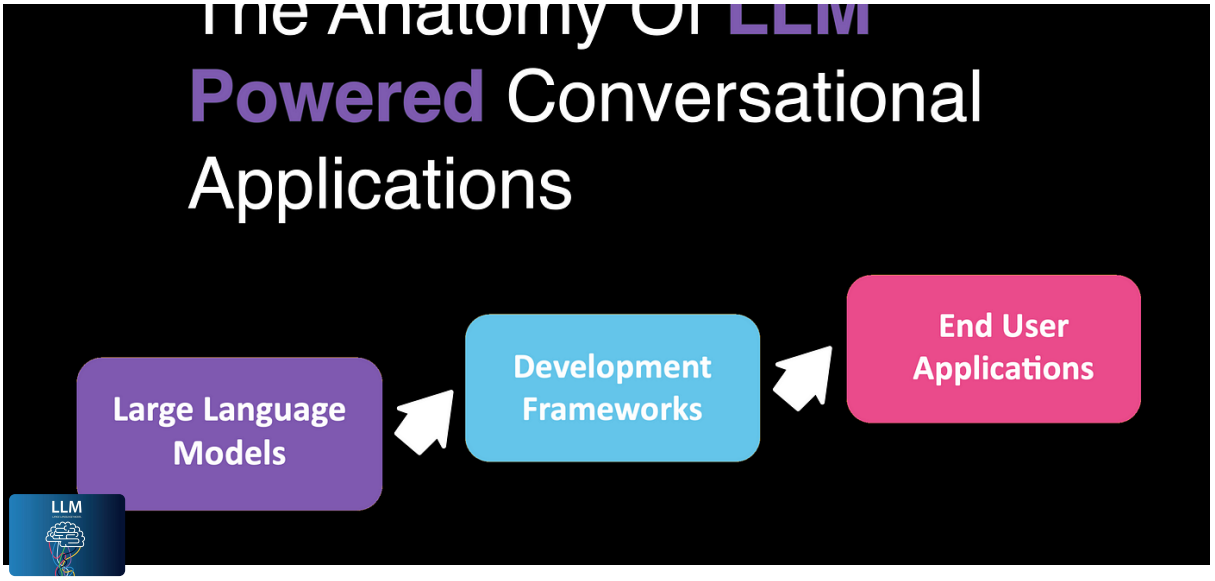


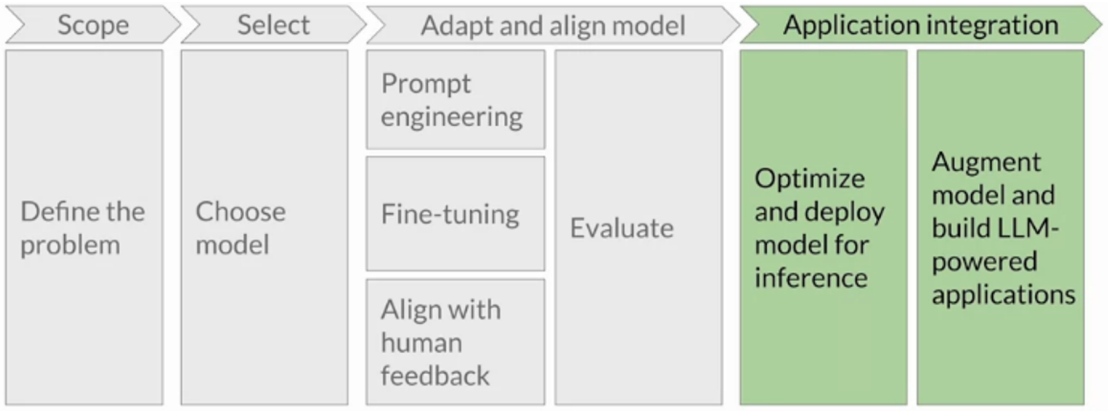
LLM-powered applications

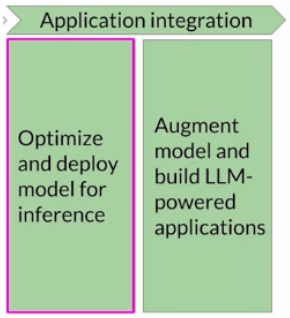
> Source: Coursera

▼ Model optimizations for deployment

▼ Generative AI project lifecycle

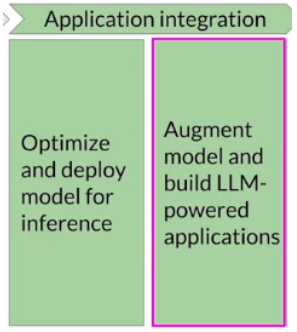
After the work required to adapt and align large language models to the tasks, some things need to be considered when integrating the model into applications.





There are several important questions to ask at this stage. The first set is related to how the LLM will function in deployment. So, how fast does the model need to generate completions? What is the available computing budget? Is trade-off model performance for improved inference speed or lower storage necessary?

The second set of questions concerns additional resources the model may need. Is the model intended to interact with external data or other applications? If so, how will it connect to those resources? Lastly, there's the question of how the model will be consumed. What is the intended application or API interface through which the model will be consumed?



▼ Model optimizations to improve application performance

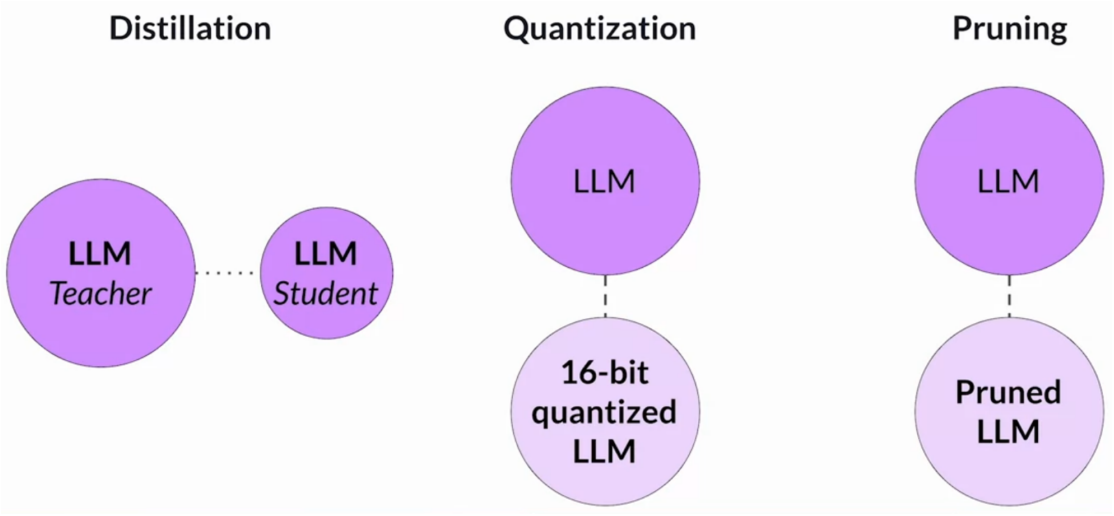
Let's start by exploring a few methods that can be used to optimize the model before deploying it for inference. While we could dedicate several lessons to this topic, this section aims to introduce the most important optimization techniques.

Large language models present inference challenges regarding computing and storage requirements and ensuring low latency for consuming applications. Whether we're deploying on-premises or to the cloud, these challenges persist and become even more of an issue when deploying to edge devices.

One primary way to improve application performance is to reduce the size of the LLM. This can allow for quicker model loading, reducing inference latency. However, the challenge is reducing the model size while maintaining model performance. Some techniques work better than others for generative models, and there are tradeoffs between accuracy and performance.

▼ LLM optimization techniques

This section includes three techniques.

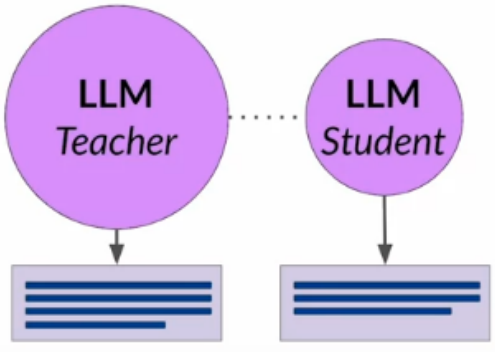


Distillation: uses a larger model, the teacher model, to train a smaller model, the student model. The smaller model is then used for inference to lower the storage and compute budget.

Quantization: Like quantization-aware training, post-training quantization transforms a model's weights to a lower-precision representation, such as a 16-bit floating point or eight-bit integer. As you learned in week one of the course, this reduces the model's memory footprint.

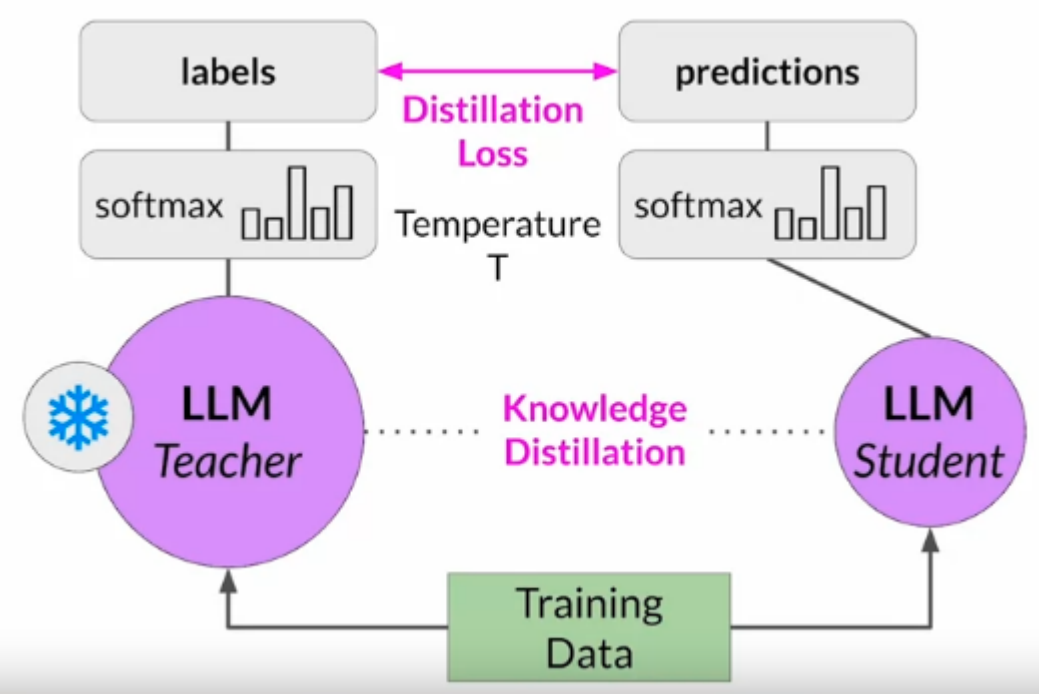
Model Pruning: Removes redundant model parameters that contribute little to the model's performance.

▼ Model Distillation



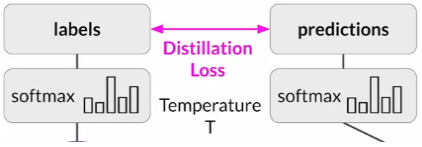
Model Distillation is a technique that focuses on having a larger teacher model train a smaller student model. The student model learns to statistically mimic the behaviour of the teacher model, either just in the final prediction layer or in the model's hidden layers.

Train a smaller student model from a larger teacher model:



Focus on the first option, start with the fine-tuned LLM as the teacher model and create a smaller LLM for the student model. We freeze the teacher model's weights and use it to generate completions for your training data. At the same time, we generate completions for the training data using the student model. The knowledge distillation between the teacher and student model is achieved by minimizing a loss function called the distillation loss. To calculate this loss, distillation uses the probability distribution over tokens produced by the teacher model's softmax layer.

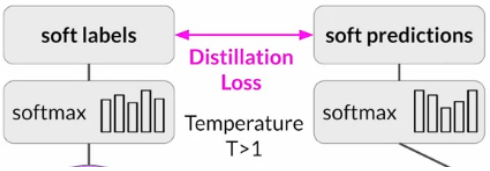
Now, the teacher model is already fine-tuned to the training data. So, the probability distribution likely closely matches the ground truth data and won't have much variation in tokens.



That's why distillation applies a little trick by adding a temperature parameter to the softmax function.

A higher temperature increases the creativity of the language the model generates.

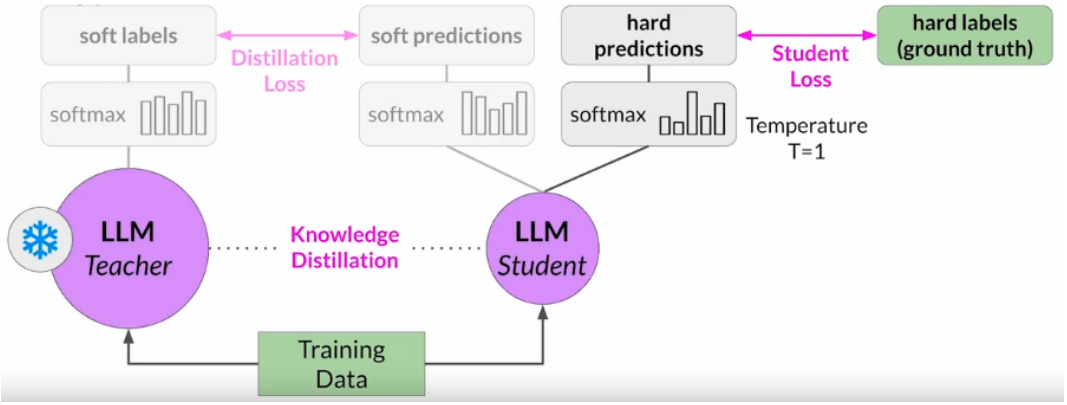
With a temperature parameter greater than one, the probability distribution becomes broader and less strongly peaked.



This softer distribution provides a set of tokens similar to the ground truth tokens.

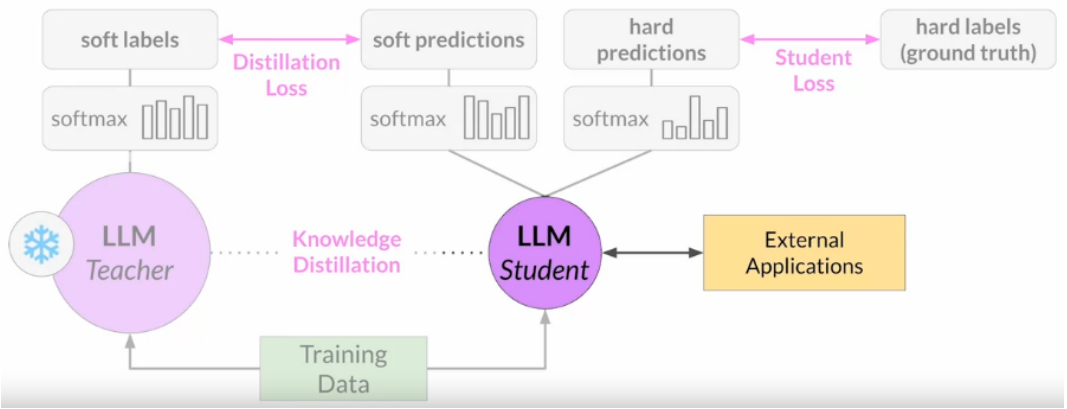
In Distillation, the teacher model's output is often called “soft labels”, and the student model's predictions are “soft predictions”.

In parallel, you train the student model to generate the correct predictions based on your ground truth training data. Here, you don't have to change the temperature setting; instead, use the standard softmax function.



Distillation refers to the student model outputs as the “hard predictions” and “hard labels.” The difference between these two is the student loss. The combined distillation and student losses are used to update the weights of the student model via backpropagation.

The key benefit of distillation methods is that the smaller student model can be used for inference in deployment instead of the teacher model. In practice, distillation is not as effective for generative decoder models. It's typically more effective for encoder-only models, such as Burt, that have a lot of representation redundancy.

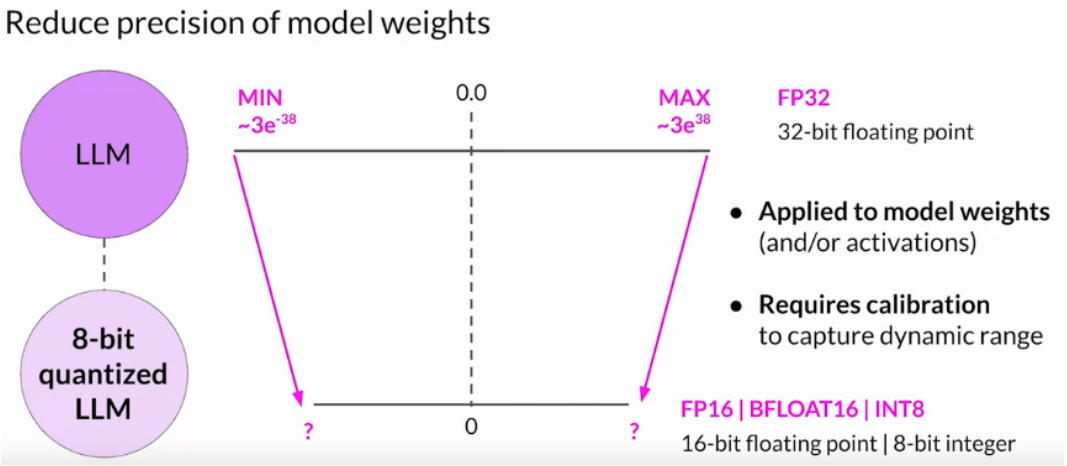


Note: A second smaller model will be trained for inference with distillation. The model size of the initial LLM will be reduced in any way.

▼ Model quantization: Post-Training Quantization (PTQ)

Let's look at the next model optimization technique that reduces the size of your LLM. The second method, quantization, is back in week one in the context of training. Specifically Quantization Aware Training, or QAT for short.

However, after a model is trained, post-training quantization, or PTQ for short, can be optimised for deployment. PTQ transforms a model's weights to a lower-precision representation, such as a 16-bit floating point or 8-bit integer. To reduce the model size, memory footprint, and compute resources needed for model serving, quantization can be applied to just the model weights or both weights and activation layers.

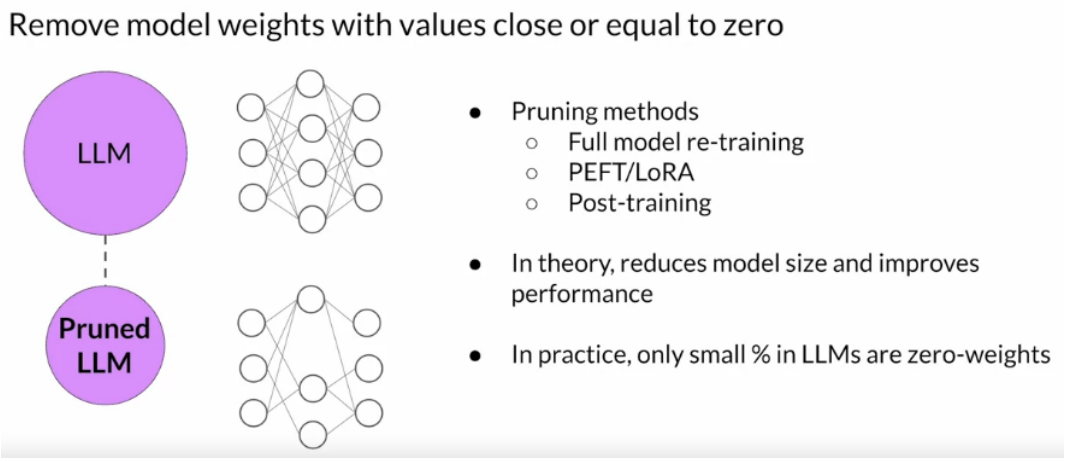


In general, quantization approaches that include the activations can impact model performance more. Quantization also requires an extra calibration step to statistically capture the dynamic range of the original parameter values.

As with other methods, there are tradeoffs because sometimes quantization results in a small percentage reduction in model evaluation metrics. However, that reduction is often worth the cost savings and performance gains.

▼ Model Pruning

The last model optimization technique is pruning. At a high level, the goal is to reduce model size for inference by eliminating weights that do not contribute much to overall model performance. These are the weights with values very close to or equal to zero.



Note: Some pruning methods require full retraining of the model, while others fall into the category of parameter-efficient fine-tuning, such as LoRA. Some methods focus on post-training Pruning. In theory, this reduces the size of the model and improves performance. In practice, however, there may not be much impact on the size and performance if only a small percentage of the model weights are close to zero.

Quantization, Distillation and Pruning aim to reduce model size to improve model performance during inference without impacting accuracy. Optimizing the deployment model will help ensure the application functions well and provide the users with the best possible experience.

▼ Generative AI Project Lifecycle Cheat Sheet

Everything in the course, from selecting the model to fine-tuning and aligning it with human preferences, will happen before the application is deployed. This cheat sheet indicates the time and effort required for each phase of work to plan out these stages of the generative AI project life cycle.

	Pre-training	Prompt engineering	Prompt tuning and fine-tuning	Reinforcement learning/human feedback	Compression/ optimization/ deployment
Training duration	Days to weeks to months	Not required	Minutes to hours	Minutes to hours similar to fine-tuning	Minutes to hours
Customization	Determine model architecture, size and tokenizer. Choose vocabulary size and # of tokens for input/context Large amount of domain training data	No model weights Only prompt customization	Tune for specific tasks Add domain-specific data Update LLM model or adapter weights	Need separate reward model to align with human goals (helpful, honest, harmless) Update LLM model or adapter weights	Reduce model size through model pruning, weight quantization, distillation Smaller size, faster inference
Objective	Next-token prediction	Increase task performance	Increase task performance	Increase alignment with human preferences	Increase inference performance
Expertise	High	Low	Medium	Medium-High	Medium

Cheat Sheet - Time and effort in the lifecycle

Pre-training: Building a large language model can be a huge effort. This stage is the most complex because of the model architecture decisions, the large amount of training data required, and the expertise needed. However, we will generally start the development work with an existing foundation model, so we'll probably be able to skip this stage.

Prompt Engineering: Working with a foundation model, we'll likely assess its performance through prompt engineering, which requires less technical expertise and no additional training in the model.

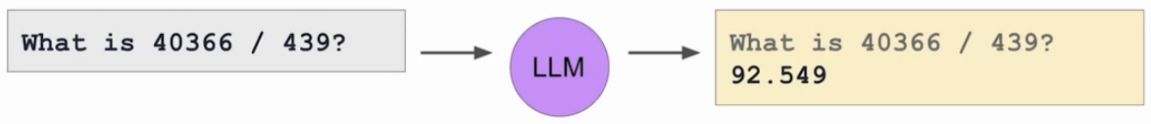
Prompt tuning and fine-tuning: The model isn't performing as needed in this case. Depending on the use case, performance goals, and compute budget, the methods to try could range from full fine-tuning to parameter-efficient fine-tuning techniques like LoRA or prompt tuning. Some level of technical expertise is required for this work. However, since fine-tuning can be very successful with a relatively small training dataset, this phase could potentially be completed in a single day.

Reinforcement learning from human feedback: Aligning the RLHF model can be done quickly with a trained reward model. We'll likely see if we can use an existing reward model for this work. However, training a reward model from scratch could take a long time because of the effort to gather human feedback.

Compression, optimization, deployment: Finally, optimization techniques typically fall in the middle in terms of complexity and effort but can proceed quite quickly, assuming the changes to the model don't impact performance too much. After working through these steps, a great LLM was hopefully trained and tuned to work well for the specific use case and be optimized for deployment.

▼ LinkedIn post

Models can also struggle with complex math. If a model is prompted to behave like a calculator, it may get the answer wrong, depending on the problem's difficulty. Here, we ask the model to carry out a division problem. The model returns a number close to the correct answer, but it's incorrect. Note the LLMs do not carry out mathematical operations. They are still trying to predict the next best token based on their training, so they can easily get the answer wrong.

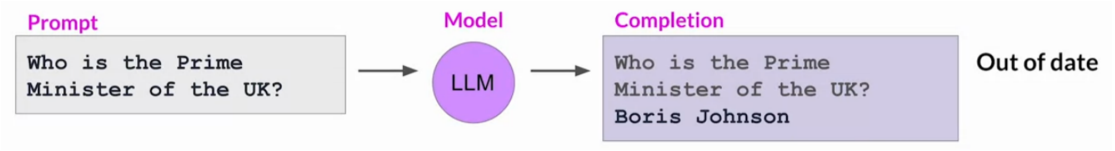


▼ Using the LLM in applications

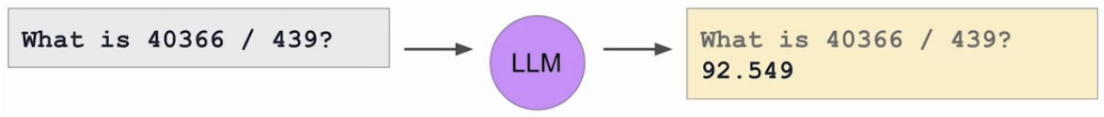
Although all the training, tuning, and aligning techniques can build a great model for the application, there are some broader challenges with large language models that can't be solved by training alone.

▼ Models having difficulty

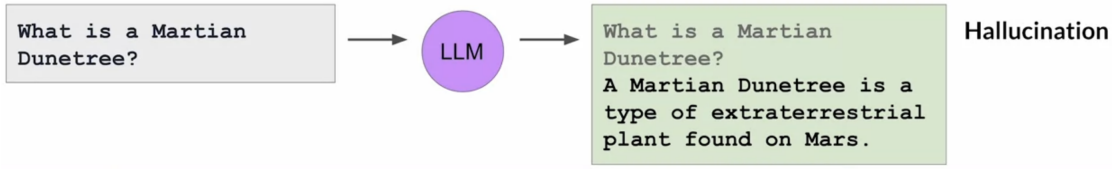
Let's take a look at a few examples. One issue is that a model's internal knowledge cuts off at pre-training. For example, if we ask a model trained in early 2022 who the British Prime Minister is, it will probably tell us Boris Johnson. This knowledge is out of date. The model does not know that Johnson left office in late 2022 because that event happened after its training.



Models can also struggle with complex math. If a model is prompted to behave like a calculator, it may get the answer wrong, depending on the problem's difficulty. Here, we ask the model to carry out a division problem. The model returns a number close to the correct answer, but it's incorrect. Note the LLMs do not carry out mathematical operations. They are still trying to predict the next best token based on their training, so they can easily get the answer wrong.

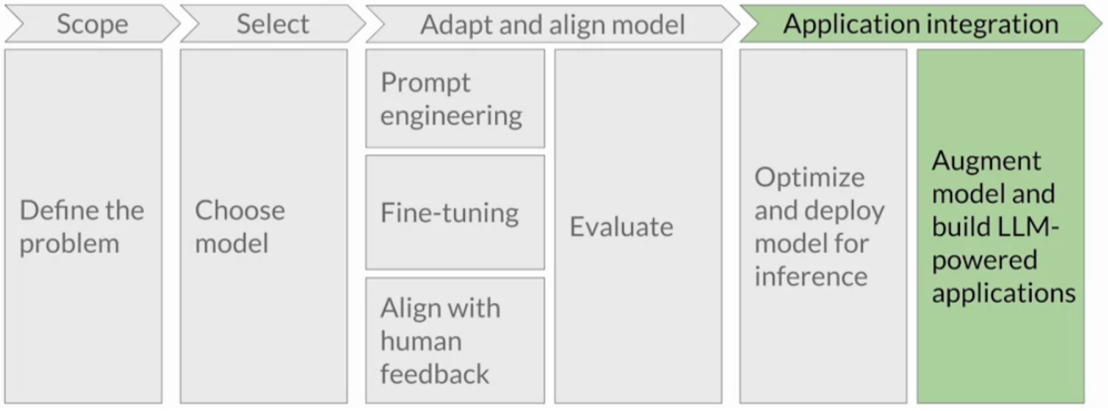


Lastly, one of the best-known problems of LLMs is their tendency to generate text even when they don't know the answer to a problem. This is often called hallucination, and here, the model describes a nonexistent plant, the Martian Dunetree. Although there is no definitive evidence of life on Mars, the model will happily tell you otherwise.

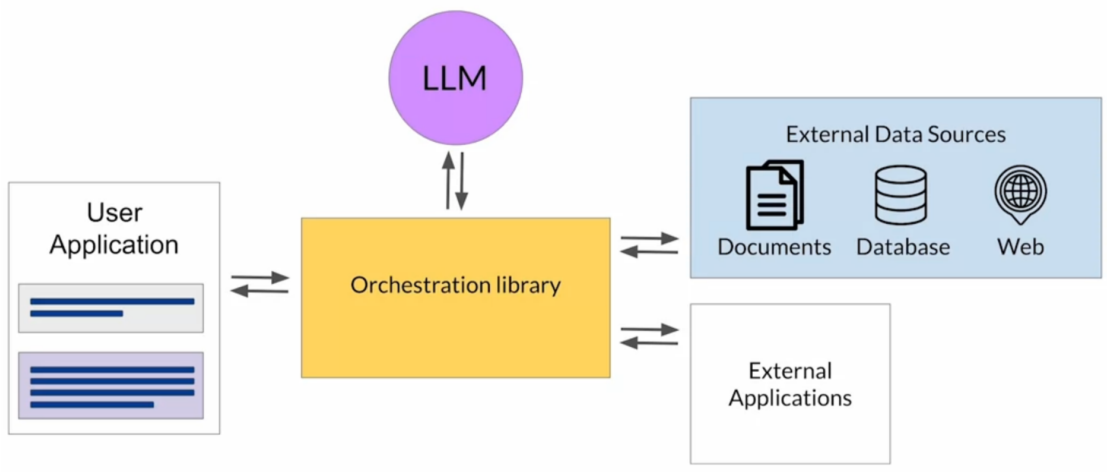


▼ LLM-powered applications

Some techniques, such as connecting to external data sources and applications, can help the LLM overcome these issues.



However, there is more work to do before connecting the LLM to these external components and fully integrating everything for deployment within the application.

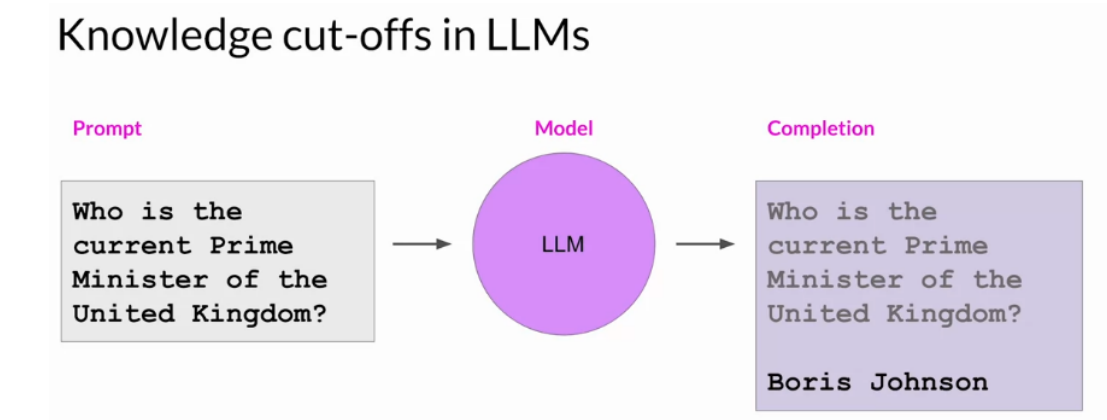


The application must manage the passing of user input to the large language model and the return of completions. This is often done through some orchestration library. This layer can enable some powerful technologies that augment and enhance the performance of the LLM at runtime by providing access to external data sources or connecting to existing APIs of other applications. One implementation example is Langchain.

▼ Retrieval augmented generation (RAG)

Let's start by considering how to connect LLMs to external data sources. Retrieval Augmented Generation, or RAG for short, is a framework for building LLM-powered systems that use external data sources and applications to overcome some of these models' limitations.

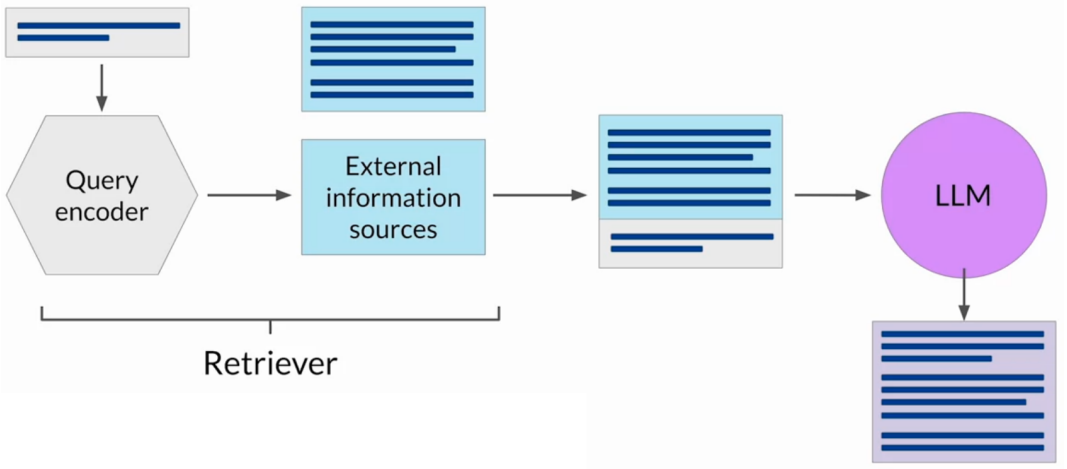
▼ Knowledge cut-offs in LLMs



RAG is a great way to overcome the knowledge cutoff issue and help the model update its understanding of the world. While the model could be retrained on new data, this would quickly become expensive and require repeated retraining to update the model regularly with new knowledge.

▼ Retrieval Augmented Generation (RAG)

A more flexible and less expensive way to overcome knowledge cutoffs is to give the model access to additional external data at inference time. RAG is useful when the language model can access data it may not have seen. This could be new information documents not included in the original training data or proprietary knowledge stored in any organization's private databases. Providing the model with external information can improve its relevance and accuracy in terms of completion.



Lewis et al. 2020 "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks."

Retrieval augmented generation isn't a specific set of technologies but a framework for providing LLMs access to data they did not see during training. Several implementations exist, and the chosen one will depend on the task details and the data format to work with.

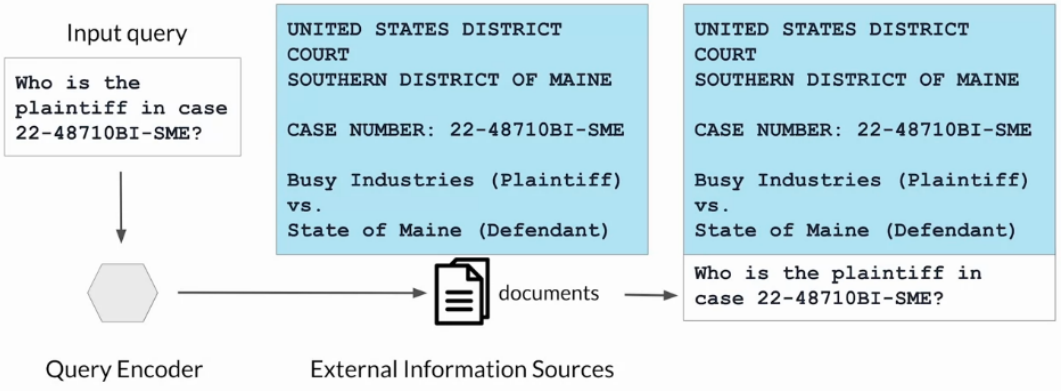
Let's walk through the implementation discussed in one of the earliest papers on RAG by researchers at Facebook, originally published in 2020. At the heart of this implementation is a model component called the Retriever, which consists of a query encoder and an external data source. The encoder takes the user's input prompt and encodes it into a form that can be used to query the data source.

In the Facebook paper, the external data is a vector store. However, it could be an SQL database, CSV files, or other data storage format. These two components are trained to find documents within the external data most relevant to the input query. The Retriever returns the best single or group of documents from the data source and combines the new information with the original user query. The newly expanded prompt is then passed to the language model, generating a completion that uses the data.

▼ Example: Searching for legal documents

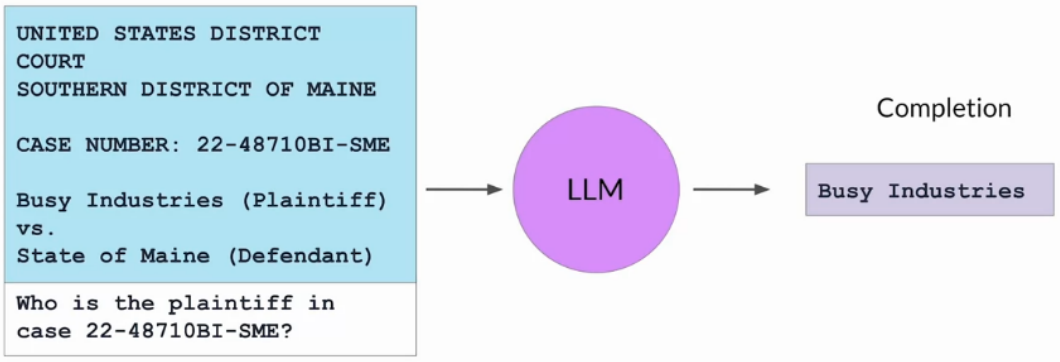
Let's take a look at a more specific example. Imagine a lawyer using a large language model in the discovery phase of a case. A Rag architecture can help ask questions about a corpus of documents, such as previous court filings.

Here, the model is asked about the plaintiff named in a specific case number.



The prompt is passed to the query encoder, which encodes the data in the same format as the external documents. And then searches for a relevant entry in the corpus of documents. Having found a piece of text that contains the requested information, the Retriever then combines the new text with the original prompt.

The expanded prompt containing information about the specific case of interest is then passed to the LLM. The model uses the information in the context of the prompt to generate a completion that contains the correct answer. The use case here is quite simple and only returns a single piece of information that could be found by other means.

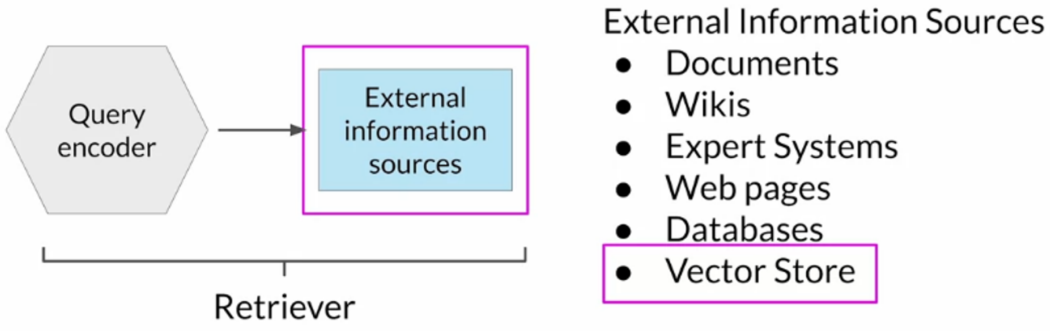


But imagine the power of Rag to generate filing summaries or identify specific people, places and organizations within the full corpus of legal documents. Allowing the model to access information contained in this external data set greatly increases its utility for this specific use case.

In addition to overcoming knowledge cutoffs, RAG helps avoid the problem of the model hallucinating when it doesn't know the answer.

▼ RAG integrates with many types of data sources

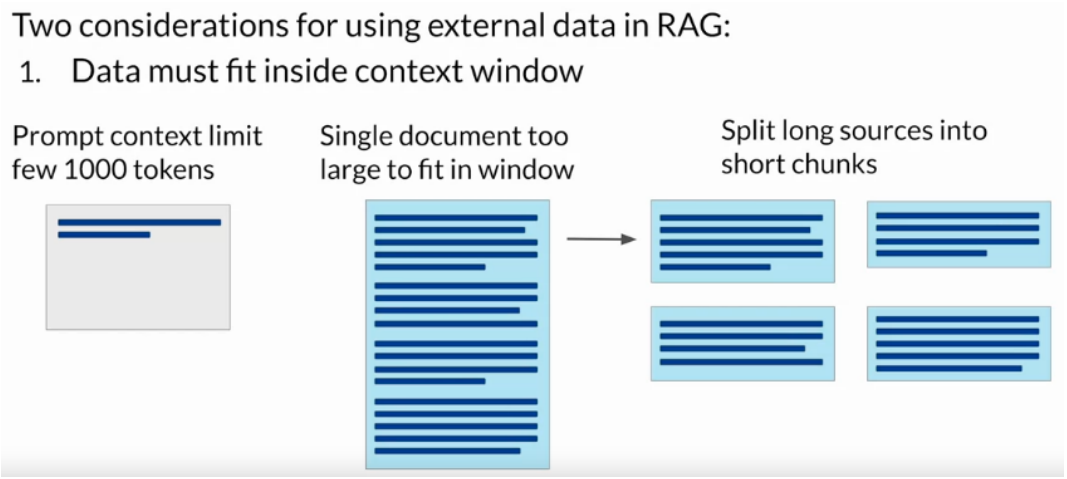
RAG architectures can integrate multiple types of external information sources. Large language models can be augmented with access to local documents, including private wikis and expert systems. Rag can also enable access to the Internet to extract information posted on web pages, such as Wikipedia. By encoding the user input prompt as an SQL query, RAG can also interact with databases.

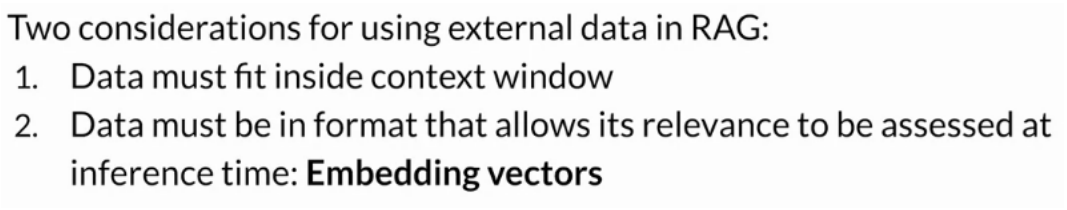


Another important data storage strategy is a Vector Store containing vector text representations. This is a particularly useful data format for language models since they internally work with vector representations of language to generate text. Vector stores enable a fast and efficient kind of relevant search based on similarity.

▼ Data preparation for vector store for RAG

Note: Implementing RAG is slightly more complicated than simply adding text to the large language model. There are a couple of key considerations to be aware of, starting with the context window size. Most text sources are too long to fit into the limited context window of the model, which is still, at most, just a few thousand tokens.

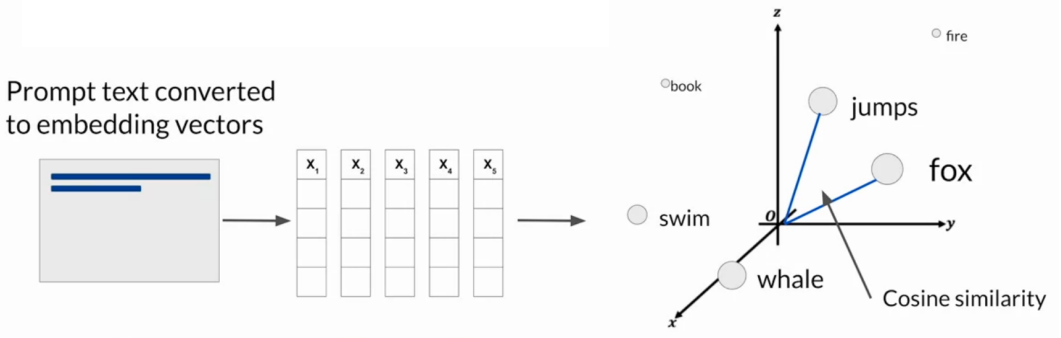




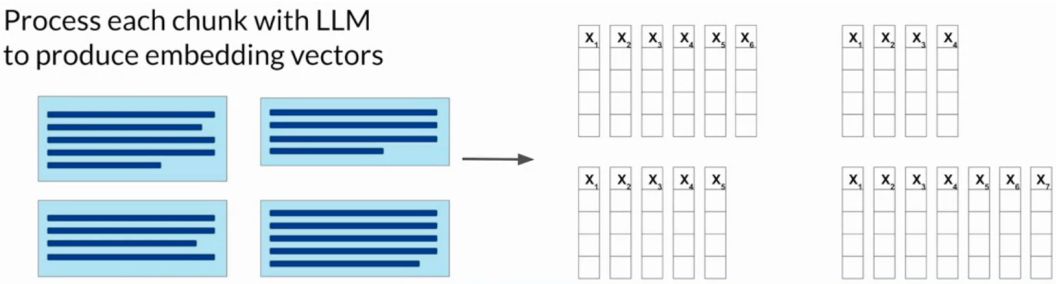
Instead, the external data sources are chopped up into many chunks, each of which will fit in the context window. Packages like Langchain can handle this work.

Second, the data must be available in a format that allows easy retrieval of the most relevant text.

Recall that large language models don't work directly with text but instead create vector representations of each token in an embedding space. These embedding vectors allow the LLM to identify semantically related words through measures such as cosine similarity.

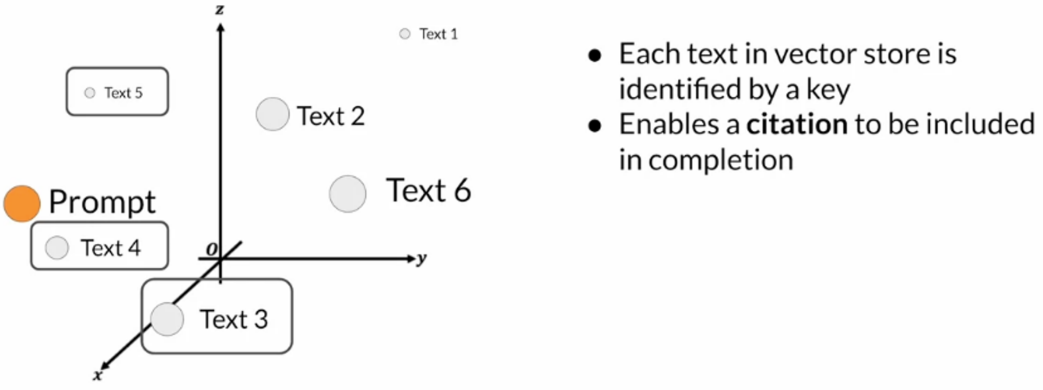


Rag methods take the small chunks of external data and process them through the large language model to create embedding vectors for each. These new representations of the data can be stored in vector stores, allowing for fast searching of datasets and efficient identification of semantically related text.



▼ Vector database search

Vector databases are a particular implementation of a vector store where each vector is also identified by a key. This can allow, for instance, the text generated by RAG to include a citation for the document from which it was received.



Access to external data sources can help a model overcome limits to its internal knowledge. Providing up-to-date, relevant information and

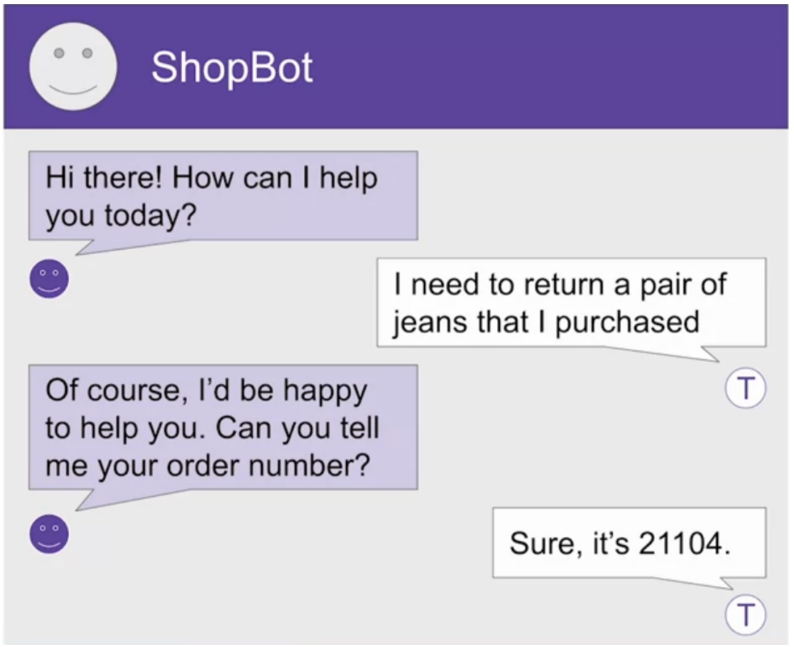
avoiding hallucinations can greatly improve the users' experience of using your application.

▼ Interacting with external applications

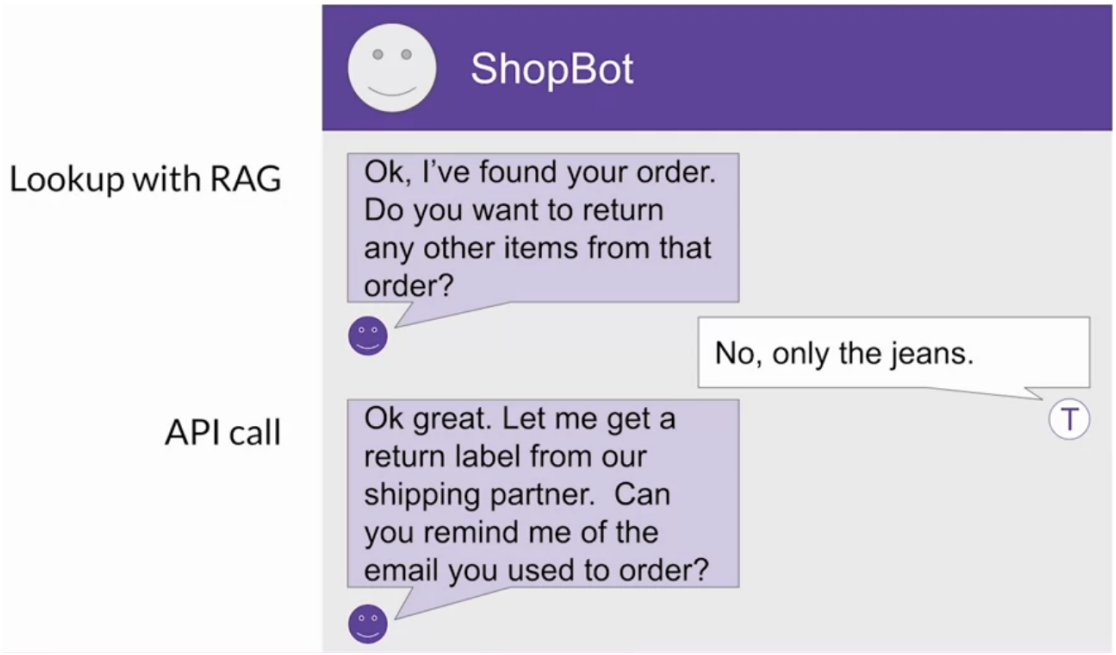
LLMs can interact with external datasets and applications. Let's revisit the customer service bot example to motivate the types of problems and use cases that require this kind of LLM augmentation.

▼ Having an LLM initiate a clothing return

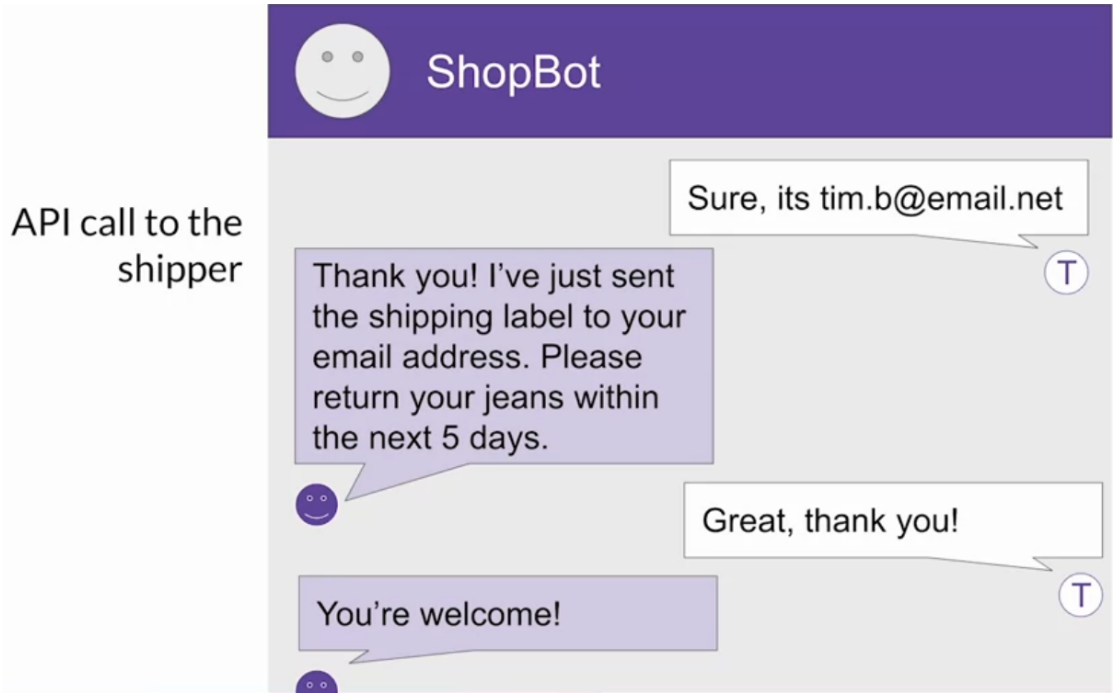
During this walkthrough of one customer's interaction with ShopBot, look at the integrations that allow the app to process return requests from start to finish. In this conversation, the customer wanted to return some of the genes they purchased. ShopBot responded by asking for the order number the customer provided.



ShopBot then looks up the order number in the transaction database. One way it could do this is by using a rag implementation. In this case, we retrieve data through an SQL query to a back-end order database rather than from a corpus of documents.

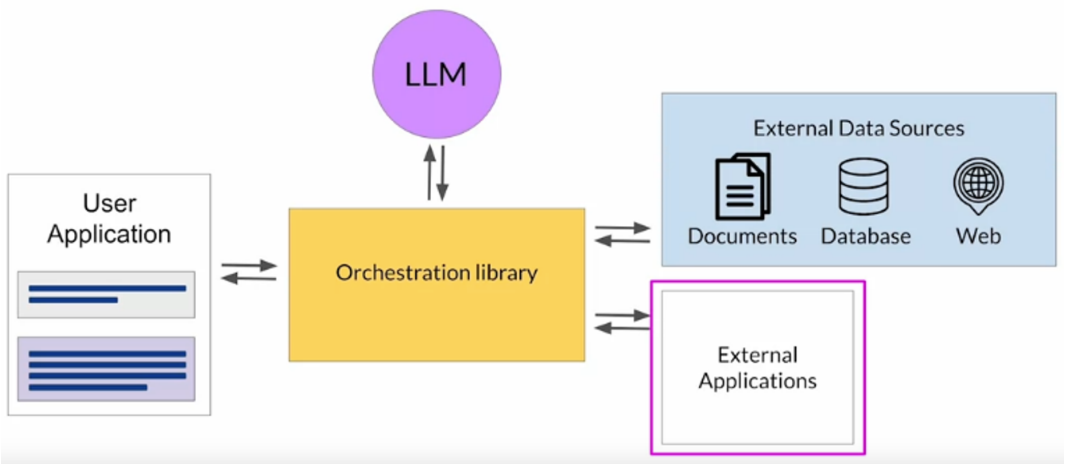


Once ShopBot has retrieved the customer's order, the next step is to confirm the items that will be returned. The bot asks the customer if they'd like to return anything other than the genes. After the user states their answer, the bot requests the company's shipping partner for a return label. The body uses the shipper's Python API to request the label. ShopBot will email the shipping label to the customer and ask them to confirm their email address.



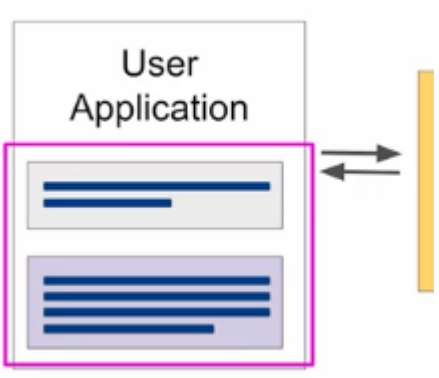
The customer responds with their email address, which the bot includes in the API call to the shipper. Once the API request is completed, Bartlett informs the customer that the label has been sent by email, and the conversation ends.

▼ LLM-powered applications



This example illustrates just one possible set of interactions that need an LLM capable of power and application. Connecting LLMs to external applications allows the model to interact with the broader world, extending their utility beyond language tasks.

As the shopbot example, LLMs can trigger actions when given the ability to interact with APIs. LLMs can also connect to other programming resources. For example, a Python interpreter can enable models to incorporate accurate calculations into their outputs.

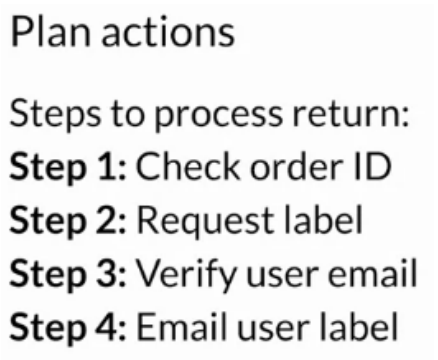


It's important to note that prompts and completions are at the heart of these workflows. The LLM, which serves as the application's

reasoning engine, determines the app's actions in response to user requests.

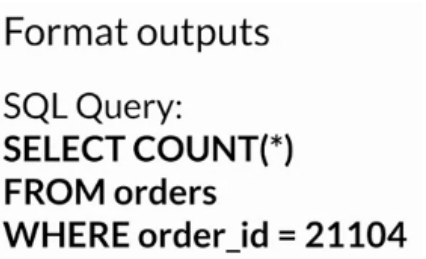
▼ Requirements for using LLMs to power applications

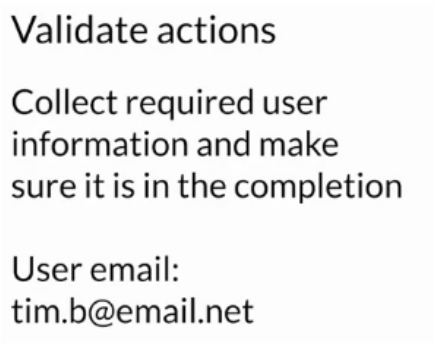
To trigger actions, the completions generated by the LLM must contain certain important information.



First, the model needs to generate a set of instructions so that the application knows what actions to take. These instructions need to be understandable and correspond to allowed actions. In the ShopBot example, the important steps were checking the order ID, requesting a shipping label, verifying the user's email, and emailing the user the label.

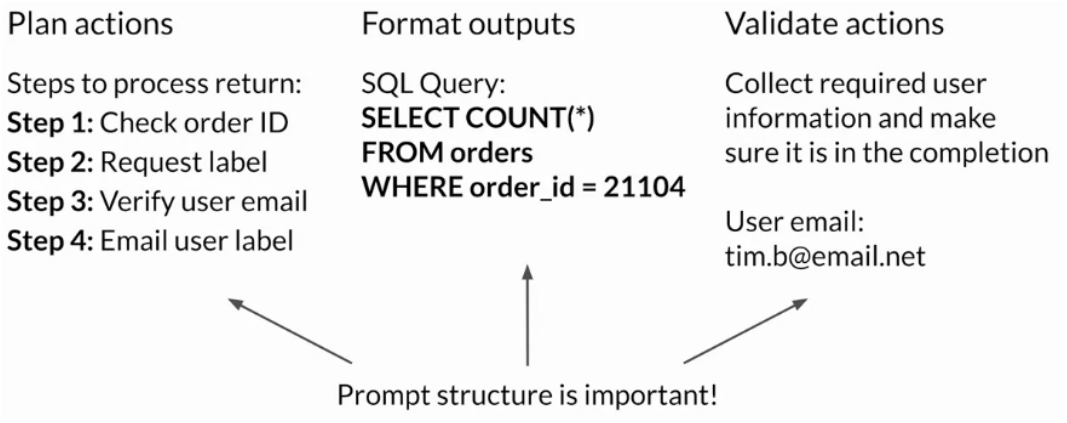
Second, the completion must be formatted so the broader application can understand it. This could be as simple as a specific sentence structure or as complex as writing a script in Python or generating an SQL command. For example, here is an SQL query that would determine whether an order is present in the database of all orders.





Lastly, the model may need to collect information to validate an action. For example, in the ShopBot conversation, the application needed to verify the customer's email address to make the original order.

Any information required for validation must be obtained from the user and contained in the completion to pass it through to the application.

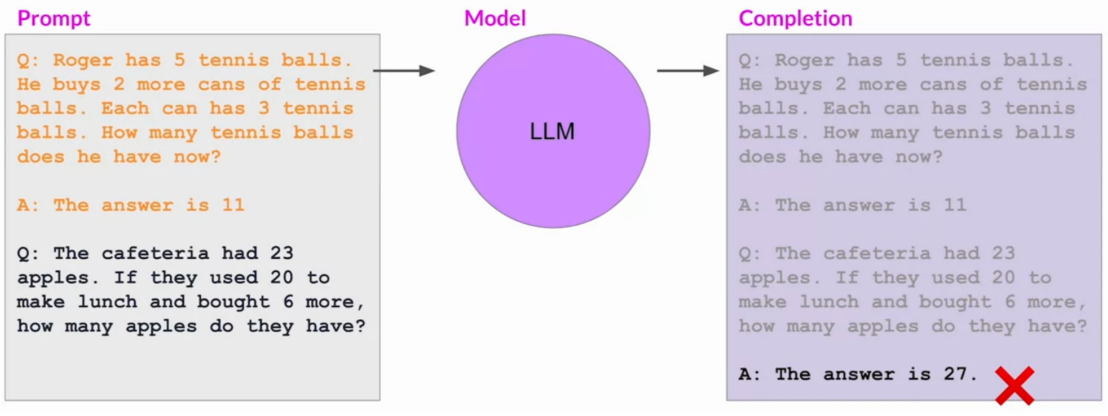


Structuring the prompts correctly is important for all of these tasks and can make a huge difference in the quality of a plan generated or the adherence to a desired output format specification

▼ Helping LLMs reason and plan with chain-of-thought

▼ LLMs can struggle with complex reasoning problems

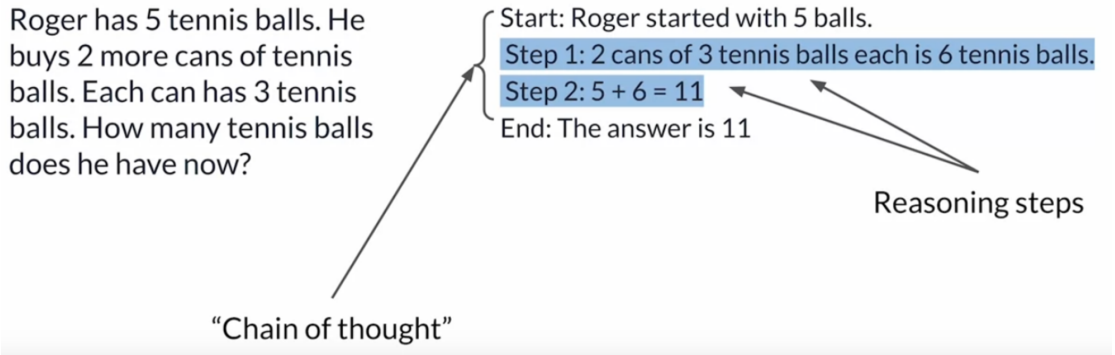
It is important that LLMs can reason through the steps that an application must take to satisfy a user request. Unfortunately, complex reasoning can be challenging for LLMs, especially for problems that involve multiple steps or mathematics. These problems exist even in large models that perform well at many other tasks.



Here's one example where an LLM has difficulty completing the task. The model is being asked to solve a simple multi-step math problem to determine how many apples a cafeteria has after using some to make lunch and then buying some more. The prompt includes a similar example problem, complete with the solution, to help the model understand the task through one-shot inference. After processing the prompt, the model generates the completion shown here, stating that the answer is 27. This answer is incorrect. The cafeteria only has nine apples remaining.

▼ Humans take a step-by-step approach to solving complex problems

Researchers have been exploring ways to improve the performance of large language models on reasoning tasks like the mathematic problem above. One strategy that has demonstrated some success is prompting the model to think more like a human by breaking the problem down into steps.

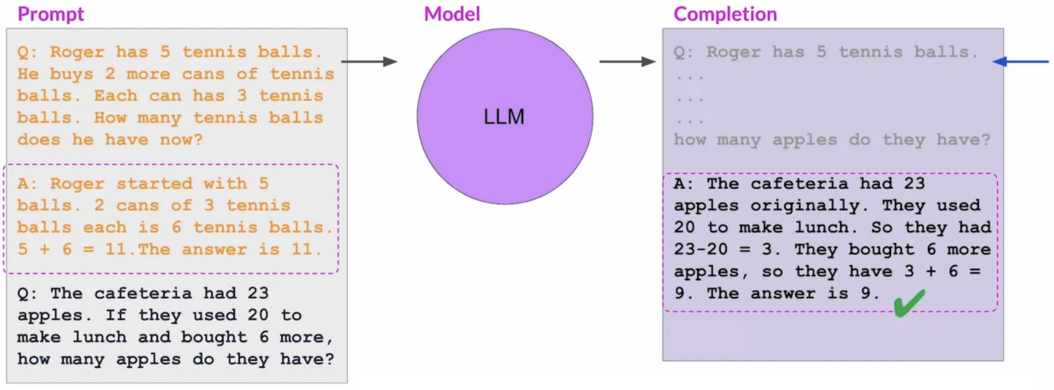


What does it mean to think more like a human? Here is the one-shot example problem from the prompt on the previous slide. The task here is to calculate how many tennis balls Roger has after buying some new ones. One way that a human might tackle this problem is as follows. Begin by determining the number of tennis balls Roger has at the start. Then note that Roger buys two cans of tennis balls. Each can contain three balls, so he has six new tennis balls. Next, add these six new balls to the original 5 for 11 balls. Then, finish by stating the answer. These intermediate calculations form the reasoning steps that a human might take, and the full sequence of steps illustrates the chain of thought that went into solving the problem.

▼ Chain-of-thought prompting can help LLMs reason

▼ Example: Apple problem

Asking the model to mimic this behaviour is called the chain of thought prompting. It includes intermediate reasoning steps into examples for one or few-shot inference. By structuring the examples this way, the model is being trained to reason through the task to reach a solution.

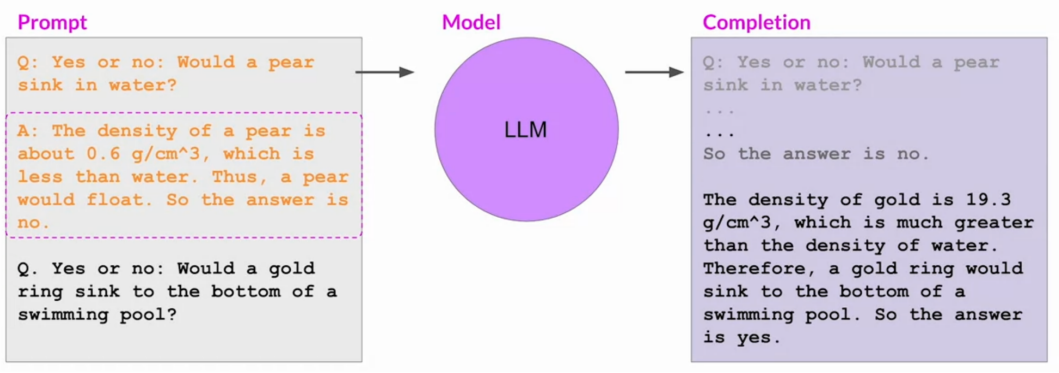


Source: Wei et al. 2022, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models"

Here's the same apple problem, now reworked as a chain of thought prompt. The story of Roger buying the tennis balls is still used as a one-shot example. But this time, the solution text includes intermediate reasoning steps. These steps are equivalent to the ones a human might take. Then, this chain of thought prompt is sent to the large language model, which generates a completion. Notice that the model has now produced a more robust and transparent response that explains its reasoning steps, following a similar structure as the one-shot example. The model now correctly determines that nine apples are left. Thinking through the problem has helped the model come to the correct answer. One thing to note is that while the input prompt is shown here in a condensed format to save space, the entire prompt is included in the output.

▼ Example: Physics problem

You can use chain-of-thought prompting to help LLMs improve their reasoning about other types of problems in addition to arithmetic.



Source: Wei et al. 2022, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models"

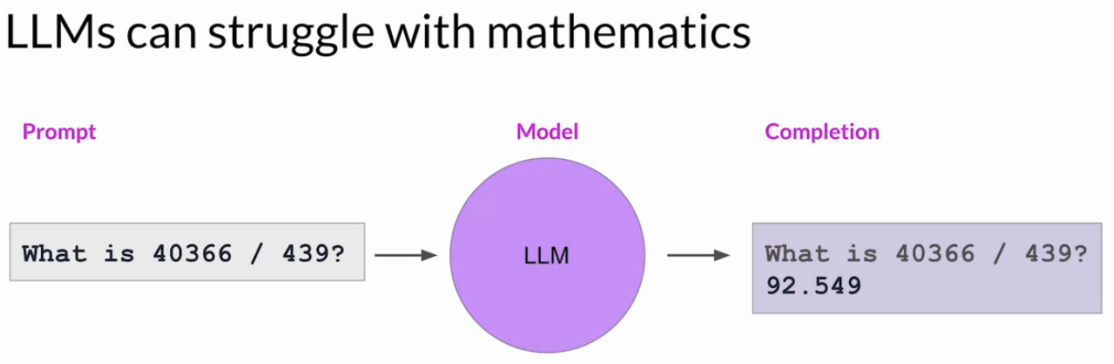
Here's an example of a simple physics problem where the model is asked to determine if a gold ring would sink to the bottom of a swimming pool. The chain of thought prompt, included as the one-shot example, shows the model how to work through this problem by reasoning that a pair would flow because it's less dense than water. Passing this prompt to the LLM generates a similarly structured completion. The model correctly identifies the density of gold, which it learned from its training data and then reasons that the ring would sink because gold is much denser than water.

Chain-of-thought prompting is a powerful technique that improves the model's ability to reason through problems. While this can greatly improve the model's performance, LLMs' limited math skills can still cause problems if the task requires accurate calculations, like totalling sales on an e-commerce site, calculating tax, or applying a discount.

▼ Program-aided language models (PAL)

▼ LLMs can struggle with mathematics

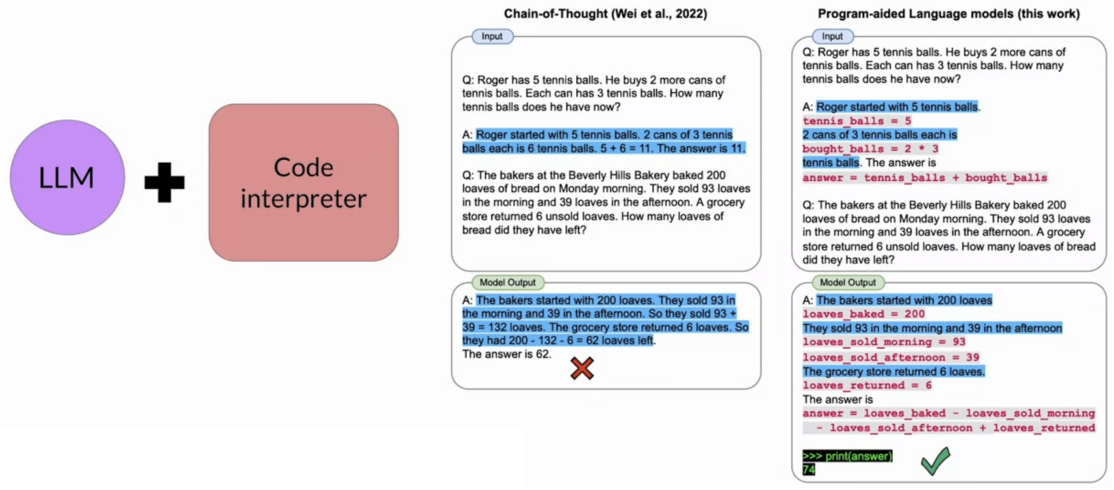
LLMs' ability to carry out arithmetic and other mathematical operations is limited. While using chain-of-thought prompting can overcome this, even if the model correctly reasons through a problem, it may still get the individual math operations wrong, especially with larger numbers or complex operations.



Here's an example of the LLM acting like a calculator but getting the answer wrong. Remember, the model isn't doing any real math here. It simply tries to predict the most probable tokens that complete the prompt. The model getting math wrong can have many negative consequences, depending on the use case, like charging customers the wrong total or getting incorrect measurements for a recipe.

▼ Program-aided language (PAL) models

This limitation can be overcome by allowing the model to interact with external applications that are good at math, like a Python interpreter. One interesting framework for augmenting LLMs in this way is called program-aided language models, or PAL for short.



Source: Gao et al. 2022, "PAL: Program-aided Language Models"

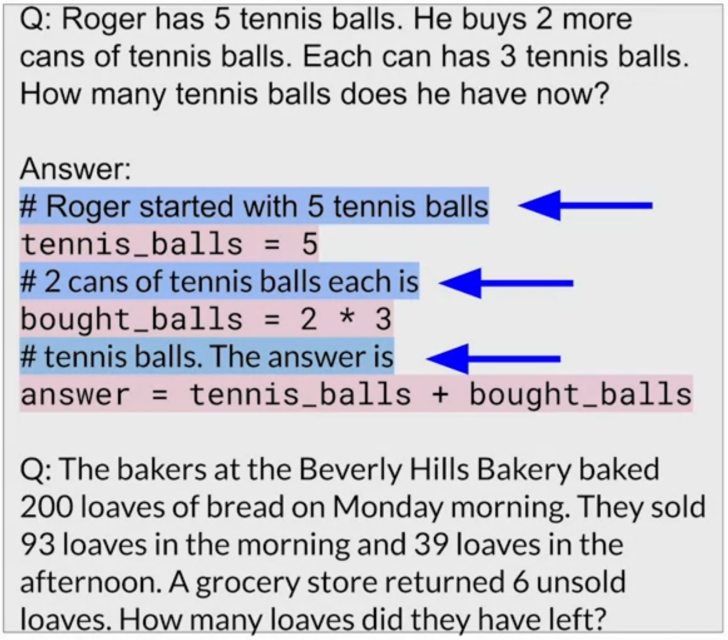
This work was first presented by Luyu Gao and collaborators at Carnegie Mellon University in 2022. It pairs an LLM with an external code interpreter to calculate. The method uses the chain of thought prompting to generate executable Python scripts. The scripts that the model generates are passed to an interpreter to execute. The image on the right

is taken from the paper and shows some example prompts and completions.

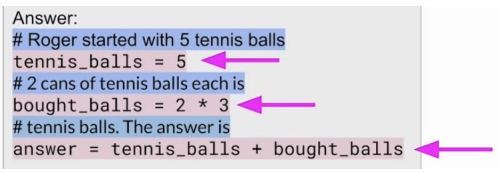
PAL example

Prompt with a one-shot example

The strategy behind PAL is to have the LLM generate completions where reasoning steps are accompanied by computer code. This code is then passed to an interpreter to carry out the calculations necessary to solve the problem. The output format for the model is specified by including examples for one or a few shot inferences in the prompt.



The story of Roger buying tennis balls is a one-shot example. The setup here should now look familiar. This is a chain of thought example. The reasoning steps are written out in words on the lines highlighted in blue.

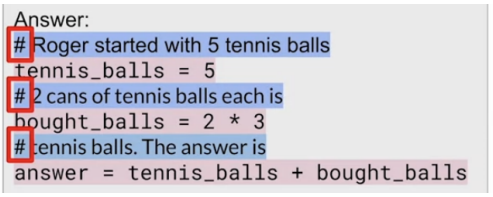


What differs from the prompts before is the inclusion of lines of Python code shown in pink.

These lines translate any reasoning steps that involve calculations into code. Variables are declared based on the text in each reasoning

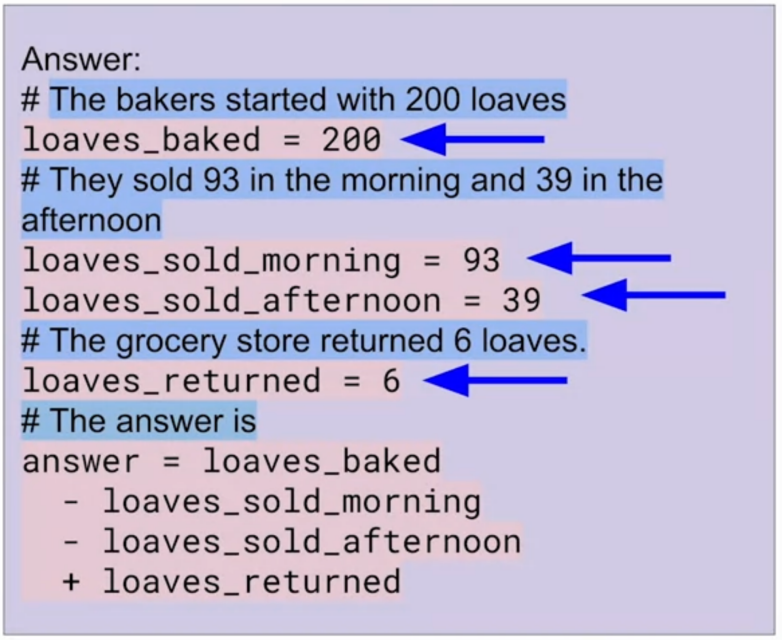
step. Their values are assigned directly, as in the first line of code here, or as calculations using numbers in the reasoning text in the second Python line. The model can also work with variables it creates in other steps, as seen in the third line.

The text of each reasoning step begins with a pound sign so that the line can be skipped as a comment by the Python interpreter.



The prompt here ends with a new problem to be solved. In this case, the objective is to determine how many loaves of bread a bakery has left after a day of sales and after some loaves are returned from a grocery store partner.

▼ Completion, CoT reasoning (blue), and PAL execution (pink)



In this second example, the completion generated by the LLM. Again, the chain of thought reasoning steps are shown in blue and the Python code is shown in pink. The model creates a number of variables to track the loaves baked, the loaves sold in each part of the day, and the loaves returned by the grocery store.

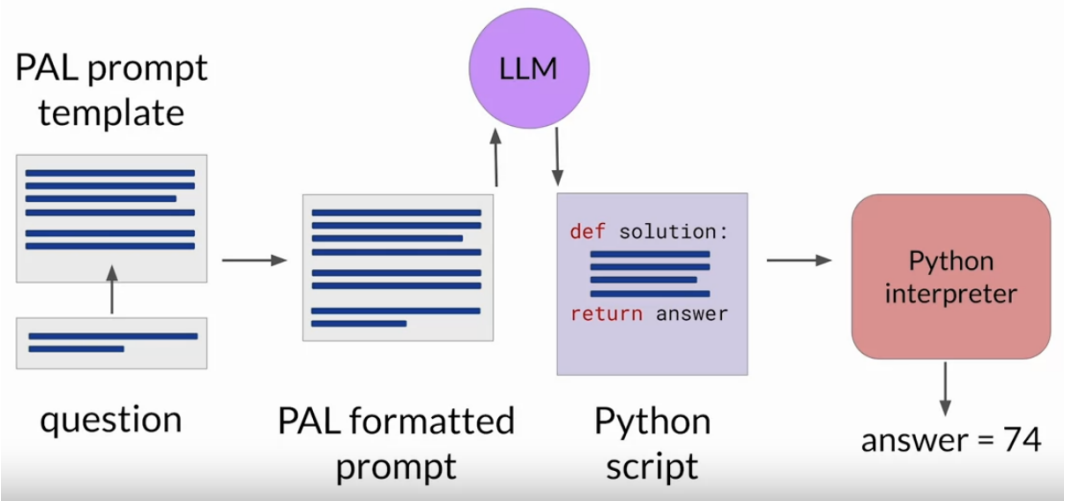
```
answer = loaves_baked
  - loaves_sold_morning
  - loaves_sold_afternoon
  + loaves_returned
```

The answer is then calculated by carrying out arithmetic operations on these variables.

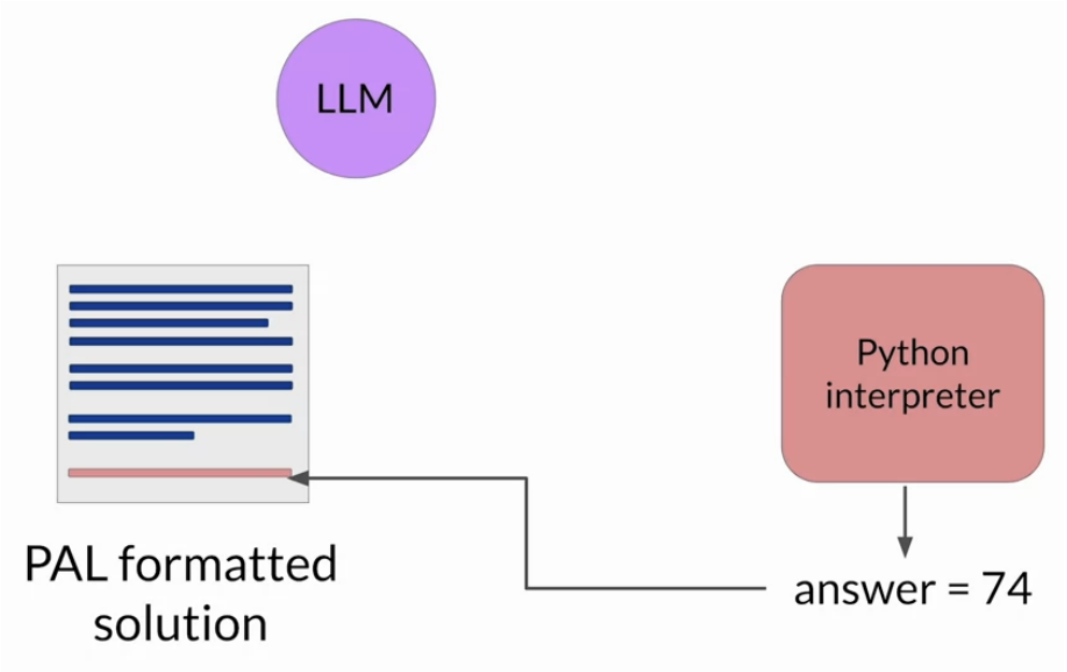
The model correctly identifies whether terms should be added or subtracted to reach the correct total. Those are how to structure examples that will tell the LLM to write Python scripts based on its reasoning steps.

▼ Program-aided language (PAL) models

How the PAL framework enables an LLM to interact with an external interpreter. To prepare for inference with PAL, the prompt needs to be formatted to contain one or more examples. Each example should contain a question followed by reasoning steps in Python code that solve the problem.



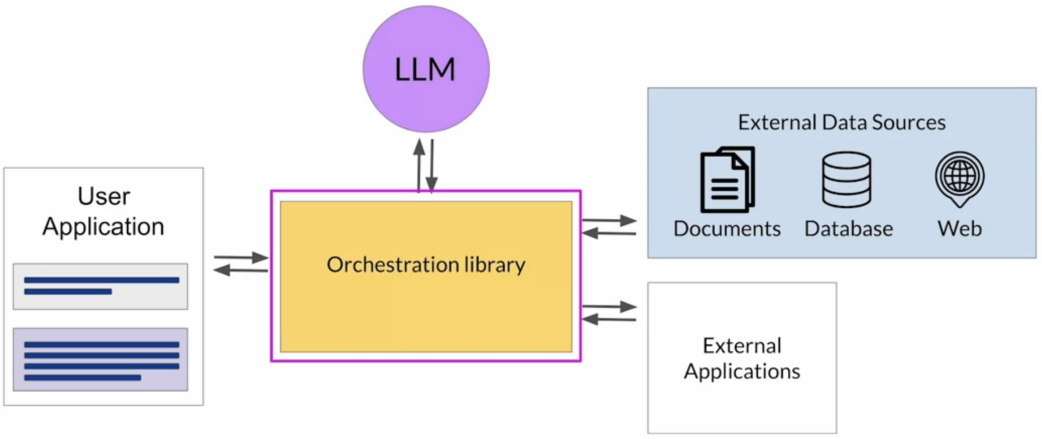
Next, append the new question that needs to be answered to the prompt template. The resulting PAL formatted prompt now contains both the example and the problem to solve. Then, pass this combined prompt to the LLM, which generates a completion in the form of a Python script after learning how to format the output based on the example in the prompt. Now, hand off the script to a Python interpreter, which will be used to run the code and generate an answer. For the bakery example script in the previous section, the answer is 74.



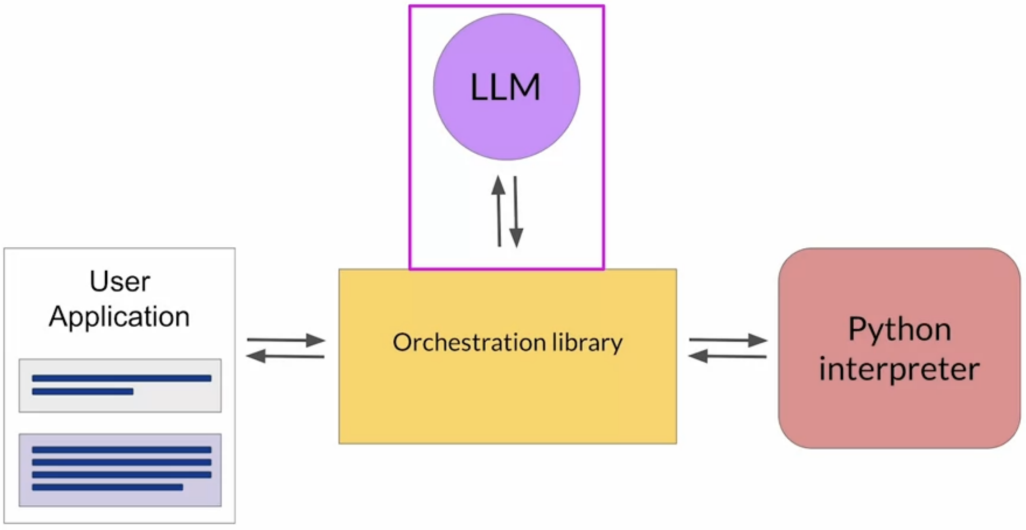
Now, append the text containing the answer, which is accurate because the calculation was carried out in Python to the PAL formatted prompt. By this point, the prompt had already included the correct answer in context. When passing the updated prompt to the LLM, it generates a completion containing the correct answer. Given the relatively simple math in the bakery bread problem, the model may have gotten the answer correct with the chain of thought prompting.

But for more complex math, including arithmetic with large numbers, trigonometry, or calculus, PAL is a powerful technique that ensures that the application's calculations are accurate and reliable.

Then, how can we automate this process so that we don't have to pass information back and forth between the LLM and the interpreter by hand? This is where the orchestrator comes in.



The orchestrator, shown here as the yellow box, is a technical component that can manage the flow of information and the initiation of calls to external data sources or applications. It can also decide what actions to take based on the information contained in the output of the LLM.



PAL architecture

The LLM is the application's reasoning engine. Ultimately, it creates the plan that the orchestrator will interpret and execute. In PAL, there's only one action to be carried out, which is the execution of Python code. The LLM doesn't have to decide to run the code, it just has to write the script which the orchestrator then passes to the external interpreter to run.

However, most real-world applications will likely be more complicated than the simple PAL architecture. The use case may require

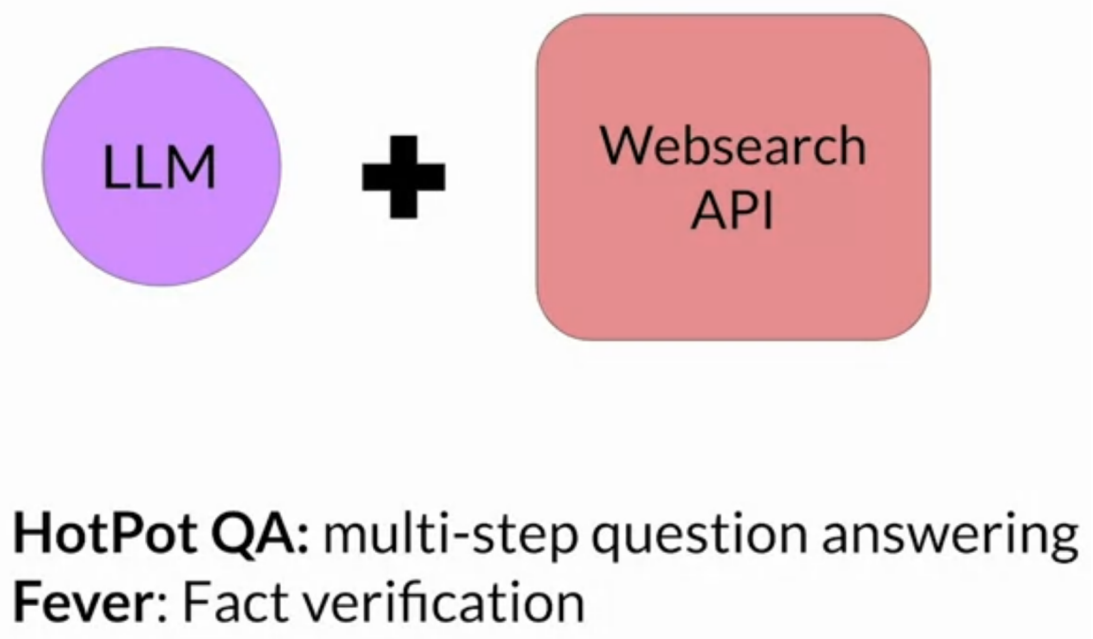
interactions with several external data sources. In the shop-bought example, we may need to manage multiple decision points, validation actions, and calls to external applications.

▼ ReAct: Combining reasoning and action

The structured prompts can help an LLM write Python scripts to solve complex math problems. A PAL application can link the LLM to a Python interpreter to run the code and return the answer to the LLM. Most applications will require the LLM to manage more complex workflows, perhaps including interactions with multiple external data sources and applications. The framework called ReAct can help LLMs plan out and execute these workflows.

▼ ReAct: Synergizing Reasoning and Action in LLMs

ReAct is a prompting strategy that combines chain-of-thought reasoning with action planning. Researchers at Princeton and Google proposed the framework in 2022.



Source: Yao et al. 2022, "ReAct: Synergizing Reasoning and Acting in Language Models”

The paper develops a series of complex prompting examples based on problems from Hot Pot QA, a multi-step question-answering benchmark. That requires reasoning over two or more Wikipedia passages and fever, a benchmark that uses Wikipedia passages to verify facts.

This figure shows some example prompts from the paper.

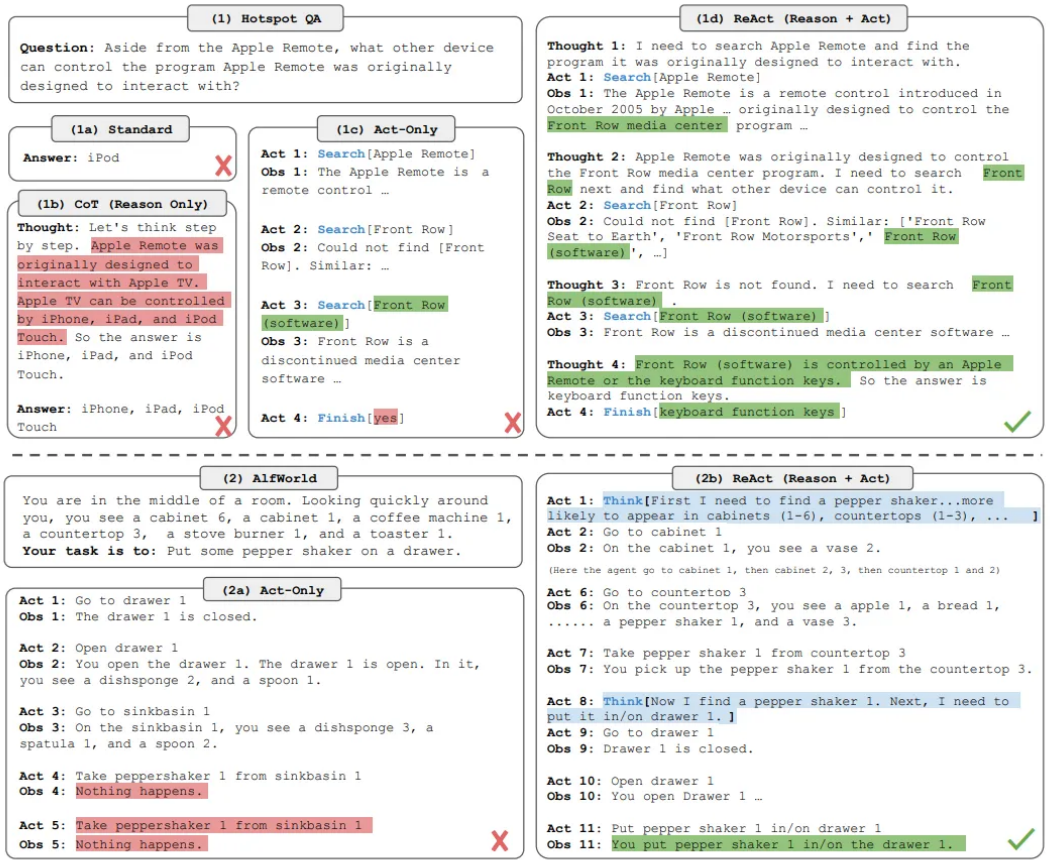
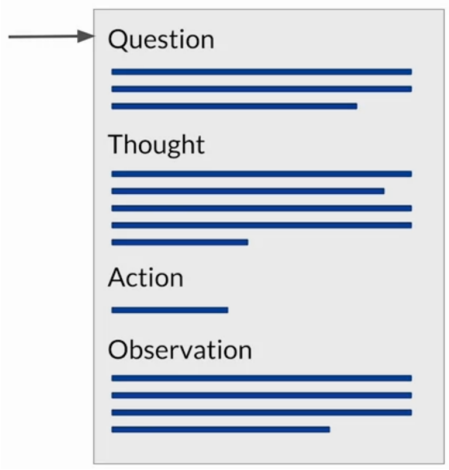


Figure 1: (1) Comparison of 4 prompting methods, (a) `Standard`, (b) Chain-of-thought (`CoT`, Reason Only), (c) `Act`-only, and (d) `ReAct` (Reason+Act), solving a HotpotQA (Yang et al., 2018) question. (2) Comparison of (a) `Act`-only and (b) `ReAct` prompting to solve an AlfWorld (Shridhar et al., 2020b) game. In both domains, we omit in-context examples in the prompt, and only show task solving trajectories generated by the model (Act, Thought) and the environment (Obs).

ReAct uses structured examples to show a large language reasoning model through a problem and decide on actions to move it closer to a solution. The example prompts start with a question that will require multiple steps to answer.



Question: Problem that requires advanced reasoning and multiple steps to solve.

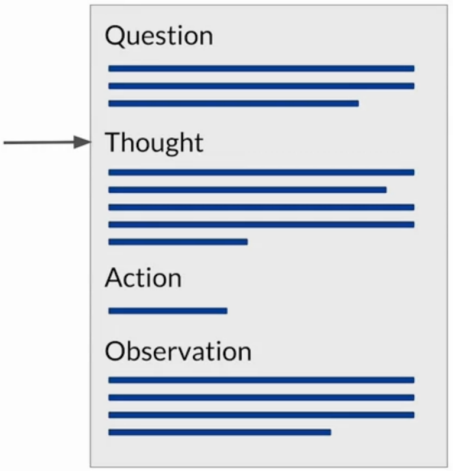
E.g.
"Which magazine was started first, *Arthur's Magazine* or *First for Women*?"

Source: Yao et al. 2022, "ReAct: Synergizing Reasoning and Acting in Language Models"

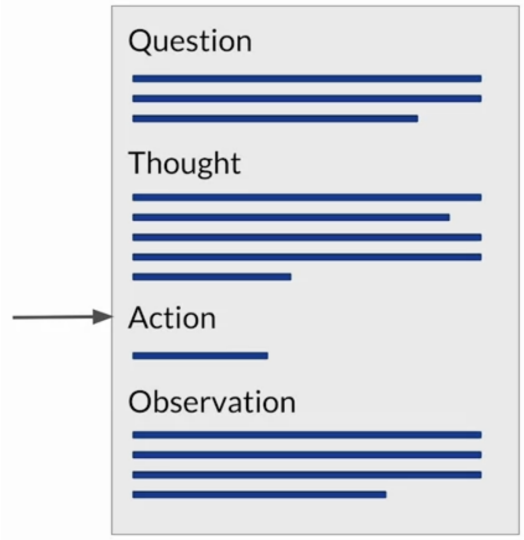
In this example, the goal is to determine which two magazines were created first. The example then includes a related thought action observation trio of strings.

Thought: A reasoning step that identifies how the model will tackle the problem and identify an action to take.

"I need to search Arthur's Magazine and First for Women, and find which one was started first."



The thought is a reasoning step that demonstrates to the model how to tackle the problem and identify an action to take. In the example, the prompt specifies that the model will search for both magazines and determine which one was published first.



Action: An external task that the model can carry out from an allowed set of actions.

E.g.
`search[entity]`
`lookup[string]`
`finish[answer]`

Which one to choose is determined by the information in the preceding thought.

```
search[Arthur's Magazine]
```

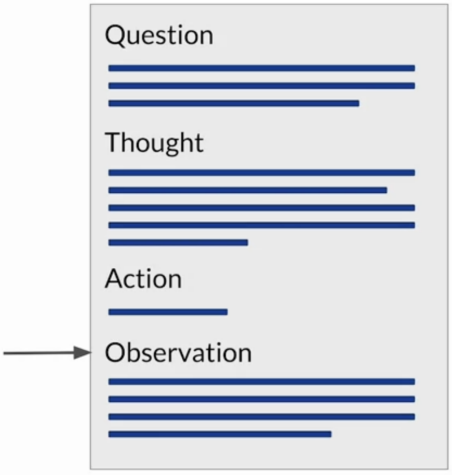
For the model to interact with an external application or data source, it has to identify an action to take from a pre-determined list.

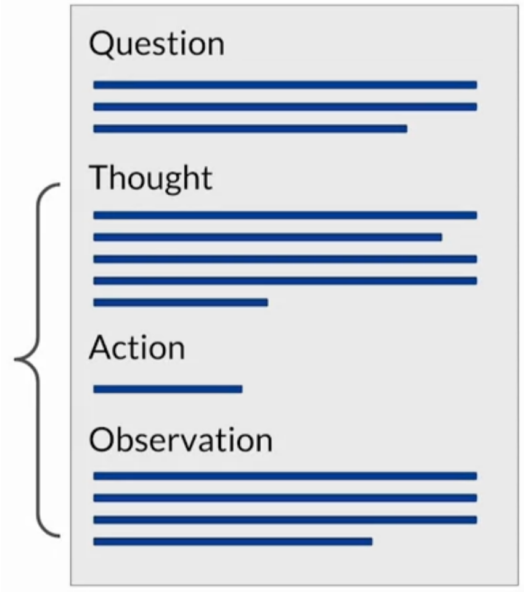
In the case of the ReAct framework, the authors created a small Python API to interact with Wikipedia. The three allowed actions are search, which looks for a Wikipedia entry about a particular topic lookup, which searches for a string on a Wikipedia page. And finish, which the model carries out when it decides it has determined the answer. As you saw on the previous slide, the thought in the prompt identified two searches to carry out, one for each magazine. In this example, the first search will be for Arthur's magazine. The action is formatted using the specific square bracket notation, so the model will format its completions similarly. The Python interpreter searches for this code to trigger specific API actions.

Observation: the result of carrying out the action

E.g.
"Arthur's Magazine (1844-1846) was an American literary periodical published in Philadelphia in the 19th century."

The last part of the prompt template is the observation. This is where the new information provided by the external search is brought into the context of the prompt for the model to interpret.





Thought 2:
"Arthur's magazine was started in 1844. I need to search First for Women next."

Action 2:
`search[First for Women]`

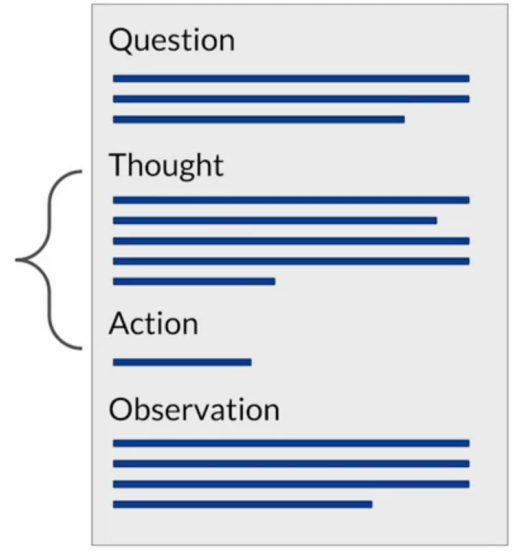
Observation 2:
"First for Women is a woman's magazine published by Bauer Media Group in the USA.[1] The magazine was started in 1989."

The prompt repeats the cycle as many times as necessary to obtain the final answer.

On second thought, the prompt states the start year of Arthur's magazine and identifies the next step needed to solve the problem. The second action is to search for women, and the second observation includes text that states the start date of the publication, in this case, 1989. All the information required to answer the question is now known.

Thought 3:
"First for Women was started in 1989. 1844 (Arthur's Magazine) < 1989 (First for Women), so Arthur's Magazine as started first"

Action 3:
`finish[Arthur's Magazine]`



The third thought states the first year of the year for women and then gives the explicit logic used to determine which magazine was published first.

The final action is to finish the cycle and pass the answer back to the user.

It's important to note that in the ReAct framework, the LLM can only choose from a limited number of actions defined by a set of instructions

pre-pended to the example prompt text.

ReAct instructions define the action space

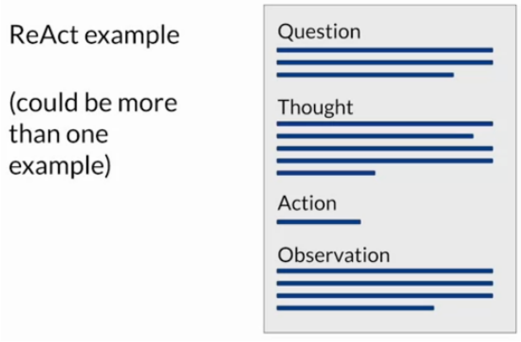
The full text of the instructions is shown here. First, the task is defined, telling the model to answer a question using the prompt structure. Next, the instructions detail what thought means and specify that the action step can only be one of three types.

```
Solve a question answering task with interleaving Thought, Action,
Observation steps.

Thought can reason about the current situation, and Action can be
three types:
(1) Search[entity], which searches the exact entity on Wikipedia and
returns the first paragraph if it exists. If not, it will return
some similar entities to search.
(2) Lookup[keyword], which returns the next sentence containing
keyword in the current passage.
(3) Finish[answer], which returns the answer and finishes the task.
Here are some examples.
```

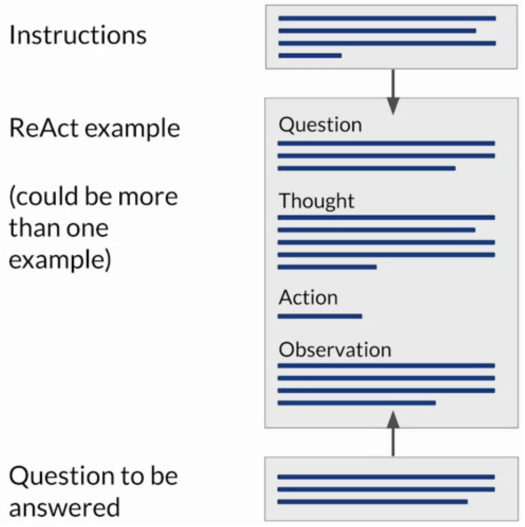
The first is the search action, which looks for Wikipedia entries related to the specified entity. The second is the lookup action, which retrieves the next sentence that contains the specified keyword. The last action is finished, which returns the answer and brings the task to an end—defining a set of allowed actions when using LLMs to plan tasks that will power applications is critical. LLMs are very creative, and they may propose taking steps that don't correspond to something that the application can do. The final sentence in the instructions lets the LLM know that some examples will come next in the prompt text.

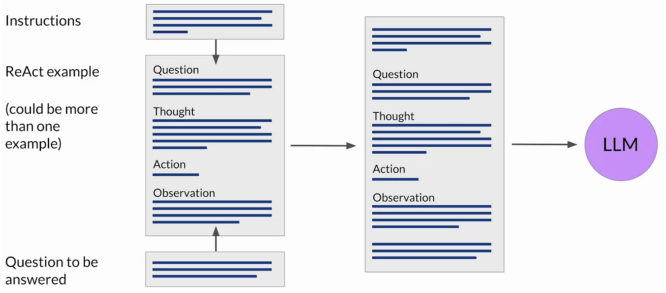
Building up the ReAct prompt



Let's combine all the pieces for inference and start with the ReAct example prompt. Note that depending on the LLM, we can include multiple examples and carry out future inferences.

Next, pre-pend the instructions at the beginning of the example and then insert the question that needs to be answered at the end.



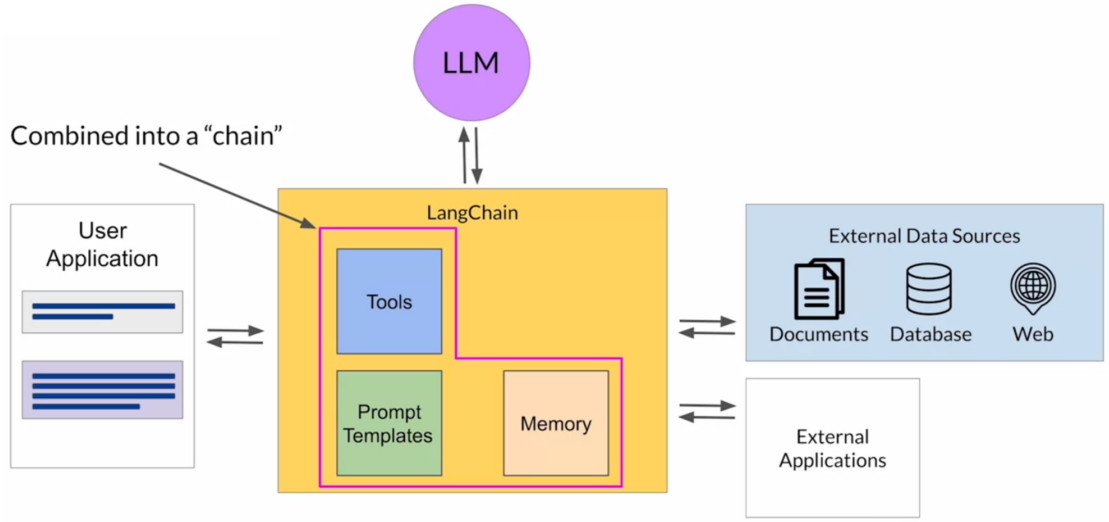


The full prompt now includes these individual pieces, which can be passed to the LLM for inference.

The ReAct framework shows one way to use LLMs to power an application through reasoning and action planning. This strategy can be extended for a specific use case by creating examples that work through the decisions and actions that will take place in the application.

▼ LangChain

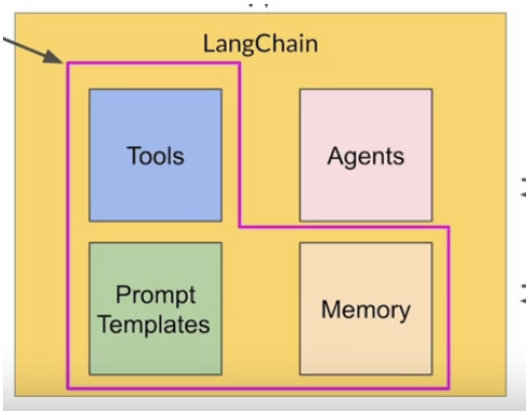
Thankfully, frameworks for developing applications powered by language models are actively developing. One solution that is being widely adopted is LangChain. The LangChain framework provides modular pieces that contain the components necessary to work with LLMs. These components include prompt templates for many different use cases that can be used to format both input examples and model completions. Memory can be used to store interactions with an LLM. The framework also includes pre-built tools for various tasks, including calls to external datasets and APIs.

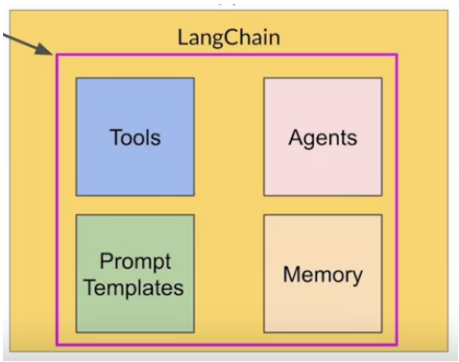


Connecting a selection of these individual components results in a chain. The creators of LangChain have developed a set of predefined chains optimized for different use cases that can be used off the shelf to get the app up and running quickly.

Sometimes, the application workflow could take multiple paths depending on the information the user provides. In this case, a pre-determined chain can’t be used; instead, we'll need the flexibility to decide which actions to take as the user moves through the workflow.

LangChain defines another construct, an agent that can interpret the user's input and determine which tool or tools to use to complete the task. LangChain currently includes agents for PAL and ReAct, among others.



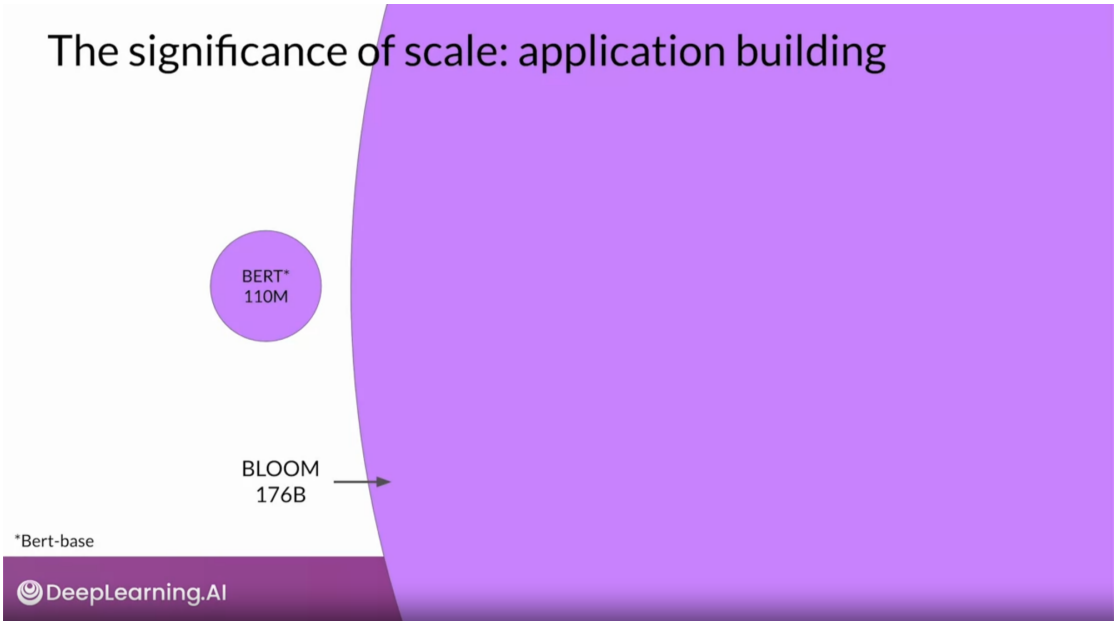


Agents can be incorporated into chains to take an action or plan and execute a series of actions.

LangChain is in active development, and new features are always being added, like the ability to examine and evaluate the LLM's completions throughout the workflow.

It's an exciting framework for fast prototyping and deployment, and it will likely become an important tool in the generative AI toolbox in the future.

The last thing to remember when developing applications using LLMs is that the model's ability to reason well and plan actions depends on its scale.



Larger models are generally the best choice for advanced prompting techniques, like PAL or ReAct. Smaller models may struggle to understand the tasks in highly structured prompts and require additional fine-tuning to improve their ability to reason and plan. This could slow down the development process. Instead, a large, capable model and lots of user data collected during deployment can be used to train and fine-tune a smaller model.

▼ Reading: ReAct: Reasoning and action

This paper introduces ReAct, a novel approach that integrates verbal reasoning and interactive decision-making in large language models (LLMs). While LLMs have excelled in language understanding and decision-making, the combination of reasoning and acting has been neglected. ReAct enables LLMs to generate reasoning traces and task-specific actions, leveraging their synergy. The approach demonstrates superior performance over baselines in various tasks, overcoming issues like hallucination and error propagation.

ReAct outperforms imitation and reinforcement learning methods in interactive decision-making, even with minimal context examples. It enhances performance and improves interpretability, trustworthiness, and diagnosability by allowing humans to distinguish between internal knowledge and external information.

ReAct bridges the gap between reasoning and acting in LLMs, yielding remarkable results across language reasoning and decision-making tasks. By interleaving reasoning traces and actions, ReAct overcomes limitations and outperforms baselines, enhancing model performance and providing interpretability and trustworthiness, empowering users to understand the model's decision-making process.

The figure provides a comprehensive visual comparison of different prompting methods in two distinct domains. The first part of the figure (1a) compares four prompting methods: Standard, Chain-of-thought (CoT, Reason Only), Act-only, and ReAct (Reason+Act) for solving a HotpotQA question. Each method's approach is demonstrated through task-solving trajectories generated by the model (Act, Thought) and the environment (Obs).

The second part of Figure (1b) compares Act-only and ReAct prompting methods to solve an AlfWorld game. In both domains, in-context examples are omitted from the prompt, highlighting the generated trajectories resulting from the model's actions and thoughts and the observations made in the environment. This visual representation enables a clear understanding of the differences and advantages offered by the ReAct paradigm compared to other prompting methods in diverse task-solving scenarios.

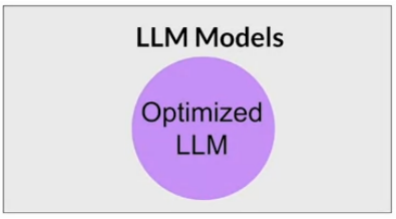
▼ LLM application architectures

There are some additional considerations for building LLM-powered applications.

Building generative applications

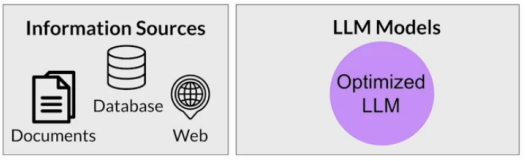
Several key components are required to create end-to-end application solutions, starting with the infrastructure layer. This layer provides the compute, storage, and network to serve the LLMs and host the application components. On-premises infrastructure can be used or provided via on-demand and pay-as-you-go cloud services.

Next, the large language models will be included in the application.



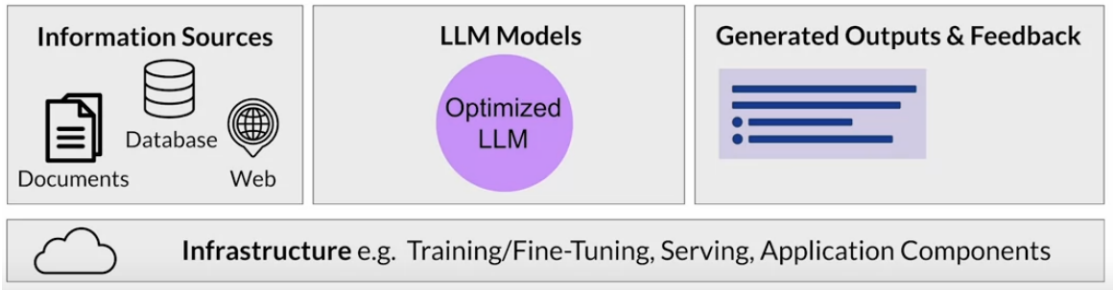
These could include foundation models and the adapted models to a specific task.

The models are deployed on the appropriate infrastructure for the inference needs. Taking into account whether real-time or near-real-time interaction is needed with the model.



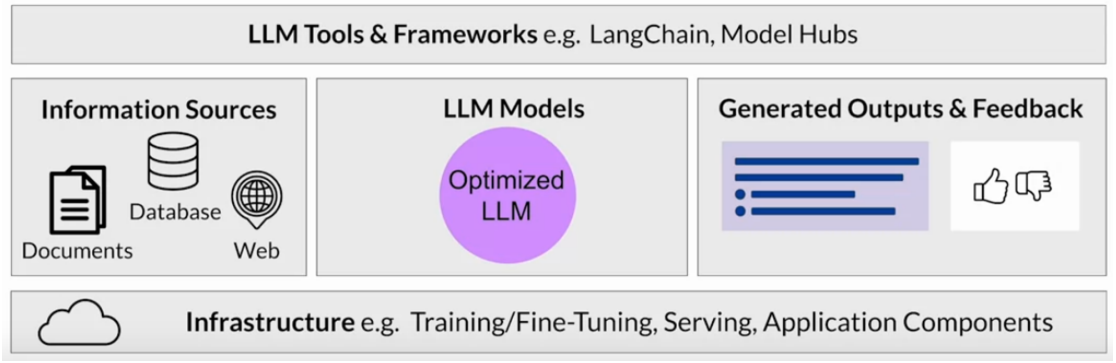
Information from external sources, such as those discussed in the retrieval augmented generation section, may need to be retrieved.

The application will return the completions from the large language model to the user or consuming application. Depending on the use case, a mechanism to capture and store the outputs may need to be implemented.



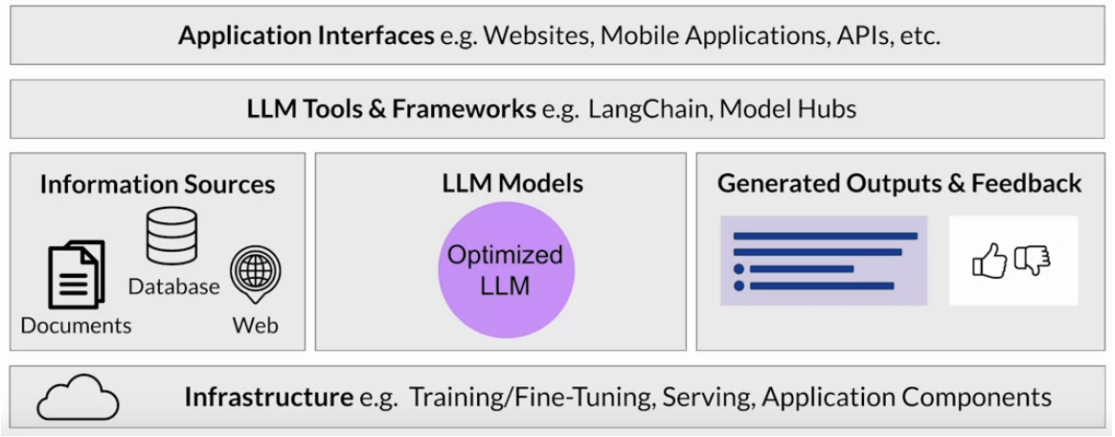
For example, the capacity to store user completions during a session to augment the LLM's fixed context window size could be built. User feedback can also be gathered, which may be useful for additional fine-tuning, alignment, or evaluation as the application matures.

Next, additional tools and frameworks for large language models may need to be used to help easily implement some of the techniques discussed.



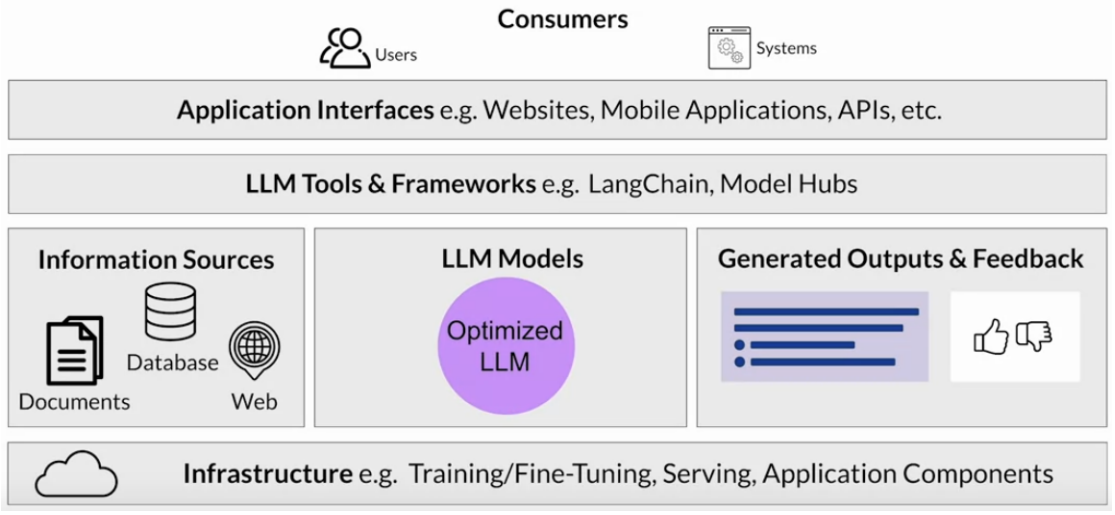
For example, LangChains' built-in libraries can implement techniques like PAL, ReAct, or Chain of Thought Prompting. Model hubs may also be utilized to centrally manage and share application models.

In the final layer, the application will be consumed through various user interfaces, such as a website or a Rest API.



This layer is also included in the security components required for interacting with the application.

At a high level, this architecture stack represents the various components to consider when building generative AI applications. The users will interact with this entire stack, whether human end-users or other systems that access the application through its APIs. The model is typically only one part of the story when building end-to-end generative AI applications.



Some techniques can help align the models with human preferences, such as helpfulness, harmlessness, and honesty, by fine-tuning reinforcement learning with human feedback, or RLHF for short. Given the popularity of RLHF, many existing RL reward models and human alignment datasets can help quickly start aligning the models.

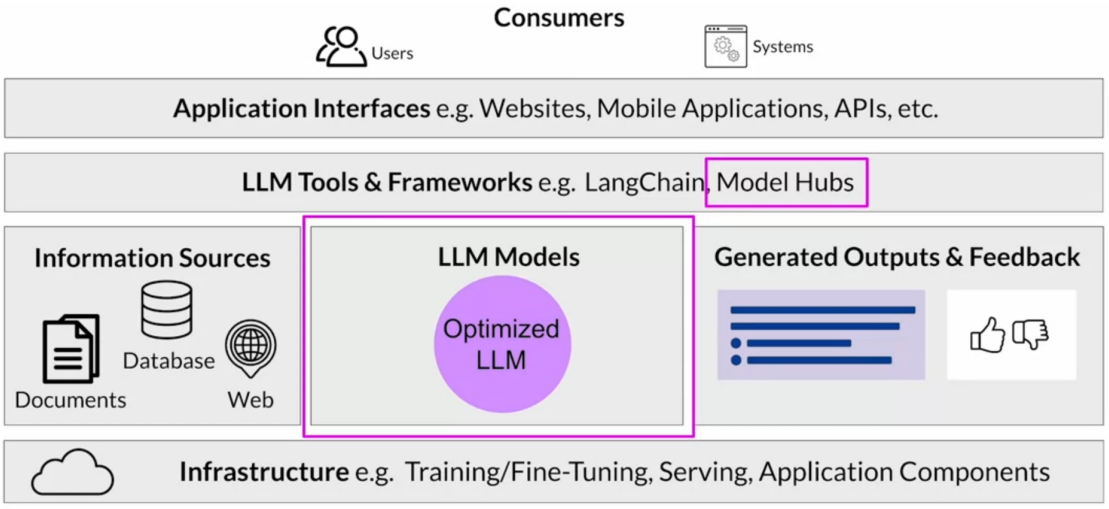
In practice, RLHF is a very effective mechanism that can improve the models' alignment and reduce their responses' toxicity. Thus, the models

can be used more safely in production. Some other important techniques can also optimize the model for inference by reducing its size through distillation, quantization, or pruning. This minimizes the hardware resources needed to serve the LLMs in production.

Lastly, some ways can help the model perform better in deployment through structured prompts and connections to external data sources and applications. LLMs can play an amazing role as the reasoning engine in an application, exploiting their intelligence to power exciting, useful applications. Frameworks like LangChain are making it possible to build, deploy, and test LLM-powered applications quickly, and it's a very exciting time for developers.

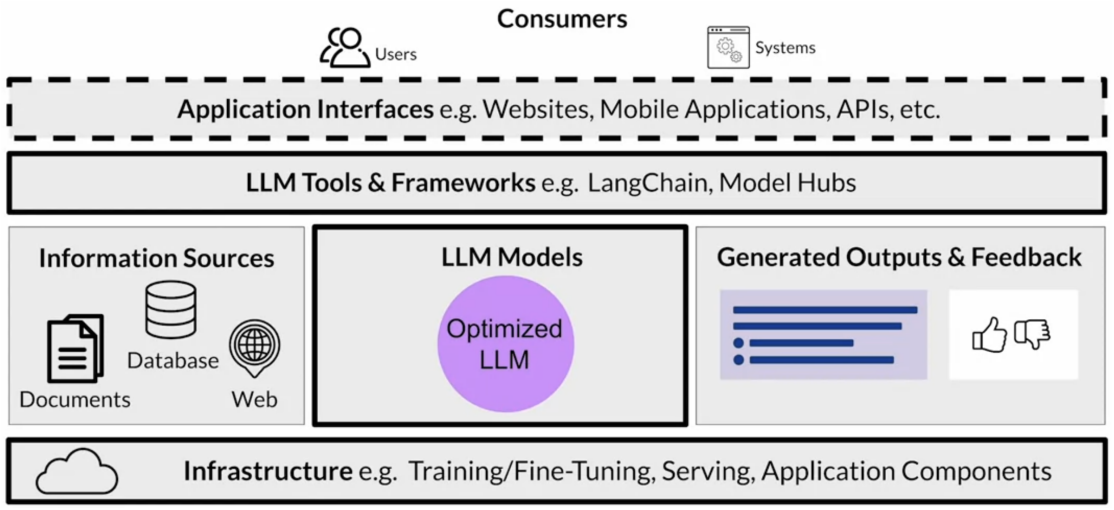
▼ Optional: AWS Sagemaker JumpStart

▼ Building generative applications



Building an LLM-powered application requires multiple components in the application stack. Sagemaker JumpStart is a model hub that helps quickly deploy foundation models available within the service and integrate them into the applications. The JumpStart service also provides an easy way to fine-tune and deploy models.

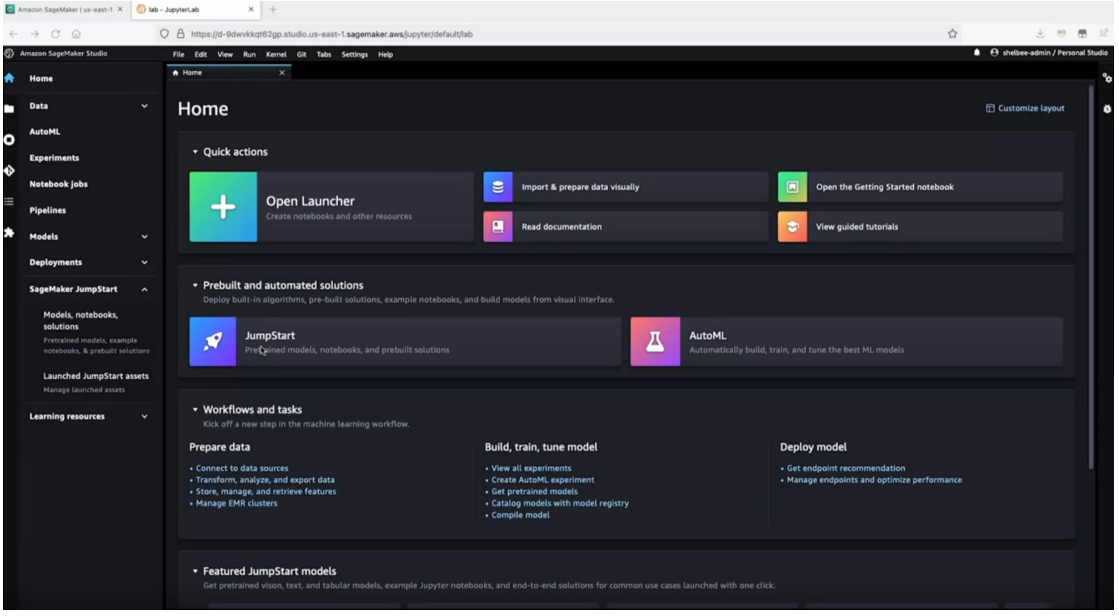
JumpStart covers many parts of this diagram, including the infrastructure, the LLM itself, the tools and frameworks, and even an API to invoke the model.



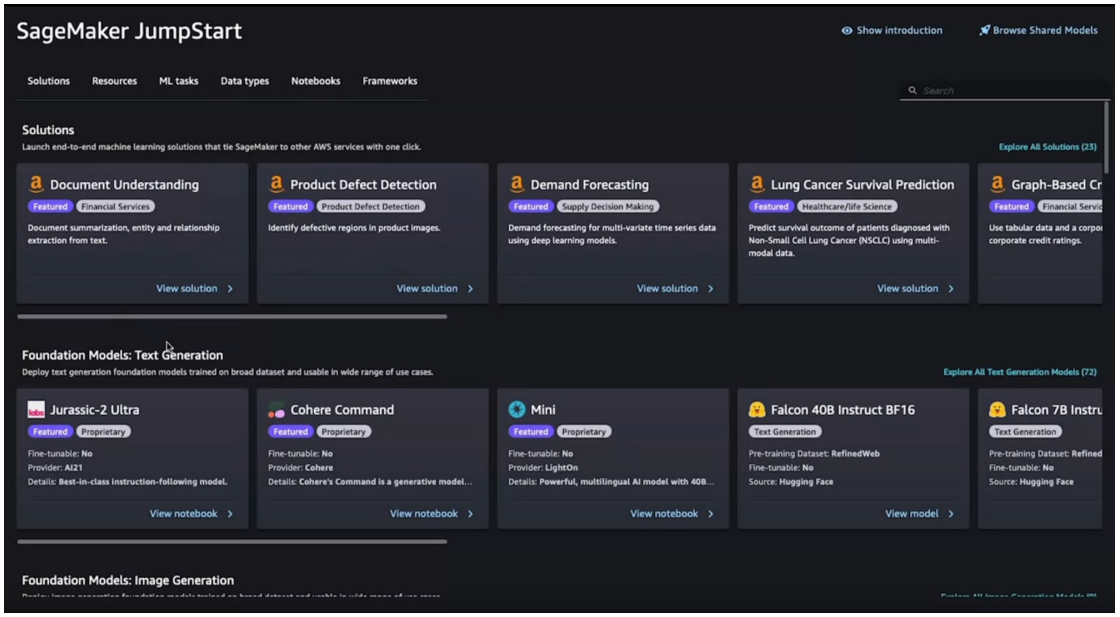
In contrast to the lab models, JumpStart models require GPUs to be fine-tuned and deployed. Remember that these GPUs are subject to on-demand pricing, refer to the Sagemaker pricing page before selecting the needed computer to use. Also, please delete the Sagemaker model endpoints when not in use and follow cost monitoring best practices to optimize cost.

▼ Sagemaker JumpStart

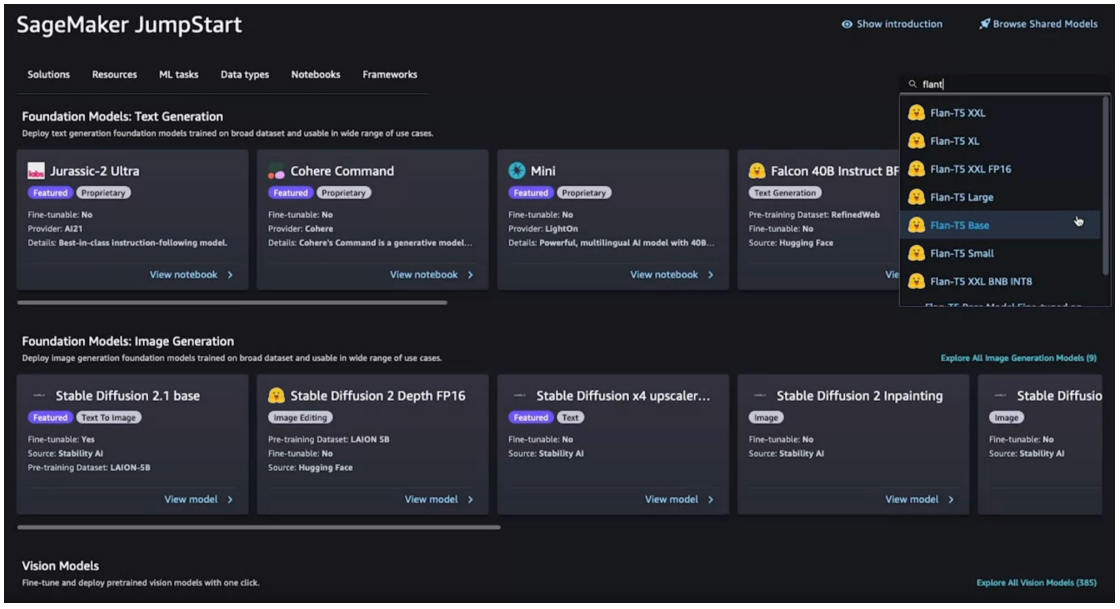
Sagemaker JumpStart is accessible from the AWS console or through Sagemaker studio. In the studio, choose JumpStart from the main screen. Optionally, choose JumpStart from the left-hand menu and select models, notebooks, and solutions as well.



After clicking on "JumpStart," the different categories include end-to-end solutions across different use cases, as well as a number of foundation models for different modalities that can be easily deployed and fine-tuned, where yes is indicated under the fine-tuning option.

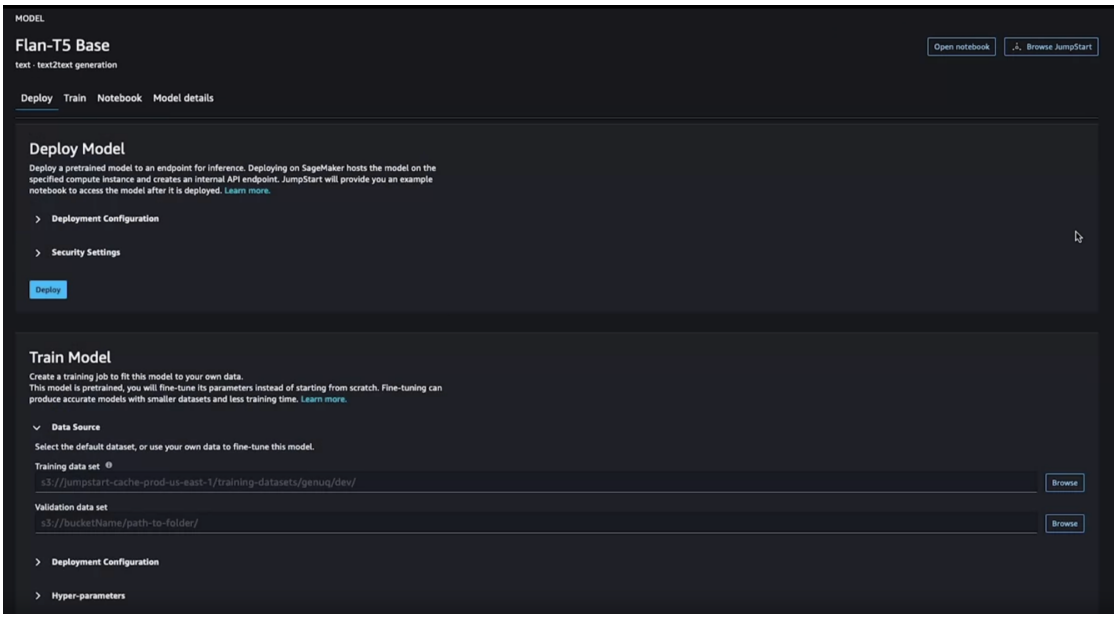


Let's look at an example you're all familiar with after working through the course: the Flan-T5 model.

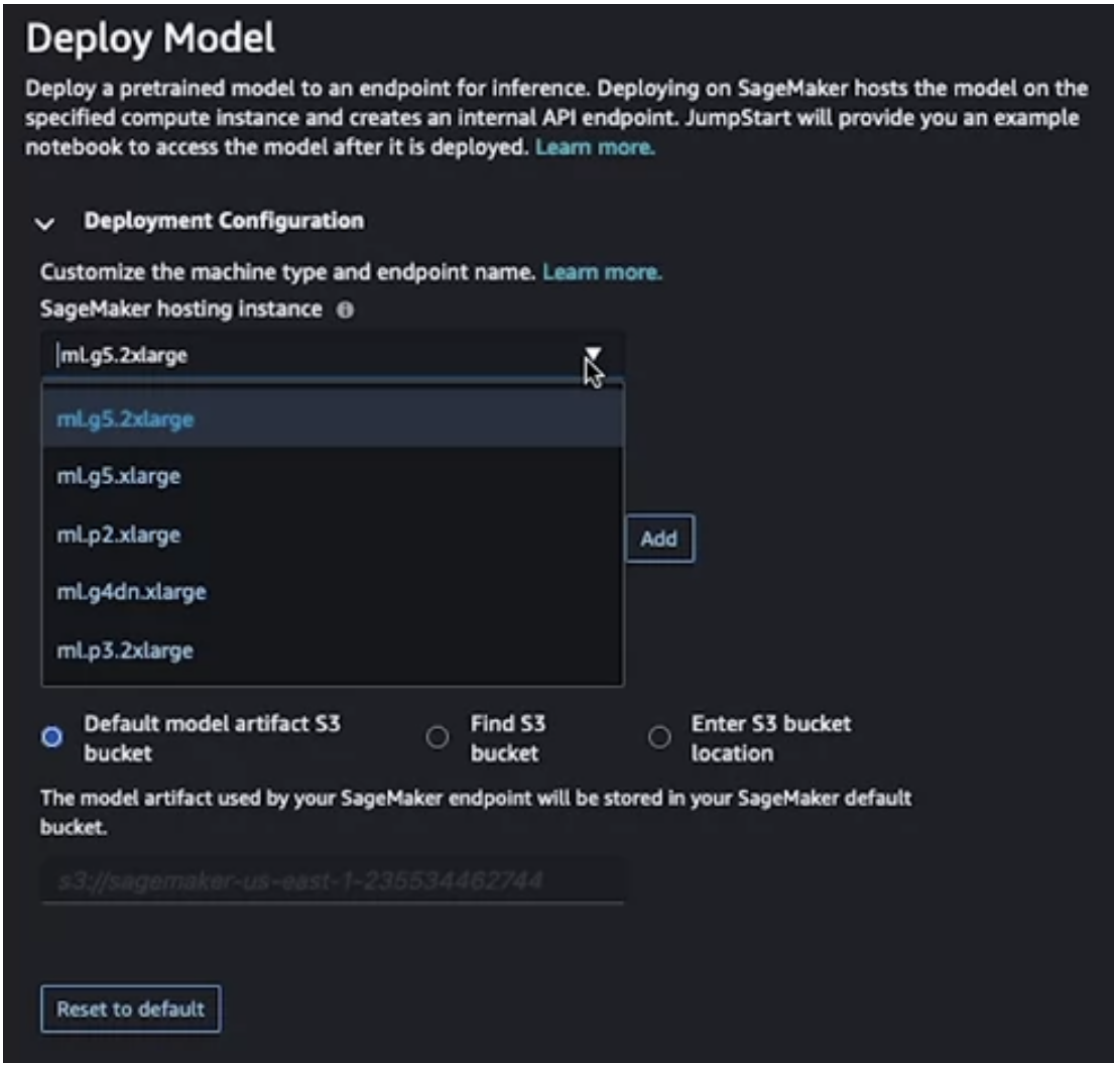


The base variant in the course can specifically minimize the resources needed by the lab environments. However, other variants of Flan-T5 through JumpStart can also be utilized depending on the needs. The

Hugging Face logo means they're actually coming directly from Hugging Face. AWS has worked with Hugging Face to the point where the model can easily be deployed or fine-tuned with just a few clicks.

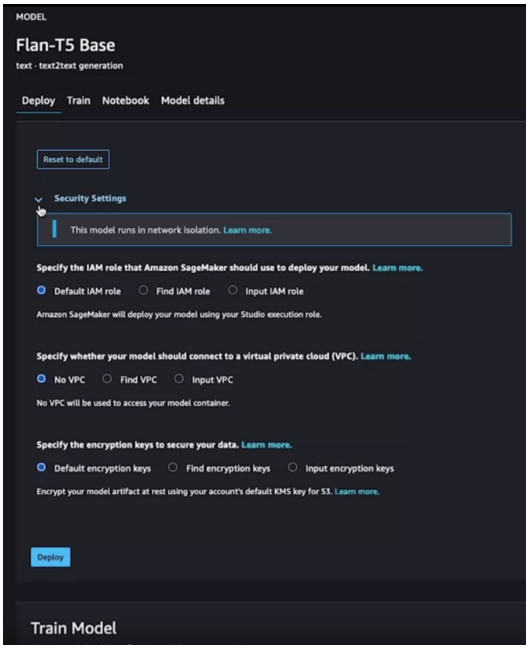


Selecting Flan-T5 Base gives a few options. First, to deploy the model, identify key parameters like the instance type and size. And this is the instance type and size that should be used to host the model.



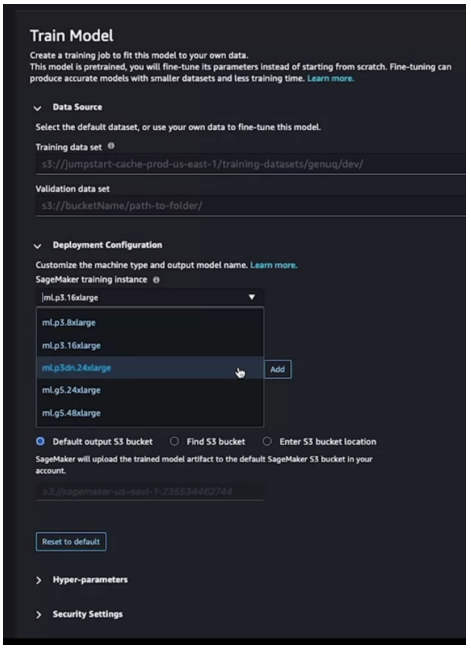
As a reminder, this deploys to a real-time persistent endpoint, and the price depends on the selected hosting instance. Some of these can be quite large, so always remember to delete any endpoints that are not in use to avoid incurring any unnecessary cost.

You'll also notice you can specify a number of security settings, allowing you to implement the controls acquired for your security requirements.



Selecting "Deploy" will automatically deploy that Flan-T5 Base model to the endpoint using the specified infrastructure.

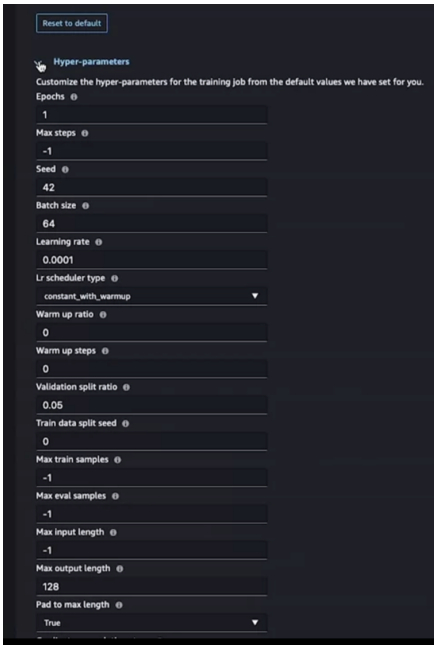
In the second tab, the option to train. Because this model supports fine-tuning, the fine-tuning jobs can also be set up by specifying the location of the training and validation data sets and then selecting the size of the compute used for training.



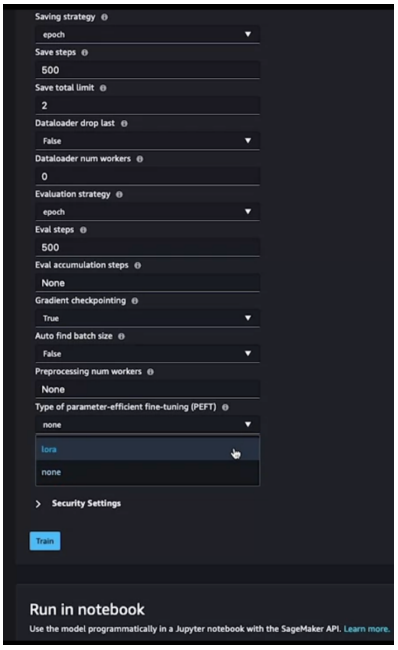
And it's just an easy adjustment to the size that computes through this drop-down; choose what type of computer to use for the training

job. And keep in mind again the underlying computer will charge for the time it takes to train the model. So, choosing the smallest instance required for a specific task is recommended.

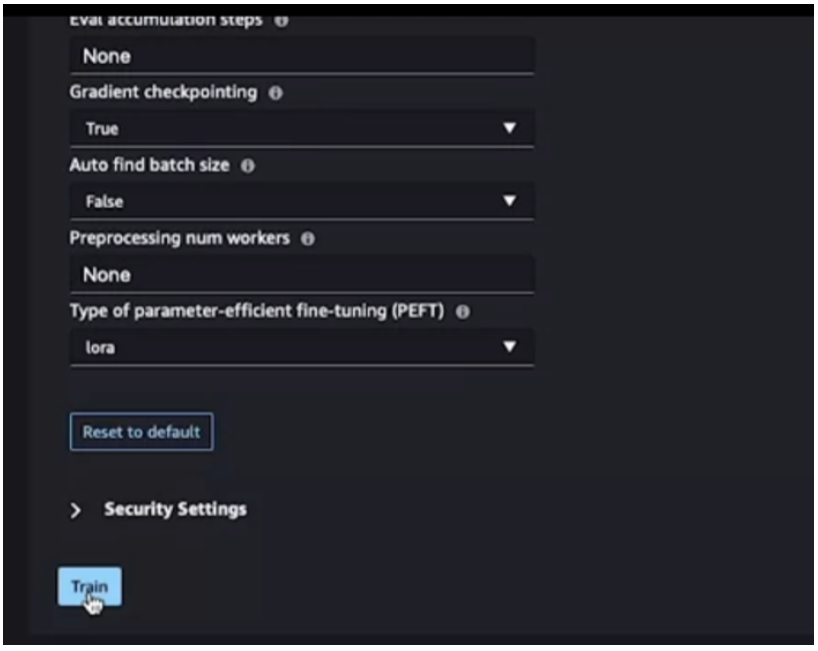
Another feature is the ability to quickly identify and modify the tunable hyperparameters for this specific model through these drop-downs.



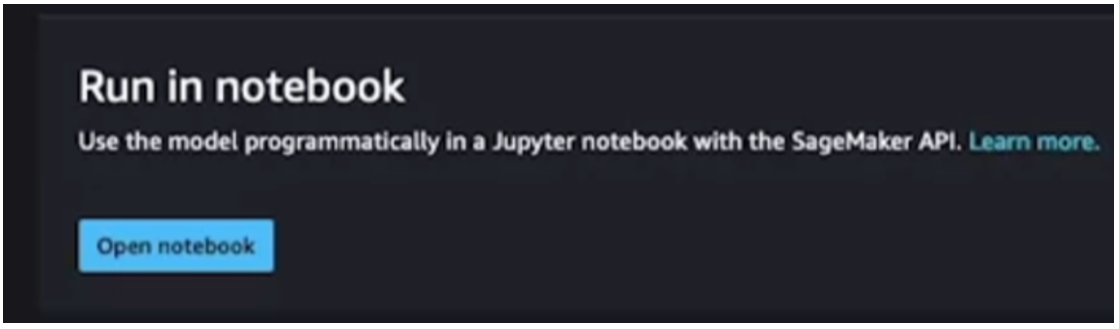
If we scroll down to the bottom, we will see a parameter type called PEFT, parameter-efficient fine-tuning.



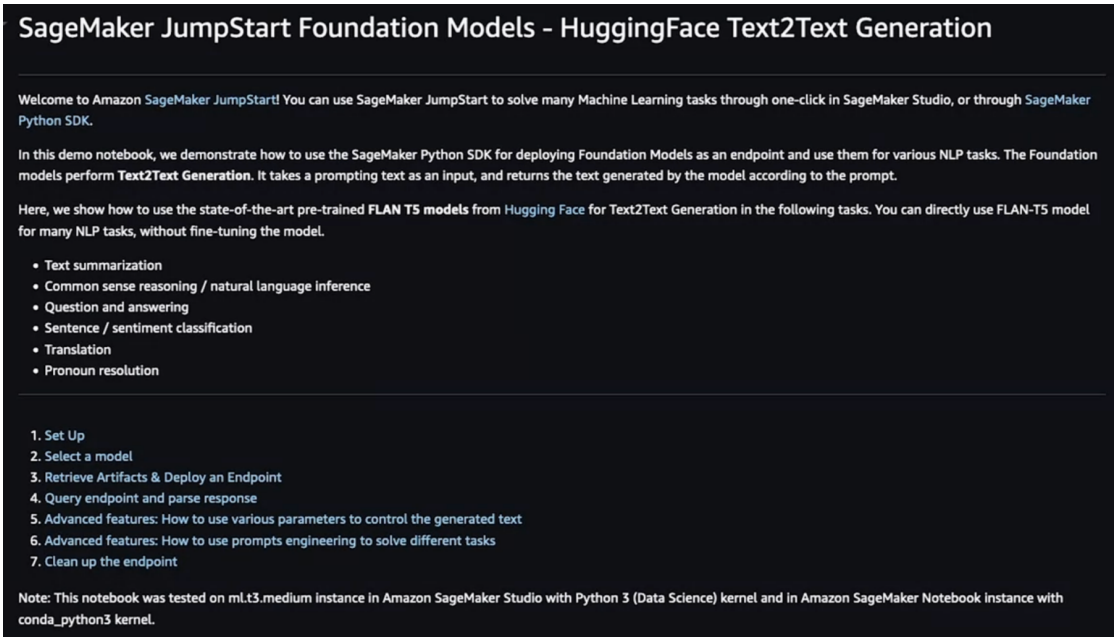
Selecting Lora through a simple dropdown makes implementing these various learned techniques easier. Then we can go ahead and hit "Train". And that'll kick off a training job to fine-tune this pre-trained Flan-T5 model using the input provided for a specific task.



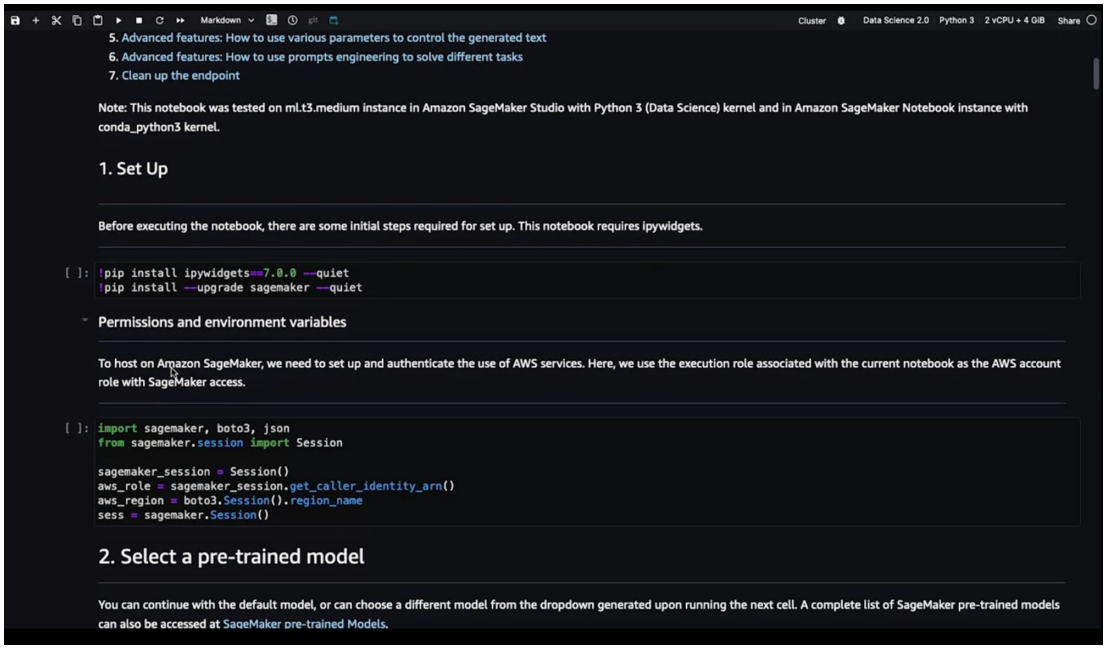
Finally, here is another option, which is to have JumpStart automatically generate a notebook. If the drop-down is not interesting, programmatically working with these models is preferred. This notebook provides all the code behind what is happening in the previously covered options.



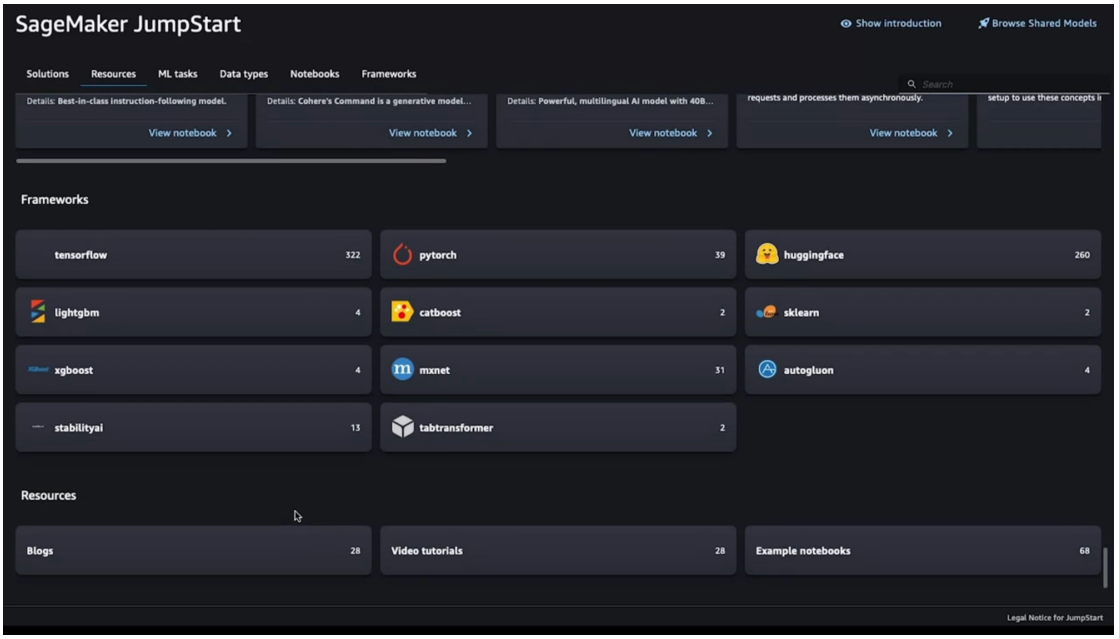
This is an option to work with JumpStart at the lowest level programmatically.



That was just a quick tour of JumpStart to illustrate the implementation of a model hub.



In addition to acting as a model hub that includes foundation models, JumpStart provides many resources in the form of blogs, videos, and example notebooks.



Exploring the different foundation models and their available variants is encouraged.

▼ Week 3 Quiz

1. Which of the following are true in regards to Constitutional AI? Select all that apply.

   ☑ **To obtain revised answers for possible harmful prompts, we must undergo a Critique and Revision process.**
   ☑ **Red Teaming elicits undesirable responses by interacting with a model.**
   ☐ **For constitutional AI, it is necessary to provide human feedback to guide the revisions.**
   ☑ **In Constitutional AI, we train a model to choose between different responses.**

2. What does the "Proximal" in Proximal Policy Optimization refer to?
   ○ **The algorithm's ability to handle proximal policies.**
   ○ **The use of a proximal gradient descent algorithm**
   ◉ **The constraint that limits the distance between the new and old policy**
   ○ **The algorithm's proximity to the optimal policy**

3. "You can use an algorithm other than Proximal Policy Optimization to update the model weights during RLHF."

Is this true or false?

◉ **True**
○ **False**

4. In reinforcement learning, particularly with the Proximal Policy Optimization (PPO) algorithm, what is the role of KL-Divergence? Select all that apply.

   ☐ **KL divergence encourages large updates to the LLM weights to increase differences from the original model.**
   ☐ **KL divergence is used to train the reward model by scoring the difference of the new completions from the original human-labeled ones.**
   ☑ **KL divergence measures the difference between two probability distributions.**
   ☑ **KL divergence is used to enforce a constraint that limits the extent of LLM weight updates.**

5. Fill in the blanks: When fine-tuning a large language model with human feedback, the action that the agent (in this case, the LLM) carries out is _____, and the action space is the _____.
   ○ **Calculating the probability distribution and the LLM model weights.**
   ◉ **Generating the next token and vocabulary of all tokens.**
   ○ **Generating the next token, the context window**
   ○ **Processing the prompt and context window.**

6. How does Retrieval Augmented Generation (RAG) enhance generation-based models?
   ○ **By optimizing model architecture to generate factual completions.**
   ○ **By applying reinforcement learning techniques to augment completions.**
   ○ **By increasing the training data size.**
   ◉ **By making external knowledge available to the model**

7. How can incorporating information retrieval techniques improve your LLM application? Select all that apply.

   ☐ **Faster training speed when compared to traditional models**
   ☑ **Improve relevance and accuracy of responses**

☐ **Reduced memory footprint for the model**
☑ **Overcome Knowledge Cut-offs**

8. What are the correct definitions of Program-aided Language (PAL) models? Select all that apply.

   ☑ **Models that offload computational tasks to other programs.**
   ☑ **Models that assist programmers in writing code through natural language interfaces.**
   ☐ **Models that integrate language translation and coding functionalities.**
   ☑ **Models that enable automatic translation of programming languages to human languages.**

9. Which of the following best describes the primary focus of ReAct?
   ○ **Investigating reasoning abilities in LLMs through chain-of-thought prompting.**
   ○ **Exploring action plan generation in LLMs.**
   ○ **Studying the separate topics of reasoning and acting in LLMs.**
   ◉ **Enhancing language understanding and decision-making in LLMs.**

10. What is the main purpose of the LangChain framework?
    ○ **To evaluate the LLM's completions and provide fast prototyping and deployment capabilities.**
    ◉ **To provide prompt templates, agents, and memory components for working with LLMs.**
    ○ **To connect with external APIs and datasets and offload computational tasks.**
    ○ **To chain together different components and create advanced use cases around LLMs, such as chatbots, Generative Question-Answering (GQA), and summarization.**

▼ Reading

Generative AI Lifecycle

- **Generative AI on AWS: Building Context-Aware, Multimodal Reasoning Applications**—This O'Reilly book explores all phases of the generative AI lifecycle, including model selection, fine-tuning, adapting, evaluation, deployment, and runtime optimizations.

Reinforcement Learning from Human-Feedback (RLHF)

- **Training language models to follow instructions with human feedback —A** Paper by OpenAI introduces a human-in-the-loop process to create a better model for following instructions (InstructGPT).
- **Learning to summarize from human feedback** - This paper presents a method for improving language model-generated summaries using a reward-based approach, surpassing human reference summaries.

Proximal Policy Optimization (PPO)

- **Proximal Policy Optimization Algorithms—**This is a paper from researchers at OpenAI who first proposed the PPO algorithm. The paper discusses the algorithm's performance on several benchmark tasks, including robotic locomotion and gameplay.
- **Direct Preference Optimization: Your Language Model is Secretly a Reward Model.** This paper presents a simpler and more effective method for precisely controlling large-scale unsupervised language models by aligning them with human preferences.

Scaling human feedback

- **Constitutional AI: Harmlessness from AI Feedback** This paper introduces a method for training a harmless AI assistant without human labels, allowing better control of AI behaviour with minimal human input.

Advanced Prompting Techniques

- **Chain-of-thought Prompting Elicits Reasoning in Large Language Models** - Paper by researchers at Google exploring how chain-of-thought prompting improves the ability of LLMs to perform complex reasoning.
- **PAL: Program-aided Language Models** - This paper proposes an approach that uses the LLM to read natural language problems and generate programs as the intermediate reasoning steps.
- **ReAct: Synergizing Reasoning and Acting in Language Models** This paper presents an advanced prompting technique that allows an LLM to

make decisions about interacting with external applications.

LLM-powered application architectures

- **LangChain Library (GitHub):** This library assists in the development of applications such as Question Answering, Chatbots, and Agents. The documentation is available here.
- **Who Owns the Generative AI Platform?** The article examines the market dynamics and business models of generative AI.