In-Database Retrieval Augmented Generation (RAG) using Select-AI-RAG in Oracle’s Autonomous Data Warehouse

By Joe Hahn & Kevin Ortiz

[Retrieval Augmented Generation](https://en.wikipedia.org/wiki/Retrieval-augmented_generation) (RAG) is a tool that allows one to swiftly extract desired facts from a large corpus of unstructured documents. Which is very relevant to business operations, because employees are consumers of business data and they rely upon their internally published facts to fulfill their duties. And it is those high-value employees that also spend significant time searching their business data that can benefit greatly from RAG. Consider these common scenarios:

* Maintenance personnel at a manufacturing firm that must quickly troubleshoot issues across a wide variety of production equipment. A RAG knowledge base built upon the production line’s many equipment manuals allows maintenance to input a problematic part name + issue description and then have the RAG pipeline display the remedy moments later. Which sidesteps the need to search for the desired part’s manual and then skimming across that document’s troubleshooting pages.
* A quality assurance manager at a software development firm queries a RAG knowledge base containing their call-center transcripts, to obtain summary descriptions for those products that are generating the most frequent customer complaints.
* Lawyers that accelerate their legal research using RAG to extract relevant text from a huge corpus of case law and auto-generate a succinct and well-referenced legal summary.

*RAG Pipeline*

Figure 1 shows the Oracle Cloud Infrastructure (OCI) architecture utilized by the RAG pipeline detailed here. First, the RAG admin deposits all desired documents in a *knowledge base* that, in this example, is an [OCI Object Store](https://www.oracle.com/cloud/storage/object-storage/) bucket. The RAG admin then launches a job that (i) splits each document into multiple chunks of text that are typically a few sentences long, (ii) sends each chunk into an [embedding model](https://www.pinecone.io/learn/vector-embeddings/) that recasts the text chunk as a vector encoding—noting that a vector is simply a list of numerical values that preserves the text chunk’s semantic meaning—and (iii) stores the text chunk alongside its vector in a vector database. And this blog post will show that this job is easily launched via simple code that the RAG admin executes within OCI’s [Autonomous Data Warehouse](https://www.oracle.com/autonomous-database/autonomous-data-warehouse/) (ADW). That ADW instance will serve as this RAG solution’s vector database, and that ADW job will also interact with a few other OCI services, namely Object Store and the GenAI service, as well as the OML and APEX subservices that are provided by ADW (see Figure 1) and are detailed below. Readers interested in rebuilding this RAG pipeline within their OCI tenancy are also referred to this [archive](https://github.com/oracle-nace-dsai/rag-23ai-demo-archive/tree/main/data) that provides all instructions and codes.

To use this RAG solution, the user enters a request, which can be a question or a command, into a frontend interface, and this example uses [APEX](https://www.oracle.com/application-development/apex/), Figure 1. That frontend passes the request to ADW that then calls the embedding model (provided by the GenAI box of Figure 1) which recasts that request as a vector. That vector database then performs a vector similarity search to find the top N text chunks that are most semantically similar to the user’s request. Those N text chunks are *context documents*, and that context is forwarded to the [large language model](https://en.wikipedia.org/wiki/Large_language_model) (LLM, also proved by GenAI service of Figure 1) with these instructions: use the context to fulfil the user’s request. Which instructs the LLM to use the relevant proprietary business context that is stored in the RAG knowledge base to answer the user’s question. And prevents the LLM from using its internal knowledge (which is always out-of-date by months or more, can be incomplete, and is drawn from public-domain sources that can differ significantly from the more germane up-to-date facts that are stored in the proprietary RAG knowledge base) to satisfy the user’s request. Instructing the LLM to be mindful of the context also reduces the chance that the LLM’s response will be speculative, incomplete, or a hallucination.

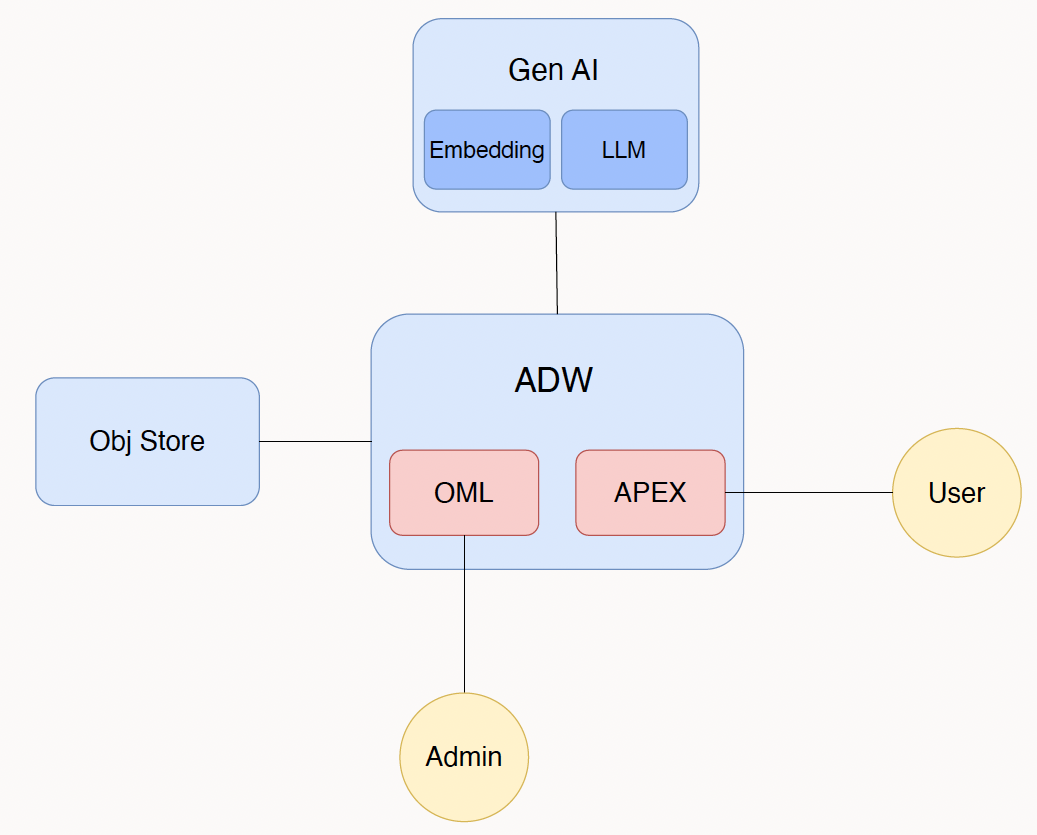


Figure 1. Architecture of the Select-AI-RAG pipeline deployed in OCI.

Note also that if open-source embeddings and LLMs are utilized, which are both quite effective in RAG, then this AI-powered solution can have a modest price tag. The principal compute costs will be the inferencing costs for the LLM and the embedding models, and the vector database. Consequently, RAG can be a very cost-effective way of keeping an AI-powered solution aware of one’s rapidly evolving business data without the need for expensive training and retraining of AI models on [GPU](https://en.wikipedia.org/wiki/Graphics_processing_unit).

*Input Data: Novels from Project Gutenberg*

The RAG experiment developed here will use 20 novels that are downloaded from the [Project Gutenberg](https://www.gutenberg.org/) website and stored in the Object Store bucket of Figure 1; see Figure 2 for a screengrab of that data. Since this post is about using RAG for business purposes, one might question the use of fictional stories here. But this dataset allows one to swiftly test the quality of this RAG solution: since all these stories are known to these blog authors and are likely known to most readers, it will be easy to tell at a glance whether the vector similarity search behaves as expected, and if the LLM’s summary answer is correct.

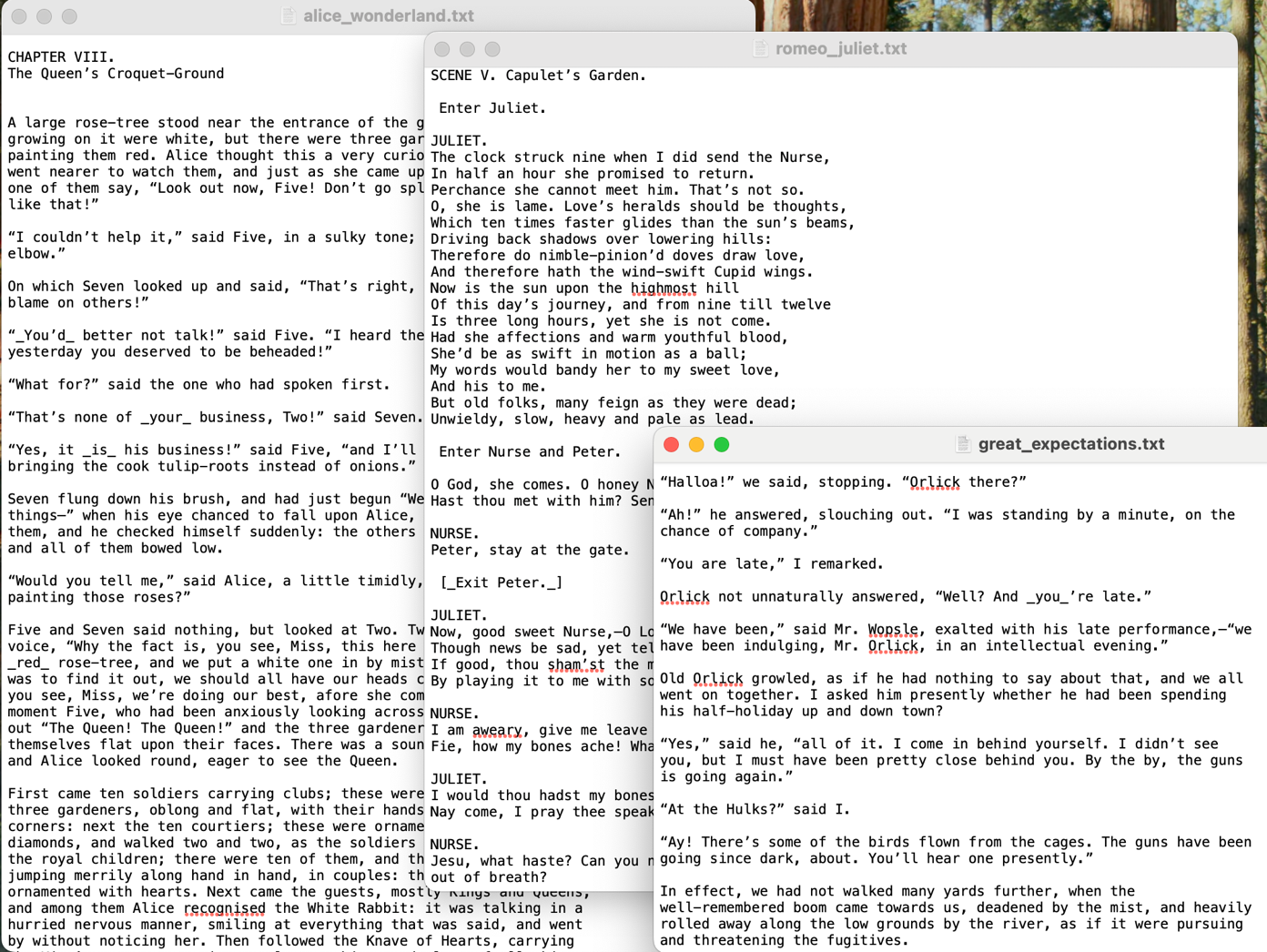


Figure 2. Screengrab of three of the 20 novels downloaded from Project Gutenberg and stored as text files in the Object Store bucket of Figure 1.

*Select-AI-RAG*

The functionality utilized by the ADW job described above is known as [Select-AI-RAG](https://blogs.oracle.com/database/post/announcing-select-ai-with-rag-on-adb), and its setup and configuration are simple and straightforward. For this experiment, all our one-time setup code is executed as SQL and PL/SQL executed within an [Oracle Machine Learning (OML) notebook](https://docs.oracle.com/en/database/oracle/machine-learning/oml-notebooks/) that is provided by ADW, Figure 1. An OML notebook is a convenient place to execute database code snippets with results displayed inline, and all of this experiment’s notebooks plus additional instructions are archived in this [github repository](https://github.com/oracle-nace-dsai/rag-23ai-demo-archive/tree/main).

Firststep is for the ADW admin to create the database user that will configure and deploy this RAG solution, and in this example that user is named SAIRAG and has these permissions granted by the admin:

create user SAIRAG identified by "<password>";

grant dwrole to SAIRAG;

grant unlimited tablespace to SAIRAG;

grant execute on DBMS\_CLOUD to SAIRAG;

grant execute on DBMS\_CLOUD\_AI to SAIRAG;

grant oml\_developer to SAIRAG;

Next step is to create a PEM-formatted SSH key pair, which is easily done in an [OCI cloud shell](https://docs.oracle.com/en-us/iaas/Content/API/Concepts/cloudshellintro.htm) session via

oci setup config

which creates a new key pair in folder ~/.oci. Then copy the public key from ~/.oci/oci\_api\_key\_public.pem and paste that as an API key into the RAG admin’s OCI account. Then add the private key ~/.oci/oci\_api\_key.pem to the following code snippet

BEGIN

DBMS\_CLOUD.CREATE\_CREDENTIAL (

credential\_name => 'RAG\_CREDENTIAL',

user\_ocid => 'ocid1.user.oc1..aaaaaaaXXXsm72wxaakgqqq',

tenancy\_ocid => 'ocid1.tenancy.oc1..aaaaXXXpbme4454xu7dq',

private\_key => '

-----BEGIN PRIVATE KEY-----

MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQC+5pZKurggk5aG

z60xAk/cx855jjeHmM6y86fjnMOQC6C7OKibloJh+WecIxPihOSiBrr/rYaMqOW0

+qeQH7OBnnvRwJzu7FHBiJckgF79sP+Zkc44iP0L1pdOxFWSQhJe7Tf6YFEF4CVa

...

QiCEbN9eZ3EJUvfKTCNb0WQQYplndCCXUTQlTwYNk3eWahoiNLrRQZQqisLESnTr

wQtGjFzioG3MX9txaijlzXvvZwjqc8aFvMA3yY63cODFur3AwqjqH1Lu223NDs99

5uzYVFg0Y4rfrG9XqUGEDbU=

-----END PRIVATE KEY-----

',

fingerprint => 'c7:6c:5a:29:f2XXXXXXX52:f7:83:31:46:b3:45'

);

END;

/

which can be executed within an OML notebook (see for example [RAG\_BOOKS.dsnb](https://github.com/oracle-nace-dsai/rag-23ai-demo-archive/blob/main/RAG_books.dsnb)) or as an SQL script or within an APEX app. Be sure to tailor the OCIDs in the above so that they refer to the RAG developer’s OCI tenancy, and that the key fingerprint matches that in ~/.oci/config. This credential, named RAG\_CREDENTIAL, allows ADW to interact with the files stored in Object Store as well as the GenAI service of Figure 1.

*Select-AI-RAG Profile*

Next step is for user SAIRAG to create the profile that will utilize Select-AI-RAG:

BEGIN

DBMS\_CLOUD\_AI.CREATE\_PROFILE(

profile\_name => 'RAG\_PROFILE',

attributes =>

'{"provider": "oci",

"credential\_name": "RAG\_CREDENTIAL",

"region":"us-chicago-1",

"oci\_compartment\_id":"ocid1.tenancy.oc1..aaaaXXXpbme4454xu7dq",

"vector\_index\_name": "RAG\_INDEX",

"embedding\_model": "cohere.embed-english-v3.0"

}'

);

END;

/

The profile named RAG\_PROFILE tells ADW that the invoking user will use the RAG\_CREDENTIAL to call the cohere.embed-english-v3.0 embedding model provided by OCI’s GenAI service when performing the vector embedding of the content that is stored in Object Store, Figure 1.

*Encode the Knowledge Base*

We are now ready to use the above to encode aka embed the RAG knowledge base composed of 20 novels stored in Object Store. That is easily done vie this PL/SQL one-liner

BEGIN

DBMS\_CLOUD\_AI.create\_vector\_index(

index\_name => 'RAG\_INDEX',

attributes => '{"vector\_db\_provider": "oracle",

"vector\_table\_name": "RAG\_VECTORS",

"object\_storage\_credential\_name": "RAG\_CREDENTIAL",

"location": "https://objectstorage.us-chicago-1.oraclecloud.com/n/<namespace>/b/<BucketName>/o/rag/doc/",

"profile\_name": "RAG\_PROFILE",

"vector\_dimension": 1024,

"vector\_distance\_metric": "cosine",

"chunk\_overlap": 64,

"chunk\_size": 1024,

"refresh\_rate": 1440

}');

END;

/

which triggers a [database job](https://docs.oracle.com/en-us/iaas/autonomous-database-serverless/doc/dbms-cloud-ai-package.html#GUID-CB37AB86-B625-4798-A3F4-8DD6DBA8B491) that (i) gathers all files (text, Word, and text-based pdf) at Object Store location, (ii) splits those files’ text content into slightly overlapping (since chunk\_overlap=64 characters) chunks that are chunk\_size=1024 characters long, and (iii) feeds each of those chunks into the embedding model specified in RAG\_PROFILE whose output vectors are vector\_dimension=1024 elements long, with all text chunks and their corresponding vectors being stored in an ADW table named RAG\_VECTORS, and with (iv) index RAG\_INDEX built on that vector table to accelerate vector searches across that content. See also [DBMS\_CLOUD package](https://docs.oracle.com/en-us/iaas/autonomous-database-serverless/doc/dbms-cloud-ai-package.html#ADBSB-GUID-000CBBD4-202B-4E9B-9FC2-B9F2FF20F246) for more details about this functionality.

Chunking and embedding this experiment’s 20 novels that are a few hundred pages of text each requires about 1 minute to complete, and concludes the setup of Select-AI-RAG in ADW.

*Query the RAG Knowledge Base*

To query the RAG pipeline via code, use

BEGIN

DBMS\_CLOUD\_AI.SET\_PROFILE(profile\_name => 'RAG\_PROFILE');

END;

/

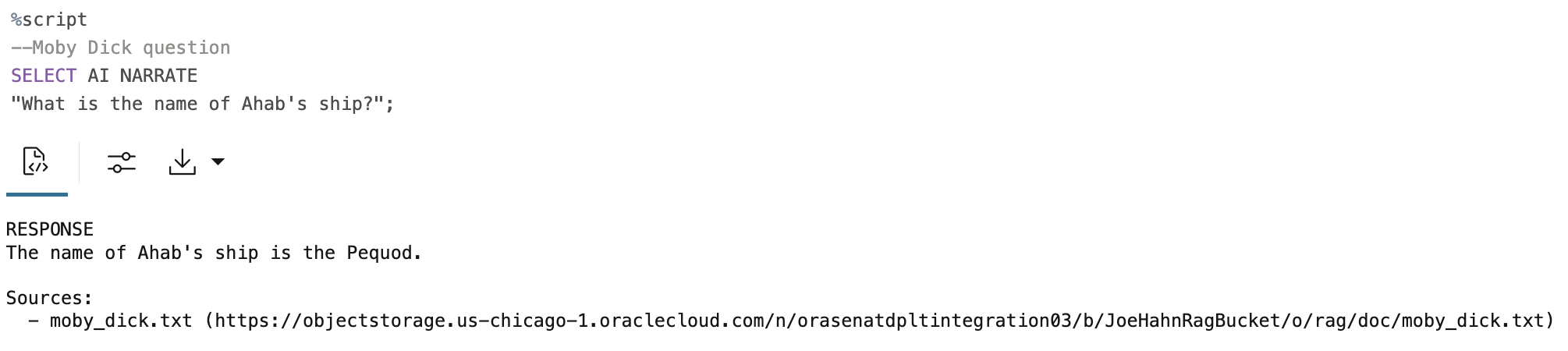
to tell your SQL script or OML notebook or database application that you intend to call Select-AI-RAG using profile RAG\_PROFILE. To submit a question to Select-AI-RAG programmatically, precede any question with SELECT AI NARRATE, such as

Figure 3. Screengrab of a Select AI RAG query executed in an OML notebook .

The SELECT AI NARRATE in the above instructs the database to encode the question, which recast’s that text as a vector, and then perform a [vector similarity search](https://docs.oracle.com/en/database/oracle/oracle-database/23/nfcoa/ai_vector_search.html) across table RAG\_VECTORS to find the top N=5 text chunks whose encodings are most similar to the question’s encoding. The database then feeds those 5 text chunks, which are known as the context, into a large language model (LLM) provide by the OCI [GenAI](https://docs.oracle.com/en-us/iaas/Content/generative-ai/overview.htm) service (which today defaults to Meta’s Llama v3.3 LLM) plus the user’s question plus these instructions: use only the context to answer the question.

The above question is clearly about Moby Dick and is intended is to spot-check Select-AI-RAG. The fact that the database reports only a single Source, moby\_dick.txt, tells us that all 5 text chunks returned by the vector similarity search came from the expected Object Store document, and that those text chunks contained sufficient relevant information to allow the LLM to formulate a correct answer to the user’s question.

*Vector Similarity Search*

To manually inspect the top 5 text chunks returned by the above vector similarity search, replace NARRATE with RUNSQL, as in Figure 4.

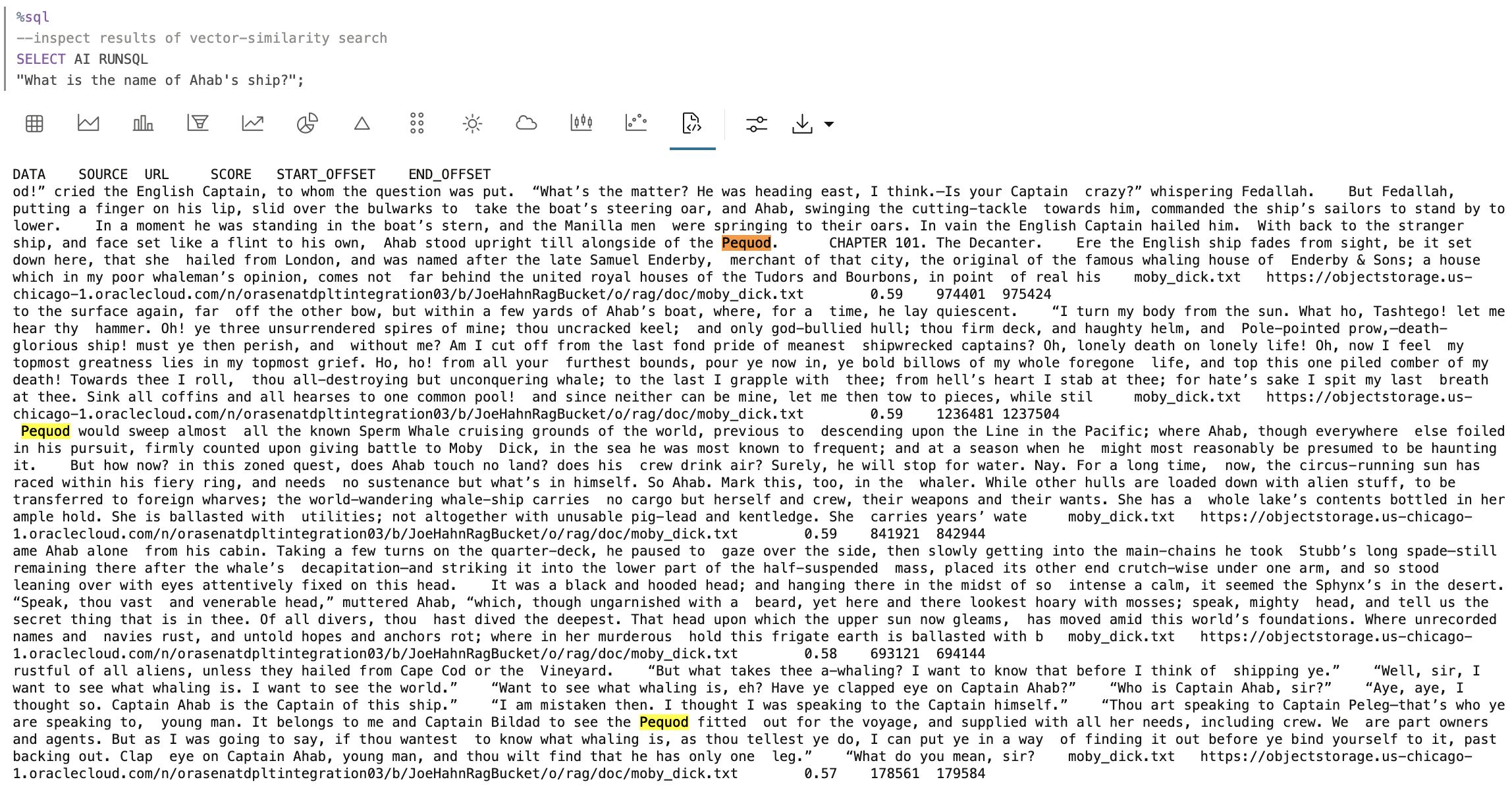


Figure 4. Using SELECT AI RUNSQL <question> to inspect vector similarity search results for <question>.

The match\_limit setting determines the number of search results returned by Select AI RAG, which defaults to N=5 and that can be changed via the following,

BEGIN

DBMS\_CLOUD\_AI.UPDATE\_VECTOR\_INDEX(

index\_name => 'RAG\_INDEX',

attribute\_name => 'match\_limit',

attribute\_value => '25'

);

END;

/

Also keep in mind that Select-AI-RAG caches its queries, so if match\_limit is changed and the same query rerun, you will still see the same 5 search results until you change a few characters in your question to force a redo.

*Building an APEX Frontend for Select-AI-RAG*

APEX is Oracle’s low-code platform for developing secure scalable business applications on top of database data. And the development of an APEX frontend for this RAG pipeline is also quite straightforward since APEX comes with and is preconnected to the ADW service where the data resides. The APEX developer begins the app-development by copy/pasting one of the code snippets from the OML notebook shown above into the APEX Code Editor, Figure 5, which when executed by APEX tells ADW to rebuild the RAG pipeline.

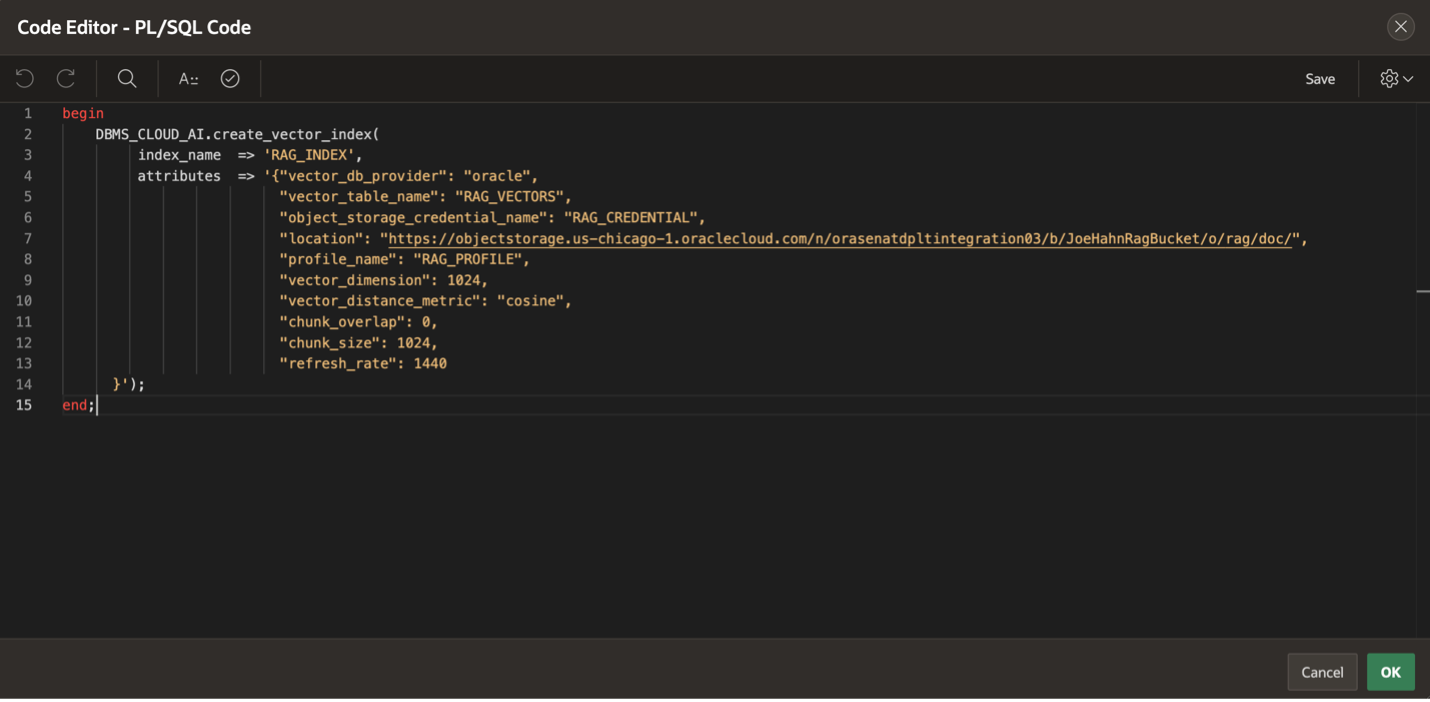


Figure 5. APEX Code Editor uses the DBMS\_CLOUD\_AI.create\_vector\_index()command to initiate the database job that will chunk and encode all documents stored in the Object Store knowledge base.

This APEX application must also collect the user’s question and then append that to a SELECT AI NARRATE SQL command, which APEX does this via dynamic actions. In the APEX developer’s backend view, Figure 6, the user’s <question> is represented by APEX object :P3\_QUESTION. And when that object changes as the user inputs a question, APEX automatically invokes the code snippet seen in Figure 6 that sends the updated :P3\_QUESTION into the custom get\_ai\_response() function defined in Figure 7, with that code snippet calling DBMS\_CLOUD\_AI.GENERATE() to execute SELECT AI NARRATE <question> in ADW.

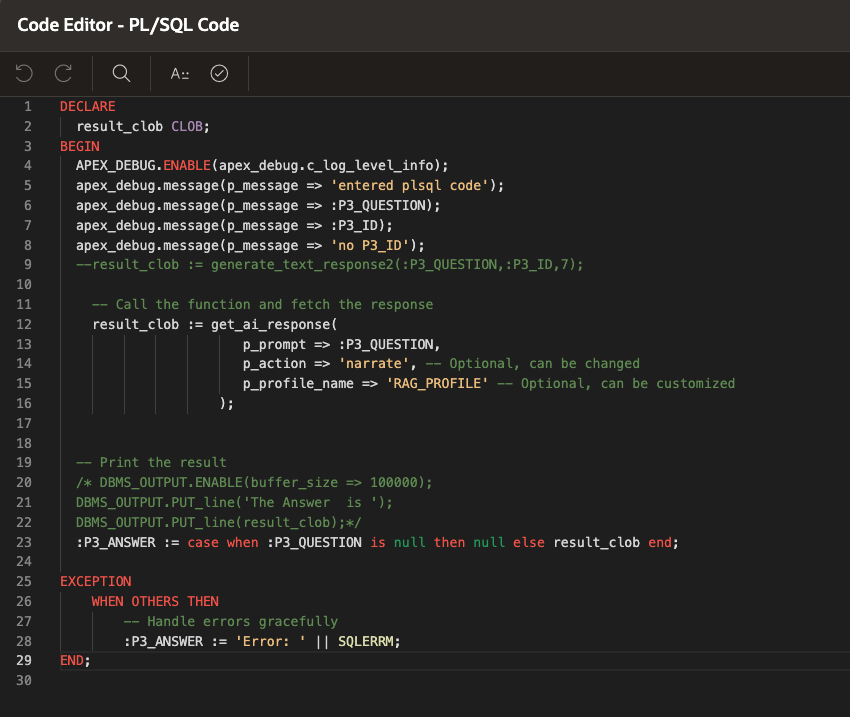


Figure 6. PL/SQL for the dynamic action that is triggered when user asks a question, which then triggers callto the get\_ai\_response() function shown in Figure 7.

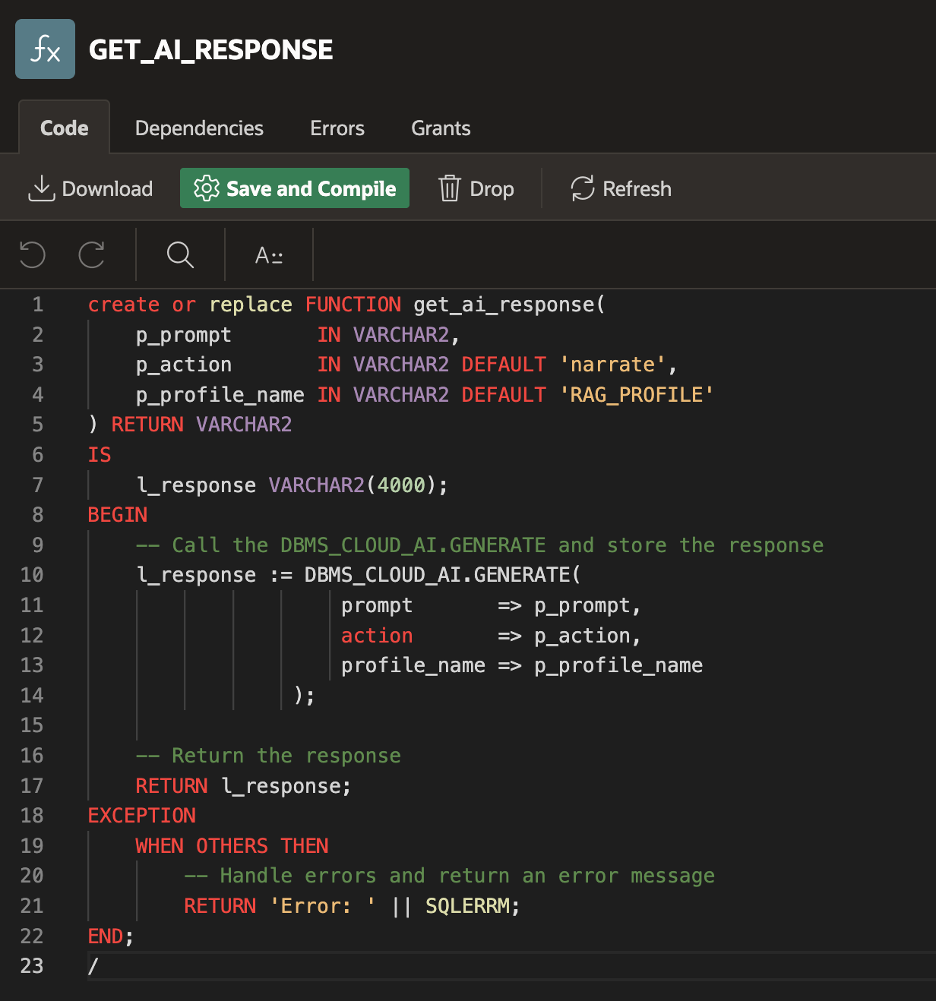


Figure 7. PL/SQL code snippet that APEX uses to gather user’s <question>, represented by p\_prompt, that is sent into the DBMS\_CLOUD\_AI.GENERATE()call that executes SELECT AI NARRATE <question> in ADW.

The user-facing side of APEX is shown in Figure 8, which displays Select-AI-RAG’s answer to “What does Scrooge think about Christmas?”

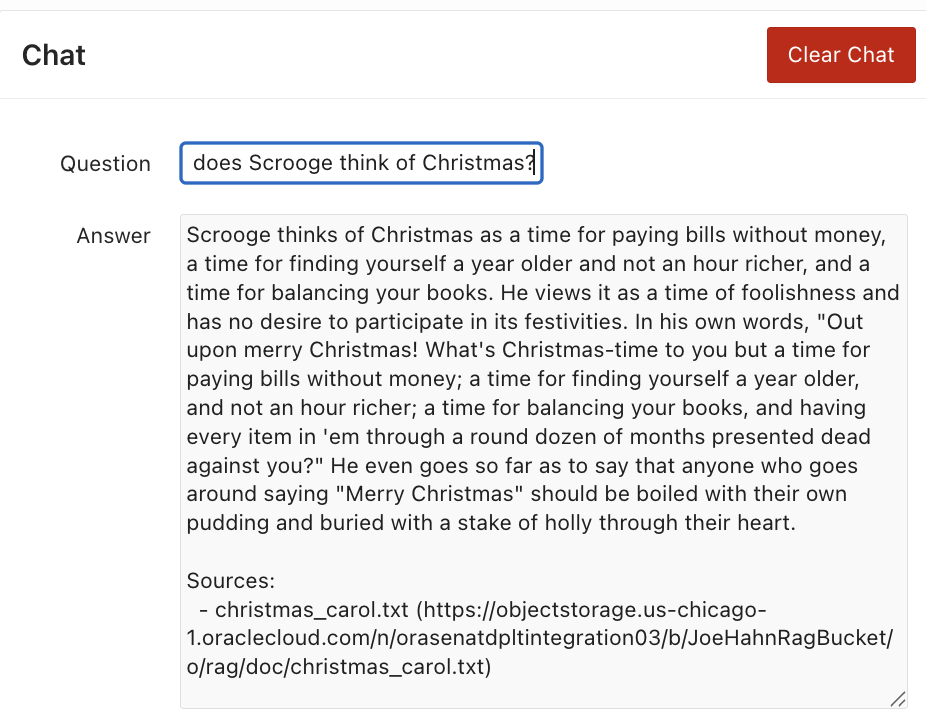


Figure 8. APEX frontend showing SELECT-AI-RAG’s answer to a Christmas Carol question.

*Main Findings*

The above shows that building a RAG pipeline upon a sea of unstructured business documents is easy to setup and configure using ADW’s Select-AI-RAG. And that doing such then makes that knowledge base query-able by the community of ADW users. And that building a tailored application upon that RAG knowledge base is straightforward using the low-code APEX development tool.

*Suggested Next Steps*

Reading this far suggests an interest in Select-AI-RAG for your Oracle database. So watch this [video](https://github.com/oracle-nace-dsai/rag-23ai-demo-archive/blob/main/video/select_ai_rag_27min.mp4) walkthrough of this experiment and then rebuild that content in your OCI tenancy by following the instructions and codes that are archived [here](https://github.com/oracle-nace-dsai/rag-23ai-demo-archive).

*Additional information*

* [Retrieval Augmented Generation (RAG)](https://en.wikipedia.org/wiki/Retrieval-augmented_generation)
* [Autonomous Data Warehouse](https://www.oracle.com/autonomous-database/autonomous-data-warehouse/)
* [APEX](https://www.oracle.com/application-development/apex/)
* [OCI Object Storage bucket](https://www.oracle.com/cloud/storage/object-storage/)
* [embedding model](https://www.pinecone.io/learn/vector-embeddings/)
* [Large Language Model (LLM)](https://en.wikipedia.org/wiki/Large_language_model)
* [GPU](https://en.wikipedia.org/wiki/Graphics_processing_unit)
* [Project Gutenberg](https://www.gutenberg.org/)
* [Select-AI-RAG](https://blogs.oracle.com/database/post/announcing-select-ai-with-rag-on-adb)
* [Oracle Machine Learning (OML) notebook](https://docs.oracle.com/en/database/oracle/machine-learning/oml-notebooks/)
* [OCI cloud shell](https://docs.oracle.com/en-us/iaas/Content/API/Concepts/cloudshellintro.htm)
* [DBMS\_CLOUD\_AI Package](https://docs.oracle.com/en-us/iaas/autonomous-database-serverless/doc/dbms-cloud-ai-package.html#ADBSB-GUID-000CBBD4-202B-4E9B-9FC2-B9F2FF20F246)
* [vector search in Oracle database](https://docs.oracle.com/en/database/oracle/oracle-database/23/nfcoa/ai_vector_search.html)
* [OCI GenAI service](https://docs.oracle.com/en-us/iaas/Content/generative-ai/overview.htm)