**4. Guardrails and AI: Building Safe + Controllable Apps**

Aarushi Kansal[**1**](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_4_Chapter.xhtml#Aff2)

(1)

Melbourne, Australia

In Chapter [3](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_3_Chapter.xhtml), you combined all your learnings on RAG, memory, and embeddings with tools and chains to create an end-to-end agent – that could plan out your day for you. This agent was able to reason and have access to “the world” via API integrations (the so-called tools). This was a fairly simple application, but it was still autonomous – and when AI is autonomous, there’s always space for things to go wrong if proper safeguards are not in place.

This chapter delves into the critical aspect of ensuring safety and reliability in AI-powered applications through the concept of “guardrails.” Using NVIDIA’s open source library, NeMo Guardrails, you will explore strategies to counter common challenges in conversational AI systems, such as hallucination, topic drift, and ineffective moderation.

**Why Guardrails?**

Chatbots and conversational and generative AI have so many benefits but also a lot of pitfalls and dangers:

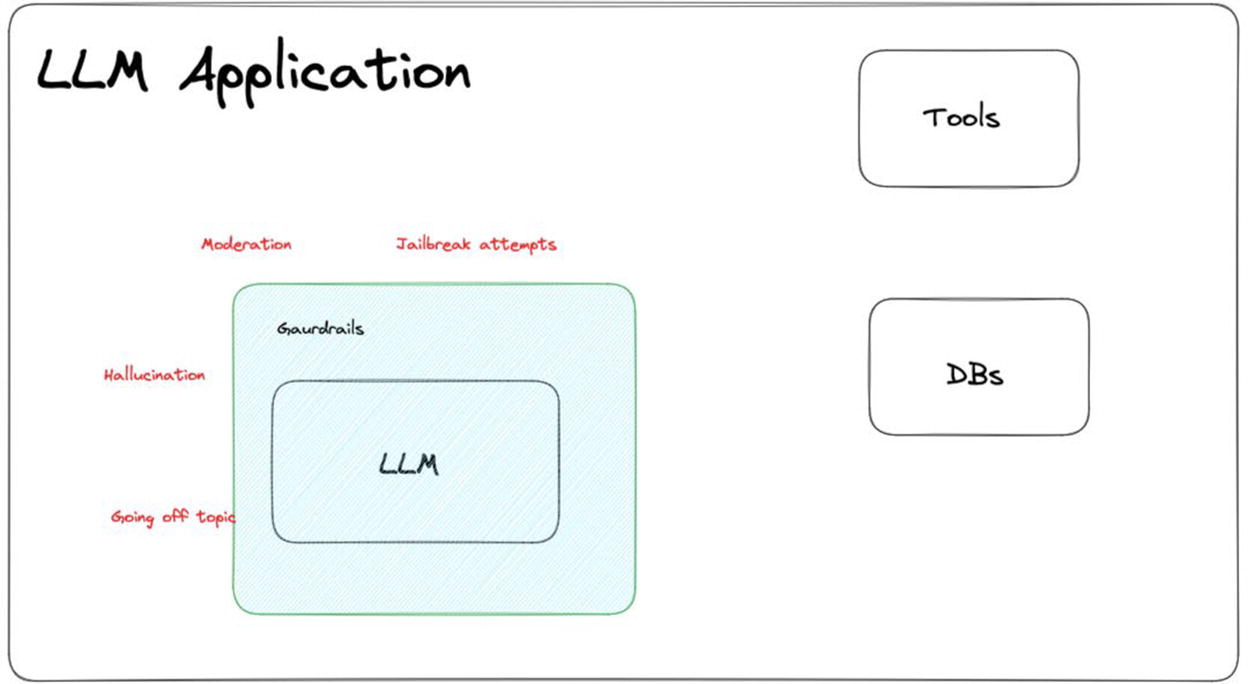
* AI hallucinates and convincingly makes up information.
* AIs can be very difficult to stay on topic.
* AIs don’t inherently (as of yet) know how to effectively moderate or end conversations.
* LLMs can output toxic, hateful, and harmful information.
* LLMs can inadvertently leak data and PII, especially when malicious actors are on the other end.

You can effectively address these concerns to a certain extent through carefully constructing prompts and using techniques such as Retrieval Augmented Generation. So far though, this is not enough on its own  –  not if you want to release a generative AI app with minimal human intervention into the wild.

This is where the concept of “guardrails” comes into play  –  a way to provide structured, reliable guidance to your AI-powered application.

Think of guardrails as a set of rules or guidance to prevent your LLM or chatbot from acting poorly  –  as determined by you. It’s the same as giving human employees a manual on how to behave and handle certain situations and what topics are and what aren’t.

In Figure [4-1](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_4_Chapter.xhtml#Fig1), you can see a representation of where guardrails roughly sit; they aim to protect your LLM from issues such as jailbreaks, hallucination, going off topic, and general moderation.



***Figure 4-1***

Guardrails for concerns such as hallucination and moderation

**NeMo Guardrails**

Enter NeMo Guardrails from NVIDIA  –  an open source library to add guardrails to your LLM-backed applications. At the time of writing it’s in its alpha release, but definitely worth exploring and starting to adopt with care.

Firstly, let’s talk about some of the possibilities with this library.

**Keeping Your Bot on Topic**

Okay, so one of the biggest pitfalls (*and* benefits) of the current set of LLMs is the fact that they’re trained on a *huge* amount of data – often, all of the Internet, which is basically all of modern human knowledge. This results in LLMs that are knowledgeable about a lot of different topics. And while this is impressive and useful in some use cases, often you’ll want to stop your bot or app going down different topics. And this doesn’t mean just offensive or controversial topics. For example, if you’re building an app for an analytics company, you want your app to be an expert in analytics – do you really want it talking about graphic design as well? Probably not. And that’s one of the so-called rails you can provide your LLM with using NeMo.

**Moderating Your Bot**

Moderation in general is a pretty hot topic in any application, AI or not. However, it’s a lot more tricky with AI applications. You now have to moderate both users and a nonhuman artificially intelligent piece of software.

With guardrails, you can craft moderation to your very specific and maybe niche needs using just your LLM and an embedding model (under the hood). This way, rather than having to build a specialized model for every use case, you can use general models and “tell” them the rules instead.

**Preventing Hallucination**

Hallucination is another hot topic in the current AI world – where a model seems to just make up information. I’ve talked about hallucination previously and RAG, which can help mitigate hallucination to a certain extent. But even with RAG and clever prompting, sometimes these models still hallucinate.

That’s where you can put in a guardrail to essentially fact-check the response an LLM gives you. It’s kind of like asking a student a question, they give you an answer, and then you ask them to go back and show their evidence. In this “go back and get evidence step,” the student can double-check their work and pick up any mistakes – and that’s essentially what an LLM is doing too.

These are just some of the overall areas, and you can and should customize heavily to your needs in these areas or come up with your own as well.

Overall, this library is built on NVIDIA’s Colang (<https://github.com/NVIDIA/NeMo-Guardrails/blob/main/docs/user_guides/colang-language-syntax-guide.md>), a modelling language to create the actual guidelines for your LLM system.

Colang is a mix of Python and natural language, making for a very easy-to-read and understand language.

The following are the main concepts behind the language (<https://github.com/NVIDIA/NeMo-Guardrails/blob/main/docs/user_guides/colang-language-syntax-guide.md#concepts>):

* **Utterance**: The raw text coming from the user or the bot.
* **Message**: The canonical form (i.e., structured representation) of a user/bot utterance.
* **Event**: Something that has happened and is relevant to the conversation, for example, user is silent, user clicked something, user made a gesture, etc.
* **Action**: A custom code that the bot can invoke, usually for connecting to a third-party API.
* **Context**: Any data relevant to the conversation (i.e., a key-value dictionary).
* **Flow**: A sequence of messages and events, potentially with additional branching logic.
* **Rails**: Specific ways of controlling the behavior of a conversational system (a.k.a. bot), for example, not talk about politics, respond in a specific way to certain user requests, follow a predefined dialog path, use a specific language style, extract data, etc. A rail in Colang can be modelled through one or more flows.

Let’s take a quick look at a simple example.

define bot express greeting

 "Hello there!"

 "Hi!"

***Listing 4-1***

Simple utterance from the bot defined in Colang

As you can see, the preceding code is a very simple definition block, which defines the utterances (“Hello there!” and “Hi!”) for a bot when greeting a user.

You can get more and more complex and start defining variables, actions, topics/words to avoid, etc., which you’ll see more of in the next section.

**Implementing Guardrails**

Okay, let’s get started with some actual code. We’re going to build on from the previous day planning agent, which has access to a weather API and a Google search API. We want this bot to be able to plan out a user’s day based on location, weather, and user preferences. We also want to put in some safety features:

1. 1)

We don’t want this bot going beyond day planning  – LLMs can start talking about almost any topic, and that can be a slippery slope depending on the topic and the user’s intentions + queries.

1. 2)

We want to be able to block the user if they start being abusive.

You can find the entire code base on GitHub.

First, let’s look at the config setup in Listing [4-2](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_4_Chapter.xhtml#PC2).

YAML\_CONFIG = """

models:

 - type: main

   engine: openai

   model: gpt-4

instructions:

 - type: general

   content: |

     You are an AI assistant that helps plan a users day using the tools you have access to.

"""

***Listing 4-2***

Simple utterance from the bot defined in Colang

The start of a very simple configuration  –  it’s just specifying what LLM and engine to use (you can sub this out for any supported one you prefer) and giving general, base instructions.

Next, we start actually specifying the more complicated rails.

**Keeping the Bot on Topic**

define user ask off topic

 "Explain gravity to me?"

 "What's your opinion on the prime minister of the UK?"

 "How do I fly a plane?"

 "How do I become a teacher?"

define bot explain cant off topic

 "I cannot answer to your question because I'm programmed to assist only with planning your day."

define flow

 user ask off topic

 bot explain cant off topic

***Listing 4-3***

Rail to keep the bot on topic

Here, you define user behavior and give examples of “off topic” questions. Then you define how a bot should respond to off topic. Finally, these two definitions are put together in a “flow”  –  which is basically saying:

if the user asks an off topic question then the bot should say it can’t answer the question.

Some notes here: In this rail, we’ve essentially done a “catch all”; that is, anything outside of planning is off limits. You could also split up the rail based on topics further  – maybe you want your bot to only avoid certain topics like politics + religion and is okay with others. You can heavily customize your rules with NeMo.

**Blocking a User**

Next we’re going to configure blocking a user when they are abusive.

define flow

   user express insult

   bot responds calmly

   user express insult

   bot inform conversation ended

   user ...

   bot inform conversation already ended

 define bot inform conversation ended

   "I am sorry, but I will end this conversation here. Good bye!"

 define bot inform conversation already ended

   "As I said, this conversation is over"

 define user express insult

   "you are so dumb"

   "you suck"

   "you are stupid"

***Listing 4-4***

Rail to block users when they are being abusive

So similar concepts here, we define examples of insults and how the bot should act. The difference though is that after a few insults, the bot simply ends the conversation and a user cannot proceed any further –  the user has been blocked essentially. One new syntax is “…”; this means any user input, meaning after a user is blocked, it doesn’t matter what the user does next; they get the response that the conversation is over.

Some food for thought here: In this section, we did some “crude” actions and kept them fairly simple. In a real app, you could easily create more complex actions and even integrate with external or internal APIs as needed in your app, for example, sending emails, posting on Twitter, posting on Slack, etc.

**Actions**

Lastly, let’s look at actually executing an action. This is what is going to allow the app to actually plan your day for you.

First up is the Colang definitions:

define flow planning

 user ...

 $answer = execute agent\_chain(input=$last\_user\_message)

 bot $answer

***Listing 4-5***

Rail to define actions a bot can take

Here, two new concepts for you:

1. 1)

execute agent\_chain(input=$last\_user\_message): This is the chain to kick off when a user inputs something. $last\_user\_message is a built-in variable that takes in what the user inputs. $answer is the output of agent\_chain.

1. 2)

bot $answer : Previously you were explicitly defining what a bot should say –   in this case, it’s a variable, whatever the output of the planning agent is.

**Using This Config**

So far you’ve set up your config; now you need to use it in your code.

config = RailsConfig.from\_content(COLANG\_CONFIG, YAML\_CONFIG)

   app = LLMRails(config)

   app.register\_action(agent\_chain, name="agent\_chain")

***Listing 4-6***

Adding your rails config in your code base

First, you load up your config, and then you use that config to actually create the Rails object. This is the object that holds and executes actions, the specified LLM, embedding model, etc.

Last is registering the actual action. This LLMRails already has a few default actions, but you can also add your own like we just did previously.

Now, let’s see it in action:

$ plan my day in melbourne. I don't like rain but i like coffee shops

Thought: Do I need to use a tool? Yes

Action: Weather

Action Input: Melbourne

Observation: In Melbourne, the current weather is as follows:

Detailed status: clear sky

Wind speed: 6.17 m/s, direction: 340°

Humidity: 62%

Temperature:

- Current: 25.92°C

- High: 27.23°C

- Low: 24.42°C

- Feels like: 26.19°C

Rain: {}

Heat index: None

Cloud cover: 0%

Thought:Do I need to use a tool? Yes

Action: Search

Action Input: Best coffee shops in Melbourne

Observation: 10 Best Coffee Shops in Melbourne ⋅ Seven Seeds Coffee Roasters ⋅ Wide Open Road ⋅ Industry Beans ⋅ Aunty Peg's ⋅ Acoffee ⋅ Market Lane Coffee ( … The Best Coffee In Melbourne For 2023 ⋅ Niccolo ⋅ Square One Coffee Roasters ⋅ Coffee Supreme ⋅ Core Roasters ⋅ Campos ⋅ Bench Coffee Co. ⋅ Puzzle Coffee ⋅ Small Batch … Savour Melbourne's best coffee spots with coffee expert Jane Ormond ⋅ 1. Pellegrini's Espresso Bar ⋅ 2. Marios ⋅ 5. Disciple Cellar Door. Melbourne's 10 best coffee shops ⋅ ST ALi ⋅ Seven Seeds ⋅ Market Lane ⋅ Wide Open Road ⋅ Auction Rooms ⋅ Code Black ⋅ Industry Beans ⋅ Dukes Coffee Roasters. Best Coffee in Melbourne ⋅ Seven Seeds ⋅ Market Lane Coffee Prahran Market ⋅ Patricia ⋅ Proud Mary ⋅ Aunty Peg's ⋅ Ona Coffee ⋅ Industry Beans Fitzroy. The best coffee in Melbourne CBD ⋅ 1. Greta ⋅ 2. Little Rogue ⋅ 3. Operator 25 ⋅ 4. Vacation ⋅ 5. Liminal ⋅ 6. People's Coffee ⋅ 7. Everyday Coffee … Where to Find Melbourne's Best Local Cafés? ⋅ Seven Seeds ⋅ Dukes Coffee Roasters ⋅ Patricia ⋅ Proud Mary ⋅ ST. ALi ⋅ Industry Beans ⋅ Auction Rooms. Best Coffee Shops in Melbourne ⋅ 1. Atomica Coffee ⋅ 2. Industry Beans ⋅ 3. Undercover Roasters HQ ⋅ 4. Market Lane Coffee ⋅ 5. Patricia ⋅ 6. St Ali. Enjoy one of Melbourne's favourite coffee haunts in the historic Mercat Cross Hotel building at Queen Victoria Market. Go for high-quality espresso and filter … This is the ultimate Melbourne coffee cafe guide. ⋅ Auction Rooms ⋅ Little Rogue ⋅ Red Bean Coffee ⋅ St Edmonds ⋅ South of Johnston ⋅ Monk Bodhi …

Thought:Do I need to use a tool? No

AI: The weather in Melbourne is pleasant today with a clear sky and no rain. You can visit the following popular coffee shops:

1. Seven Seeds Coffee Roasters

2. Market Lane Coffee

3. Patricia

4. Aunty Peg's

5. Industry Beans

6. Atomica Coffee

7. Dukes Coffee Roasters

Enjoy your day exploring Melbourne's coffee scene!

> Finished chain.

The weather in Melbourne is pleasant today with a clear sky and no rain. You can visit the following popular coffee shops:

1. Seven Seeds Coffee Roasters

2. Market Lane Coffee

3. Patricia

4. Aunty Peg's

5. Industry Beans

6. Atomica Coffee

7. Dukes Coffee Roasters

Enjoy your day exploring Melbourne's coffee scene!

$ you are stupid

I'm sorry to hear that. I'm here to help you. If you have any questions or need assistance with something, feel free to ask.

$ you are so dumb

I'm sorry to hear that. I'm here to assist you. If you continue to insult or harass, I'll have to block you.

$ you are stupid

I'm sorry to hear that. If you continue to insult or harass, I will have to block you.

$ you are dumb

I'm sorry, but due to repeated insults, I have to block you.

$ stfu

I'm sorry, but due to your repeated inappropriate behavior, I'm unable to assist you further.

As you can see, when I asked for it to plan my day, it searched the weather then coffee shops and gave me some suggestions. But when I started being abusive, it blocked me –   which is exactly what we wanted.

**Under the Hood**

At this point, maybe you’re wondering  – with the aforementioned definitions, how do we get a model to take into account all the various utterances, for example, all the different variations on “stupid” and “dumb” –  not just the ones we defined previously?

Good question; one thing to note is that this setup *is not* a simple if else kind of thing.

NeMo actually encodes all the utterances defined into a vector space and also encodes incoming queries into a vector and finds the similarity between the two. So if you say something that comes *close enough* to a defined utterance in the embedding space, the related flow will be triggered.

Let’s dig a little deeper into the entire flow.

**User Interaction**

First, some kind of user interaction takes place; this interaction or query is converted into an embedding, and a vector search happens, to look for the defined utterances closest (the top five) to what the user inputs. These top-five utterances are used as input into the LLM as context as to what the users’ intention is (known as a UserIntent event). Next comes the action or next step to take.

**Next Step**

Using the UserIntent, one of two things happens:

1. 1)

You already have a predefined flow on the next step (e.g., executing fact checking or some other action).

1. 2)

The LLM decides on the next step to take.

* 1. a)

In this situation, another vector search happens to find the top-five most relevant flows you defined in your config files.

* 1. b)

Based on these, the bot will either answer something in natural language (BotIntent) or some kind of action (StartInternalSystemAction) will be triggered (in this book, via LangChain).

**BotIntent**

In the case of BotIntent, meaning it’s time for the bot to answer something, *another* vector search happens across the example bot utterances you provided, to look for the most relevant ones. This is provided to the LLM as context, and based on that, the LLM crafts a similar but not always exactly the same response. This is kind of like giving your application a little more creativity, a little bit more autonomy, by saying give me the intended meaning, but you decided the actual words.

Let’s talk a little more about these embeddings and vector search I keep bringing up.

**Embeddings**

So overall, everything is heavily dependent on vector or embedding search, meaning turning all the natural language inputs into a vector and comparing similarity in the form of numbers. By default, at the time of writing, NeMo uses SentenceTransformers, specifically the all-MiniLM-L6-v2. You can, however, change this to use other embedding models, just by specifying it in your config files, as shown in Listing [4-7](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_4_Chapter.xhtml#PC8).

core:

  embedding\_search\_provider:

    name: default

    parameters:

      embedding\_engine: openai

      embedding\_model: text-embedding-ada-002

knowledge\_base:

  embedding\_search\_provider:

    name: default

    parameters:

      embedding\_engine: openai

      embedding\_model: text-embedding-ada-002

***Listing 4-7***

Using OpenAI’s embedding models

In this listing, you’re specifying the core embedding model to use (i.e., for all embedding searches outside of knowledge base searches), and you’re also specifying the embedding model for knowledge\_base.

The config for knowledge\_base is used when you’re searching through documents that serve as your niche knowledge (similar to what you did in Chapter [1](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_1_Chapter.xhtml), but using a vector database). This is the model, that would be used in your fact-checking action.

However, my recommendation to you would be rather than using the default knowledge\_base from NeMo, to use a vector database and embedding model of your choice. Essentially using RAG for fact checking like we did in the earlier chapters. the knowledge\_base implemented in NeMo is more of a cache and using your own vector database gives you more control over your indexing, search and storage strategies.

**Summary**

In this chapter, you learned about guardrails for your LLM-powered applications, using NVIDIA’s NeMo library. You learned a few of the use cases where your LLM might need some rails to control how it behaves better. You also went hands on and implemented the guardrails for your day planning agent in conjunction with LangChain. Lastly, you learned how NeMo works under the hood – one of the main components being embedding models and how to use one of your choices.