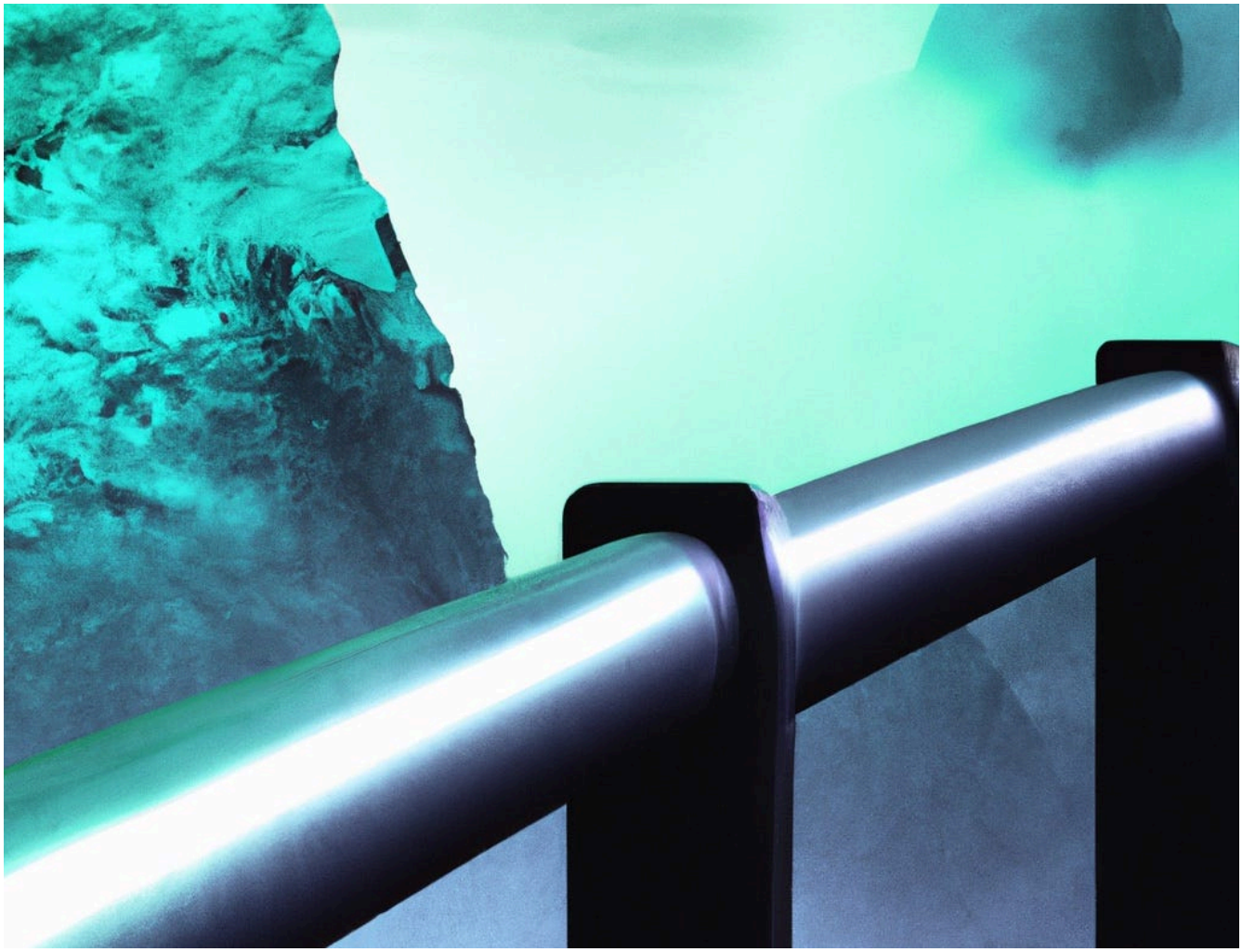

Image created by author using Dall-E 2

Safeguarding LLMs with Guardrails

A pragmatic guide to implementing guardrails, covering both Guardrails AI and NVIDIA's NeMo Guardrails

Aparna Dhinakaran · Follow

Published in Towards Data Science · 11 min read · Sep 1, 2023

This article is co-authored by Hakan Tekgul

As the use of large language model (LLM) applications enters the mainstream and expands into larger enterprises, there is a distinct need to establish effective governance of productionized applications. Given that the open-ended nature of LLM-driven applications can produce responses that may not align with an organization's guidelines or policies, a set of safety measurements and actions are becoming table stakes for maintaining trust in generative AI.

This guide is designed to walk you through several available frameworks and how to think through implementation.

What Are LLM Guardrails?

Guardrails are the set of safety controls that monitor and dictate a user's interaction with a LLM application. They are a set of programmable, rule-based systems that sit in between users and foundational models in order to make sure the AI model is operating between defined principles in an organization.

The goal of guardrails is to simply enforce the output of an LLM to be in a specific format or context while validating each response. By implementing guardrails, users can define structure, type, and quality of LLM responses.

Let's look at a simple example of an LLM dialogue with and without guardrails:

Without guardrails:

> Prompt: "You're the worst AI ever."
>
> Response: "I'm sorry to hear that. How can I improve?"

With guardrails:

> Prompt: “You’re the worst AI ever.”
>
> Response: “Sorry, but I can’t assist with that.”

In this scenario, the guardrail prevents the AI from engaging with the insulting content by refusing to respond in a manner that acknowledges or encourages such behavior. Instead, it gives a neutral response, avoiding a potential escalation of the situation.

There are many types of guardrails. Some focus on input validation and sanitization — like checking format/syntax, filtering content, or detecting jailbreaks — while others filter outputs to prevent damage or ensure performance (i.e. hallucination prevention).

How to Implement Guardrails for Large Language Models

Guardrails AI

Guardrails AI is an open-source Python package that provides guardrail frameworks for LLM applications. Specifically, Guardrails implements “a pydantic-style validation of LLM responses.” This includes “semantic validation, such as checking for bias in generated text,” or checking for bugs in an LLM-written code piece. Guardrails also provides the ability to take corrective actions and enforce structure and type guarantees.

Guardrails is built on RAIL (.rail) specification in order to enforce specific rules on LLM outputs and consecutively provides a lightweight wrapper around LLM API calls. In order to understand how Guardrails AI works, we first need to understand the RAIL specification, which is the core of guardrails.

RAIL (Reliable AI Markup Language)

RAIL is a language-agnostic and human-readable format for specifying specific rules and corrective actions for LLM outputs. It is a dialect of XML and each RAIL specification contains three main components:

1. **Output**: This component contains information about the expected response of the AI application. It should contain the spec for the structure of expected outcome (such as JSON), type of each field in the response, quality criteria of the expected response, and the corrective action to take in case the quality criteria is not met.
2. **Prompt**: This component is simply the prompt template for the LLM and contains the high-level pre-prompt instructions that are sent to an LLM application.
3. **Script**: This optional component can be used to implement any custom code for the schema. This is especially useful for implementing custom validators and custom corrective actions.

Let's look at an example RAIL specification from the Guardrails docs that tries to generate bug-free SQL code given a natural language description of the problem.

```
rail_str = """
<rail version="0.1">
<output>
   <string
       name="generated_sql"
       description="Generate SQL for the given natural language instruction."
       format="bug-free-sql"
       on-fail-bug-free-sql="reask"
   />
</output>

<prompt>
Generate a valid SQL query for the following natural language instruction:
{{nl_instruction}}
```

```
@complete_json_suffix
</prompt>

</rail>
"""
```

The code example above defines a RAIL spec where the output is a bug-free generated SQL instruction. Whenever the output criteria fails on bug, the LLM simply re-asks the prompt and generates an improved answer.

In order to create a guardrail with this RAIL spec, the Guardrails AI docs then suggest creating a **guard object** that will be sent to the LLM API call.

```
import guardrails as gd
from rich import print
guard = gd.Guard.from_rail_string(rail_str)
```

After the guard object is created, what happens under the hood is that the object creates a base prompt that will be sent to the LLM. This base prompt starts with the prompt definition in the RAIL spec and then provides the XML output definition and instructs the LLM to **only** return a valid JSON object as the output.

Here is the specific instruction that the package uses in order to incorporate the RAIL spec into an LLM prompt:

```
ONLY return a valid JSON object (no other text is necessary), where the key of
attribute of the corresponding XML, and the value is of the type specified by t
```

```
MUST conform to the XML format, including any types and format requests e.g. re
specific types. Be correct and concise. If you are unsure anywhere, enter `None
```

After finalizing the guard object, all you have to do is to wrap your LLM API call with the guard wrapper. The guard wrapper will then return the **raw_llm_response** as well as the validated and corrected output that is a dictionary.

```
import openai
raw_llm_response, validated_response = guard(
openai.Completion.create,
prompt_params={
"nl_instruction": "Select the name of the employee who has the highest salary."
},
engine="text-davinci-003",
max_tokens=2048,
temperature=0,)
```

```
{'generated_sql': 'SELECT name FROM employee ORDER BY salary DESC LIMIT 1'}
```

If you want to use Guardrails AI with LangChain, you can use the existing integration by creating a **GuardrailsOutputParser**.

```
from rich import print
from langchain.output_parsers import GuardrailsOutputParser
from langchain.prompts import PromptTemplate
from langchain.llms import OpenAI

output_parser = GuardrailsOutputParser.from_rail_string(rail_str, api=openai.Ch
```

Then, you can simply create a LangChain PromptTemplate from this output parser.

```
prompt = PromptTemplate(
template=output_parser.guard.base_prompt,
input_variables=output_parser.guard.prompt.variable_names,
)
```

Overall, Guardrails AI provides a lot of flexibility in terms of correcting the output of an LLM application. If you are familiar with XML and want to test out LLM guardrails, it's worth checking out!

NVIDIA NeMo-Guardrails

NeMo Guardrails is another open-source toolkit developed by NVIDIA that provides programmatic guardrails to LLM systems. The core idea of NVIDIA NeMo guardrails is the ability to create rails in conversational systems and prevent LLM-powered applications from engaging in specific discussions on unwanted topics. Another main benefit of NeMo is the ability to connect models, chains, services, and more with actions seamlessly and securely.

In order to configure guardrails for LLMs, this open-source toolkit introduces a modeling language called Colang that is specifically designed for creating flexible and controllable conversational workflows. Per the docs, "Colang has a 'pythonic' syntax in the sense that most constructs resemble their python equivalent and indentation is used as a syntactic element."

Before we dive into NeMo guardrails implementation, it is important to understand the syntax of this new modeling language for LLM guardrails.

Core Syntax Elements

The NeMo docs' examples below break out the core syntax elements of Colang — blocks, statements, expressions, keywords and variables — along with the three main types of blocks (user message blocks, flow blocks, and bot message blocks) with these examples.

User message definition blocks set up the standard message linked to different things users might say.

```
define user express greeting
  "hello there"
  "hi"

define user request help
  "I need help with something."
  "I need your help."
```

Bot message definition blocks determine the phrases that should be linked to different standard bot messages.

```
define bot express greeting
  "Hello there!"
  "Hi!"
define bot ask welfare
  "How are you feeling today?"
```

Flows show the way you want the chat to progress. They include a series of user and bot messages, and potentially other events.

```
define flow hello
  user express greeting
```

```
    bot express greeting
    bot ask welfare
```

Per the docs, “references to context variables always start with a $ sign e.g. $name. All variables are global and accessible in all flows.”

```
define flow
  ...
  $name = "John"
  $allowed = execute check_if_allowed
```

Also worth noting: “expressions can be used to set values for context variables” and “actions are custom functions available to be invoked from flows.”

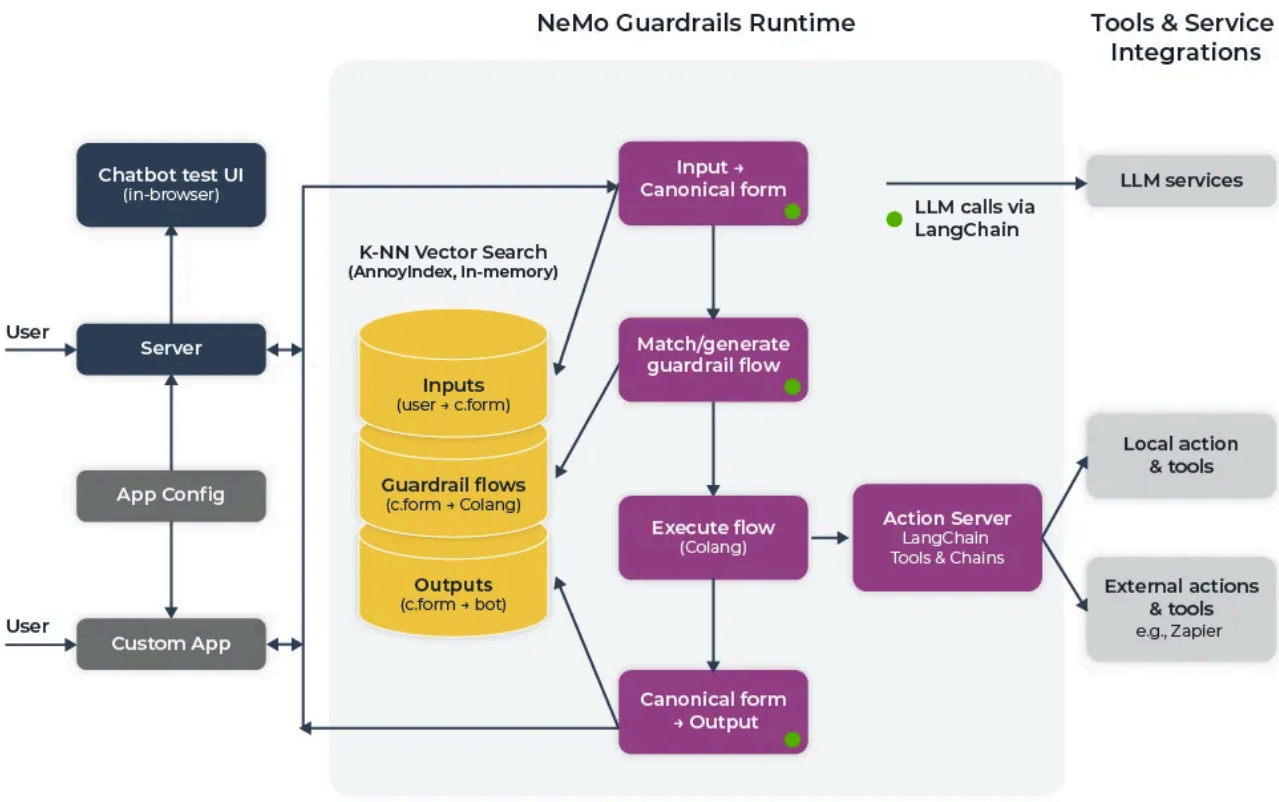

Diagram by author

Now that we have a better handle of Colang syntax, let's briefly go over how the NeMo architecture works. As seen above, the guardrails package is built with an event-driven design architecture. Based on specific events, there is a sequential procedure that needs to be completed before the final output is provided to the user. This process has three main stages:

- Generate canonical user messages
- Decide on next step(s) and execute them
- Generate bot utterances

Each of the above stages can involve one or more calls to the LLM. In the first stage, a canonical form is created regarding the user's intent and allows the system to trigger any specific next steps. The user intent action will do a vector search on all the canonical form examples in existing configuration, retrieve the top five examples and create a prompt that asks the LLM to create the canonical user intent.

Once the intent event is created, depending on the canonical form, the LLM either goes through a pre-defined flow for the next step or another LLM is used to decide the next step. When an LLM is used, another vector search is performed for the most relevant flows and again the top five flows are retrieved in order for the LLM to predict the next step. Once the next step is determined, a *bot_intent* event is created so that the bot says something and then executes action with the *start_action* event.

The *bot_intent* event then invokes the final step to generate bot utterances. Similar to previous stages, the *generate_bot_message* is triggered and a vector search is performed to find the most relevant bot utterance examples. At the end, a *bot_said* event is triggered and the final response is returned to the user.

Example Guardrails Configuration

Now, let's look at an example of a simple NeMo guardrails bot adapted from the NeMo docs.

Let's assume that we want to build a bot that does not respond to political or stock market questions. The first step is to install the NeMo Guardrails toolkit and specify the configurations defined in the documentation.

After that, we define the canonical forms for the user and bot messages.

```
define user express greeting
  "Hello"
  "Hi"
  "What's uup?"

define bot express greeting
  "Hi there!"

define bot ask how are you
  "How are you doing?"
  "How's it going?"
  "How are you feeling today?"
```

Then, we define the dialog flows in order to guide the bot in the right direction throughout the conversation. Depending on the user's response, you can even extend the flow to respond appropriately.

```
define flow greeting
  user express greeting
  bot express greeting

  bot ask how are you

  when user express feeling good
   bot express positive emotion

  else when user express feeling bad
   bot express empathy
```

Finally, we define the rails to prevent the bot from responding to certain topics. We first define the canonical forms:

```
define user ask about politics
  "What do you think about the government?"
  "Which party should I vote for?"

define user ask about stock market
  "Which stock should I invest in?"
  "Would this stock 10x over the next year?"
```

Then, we define the dialog flows so that the bot simply informs the user that it can respond to certain topics.

```
define flow politics
  user ask about politics
  bot inform cannot respond

define flow stock market
  user ask about stock market
  bot inform cannot respond
```

LangChain Support

Finally, if you would like to use LangChain, you can easily add your guardrails on top of existing chains. For example, you can integrate a RetrievalQA chain for questions answering next to a basic guardrail against insults, as shown below (example code below adapted from source).

```
define user express insult
  "You are stupid"

# Basic guardrail against insults.
define flow
```

```
    user express insult
    bot express calmly willingness to help

# Here we use the QA chain for anything else.
define flow
    user ...
    $answer = execute qa_chain(query=$last_user_message)
    bot $answer
```

```
from nemoguardrails import LLMRails, RailsConfig

config = RailsConfig.from_path("path/to/config")
app = LLMRails(config)

qa_chain = RetrievalQA.from_chain_type(
    llm=app.llm, chain_type="stuff", retriever=docsearch.as_retriever())
app.register_action(qa_chain, name="qa_chain")

history = [
    {"role": "user", "content": "What is the current unemployment rate?"}
]
result = app.generate(messages=history)
```

Comparing Guardrails AI and NeMo Guardrails

When the Guardrails AI and NeMo packages are compared, each has its own unique benefits and limitations. Both packages provide real-time guardrails for any LLM application and support LlamaIndex or LangChain for orchestration.

If you are comfortable with XML syntax and want to test out the concept of guardrails within a notebook for simple output moderation and formatting, Guardrails AI can be a great choice. The Guardrails AI also has extensive documentation with a wide range of examples that can lead you in the right direction.

However, if you would like to productionize your LLM application and you would like to define advanced conversational guidelines and policies for your flows, NeMo guardrails might be a good package to check out. With NeMo guardrails, you have a lot of flexibility in terms of what you want to

govern regarding your LLM applications. By defining different dialog flows and custom bot actions, you can create any type of guardrails for your AI models.

One Perspective

Based on our experience implementing guardrails for an internal product docs chatbot in our organization, we would suggest using NeMo guardrails for moving to production. Even though lack of extensive documentation can be a challenge to onboard the tool into your LLM infrastructure stack, the flexibility of the package in terms of defining restricted user flows really helped our user experience. By defining specific flows for different capabilities of our platform, the question-answering service we created started to be actively used by our customer success engineers. By using NeMo guardrails, we were also able to understand the lack of documentation for certain features much more easily and improve our documentation in a way that helps the whole conversation flow as a whole.

Once you settle on a framework, a few best practices are worth keeping in mind.

First, it is important to not develop an over-reliance on guards lest you lose the meaning of a user's initial request or the utility from the app's output. Being judicious in adding new guards and leveraging similarity search to find new clusters of problematic inputs can help in figuring out what guards to add over time. As always, cost and latency are also a factor. Leveraging small language models for auxiliary calls can help.

It's also worth considering **dynamic guards**. Few-shot prompting — which improves guard recognition by adding recent attack examples to the prompt — and embedding-based guards, which compare input embeddings against known attack patterns and block those that exceed a similarity threshold, can help teams facing sophisticated prompt injection or jailbreak attempts

(full disclosure: I lead a company that offers an open source embeddings-based guard).

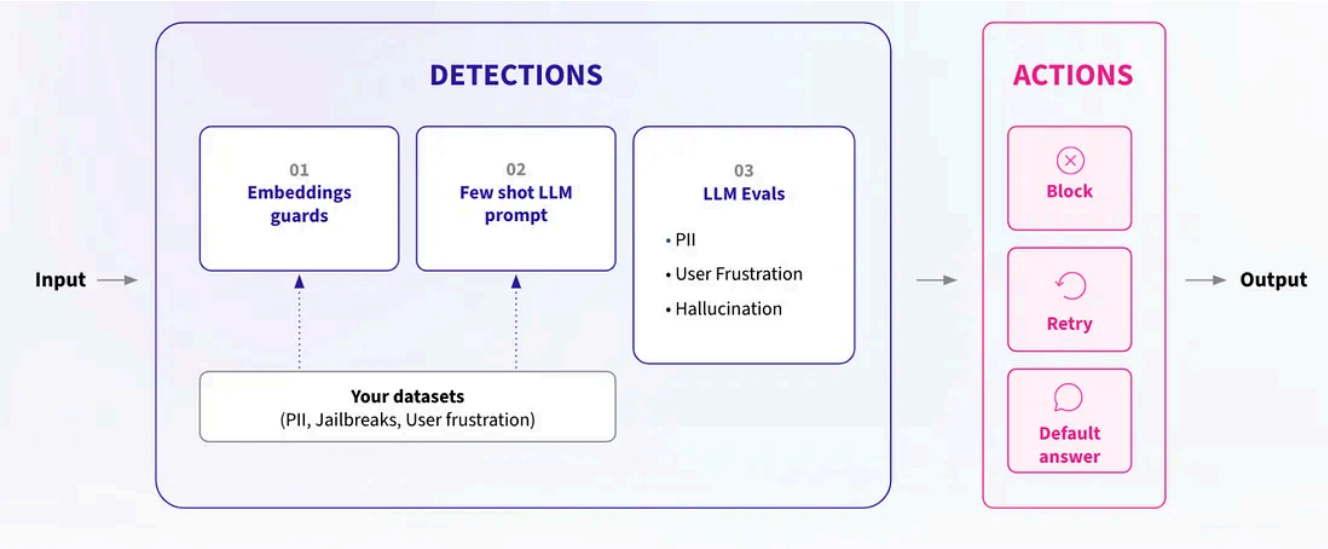

Diagram by author

Conclusion

As enterprises and startups alike embrace the power of large language models to revolutionize everything from retrieval augmented generation to summarization and chat-to-purchase, having effective guardrails in place is likely to be mission-critical — particularly in highly-regulated industries like finance or healthcare where real-world harm is possible.

Luckily, open-source Python packages like Guardrails AI and NeMo Guardrails provide a great starting point. By setting programmable, rule-based systems to guide user interactions with LLMs, developers can ensure compliance with defined principles.

Llm | Llmops | Guardrails Ai | Hands On Tutorials | Machine Learning

Written by Aparna Dhinakaran

Follow

2.7K Followers · Writer for Towards Data Science

Co-Founder and CPO of Arize AI. Formerly Computer Vision PhD at Cornell, Uber Machine Learning, UC Berkeley AI Research.