BRIGADE

Bastes-Roig Insights into Generative AI Development for the Enterprise

VOLUME 1

Get started on your generative AI journey and harness its power for your org - from a gentle intro to prompt engineering, all the way to foundation models and fine-tuning, with a focus on enterprise use cases and cost control.

Debbie Bastes & JV Roig

Nov 2023

Jared Reimer
Founder & CTO
Cascadeo

Foreword

Generative AI is the most important invention of the last 100 years - full stop. That is saying a lot and yet is not an exaggeration. Its existence, capabilities and immediate societal impact are far beyond anything most of us had dreamed possible. It is as if a massive undersea earthquake has occurred, a tsunami has formed and is racing towards the shoreline, and things as we know them - in the tech space and beyond - will never be the same again.

Marc Andreessen said it most succinctly: **AI Will Save The World**. Andreessen shows the path to a vastly improved world. Like all major milestones in technology before it, life is transformed in ways that lift people out of poverty, give them access to opportunity, reduce toil and enable further innovations.

Our older children will remember two worlds: *life before large language models and life after them*. Younger children will not know a world where you cannot have a conversation with AI – in fact, they will have those conversations all day, every day, as they go throughout life and education. The effect will be even more profound than "*before vs. after the Internet*" and even "*before vs. after the iPhone.*"

Generative AI tools are particularly important because they accomplish things that I, as a computer scientist and engineer, never actually thought possible: they are capable of synthesizing net-new information and insights, literature -- even images, songs and videos. It is as if we have unlocked something that *feels like it should be impossible* but yet somehow is **very, very real**.

A year ago, I was invited to co-present with AWS at **re:Invent 2022**. One of the engineers presenting with me did a live demonstration of **CodeWhisperer**. He built, launched, and used an infinitely-scalable serverless application capable of finding and describing dogs in a collection of photographs – *and did all of this in about five minutes*, with AI producing most of the code based on his stated intent. There were audible gasps in the room. It was a very tough act to follow, as you can imagine, as nearly no one had seen generative AI at this point.

Fast forward just a single year and the world looks completely different. At every trade show and conference, generative AI is the only thing on the agenda. At the bar, at parties, at family get-togethers, and certainly in the business media, there is one topic that has been top-of-mind from the moment OpenAI shocked the entire world by releasing ChatGPT.

It is also worth noting that this is the first time in history that a major, disruptive innovation has been instantly accessible to the entire connected population of the planet – billions of people – rather than subject to "trickle-down" economics. Anyone and

Jared Reimer
Founder & CTO
Cascadeo

Foreword

everyone can use these tools, even with the simplest of smartphones or tablets. This results in a massively-parallel wave of innovation, a warp-speed cycle of Darwinian evolution, and possibly a flywheel that will only continue to accelerate.

In short, this represents the opportunity of a lifetime. It is as important as the invention of electricity and the first electric light bulb. Understanding and being able to apply this technology will be a mandatory job skill for professional-class jobs in a matter of years.

Those who embrace it early will ride the giant tsunami wave.

Those who fight or ignore it will almost certainly be flattened by it.

The tsunami is racing towards shore at hundreds of miles per hour. There is no way to put the toothpaste back in the tube or uninvent the atomic bomb. The technology is already in the wild, open-source in some cases, and can be leveraged on even a modern laptop computer or smartphone.

Each of us has a decision to make. Do we ride the wave? Do we turn our backs to it and hope it isn't real? Do we run for the hills, despite being clearly far too late to reach them?

As I see it, there is only one reasonable option, and that is to learn as much as you can, as quickly as you can, about generative AI and its implications.

Most importantly, you need to work with these tools day-to-day, finding reasons to incorporate them into your workflows and companies. There is no substitute for direct experience.

The BRIGADE book is an excellent resource to get you started on such a journey, no matter your technical level. The authors - *JV and Debbie* - are two of the nicest and best people I've had the pleasure of working with. I am not surprised at all that they delved into this emerging new technology and created this fantastic resource, and then released it to the public.

If you are unsure how to get started, or want a guided tour into the various nooks and crannies of generative AI for yourself or your entire company, the BRIGADE book will help you learn to ride this incoming giant tsunami wave.

Enjoy the surfing, and buckle up – this is going to be a very bumpy, but incredibly exciting time in human history. The world will literally never be the same, and what we have seen thus far is only the beginning.

00 CONTENTS

CHAPTER 1: INTRODUCTION

Generative AI is one of the most important and impactful technologies in recent memory.

Large language models (LLMs) in particular have unlocked incredible capabilities that at first may seem like they previously only existed in science fiction - from being able to understand slang, analyze complex problems, follow tasks, show creativity, and just generally being “human-like” in responses.

In the enterprise, generative AI offers a lot of potential benefits:

- ➔ Operations, products and services infused with generative AI can lead to incredible efficiencies and new capabilities.
- ➔ More secure and reliable software and infrastructure through generative AI assistance across the DevOps pipeline.
- ➔ Boosting human capital by making employees more productive through generative AI enablement, as well as enabling internal support departments like HR to better analyze and understand employee issues.

Not everything is rosy, of course, as it always is with new technology. LLMs are resource hogs, demanding huge amounts of GPU memory (*and thus, many expensive GPUs in a cluster*) in order to inference at scale.

LLMs also aren’t infallible, and can be “confidently wrong” or **hallucinate**, presenting wrong information as correct and in an authoritative manner.

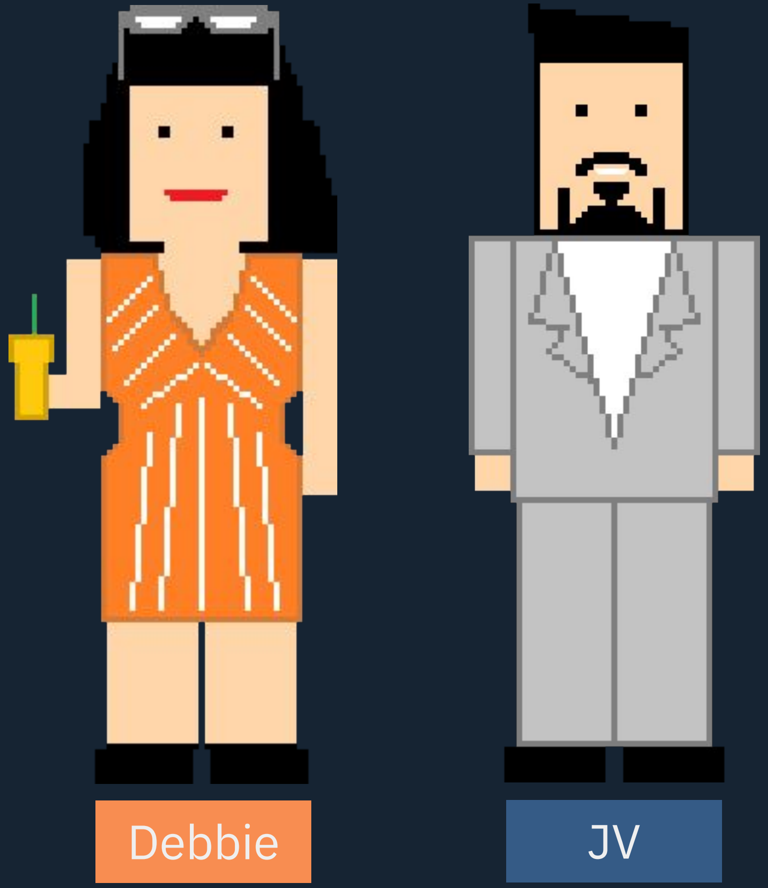
But the potential benefits are far too valuable, and these downsides can be managed and mitigated.

Generative AI is something that the enterprise cannot - and must not - ignore.

Generative AI is a required journey for the enterprise

No matter where you are, what your company offers or what market you compete in, generative AI presents an almost “do or die” choice. If you don’t go on this journey, well, your competitors will, and you’ll soon find yourself in the unenviable position of having to fight with competitors whose operations, products, services and human capital are all boosted with generative AI, making them more productive, reducing time to market, and perhaps even cutting key costs through superior decision-making enabled by generative AI analytics. The gap in technology will eventually be so large that you wouldn’t be in a fair fight anymore. It’ll almost be like an 18th century army - swords, horses and muskets - going to battle against modern mechanized infantry, with modern guns and armored vehicles. You don’t want to be the one stuck holding the muskets.

In this book, we - **JV Roig** and **Debbie Bastes**, a pair of IT & Cloud professionals who dove into generative AI - present a roadmap for successful enterprise adoption of generative AI to unlock those aforementioned benefits. We also share some experiments we’ve done and insights derived, relevant to the needs of enterprises in this generative AI journey.



The Generative AI roadmap

Going through the generative AI journey is a requirement, but how do you actually start?

No one becomes an expert overnight, so don’t worry. What matters is you start, have a plan to sustain the effort, and a reasonable, sensible plan.

The generative AI roadmap presented here is one such sensible plan. It shows a journey from unsophisticated public model use, all the way to customized private models from fine-tuned foundation models. There’s also the goal of a DevOps pipeline for engineering discipline and operational excellence. The culmination: a **Center of Excellence for Generative AI**, for an org-wide, cross-disciplinary approach to reaping the benefits of generative AI.

Kickstarting your generative AI journey

Alright, with the roadmap in hand, how do you start?

I know, there's a veritable firehose of information about generative AI across the internet, and even more daily updates. It's daunting even just figuring out how to start, or how to filter what's useful or not in an enterprise setting.

The rest of this book covers Levels 0 - 5 of the generative AI roadmap, to help get your various engineering / software development teams into the generative AI journey and producing value:

Chapter 2 offers an introduction to prompt engineering - why it is trickier than expected, how to engineer prompts to make them more reliable, and resources you can use to practice and develop this skill and go even deeper.

Chapter 3 tackles integrating generative AI capabilities into your own applications using public models through hosted services, such as OpenAI's GPT 3.5 and GPT 4.

Chapter 4 discusses private models - why use private instead of public models, and popular hosting services you can use for your own private models, such as Hugging Face and Sagemaker.

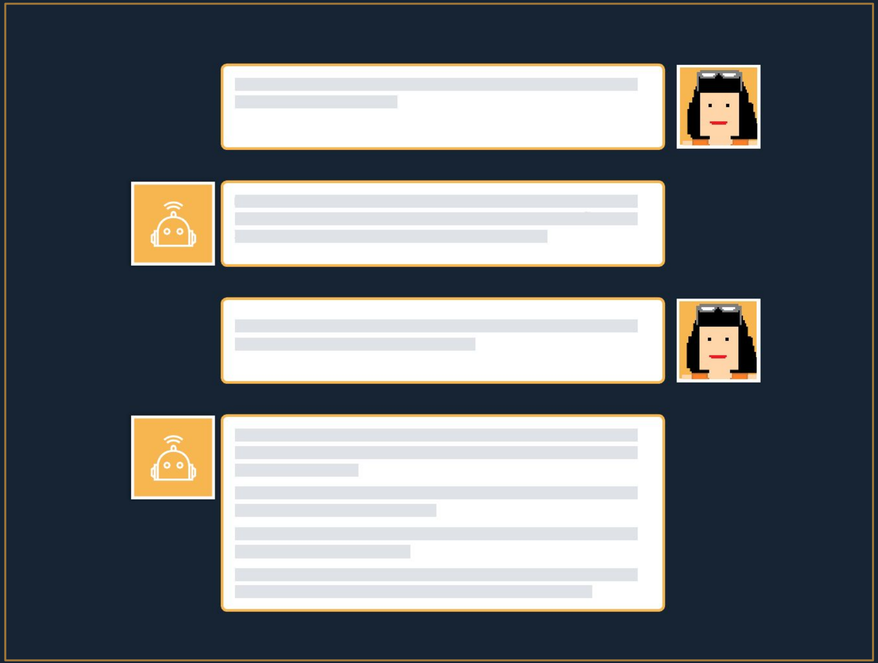
Chapter 5 goes a little more in-depth on foundation models, a concept that Chapter 4 deals with only in passing. Some popular foundation models are discussed here, along with running them locally. Some interesting experiments for different enterprise use cases are presented here, along with some very important cost analysis.

Chapter 6 deals with Retrieval Augmented Generation (RAG), a way to quickly enable your chosen LLM (be it a public or private model) to work with proprietary data that may be necessary for its intended function (e.g., your own internal company policies so that it can serve as a chatbot for employee concerns)

Chapter 7 ends this current journey with specialized smaller private models in order to reduce costs and increase performance for LLM deployments. A more in-depth enterprise cost analysis is presented here that improves upon the cost analysis in Chapter 5 using new concepts discussed here.

CHAPTER 2: PROMPT ENGINEERING

ChatGPT, with its chatbot-like features makes it easier for everyone to interact with a Large Language Model. A prompt is the initial message or input that we provide to LLMs like ChatGPT, which in turn generates an output. The input can be a question, an instruction, a complex problem, or any piece of text. A good prompt is clear, concise, and provides enough context for the model to generate a meaningful response.



The power of prompt engineering

Prompt engineering is the practice of crafting, refining, or designing prompts to elicit specific responses or behaviors from language models. Since the way we frame a question or instruction can greatly influence the quality of the model's response, prompt engineering has become an essential skill when working with models like GPT.

Effective prompt engineering allows us to leverage large language models by formulating detailed inputs that guide the model's responses to align with our specific use cases.

We might not always receive the desired response on our first attempt, but it is crucial to develop a systematic process for adjusting and iterating on our prompts, learning from each interaction with the model.

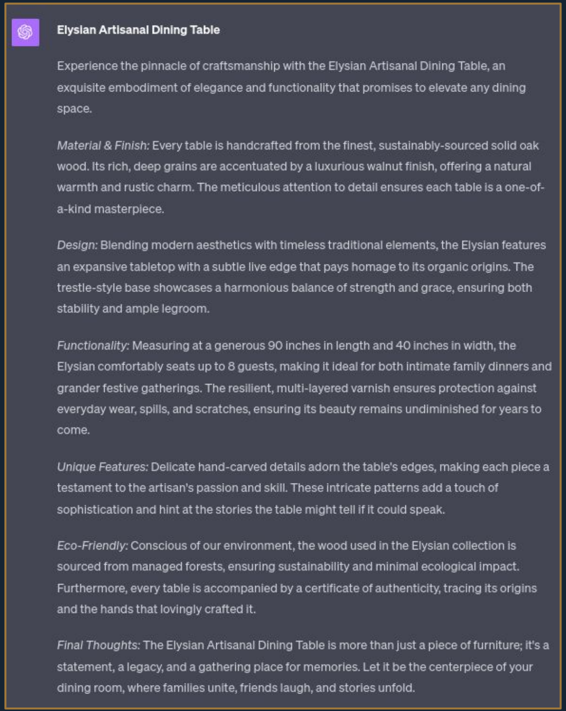
In this chapter, we'll use ChatGPT to practice prompt engineering. If you haven't already, just head on to https://chat.openai.com/ to create a free account and follow along our prompt engineering journey.

Use case 1: Creating marketing descriptions

Let's explore a few examples as we try to use prompt engineering to achieve our desired results. Our first mission? Instruct ChatGPT to create compelling marketing descriptions for furniture products in an online store backend, as seen below.

Let's begin by crafting a basic prompt to request for a marketing description, say for a dining table:

Prompt v1: *"Generate a compelling and detailed description for a dining table."*



Elysian Artisanal Dining Table

Experience the pinnacle of craftsmanship with the Elysian Artisanal Dining Table, an exquisite embodiment of elegance and functionality that promises to elevate any dining space.

Material & Finish: Every table is handcrafted from the finest, sustainably-sourced solid oak wood. Its rich, deep grains are accentuated by a luxurious walnut finish, offering a natural warmth and rustic charm. The meticulous attention to detail ensures each table is a one-of-a-kind masterpiece.

Design: Blending modern aesthetics with timeless traditional elements, the Elysian features an expansive tabletop with a subtle live edge that pays homage to its organic origins. The trestle-style base showcases a harmonious balance of strength and grace, ensuring both stability and ample legroom.

Functionality: Measuring at a generous 90 inches in length and 40 inches in width, the Elysian comfortably seats up to 8 guests, making it ideal for both intimate family dinners and grander festive gatherings. The resilient, multi-layered varnish ensures protection against everyday wear, spills, and scratches, ensuring its beauty remains undiminished for years to come.

Unique Features: Delicate hand-carved details adorn the table's edges, making each piece a testament to the artisan's passion and skill. These intricate patterns add a touch of sophistication and hint at the stories the table might tell if it could speak.

Eco-Friendly: Conscious of our environment, the wood used in the Elysian collection is sourced from managed forests, ensuring sustainability and minimal ecological impact. Furthermore, every table is accompanied by a certificate of authenticity, tracing its origins and the hands that lovingly crafted it.

Final Thoughts: The Elysian Artisanal Dining Table is more than just a piece of furniture; it's a statement, a legacy, and a gathering place for memories. Let it be the centerpiece of your dining room, where families unite, friends laugh, and stories unfold.

Result: *As seen on the left.*

From a single sentence as input, ChatGPT generated a detailed write-up, including a product name, its features, materials and design descriptions.

While we got a creative response, it is too verbose for our use case because we only need a short description for our product database.

Let’s try to improve our prompt: if we need short product blurbs instead of a one-pager, a way to do that is to set a word or sentence limit. Let’s take it further by specifying which features of the product we want to focus on.

Prompt v2: *“Generate a compelling and detailed description for a dining table in under 100 words, emphasizing its durability and design.”*

Result:

> **Athena Modern Dining Table**
>
> Discover the Athena, a blend of contemporary design and robust durability. Handcrafted from premier solid oak, its rich grains sing tales of nature's beauty. With a sleek silhouette and a deep chestnut finish, this table radiates modern elegance. Its sturdy trestle base ensures stability, while the expertly applied protective varnish resists daily wear and spills. Perfectly seating eight, the Athena isn't just a table—it's a resilient canvas for memories, designed to withstand time and trends.

That’s more like it! With just a few tweaks, ChatGPT produced a description we can use for our system.

The length of the generated output can also have *inference time* and *cost implications* when we are already managing our own GenAI-powered applications. We'll delve deeper into this in subsequent chapters.

Back to our example – we already have a list of products with their respective features and specifications, so let’s choose one to include in our prompt. This way, the description will be tailored to our product, not teeming with random features that might mislead our customers.

As we add more elements to our prompt, we can use delimiters like **triple backticks** to separate distinct sections of our input. In other use cases, you might want to include sample marketing descriptions, detailed factsheets, or even marketing guidelines that can help shape the tone, style and format of the output.

So for v3 of our prompt, we’ll include an instruction section, and we’ll use delimiters to specify which section contains the details about the product. Let’s see how this affects the result.

Prompt v3:

Generate a compelling and detailed description of a product for a marketing website of a furniture store, in under 100 words. The product name and specifications are provided below which are delimited with triple backticks.

Product:
` ` `
Product Name: Aspen Dining Table
Style: Rustic
Color Variations: Oak, Maple
Material: Solid Wood
Furniture type: 4-seater Dining Table
Product Category: Dining Room Furniture - Aspen Series
Weight in kilograms: 12
Length in meters: 2
Width in meters: 0.76
` ` `

Result:

Aspen Dining Table: Where Rustic Meets Refinement

Elevate your dining space with the Aspen Dining Table—a harmony of rustic charm and modern design. Skillfully crafted from premium solid wood, its robust nature is showcased in rich Oak and Maple finishes. This 4-seater beauty, part of the coveted Aspen Series, stretches to a generous 2m in length while maintaining a weight of just 12 kg. With its slim 0.76m width, it's a perfect fit for intimate gatherings. Dine in style and let every meal become a memory, only with Aspen.

Now, the description maintains its creativity without being overly verbose, and it incorporates the product specifications we provided.

We're achieving better results, but we shouldn't stop here. We should test the prompt multiple times to ensure consistency and try to generate descriptions for other products.

Creating prompt templates like this comes in handy when handling various inputs for the same task. We can test the same instructions, swap the product and check for consistent results. Alternatively, we can refine the instruction section and compare the generated descriptions for the same product. Yes, our journey doesn't end here; we're just getting started!

And don't worry; when managing hundreds of products and testing numerous iterations, there's no need to paste our prompts into a ChatGPT window individually. We'll learn a more efficient method in Chapter 3, so stay tuned! But before that, let's move onto our next use case.

Use case 2: Sentiment analysis

Now that we have a better understanding of prompt engineering, let's try another enterprise use case. This time, we'll use ChatGPT to determine the sentiment of customer reviews for our products.

Drawing from the techniques we already learned, we'll use delimiters to specify the customer review text which we want to be analyzed. For added context, we will also include the name of the product being reviewed.

Prompt v1-A:
What is the sentiment of the following product review, which is delimited with triple backticks?

Product Name: Bliss Rocking Chair
Review text:
` ` `
I am absolutely in love with it! The chair is not only beautiful, but also incredibly comfortable. The rocking motion is smooth and soothing, making it perfect for relaxing after a long day. The craftsmanship is top-notch, with sturdy construction and high-quality materials. I also appreciate the attention to detail in the design, such as the curved armrests and the ergonomic backrest. Overall, I highly recommend it to anyone looking for a stylish and comfortable addition to their home. It truly brings joy and relaxation to my life!
` ` `

Result:

> The sentiment of the provided product review is overwhelmingly positive. The reviewer expresses their love for the Bliss Rocking Chair, highlighting its beauty, comfort, smooth rocking motion, craftsmanship, and attention to detail. They also highly recommend it and mention that it brings joy and relaxation to their life.

Not bad for our first attempt! The model not only identified the correct sentiment but also summarized the review and highlighted what the customer liked about our product.

Be wary though - while GPT-3.5 and GPT-4 excel at simple sentiment analysis tasks, certain real-life customer reviews can be tricky to analyze, especially if they don't express strong emotions, contain sarcasm or use informal language.

Let’s take a look at another example using the same prompt template, but for a different product review.

Prompt v1-B*:*
What is the sentiment of the following product review, which is delimited with triple backticks?

Product Name: Kyushu Calm Lounge Sofa
Review text:
` ` `
The quality of the fabric on this couch is okay, but it's not the most comfortable seating I've experienced. It looks nice in my living room, though.
` ` `

Result:

> The sentiment of the provided product review is somewhat mixed but leans more towards the neutral or slightly negative side.
>
> The reviewer mentions that the quality of the fabric on the Kyushu Calm Lounge Sofa is "okay," which suggests that they are not overly impressed with it. Additionally, they state that it's "not the most comfortable seating" they've experienced, indicating some dissatisfaction with the comfort level.
>
> However, they do mention a positive aspect by saying that it "looks nice" in their living room, which implies some level of aesthetic appreciation.
>
> Overall, the sentiment in this review is not as positive as the previous one and is more on the lukewarm or slightly negative side.

That doesn’t seem like a definitive answer, because the review doesn’t convey strong emotions unlike our first example.

While the analysis provided seems sound, it is even longer than the review itself. In situations like this, we can guide the LLM further by specifying the output we want. In version 2 below, we’ll provide 3 options to ChatGPT: Positive, Negative or Neutral.

Prompt v2:
Here is a product review from a customer, which is delimited with triple backticks.

Product Name: Kyushu Calm Lounge Sofa
Review text:
` ` `
The quality of the fabric on this couch is okay, but it's not the most comfortable seating I've experienced. It looks nice in my living room, though.
` ` `
Overall sentiment must be one of the following options:
A) Positive
B) Negative
C) Neutral

What is the overall sentiment of that product review?

Result:

> The overall sentiment of the product review you provided is C) Neutral.

This time we got a straightforward answer based on the choices we provided. However, we lost some details in the original review - it's useful to know what our customers like or hate about our product. ChatGPT can extract the relevant information so we don’t have to go through hundreds or thousands of reviews ourselves. Let's do that by asking for a breakdown of the positive and negative comments for our product.

Prompt v3:
Here is a product review from a customer, which is delimited with triple backticks.

Product Name: Kyushu Calm Lounge Sofa
Review text:
` ` `
The quality of the fabric on this couch is okay, but it's not the most comfortable seating I've experienced. It looks nice in my living room, though.
` ` `

What is the sentiment of that product review?
Identify the product being reviewed.
Enumerate the positive and negative aspects of the product review.
The response should have the following elements:
- *Product name*
- *Review Sentiment (Positive/Negative/Neutral)*
- *Positive comments about the product (Enumerate)*
- *Negative comments about the product (Enumerate)*

Result:

Okay, so now the model properly labeled the sentiment and extracted pertinent information from the review. This will be easier to process when we’re consolidating results and creating reports.

Another way to improve results is to provide good examples of actual reviews it might encounter in production, paired with the expected outputs. This technique is known as **one-shot** or **few-shot prompting**. While it might be unnecessary for more powerful models like GPT-4, in Chapter 5 you’ll see how it can drastically improve performance in smaller models. There are also other valuable insights from experiments on model performance in that chapter, so don’t miss it!

Use case 3: Generating sample customer feedback

For our final use case, let's generate sample customer reviews to put our freshly engineered sentiment analysis prompts to the test. This is yet another area where LLMs excel — they are masters at producing text that is indistinguishable from human writing, creating synthetic data suitable for a variety of applications.

Let's create a prompt template with the *Product Name* and *Star Rating* as input so we can vary the samples. We want a variety of sample customer feedback texts that exhibit different sentiments (e.g., positive, negative, neutral) so we can thoroughly test our sentiment analysis prompt.

Here are two examples using the same prompt template and product but different star rating.

Prompt v1-A:

Create a realistic customer review for the product below. It can be positive, negative or neutral sentiment given the provided Star Rating. Limit your response to 150 words.

Product:
```
Aspen Dining Table
```
Star Rating:
```
3
```

I recently purchased the Aspen Dining Table with a rating of 3 stars. Overall, I have mixed feelings about this product.

On the positive side, the design of the table is quite appealing, and it fits well with my dining room decor. The wood finish is also of decent quality, giving it a rustic charm. Assembly was relatively straightforward, and it seems sturdy enough to withstand regular use.

However, there are some downsides to consider. The table arrived with a few minor scratches, which was disappointing. Additionally, I noticed that the table wobbles slightly on uneven flooring, and the legs don't feel as stable as I had hoped for.

In conclusion, the Aspen Dining Table is an attractive piece of furniture with some drawbacks. It's decent for the price, but the quality and stability could be improved.

Prompt v1-B:

Create a realistic customer review for the product below. It can be positive, negative or neutral sentiment given the provided Star Rating. Limit your response to 150 words.

Product:
```
Aspen Dining Table
```
Star Rating:
```
4
```

I recently purchased the Aspen Dining Table and I must say I'm quite pleased with it. The sleek and modern design of the table fits perfectly in my dining room, giving it an elegant touch. The quality of the materials used is evident, and the table feels sturdy and well-made.

Assembly was relatively straightforward, and I appreciated that all the necessary tools and instructions were included. The table's size is just right for my family, and it comfortably seats six people, making it ideal for gatherings and dinner parties.

The only reason I didn't give it a perfect 5-star rating is that I noticed a small scratch on the tabletop upon unpacking it. It's not very noticeable, but it's a minor imperfection that slightly disappointed me.

Overall, I'm happy with my purchase, and the Aspen Dining Table has added a touch of sophistication to my home.

In v2, we'll specify the output format to be a Python dictionary containing the product name and review text. Depending on your application, you can choose to specify the output format as something you can process easily.

Prompt v2:

Create a realistic customer review for the product below. It can be positive, negative or neutral sentiment given the provided Star Rating. Limit your response to 150 words.

The response should be in a python dictionary format with the following elements:
- Product Name
- Product Review

Product:
```
Aspen Dining Table
```
Star Rating:
```
3
```

```
{
"Product Name": "Aspen Dining Table",
"Product Review": "I recently purchased the Aspen Dining Table and have mixed feelings about it. On the positive side, it's a sturdy and well-built piece of furniture. The design is elegant, and it looks great in my dining room. However, I do have a few reservations. The assembly process was a bit challenging, and some of the screws provided were of lower quality. Also, there was a small scratch on the tabletop when it arrived, which was disappointing. Overall, it's a decent dining table with a fair price point, but the minor issues with assembly and the scratch prevent me from giving it a higher rating. I would recommend it with some caution, but make sure to inspect it thoroughly upon delivery."
}
```

If you're up for a challenge, consider refining the prompts further to enhance the model's ability to generate more nuanced outputs. Keep in mind that real-world feedback can often be more subtle and complex than the straightforward examples we've encountered so far. Therefore, it's crucial that the system can effectively handle various nuances, including sarcasm, indirect language, and mixed emotions.

Consider our third example where we added instructions and asked for comments with idiomatic expressions and varying levels of subtlety. Notice how the style of the reviews changed, but GPT-4 might have gone a little overboard with the idioms. Now try to improve on that and see what kind of reviews you'll get. Afterwards, use these sample reviews to test the sentiment analysis prompt in **use case 2**.

Prompt:

Create a realistic customer review for the product below. It can be positive, negative or neutral sentiment given the provided Star Rating. Limit your response to 150 words.

Include comments with idiomatic expressions, varying levels of subtlety, and different ways people may express sentiments to ensure robustness in my sentiment analysis model.

Product:
```
Aspen Dining Table
```
Star Rating:
```
2,3,4
```

Result:

2-Star Review:
Title: "Not Exactly a Feast for the Eyes"

I was stoked to get the Aspen Dining Table, but my enthusiasm was short-lived. It arrived with a chip off the old block – literally. The tabletop had a noticeable nick on the edge, and one of the legs was wobblier than a jelly on a roller coaster. Assembly was a beast, too; the instructions were as clear as mud. It's not all bad – the size is perfect for my nook, and when it's all dressed up, it looks halfway decent. But honestly, I expected more bang for my buck.

3-Star Review:
Title: "Middle of the Road Mealtime Companion"

The Aspen Dining Table is a mixed bag. On the plus side, it's got a charm that could warm the cockles of your heart with its rustic vibe. On the downside, it's as stable as the weather in spring – one of the legs just can't seem to stay put. Putting it together was no picnic, but once it's up, it does its job. It's the epitome of 'you get what you pay for' – not the cream of the crop, but not the bottom of the barrel either.

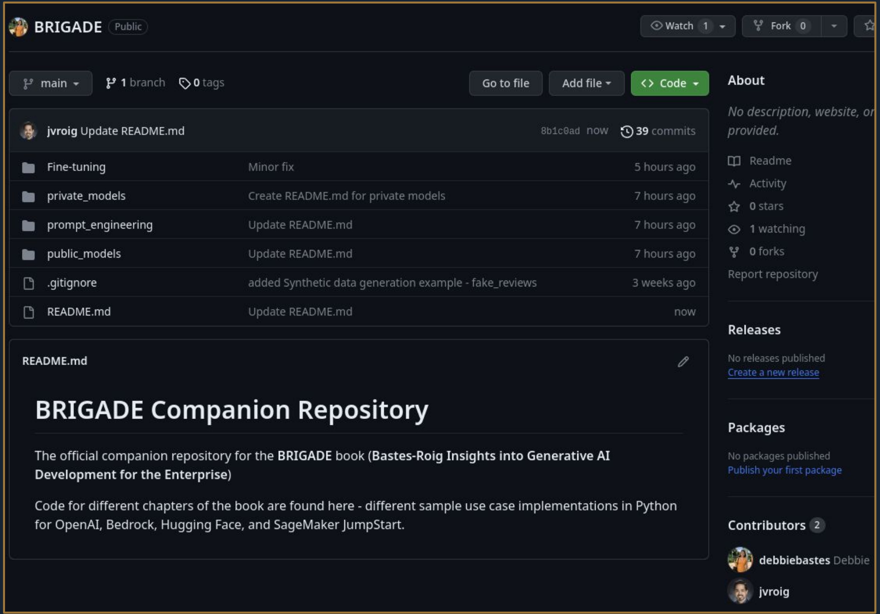
4-Star Review:
Title: "Solid Table with Minor Quirks"

I must say, the Aspen Dining Table mostly hits the nail on the head. It's got a sturdy, homey feel that's like a breath of fresh air in my dining area. The wood has a rich grain that's as inviting as a warm hug. Assembly wasn't exactly a walk in the park, and it did have a quirk or two; a leg needed a bit of coaxing to align properly. But once set up, it's been standing strong. It's not quite the belle of the ball, but it's definitely not a shrinking violet either – a solid four stars in my book!

Wrap up

Equipping ourselves with prompt engineering skills enables us to harness the power of GenAI models. As we saw in our examples, by continuously refining our prompts, we can align an LLM’s capabilities with our specific business objectives.

Try experimenting on your own! The prompts we utilized in this chapter can be accessed [here in the BRIGADE GitHub companion repo](#).



Below, we've compiled a list of the prompt engineering techniques we employed in our examples:

➔ **Provide explicit instructions:** *"Give a detailed description of this dining table, emphasizing its durability and design."*

➔ **Limit response length:** *"Provide a marketing description of this dining table in under 40 words."*

➔ **Cite examples:** This is also called one-shot or few-shot prompting. We can set the style, tone and format of the desired output through our examples.

➔ **Use delimiters to indicate parts of the prompt:** You can use delimiters like *<begin> <end>*, ` ` ` ,””” to specify distinct sections of the prompt. This is essential in the case of more complex instructions or multi-line requests.

➔ **Ask for output in a specified format.**

➔ **Create prompt templates.**

INTERESTING READS AND RESOURCES

Prompt Engineering: How to Talk to the AIs
This is a course by **Xavier Amatriain, VP of Engineering, AI Product Strategy at LinkedIn**.
Great insights into prompt crafting for image generation and large language models!

Mitigating LLM Hallucinations: A Multifaceted Approach
Dealing with hallucinations is one of the most difficult challenges in almost every generative AI solution.
This blog post is a must-read for prompt engineers.

What's the Future of Generative AI? An Early View in 15 Charts
A visual explainer from **McKinsey** about the future of generative AI.
Tables and graphs instead of walls of text.

Prompt Engineering a Prompt Engineer
Catchy title. One of the more recent papers I saw about developments in the field of prompt engineering. Worth a look if you're interested in the the rapid innovations in prompt engineering.

CHAPTER 3:
PUBLIC MODELS INTEGRATION

So far we've practiced prompt engineering using the ChatGPT browser interface. Now that we've prompt engineered capabilities we like, how do we integrate these capabilities into our own systems and applications?

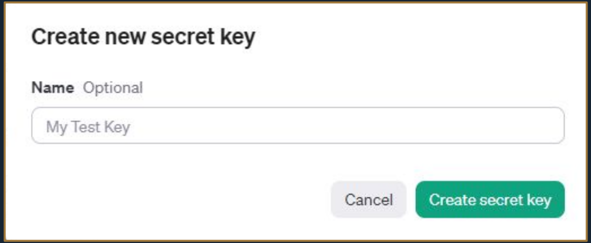
OpenAI provides paid API access to the models that power ChatGPT: GPT-3.5 and GPT-4. If we want to integrate the generative AI capabilities we prompt-engineered in the last chapter, this is the way to go about it.

Getting started with OpenAI API integration

To begin, we must first gain access to OpenAI's API by following these steps:

1. **Sign up:** Head over to the OpenAI API portal, sign-up for an account and complete the verification steps.

2. **Create an API key:** Go to your account's API keys page and create a new secret key. These keys are your credentials for accessing the OpenAI API and should be kept confidential.



3. Download the SDK: OpenAI provides a convenient Software Development Kit (SDK) that simplifies interaction with the API. You can install this SDK in your preferred programming environment. If you are using Python, for example, you can install the OpenAI package using pip:

```
$ pip install openai
```

4. API key configuration: Set up your environment variables to store your API keys. It's important not to hard code your keys directly into your application for security reasons. *Remember that this key is a sensitive piece of information*—treat it as you would a password.

   For our examples, let's create a local .env file, and store our API key there. Copy the text below and replace the value with your newly created secret key from Step 2.

```
# Once you add your API key below, make sure to not
share it with anyone! The API key should remain
private.
OPENAI_API_KEY='Add your OpenAI API Key here'
```

Note: In a production environment, use secrets management services to handle sensitive data securely. We will revisit secure API key storage later when discussing AWS cloud infrastructure.

3. **Test your setup:** Once everything is installed and configured, let's run this snippet of code to verify that the setup is successful. This simple Python example sends a prompt to the API and prints the response:

```
#import modules needed for Chat GPT comms
import os
import openai

# read local .env file
from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv())

# Your API key from OpenAI
openai.api_key = os.getenv('OPENAI_API_KEY')

completion = openai.ChatCompletion.create(
  model="gpt-3.5-turbo",
  messages=[
    {"role": "system", "content": "You are a helpful
assistant."},
   {"role": "user", "content": "What were the main former
capitals of Japan?"}
  ]
)
print(completion.choices[0].message)
```

Run that script, and you should see a response with the information you requested, verifying that your integration is successful. Assuming you saved that in a file called **openai_test.py**, then you'd trigger it like so:

```
$ python3 openai_test.py
```

Creating marketing descriptions through the API

Now that we've successfully set up the OpenAI SDK, we can start implementing our previous prompt-engineered use cases.

Let's go through the components of our Python script. First, we'll need to import the necessary modules, including the OpenAI package we just installed.

```
# Import modules needed for OpenAI API communication
import os
import openai
```

Using the python-dotenv package, let's read the local .env file to retrieve our API key. This API key is necessary to authenticate requests to the OpenAI API.

```
# Read local .env file
from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv())

# Set the API key from the environment variable
openai.api_key  = os.getenv('OPENAI_API_KEY')
```

Next, let's define a function named **call_ai** whose task is to send a prompt to the chat completions API and receive a response. It takes in two parameters: prompt and model (with a default value of "*gpt-3.5-turbo*").

```
# Call the Generative AI Service like OpenAI
def call_ai(prompt, model="gpt-3.5-turbo"):
    messages = [{"role": "user", "content": prompt}]
    response = openai.ChatCompletion.create(
        model=model,
        messages=messages,
        temperature=1,# this is the degree of
                      # randomness of the model's output
    )
    return response.choices[0].message["content"]
```

It's time to add our prompt—we'll use the marketing description example from **Chapter 2** and store it in a multi-line string variable named **prompt**.

(All sample codes in this chapter can be found in our GitHub repo.)

```
# Create a prompt to send to the Generative AI Service
prompt = f"""Generate a compelling and detailed
description of a product for a marketing website of a
furniture store, in under 100 words. The product name and
specifications are provided below which are delimited
with triple backticks.

Product:
```
Product Name: Aspen Dining Table
Style: Rustic
Color Variations: Oak, Maple
Material: Solid Wood
Furniture type: 4-seater Dining Table
Product Category: Dining Room Furniture - Aspen Series
Weight in kilograms: 12
Length in meters: 2
Width in meters: 0.76
```

"""
```

Let's print the prompt to the console so we can see what is being sent to the API.

```
print('Prompt: %s' %prompt)
```

What's left is to pass our prompt to the function, retrieve the model-generated text, and store it in a variable named **response**. Let's also print it to the console to see the output.

```
response = call_ai(prompt)
print('Response:\n%s ' %response)
```

Finally, let's run our script:

````
$ python3 marketing_description.py
Prompt: Generate a compelling and detailed description of a product for a marketing website of a furniture store, in
 under 100 words. The product name and specifications are provided below which are delimited with triple backticks.


Product:
```
Product Name: Aspen Dining Table
Style: Rustic
Color Variations: Oak, Maple
Material: Solid Wood
Furniture type: 4-seater Dining Table
Product Category: Dining Room Furniture - Aspen Series
Weight in kilograms: 12
Length in meters: 2
Width in meters: 0.76
```


Response:
Introducing the Aspen Dining Table, the epitome of rustic charm and elegance. Crafted from solid wood, this 4-seater
 dining table showcases stunning craftsmanship and durability. Available in two gorgeous color variations, Oak and M
aple, you can easily match it with your existing décor. The Aspen Dining Table is a perfect addition to your dining
room, bringing warmth and style to every meal. With its dimensions of 2 meters in length and 0.76 meters in width, i
t comfortably accommodates your family and friends. Experience the allure of timeless design and exceptional quality
 with the Aspen Dining Table from our exclusive Aspen Series.
````

Sentiment analysis through the API

Alright, our marketing description generator is done. That wasn't so hard at all! Let's move on to sentiment analysis. This is extremely important to integrate into a real program, instead of just being done through the ChatGPT web interface.

In an enterprise setting, the volume of feedback that you will need to analyze is simply too much for a person (or even multiple persons) to just copy-and-paste individually into a ChatGPT window, and then copy-paste results into a document afterwards.

That's too tedious, expensive, and impossible to scale. What if you have 1,000 comments, feedback or complaints to analyze daily?

Let's see how we can implement our prompt-engineered sentiment analysis capability using the API.

```
# Import modules needed for OpenAI API communication
import os
import openai

# Read local .env file
from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv()) # read local .env file
```

```
# Set the API key from the environment variable
openai.api_key  = os.getenv('OPENAI_API_KEY')

# Call the Generative AI Service like OpenAI
def call_ai(prompt, model="gpt-3.5-turbo"):
    messages = [{"role": "user", "content": prompt}]
    temperature=0.5
    response = openai.ChatCompletion.create(
        model=model,
        messages=messages,
        temperature=temperature, # this is the degree of
randomness of the model's output
    )
    return {
        'body': response.choices[0].message["content"],
        'context': {
            'ChatGPT model': response.model,
            'temperature': temperature,
            'ChatGPT token usage': response.usage,
        }
    }
```

Most of that code will be familiar from our first use case - importing the necessary modules, retrieving the API key and sending a request to the OpenAI API. You'll notice we slightly modified our call_ai function to now return the 'context', along with the model generated text.

The context provides details on the AI model used, temperature setting, and the token usage. This data is useful for usage monitoring, prompt testing and model performance benchmarking and can be retrieved and processed separately.

The API call actually returns additional data, along with the model response. The model and token usage of our context comes from the **ChatCompletion** object returned by our API call. We won't cover all of them in our example, but you can refer to the official OpenAI documentation for more details.

As for our use case, we'll focus on the token usage as it is the basis for the usage cost of our application. This part of the object provides information about how the request counts against the API usage. It typically includes:

➔ **prompt_tokens**: The number of tokens used in the prompt [*input tokens*].

➔ **completion_tokens**: The number of tokens generated in the completion [*output tokens*].

➔ **total_tokens**: The sum of *prompt_tokens* and *completion_tokens*.

We'll be able to compute the usage cost of our sample use case later after we run our script, so let's continue!

Here we'll define two variables and set their values. First is the information about our product (**product_name**) and the accompanying review from our customer (**review_text**), which we want to be analyzed by the LLM. Right now, it is hardcoded in our script for simplicity but typically our data will come from an external data storage (like a customer review database).

```
product_name = "Kyushu Calm Lounge Sofa"
review_text="""
The quality of the fabric on this couch is okay, but
it's not the most comfortable seating I've experienced.
It looks nice in my living room, though.
"""
```

Now we're ready to assemble our prompt and print the response. We will pass our customer review data, along with the prompt when we execute an API call.

```
prompt = f"""Here is a product review from a customer, which is delimited with triple backticks.

Product Name: {product_name}
Review text:
```
{review_text}
```

What is the sentiment of that product review?
Identify the product being reviewed.
Enumerate the positive and negative aspects of the product review.
The response should have the following elements:
        - Product name
        - Review Sentiment (Positive/Negative/Neutral)
        - Positive comments about the product (Enumerate)
        - Negative comments about the product (Enumerate)
"""

# Do something with the AI response
print('Prompt: %s' %prompt)
response = call_ai(prompt)
print('Response:\n%s ' %response['body'])
print('\nContext:\n%s ' %response['context'])
```

Running our completed script outputs the following information to the console:

- ➔ The prompt sent to the AI model (gpt-3.5-turbo).
- ➔ The body of the response, which includes the analysis of the product review based on the prompt. We can see the model determined the review sentiment to be Neutral, and enumerated the positive and negative comments from the customer.
- ➔ The context, providing details on the AI model used, temperature setting, and the token usage.

(sample output on next page)

Notice the **prompt_tokens** and **completion_tokens** in the **Context** output at the bottom.

Those refer to our input and output token usage, respectively. We'll use those figures for estimating our costs.

```
$ python3 sentiment_analysis.py
Prompt: Here is a product review from a customer, which is delimited with triple backticks.

Product Name: Kyushu Calm Lounge Sofa
Review text:
```

The quality of the fabric on this couch is okay, but it's not the most comfortable seating I've experienced. It looks nice
in my living room, though.

```


What is the sentiment of that product review?
Identify the product being reviewed.
Enumerate the positive and negative aspects of the product review.
The response should have the following elements:
        - Product name
        - Review Sentiment (Positive/Negative/Neutral)
        - Positive comments about the product (Enumerate)
        - Negative comments about the product (Enumerate)

Response:
Product Name: Kyushu Calm Lounge Sofa
Review Sentiment: Neutral

Positive comments about the product:
1. The fabric quality is okay.
2. It looks nice in the living room.

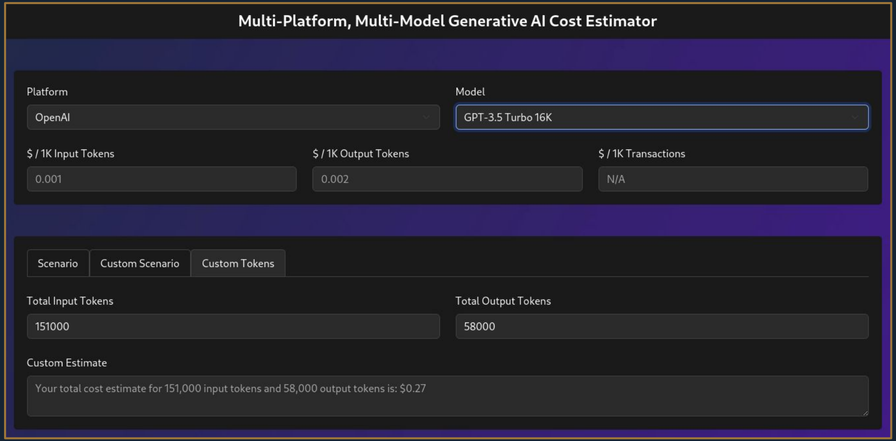
Negative comments about the product:
1. It's not the most comfortable seating experience.

Context:
{'ChatGPT model': 'gpt-3.5-turbo-0613', 'temperature': 0.5, 'ChatGPT token usage': <OpenAIObject at 0x1b47fbfa6f0> JSON: {
  "prompt_tokens": 151,
  "completion_tokens": 58,
  "total_tokens": 209
}}
```

API Usage Cost Estimate

As promised, let's try to compute our usage cost using the token usage data returned by our function. Using our handy dandy GenAI cost estimator, all we need to do is choose our platform (OpenAI) and model (gpt-3.5-turbo 16K), choose the **Custom Tokens** tab, and then type in our total input and output tokens.

The sample computation below is for 1,000 requests (*input tokens = 151 x 1,000, output tokens = 81 x 1,000; see stats from sample output in previous page*). Running our script to analyze ~1,000 customer reviews will cost us around $0.27. The actual usage cost will vary slightly, since the review text length and model response length will be different per customer review.



Generating human-like feedback through the API

Our final prompt engineering use case from Chapter 2 was generating human-like feedback as fake data.

Now, we can integrate that into a script, so we can actually use this capability to augment traditional faker libraries.

(Faker libraries are dev utilities commonly found in popular programming languages, designed to make it easy for developers to create fake-but-realistic-looking data to aid in testing, quality assurance, and product demos. While these are good with names, dates, places and numbers, these aren't really designed to create fake human-looking text, and mostly just end up using "lorem ipsum" random words. This is a good use case for generative AI augmentation)

Just like in the previous section where we implemented sentiment analysis, this means we'd be able to use this capability at scale, avoiding the tediousness of repeatedly using the ChatGPT window each time we need one piece of human-looking fake text. *(Would any dev want to do that 100 times when the team needs a product demo with at least 100 fake records that include customer feedback?)*

We'll be creating two components this time - a custom Python module named GenAI_faker and a Python script to read the product catalog and make use of the GenAI_faker module to generate sample reviews.

GenAI_faker library

We first need to create the GenAI_faker Python module that provides functionality to generate fake product reviews by using the OpenAI API. The module also uses the traditional faker library to create realistic-looking names and generate random star ratings for products.

Imports and initial setup:

➔ We'll import the following modules: **openai, os, time, json,** and **Faker** from the faker library.

➔ Then create an instance of **Faker** for generating fake data.

➔ We'll then define an empty string **api_key** to store the OpenAI API key. Our other Python script will retrieve our key and pass it along later.

```
# Import modules needed for OpenAI API communications
import openai
import os
import time
import json
from faker import Faker

fake = Faker()
api_key=''
```

Next, the **call_ai** function makes another appearance here to facilitate the API call. Everything is the same as our last example.

```
# Call the Generative AI Service like OpenAI
def call_ai(prompt, model="gpt-3.5-turbo"):
    messages = [{"role": "user", "content": prompt}]
    temperature=0.5
    response = openai.ChatCompletion.create(
        model=model,
        messages=messages,
        temperature=temperature, # this is the degree of
randomness of the model's output
    )
    return {
        'body': response.choices[0].message["content"],
        'context': {
            'ChatGPT model': response.model,
            'temperature': temperature,
            'ChatGPT token usage': response.usage,
        }
    }
```

Finally, let's add a new function called **product_review**. Here's a detailed breakdown of its functionality:

- ➔ It takes a payload dictionary expected to contain details about a product, particularly with keys for **Furniture_Name** and **Furniture_ID**.
- ➔ It then generates a basic **product_review** dictionary with keys for *"Product ID", "Product Name", "Name of Reviewer", and "Star Rating"*. The *"Product Name"* and *"Product ID"* values are pulled from the payload, and the *"Name of Reviewer"* and *"Star Rating"* are generated using the standard faker library.
- ➔ The prompt for the OpenAI API call is constructed using **Product Name** and **Star Rating**. This prompt asks the AI model to create a customer review in a Python dictionary format.
- ➔ It then calls the **call_ai** function with this prompt to get a review text from the OpenAI API.
- ➔ The response from the AI is expected to be JSON-formatted text that includes a product review. This is parsed from the JSON and added to the **product_review** dictionary.
- ➔ If an error occurs during the process, it is caught, printed to the console, and the function returns **None**. Otherwise, the function returns the completed **product_review** dictionary containing the generated review.

```
def product_review(payload):
    product =payload.get('Furniture_Name', 'Unknown Product')

    #create product review for each product in the catalog
    product_review = {
        "Product ID": payload.get('Furniture_ID', 'Unknown
Product ID'),
        "Product Name": payload.get('Furniture_Name', 'Unknown
Product'),
        "Name of Reviewer": fake.name(),
        "Star Rating": fake.pyint(min_value=1, max_value=5),
    }

    prompt = f"""
    Create a sample customer review for the ``{product}```.
    It can be positive, negative or neutral sentiment given the
provided Star Rating.
    ```{product_review['Star Rating']}```
    The response should be in a python dictionary format with
the following elements:
    - Product Name
    - Product Review
    """
```

(code continues on next page)
```

```
    try:
        GenAI_review_text = call_ai(prompt)
        product_review['Product Review'] =
json.loads(GenAI_review_text['body'])['Product Review']
    except Exception as e:
        print(f"An error occurred: {str(e)}")
        return None

    return product_review
```

Fake_reviews.py

And just like that, we already have our GenAI_faker Python module! Time to create the fake_reviews python script and bring it all together.

This script is designed to read a CSV file containing a product catalog and generate fake human-looking product reviews using the GenAI_faker module we just made. Here's what each part of the code does:

Import modules:

➔ We start by importing the **GenAI_faker** library we just made

➔ We import the **os** module to interact with the operating system and read the environment variables.

➔ Lastly, we'll use the csv module to read from and write to CSV files.

Environment variables:

We already know this section – this is where we load the environment variables from a **.env** file. Next we just set the api_key attribute of **GenAI_faker** to the value of the **OPENAI_API_KEY** we just retrieved.

```
# Read local .env file
from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv()) # read local .env file

# Set the API key attribute of the GenAI_faker library
GenAI_faker.api_key  = os.getenv('OPENAI_API_KEY')
```

Parsing a CSV file:

Furniture_ID	Furniture_Name	Style	Color_Variations	Material	Furniture_Type
F-000010	Serenity Bed	['Vintage', 'Ornate']	['Ivory', 'Gold', 'Antique Silver']	Mahogany Wood, Velvet	Bed
F-000011	Echo Coffee Table	['Industrial', 'Reclaimed']	['Black', 'Brown', 'Weathered Gray']	Metal, Reclaimed Wood	Coffee Table
F-000012	Cascade Bookshelf	['Scandinavian', 'Modern']	['White', 'Natural Wood', 'Gray']	Wood, Metal	Bookshelf
F-000013	Ember Ottoman	['Mid-Century', 'Elegant']	['Light Gray', 'Teal', 'Cream']	Velvet, Walnut Wood	Ottoman

Next, let's define a function that opens a CSV file named **product_catalog.csv**, which contains a list of available products, as shown in the example above. This function returns a list of dictionaries, each representing a product.

```
def parse_file_to_dictionary():
    with open('test_data/product_catalog.csv', 'r') as f:
        reader = csv.DictReader(f)
        data_dict = [row for row in reader]
        return data_dict
```

Let's build our **product_catalog** dictionary by calling our function above. It should now contain a list of product dictionaries from our CSV file. After that we just iterate on each product on the list and let the GenAI_faker library do its magic. Below we just use it as we would a traditional Faker library, except this time it's powered by GPT-3.5.

```
product_catalog = parse_file_to_dictionary()

#create product review for each product in the catalog
for product in product_catalog:
    print("Product: \n", product)
    product_review = GenAI_faker.product_review(product)
    print("Product Review: \n", product_review)
```

Finally when we run our script, we'll see the AI model generated reviews for each product entry on the catalog. Here is one example of the output showing a review for the product named Serenity Bed:

```
Product Review:
{
    'Product ID': 'F-000010',
    'Product Name': 'Serenity Bed',
    'Name of Reviewer': 'Jasmine Walton',
    'Star Rating': 4,
    'Product Review': "I recently purchased the Serenity Bed and I am extremely satisfied with my purchase. The bed is incredibly
comfortable and provides excellent support for a good night's sleep. The design is stylish and the quality of the materials used is
top-notch. The assembly was straightforward and the bed feels very sturdy. The only reason I am not giving it a full 5-star rating
is because the delivery took longer than expected. Overall, I highly recommend the Serenity Bed for its comfort and quality."
}
```

In just a few seconds, the model can generate several customer review samples for us which we can use to test our sentiment analysis prompt from the previous section.

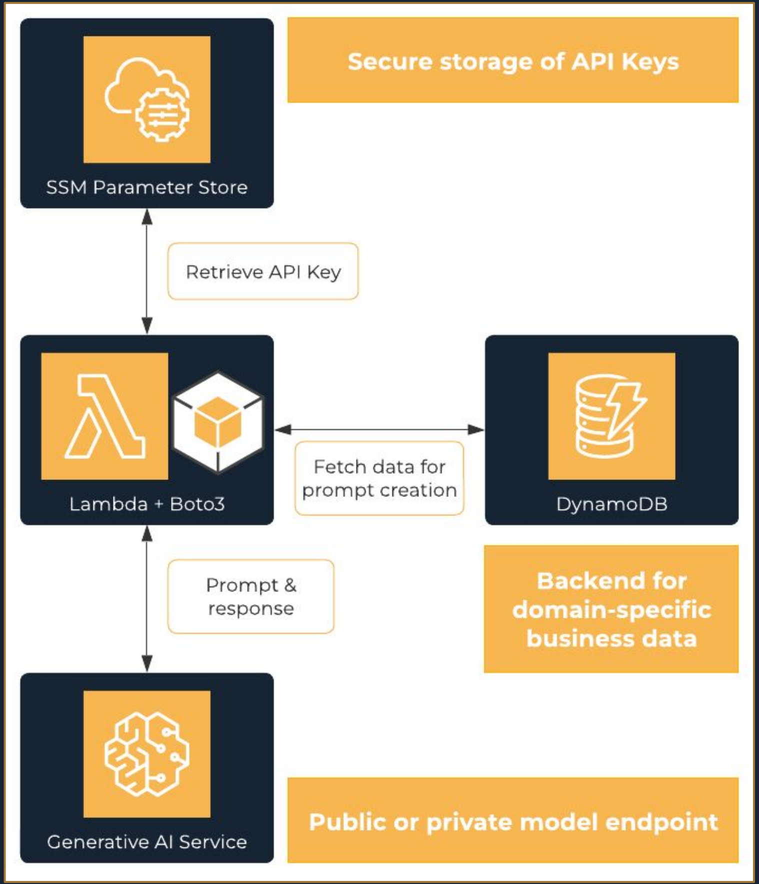
```
$ python3 fake_reviews.py
Product:
 {'Furniture_ID': 'F-000010', 'Furniture_Name': 'Serenity Bed', 'Style': "['Vintage', 'Ornate']", 'Color_Variations': "['Ivory', 'G
old', 'Antique Silver']", 'Material': 'Mahogany Wood, Velvet', 'Furniture_Type': 'Bed', 'Product_Category': 'Bedroom Furniture', 'D
escription': 'A vintage-style bed frame with an ornate headboard and luxurious velvet upholstery', 'Weight_in_kg': '15', 'Lenght_in
_meters': '2.2', 'Width_in_meters': '1.8', 'Height_in_meters': '1.5'}
Product Review:
 {'Product ID': 'F-000010', 'Product Name': 'Serenity Bed', 'Name of Reviewer': 'Jasmine Walton', 'Star Rating': 4, 'Product Review
': "I recently purchased the Serenity Bed and I am extremely satisfied with my purchase. The bed is incredibly comfortable and prov
ides excellent support for a good night's sleep. The design is stylish and the quality of the materials used is top-notch. The asse
mbly was straightforward and the bed feels very sturdy. The only reason I am not giving it a full 5-star rating is because the deli
very took longer than expected. Overall, I highly recommend the Serenity Bed for its comfort and quality."}
Product:
 {'Furniture_ID': 'F-000011', 'Furniture_Name': 'Echo Coffee Table', 'Style': "['Industrial', 'Reclaimed']", 'Color_Variations': "[
'Black', 'Brown', 'Weathered Gray']", 'Material': 'Metal, Reclaimed Wood', 'Furniture_Type': 'Coffee Table', 'Product_Category': 'L
iving Room Furniture', 'Description': 'An industrial-style coffee table with a sturdy metal frame and a reclaimed wood top', 'Weigh
t_in_kg': '12', 'Lenght_in_meters': '1.2', 'Width_in_meters': '0.8', 'Height_in_meters': '0.4'}
Product Review:
 {'Product ID': 'F-000011', 'Product Name': 'Echo Coffee Table', 'Name of Reviewer': 'Henry Gross', 'Star Rating': 1, 'Product Revi
ew': 'I am extremely disappointed with the Echo Coffee Table. The quality is very poor and it started falling apart within a few we
eks of use. The wood is flimsy and the table wobbles all the time. I regret purchasing this product and would not recommend it to a
nyone.'}
Product:
 {'Furniture_ID': 'F-000012', 'Furniture_Name': 'Cascade Bookshelf', 'Style': "['Scandinavian', 'Modern']", 'Color_Variations': "['
White', 'Natural Wood', 'Gray']", 'Material': 'Wood, Metal', 'Furniture_Type': 'Bookshelf', 'Product_Category': 'Home Office Furnit
ure', 'Description': 'A Scandinavian-style bookshelf with open shelving and a combination of wood and metal materials', 'Weight_in_
kg': '18', 'Lenght_in_meters': '1.5', 'Width_in_meters': '0.6', 'Height_in_meters': '2.2'}
Product Review:
 {'Product ID': 'F-000012', 'Product Name': 'Cascade Bookshelf', 'Name of Reviewer': 'Diana Hayes', 'Star Rating': 2, 'Product Revi
ew': 'The Cascade Bookshelf is not up to my expectations. The quality of the materials used is subpar and the construction feels fl
imsy. Additionally, the design is not very appealing. I am disappointed with my purchase and would not recommend this product.'}
Product:
 {'Furniture_ID': 'F-000013', 'Furniture_Name': 'Ember Ottoman', 'Style': "['Mid-Century', 'Elegant']", 'Color_Variations': "['Ligh
t Gray', 'Teal', 'Cream']", 'Material': 'Velvet, Walnut Wood', 'Furniture_Type': 'Ottoman', 'Product_Category': 'Living Room Furnit
ure', 'Description': 'A mid-century-style ottoman with elegant velvet upholstery and walnut wood legs', 'Weight_in_kg': '4', 'Lengh
t_in_meters': '0.6', 'Width_in_meters': '0.6', 'Height_in_meters': '0.4'}
```

Integrating OpenAI capabilities with your AWS infra

We've successfully implemented generative AI capabilities in our own code using the OpenAI SDK through their API. What would a real-world implementation in the Cloud look like?

The diagram on the right shows a generic architecture for generative AI integration in an application that runs on AWS:

- First, we securely store our API keys in a service like **SSM Parameter Store**.
- Our program lives in a Lambda function, and this function needs to be configured with both the **OpenAI SDK** and **boto3** (the AWS SDK for Python). If you aren't using Lambda, you could just imagine this as your favorite compute service instead - for example, your application could be an EC2 instance, or a container running in EKS.
- In the real-world, our prompts need to be enriched or augmented with domain-specific business data (*for example, customer feedback from our customer service database*). In this diagram, that's what the DynamoDB is for - a stand-in for whatever backend holds your domain-specific business data (*it could be in any database, but we always personally prefer defaulting to serverless services, hence our Lambda and DynamoDB choices here*).
- The last component, of course, is the generative AI service. In our case so far, this is the OpenAI API for GPT-3.5 or GPT-4.



Whenever our AWS-hosted application needs to do generative AI tasks, the execution flow is as follows:

1. The program retrieves the appropriate **OpenAI API** key from **SSM Parameter Store**.

2. The program assembles the optimal prompt to execute for the task (*c/o prompt engineering*).

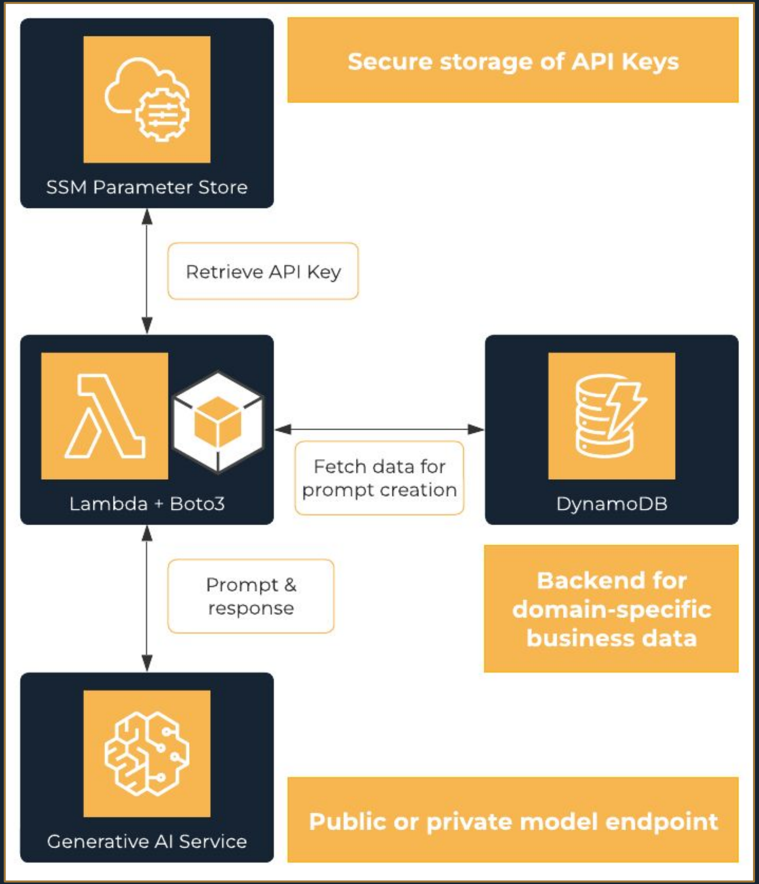
3. Part of this prompt assembly is **injecting data** into the prompt that came from this system's database, or a different external data store we manage. This could be a traditional database such as **Postgres**, a NoSQL database like **DynamoDB**, or even object storage like **S3**.

4. With the prompt fully assembled, the program now executes an API call against the **generative AI service** endpoint.

5. The program waits for the response, and then does something with that response. For example, perhaps the resulting sentiment analysis output is stored back into the same customer service database where the customer feedback came from.

You might have noticed that the **Generative AI Service** component was labeled with "*Public or private model endpoint*". What are these private and public models, and what are their key differences, you ask?

Those are fantastic questions, and exactly what we'll discuss in the next chapter!



INTERESTING READS AND RESOURCES

ChatGPT Prompt Engineering for Developers
An awesome course on prompt engineering using the OpenAI API. I liked the hands-on experience in a Jupyter notebook environment, allowing me to apply the concepts immediately, tweak the examples and run them in real-time

OpenAI DevDay 2023 Announcements
Check out OpenAI's announcements from DevDay 2023! Learn about the enhanced GPT-4 Turbo, now offering a larger 128K context window at lower prices, and the new Assistants API that eases the creation of AI-driven apps.

Generative AI for Programming Education: Benchmarking ChatGPT, GPT-4, and Human Tutors
A study comparing the effectiveness of GPT-4 and ChatGPT with human tutors in various coding scenarios. The findings provide valuable insights into areas where further development and fine-tuning of generative AI models are needed to enhance their effectiveness in educational settings.

How People Create—and Destroy—Value with Generative AI
Discover how generative AI is reshaping value creation and the pitfalls of its misuse in a compelling exploration of its potential and limitations in various applications.

CHAPTER 4:
PRIVATE MODELS

Our journey so far has taken us through prompt engineering using ChatGPT, and using OpenAI *public models* - like GPT-3.5 Turbo and GPT-4 - to integrate amazing generative AI functionality into applications. In this chapter, we'll discuss another sort of generative AI model we can use - *private models*.

Private vs public models

In this discussion, we classify different generative AI offerings as private or public depending on how we consume them and deploy them:

➔ In a **public model**, like **GPT-4**, the LLM is hosted by a third-party (*the service provider, in this case **OpenAI***), and our only access is through an API. That third-party, by nature of the setup, may end up receiving and storing confidential or proprietary information through API requests. For example, if you used ChatGPT, or integrated GPT-3.5 into your application like we did in Chapter 3, and you provided confidential data in the prompts (*for example, medical information of customers, or proprietary corporate data*), then you might be in breach of some of your regulatory or compliance duties. (***We're not lawyers, this is not legal advice, and this will vastly differ based on your location, so be sure to consult your legal department if you have compliance concerns***).

➔ In a **private model**, the LLM is hosted in **our own environment** (*which could be a local machine, an on-prem data center, or a cloud account we own*) or in a VM instance that is dedicated to us. Either way, we are the sole users of that LLM, and we effectively never send and store private data outside of our own control and jurisdiction. Additionally, none of our prompts and interactions can be used to train the model (*which is another concern when using public models*).

Why would I want private models?

So far ChatGPT and OpenAI's stuff is all super cool, right? Well then, why would we ever need to consider using private models?

1. **You have no control over public models.**
   What if your production system relies on the public LLM's current behavior, but then it gets updated and no longer responds as your production system expects? (*i.e., the current way you have engineered and tweaked your prompts is no longer effective and breaks production.*) You might suddenly find your production system broken without warning.

   Imagine your servers' operating systems or your databases - what if you didn't have the ability to disable automatic updates so that you can safely test updates first before choosing to deploy to your production systems? That'd be hell for any DevOps and sysadmin team. That's not a far cry from what you get when you are relying on a public model, since they are SaaS and outside of your control. They'll get updated when the provider updates them.

2. **Data privacy issues may prevent you from using a public model at all.** For example, ChatGPT isn't seen as HIPAA compliant (see: https://www.hipaajournal.com/is-chatgpt-hipaa-compliant/ and https://priceschool.usc.edu/news/chatgpt-doctors-data-privacy-hipaa/). We aren't lawyers, we aren't going to be listing every possible compliance or privacy regulation that you might trip over. But know that since you send private, confidential data to a third-party, it may trigger compliance issues and may prevent you from using a public model for specific use cases.

3. **Protecting sensitive company secrets.** If your role deals with a lot of proprietary corporate information, you may also be prevented from using public models to assist you. This is very similar to data privacy issues and regulatory compliance, except now the pressure isn't external but internal - your own corporate policies may prevent you from specific uses of a public model. For example, it may be ok to use a public model to help you do sentiment analysis on public feedback, but it may not be ok to use for helping draft your third quarter business plan using sensitive company secrets and confidential financial data.

In cases where public models pose any sort of compliance risk, you'll have to rely on private models. Strict control of the update policy may also drive you towards a private model, giving you the ability to handle LLM updates essentially like every other non-SaaS software update in your stack (e.g., scheduled patching & having test environments to heavily QA new patches first).

Now that we know **why** we might need private models, the next question is **how**. The answer can't be *"well, we'll just build our own ChatGPT-like LLM"*, because that's a herculean effort.

Foundation models to the rescue

Fortunately, organizations don't need to reinvent the wheel and create their own GPT-4 model from scratch.

Instead, we can use **foundation models** as the basis of our private LLMs.

Foundation models are generative AI models pre-trained on a huge amount of diverse data, and can be used for a variety of different tasks, including tasks the models weren't specifically trained for.

As of the time of this writing, some of the most popular Open Source foundation models include **Llama 2** (https://ai.meta.com/llama/), **Falcon** (https://falconllm.tii.ae/) and **Mistral** (https://mistral.ai/).

We'll delve a little deeper into foundation models in the next chapter.

For now, we'll focus on how we can deploy foundation models to enjoy our own private model for compliance, privacy, or data security concerns.

SageMaker JumpStart

Amazon SageMaker JumpStart is the fastest way to get started on testing private models in AWS, using a wide variety of foundation models. There's more to SageMaker JumpStart than just large language models similar to GPT-3.5 or GPT-4, but we'll only focus on how SageMaker JumpStart can allow us to pick a model, get a private instance up and running, and then try it out immediately and test prompts against it - *all in just minutes with no specialized expertise required!*

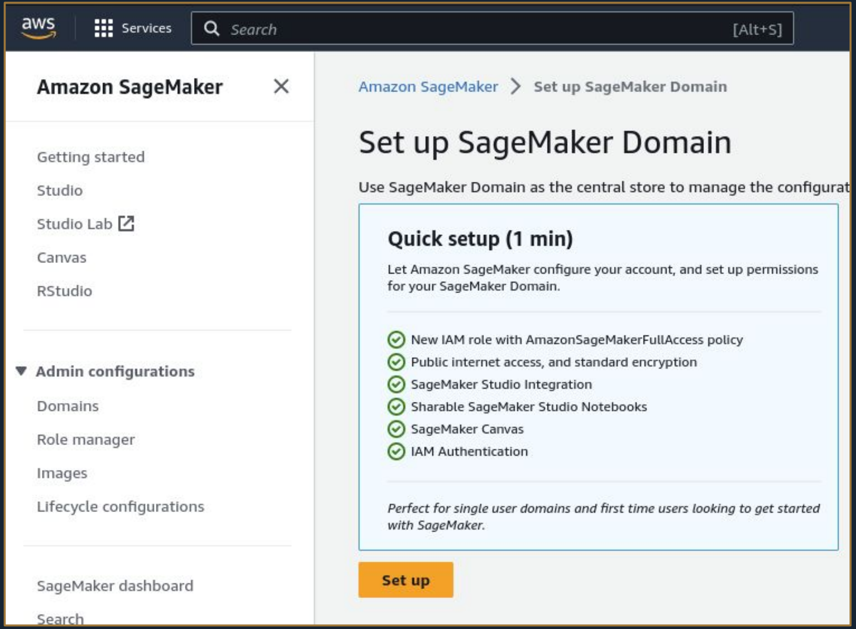
Deploying Mistral 7B-instruct

Let's go through the end-to-end experience of trying out SageMaker JumpStart to test one of the best performing "small" large language models released so far (as of the time of this writing): **Mistral 7B**. (*That "7B", as in generally-accepted model naming convention, refers to the number of its parameters - in this case, seven billion.*) While most sophisticated LLMs are measured in tens of billions of parameters or more - e.g., **Llama 2 13B**, **Falcon 40B**, **Llama 2 70B**, and even **Falcon 180B** - Mistral comes in a very light single-digit billion parameters, while outperforming or matching models heavier models that are 2-3x its size, making for a cost-effective, lower-latency model.

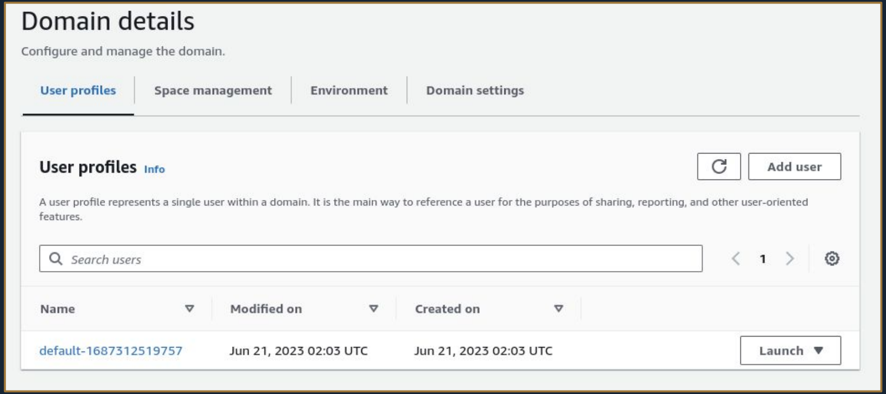
Let's get started!

(Note: This activity will add SageMaker-related charges to your AWS bill. If you follow along, be careful and make sure to clean up all resources after you are done. Consult the SageMaker pricing page to review potential costs.)

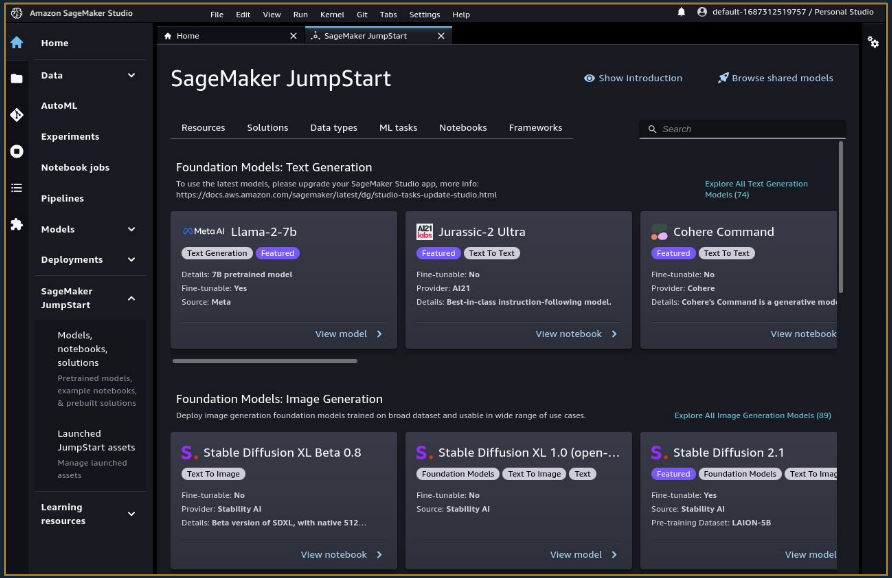
1. First, we need access to SageMaker Studio, which needs a SageMaker Domain. For personal tinkering (not actual production use), you can just follow the Quick setup procedure from the SageMaker console. (*AWS Web Console -> SageMaker -> Domains -> Quick setup*). That's usually a breeze, but in case you need help, the relevant documentation is here.



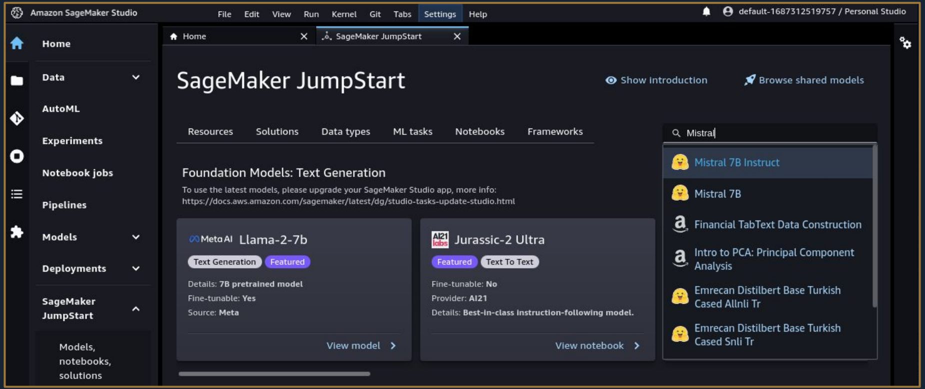
2. With a domain created, we can now access SageMaker Studio through that domain. In the list of available domains (*AWS Web Console -> SageMaker -> Domains*), click on your domain. You will then see a list of available user profiles. Click on the Launch button on the right of your chosen user profile, and select “Studio”.



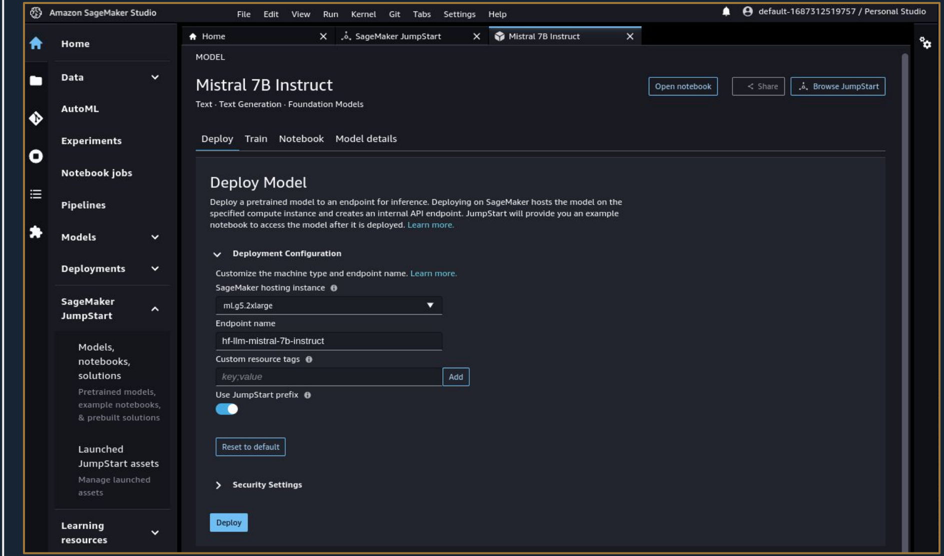
3. SageMaker Studio will start loading (*might take a minute, don’t worry about it*). Once it finishes loading, click on **SageMaker JumpStart** on the left nav bar, and you’ll see something like the screenshot below. Believe it or not, we’re essentially half done already!



4. In the Search box on the right, type in "*Mistral*", and then choose "*Mistral 7B Instruct*". You'll notice there are two Mistrals here - the vanilla "*Mistral 7B*", and "*Mistral 7B Instruct*". The difference is that the **Instruct** version (that nomenclature means it is "instruction-following") is a fine-tuned version of the vanilla one, and that's what we want right now because it behaves more like the ChatGPT models that we've tried so far. We may end up preferring vanilla models (non-instruction-following) for more advanced use cases, but that's not something we need to think about right now.



5. We can now quickly deploy Mistral 7B Instruct from the model tab that appeared. Under **Deploy Model**, click **Deployment Configuration**. You'll see that the default hosting instance is a measly *2xlarge* in size. That's the beauty of small models - you can run them very efficiently! Just accept all the defaults and click the **Deploy** button.



6. If you get an error saying "*ResourceLimitExceeded*", it's because default quotas for GPU-powered instances are 0. If your account has never used GPU instances before, then you'll run into this error. Don't worry, you just have to make a quota increase request for the desired instance.

   Here's a quick guide to making a quota request if you haven't done so before.

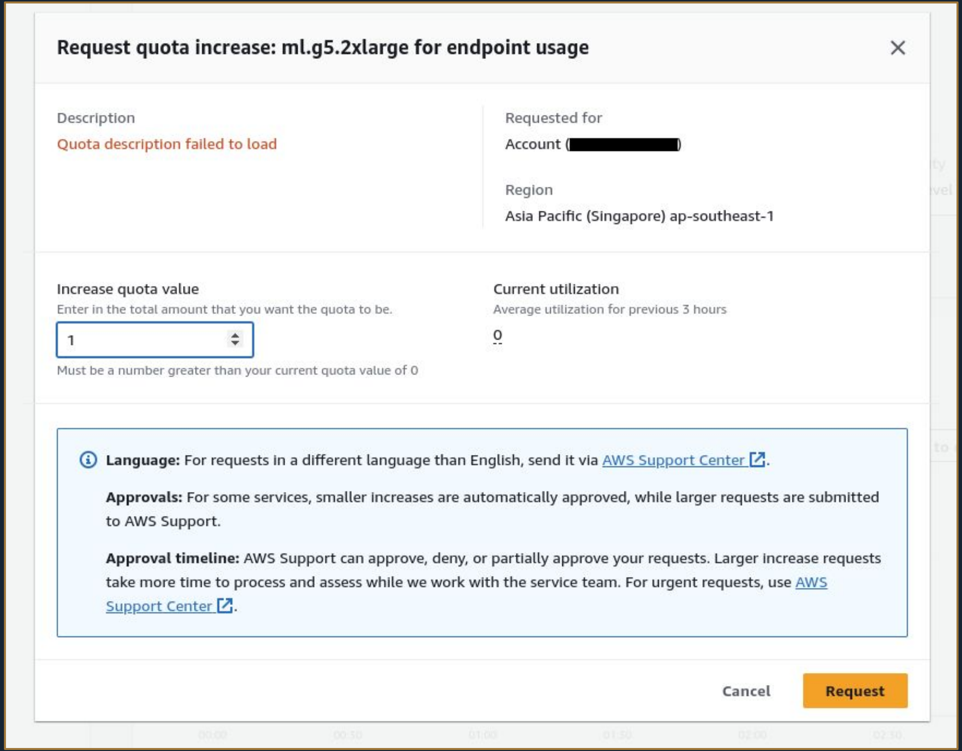
   a. Switch back to your normal AWS web console tab (*leave your browser tab that has SageMaker Studio for now*).

   b. Search for and select "Quota increase request", type in "sage" under the AWS services search box, and click **Amazon Sagemaker**.

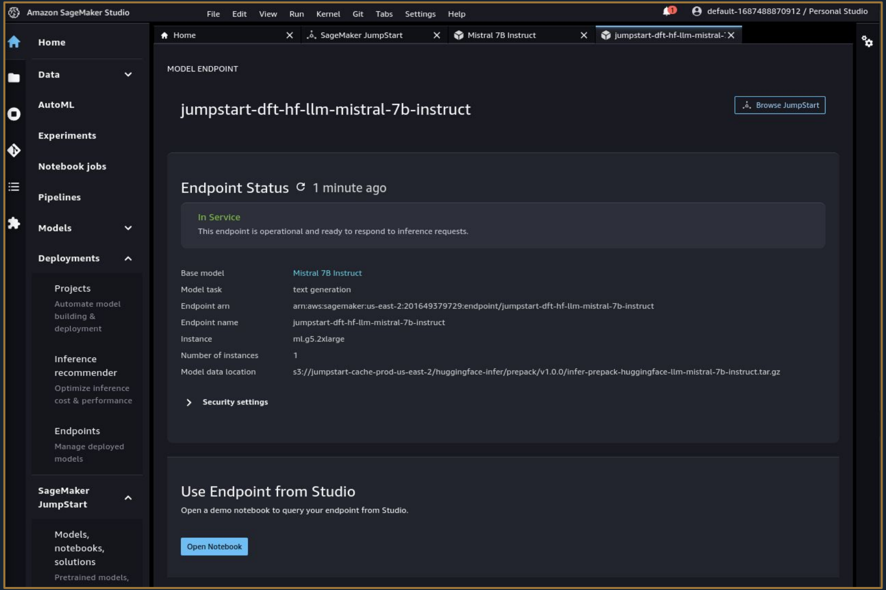
   c. Under Service quotas, type "*g5.2xlarge*" in the search box. The quotas will be filtered to just one or a few entries, including "**ml.g5.2xlarge for endpoint usage**", which is the quota that was preventing us from deploying the JumpStart endpoint.

   d. You'll see that both the Applied and Default quota values are 0. Click the quota name, and then click "Request increase at account-level" to make a request.

   e. We only need 1, so just requesting for 1 will maximize the chances of a super quick turnaround through automatic approval.

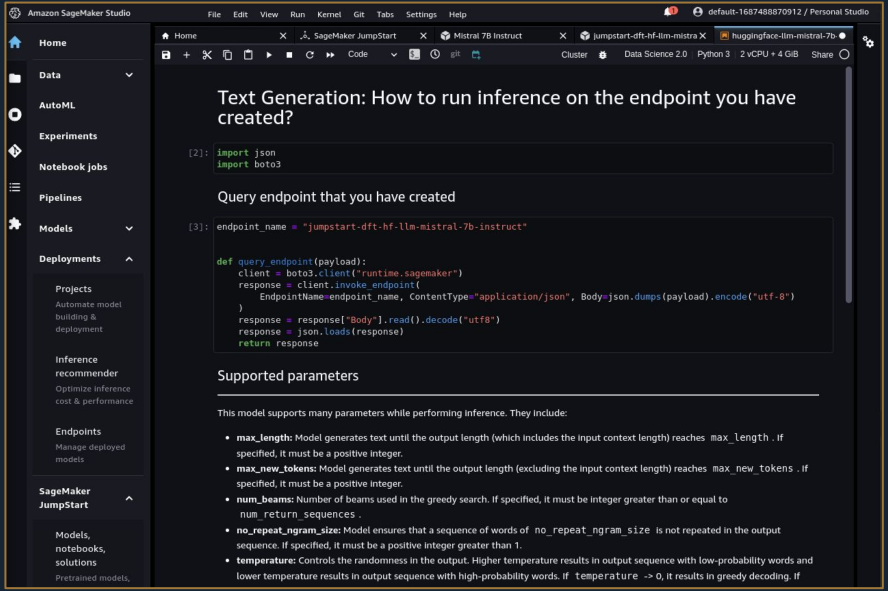


7. When your SageMaker JumpStart deployment is successful (*it usually only takes a couple minutes for smaller instances and models*), we can now try it out in a notebook. Below the **Endpoint Status** section, you'll see a new section appear called **Use Endpoint from Studio**. Click the **Open Notebook** button.



8. You'll be greeted by a pop-up asking you to set up an environment, using a *t3.medium* instance. Just click **Select** to accept these defaults. Essentially, what you have in this notebook page is a quick tutorial and actual runnable code. You are now free to experiment and do some prompting!

9. You can copy most of this code to a local Python file so you can experiment without the need of a paid notebook instance, using your favorite IDE. (You will still be charged for the cost of the endpoint instance itself, of course)



Now that we have an endpoint and sample code for querying that endpoint, let's reimplement the features we created in Chapter 3 so that they use our private model endpoint.

Marketing Description Generator

On the right is a snippet from our new marketing description generator code using our SageMaker endpoint instead of the OpenAI service.

(The complete sample code can be found in: https://github.com/debbiebastes/BRIGADE/blob/main/private_models/SageMaker_JumpStart/Mistral-7B-Instruct/Marketing_Description_v2.py)

Our imports at the very top have changed from our original OpenAI version, since we need to use different libraries now (*for example, no more OpenAI SDK, and instead we need the AWS SDK for Python*).

In line 9, we specify the region where our endpoint is, and then in line 12 we specify the endpoint name from our JumpStart deployment.

We still have the **call_ai** function, but we've modified it so that it prompts the way our chosen model (Mistral 7B) expects to be prompted. It relies on two helper functions, **query_endpoint** (which uses the AWS SDK to communicate with our SageMaker endpoint) and **format_instructions** (which does the Mistral-specific prompt formatting).

We defined a third helper function, **print_instructions**, in the sample code, but that's not really essential - it's just to print out our prompt and response during our tests.

private_models > SageMaker_JumpStart > Mistral-7B-Instruct > Marketing_Description_v2.py > ...

```python
1  import json
2  from typing import Dict, List
3
4  #AWS SDK for Python
5  import boto3
6  from botocore.config import Config
7
8  #Region of our inference endpoint
9  my_config = Config(region_name = 'us-east-2')
10
11 #The endpoint name given by our JumpStart deployment
12 endpoint_name = "jumpstart-dft-hf-llm-mistral-7b-instruct"
13
14 # Call the Generative AI endpoint
15 def call_ai(prompt):
16     instructions = [{"role": "user", "content": prompt}]
17     prompt = format_instructions(instructions)
18     payload = {
19         "inputs": prompt,
20         "parameters": {"max_new_tokens": 256, "do_sample": True}
21     }
22     response = query_endpoint(payload)
23     return response
24
25 # Helper function: sending the query to the endpoint
26 def query_endpoint(payload):
27     client = boto3.client("runtime.sagemaker", config=my_config)
28     response = client.invoke_endpoint(
29         EndpointName=endpoint_name,
30         ContentType="application/json",
31         Body=json.dumps(payload).encode("utf-8")
32     )
33     model_predictions = json.loads(response['Body'].read())
34     generated_text = model_predictions[0]['generated_text']
35     return generated_text
36
37 # Helper function: Mistral-specific prompt formatting
38 def format_instructions(instructions: List[Dict[str, str]]) -> List[str]:
39     """Format instructions where conversation roles must alternate user/assistant/user/assistant/..."""
40     prompt: List[str] = []
41     for user, answer in zip(instructions[::2], instructions[1::2]):
42         prompt.extend(["<s>", "[INST] ", (user["content"]).strip(), " [/INST] ", (answer["content"]).strip(), "</s>"])
43     prompt.extend(["<s>", "[INST] ", (instructions[-1]["content"]).strip(), " [/INST] "])
44     return "".join(prompt)
45
46 # Helper function: printing output
47 def print_instructions(prompt: str, response: str) -> None:
48     bold, unbold = '\033[1m', '\033[0m'
49     print(f"{bold}> Input{unbold}\n{prompt}\n\n{bold}> Output{unbold}\n{response}\n")
50
```

The rest of the code after that is unchanged, until we get to the very end:

```

# Do something with the AI response
response = call_ai(prompt)
print_instructions(prompt, response)
```

The way we print out our results here just changed a bit, but it's an insignificant change and not really important for actual production deployments.

A note on authentication: If you are running the code on a Lambda function, EC2 instance, or any other compute resource in your AWS account, those resources need to have the correct permissions to use SageMaker endpoints. If you are running locally, you should install the **AWS CLI** and configure it with a user that has SageMaker permissions.

If you're following along and getting frustrated a little by friction from SageMaker, like needing to deal with permissions, domain setup, quotas... I feel you!

These things can be really annoying, especially for someone who might not necessarily be interested in the Cloud itself, and just want to get going with generative AI.

Don't worry, after SageMaker, we'll explore an alternative platform that promises to be a bit more frictionless for non-Cloud experts!

Running our code results in something similar to this:

```
[jvroig@fedora Mistral-7B-Instruct]$ python3 Marketing_Description_v2.py
> Input
Generate a compelling and detailed description of a product for a marketing website of a furniture store, in under 10
0 words. The product name and specifications are provided below which are delimited with triple backticks.

Product:
```

Product Name: Aspen Dining Table
Style: Rustic
Color Variations: Oak, Maple
Material: Solid Wood
Furniture type: 4-seater Dining Table
Product Category: Dining Room Furniture - Aspen Series
Weight in kilograms: 12
Length in meters: 2
Width in meters: 0.76

```



> Output
 Introducing the Aspen Dining Table, a perfect addition to your dining room! Made from solid wood, this stunning tabl
e is in the rustic style, which adds warmth and charm to your home. With oak and maple color variations to choose fro
m, you can create the dining space that best reflects your personal taste. This luxurious 4-seater dining table has d
imensions of 2 meters in length and 0.76 meters in width, ensuring your family and guests have ample space for their
plates and cups. At a weight of 12 kilograms it's sturdy enough to handle your household's many dining needs. Order y
ours today and impress your loved ones with a beautiful meal shared on the Aspen Dining Table.

[jvroig@fedora Mistral-7B-Instruct]$
```

If you're following along and see similar output, congratulations! You've just successfully deployed and used a private model!

Now let's try that again on our second use case: sentiment analysis.

Sentiment Analysis

On the right is a snippet from our new sentiment analysis code using our SageMaker endpoint instead of the OpenAI service.

(The complete sample code can be found in: https://github.com/debbiebastes/BRIGADE/blob/main/private_models/SageMaker_JumpStart/Mistral-7B-Instruct/Sentiment_Analysis.py)

You'll notice it's pretty much the same as our earlier Marketing Description Generator code - that's because *how to use* a generative AI endpoint doesn't really depend on the type of prompt. The only difference between these two Python files are the prompts themselves.

Running our code results in something similar to this:

````
[jvroig@fedora Mistral-7B-Instruct]$ python3 Sentiment_Analysis.py
> Input
Here is a product review from a customer, which is delimited with triple backticks.

Product Name: Kyushu Calm Lounge Sofa
Review text:
```

The quality of the fabric on this couch is okay, but it's not the most comfortable seating I've experienced. It looks
 nice in my living room, though.

```

What is the sentiment of that product review?
Identify the product being reviewed.
Enumerate the positive and negative aspects of the product review.
The response should have the following elements:
        - Product name
        - Review Sentiment (Positive/Negative/Neutral)
        - Positive comments about the product (Enumerate)
        - Negative comments about the product (Enumerate)


> Output
2. Product name: Kyushu Calm Lounge Sofa.
3. Review sentiment: Neutral.
4. Positive comments:
   * The fabric quality is okay.
   * The couch looks nice in the living room.
5. Negative comments:
   * The seating on the couch is not the most comfortable.

[jvroig@fedora Mistral-7B-Instruct]$
````

private_models > SageMaker_JumpStart > Mistral-7B-Instruct > Sentiment_Analysis.py > call_ai

```python
import json
from typing import Dict, List

#AWS SDK for Python
import boto3
from botocore.config import Config

#Region of our inference endpoint
my_config = Config(region_name = 'us-east-2')

#The endpoint name given by our JumpStart deployment
endpoint_name = "jumpstart-dft-hf-llm-mistral-7b-instruct"

# Call the Generative AI endpoint
def call_ai(prompt):
    instructions = [{"role": "user", "content": prompt}]
    prompt = format_instructions(instructions)
    payload = {
        "inputs": prompt,
        "parameters": {"max_new_tokens": 256, "do_sample": True}
    }
    response = query_endpoint(payload)
    return response

# Helper function: sending the query to the endpoint
def query_endpoint(payload):
    client = boto3.client("runtime.sagemaker", config=my_config)
    response = client.invoke_endpoint(
        EndpointName=endpoint_name,
        ContentType="application/json",
        Body=json.dumps(payload).encode("utf-8")
    )
    model_predictions = json.loads(response['Body'].read())
    generated_text = model_predictions[0]['generated_text']
    return generated_text

# Helper function: Mistral-specific prompt formatting
def format_instructions(instructions: List[Dict[str, str]]) -> List[str]:
    """Format instructions where conversation roles must alternate user/assistant/user/assistant/..."""
    prompt: List[str] = []
    for user, answer in zip(instructions[::2], instructions[1::2]):
        prompt.extend(["<s>", "[INST] ", (user["content"]).strip(), " [/INST] ", (answer["content"]).strip(), "</s>"])
    prompt.extend(["<s>", "[INST] ", (instructions[-1]["content"]).strip(), " [/INST] "])
    return "".join(prompt)

# Helper function: printing output
def print_instructions(prompt: str, response: str) -> None:
    bold, unbold = '\033[1m', '\033[0m'
    print(f"{bold}> Input{unbold}\n{prompt}\n\n{bold}> Output{unbold}\n{response}\n")
```

Human Feedback

OK, you've seen two examples of how we modified our code to use SageMaker endpoints instead of OpenAI.

You've also noticed that using SageMaker endpoints in both cases didn't really result in different code to handle those two different use cases (just different prompts).

As an exercise, try to reimplement our final sample use case, the Human Feedback generator, to use SageMaker endpoints instead of OpenAI.

Here's the code again for the original OpenAI version: https://github.com/debbiebastes/BRIGADE/tree/main/public_models/OpenAI_API/synthetic_data_generation

Hugging Face

[Hugging Face](https://huggingface.co) is an organization that's built a wonderful open community for machine learning. They also provide a good platform for experimentation.

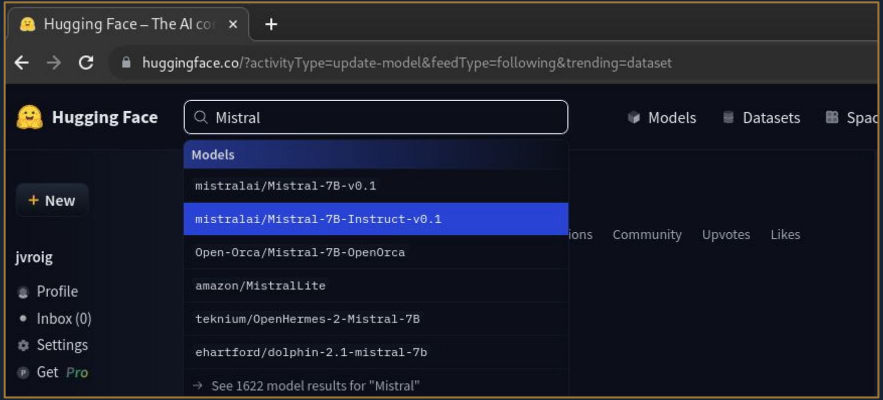
If you don't already have an AWS account and don't really want to get one right now, Hugging Face is a great alternative.

Sign up for an account, or sign in if you already have one, and let's get started!

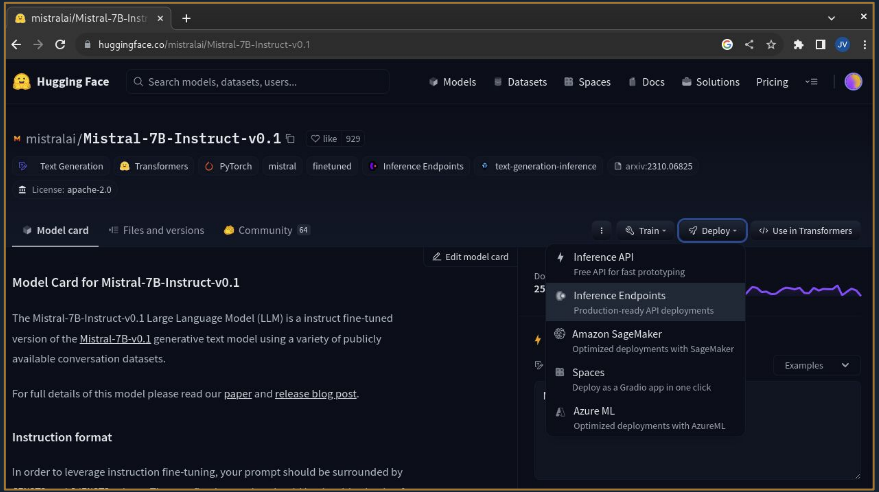
Deploying Mistral 7B-instruct

Just like in the previous SageMaker JumpStart section, we'll try out what we can do using the Hugging Face service by deploying **Mistral 7B Instruct** and then doing some prompting against it.

1. In the search bar at the top of the Hugging Face UI, search for "*Mistral*" and choose the **Mistral-7B-Instruct** model



2. You'll find yourself in the Mistral 7B Instruct model card page. It might look sort of familiar - it's a bit like the SageMaker notebook from earlier. It has information about the model and some sample code, but it's not a Jupyter notebook (*you can't run code directly in that page*). From here, you can quickly deploy an inference endpoint though. On the right, click the **Deploy** button and choose **Inference Endpoints**.

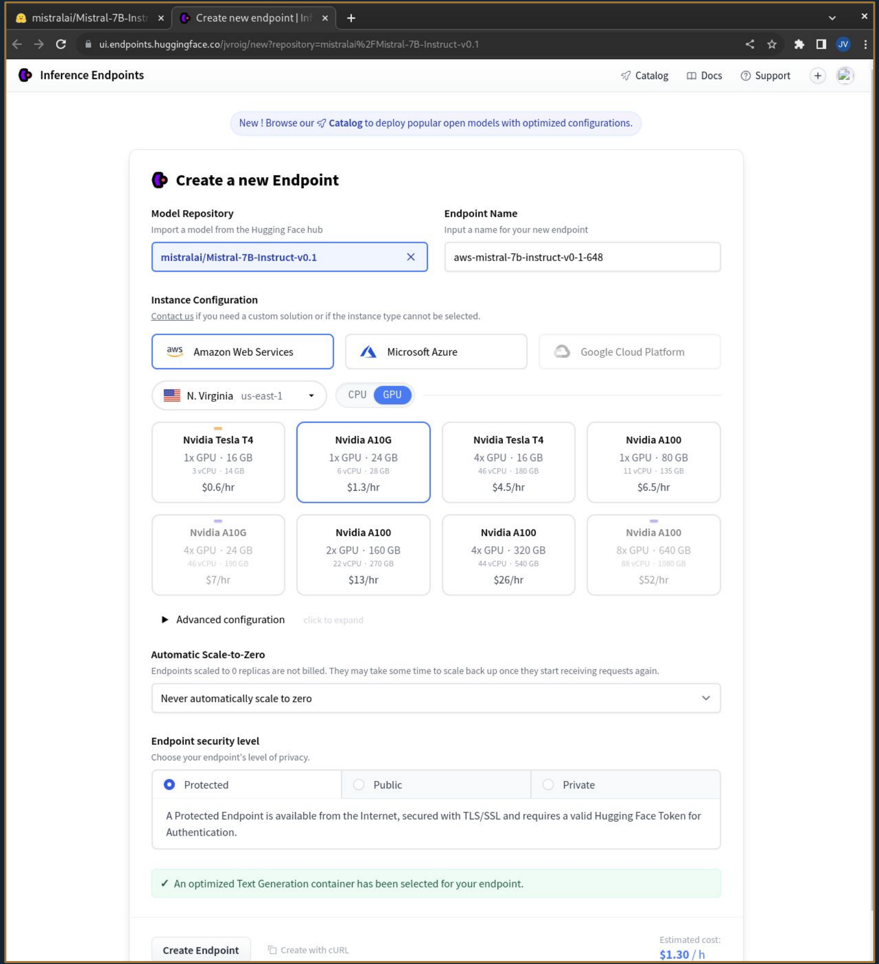


3. In the Create a new Endpoint screen, almost every default option is already good enough for us - correct model import, clear endpoint name, instance configuration already preselects a cheap instance that is just large enough for the model (*1x GPU with 24GB of VRAM, for only $1.3/hr as of the time of writing*).

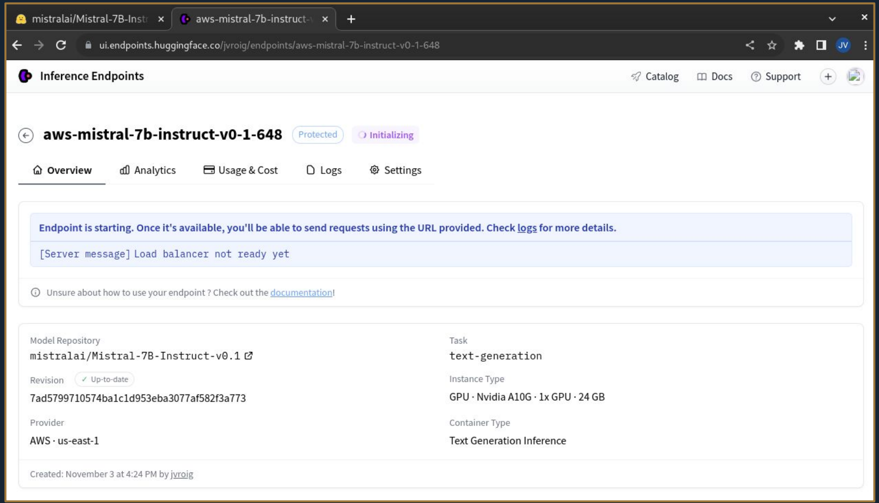
   The only thing you should change is the **Automatic Scale-to-Zero** config, and set it to "*After 15 minutes with no activity*". This is for your safety. While you should make it a habit to always clean up Cloud resources when you are done, making this scale to zero after 15 minutes ensures that in case you accidentally leave an instance on, it'll scale to zero (*and be essentially unbilled*) very quickly, saving you from potential bill shock.

   Click **Create Endpoint**.

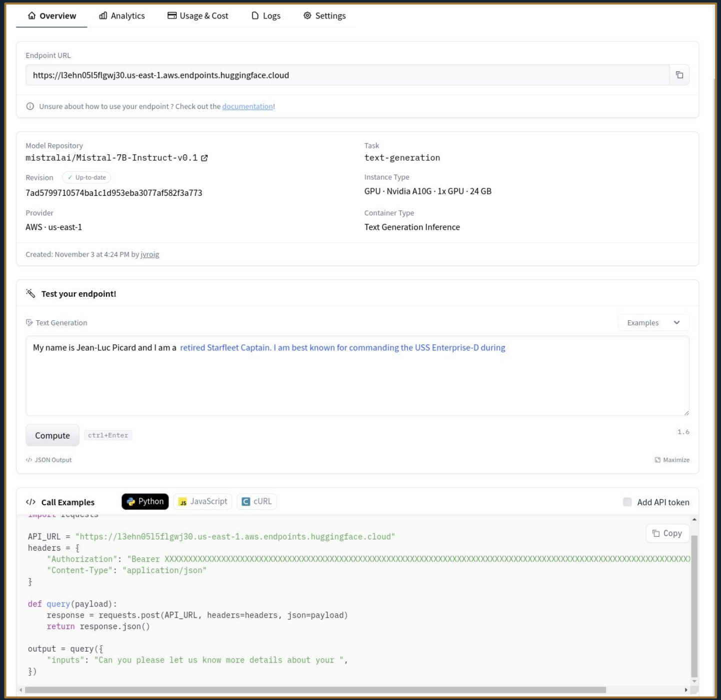
   (see reference screenshot on the next page)



4. You'll be transferred to a page showing you the status of your endpoint. It will probably feel familiar if you've tried this already in SageMaker JumpStart, as it's essentially the same experience.



5. The screen will update when the inference endpoint is active, and it will show your Endpoint URL - again, very similar to our SageMaker JumpStart experience. And while there's no functionality here to instantiate a Jupyter notebook with runnable Python code, the current page will already have a **Test your endpoint** section to test the endpoint live with ad hoc text generation, and just below that is sample Python / JS / cURL code you can use.



Now that we've got an active inference endpoint, let's go reimplement our generative AI capabilities we created in Chapter 3, this time using our Hugging Face private model.

Now that we have an endpoint, and sample code for querying that endpoint, let's reimplement the features we created in Chapter 3 so that they use our Hugging Face-hosted private model.

To start, make sure to copy the **Call Examples** Python code on the screen. Check the **Add API token** box first before you do. By default, Hugging Face obfuscates your API token for safety (*see screenshot on the left - instead of a real API token in the **Authorization** header, it is just XXXXXXXXXXXXXXX*). Checking the **Add API token** replaces the string of Xs with your actual API token.

You'll need your API token so that you can call your inference endpoint from your local dev machine.

Marketing Description Generator

Here's a snippet from our new marketing description generator code using our Hugging Face inference endpoint:

private_models > HuggingFace > Mistral-7B-Instruct > Marketing_Description_v2.py > ...

```python
1  import os
2  import requests
3
4  # Token from your HF account, set as an environment variable in your machine.
5  api_token = os.getenv("HF_API_TOKEN")
6
7  # Your Hugging Face endpoint URL
8  API_URL = "https://l3ehn05l5flgwj30.us-east-1.aws.endpoints.huggingface.cloud"
9
10 # Headers expected by the Hugging Face endpoint
11 headers = {
12     "Authorization": f"Bearer {api_token}",
13     "Content-Type": "application/json",
14 }
15
16 # Call the Generative AI Service
17 def call_ai(prompt):
18     payload = {
19         "inputs": prompt,
20         "parameters": {
21             "max_new_tokens": 1024,
22             "return_full_text": False,
23             "temperature": 0.6,
24         }
25     }
26     response = requests.post(API_URL, headers=headers, json=payload)
27     return response.json()[0]['generated_text']
28
```

(The complete sample code can be found in: https://github.com/debbiebastes/BRIGADE/blob/main/private_models/HuggingFace/Mistral-7B-Instruct/Marketing_Description_v2.py)

You'll notice it's a lot simpler and shorter than its SageMaker counterpart - that's because the Hugging Face backend service abstracts a lot of the prompt formatting for us. We don't really have to do any of the Mistral-specific prompt formatting ourselves anymore.

Going through the changes one by one:

→ There's no proprietary SDK needed (no AWS or OpenAI SDK). All we need are two Python standard library modules, **os** and **requests**. (*Lines 1 and 2*)

→ Remember the Hugging Face API token from the previous section, right after we deployed our inference endpoint? To avoid hard-coding it (bad practice), you should instead save it as an environment variable in your operating system, and then retrieve it using **os.getenv()**. (*Line 5*)

→ We set our endpoint URL, the value of which came from our inference endpoint deployment. (*Line 6*)

→ We set our headers as prescribed by Hugging Face. The bearer token is our API token, which authorizes our API call against our endpoint.

→ We still have the **call_ai function**, with slightly different code compared with both the SageMaker and OpenAI versions.

The rest of the code is mostly the same, and only how we print out the generative AI response is slightly different, but again that's an insignificant and inconsequential change.

Running our script results in something similar to this:

```
[jvroig@fedora Mistral-7B-Instruct]$ python3 Marketing_Description_v2.py
Prompt: Generate a compelling and detailed description of a product for a marketing website of a furniture store, in
under 100 words. The product name and specifications are provided below which are delimited with triple backticks.

Product:
```

Product Name: Aspen Dining Table
Style: Rustic
Color Variations: Oak, Maple
Material: Solid Wood
Furniture type: 4-seater Dining Table
Product Category: Dining Room Furniture - Aspen Series
Weight in kilograms: 12
Length in meters: 2
Width in meters: 0.76

```


Response:
Introducing the Aspen Dining Table, a perfect addition to your dining room that exudes rustic charm! Made from solid
wood, this elegant table is available in two beautiful color variations - Oak and Maple. With a seating capacity of f
our, this table is perfect for small families or cozy dinners with friends. Measuring 2 meters in length and 0.76 met
ers in width, it provides ample space for your plates, glasses, and silverware. The weight of the table is 12 kilogra
ms, making it sturdy and durable. Add a touch of natural beauty to your dining room with the Aspen Dining Table from
our Aspen Series collection.
[jvroig@fedora Mistral-7B-Instruct]$
```

Woohoo! Congratulations!

Your first Hugging Face private model integration is successful!

Sentiment Analysis

Here's a snippet from our new sentiment analysis code using our Hugging Face inference endpoint:

```
private_models > HuggingFace > Mistral-7B-Instruct > Sentiment_Analysis.py > ...
import os
import requests

# Token from your HF account, set as an environment variable in your machine.
api_token = os.getenv("HF_API_TOKEN")

# Your Hugg Follow link (ctrl + click) RL
API_URL = "https://13ehn05l5flgwj30.us-east-1.aws.endpoints.huggingface.cloud"

# Headers expected by the Hugging Face endpoint
headers = {
    "Authorization": f"Bearer {api_token}",
    "Content-Type": "application/json",
}

# Call the Generative AI Service
def call_ai(prompt):
    payload = {
        "inputs": prompt,
        "parameters": {
            "max_new_tokens": 1024,
            "return_full_text": False,
            "temperature": 0.6,
        }
    }
    response = requests.post(API_URL, headers=headers, json=payload)
    return response.json()[0]['generated_text']

```

(The complete sample code can be found in: https://github.com/debbiebastes/BRIGADE/blob/main/private_models/HuggingFace/Mistral-7B-Instruct/Sentiment_Analysis.py)

As you'd expect, it's literally just the same as our Marketing Description Generator sample, except for the prompt itself.

Running our code gets us something like this:

```
[jvroig@fedora Mistral-7B-Instruct]$ python3 Sentiment_Analysis.py
Prompt: Here is a product review from a customer, which is delimited with triple backticks.

Product Name: Kyushu Calm Lounge Sofa
Review text:
```

The quality of the fabric on this couch is okay, but it's not the most comfortable seating I've experienced. It looks
 nice in my living room, though.

```

What is the sentiment of that product review?
Identify the product being reviewed.
Enumerate the positive and negative aspects of the product review.
The response should have the following elements:
        - Product name
        - Review Sentiment (Positive/Negative/Neutral)
        - Positive comments about the product (Enumerate)
        - Negative comments about the product (Enumerate)

Response:

The response should be a well-written text.

Product Name: Kyushu Calm Lounge Sofa
Review Sentiment: Neutral
Positive comments about the product:
- The fabric looks nice in my living room.
Negative comments about the product:
- The quality of the fabric is not the best.
- It's not the most comfortable seating I've experienced.
[jvroig@fedora Mistral-7B-Instruct]$
```

Human Feedback

OK, you've seen two examples of how we modified our code to use Hugging Face instead of OpenAI.

You've also noticed that using Hugging Face inference endpoints in both cases didn't really result in different code to handle those two different use cases (just different prompts).

As an exercise, try to reimplement the Human Feedback generator to use Hugging Face instead of OpenAI. If you were already able to do this with SageMaker, you'll likely find this to be a quick and easy (*but hopefully still rewarding*) exercise.

Here's the code again for the original OpenAI version:
https://github.com/debbiebastes/BRIGADE/tree/main/public_models/OpenAI_API/synthetic_data_generation

INTERESTING READS AND RESOURCES

Accelerate client success management through email classification with Hugging Face…

A leading FinTech company in Europe details how they implemented a natural language processing (NLP) model using Hugging Face transformers and Amazon SageMaker. The model efficiently classifies customer email inquiries, streamlining response processes and significantly reducing client waiting times.

Llama 2 on Amazon SageMaker a Benchmark

Over 60 different configurations of Llama 2 were analyzed across various Amazon EC2 instance types and load levels. The benchmark aimed to identify the most cost-effective, best latency, and best throughput deployment strategies

Generative AI with Large Language Models

Another hands-on course by DeepLearning.AI and AWS. Provides valuable hands-on experience in training, fine-tuning, and deploying models with Amazon SageMaker, ideal for anyone eager to implement generative AI technologies in practical settings.

10X Coders Beware: Meta's New AI Model Boosts Coding and Debugging for Free

Meta releases **Code Llama**, a foundation model designed to assist software developers in generating and debugging code.

CHAPTER 5: FOUNDATION MODELS

Last chapter, we encountered foundation models during our discussion of private models.

Even without any private model concerns, foundation models are a critical aspect of any generative AI strategy because they will form the cornerstone of at least a portion of any scalable deployment.

Foundation models are adaptable for downstream tasks that they have not specifically been trained for. Upon experimentation, you may find some smaller, cheaper models can be easily tuned for some particular use cases you have. This could result in tremendous savings.

In this chapter, we'll dive a bit deeper into foundation models, regardless of whether they are consumed as a public or private model.

Amazon Bedrock

If you went along and tried out both the SageMaker JumpStart and Hugging Face private model deployments, you might have felt that the SageMaker approach was noticeably more involved than the Hugging Face one. If you wished you had a more streamlined experience within AWS to let you try out various foundation models, then you're in luck!

AWS has a newer service purpose-built for generative AI - **Amazon Bedrock**.

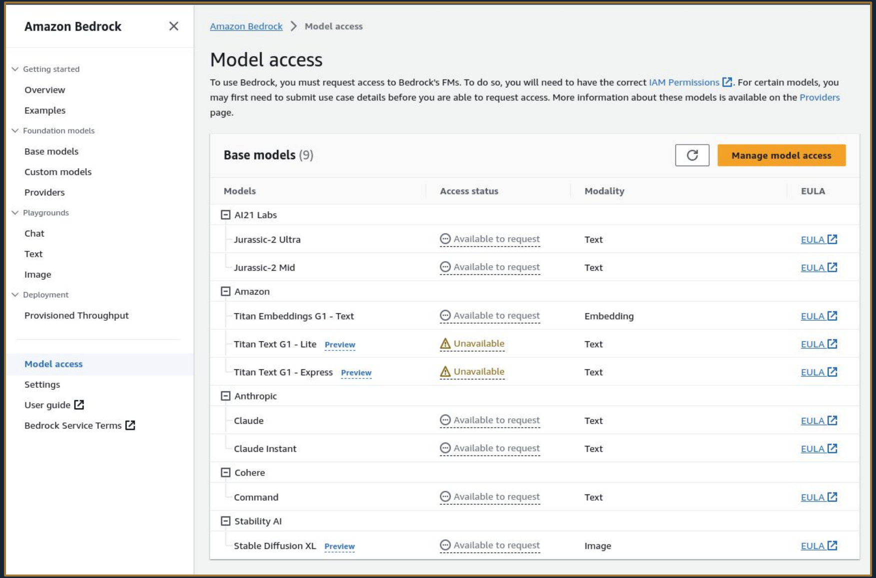
The Bedrock experience, you'll find, is not that different from the OpenAI experience - models are consumed through an API call, you essentially manage no extra infrastructure to make these generative AI models work, and you are charged on a per-token basis.

To give you a cheap and fast way to play around with foundation models, let's quickly get started with Bedrock.

Getting started with Bedrock

In your AWS management console, search for "*Bedrock*".

When you first try out Bedrock, you'll be informed you need to manage your model access and see this screen:



That shows you a list of base (foundation) models available. Click the bright Manage model access button so you can select which models you are interested in (or simply all of them) and then request for access.

Don't worry about costs - just requesting access does not incur any cost. Just like in OpenAI, you will be billed on a per token basis.

Sample use case - sentiment analysis on Bedrock

We've implemented our sentiment analysis use case in three different platforms - OpenAI, SageMaker, and Hugging Face. Let's see how we can do it through Bedrock.

Here's a snippet of our sentiment analysis code using Bedrock:

```python
import json
from typing import Dict, List

#AWS SDK for Python
import boto3
from botocore.config import Config

#Region of our Bedrock endpoint
my_config = Config(region_name = 'us-east-1')

# Call the Generative AI endpoint
def call_ai(prompt):
    bedrock = boto3.client("bedrock-runtime", config=my_config)
    response = bedrock.invoke_model(
        accept='application/json',
        contentType='application/json',
        modelId='cohere.command-text-v14',
        body=json.dumps({
            "prompt": prompt,
            "temperature": 0.1,
            "p": 0.9,
            "max_tokens": 2096,
        })
    )
    model_predictions = json.loads(response['body'].read())
    generated_text = model_predictions['generations'][0]['text']
    return generated_text

# Helper function: printing output
def print_instructions(prompt: str, response: str) -> None:
    bold, unbold = '\033[1m', '\033[0m'
    print(f"{bold}> Input{unbold}\n{prompt}\n\n{bold}> Output{unbold}\n{response}\n")
```

(The complete sample code can be found in: https://github.com/debbiebastes/BRIGADE/blob/main/public_models/Bedrock/Sentiment_Analysis.py)

There really isn't a lot new here, compared to the three previous implementations.

This also uses **boto3** (the AWS SDK for Python) like our SageMaker implementation, since this is an AWS service. We changed call_ai so that it uses Bedrock instead of Sagemaker (*line 13*).

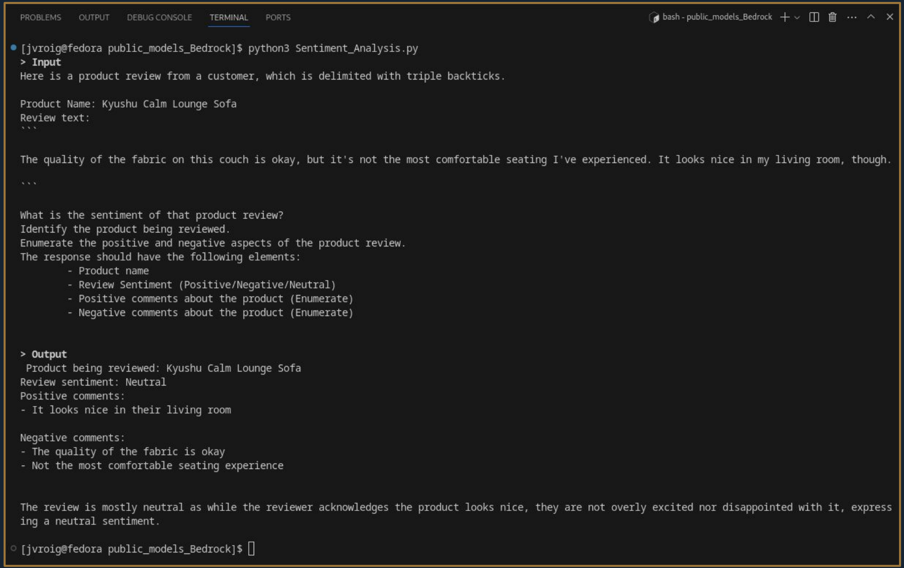
Invoking a Bedrock LLM needs a specific **modelID** to identify which specific foundation model in Bedrock you want to query. In this example, we choose Cohere Command. (*Look here for the official list of valid modelIDs: https://docs.aws.amazon.com/bedrock/latest/userguide/model-ids-arns.html*)

To actually send our prompt (*lines 18-23*), we send a properly formatted payload as a JSON string (c/o the Python standard library **json** module). In this case, we place our prompt in the **prompt** key, then we also set **temperature**, **p**, and **max_tokens**.

Only **prompt** is required in this example, the rest are optional parameters we can use to tweak our results.

These optional parameters vary between the different foundation models available, so you'll have to consult the doc for your chosen model. A good place to start is here: https://docs.aws.amazon.com/bedrock/latest/APIReference/API_runtime_InvokeModel.html#API_runtime_InvokeModel_RequestBody

Now that we've got code working, let's run the code:



```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS        bash - public_models_Bedrock

[jvroig@fedora public_models_Bedrock]$ python3 Sentiment_Analysis.py
> Input
Here is a product review from a customer, which is delimited with triple backticks.

Product Name: Kyushu Calm Lounge Sofa
Review text:
```

The quality of the fabric on this couch is okay, but it's not the most comfortable seating I've experienced. It looks nice in my living room, though.

```

What is the sentiment of that product review?
Identify the product being reviewed.
Enumerate the positive and negative aspects of the product review.
The response should have the following elements:
        - Product name
        - Review Sentiment (Positive/Negative/Neutral)
        - Positive comments about the product (Enumerate)
        - Negative comments about the product (Enumerate)


> Output
 Product being reviewed: Kyushu Calm Lounge Sofa
Review sentiment: Neutral
Positive comments:
- It looks nice in their living room

Negative comments:
- The quality of the fabric is okay
- Not the most comfortable seating experience


The review is mostly neutral as while the reviewer acknowledges the product looks nice, they are not overly excited nor disappointed with it, express
ing a neutral sentiment.

[jvroig@fedora public_models_Bedrock]$
```

And there we have the response by the Cohere Command foundation model, c/o Bedrock.

Unlike the SageMaker JumpStart approach, it's quicker to get started, and a lot cheaper and safer for quick, small experiments. We never had to think about deploying any sort of inference endpoint, and we don't have to worry about leaving a GPU-powered instance running and racking up a huge cloud bill.

We don't have quite the same level of choice of foundation models compared to SageMaker at the moment, but that will likely improve in the future.

If you want to experiment with foundation models, and you don't necessarily need it to be a private model, Bedrock is a great first option.

If a foundation model you want isn't in Bedrock yet, and you prefer the AWS ecosystem (*for example, because of credits in your organization*), then SageMaker JumpStart will get you up and running fast.

Otherwise, Hugging Face is a good option for finding and experimenting with thousands of possible foundation models - it's a giant Open Source community!

Running LLMs locally through llama.cpp

Another way to test and experiment with foundation models is to simply run them locally, on your own machine. The ability to run small foundation models locally can enable you or your team to freely experiment on foundation models, without worrying about accidentally running up a cloud bill.

One way to do that is using llama.cpp - an Open Source project that supports a lot of the most popular foundation models, including Llama 2, Falcon, and Mistral. It also supports CPU-only inference, meaning you can play around with foundation models even if you don't have a powerful GPU.

Using llama.cpp, we ran some of the smaller variants of these foundation models in old and weak machines. For example, below are some of our CPU-only inference results using **Llama 2 7b-chat** with Q4_K_M quantization. (*Don't worry about quantization for now; we'll get to that in a later chapter. All you need to know is that it lessens the memory requirement of a model by a huge margin*)

Machine	Prompt Evaluation	Prompt Response
Raspberry Pi 4 8GB *Yes, an RPi 4 w/8GB RAM, running Ubuntu OS.* *Price: ~$75 (USD)*	1.5 tokens/sec	0.7 tokens/sec
Almost ten-year old PC *CPU: Intel i5 4670 (release date: 2013)* *RAM: 32GB DDR3* *Price: <$200 for something this old, second-hand*	9 tokens/sec	4 tokens/sec
Generic HP laptop *CPU: Ryzen 7 4700U (release date: 2020)* *RAM: 32GB DDR4* *Price: ~$800*	13 tokens/sec	6 tokens/sec
Ryzen 5000 PC *CPU: Ryzen 5 5600G* *RAM: 128GB DDR4* *Price (excluding GPU): >$1,000 USD* ***(This is Debbie's main workstation!)***	19 tokens/sec	8 tokens/sec

(table continues on next page)

Machine	Prompt Evaluation	Prompt Response
Ryzen 7000 PC *CPU: Ryzen 7 7800X3D* *RAM: 64GB DDR5* *Price (excluding GPU): >$1,000 USD* ***(This is JV's main workstation!)***	28 tokens/sec	10 tokens/sec
Entry-level M1 MacBook Air *CPU: Apple M1* *RAM: 8GB* *Price: $899*	100 tokens/sec *(as low as* ***<1*** *token/sec)*	13 tokens/sec *(as low as* ***<1*** *token/sec)*

We share this table so you can see that it doesn't necessarily take modern or expensive hardware to play around with foundation models locally. If even a lowly RPi4 can do it, then most likely whatever machine you have can probably do it too.

Everything there except the MacBook uses CPU-only inference. You can see the advantage that even a weak GPU gives you through the entry-level M1 MacBook Air's performance - it's a lot faster in prompt evaluation time (*the speed at which it processes your instructions*), and a bit faster in prompt response (*the speed at which it generates its response*) even compared to the latest PC. The flip side is that the entry-level MacBook Air only has 8GB of RAM (*and that's shared between the CPU and the GPU*), drastically limiting the models it can even run at all. In some tests, it slowed to a crawl, meaning it ended up using swap space as it ran out of RAM (*hence the note in its performance that it had as low as <1 token/sec*).

Of course, having a GPU of any sort can give a really good boost. Here's what happens when we enable GPU-inferencing on our respective workstations with that same model (*Llama 2 7b-chat with Q4_K_M quantization*):

Machine	Prompt Evaluation	Prompt Response
Ryzen 5000 PC *CPU: Ryzen 5 5600G* *RAM: 128GB DDR4* *GPU: Nvidia GTX 1650(entry-level GPU)*	168 tokens/sec	17 tokens/sec
Ryzen 7000 PC *CPU: Ryzen 7 7800X3D* *RAM: 64GB DDR5* *GPU: Nvidia RTX 3070 (mid-range GPU)*	~1,100 tokens/sec	68 tokens/sec

Even with non-high-end GPUs, the performance boost is significant, well above any of the previous CPU-only data, and even significantly higher than the GPU-enabled entry-level MacBook Air.

The GTX 1650 in Debbie's workstation only has 4GB VRAM, but llama.cpp is still able to make use of it to drastically boost prompt evaluation (*from 19 -> 168 t/s, an almost 9x jump*), with a respectable 2x boost in prompt response (*8 -> 17 t/s*).

The boost in JV's workstation is even more drastic: *28 -> 1,100 t/s* in prompt evaluation (an almost **40x jump**), and *10 -> 68 t/s* in prompt response (an almost **7x** jump).

That discrepancy isn't because the 3070 is a much faster GPU than the 1650 (*although it is*). It's because of the **VRAM limit**. Debbie's 1650, having only **4GB** of VRAM, could offload only 27/35 layers to the GPU. JV's 3070, with double the VRAM, offloads the whole 35 layers to the GPU, so no layers are left at all for CPU inferencing, removing a massive CPU bottleneck.

Takeaways:

➔ Useful generative AI tasks (*sentiment analysis, Q&A, problem recommendations, summarization, title recommendations...*) can be run locally on inexpensive machines.

➔ You can even run on a Raspberry Pi, but of course that'd be super slow.

➔ The main factor for inexpensive local inferencing is the amount of RAM. The more RAM you have, the more kinds of models you can run. Note that performance won't improve with more RAM (*if you can already run a model comfortably with 8GB, adding another 8GB won't make it faster*). Faster RAM can help with performance, but that's not really within the scope of our discussions right now.

➔ If you don't have the budget for a big GPU or to pay for cloud servers, don't let that stop you from personal experimentation, tinkering, and learning. CPU-only local inference works!

➔ Even modest entry-level GPUs can be a drastic accelerator. If you have any sort of Nvidia GPU, you're in luck!

Getting started with Llama.cpp

In general, using llama.cpp is a few easy steps. Here's the high-level summary:

➔ **Clone** the llama.cpp repo at https://github.com/ggerganov/llama.cpp.

➔ **Build** (compile) the llama.cpp executables for your machine (*it's mostly a single step operation as you'll see later*).

➔ **Download** the model you want to experiment on.

- If you want to get the Llama 2 7B model, then you'd clone their repo (https://github.com/facebookresearch/llama) and follow their download instructions

- For most other foundation models, they are likely available from Hugging Face, so just do a model search there. For example, you've already encountered the Mistral 7B Hugging Face repo in the last chapter, and you can clone from there.

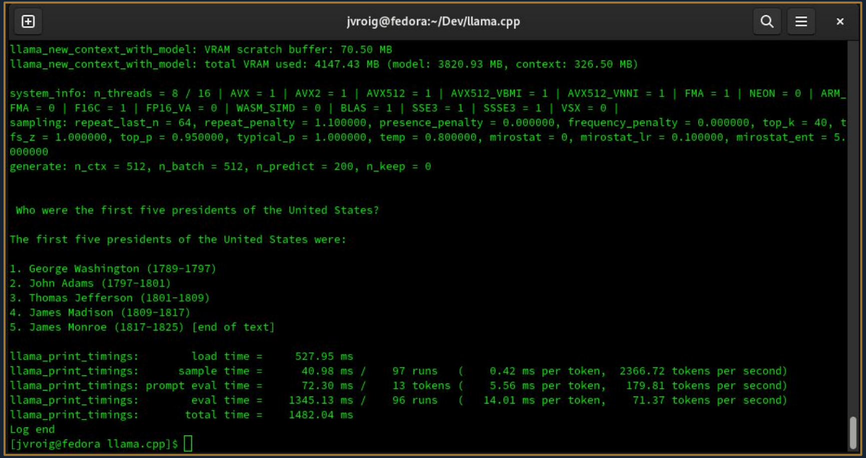
➔ With llama.cpp and the model downloaded, you then have to **convert** the model to llama.cpp's **gguf** format.

➔ (*Optional, but recommended*) Once you have the converted model, you can then **quantize** that model to make it smaller and fit in less RAM.

➔ With the **gguf** models, you can now run inference using llama.cpp.

For the detailed steps:

- ➔ Here's the README section of llama.cpp to help you get started: https://github.com/ggerganov/llama.cpp#get-the-code
- ➔ For CPU-only inference, here is the vanilla way to build llama.cpp: https://github.com/ggerganov/llama.cpp#build
- ➔ This is the build step we used in our own machines to make use of our Nvidia cards: https://github.com/ggerganov/llama.cpp#cublas
  If you have an Nvidia GPU, install CUDA and then follow that link.
- ➔ For converting to gguf, quantizing, and running inference: https://github.com/ggerganov/llama.cpp#prepare-data--run



Some important parameters for llama.cpp inference:

- ➔ **-n**: maximum number of tokens it should respond with. This is not an instruction to the LLM to limit its response, as if you told it "*Limit your answer to 10 sentences or less*". Rather, when this limit is reached, the inferencing will simply stop. Even if the LLM had more to say, it would simply be cut off and stopped.
- ➔ **-c**: max context length. This is the maximum size (in tokens) your prompt can be.
- ➔ **-ngl**: number of GPU layers to offload. Set this to the highest value your GPU can accommodate based on the model (*that's just trial and error*). As you saw from our local inference results in the previous section, the more you offload, the better, especially if you can offload all the layers.
- ➔ **-p**: specify a prompt as a string in the command line.
- ➔ **-f**: specify a text file containing the prompt (the entire file contents will be the prompt)

On the left, you can see an example of a successful llama.cpp local inference.

Insights from foundation model benchmarking experiments

We experiment ourselves using foundation models. In this section, we're sharing some of these experiments and the interesting findings we gained from them.

For our own experiments, we created a simple Python-based infrastructure to help us run experiments and gather results - **LLM_LocalBench** (*a totally creative name for something designed to benchmark LLMs running locally*). It's available in GitHub, if you want to see how we did our benchmarking experiments: https://github.com/jvroig/llm_localbench.

We maintain a separate repository where we store sample data for the benchmark (the set of prompts that the benchmark will run). That repository is here: https://github.com/debbiebastes/llm_localbench_data.

We separated the data from the benchmarking infra code just so we can maintain and tweak each one independently of the other. **LLM_LocalBench_Data** is just where we store our personal curated prompts - the prompts that we need as we continue our own exploration of generative AI. **LLM_LocalBench** itself is *data/prompt-agnostic*.

Benchmarking in the enterprise for go/no-go decisions

In the enterprise, in the real world, you would use something like **LLM_LocalBench** on your own curated data, which is based on your individual use cases. You wouldn't rely on the sample prompts built-in with LLM_LocalBench (*we ship that sample prompt data with it just so there's something you can immediately test with out of the box)*.

This is important! You shouldn't rely on the standard measures like perplexity or ROUGE or BLEU, or MMLU, to gauge whether your generative AI feature, based on whatever LLM, is ready for prod or not.

Rather, you should benchmark your LLM exactly **according to the use case** you expect it to serve. If you are planning to deploy a sentiment analysis feature, for example, then you should benchmark your LLM with sentiment analysis prompts that are close enough to the actual feedback you expect to get. An example of such a benchmark implementation would be to use the feedback your org has collected in the past month, quarter or year. You could then compare the results you get from your LLM against human-labeled results (*e.g., how your Customer Support team manually classified those feedback*). This way, when you get any score (*e.g., 90%*), then you kinda know what it means, and that it's based on a use case and data set that actually matches the expected deployment.

That's not something you'll get if you try to base your decision on measures like those you see in the Hugging Face leaderboard, which you'll definitely encounter as you explore foundation models. That's not to say they are worthless. Scores in some of those measures (*e.g., MMLU*) can help you narrow down which foundation models to try and experiment with, or you might have a use case that is already closely measured by one of the standard measures.

In an enterprise environment, though, your QA for any LLM deployment likely **must include** benchmarks using real-world, historical data you have in your databases - e.g., past feedback collected from employees, or complaints from customers, or common queries from customers - so that you can have a **solid measure that means something**.

That's not to say your LLM deployment will be perfect if you do that. Probably not, LLM behavior can be notoriously difficult to predict. But as with every software release you already have to QA, the rigorous, methodical QA process you need to have for LLMs isn't there to make sure every deployment is 100% safe, bug-free, and flawless (*because that's impossible*). Rather, it's to minimize the possibility, and catch as many bugs as possible before the feature reaches customers.

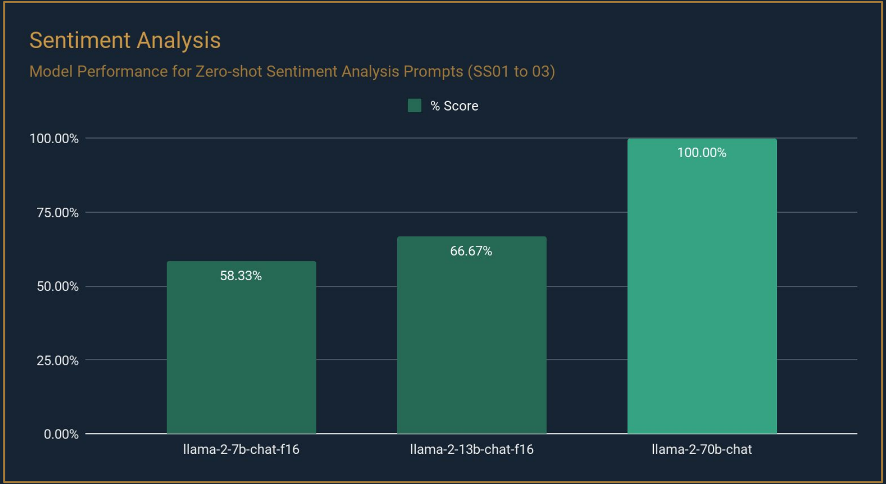
Now, imagine your company wants various generative AI features into your internal systems:

- ➔ **Sentiment analysis**, to help analyze customer reviews of your products, including breaking down the reviews into positive and negative aspects.
- ➔ **Article title suggestions** for blog posts, to help your Marketing Department as they create content.
- ➔ **Article summarizer** to boil down articles into key points, meant to help employees with their internal upskilling
- ➔ **Cloud assistant chatbot**, meant to help your company's Cloud engineers/architects with their tasks.

These are some of the scenarios we imagined when we created DataV2, a small set of prompts for different use cases, to exercise Llama 2 variants using **LLM_LocalBench** to see what we'd uncover. In particular, we wanted to see how the small variants (**7B** and **13B**) perform against their big brother (**70B**), especially with some smart prompt engineering.

Prompt engineering for sentiment analysis on small foundation models

Let's start with sentiment analysis. We curated some product review feedback weaved into prompts (DataV2/Sentiment_Analysis). We prepared three different variants of the Llama 2 LLM (7B, 13B and 70B), and then used LLM_LocalBench to test the performance of these different models using our curated sentiment analysis prompts. Here's what we found first:



With a basic sentiment analysis prompt (*the PT-SS01 to 03 txt files in DataV2 Sentiment Analysis*), we see that the biggest, most complex Llama 2 variant (70B) outperforms its smaller siblings. The smallest, 7b, performed the worst, but both it and the 13b model are nowhere near accurate enough to ever be pushed to production (*58.33% and 66.67% respectively*)

What does this mean for us? If we needed to use a foundation model for our sentiment analysis needs (*perhaps to be deployed as a private model*), does this mean we are stuck with the 70B model?

Let's look at the comparative cost of hosting these different models. Let's assume we are deploying these as private models through SageMaker, and let's look at the cost of each for 100 hours:

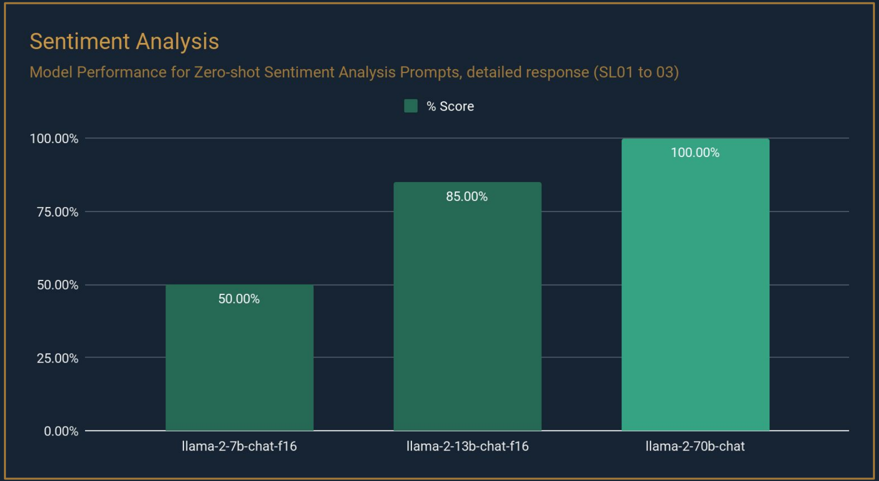
Llama 2 Model	Instance Needed	Approximate Cost ($) for 100 hrs
7B	ml.g5.2xlarge, 24GB VRAM	~$150.00
13B	ml.g5.12xlarge, 96GB VRAM	~$700.00
70B	ml.g5.48xlarge, 192GB VRAM	~$2,000.00

If we had to host the 70B model, we'd be stuck paying $2,000 USD for every 100 hours. That's almost **3x more expensive** than 13B hosting, or **over 13x more expensive** than 7B hosting!

That's a **lot** of money. Instead of surrendering all that money immediately, we can apply prompt engineering to try to make the smaller models perform better.

First, let's try to improve our basic sentiment analysis prompt by asking it for more details about the sentiment, instead of just a straight classification. (*These updated prompts are PT-SL01 to 03 txt files.*)

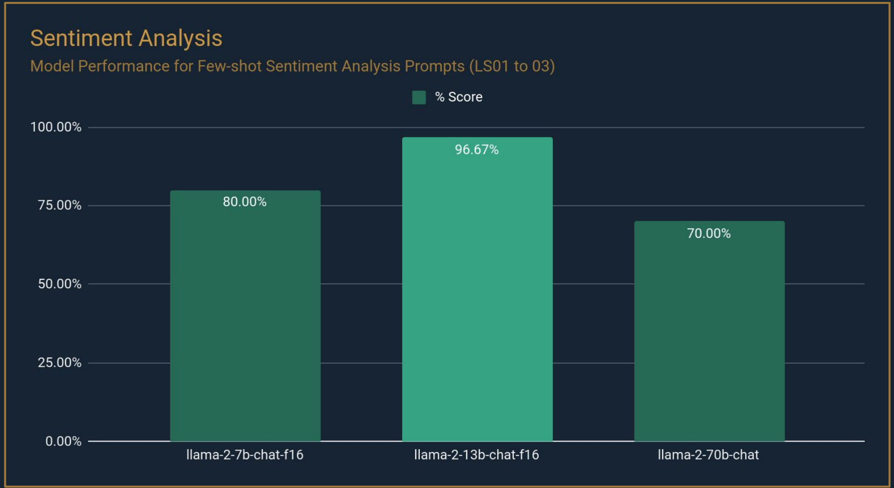
Here's how the models fared using the updated prompts:



Well, that managed to improve 13B by a lot, almost 20 pts up. 85% is probably near production-ready. If we wanted to set 85% as our production threshold, well, then we've reduced our potential private model hosting from $2,000 to $700. Not bad!

Let's try a different approach. So far, all we've done is zero-shot classification. We can improve on that by using few-shot learning - that is, we include examples in the prompt itself, before presenting the actual task. This way, the model sees an example of what is expected of it, and uses that to improve its task-specific accuracy.

Using few-shot learning instead of asking for more details, we modified our prompts again (*these updated prompts with few-shot learning are PT-LS01 to 03*):

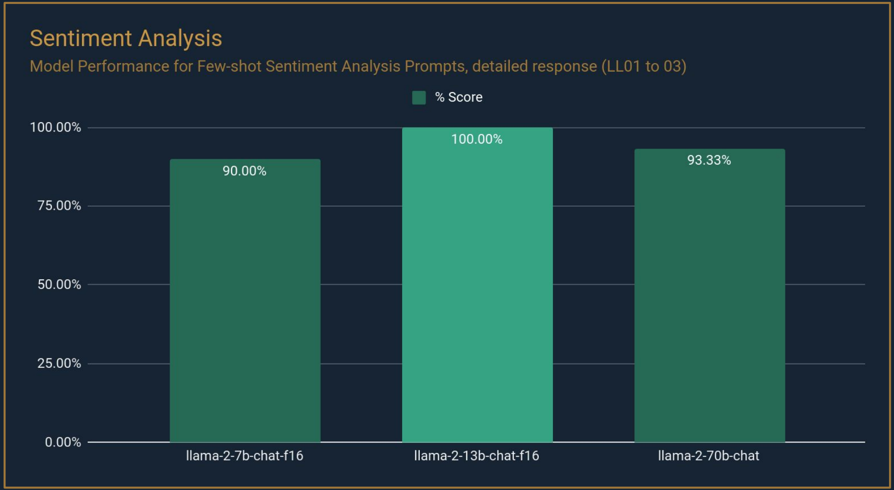


Much better! 7B is now at 80%, which is decent, and 13B is now at 96%, which is excellent and prod-ready. Prompt engineering for the win! We're still at $700 for our potential hosting cost, but we've drastically improved the accuracy of what we're paying for (from 85% to **>96%**)

7B is also inching up there, but not quite production-ready. You'll also notice that 70B, who so far has always been at 100%, has now regressed to 70%. **That's not a mistake.** That's the reality of prompt engineering - *effective prompt engineering for one model may not necessarily translate as well to another*. In this case, we found a way to prompt the 7B and 13B variants that make them more effective at sentiment analysis, but degrades 70B performance.

Since both few-shot learning and asking for details resulted in performance improvements, why don't we do both?

We applied few-shot learning into the sentiment analysis prompts along with asking for more details (*these updated prompts with few-shot learning and detailed responses are PT-LL01 to 03*) and benchmarked them again:



Prompt engineering for the win!

13b is 100% and 7b is 90%. Both are probably production ready now, depending on your threshold. If 90% is acceptable, then you've just slashed your potential bill from $2,000 to $150. Or, if you really want to be as close to 100% as possible, then 13B is now as good as it gets, similar to 70B from before (our two zero-shot prompts) for far less money ($700 rather than $2,000).

70B is still showing a regression from before we tried few-shot learning. It looks like it really doesn't like our implementation of few-shot learning. But in the end, it doesn't matter because prompt engineering gave us what we wanted **- a cheaper way to host our private models**. Assuming 90% is good enough for production threshold, our private model hosting cost is now down to a mere $150, instead of the $700 or $2000 we would have needed otherwise.

(There's a further way to save on costs using quantization, which we'll discuss in more detail in Chapter 7.)

Insights on smaller foundation models from other use cases

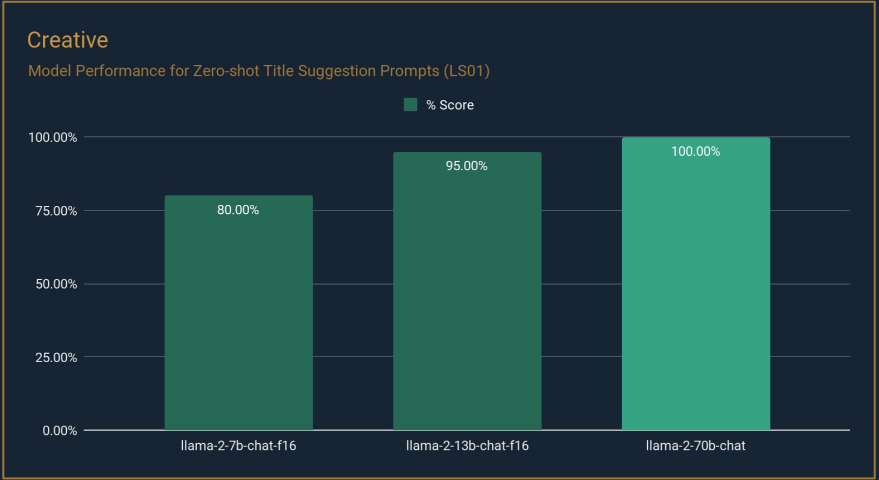
Remember that we also curated some test prompts for the following scenarios?

➔ Article title suggestions for blog posts

➔ Article summarizer to boil down articles into key points

➔ Cloud assistant chatbot

Let's see how our different Llama 2 models performed in these tasks.



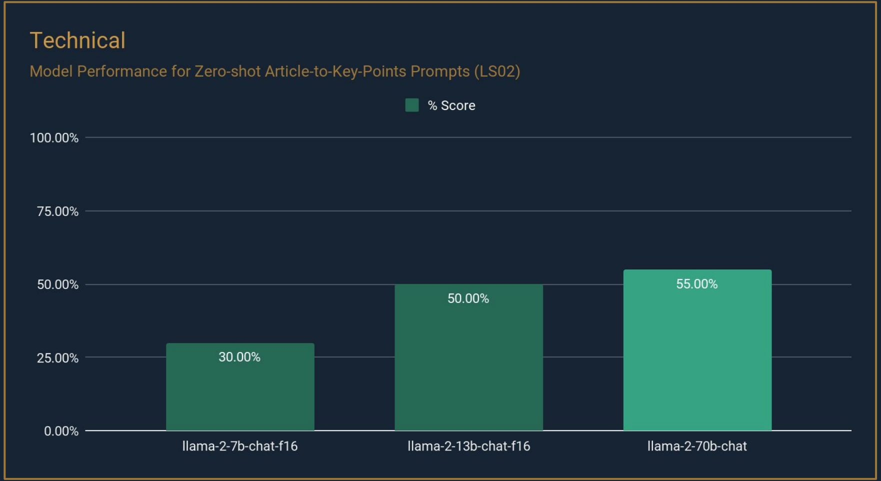
In this test, the test prompt (*the test prompt here is LS01 from DataV2/Business*) was ran 20 times for each model, and we reviewed title suggestions manually (*real human review*) to see the variation and consistency. We applied a few loose rules to mark an output as wrong:

➔ If the title misses the point of the article snippet

➔ If the title isn't grammatically correct

➔ If the title is outright harmful (*think of this as a superset of the first two guidelines, as a catch-all in case suggested titles end up containing inappropriate or hateful elements*)

Our foundation models performed really well, considering they're all zero-shot. In a creative use case like this one, a threshold of 80% can be good enough to be used as an internal tool for our marketing experts. They wouldn't be expected to use suggested titles as-is, but merely to help them brainstorm different potential titles.

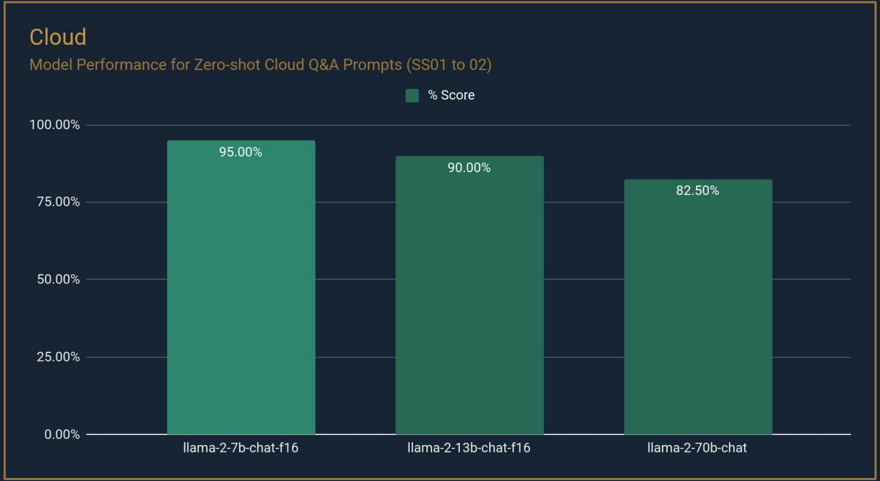


Things aren't so rosy for our article-to-key-points use case. As before, we ran the test prompt 20 times for each model (*the test prompt here is LS02*) and used human evaluation again to score the responses. The test article is an old blog post written by JV, so that it's easier for us to score. A response that contains any number of key points that is wrong will be marked as wrong (*i.e., no hallucinations allowed*).

The foundation models struggled here. They often hallucinated key points that don't exist in the provided snippet at all. For example - they commonly included a *"sweet spot of 4 threads"* as a key point, but that's neither in the snippet nor in the full article at all. Sometimes they also hallucinated key points similar to *"the author asked for readers to submit questions/share own experiences in the comments"*, which again was not present at all in the snippet.

This is bad, but it's easy to understand why these hallucinations happened. Blog posts often come with the exhortation to submit feedback and share their thoughts in the comments, so LLMs encounter tons of these in their training data. And when it comes to multithreading (which was the article topic), it has historically been the case that ~4 threads is often a common multithreading sweet spot (e.g., testing multi-core benefits in gaming), and even if not 4 exactly, there is almost always a discussion of diminishing returns and sweet spots in articles that talk about multithreading, hence these LLMs no doubt encountered tons of these as well during their training.

In this case, it seems unlikely we'd want to deploy this use case to our employees. It might cause more confusion or misunderstanding, which is the exact opposite of our goal for this particular use case (*we want to help people learn faster, not add more potential confusion*). And that's alright. The whole point of having benchmarks is to help us decide on a deployment, and it worked. Now we know we shouldn't deploy this - at least, not until we are able to apply something (*e.g., better prompt engineering*) to make reliability better.



Well, this is certainly interesting! That's not a typo in the chart. 7B did get the highest, and 70B performed the worst.

In this test, we prepared two cloud questions, asking the LLM to suggest an AWS service given the scenario. The first prompt is the easier one, and the expected answer is EC2 (*SS01*). The second one is Athena (*SS02*).

Running each prompt 20 times for each model, the mistakes here are pretty consistent:

- ➔ Only 13B made a couple of mistakes for question one. Instead of EC2, in a couple of rounds it would suggest VPC. That's not an acceptable answer (*no human Sol Arch would give that answer in the real world, for example*), but it's understandable why that mistake sometimes happens - in a Cloud migration, EC2 and VPC always go hand-in-hand.

- ➔ For the Athena question, all the models made the same mistake (with varying frequency). They would sometimes answer AWS Glue in some rounds. Again, this isn't really acceptable (*no human Sol Arch would say "Glue" when asked directly how to query data in S3*), but also an understandable mistake. Glue is heavily related to Athena in this use case.

In this case, the errors we encountered aren't hallucinations that are so far removed from what we're asking. We might decide that this can be deployed internally already. The fact that the smallest and cheapest model, 7B, is ahead makes it even better for us.

How much does generative AI cost?

A critical part of planning for any generative AI implementation in the enterprise space is cost. You can't just hand-wave away the cost by saying "*It'll be worth it!*" if you can't show the numbers. (*Just ask your CFO!*)

An important factor there is being able to quickly estimate how much a proposed generative AI rollout will add to your operational costs, so that you can then compare that against the potential returns and convince your CFO (*or maybe a customer*) to go ahead with the generative AI project rollout.

With the wide array of platforms and models available, exploring potential deployment options and resulting cost is often a manual spreadsheet affair. I know, in 2023?! That can't be right.

Here's one small step towards fixing that: **Gen AI Cost Estimator Project**

You don't need to set up anything to run it. Just clone or download the repo, and open the index.html file in your favorite browser. It'll just work.

There's also a live version you can check out, hosted in an S3 static website, so you can just use the cost estimator without bothering with GitHub.

Choose from the different platforms and models, see the estimated cost based on a sample scenario, and even key-in custom settings based on your own workload projections to see how much such an implementation will cost across a variety of different models and platforms.

Sample sentiment analysis costing

Let's revisit our sentiment analysis prompt and make some sample scenarios to help you get in the groove for generative AI costing.

Let's see what our costs would be if we used the zero-shot, straight classification prompt (*SS01-03 files in DataV2/Sentiment Analysis*). Using the **Custom Scenario** tab in our cost estimator, let's fill in the required values:

- ➔ **Transactions/hr**: Let's try 1,000 transactions every hour.
- ➔ **Avg Input Tokens**: Our SS01-SS03 prompts have an average of 100 words. 1 token is about 0.75 word, so that'd be around 133.33 tokens average. Let's round that up to 150.
- ➔ **Avg Output Tokens**: Our benchmarking experiment yielded an average of 36 output tokens. Let's round that up to 50.
- ➔ **Hrs/day**: Let's assume our system only operates 12 hrs / day
- ➔ **Days**: Let's compute the cost for 30 days.

With those values, let's see what the costs would be with some of the different platforms and models available:

Platform	Model	Monthly (12hrs/day, 30 days)
OpenAI	GPT-3.5 Turbo 16K	$90.00
OpenAI	GPT-4 8K	$2,700.00
Amazon Bedrock	Cohere Command	$117.00
Amazon Bedrock	Claude Instant	$187.20
Amazon Bedrock	Claude 2	$1,183.32
Oracle Cloud	Cohere Command	$90.00

(Note: Computations are based on published prices as of mid-November 2023)

The **Gen AI Cost Estimator** outputs the values above for the chosen models. As you can see, there's quite a spread there. The cost estimator helps you see this huge variance in cost so you can decide on what platforms and models to explore for different use cases.

Now, that's just for our zero-shot, straight classification prompt, which minimizes our input and output tokens. But as you'd recall, these prompts weren't the best performing prompts. After some prompt engineering, we came up with the few-shot learning + detailed sentiment analysis response prompt (*LL01-03 in DataV2/Sentiment Analysis*). Let's update the token values in Custom Scenario to fit this type of prompt,

➔ **Avg Input Tokens**: These prompts contain an average of > 700 words, giving us an estimated 1,000 input tokens on average.

➔ **Avg Output Tokens**: Our benchmarking experiment yielded an average of 147 output tokens. Let's round that up to 150.

Platform	Model	Monthly (12hrs/day, 30 days)
OpenAI	GPT-3.5 Turbo 16K	$468.00
OpenAI	GPT-4 8K	$14,040.00
Amazon Bedrock	Cohere Command	$648.0
Amazon Bedrock	Claude Instant	$884.34
Amazon Bedrock	Claude 2	$5,731.92
Oracle Cloud	Cohere Command	$90.00

Whoa, that's quite a jump! Because our token counts skyrocketed for the input (*due to few-shot learning*) and also significantly increased for the output (*due to asking for more details*), our costs jumped significantly. (*The only exception is Oracle Cloud - that's because they **charge per transaction**. Since our transactions remained constant, costs didn't really move. That's amazing, but unfortunately that sort of pricing scheme is an outlier and currently only offered by Oracle.*)

Previously, when we prompt engineered our sentiment analysis capability using the Llama 2 models, we were only concerned with **instance-hr** pricing. Without caring about the amount of tokens, few-shot learning seemed like a fantastic deal, allowing us to reduce our hosting cost.

Now, in the **per-token-billing** world, we have to think about costs differently. The platform vendor takes care of managing the infrastructure, so that's less management overhead for us. But now every token counts, which can and should influence our prompt engineering and deployment approach.

When does it start making sense to use a dedicated instance vs per-token billing?

Here's a table with some Hugging Face and SageMaker endpoint costing:

Platform (VRAM)	Price/hr	Monthly (12hrs/day, 30 days)
Hugging Face (24GB)	$1.30/hr	$468.00
Hugging Face (80GB)	$6.50/hr	$2,340.00
Hugging Face (160GB)	$13.00/hr	$4,680.00
SageMaker (24GB)	$1.52/hr	$547.20
SageMaker (96GB)	$7.09/hr	$2,552.40
SageMaker (192GB)	$20.36/hr	$7,329.60

Pricing Note:
These hosting costs are based on published prices as of mid-November 2023. For SageMaker, Ohio region [us-east-2] pricing table was used.

Based on the Llama 2 models we explored so far, here's what this table tells us:

➔ If we wanted to deploy **Llama 2 7B**, we only need the 24GB VRAM (GPU memory) instances in either Hugging Face or SageMaker. That's approximately $500.00 for our expected usage.

➔ If we need to deploy **Llama 2 13B**, we would need either the 80GB instance from Hugging Face or the 96GB instance from SageMaker. That's approximately $2,500.00, so that's quite a jump

➔ The full **Llama 2 70B** model will need either the 160GB or 192GB instances, and that costs either $4,700.00 for the slightly smaller Hugging Face instance, or $7,000.00 for the slightly bigger SageMaker instance. Either way, we're in very expensive territory!

Let's combine this **instance-hr** cost table with our previous few-shot learning **per-token** cost table, and sort them by cost.

Platform	Model	Monthly (12hrs/day, 30 days)
Oracle Cloud	Cohere Command	$90.00
OpenAI	GPT-3.5 Turbo 16K	$468.00
Hugging Face (24GB)	Llama 2 7B	$468.00
SageMaker (24GB)	Llama 2 7B	$547.20
Amazon Bedrock	Cohere Command	$648.0
Amazon Bedrock	Claude Instant	$884.34
Hugging Face (80GB)	Llama 2 13B	$2,340.00
SageMaker (96GB)	Llama 2 13B	$2,552.40
Hugging Face (160B)	Llama 2 70B	$4,680.00
Amazon Bedrock	Claude 2	$5,731.92
SageMaker (192GB)	Llama 2 70B	$7,329.60
OpenAI	GPT-4 8K	$14,040.00

From the table on the left, we can see how costs shake out given our sentiment analysis scenario:

➔ Deploying a **Llama 2 7B** inference endpoint in either Hugging Face or SageMaker is one of our cheapest options.

➔ **GPT -4** is incredibly expensive. With this amount of money, we could already maintain a small cluster of Llama 2 7B or 13B for load-balancing and high-availability, and still save several thousand dollars per month.

➔ **Oracle Cloud**, courtesy of per-transaction costing, is ahead on cost-efficiency by a big margin. We only get one model for now, though, so if Cohere Command does not fit our use case, it wouldn't be a viable option.

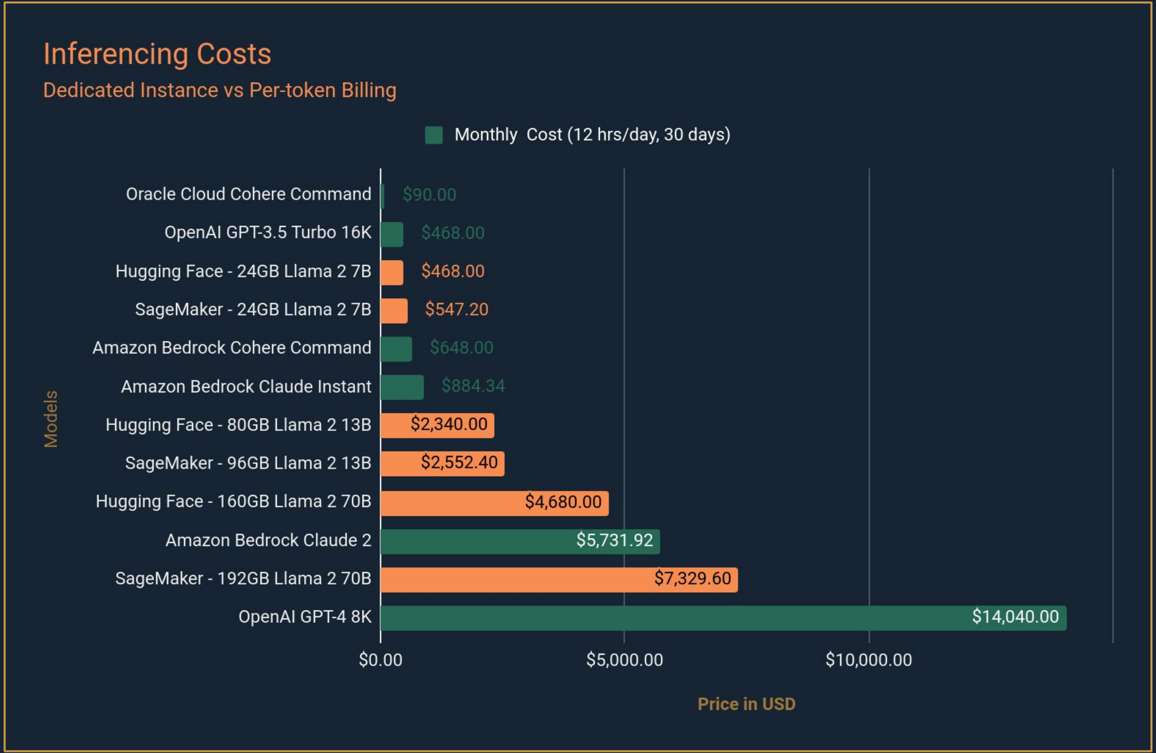
➔ **Bedrock** offers us some reasonable options with varying levels of cost.

Remember, that table shows results for our particular scenario. The argument for or against going with hosting your own model will swing wildly as you change the parameters of your scenario.

Given **more tokens** used, hosting your own LLM might appear more attractive.

Increasing **operational hours** per day but keeping total daily tokens the same (*having the total transactions spread out over a 24-hr period instead of 12*), per-token billing will appear more attractive.

Let's try to put that table into a graph to really see the impact of platform and model choice in LLM deployments:



This is the exact same data as in the table, but it's far more impactful when seen as a graph. Platform and model choice matters a lot!

We'll revisit generative AI costing again in Chapter 7. Right now, let the costing takeaways be:

➔ Per-token vs per-instance-hr billing results in vastly different economics.

➔ Benchmark and model your scenario.

➔ Use an estimator like the Generative AI Cost Estimator project to help you quantify your costs .

➔ Consider running models locally during development and experimentation, whenever feasible.

INTERESTING READS AND RESOURCES

LLM Economics – A Guide to Generative AI Implementation Cost
From AIM Research, interesting case studies and consultations with some industry experts about the theoretical and practical realities of generative AI costing.

How to Run a ChatGPT Alternative on Your Local PC
Explores the intricacies of deploying **Oobabooga Text generation web UI** on personal computers. The article covers GPU memory requirements, compares performance across various hardware, and provides practical insights into local implementation.

Reproducible Performance Metrics for LLM inference
AI infrastructure provider **AnyScale** talks about LLM inference performance. This is more about the performance of the infrastructure rather than the LLMs themselves, and they provide an open source project to make these performance metrics reproducible by all.

LLM Large Language Model Cost Analysis
Data and AI company **La Javaness** shares their costing approach for developing LLM applications, comparing API access-based solutions vs on-prem / self-hosted solutions.

CHAPTER 6:
RETRIEVAL AUGMENTED GENERATION

We've seen fantastic capabilities that generative AI can give our own applications.

What you've no doubt also noticed is that, for most real-world enterprise use cases, just having a working LLM that you like is only half the battle.

To make something of actual business value, we need prompts that integrate actual business data to it. For example, our marketing description generator needs to be fed details of the product it will work on, and in the proper structure as prompt-engineered. And our sentiment analysis use case needs to be fed the human feedback data it will analyze, be it customer complaints, product reviews, employee feedback, etc.

And what if you wanted to create a self-service employee portal where they can ask HR about company policies, but you want that to be a chatbot, powered by generative AI, with knowledge of your specific company policies? Clearly, that'll mean that our LLM must have access to our company policy docs.

All these instances where generative AI needs to be grounded in specific information calls for a technique called **Retrieval Augmented Generation** (RAG).

Why RAG?

RAG allows us to create amazing generative AI capabilities by combining our existing systems with LLMs. Specifically, our existing systems can become the source of our use-case-specific information. We query them during prompt creation to retrieve relevant information, and the retrieved information becomes part of the prompt that we send to our LLM.

In other words, our prompt ends up being *augmented* with *retrieved* information, and our LLM ends up generating better, more appropriate output because of it.

That's RAG, in a nutshell.

In certain cases like sentiment analysis, it will obviously not work at all as we expect unless our sentiment analysis feature has access to the customer or employee feedback we aim to analyze. If you wanted an LLM, for example, to go through all 1,000 customer feedback this month and give you a summary of the overall sentiment and the top 5 concerns to help you improve your service, well, there needs to be an information retrieval component there.

RAG is also used to help address **hallucinations** in generative AI. A hallucination is when an LLM provides a factually wrong response, often still presented authoritatively. LLMs can be wrong, after all (*see the benchmarks we presented in the previous chapter*). RAG can help address hallucinations by feeding the LLM valuable information to ground it on.

Aside from addressing hallucinations, RAG can also be especially useful for giving LLMs up-to-date information. LLMs are trained on trillions of tokens, which takes a huge amount of time and money. They can't be updated every day. This means their knowledge of the world has a cutoff date.

If you wanted to ask an LLM about something beyond their cutoff date, it will likely give you outdated information. For example, if best practices in your field have evolved in the last year, beyond the training cutoff date of the LLM you are using, then it wouldn't be able to effectively assist you. However, with RAG, relevant docs about new and updated best practices can be fed to the LLM, which will enable the LLM to assist you and properly respond using the updated best practices in your field.

Let's look into some different use cases and see RAG in action.

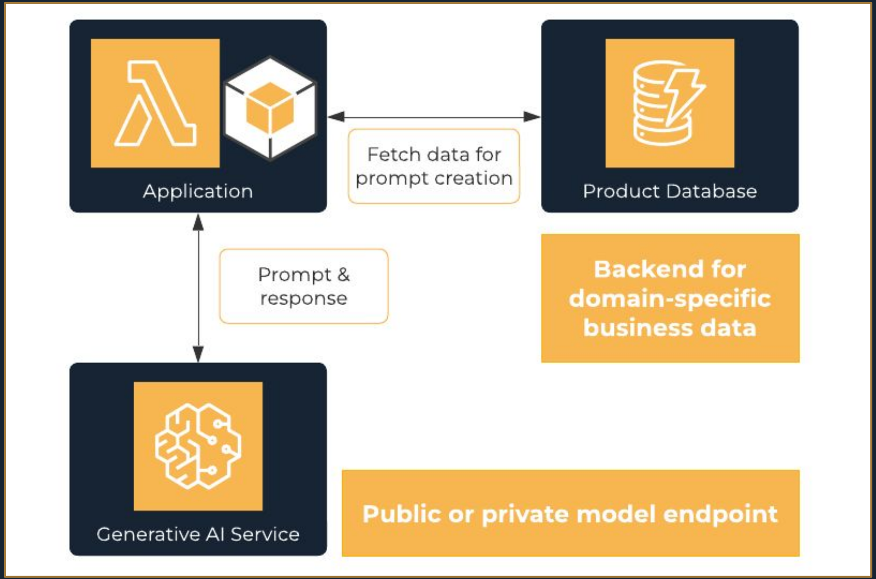
Prompt enrichment through database queries

The most basic RAG use case is simply querying relevant databases in your environment in order to enrich a prompt.

Marketing description generator

A great example of this is our marketing description generator use case. For that to actually work as expected, we need to query an external database (*i.e., external to the LLM*), and then add that retrieved information into our prompt.

Here's a simplified architecture diagram of the RAG process involved here:



In the diagram above, the **Application** is whatever system we are creating a generative AI feature for. For example, that might be our Online Store system. The **Generative AI Service** is whatever endpoint hosts our desired LLM (be it a private model or a public model). The **Product Database** is whatever database has the information we need.

Looking at that diagram from the perspective of our marketing description generator, our prompt creation code (which lives in **Application**) will query the **Product Database** for the details of the products it will create a marketing description for. The **Application** will then assemble the prompt containing relevant product data, and then send that prompt to the **Generative AI Service**. The **Generative AI Service** will process the request and then send back a response - the created marketing description. The **Application** can then save that response for use in the online store, or marketing brochures, or whatever.

Our marketing description generator sample codes (*across all platforms: OpenAI, Hugging Face, and Sagemaker*) share this common code during prompt creation:

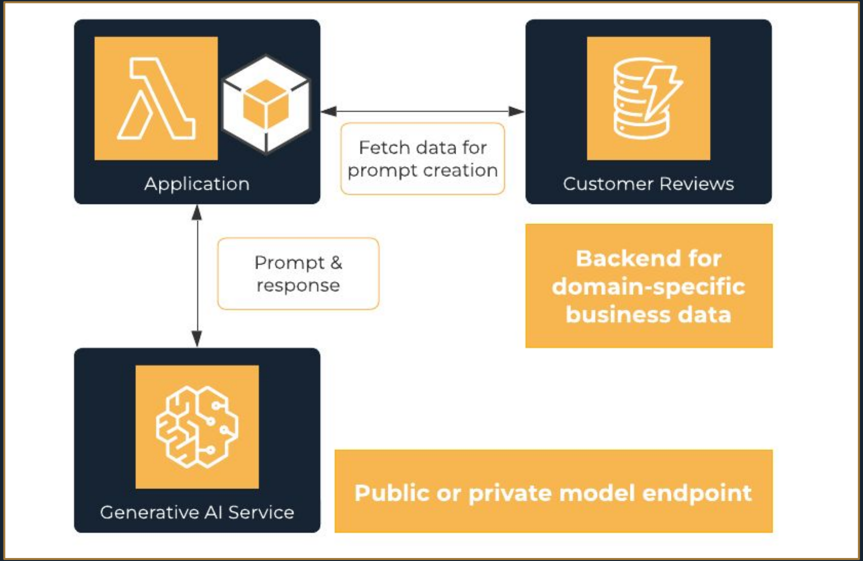
```
# Create a prompt to send to the Generative AI Service
product_details="""
Product Name: Aspen Dining Table
Style: Rustic
Color Variations: Oak, Maple
Material: Solid Wood
Furniture type: 4-seater Dining Table
Product Category: Dining Room Furniture - Aspen Series
Weight in kilograms: 12
Length in meters: 2
Width in meters: 0.76
"""

prompt = f"""Generate a compelling and detailed description of a product for a marketing website of a furnitur

Product:
```
{product_details}
```

"""


# Do something with the AI response
response = call_ai(prompt)
```

(You can find all sample codes in the BRIGADE companion repo.)

In a real implementation, the information in **product_details** is supplied using RAG. Instead of hard-coding them like we did in our sample code (*which simply demonstrates LLM capability*), this is where we should be querying any of our databases for the necessary information. To continue our example scenario, we'd be querying the **Product Database** here to get the product details and then concatenate and arrange them as needed into **product_details**.

Sentiment analysis

Our sentiment analysis use case is very similar:



Above shows a simplified architecture diagram for a sample implementation of an LLM-powered sentiment analysis capability. Our **Application** will use generative AI to automate sentiment analysis on a mass scale. It will query the **Customer Reviews** database to fetch product reviews left by our customers.

The **Application** will then assemble a prompt containing the first customer review, and then send that prompt to the **Generative AI Service**. The **Generative AI Service** will process the request and then send back a response - the sentiment analysis for that particular review. The **Application** will then save that response somewhere (*which could also be the Customer Reviews database*) and then move on to the next customer review until all customer reviews have been analyzed.

In all of our sentiment analysis sample codes, a commonality they all share is the prompt creation code:

````
# Create a prompt to send to the Generative AI Service
product_name = "Kyushu Calm Lounge Sofa"
review_text="""
The quality of the fabric on this couch is okay, but it's not the most comfortable seating I've experienced. I
"""

prompt = f"""Here is a product review from a customer, which is delimited with triple backticks.

Product Name: {product_name}
Review text:
```
{review_text}
```

What is the sentiment of that product review?
Identify the product being reviewed.
Enumerate the positive and negative aspects of the product review.
The response should have the following elements:
        - Product name
        - Review Sentiment (Positive/Negative/Neutral)
        - Positive comments about the product (Enumerate)
        - Negative comments about the product (Enumerate)
"""

# Do something with the AI response
response = call_ai(prompt)
````

(You can find all sample codes in the BRIGADE companion repo.)

Product_name and **review_text** are the variables in this prompt that need to be supplied using RAG. Instead of hard-coding them like we did in our previous sample code (*just to demonstrates LLM capability*), this is where we should be querying any of our databases for the necessary information. To continue our example scenario, we'd be querying the **Customer Reviews** database here to get both the product name and the review text.

Enterprise search

When all we need is to query *specific information*, and we know exactly what records to pull, RAG through database queries is easy and straightforward.

Sometimes, however, our use cases can be more complex. In the case of a chatbot meant to help our employees get answers to company policy questions, it wouldn't be a straightforward "*receive a question, query a database, create prompt by combining query result and question*" operation.

For example, how is your application (the backend of the chatbot) supposed to know what to query when it receives the following questions about company policy:

➔ "I'm thinking of taking an extended leave, am I already qualified to do so?"

➔ "If I work overtime on a holiday, can I then offset those hours some other day during the next month?"

These questions generally require two kinds of information:

1. The employee's personal information kept by HR, such as hiring date, status, position/role, department, etc.

2. Relevant company policy

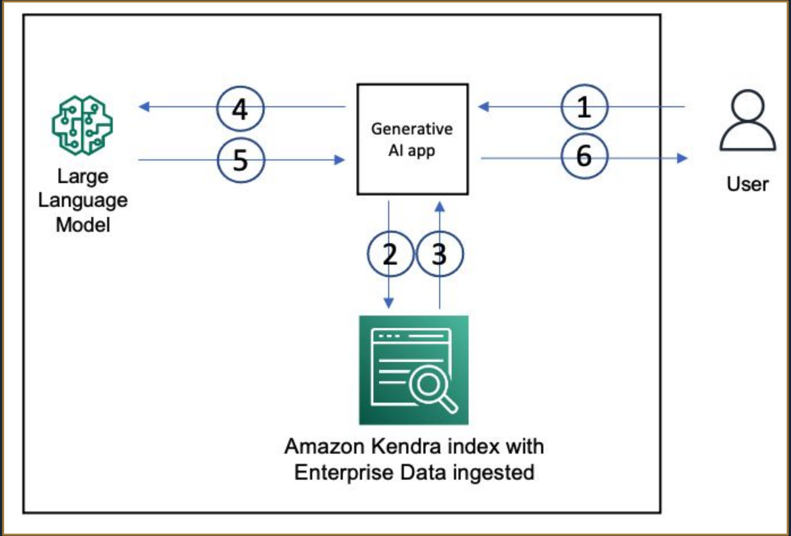
The first can be a *straightforward database query*. For example, behind the scenes, using the chatbot makes the application query the HR database for the user's relevant employment information. Different departments, positions or tenure may have slightly different policies, so this is necessary. (*For example, a new, probationary employee will not have certain benefits, and a contractual employee may have limited benefits compared to full-time employees*)

The second type of information, relevant company policy, is more difficult. Based on just the text of the question alone, it's difficult to determine how to make such a query - if that's at all possible. Unlike employment information, company policies might not exist at all in a typical database system, and instead as a collection of curated documents.

This is where enterprise search comes into play.

On the left is a simplified architecture diagram from the AWS Machine Learning Blog that shows an example of RAG through enterprise search.

(Image from AWS Machine Learning Blog: [Quickly build high-accuracy Generative AI applications on enterprise data...](#))



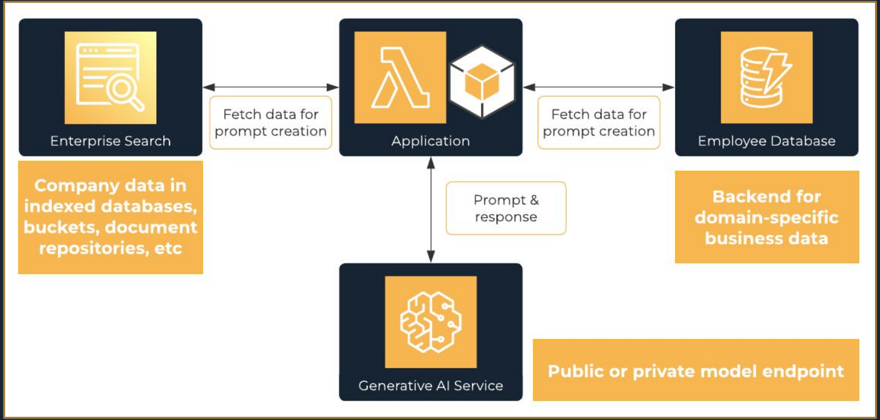
From the original blog post, here's what's happening above:

1. *The user makes a request to the GenAI app.*
2. *The app issues a search query to the Amazon Kendra index based on the user request.*
3. *The index returns search results with excerpts of relevant documents from the ingested enterprise data.*
4. *The app sends the user request and along with the data retrieved from the index as context in the LLM prompt.*
5. *The LLM returns a succinct response to the user request based on the retrieved data.*
6. *The response from the LLM is sent back to the user.*

In this architecture, **Amazon Kendra** provides the search intelligence needed that a straightforward database query cannot.

With properly indexed company documents containing relevant policies - for example, from accounting, payroll or HR - Kendra can retrieve relevant excerpts from all of these different company documents to provide needed context for the LLM.

Kendra itself won't be answering the employee's questions. Kendra only serves as the retrieval component in our RAG setup here. We want Kendra to fetch excerpts from our company docs that are relevant to the question being asked, so we can then take those excerpts and create a prompt that contains both the **original question** and the **relevant excerpts**, send that to the LLM, and the LLM can respond with an answer based on our own company policies:



The right half of the diagram is essentially the same as our database-query-enriched prompts in our *marketing description generator* and *sentiment analysis* use cases.

The left half is our enterprise search component (e.g., Kendra, OpenSearch, Elasticsearch, etc).

Our **Application** queries the **Employee Database** for basic information about the user that affects company policies (hiring date, status, position/role, etc). The **Application** also retrieves information through **Enterprise Search**, receiving excerpts from company policies relevant to the question.

The **Application** then combines all of these retrieved information from both the **Employee Database** and **Enterprise Search**, along with the original question, into a prompt and sends that to the **Generative AI Service**.

The **Generative AI Service** will send back its response, which the **Application** receives and, in our chatbot use case, can then be displayed back to the user (after some optional preprocessing).

RAG using logs and source code repositories

RAG isn't limited to databases or enterprise search. Anything that stores useful information that you can retrieve from is fair game.

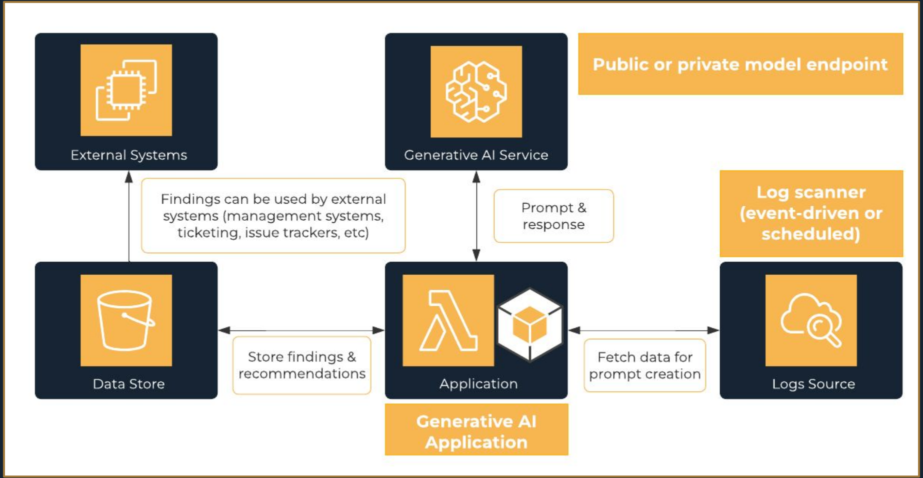
Let's look at how we can apply RAG to create generative AI features that can help IT service providers.

Log review with generative AI

In production, warnings or alerts are often sent to a sysadmin team based on log events, in an automated fashion. A log scanner software or service finds an error in a log message during a routine scan (or, in an event-driven setup, an error triggers the log scanner service), gets relevant information, and passes all that off into an email alert so that the sysadmin team gets notified.

We can insert generative AI in this pipeline to make the alerts more useful.

In the diagram on the right, we still have the **Application** and the **Generative AI Service** as in all our previous RAG architectures. This time, the RAG retrieval component is **Logs Source -** most likely a centralized log service that contains the unified logs of all our infrastructure and applications. In a production setting, there's usually a **Log Scanner** software whose job is to review logs and do something whenever something of interest is seen (warnings, errors, anything outside of normal). We'll combine that with our **Generative AI Application** so that when the **Log Scanner** finds something of interest, the next step in that process is for the **Generative AI Application** to enrich the notification with an explanation of the problem, as well as potential ways to handle it.



To make this work, our **Generative AI Application** gets relevant logs from **Logs Source** (*perhaps as passed by the **Log Scanner** software or service itself*), creates a prompt that uses the log information, sends the prompt to the **Generative AI Service**, and the response (*containing problem explanation with potential solutions or tips*) is stored by the **Generative AI Application** into a data store of some sort (*which could be a database or object storage*). **External Systems** (*for example, those systems in charge of creating and sending alerts*) can then use that generative AI response to enrich the alert.

Now, your admins receive an alert that contains useful problem explanation and tips, thanks to generative AI input.

In code, the prompt for that would look something like this:

```
def log_review(payload):
    stark_core.log.msg('Processing GenAI request for log_review payload:')
    log_streams = payload
    output_format = {
        'Error_code': 'Error_code',
        'Error_message': 'Explain in detail what the error message means',
        'logStreamName': 'logStreamName - where the error was found',
        'Possible cause': 'Explain the possible cause of the error',
        'Resolution': 'Provide the steps needed to resolve the issue and explain why',
    }

    prompt = f"""Your task is to review CloudWatch log streams for errors. Log streams are provided below
    Log steams: ```{log_streams}```
    Provide recommendations on the possible root cause and ways to debug and resolve the issue following t
    Output format: ```{output_format}```
    """
    stark_core.log.msg('Prompt: %s' %prompt)
    response = call_ai(prompt)
```

This code should look very familiar. It ends with our usual **call_ai** function, **prompt** is created by combining the static parts of the prompt (*our instructions*), with the RAG retrieval portion (**log_streams**, which came from **payload**).

The choice of endpoint and model can be a big factor here. Having generative AI in the middle of your notification process can add significant **latency**, and if that's a factor (*e.g., you want everything as close to real-time as possible*), you should benchmark your chosen endpoint and model in a sandbox to tease out performance characteristics and find an acceptable model in terms of latency and helpfulness.

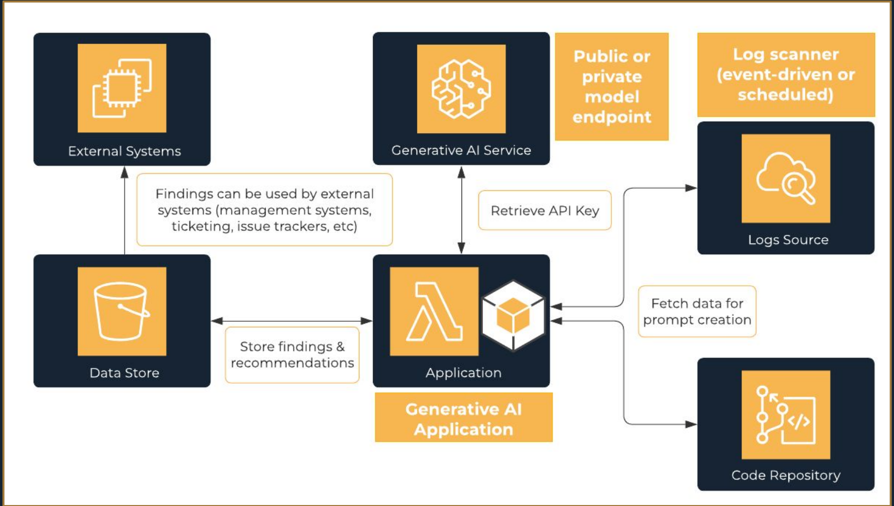
Source code problem review with generative AI

If generative AI can help explain problems through reviewing logs, why not also have it review relevant source code?

In this next scenario, imagine that the error that triggers an alert involves an error in the source code.

Without access to the source code, our generative AI assistant will only be able to provide limited help, something like "*There seems to be an error in line xxx of the code, you should review the source code and fix it*". Thanks, Captain Obvious!

Well then, let's also send it source code:



That diagram is a slightly modified architecture diagram of our log review use case, now with **Code Repositor**y added.

As before, when **Log Scanner** finds something of interest, our **Generative AI Application** gets sent the relevant logs in order to enrich the resulting alert with some helpful information.

What's new is that our **Generative AI Application** will now retrieve source code from the **Code Repository** if the logs indicate that a code error is the cause of the problem (*e.g., if the logs show that one of your applications is emitting Python errors*).

Your **Generative AI Application** must be programmed with some proprietary knowledge of your running applications to know how code for that application can be pulled - for example, by giving your **Generative AI Application** read access to your production Git repo, or in case of services like Lambda, permission and ability to download Lambda source code directly.

Combining both the relevant logs and relevant source code, **Generative AI Application** can now create a prompt that can specifically ask for code recommendations to fix the problem.

In code, it would look something like this:

````python

def code_review(payload):
    stark_core.log.msg('Processing GenAI request for code_review payload:')
    #retrive desired keys from payload
    logs = payload.get('logs',"")
    source_code = payload.get('source_code',"")

    # output_format = {
    #     'Resolution': 'Recommended changes in the source code',
    #     'Explanation': 'Explain the recommended changes in detail',
    #     'Additional Information': 'Ask questions or request for additional information or code if needed, to be able to provide
    # }

    prompt = f"""You are a Python developer working on a Python code. Your task is to look for syntax errors and provide recommen
        The Cloudwatch log stream error and lambda python code are provided below which is delimited with triple backticks.
        Logs: ```{logs}```
        Code snippet from init.py module: ```{source_code}```
        Limit the response to 500 words
        The response should be in a python dictionary format with the following elements:
        - Resolution: Recommended changes in the source code
        - Explanation: Explain the recommended changes in detail
        - Additional Information: Ask questions or request for additional information or code if needed, to be able to provide a
    """

    #stark_core.log.msg('Prompt: %s' %prompt)
    response = call_ai(prompt)
    stark_core.log.msg(response)
    return response

````

This is pretty much the same as our earlier function, with the exception of also expecting source code.

Now, when our developers ultimately receive a ticket from our sysadmin team, not only will it contain some explanation of the problem, it will also contain some recommendations on how to fix the code.

As before, if latency is an issue, you will have to be mindful about your choice of endpoint and model. Make a performance benchmark of your different options to help guide your choice.

Vector databases

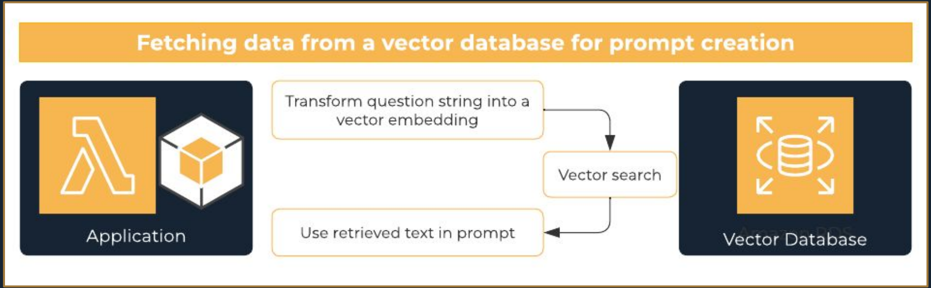
When you have a large body of knowledge that consists mostly of unstructured documents (*not databases you can directly query - for example, documents*), the main problem is: "*In question XYZ, what pieces of texts in my sea of curated documents are most related and appropriate?*"

We can't do RAG, after all, if our application can't figure out what to retrieve in the first place.

Traditional enterprise search can be limited when finding the right documents and excerpts rely on **semantic distance** or how closely-related a particular word is to another. In other words when you need to rely more on *meaning* and *context* rather than just keywords.

These cases are where we need to rely on technology that is similar to how LLMs think: not in terms of words, but in vector embeddings. **Vectors** are a mathematical representation of words where related terms are closer to each other in the vector space. Imagine that each vector is a multidimensional array containing numbers, and you have hundreds of dimensions in that array. If you think of those numbers as coordinates, then terms that are related to each other have values that put them closer to each other than unrelated terms.

With information transformed to that kind of embedding, we can then take a user question, transform it into vectors using an embedding model, and then find related information in our database that stores information also as vectors using the same embedding model. We then use the **original** text of the related information we found, and it to our prompt.



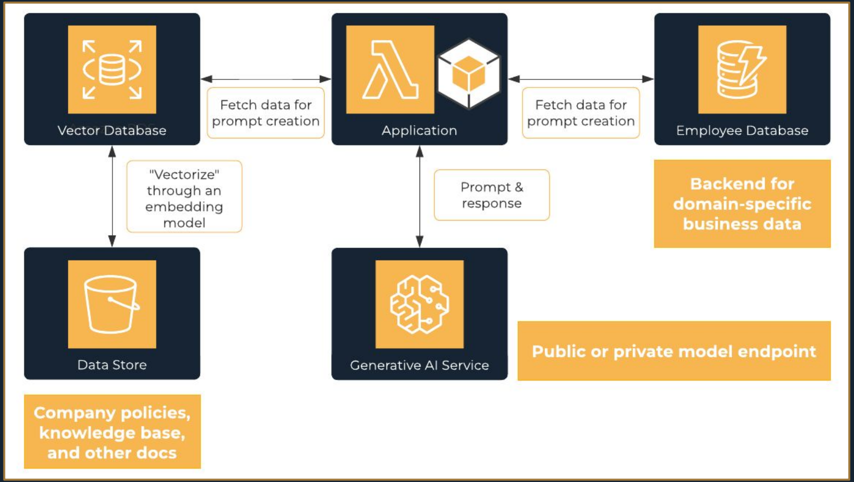
Such a database is called a **vector database**. Stand-alone vector databases include Pinecone and Weaviate.

Vector support has also been added to traditionally non-vector-database products. For example, **PostgreSQL** has the pgvector extension to work with vector search, and **Amazon OpenSearch** has the vector search collection type.

If you already use a traditional database with vector support (Oracle, Postgres, and Microsoft SQL Server all do), or already use a service like OpenSearch which also provides support for vector search, that's the best way to get started. It's almost always a bigger lift to start with a new, stand-alone product, rather than just consuming a new feature of a service you already use.

Stand-alone vector databases might be a good idea if you find that the performance and scalability you desire isn't available in your current setup.

A generic architecture diagram for RAG using a vector database would look something like this:



That diagram shows a vector-search-powered RAG using a vector database for our previous use case of a chatbot meant to help our employees get answers to their company policy questions. It looks almost exactly like the enterprise search architecture diagram, just with the enterprise search tool replaced by a vector database.

A critical difference is the part that is vector-database specific: transforming our data into vectors. Our vector database will store both the plaintext and vectorized forms of our data, so these vectors must be created using our chosen embedding model. The embedding model determines how a particular word is transformed into a vector, including the number of dimensions.

Maintaining a vector database is an entirely new discipline in itself, especially for fast-changing data. You should create an automated pipeline in your environment (Cloud or otherwise) to manage the freshness of the records in your vector database. RAG will end up giving terrible results if the actual knowledge base has been updated significantly, but your vector database still has the old version because someone in the team forgot to update it.

RAG examples and demos

Weaviate provides a couple good examples of RAG through vector search in action:

➔ https://github.com/weaviate-tutorials/DEMO-text-search-plants
In this demo app, users can find plants using natural language queries, and pull up detailed information.

➔ https://github.com/weaviate/Verba
Verba is a tool you can use locally or in the Cloud to use your personal documents as a source of truth for your questions, to "*chat with your documents.*" Verba is the kind of implementation you'd want for that self-service employee portal where they can ask random questions about company policies and have it answered by an LLM-powered chatbot that "understands" your documents.

Pinecone has an interactive Python notebook that demonstrates RAG: https://colab.research.google.com/github/pinecone-io/examples/blob/master/docs/langchain-retrieval-augmentation.ipynb

INTERESTING READS AND RESOURCES

Everything About Vector Databases... and Top Vector Databases for LLMs
A short and simple article about vector databases, with a collection of five of their chosen vector databases. You can use this list as a jumping off point for more vector database exploration.

Vector Search with OpenAI Embeddings: Lucene Is All You Need
This paper presents an argument against immediately assuming you "need" a vector database for your LLM deployments.

RA-DIT: Retrieval-Augmented Dual Instruction Tuning
Philipp Schmid of Hugging Face shares his insights on a new paper from Meta detailing a novel approach to reduce LLM hallucinations in RAG scenarios. Original paper is also linked.

RAG is Just Fancier Prompt Engineering
An interesting take on RAG. Also links to other interesting papers about RAG, including comparing a RAG setup vs using a long context window model provided with the full context instead of retrieval.

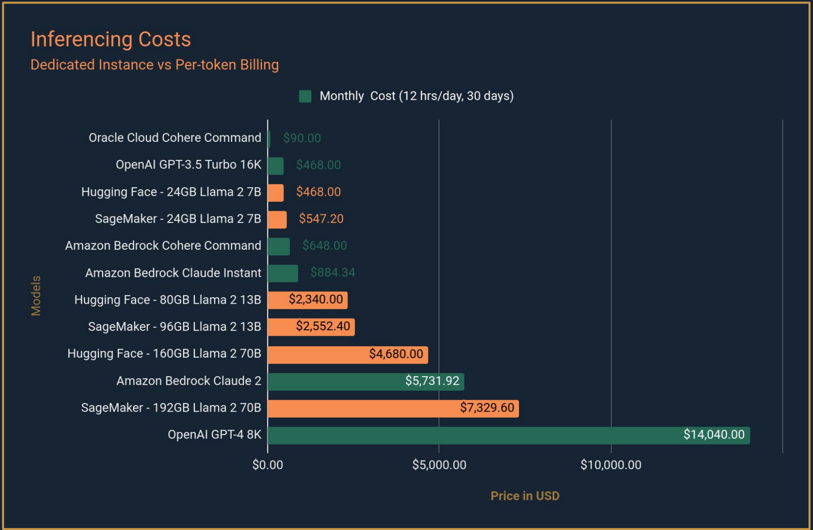
CHAPTER 7: SPECIALIZED SMALLER PRIVATE MODELS

Deploying any single LLM is easy. Making a proof of concept (POC) for a single use case on it is also easy.

When it's just your team creating, testing, and demonstrating use cases on the POC, the economics of the LLM barely matter. Your team probably won't even break a few million tokens of usage. Even 20M tokens in a month is going to be barely a blip in your operations bill for almost any model you could be using.

The problem is **scale**.

At scale, when your POC now needs to handle huge amounts of volume, the economics begin to matter.



Revisiting the cost of LLMs

Recall our first foray into LLM costing in Chapter 5. (*See the graph on the left*)

That's quite a huge spread there. Recall that we were modeling this scenario:

- ❑ **Transactions/hr**: 1,000
- ❑ **Avg Input Tokens**: 1,000
- ❑ **Avg Output Tokens**: 150
- ❑ **Hrs/day**: 12
- ❑ **Days**:30

The orange bars show the cost of hosting an LLM yourself (*instance-hr pricing*), versus the cost of paying per token, shown in green bars (*Bedrock, OpenAI*).

If we make the operating hours longer (*from 12 to 24 hrs per day*), hosting becomes relatively more expensive compared to per-token billing. Conversely, if we increase transactions or token amounts, hosting generally becomes relatively less expensive compared to per-token billing. When per-token costs start becoming 2-3x more expensive than the cost of hosting a private model, then suddenly it becomes far more attractive to maintain your own endpoints.

These hosting costs are actually a worst case. We can host Llama 2 13B on a 24GB machine, or host Llama 2 70B on an 80GB or 96GB machine. In fact, we really should. But before they can run smaller machines than we initially estimated them for, the models need to be slimmed down and made smaller.

This process of making models smaller is called *quantization*.

Quantization

Quantization is a mathematical process of mapping a bigger set of values to a smaller set of values.

In LLMs, this is helpful because this allows us to represent what would be **32-bit** or **16-bit** floating point values (*the weights of the LLM parameters*) into just an 8-bit format or even smaller.

Quantization has two major benefits - it's **cheaper** to run them (*we only need smaller instances with less GPU memory*), and they inference **faster** (*which improves the user experience by reducing latency*).

Let's take **Llama 2 13B**. It has 13 billion parameters. Those parameters are mostly 16-bit floating point values (f16). Space-wise, that means 13 billion x 2 bytes (*16 bits / 8 bits per byte*) = 26 billion bytes, or **26GB**. If you downloaded Llama 2 13B, you'll see that the model does take up 26GB of disk space - loading that into memory therefore also needs 26GB of RAM.

If we use quantization to make this smaller, say **q4** (4-bit quantization), then we'd be slicing our storage (and RAM) needs by 4 (from 16-bit values to 4-bit values), so we'd lower storage and memory needs significantly:

*13 billion x 0.5 bytes = **~6.5GB***

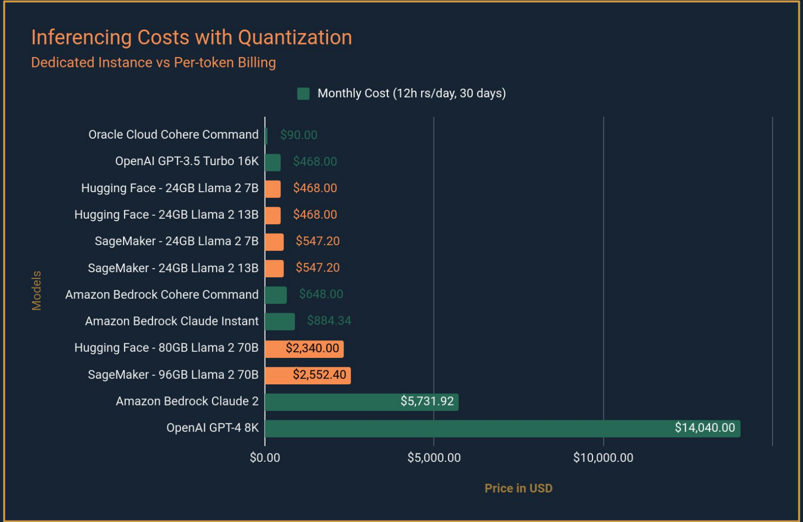
If you did this yourself in Chapter 5 during the llama.cpp section, you'd have found we're off a little: ~**7.9GB** is the actual size of Llama 2 13B with the q4_K quantization. That's still **3.3x smaller** than the original!

We could easily fit 3 of this **Llama 13B q4_K** model into the smallest Hugging Face or SageMaker instance. Not necessarily that we'd want to - that's just to show how much capacity there is here, and the amount of volume you can expect such a server can now handle, given the slimmed down 13B model.

Best of all, we've just reduced our 13B hosting cost from ~$2,500 to just ~$500!

If we apply the same treatment for Llama 2 70B, we go from ~140GB of space, to just under 45GB! From the table above, that means our hosting costs dropped from $5.7-7.5K USD, to just ~$2.5K USD (cutting costs by more than half!)

With quantization, we can revise the cost graph from earlier:



Suddenly, both **Llama 2 7B** and **13B** make much more sense economically, and we've lowered the threshold where self-hosting the **70B** starts to make sense.

With quantization, private model hosting suddenly starts to make a lot more sense for any deployment at scale.

This is one of the most important goals in the generative AI journey as recommended in the **Generative AI Roadmap** we prescribed in Chapter 1: **specialized smaller private models.**

With quantization, private models can be hosted a lot cheaper than their full-precision vanilla versions. Further on in this Chapter, we'll take a look at **fine-tuning** - how to make these foundation models more suited for our specific use cases as needed. Together, fine-tuning and quantization make generative AI enterprise deployments more scalable, reliable and cost-effective.

Quantization impact on LLM performance

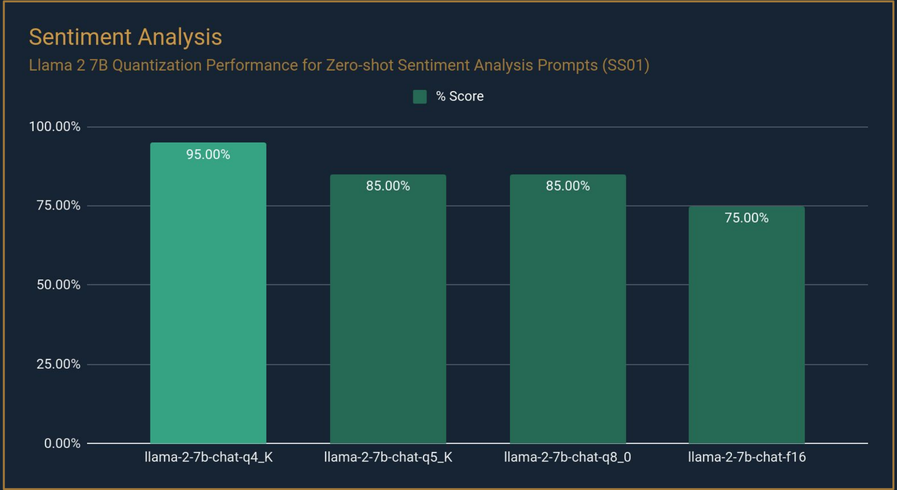
We're able to make these foundation models much smaller and cheaper to operate - but what about the actual performance? If we reduce the precision of the weights, wouldn't that also affect the accuracy and reliability of the LLM?

Great question!

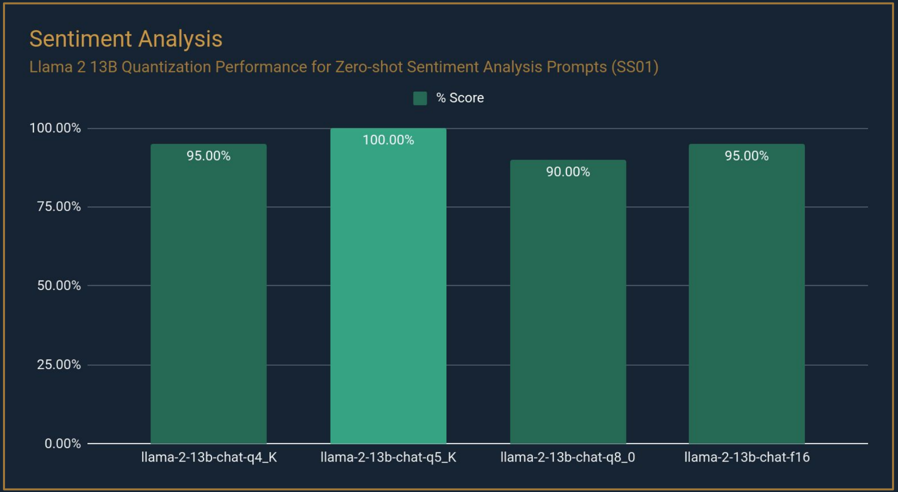
In Chapter 5, we talked about LLM_LocalBench, and how we benchmarked the different variants of Llama 2 (7B, 13B and 70B) in a few use cases.

Aside from their native **f16** (non-quantized) version, that experiment also included a few different quantizations for each model to compare how different levels of quantization perform against the non-quantized vanilla version in these different use cases.

Let's look at the sentiment analysis results.



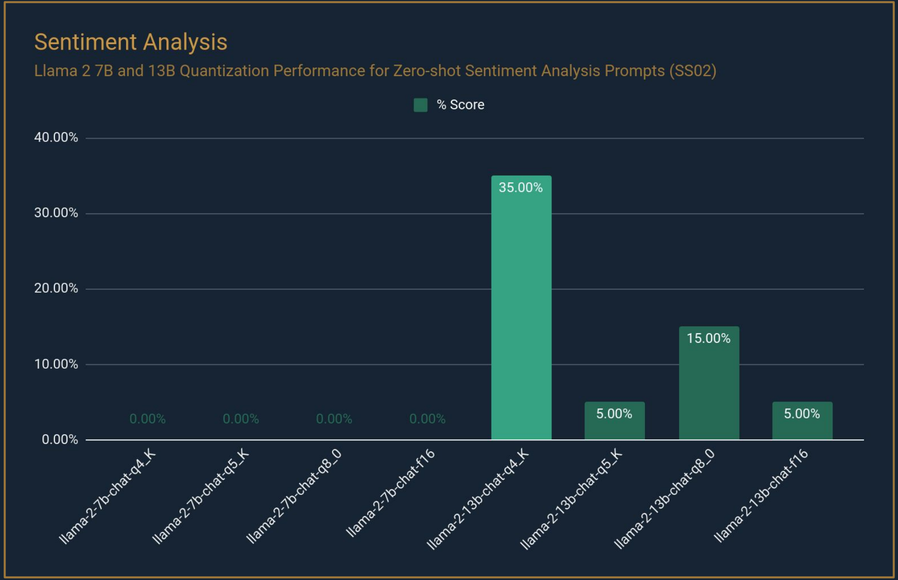
In the first sentiment analysis prompt (*you can review all the prompts here: DataV2/Sentiment_Analysis*), we **don't see a performance degradation** with q8, q5, or q4 quantizations in the 7B (smallest) model. In fact, f16 showed the worst performance. Don't read that as quantization being better - that just means from 20-40 samples, the impact of quantization is less than the impact of the inherent randomness from LLMs.



Now with 13B, we can see that there's also **no clear difference in performance** in this prompt. And the f16 version **still isn't the top performer** here.

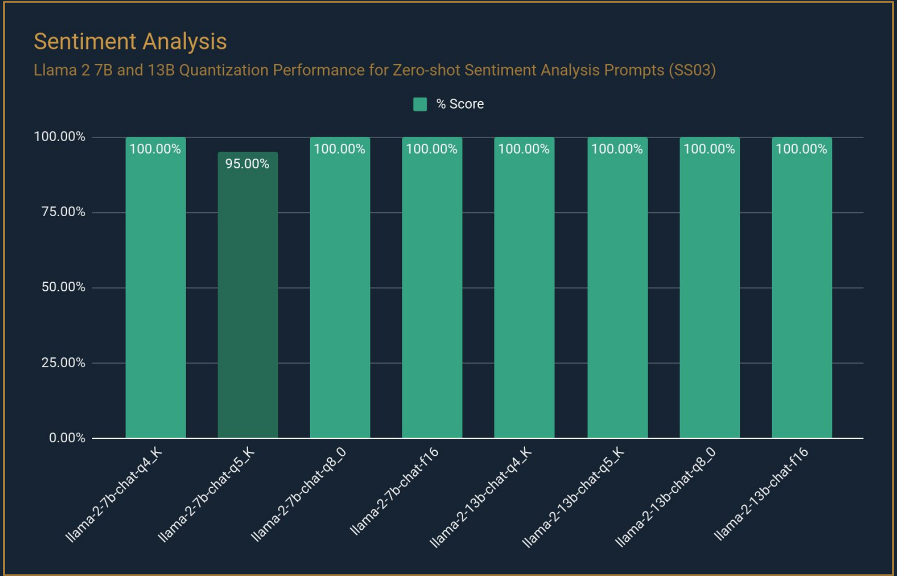
If this seems weird, that's just LLM randomness for you. **You don't have to take our word for it though.** Use LLM_LocalBench with the DataV2 dataset and try it out for yourself. Every single piece used in this experiment is available at no cost: our repositories for the infra and data, the Llama 2 models, and llama.cpp for local inference. You'll see a lot of natural variance as you keep running and re-running more rounds.

This was an easy prompt for the LLM, so let's look at a harder prompt:



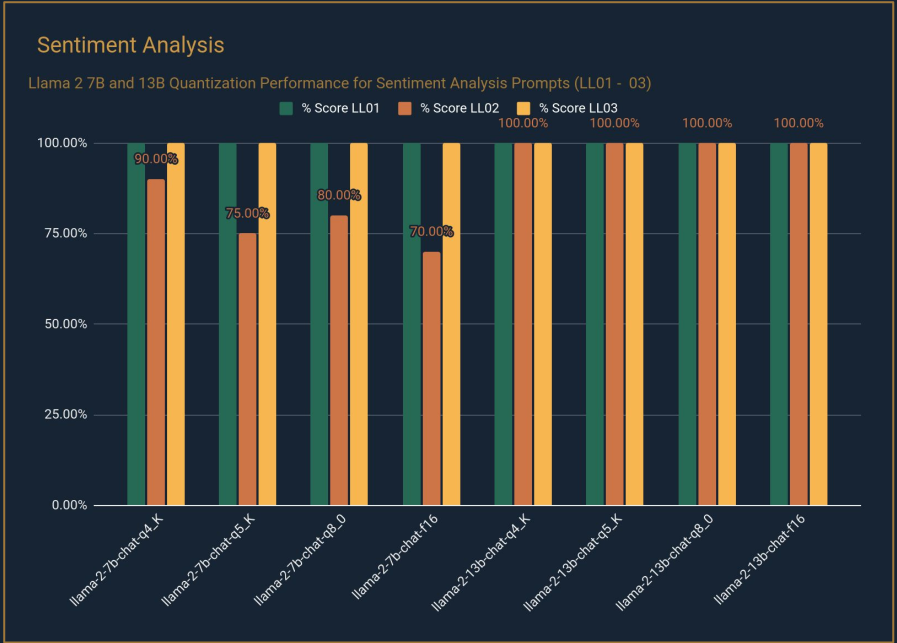
Again, there's not really a clear quantization performance hit. For **7B**, that's because even the full, non-quantized version didn't really get anything right. For **13B**, it even looks like the smallest, most aggressive quantization (q4_K) outperformed the full version. Again, like our first graph, don't read that to mean quantization increases performance (*unlikely*). There's a ton of variance in LLM responses, and this variance so far is stronger than any performance hit from quantization - so, as far as our quantization is concerned, there's still no real performance hit that we can measure.

The last basic prompt, SS03, is another easy one:



Again, we can see that quantization still has no measurable performance hit for our use case.

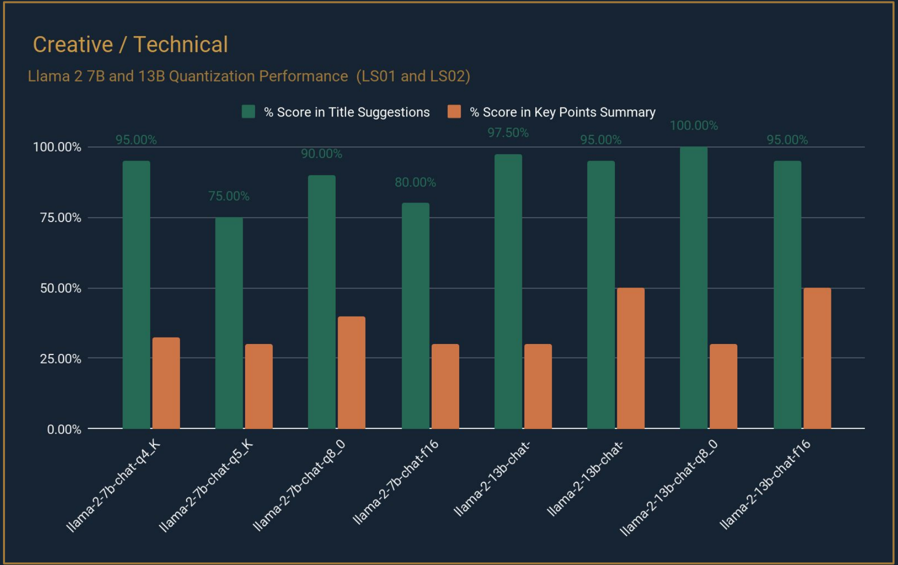
So far we've looked at performance in the most basic prompt sample. Now let's look at the best prompt-engineered version of our sentiment analysis use case (*few-shot learning + detailed response*):



Across all 3 of our best prompt-engineered samples (LL01 to LL03), quantization doesn't seem to be harmful at all.

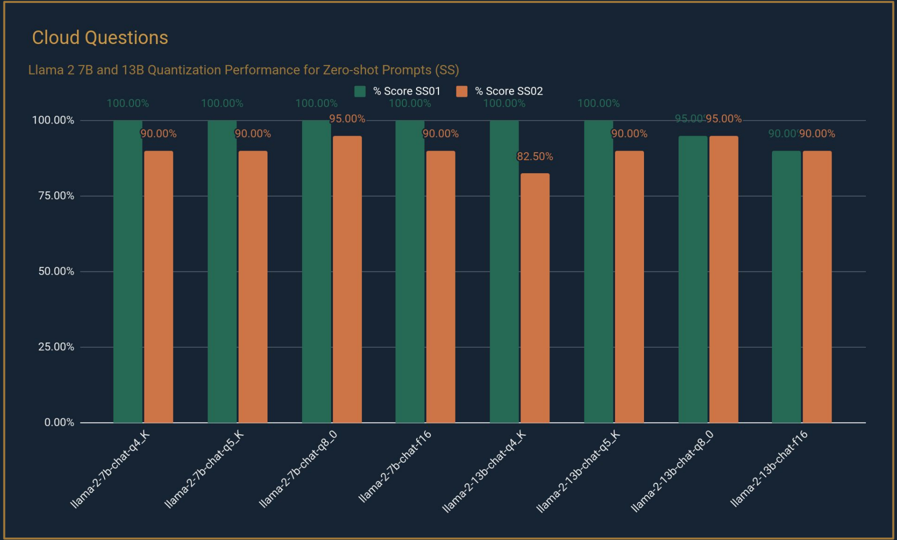
At least for our sentiment analysis use case, quantization doesn't seem to hurt! We can quantize our models aggressively, down to q4, and expect minimal-to-zero performance hit - at least according to our test samples.

We also benchmarked other use cases, but still found no significant difference between the full and quantized versions of 7B and 13B:



Across these different use cases and prompts (from above and on the right), we still saw no performance impact from quantizations. This is awesome - we get to host these private models a lot easier! And remember, Cloud isn't your only choice. Quantization and the lack of a significant performance hit in our use cases means it'd be a lot easier to run private models locally on dev machines too.

Important! Don't take our testing as a blank check for your own use cases. It bears repeating: **Always benchmark on your own use cases and data!**



It is completely possible (*and highly likely*) that for a different use case or a different set of data, the models and quantizations will perform differently.

Two critical takeaways:

➔ Quantization is an **essential** part enterprise-level generative AI deployment for scalable private models

➔ You must perform **realistic benchmarks** against your LLMs for each of your expected use cases, and on the **quantized** models you wish to deploy.

Fine-tuning

We've already gone through a lot of useful tools - *technologies and processes* - to help make our generative AI projects successful:

- **Prompt engineering**, to get more effective results from our LLMs
- **Public models**, usually the quickest way to get started and prototype with generative AI
- **Private models**, to allow us to harness generative AI without sacrificing strict regulatory compliance and privacy controls
- **Foundation models**, to enable wide experimentation, matching appropriate models to specific use cases, and benefitting from the rapid pace of Open Source LLM development.
- **Retrieval Augmented Generation (RAG)**, to make our LLMs more useful in a business context through feeding it specific or curated data as needed in different use cases
- **Quantization**, to reduce LLM size and memory needs, drastically reducing LLM self-hosting costs and making local LLM deployments far easier, providing a huge boost for private model hosting.

There's one more tool for our LLM strategy toolbox we need to cover: *fine-tuning*.

In **Chapters 4 and 5**, we learned that foundation models are pre-trained on a huge amount of diverse data, and this makes them adaptable for a lot of tasks that they haven't been specifically trained for. You've seen this in action our foundation model benchmarks data, vanilla and quantized - they can be made to perform acceptably well in different use cases.

Fine-tuning is the process of making foundation models perform better in specific tasks. If we look at the results data from our LLM_LocalBench experiments, we can see that even in the correctly answered items, our foundation models aren't very compliant with any specific formatting.Ffor example, when we ask for the sentiment analysis output to be in a particular format, the responses may be correct (*the sentiment was identified correctly*) but the formatting is not uniform, especially for the zero-shot prompts. It's significantly better for the few-shot ones, but still not as consistent as it could be.

This is where fine-tuning can come in. With fine-tuning, we provide **additional nice, clean, sample data** that is representative of the specific task we want to tune the LLM for. In the case of our sentiment analysis use case, that could be a data set containing *real product reviews from our customers*. We label these samples correctly (*i.e., also provide the ideal sentiment analysis output for these, according to our human experts*), and our fine-tuning training set then becomes a few dozen review samples coupled with their ideal answers.

Through the fine-tuning process, this new training data set is "added" to our LLM's knowledge, so that the LLM understands better how to handle this specific task.

A good case for this is to strictly control the response format of the LLM so that its responses can then be easily consumed by connected enterprise systems. In our sentiment analysis example, if we could rely that the sentiment analysis responses are strictly formatted as a Python dictionary like this:

```
{
    "Product Name": "<product name>",
    "Review Sentiment": "Positive|Negative|Neutral",
    "Positive Comments": [
        "<comment1>",
        "<comment2>",
    ],
    "Negative Comments": [
        "<comment3>".
    ]
}
```

...then we can immediately take that response and have it consumed by our enterprise application since it's already properly formatted. We don't need to have complex and failure-prone code to take the sometimes-unstructured response, parse it, and then transform it to our usable structured format.

Important! *It probably won't work very well if you are trying to fine-tune it with additional facts - for example, company policies or your project knowledgebase. With only very few samples and a limited amount of tokens in the fine-tuning training set, that'd be a drop in the bucket compared to the hundreds of billions or couple trillion tokens the foundation model was already trained with. And if it's not good for facts, then it also won't be a good way to mitigate hallucinations. For mitigating hallucinations and making LLMs stick to prescribed facts like your company policies or a support spiel, prompt engineering and RAG is the way to go.*

All platforms we've encountered so far offer fine-tuning for some of their models. Let's go through the fine-tuning process for two of them: **OpenAI** and **SageMaker JumpStart**.

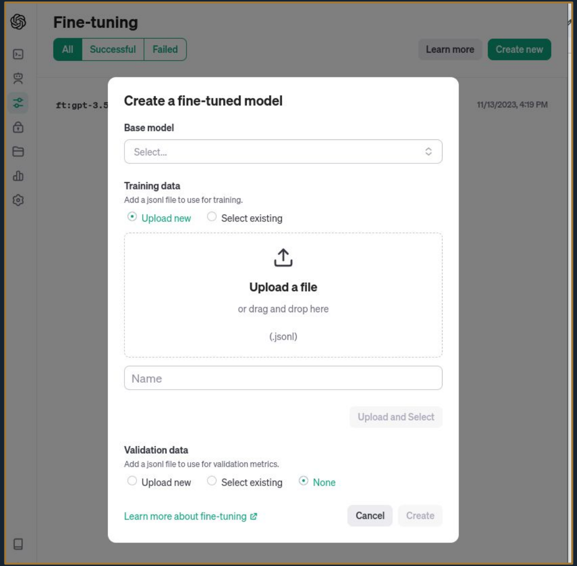
Fine-tuning on OpenAI

Now, we'll go through the steps we did to fine-tune GPT-3.5 for our sentiment analysis needs. (*For more in-depth information, consult the official OpenAI fine-tuning documentation.*)

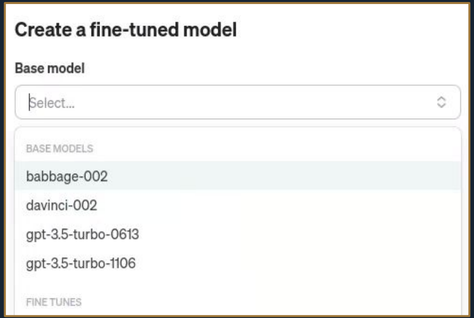
1. **Prepare and validate training data**
    a. Data Collection
        i. **Collect** text data that is representative of the tasks you want the model to perform. This could be conversational exchanges, article excerpts, or any other relevant text.
        ii. **Format** your data as JSONL (JSON Lines format). Each line in a JSONL file represents a separate JSON object.
        iii. For each object, include the following fields: **system message**, **input** (prompt) and **output** (response). You can find examples based on our sentiment analysis use case in the BRIGADE Companion Repo.
        iv. **Data Cleaning and Preprocessing** - Ensure your data is clean, which means it should be free of irrelevant content, formatting issues, and sensitive information.
    b. Validating Your Dataset - You can use this tool from OpenAI to validate your dataset before fine-tuning. It provides a comprehensive approach to preparing and analyzing your data for fine-tuning a chat model with OpenAI, ensuring the data is in the correct format, identifying potential issues, and estimating the associated costs. Here's a sample output when you run that validation tool on your training dataset:

```
[debbie@fedora-dpc Fine-tuning]$ python3 data_validation.py
Num examples: 10
First example:
{'role': 'system', 'content': 'Your task is to identify the sentiment of a product review. Here is an
example of a product review from a customer, along with the corresponding sentiment analysis of the re
view:\n\nExample #1:\nHere is a product review from a customer, which is delimited with triple backtic
ks.\nProduct Name: Zen End Table\nReview text: \n```\nI have mixed feelings about it. On the positive
side, the design is sleek and modern, which complements my living room aesthetic. However, the quality
 is not up to par. The table arrived with a few scratches and the wood feels a bit flimsy. For the pri
ce I paid, I expected better craftsmanship. Overall, I would give it 3 stars. \n```\n\n    What is the
 sentiment of that product review?\n    Identify the product being reviewed.\n    Enumerate the positi
ve and negative aspects of the product review.\n    The response should be in JSON format  with the fo
llowing elements:\n            - Product name\n            - Review Sentiment (Positive/Negative/Neutr
al)\n            - Positive comments about the product (Enumerate)\n            - Negative comments ab
out the product (Enumerate)\n\nAnswer:\n{\n    "Product name": "Zen End Table",\n    "Review Sentiment
": "Neutral",\n    "Positive comments about the product": [\n        "Sleek and modern design",\n
    "Complements the living room aesthetic"\n    ],\n    "Negative comments about the product": [\n
    "Quality not up to par",\n        "Arrived with scratches",\n        "Wood feels flimsy",\n
 "Craftsmanship not as expected for the price"\n    ]\n}'}
{'role': 'user', 'content': 'Here is a product review from a customer, which is delimited with triple
backticks.\n\n    Product Name: Sahara Sands Canopy Bed\n    Review text: \n    ```\n    I absolutely
love my Sahara Sands Canopy Bed! It adds a touch of elegance to my bedroom and the quality is exceptio
nal. The assembly was a bit time-consuming, but the end result is worth it. The design is beautiful an
d the bed is very sturdy. Definitely worth the investment.\n    ```\n\n    What is the sentiment of th
at product review?\n    Identify the product being reviewed.\n    Enumerate the positive and negative
aspects of the product review.\n    The response should be in JSON format  with the following elements
:\n            - Product name\n            - Review Sentiment (Positive/Negative/Neutral)\n
 - Positive comments about the product (Enumerate)\n            - Negative comments about the product
(Enumerate)\n\nAnswer:'}
{'role': 'assistant', 'content': '{\n    "Product name": "Sahara Sands Canopy Bed",\n    "Review Senti
ment": "Positive",\n    "Positive comments about the product": [\n        "Adds a touch of elegance to
 the bedroom",\n        "Exceptional quality",\n        "Beautiful design",\n        "Very sturdy",\n
       "Worth the investment"\n    ],\n    "Negative comments about the product": [\n        "Assembly
 was time-consuming"\n    ]\n}'}
No errors found
Num examples missing system message: 0
Num examples missing user message: 0

#### Distribution of num_messages_per_example:
min / max: 3, 3
mean / median: 3.0, 3.0
p5 / p95: 3.0, 3.0

#### Distribution of num_total_tokens_per_example:
min / max: 560, 1036
mean / median: 677.4, 607.0
p5 / p95: 565.4, 851.4999999999999

#### Distribution of num_assistant_tokens_per_example:
min / max: 72, 200
mean / median: 110.3, 85.5
p5 / p95: 72.9, 164.0

0 examples may be over the 4096 token limit, they will be truncated during fine-tuning
Dataset has ~6774 tokens that will be charged for during training
By default, you'll train for 10 epochs on this dataset
By default, you'll be charged for ~67740 tokens
```

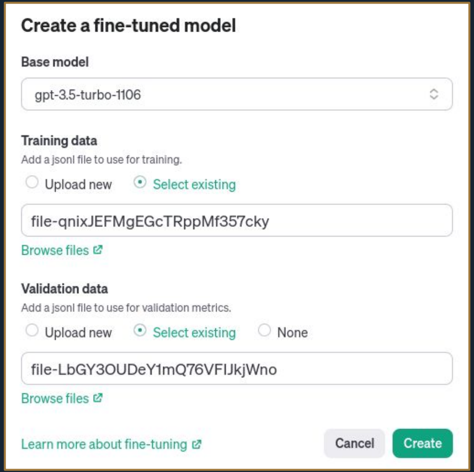
2. **Train a New Fine-tuned Model** - Now that you have prepared your data and ensured that it follows the required format, the actual training process is actually the easy part. Fine-tuning a model using OpenAI's platform can be done either through their Fine-Tuning UI or via the API. Here's a walkthrough of the fine-tuning process for both methods:
   a. Fine-tuning via the OpenAI fine-tuning UI
      i. Go to https://platform.openai.com/finetune and click **Create New**



ii. Choose your base model.



iii. Upload training and validation data or choose from your existing files. Then just click **Create** and you're good to go!



vi. **Monitoring and Adjustments**: You can easily keep track of the training progress using the UI. It only took around 10 mins for our sentiment analysis fine-tuning job with 10 examples.

b. Fine-tuning via the OpenAI API - The API allows for more detailed control over the fine-tuning process, including the ability to adjust a wider range of parameters and settings. (*see: OpenAI API Reference*)

i. Upload the files using the Files API. Maximum individual file size is 512 MB and should be in **.jsonl** format.

```
from openai import OpenAI
from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv()) # read local .env file

client = OpenAI()

#upload training file
file = client.files.create(
 file=open("./FT_data/senti_training_dataset.jsonl" , "rb"),
 purpose="fine-tune"
)

#upload validation file
validation_file = client.files.create(
 file=open("./FT_data/senti_validation_dataset.jsonl" ,
"rb"),
 purpose="fine-tune"
)

print(file)
print(validation_file)
```

The API returns the uploaded File object. This response includes information about the file, such as its ID, which you'll need for fine-tuning.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   ...        bash - FineTuning
[debbie@fedora-dpc FineTuning]$ python3 finetune_job.py
FileObject(id='file-qt4TEUy0D2ZTJyJbdstvLpdL', bytes=33565, created_at=1700113302, filename='se
nti_training_dataset.jsonl', object='file', purpose='fine-tune', status='processed', status_det
ails=None)
FileObject(id='file-24SsZj70W2aQSktO9sMNiRlf', bytes=9036, created_at=1700113303, filename='sen
ti_validation_dataset.jsonl', object='file', purpose='fine-tune', status='processed', status_de
```

ii. Create a fine-tune request. Send a request to OpenAI to create a new fine-tuned model using your data. You are required to specify the base model you're fine-tuning and to include your training file ID. The validation file is optional, but we included one in our example. The use of validation data helps in monitoring the model's performance and avoiding overfitting. All the other parameters are kept at default values.

```
#start fine-tune job
response = client.fine_tuning.jobs.create(
 training_file=file.id,
 validation_file=validation_file.id,
 model="gpt-3.5-turbo-1106"
)
```

The API returns a **fine-tuning.job** object, which includes details of the enqueued job, such as job status and the name of the fine-tuned models once complete.

```
FineTuningJob(id='ftjob-bKeWAk8r9qESRZMzG22DQJNK', created_at=1700113305, error=None, fine_tune
d_model=None, finished_at=None, hyperparameters=Hyperparameters(n_epochs='auto', batch_size='au
to', learning_rate_multiplier='auto'), model='gpt-3.5-turbo-1106', object='fine_tuning.job', or
ganization_id='org-k2hDqIeYqqiZJLLHg3tX2Ajs', result_files=[], status='validating_files', train
ed_tokens=None, training_file='file-qt4TEUy0D2ZTJyJbdstvLpdL', validation_file='file-24SsZj70W2
aQSktO9sMNiRlf')
```

iii. Monitor the training process. Use the code below to check for the status of your fine-tuning job. The **job-id** will be returned by the Create fine-tuning job API we used in the previous step. You can also monitor the process through the Fine-tuning UI.

```
# Retrieve the state of a fine-tune job
print("FT_job status: \n")
print(client.fine_tuning.jobs.retrieve("ftjob-id"))
```

Fine Tuning status = queued:

```
[debbie@fedora-dpc FineTuning]$ python3 finetune_job.py
FT_job status:

FineTuningJob(id='ftjob-bKeWAk8r9qESRZMzG22DQJNK', created_at=1700113305, error=None, fine_tune
d_model=None, finished_at=None, hyperparameters=Hyperparameters(n_epochs=10, batch_size=1, lear
ning_rate_multiplier=2), model='gpt-3.5-turbo-1106', object='fine_tuning.job', organization_id=
'org-k2hDqIeYqqiZJLLHg3tX2Ajs', result_files=[], status='queued', trained_tokens=None, training
_file='file-qt4TEUy0D2ZTJyJbdstvLpdL', validation_file='file-24SsZj70W2aQSktO9sMNiRlf')
```

Fine Tuning status = succeeded:

```
[debbie@fedora-dpc FineTuning]$ python3 finetune_job.py
FT_job status:

FineTuningJob(id='ftjob-bKeWAk8r9qESRZMzG22DQJNK', created_at=1700113305, error=None, fine_tune
d_model='ft:gpt-3.5-turbo-1106:personal::8LPZfKpe', finished_at=1700113885, hyperparameters=Hyp
erparameters(n_epochs=10, batch_size=1, learning_rate_multiplier=2), model='gpt-3.5-turbo-1106'
, object='fine_tuning.job', organization_id='org-k2hDqIeYqqiZJLLHg3tX2Ajs', result_files=['file
-eL2zq7ORSfEwOQ09lTqOrYky'], status='succeeded', trained_tokens=67540, training_file='file-qt4T
EUy0D2ZTJyJbdstvLpdL', validation_file='file-24SsZj70W2aQSktO9sMNiRlf')
```

And just like that, you now have your **own fine-tuned GPT-3.5 model!**

This isn't the end, though.

After testing and benchmarking your fine-tuned model, you may need to rerun the fine-tuning and adjust parameters to improve the model's learning and resulting performance.

Fine-tuning is an **iterative process**.

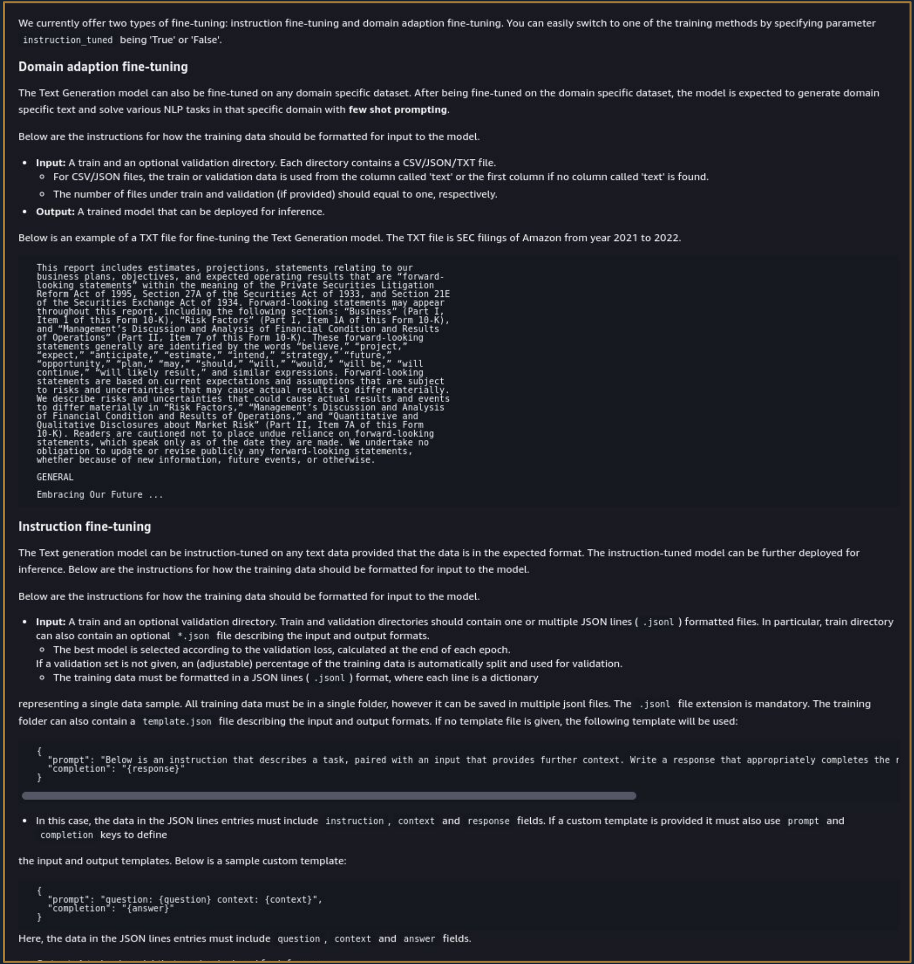
Fine-tuning through SageMaker JumpStart

Many models available in SageMaker JumpStart allow you to further fine-tune them. You can find these models labeled with "*Fine-tunable: Yes*" in their model card.

Llama 2 is one of these available fine-tunable models. Let's go through the process of fine-tuning the small Llama 2 7B Chat model so that it becomes more appropriate for our sentiment analysis use case.

1. Prepare data (JSON Lines)
   a. In SageMaker JumpStart, you can find the possible data formats for training under the **Fine-tune the Model on a New Dataset** section of the model card (*that's usually at the very bottom; see screenshot on the right*). In the case of Llama 2 7B Chat, we have three options: chat fine-tuning, domain adaptation fine-tuning, and instruction fine-tuning.
   b. Our use case is best-served with **instruction fine-tuning**, so that we can teach it to follow a specific format to handle the sentiment analysis prompts it will encounter. We'll prepare data for that type of fine-tuning.
   c. Under the **Instruction fine-tuning** subsection of the **Fine-tune** section, it explains the different ways you can format your data. In our case, we prepared the simplest format using the most straightforward template:

```
{
    "prompt": "{question}",
    "completion": "{answer}"
}
```



We currently offer two types of fine-tuning: instruction fine-tuning and domain adaption fine-tuning. You can easily switch to one of the training methods by specifying parameter `instruction_tuned` being 'True' or 'False'.

Domain adaption fine-tuning

The Text Generation model can also be fine-tuned on any domain specific dataset. After being fine-tuned on the domain specific dataset, the model is expected to generate domain specific text and solve various NLP tasks in that specific domain with **few shot prompting**.

Below are the instructions for how the training data should be formatted for input to the model.

- **Input:** A train and an optional validation directory. Each directory contains a CSV/JSON/TXT file.
  - For CSV/JSON files, the train or validation data is used from the column called 'text' or the first column if no column called 'text' is found.
  - The number of files under train and validation (if provided) should equal to one, respectively.
- **Output:** A trained model that can be deployed for inference.

Below is an example of a TXT file for fine-tuning the Text Generation model. The TXT file is SEC filings of Amazon from year 2021 to 2022.

```
This report includes estimates, projections, statements relating to our
business plans, objectives, and expected operating results that are "forward-
looking statements" within the meaning of the Private Securities Litigation
Reform Act of 1995, Section 27A of the Securities Act of 1933, and Section 21E
of the Securities Exchange Act of 1934. Forward-looking statements may appear
throughout this report, including the following sections: "Business" (Part I,
Item 1 of this Form 10-K), "Risk Factors" (Part I, Item 1A of this Form 10-K),
and "Management's Discussion and Analysis of Financial Condition and Results
of Operations" (Part II, Item 7 of this Form 10-K). These forward-looking
statements generally are identified by the words "believe," "project,"
"expect," "anticipate," "estimate," "intend," "strategy," "future,"
"opportunity," "plan," "may," "should," "will," "would," "will be," "will
continue," "will likely result," and similar expressions. Forward-looking
statements are based on current expectations and assumptions that are subject
to risks and uncertainties that may cause actual results to differ materially.
We describe risks and uncertainties that could cause actual results and events
to differ materially in "Risk Factors," "Management's Discussion and Analysis
of Financial Condition and Results of Operations," and "Quantitative and
Qualitative Disclosures about Market Risk" (Part II, Item 7A of this Form
10-K). Readers are cautioned not to place undue reliance on forward-looking
statements, which speak only as of the date they are made. We undertake no
obligation to update or revise publicly any forward-looking statements,
whether because of new information, future events, or otherwise.

GENERAL

Embracing Our Future ...
```

Instruction fine-tuning

The Text generation model can be instruction-tuned on any text data provided that the data is in the expected format. The instruction-tuned model can be further deployed for inference. Below are the instructions for how the training data should be formatted for input to the model.

Below are the instructions for how the training data should be formatted for input to the model.

- **Input:** A train and an optional validation directory. Train and validation directories should contain one or multiple JSON lines (`.jsonl`) formatted files. In particular, train directory can also contain an optional `*.json` file describing the input and output formats.
  - The best model is selected according to the validation loss, calculated at the end of each epoch.
  If a validation set is not given, an (adjustable) percentage of the training data is automatically split and used for validation.
  - The training data must be formatted in a JSON lines (`.jsonl`) format, where each line is a dictionary

representing a single data sample. All training data must be in a single folder, however it can be saved in multiple jsonl files. The `.jsonl` file extension is mandatory. The training folder can also contain a `template.json` file describing the input and output formats. If no template file is given, the following template will be used:

```
{
  "prompt": "Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the r
  "completion": "{response}"
}
```

- In this case, the data in the JSON lines entries must include `instruction` , `context` and `response` fields. If a custom template is provided it must also use `prompt` and `completion` keys to define

the input and output templates. Below is a sample custom template:

```
{
  "prompt": "question: {question} context: {context}",
  "completion": "{answer}"
}
```

Here, the data in the JSON lines entries must include `question` , `context` and `answer` fields.

d. You can find the sample training dataset we used here: BRIGADE/Fine-tuning/SageMaker

e. What you'll see there is that we prepared a minimal training dataset of 13 question and answer pairs, in JSON Lines format. There is also a template.json file there describes our training data format.

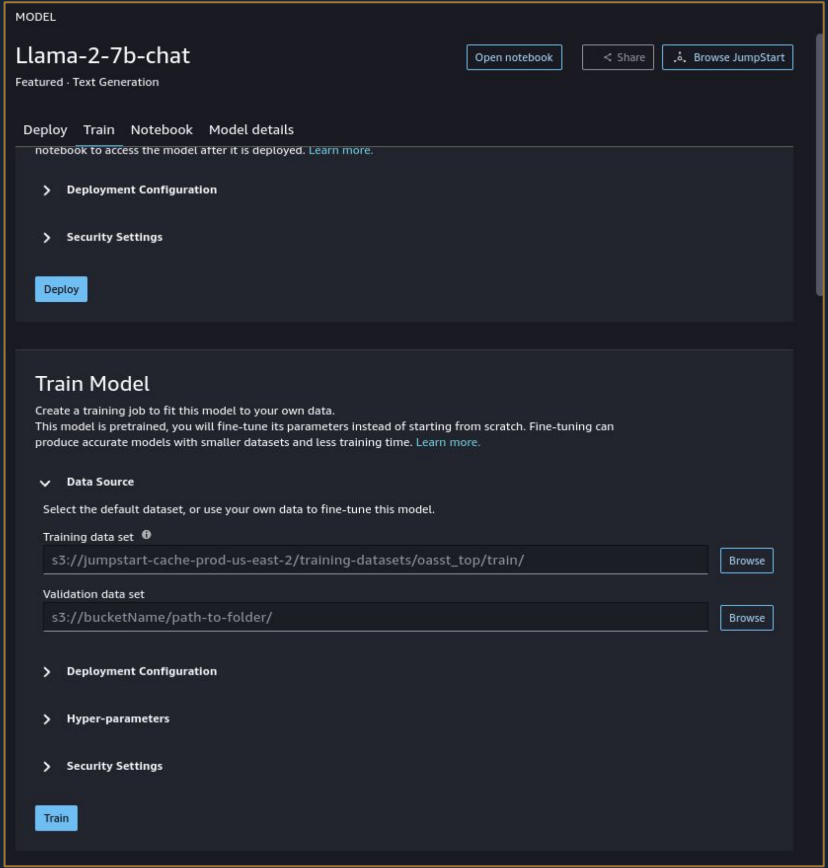
f. **Make sure there isn't a newline or empty line** at the end of your training file. That will cause an error during the training job and cause it to fail.

g. **We also didn't prepare a separate validation dataset**. There is a known-bug for SageMaker training with Llama 2 that will cause the training job to fail if a validation dataset is explicitly used. Instead, we'll just configure the training job to split the training data into training and validation using a split ratio.

h. When you have your training data prepared, create a folder in an S3 bucket and place your **JSON Lines files** (*there can be more than one*) and **template.json** file inside that folder.

   Make sure SageMaker can access that folder. Simply use the default bucket created by SageMaker if you want to bypass the hassle of access control in this step.

2. Back to SageMaker Studio and JumpStart, scroll almost all the way up, back to the **Train Model** section of the model card of Llama 2 7B Chat.



a. Under **Data Source**, set the **Training data set** field to the S3 bucket and folder that contains your JSON Lines and template.json files.

b. Leave the **Validation data set** empty (*see known-bug note in item **1.g.***)

c. Under **Deployment Configuration**, you can leave everything at default values. Note that this means the training job output (model and logs) will be found in the default bucket created by SageMaker.

d. Under Hyper-parameters, we'll change a few of the default settings:

  i. Chat dataset format: **False** (*since we want Instruction fine-tuning*)

  ii. Instruction-train the model: **True** (*as above*)

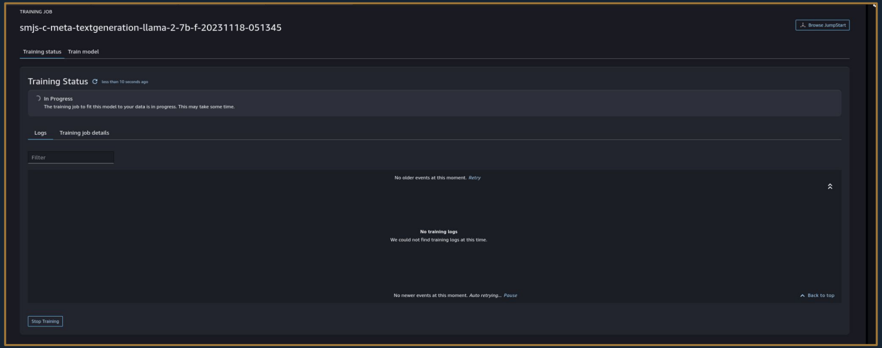
  iii. Add input output demarcation key: **False** (*our training set already has a natural demarcation included as part of the prompts, so this would be redundant*)

  iv. Epochs: **10** (*with a rather small training data set, we could use a bit more epochs than the default. Figuring out the ideal epoch can be a somewhat trial-and-error process*)

  v. Validation split ratio: **0.25** (*we just increased the training and validation split ratio so that we get at least 3 samples for validation from our training data*)

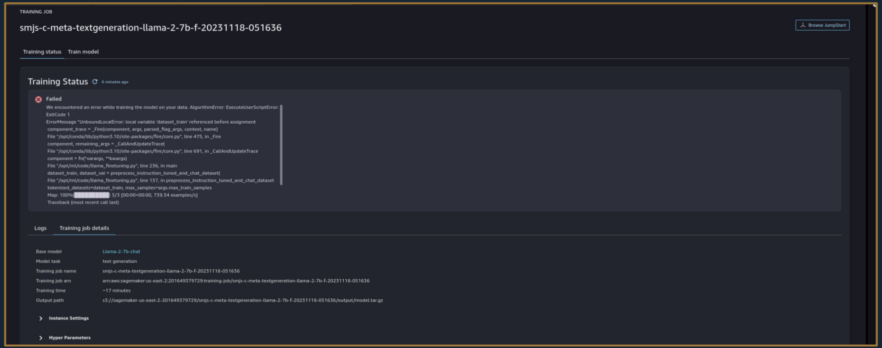
2. Start the training by clicking the **Train** button.

  a. A new tab in SageMaker Studio will open that looks like this:



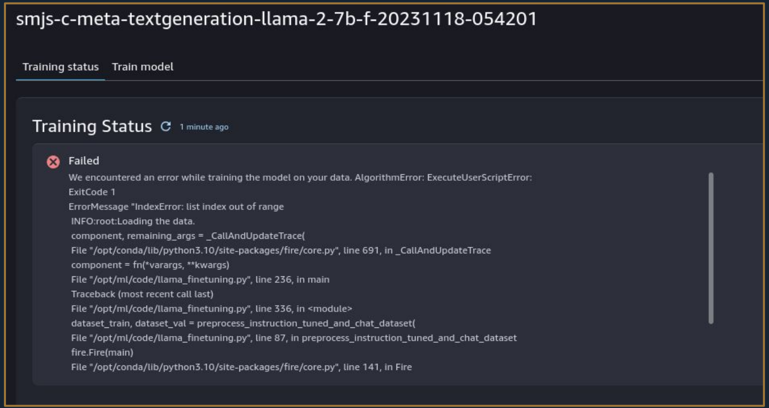
  b. You can monitor the training job from here. It may take a while before the training starts as the instance starts up and is configured.

  c. Here's one error you might encounter, "UnboundLocalError":



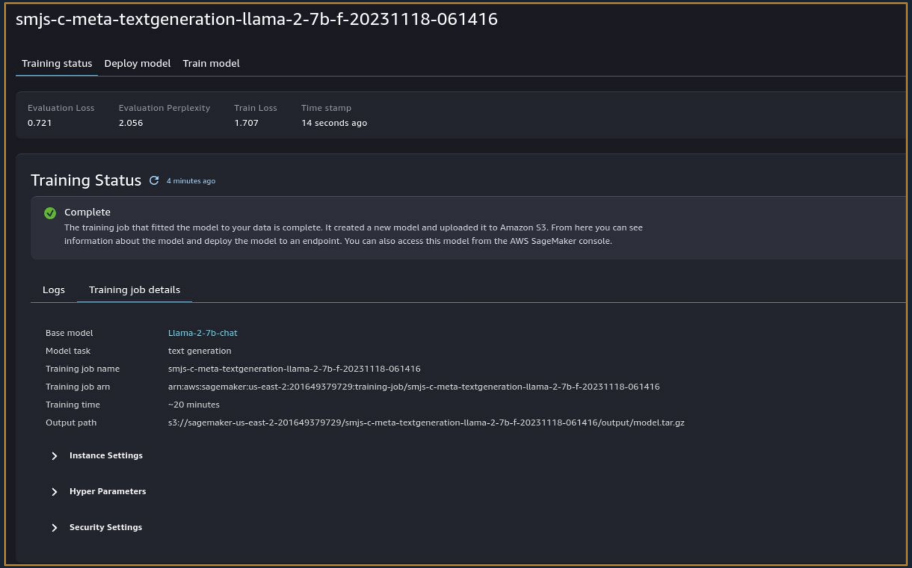
That error was due to the known-bug we talked about in item *1.g.*, when a validation data set is explicitly used. If you used a validation set and encountered this error, just skip the validation set, combine your validation data into your training data, and specify an appropriate validation split ratio under hyper-parameters.

d. Here's another common error you might encounter:



This error is caused by having a newline or empty blank line at the end of your JSON Lines file. Remove blank lines at the end of your file, reupload into your bucket, and try again.

4. Training job success!

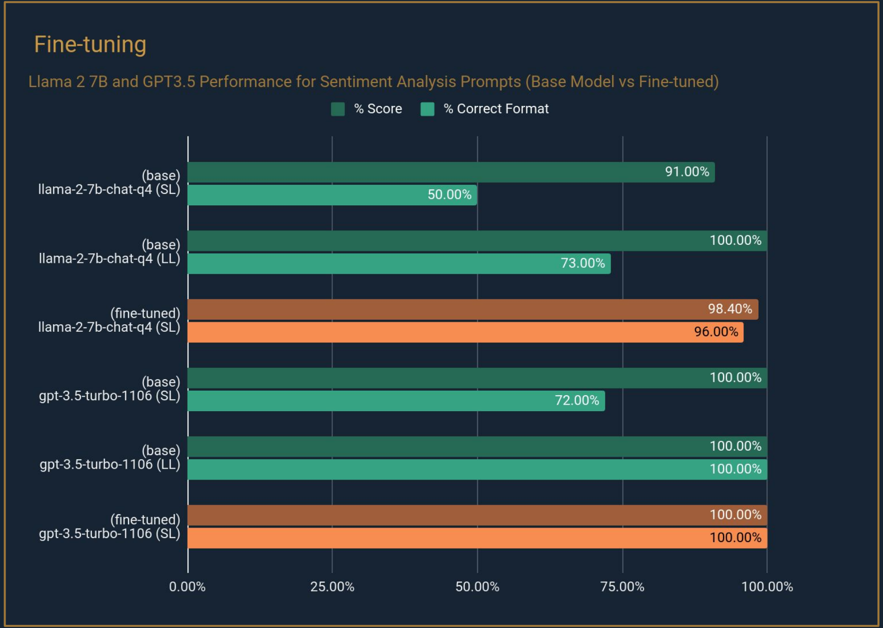


Above is the success screen when a training job completes. You'll see some stats above that measure the effects of the training, based on the training and validation sets.

When the training job is done, the model can be downloaded from your S3 bucket, and you can deploy it as needed. Just like the vanilla Llama 2 models, this fine-tuned model can be used in llama.cpp, converted to **gguf** format and quantizing it to a smaller version.

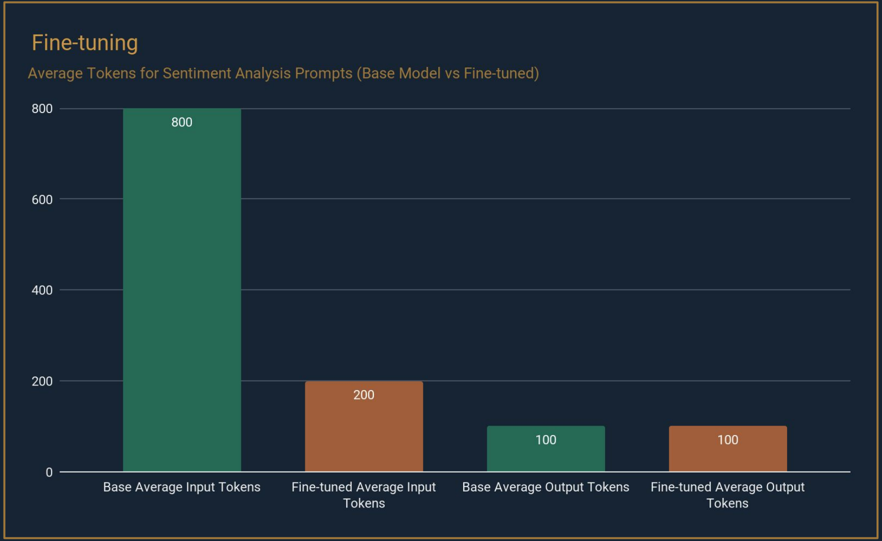
Benchmarking our fine-tuned models

Now that we've done the fine-tuning process, the real test of success is seeing if they have indeed improved their performance for the use case we fine-tuned them for, compared to the vanilla (non-fine-tuned) base model.



Running the fine-tuned models through LLM_LocalBench, we see a massive increase in adhering to our desired format (**% Correct Format**). This means we can stick with zero-shot (**SL** prompts) instead of needing few-shot (**LL** prompts) to try to make the LLMs follow the expected format.

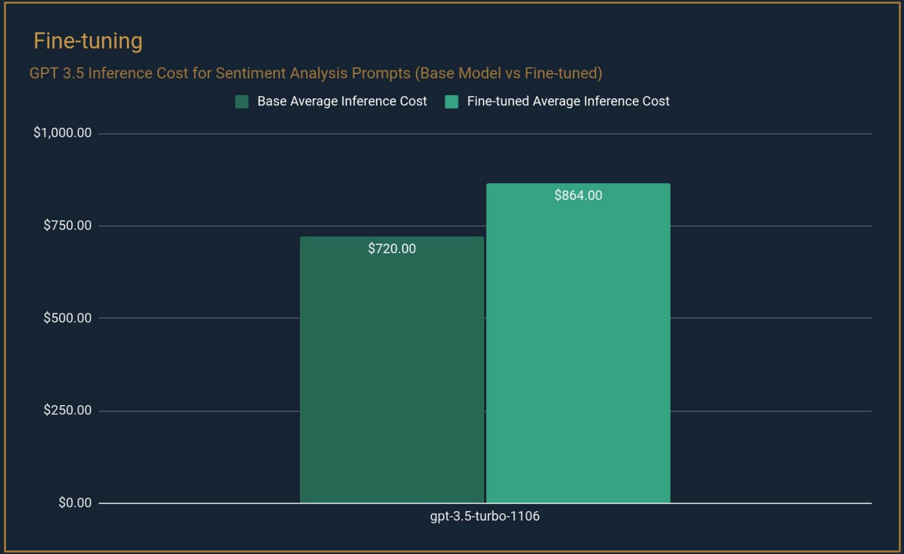
Let's look at how our fine-tuning affected our inference costs:



Fine-tuning gave us a huge decrease in average input tokens - because our fine-tuned models don't need few-shot learning anymore.

For Llama 2, since we're looking at it from a private model perspective, there's not a direct effect of lowering costs since we don't pay per token. We are saving on some VRAM and a little inference time, however, so our inference servers will be able to handle a little more volume without scaling up.

We do pay per token for GPT-3.5, so let's look at how much we're saving.



Oops, we aren't saving money!

Fine-tuned GPT-3.5 comes with a much higher per-token price tag, so although we heavily reduced our token counts, we didn't end up reducing it enough to actually offset the higher price.

There are two lessons here:

1. Always benchmark your fine-tuned models **after the fine-tuning process** to make sure the training is a success. Don't just rely on the validation set and the training and validation loss metrics.

2. Don't forget to evaluate the cost impact as well.

In this specific use case, we found that, cost-wise, we'd be better off just implementing few-shot learning on the base GPT-3.5 instead of using our own fine-tuned model.

Few-shot gave us comparable performance (*accuracy and compliance to output format*), but for almost 20% less money than using a fine-tuned model.

Fine-tuning can be very powerful, but it requires **preparing data** and paying for the **fine-tuning process** - both of which can be expensive activities.

Before settling for fine-tuning as the solution to your problem, try **prompt engineering** first. It can be a much smaller lift.

Controlling generative AI costs in the enterprise

With our gen AI strategy toolbox complete for this leg of our journey, let's put them all together to highlight essential cost control measures.

Development costs (R&D, softdev, QA)

➔ For initial R&D, exploratory tinkering and use case development, default to public models and per-token billing. In these phases, your LLM usage will likely be low enough that per-token billing will be far more cost-effective than paying for instances.

➔ Look at all the different public models that are compatible with your enterprise environment. Different platforms offer a variety of models, with varying costs.

- ◆ OpenAI's GPT-3.5 Turbo is relatively cheap and useful, and is a good default for prototyping or exploratory development of use cases.
- ◆ Bedrock offers models that are around the same cost GPT 3.5-Turbo, or even lower. For AWS-centric orgs, this is also a good first step.
- ◆ Don't overuse GPT-4 during early development or tinkering phases. It can get really expensive, *really fast*, especially if you are testing long context scenarios (*e.g., RAG use cases where the information being retrieved and added to the prompt is huge, like logs or source code*).
- ◆ The Gen AI Cost Estimator can help you shortlist the first platforms and models to test based on estimated cost.

➔ Identify a team of engineers assigned to the exploration and development of LLM use cases. Equipping them to run LLMs locally can be a great way to accelerate experimentation without adding a ton of operational costs.

- ◆ This does not have to be an expensive, brand new equipment purchase for a couple thousand of dollars each.
- ◆ Mid-range laptops that have Nvidia GPUs with as little as 4GB of VRAM are relatively cheap, but can already locally run smaller LLMs for experimentation. **Quantization** + **llama.cpp** + **partial GPU layers** offloading can be your best friend here.
- ◆ Similarly, any Mac with Apple Silicon (even just M1), and at least 16GB of RAM, can comfortably run GPU-accelerated quantized small LLMs locally.
- ◆ **If your team already has such hardware, then they're ready to go!** It's now just a matter of exposure and training.

➔ Remember: *fine-tuning is for structure, not facts*. Trying to brute-force facts by going for huge fine-tuning data sets will likely be expensive, slow and ultimately ineffective (*the absolute worst combo you could ever report to your CFO*).

- ◆ Small data sets with **good** examples can be enough. Don't go overboard. Start small, fine-tune + test, and refine and increase the data set as needed. Fine-tuning is an iterative process.

Production costs

- Look at all the different public models that are compatible with your enterprise environment. Different platforms offer a variety of models, with varying costs.
    - Use the Gen AI Cost Estimator to get an estimate of potential cost for the deployment
- Deploying specialized smaller private models can help make a wide-scale enterprise deployment more CFO-friendly.
    - See how the per-token billing costs for your scenario match up against having a dedicated LLM instance (through platforms like Hugging Face or your favorite Cloud provider)
    - Remember, though, that deciding on a private model deployment will mean extra management overhead that wouldn't exist compared to public model SaaS providers.
- Watch out for ballooning costs through RAG.
    - Particularly for per-token billing, be careful that you don't overdo your RAG information retrieval component. If your RAG adds 3,000 tokens to the prompt when a lighter but smarter 500 tokens will do, then you're adding +2,500 input tokens per transaction unnecessarily.
        - For example, don't just blindly attach 500 lines of logs for analysis. Preprocessing the logs (stripping out verbose but known useless lines, for example) before it is appended to the prompt might reduce your transaction costs significantly without impacting performance. Benchmark!
        - Experiment with optimizing your enterprise search and vector database retrieval results.
    - For private model hosting, this matters less, but still worth watching out for. As the prompt becomes bigger, more memory is used and the prompt evaluation will get slower. If it takes more time and memory to handle individual requests, then it lowers the effective capacity of your LLM instance, forcing you to scale sooner.
- Few-shot learning is an awesome prompt engineering technique to make an LLM effective for your specific use cases, but keep costs in mind
    - Just like ballooning RAG costs, few-shot learning costs can creep up on you.
    - If the few-shot learning examples are just dozens of tokens, then it isn't a big deal.
    - When they end up being over a thousand tokens or longer, then look for ways to make them smaller without sacrificing effectiveness. Benchmark!
    - Explore if fine-tuning can be an effective replacement for few-shot learning that involves a ton of tokens.

- ➔ Quantize!
    - If you don’t quantize your private model deployments, **you are burning money.**
    - Make sure quantization **begins in development**, not in production. If non-quantized models are used for dev, benchmarking, and QA, but then you deploy using a quantized model, you will still save on operational costs thanks to slimming down the model, but you will be asking for trouble.
- ➔ Proprietary models are a great way to prototype a feature, and maybe even push an MVP into production. Don’t forget Open Source foundation models, though.
    - Over time, expensive proprietary models can be replaced by cheaper foundation models.
    - This means a different set of prompt engineering, benchmarking and QA .
    - Quantization and fine-tuning are likely going to be essential.
    - Gradually, specific use cases can be taken over by more cost-optimized foundation model deployments.
- ➔ Use the most expensive proprietary models like GPT-4 judiciously.
    - **Don’t simply default on using the biggest and most expensive models** as the backbone of your production features, even though they are commonly ranked in various LLM leaderboards as the “best”.
    - The evaluations in LLM leaderboards **likely don’t translate well** to your specific enterprise use cases anyway, and you must do your own use-case-specific benchmarks.
    - Instead of using them directly in production, use expensive, more capable models like GPT-4 as a way to test and fine-tune smaller, cheaper models. These expensive, larger models are excellent for helping you create fine-tuning datasets and benchmarking and improving your actual production LLMs.
- ➔ If your production deployment is for an internal tool, consider local hosting options.
    - LLM assistants can be deployed and run locally on employee machines if they are already decent (not necessarily high-end).
    - For privacy-critical deployments, a single powerful local machine in your local network can serve as the backbone of your local LLM deployment

Misc infrastructure costs

Whether for Dev or Prod, you'll also incur some miscellaneous infrastructure costs depending on how you deploy your generative AI solution. Keep these in mind when designing your generative AI features and solutions:

- **RAG information retrieval components.**
  Enterprise search and vector databases can be a big chunk of your overall generative AI solution. Aside from the additional tokens per transaction that RAG can potentially add (we already covered that under **Production Costs**), there can also be direct costs from the usage of your enterprise search or vector database - either transaction charges from queries done for your generative AI solution, or directly paying for new instances.

- **Data ingestion, processing, and curation.**
  Especially for a RAG setup, you are probably maintaining an automated pipeline where you ingest, process (including vectorization), and curate data, to make sure you always have comprehensive and fresh data available for your generative AI use cases. These infrastructure costs can be significant, depending on your needs.

- **Logging infrastructure.**
  It's likely that you would want to log in detail the prompts and responses from your generative AI solution so that you can monitor and review what your customers are doing, and how well your LLM is performing over time. Depending on how active your generative AI solution is, that additional logging cost can be significant, and something you should also benchmark with your current logging/monitoring service.

INTERESTING READS AND RESOURCES

Optimizing Your LLM in Production
A Hugging Face blog post about effective techniques for scaling your LLM in production - quantization and beyond! Also comes with a Python notebook hosted on Google Colab, providing a hands-on experience.

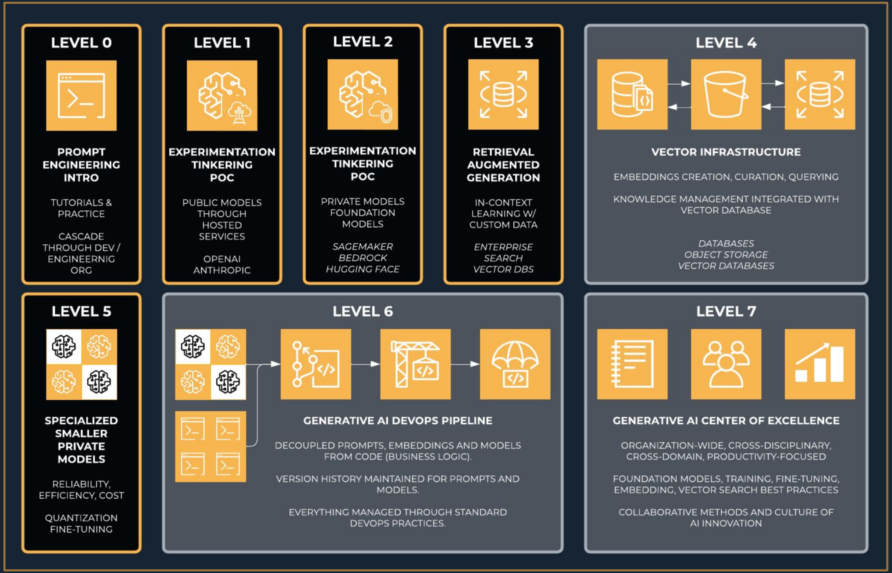
BloombergGPT: A Large Language Model for Finance
What do you get when you take an LLM and train it on a 363 billion tokens from Bloomberg data?
Cash cash monies, baby! Just kidding. We get a finance-oriented LLM, and the linked paper details their entire adventure, from modeling and training to evaluation.

How to Run Multiple Fine-Tuned LLMs for the Price of One
LoRA is yet another way to scale LLM deployment, especially when your deployment consists of multiple different fine-tuned models.

Textbooks are All You Need
There are two things I loved about this paper from **Microsoft Research**. First, it shows that smaller data sets can achieve amazing results, as long as they are high-quality; they can even have a significant amount of synthetic data (LLM-generated). Second, smaller models can kick ass with great training. *#SizeIsntEverything*

EPILOGUE:
THE JOURNEY AHEAD

Let's wrap up this current journey into generative AI by looking at our progress so far in the roadmap:



We tackled five of the eight levels in the roadmap:

- ❑ Chapter 2 dealt with Level 0 - *Prompt engineering*
- ❑ Chapter 3 dealt with Level 1 - *Experimentation and tinkering with public models*
- ❑ Chapters 4 and 5 dealt with Level 2 - *Experimentation and tinkering with private models and foundation models*
- ❑ Chapter 6 dealt with Level 3 - *Retrieval augmented generation (RAG), including vector databases.*
- ❑ Chapter 7 dealt with Level 5 - *Specialized smaller private models*

Level 4 was only barely touched in Chapter 6 - just a quick mention of the infrastructure and supporting process demands of using vector databases for RAG - and Levels 6 & 7 are still ahead of us.

Beyond these untouched levels, there's still a wealth of generative AI info, as updates in this field are incredibly rapid. New prompting techniques, research results, foundation models, evaluation methods, services, updated libraries and software all come at a breakneck speed.

So while this leg of our current adventure is over, the adventure itself is far from over. Generative AI has only just begun. Strap in, because as exciting and as wild it may already seem, it's going to get even wilder - and that means more volumes in the BRIGADE book!

In the meantime, if you haven't already, check out the main companion repo of this book: https://github.com/debbiebastes/BRIGADE. Sample codes and data sets referenced in this book can be found there.

We also referenced other repositories we maintain, and they are also worth looking into:

- https://github.com/jvroig/llm_localbench
- https://github.com/debbiebastes/llm_localbench_data
- https://github.com/debbiebastes/llm_localbench_results
- https://github.com/jvroig/genai_cost_estimator

Most of these will be continually updated, especially as we continue to run more generative AI experiments. If you want a preview of some material that'll get into volume 2, watch these repos!

Matt Wallace

Veteran CTO & Tech Executive, Host of AI Everyday Podcast

Afterword

I remember being at **AWS re:Invent in 2022** when ChatGPT released. I tried it the first day, and it helped me solve a daunting problem almost immediately. My ideas became code before my eyes. Contextual knowledge summoned as if from an Oracle, with context, answered my questions. By the time March came around, and the API was opened up, and then GPT-4 came out, it was clear the world would never be the same.

As a kid, I wanted superpowers. When I first saw **Superman II**, I started crying when Superman lost his powers. I've come to realize that this is the feeling I get leveraging AI tools - *I've been given superpowers*. You have too. Like Prometheus, we have stolen fire from the gods. And like all those origin stories, where some inept "hero" with new powers is crashing into the ground or accidentally knocking a wall in the house down, we aren't perfect with them. And like any superhero story, you and your powers are not without weaknesses. But you have the gift - and the better you can learn to use them, the more you can do.

If you've finished the BRIGADE book, you have a primer. As the book says, *"Generative AI is something that the enterprise cannot – and must not – ignore."* I appreciate that the BRIGADE book speaks to the needed journey, as much as it surveys the landscape of the technology.

It's time to take what you learned, and start to apply it. It's cliché, but the journey of a thousand miles begins with a single step. You need to keep practicing, keep creating. Whatever we thought we could do – individually, collectively – all bets are off. You can get better faster, do more, find and fill your weak spots, collaborate better, and learn more than ever before. There are all these other superheroes, too, coming into their powers. Doing more than we thought possible.

For the enterprise, it is simply a strategic imperative. Even as estimates on the economic impact spiral up into the trillions, I think they are still low. This is a young technology, rapidly maturing on every level (*the hardware, the software, the tools, and the domain knowledge like what you've learned in this book*). We currently have the 56kbps modem version of AI. That was once a high end Internet connection. Now, your phone will go thousands of times as fast, without even a wire. That is coming for AI, and it's hard to predict what the world looks like with this impact.

It's a new day.

Time to put on your cape and start creating the future with us.