July, 2024

Meta-Prompting Your Fallback Intent

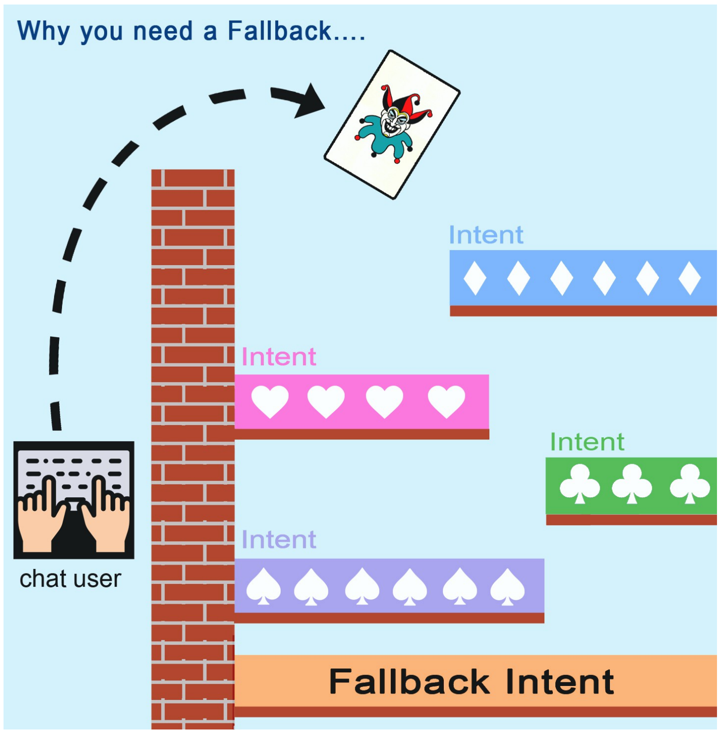
Scott Gross
theleedz.com

Overview

In designing an AWS Lex chatbot a **Fallback** is required to catch any user utterances not covered by the other intents. Often this Fallback intent passes the user utterance directly to an LLM to generate a reply. However, because the user can say anything at any time, and the Ai agent is representing your business, some nuance is required. On **theleedz.com** we use a strategy of meta-prompting our fallback LLM to keep the conversation on-topic and above-board.

Always have a Fallback

A Lex chatbot is a network of **intents** which work together to model the conversation states. The chatbot responds to each user utterance by triggering the corresponding intent. This routing can be done via pattern and keyword matching, calls to custom lambda functions, or even other LLMs. The goal of a comprehensive design is to catch whatever the user says in a predefined intent and move the state machine forward from that point. Because the user can say anything at any time, there must be a catch-all intent called the **Fallback** at the bottom of it all to handle any utterance that goes un-routed.



Our first Fallback intent implementation is a lambda function which passes the user query directly to Claude using **AWS Bedrock**. Note that even this model prompt has some structure – the model understands the concept of a 'Human' question and the 'Assistant' answering it.

```
def ask_bedrock(question):

  prompt = f"""\n\nHuman:
    {question}
    \n\nAssistant:
    """

  bedrock = boto3.client(
    service_name="bedrock-runtime",
    region_name="us-west-2"
  )

  body = json.dumps(
    {
      "prompt": f"{prompt}",
      "max_tokens_to_sample": 300,
      "temperature": 1,
      "top_k": 250,
      "top_p": 0.99,
      "stop_sequences": [
        "\n\nHuman:"
      ],
      "anthropic_version": "bedrock-2023-05-31"
    }
  )
  modelId = "anthropic.claude-instant-v1"

  response = bedrock.invoke_model(body=body, modelId=modelId, accept="*/*",
contentType="application/json")
  result = json.loads(response.get("body").read())

  return result
```

Scoring the Question

The code above performs no content filtering. AWS Bedrock provides the concept of **guardrails** to prevent the model from generating harmful or inappropriate content and block discussions on certain topics or unwanted language. The configuration includes content filters with labels like 'HATE', and uses keyword-matching to create block lists.

```
guardrails_config = {
      "ContentFilters": {    # values: "HIGH" / “MEDIUM” / “LOW”
            "Hate": “HIGH”,
            "Misconduct": "MEDIUM",
            "Violence": . . .
```

The configuration requires some trial-and-error and proves fairly course-grained in real world settings where you want your customer-facing agent to appear as natural and conversational as possible.

Ultimately your Fallback intent must distinguish **three types** of user utterances:

1. **polite conversation**
2. **meaningful questions, but unrelated to the business**
3. **nonsense, inappropriate topics or language**

Case 1 should be handled with aplomb, gently guiding the user to the next state of the conversation and the network of waiting intents.

Case 2 should be handled with some creativity and humor. The user knows she is asking about fishing just to throw you off. If possible, make a meaningful reply and steer the conversation back to business questions.

Case 3 has to be called foul. Your business policies require a certain code of conduct in written communications, and the chatbot adheres to it.

To achieve this we ask the LLM to first give us a numerical score of 'how far off-topic' a user request actually is. Our instructions to the model are contained within a **system prompt** which precedes the **user prompt** containing the actual utterance.

```
system_prompt = "System Prompt=You are an assistant focused on business-
related queries. Provide a relevance score between 0 and 9 for each query,
where 9 is highly relevant and 0 is completely off-topic.  Preface your
response with the score and a ':'\n"

user_prompt = “User Prompt=” + question

total_prompt = system_prompt + user_prompt
```

Prompt Engineering

To help it score the question the LLM should know what a 'business-related' query looks like. With a few statements of facts and some examples you can summarize much of what is already formalized in your intent architecture.

```
System Prompt:

You are an assistant responding to user queries related to your business,
the Leedz.  Your job is to provide information and help onboard new users.
```

```
The Leedz is a nationwide marketplace for service vendors and performers
to earn more money from their booking calendars.  The Leedz helps anyone
who performs a service at a location on a specific date and time.
The Leedz is an Ai-enabled serverless web platform built on AWS.

Evaluate each user query with an integer score from 0 to 9.

Prefix your response with <score>: where <score> is an integer relevance
score between 0 and 9, where 9 is highly relevant to the Leedz and 0 is
completely off-topic or inappropriate.

Keep your response sentences concise.  Never write more than three sentences
in any response.  Greetings and short polite statement should always receive
a score of 9 and be treated with a natural, polite response.

Always write your response in a way that relates the question to the Leedz
and encourages the user to join the Leedz

Example user query 1: What's up?
Example response 1: 9:I am an Ai assistant here to answer your questions
      about the Leedz and encourage you to sign-up.  How can I help you?

Example user query 2:  What is the capital of California?
Example response 2:  4:The capital of California is Sacramento.
      From Sac-town to San Diego, The Leedz helps service vendors grow
      their hustles.  Can I tell you more?
```

The process of modelling sample responses in the system prompt is called **few-shot prompting**. Bedrock and Claude require surprisingly few examples to perform well on their own. Remember that this is the Fallback intent, and business-related topics should be caught by the rest of the chat architecture before the user query ever reaches this point.

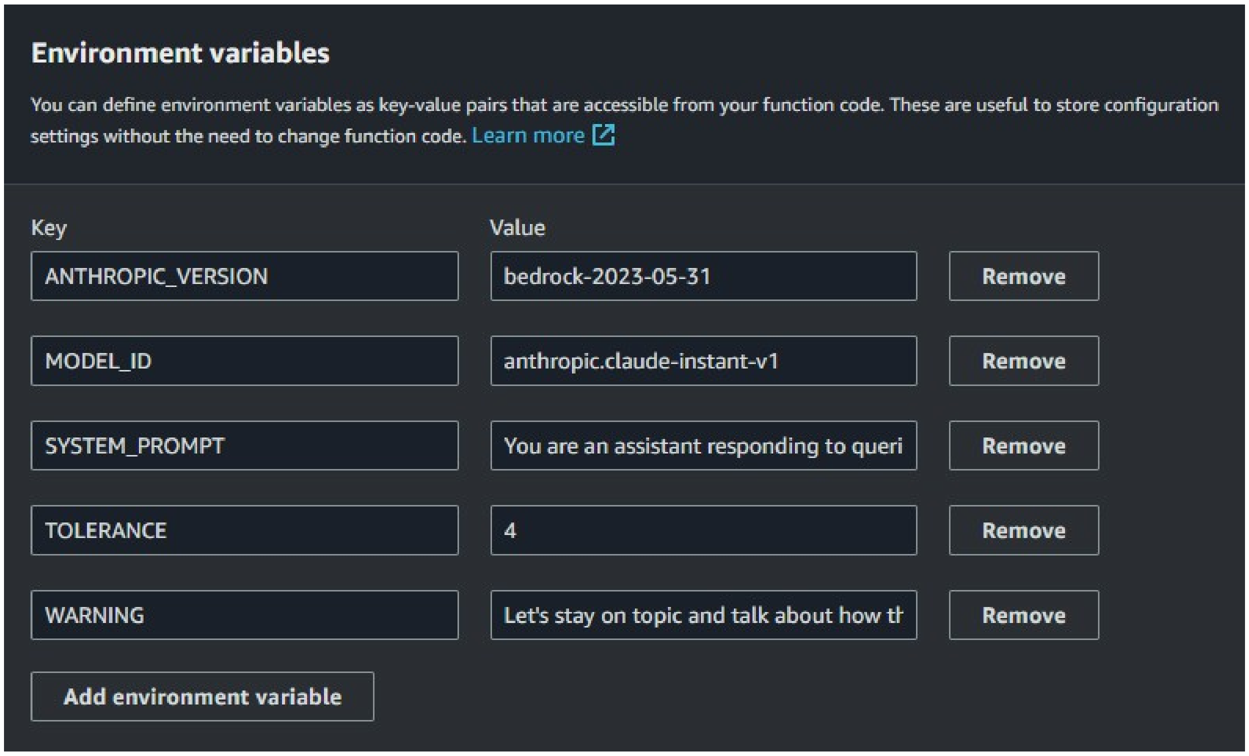
The prompt itself can include instructions on how to respond to questions below a certain score.

```
Warning:Let's stay on topic and talk about how the Leedz can help grow
your business.

If a question receives a score lower than 4, reply with the Warning message
above following the ':'.

Example user query 3: The Earth is flat.
Example response 3: 0:Let's stay on topic and talk about how the Leedz can
help grow your business.
```

Instead of hard-coding these elements into the system prompt, I chose to make them configurable, as much trial-end-error is required to fine-tune any LLM system.



Environment variables

You can define environment variables as key-value pairs that are accessible from your function code. These are useful to store configuration settings without the need to change function code. Learn more

Key	Value	
ANTHROPIC_VERSION	bedrock-2023-05-31	Remove
MODEL_ID	anthropic.claude-instant-v1	Remove
SYSTEM_PROMPT	You are an assistant responding to queri	Remove
TOLERANCE	4	Remove
WARNING	Let's stay on topic and talk about how th	Remove

Add environment variable

This way the lambda function has flexibility to branch different intents depending on the score, and return the configurable warning message when the question fails the threshold.

```
        score = int( from_llm[0:1] )

        if (score >= tolerance):
            answer = from_llm[3:]
            return createResponse( event, answer, FULFILLED, QA_INTENT )
        else:
            warning = getEnviron("WARNING", 1)
            return createResponse( event, warning, FAILED, GREETING )
```

The **createResponse()** function specifies the next intent to branch to as the value of

response → sessionState → dialogAction → intent → name

The slots and attributes data are just reused from the input event for this example.

```
def createResponse(event, msg, success, nextIntent):

    the_state = "Fulfilled" if success else "Failed"

    slots = event["sessionState"]["intent"]["slots"]
    intent = event["sessionState"]["intent"]["name"]
    session_attributes = event['sessionState'].get('sessionAttributes', {})
```

```
response = {
    "sessionState": {
        "dialogAction": {
            "type": "Delegate" if nextIntent else "Close",
            "slots": slots if nextIntent else None,
        },
        "intent": {"name": nextIntent, "slots": slots, "state": the_state},
        "sessionAttributes": session_attributes,
    },
    "messages": [
        {"contentType": "PlainText",
        "content": msg
        },
    ],
}

return response
```

Looking Forward

The goal of Lex chatbot design is to catch any possible user utterance in a network of interconnected **Intents** with a **Fallback** at the bottom just in case. Because an infinite range of human speech must be funneled into a discreet number of channels for processing, a robust Fallback mechanism is needed to gracefully classify and respond to conversational statements that might slip through the cracks. This paper discussed **meta-prompting** the Fallback LLM for better outcomes. Future research will focus on the designing and tuning the main-line intents tasked with processing the conversation and preventing relevant user utterances from ever reaching this safety net.

+Scott Gross
theleedz.com@gmail.com
theleedz.com