Amazon Bedrock’s “Sorry, I’m unable to assist you with this request” solved: a journey into troubleshooting and resolution.

15 January 2025 - 4 min. read

AI | Amazon Bedrock | Generative AI

"When you have eliminated the impossible, whatever remains, however improbable, must be the truth."

Sherlock Holmes

Having a companion (let’s call it Watson!) that helps us retrieve information and answer questions is always helpful. You know, digging and searching vast knowledge bases can be a time-consuming task. Lucky for us, these days, large language models (LLM) are here to help us—until they refuse to do their job.

In this article, we will describe the process of troubleshooting the “Sorry, I’m unable to assist you with this request” response when programmatically running a “retrieve&generate” query using Amazon Bedrock and Python boto3 library.

TL;DR

Use the boto3 Amazon Bedrock “retrieve&generate” query to provide a prompt template with the $output_format_instructions$ placeholder that has already been evaluated and create an XML parser.

This will always give you the output text that, if syntactically correct, you can parse it with your custom XML parser to extract the real text. Moreover, this still activates the boto3 built-in references XML parser to extract knowledge-base document citations.

Otherwise, if it's not aligned with the XML-defined structure, you just need to sanitize the XML tags to extract the output text.

The issue

We were tasked with implementing a knowledge base search system for a pharmaceutical company.

We chose Amazon Bedrock to solve this problem since summarization and responding to natural language queries are easy to achieve.

Unfortunately, things don't always go as planned, so when we first implemented the solution, we found that programmatically using boto3 to run a retrieve and generate query was unpredictably giving us the error **"Sorry, I'm unable to assist you with this request"** without additional information or logs.

Needless to say, no documentation is available, nor are articles on the Internet, so we started the quest to troubleshoot.

Today, we are sharing our (successful) troubleshooting journey so that you can find the solution if you are experiencing the same issue.

Keep reading!

Our implementation and use case

Let's briefly introduce the use case to better understand the context and the solution. This will be very important to understand, check by check, where the problem is.

Our task was to implement a search system to easily find drug package leaflets amongst all the documentation of the product catalog. The drug leaflet text must be accessible via API so that the feature can be integrated inside the customer web application and used by their user base to ease the user experience (UX).

The overall system can be described using 3 macro-areas:

- Storage system
- Ingestion system
- Retrieve system

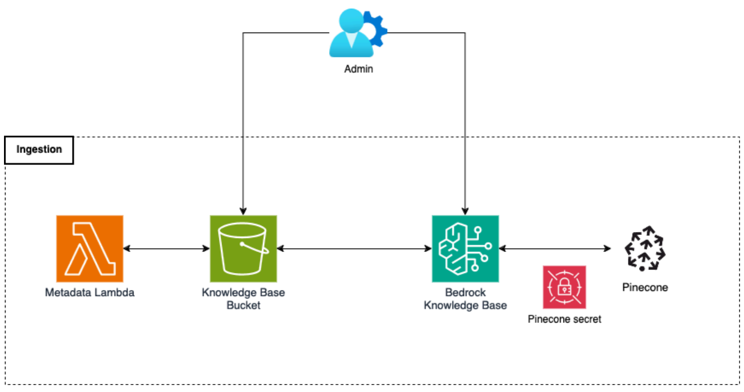
Starting from the foundations of our infrastructure, **the storage system** will support the application using an S3 bucket, vector storage, Pinecone, and a Bedrock Knowledge Base.

The S3 bucket contains all our knowledge base documents, along with their associated metadata files.

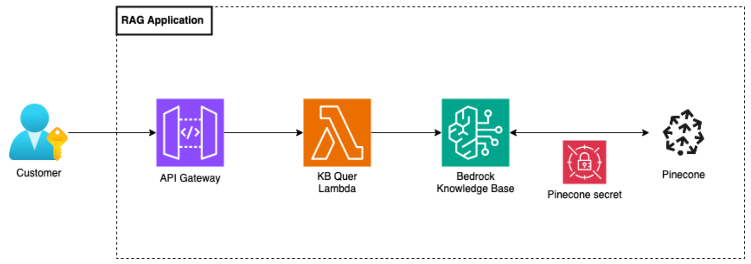
Alongside, the vector storage is the beating heart of the storage infrastructure. It contains the embedding of all chunks of our documents, along with their associated metadata. In other words, this is just a funky way to say that it holds the entire knowledge base text, split into pieces (chunks), each one transformed with a vectorial representation (embedding) that is very effective and efficient for search and retrieval tasks since it encodes the semantic information of the text inside. Given the overall size of the documents, we chose Pinecone for our vector storage solution because it fully integrates with our AWS infrastructure pieces and it was very cost-efficient for the use case.

The S3 bucket documents are processed and transferred into vector storage using the Bedrock Knowledge Base. It's an infrastructural piece that prepares and uploads documents from various configured data sources into vector storage. Moreover, the knowledge base also handles the retrieval of meaningful documents given a user query, a key element of the solution.

The ingestion system automates the processing and population of the vector storage. When an administrator user uploads a new document inside the S3 bucket, a Lambda function starts and generates a metadata file that is associated with the source document. The administrator then syncs new knowledge base documents into the Pinecone vector storage using the aforementioned Bedrock Knowledge Base feature.



Lastly, **the retrieve system** is composed of an API Gateway that will handle requests from the customer application using a Lambda function. The function takes an input question from the user and queries the knowledge base, using the retrieve&generate boto3 API call to output the requested drug leaflet text. The retrieve&generate API call uses this Anthropic Claude 3 Sonnet LLM, the best one for our use case at the time of the project.



To finalize the context, we uploaded the entire catalog documentation - about 70 documents - inside the S3 bucket and started synchronizing. After it was finished, we started testing the solution and we ran into the infamous error: "Sorry, I'm unable to assist you with this request", a 200 OK status code, and no other clue left.

Searching for the error

The quest to find the real source of this error was very hard. The road was full of traps: logical threats that tried to drive us offroad. Let's review the search process in a more structured and step-by-step ordered way.

Step 1: Generate the error

Very easy and basic step: we tried the solution with various prompts and user questions to understand if there are patterns that generate the error.

The error seemed to occur randomly. To troubleshoot the problem better, we needed to fix some variables, generate some hypotheses, and perform some further tests.

We tested a few prompts for the generation part and defined the best one to be used in our next set of tests. Once the prompt variable was set, we could create some hypotheses to understand if the problem was in the documents. Let's define them:

- **H1:** is the problem caused by user requests about missing/non-existing documents?
- **H2:** is the problem caused by user requests about specific documents?

Step 2: Documents hypothesis tests

We started testing the first hypothesis, "H1." We asked a bunch of questions about specific drugs and some about non-existent medications. The outcome of the test invalidates the assumption: the output text for non-existent drugs was sometimes correct and sometimes affected by error.

Then, we performed a similar test on "H2," gathering previous user questions and their outputs; we performed the test understanding that the issue seems to be related to specific documents because there is no correlation between "faulty" documents.

Here, the mystery deepens. The error was caused by a set of specific documents that were very different from one another. Moreover, some faulty documents are very similar to correct ones.

It seemed that we were taking steps to get closer to the solution. The next step was understanding what made a document "faulty." We limited the entropy by restricting our knowledge base to just this last set of documents and performing the retrieve API call only to see the differences in retrieved chunks between good documents and bad ones.

Step 3: Back to the basics, a few documents, retrieve only

We created an isolated environment that reflected the current solution and loaded only the documents from the last test into the knowledge base, trying to understand the differences between them.

We created a notebook to perform some analysis on our vector database and gain some insights.

First, we inspected the number of chunks inside the vector storage. The result was that the number of chunks was comparable to the number of original documents. This was not surprising since drug leaflets were small documents, and almost each one could be contained inside the chunk dimension.

Then, we performed some retrieval operations using the boto3 retrieve API call and the questions of our last test. We got an unforeseen outcome: we were able to retrieve all documents!

With this information, the problem should be inside the generation part of the retrieve&generate boto3 API call. We had just two variables: the prompt and the questions.

We were getting closer to the solution, but how could we get more information to reach it? Obviously... more logging!

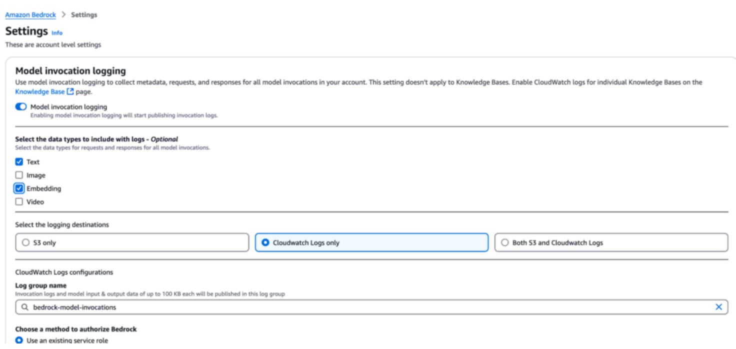
Step 4: Bedrock Logging

We needed more information to better understand how the retrieve&generate boto3 API call works, how prompts are managed, and whether specific questions could trigger unexpected conditions.

To gain more insights, we activated Bedrock Logging!

To do so, go to Settings, under Bedrock Configurations, and activate the Model invocation logging.

You need to specify which type of model you want to log their invocations and specify an IAM role to deliver logs to the proposed locations: CloudWatch, S3, or both. Since we were working with text documents, we just needed to log embeddings and text model invocations. For this case, CloudWatch logs were enough. Here is a configuration example:



Amazon Bedrock > Settings

Settings Info

These are account level settings

Model invocation logging

Use model invocation logging to collect metadata, requests, and responses for all model invocations in your account. This setting doesn't apply to Knowledge Bases. Enable CloudWatch logs for individual Knowledge Bases on the Knowledge Base page.

✓ Model invocation logging
Enabling model invocation logging will start publishing invocation logs.

Select the data types to include with logs - *Optional*
Select the data types for requests and responses for all model invocations.

- [✓] Text
- [ ] Image
- [✓] Embedding
- [ ] Video

Select the logging destinations

S3 only [✓ Cloudwatch Logs only] Both S3 and Cloudwatch Logs

CloudWatch Logs configurations

Log group name
Invocation logs and model input & output data of up to 100 KB each will be published in this log group

bedrock-model-invocations

Choose a method to authorize Bedrock
[✓ Use an existing service role]

A quick yet important note here: every single model invocation will be logged. These logs can grow very fast, and so can your billing. To avoid unexpected bills at the end of the month, activate these logs only when you really need them!

Let’s retry some retrieve&generate and understand what’s happening to the prompt, the questions, and the output.

What we found (the missing piece)

From invocation logs, we finally saw the light!

The issue was inside the prompt, in particular, the usage of the $output_format_instructions$ placeholder.
The placeholder forces the LLM, Claude 3 Sonnet in this case, to output a specific XML structure. This structure is needed to find citations inside the knowledge base text. As the AWS Documentation states: “Without this placeholder, the response won't contain citations.”

For those who are curious, in our setup, the placeholder is translated with this text:

```
If you reference information from a search result within your answer, you must include a citation to source where the information was found. Each result has a corresponding source ID that you should reference.

Note that <sources> may contain multiple <source> if you include information from multiple results in your answer.
```

```
Do NOT directly quote the <search_results> in your answer. Your job is to
answer the user's question as concisely as possible.

You must output your answer in the following format. Pay attention and fo
llow the formatting and spacing exactly:
<answer>
        <answer_part>
        <text>first answer text</text>
        <sources>
                <source>source ID</source>
        </sources>
        </answer_part>
        <answer_part>
        <text>second answer text</text>
        <sources>
                <source>source ID</source>
        </sources>
        </answer_part>
        <answer_part>
        <text>n-th answer text</text>
        <sources>
                <source>source ID</source>
        </sources>
        </answer_part>
</answer>

Assistant:
```

The issue showed up when the output text produced by the LLM was not compliant with the expected XML structure. When this happened, the boto3 retrieve&generate method internal code raised an exception, which was handled as we've seen before: a 200 OK status code and the output text "Sorry, I'm unable to assist you with this request."

Our guess is that there is an XML parser that is activated whenever the prompt template contains the **$output_format_instructions$** placeholder. This parser is responsible for extracting the clear output text and related citations. When the LLM output text is not compliant with the XML structure, the parser is not able to perform its job and raises an exception, handled as stated: "Sorry, I'm unable to assist you with this request."

The solution

Now that we've found the culprit, let's dive into how we solved this problem!

We edited the prompt template by manually replacing the $output_format_instructions$ placeholder with the previously mentioned text. We were always getting the LLM response, but unfortunately, it was usually encapsulated inside the XML structure.

Here, we noticed a peculiar feature: even though the $output_format_instructions$ placeholder was not included in the prompt template, the boto3 parser still got the citations if the LLM output was XML properly formatted. This hinted that the error was not caused by the mere parsing of the LLM XML output but rather by extracting the output text from the XML.

We wrote a bunch of lines of code to develop a very simple XML parser/sanitizer and caught two birds with one stone: the output text and the related citations.

After performing some tests, the system was stable, and we were able to integrate the solution with the customer's web application.

Wrap-up and Key Take-Aways

In this article, we focused on the interaction with a Bedrock Knowledge Base. We explored the integration with a vector database and some insights about document distribution. We investigated the boto3 retrieve&generate API call and its interplay with the prompt template, as well as its internal working mechanism, and presented a way to deep-dive into it using the Bedrock model invocation logging feature. After all this journey, we finally solved the infamous "Sorry, I'm unable to assist you with this request" output text. Eureka!

Have you ever experienced this kind of issue?

Sometimes, shedding light on the unknown can help us learn a lot about the internals and how a service works.

Share your journey in the comments!

About Proud2beCloud

Proud2beCloud is a blog by beSharp, an Italian APN Premier Consulting Partner expert in designing, implementing, and managing complex Cloud infrastructures and advanced services on AWS. Before being writers, we are Cloud Experts working daily with AWS services since 2007. We are hungry readers, innovative builders, and gem-seekers. On

Proud2beCloud, we regularly share our best AWS pro tips, configuration insights, in-depth news, tips&tricks, how-tos, and many other resources. Take part in the discussion!

Matteo Goretti

DevOps Engineer @ beSharp. Passionate about Artificial Intelligence, in particular, Machine Learning and Deep Learning, and interested in Cloud Computing. I love trekking and nature in general. I relax with my guitar, play video games, and watch TV series.
