Code Generation with GraphRAG

Introduction

In this notebook, we demonstrate that **GraphRAG significantly outperforms standard vector-based retrieval** for generating working code from documentation. While traditional vector search retrieves relevant snippets, it often lacks the structured understanding needed to produce executable results. In contrast, **GraphRAG enables the LLM to follow logical relationships within documentation, leading to functional code generation**.

We achieve this by leveraging a custom traversal strategy, selecting nodes that contain both **code examples and descriptive text**, allowing the LLM to assemble more complete responses.

Getting Started

Below we will experiment with the AstraPy documentation to evaluate how well GraphRAG can generate working code.

Using AstraDB as the vector store, we compare GraphRAG's structured retrieval with standard vector search to solve a specific coding task. The query we will be sending to the LLM is the following:

In [ ]:
```
query = """
Generate a function for connecting to an AstraDB cluster
using the AstraPy library,
and retrieve some rows from a collection. The number of
rows to return should be a
parameter on the method. Use Token Authentication.
Assume the cluster is hosted on
AstraDB. Include the necessary imports and any other
necessary setup. The following
environment variables are available for your use:

- `ASTRA_DB_API_ENDPOINT`: The Astra DB API endpoint.
- `ASTRA_DB_APPLICATION_TOKEN`: The Astra DB Application
token.
- `ASTRA_DB_KEYSPACE`: The Astra DB keyspace.
- `ASTRA_DB_COLLECTION`: The Astra DB collection." \
"""
```

The following block will configure the environment from the Colab Secrets. To run it, you should have the following Colab Secrets defined and accessible to this notebook:

- `OPENAI_API_KEY` : The OpenAI key.
- `ASTRA_DB_API_ENDPOINT` : The Astra DB API endpoint.
- `ASTRA_DB_APPLICATION_TOKEN` : The Astra DB Application token.
- `LANGCHAIN_API_KEY` : Optional. If defined, will enable LangSmith tracing.
- `ASTRA_DB_KEYSPACE` : Optional. If defined, will specify the Astra DB keyspace. If not defined, will use the default.

If you don't yet have access to an AstraDB database, or need to check your credentials, see the help here.

In [ ]:
```
# Install modules.

%pip install \
    langchain-core \
    langchain-astradb \
    langchain-openai \
    langchain-graph-retriever \
    graph-rag-example-helpers
```

The last package -- `graph-rag-example-helpers` -- includes the helpers and example documents that we will use in this notebook.

In [ ]:
```
# Configure import paths.
import os
import sys

from langchain_core.documents import Document

sys.path.append("../../")

# Initialize environment variables.
from graph_rag_example_helpers.env import Environment,
initialize_environment

initialize_environment(Environment.ASTRAPY)

os.environ["LANGCHAIN_PROJECT"] = "code-generation"
os.environ["ASTRA_DB_COLLECTION"] = "code_generation"


def print_doc_ids(docs: list[Document]):
    [print(f"`{doc.id}` has example: {'example' in
doc.metadata}") for doc in docs]
```

Part 1: Loading Data

First, we'll demonstrate how to load the example AstraPy documentation into `AstraDBVectorStore`. We will be creating a LangChain Document for every module, class, attribute, and function in the package.

We will use the pydoc description field for the `page_content` field in the document. Note that not every item in the package has a description. Because of this, there will be many documents that have no page content.

Besides the description, we will also include a bunch of extra information related to the item in the `metadata` field. This info can include the item's name, kind, parameters, return type, base class, etc.

The item's `id` will be the items path in the package.

Below are two example documents... One with page content and one without.

Example doc with page content

Click to expand

```
id: astrapy.client.DataAPIClient

page_content: |
  A client for using the Data API. This is the main entry point
and sits
  at the top of the conceptual "client -> database ->
collection" hierarchy.

  A client is created first, optionally passing it a suitable
Access Token.
  Starting from the client, then:
    - databases (Database and AsyncDatabase) are created for
working with data
    - AstraDBAdmin objects can be created for admin-level work

metadata:
  name: DataAPIClient
  kind: class
  path: astrapy.client.DataAPIClient
  parameters:
    token: |
      str | TokenProvider | None = None
      an Access Token to the database. Example:
`"AstraCS:xyz..."`.
      This can be either a literal token string or a subclass of
      `astrapy.authentication.TokenProvider`.

    environment: |
      str | None = None
      a string representing the target Data API environment.
      It can be left unspecified for the default value of
`Environment.PROD`;
      other values include `Environment.OTHER`,
`Environment.DSE`.

    callers: |
      Sequence[CallerType] = []
      a list of caller identities, i.e. applications, or
frameworks,
      on behalf of which Data API and DevOps API calls are
performed.
      These end up in the request user-agent.
      Each caller identity is a ("caller_name",
"caller_version") pair.
```

```
  example: |
    >>> from astrapy import DataAPIClient
    >>> my_client = DataAPIClient("AstraCS:...")
    >>> my_db0 = my_client.get_database(
    ...     "https://01234567-....apps.astra.datastax.com"
    ... )
    >>> my_coll = my_db0.create_collection("movies",
dimension=2)
    >>> my_coll.insert_one({"title": "The Title", "$vector":
[0.1, 0.3]})
    >>> my_db1 = my_client.get_database("01234567-...")
    >>> my_db2 = my_client.get_database("01234567-...",
region="us-east1")
    >>> my_adm0 = my_client.get_admin()
    >>> my_adm1 =
my_client.get_admin(token=more_powerful_token_override)
    >>> database_list = my_adm0.list_databases()

  references:
    astrapy.client.DataAPIClient

  gathered_types:
    astrapy.constants.CallerType
    astrapy.authentication.TokenProvider
```

This is the documentation for `astrapy.client.DataAPIClient` class. The `page_content` field contains the description of the class, and the `metadata` field contains the rest of the details, including example code of how to use the class.

The `references` metadata field contains the list of related items used in the example code block. The `gathered_types` field contains the list of types from the parameters section. In GraphRAG, we can use these fields to link to other documents.

Example doc without page content

Click to expand

```
id: astrapy.admin.AstraDBAdmin.callers

page_content: ""

metadata:
  name: callers
  path: astrapy.admin.AstraDBAdmin.callers
  kind: attribute
```

This is the documentation for `astrapy.admin.AstraDBAdmin.callers`. The `page_content` field is empty, and the `metadata` field contains the details.

Despite having no page content, this document can still be useful for Graph RAG. We'll add a `parent` field to the metadata at vectorstore insertion time to link it to the parent document: `astrapy.admin.AstraDBAdmin`, and we can use this for traversal.

Create the AstraDBVectorStore

Next, we'll create the Vector Store we're going to load these documents into. In our case, we'll use DataStax Astra DB with Open AI embeddings.

In [ ]:
```
from langchain_astradb import AstraDBVectorStore
from langchain_openai import OpenAIEmbeddings

store = AstraDBVectorStore(
    embedding=OpenAIEmbeddings(),
    collection_name=os.getenv("ASTRA_DB_COLLECTION"),
)
```

Loading Data

Now its time to load the data into our Vector Store. We'll use a helper method to download already prepared documents from the `graph-rag-example-helpers` package. If you want to see how these documents were created from the AstraPy package, see details in the Appendix.

We will use the `ParentTransformer` to add a parent field to the metadata document field. This will allow us to traverse the graph from a child to its parent.

In [ ]:
```
from graph_rag_example_helpers.datasets.astrapy import
fetch_documents
from langchain_graph_retriever.transformers import
ParentTransformer

transformer = ParentTransformer(path_delimiter=".")
doc_ids =
store.add_documents(transformer.transform_documents(fetc
h_documents()))
```

We can retrieve a sample document to check if the parent field was added correctly:

In [ ]:
```
from graph_rag_example_helpers.examples.code_generation
import format_document

print(
    format_document(

store.get_by_document_id("astrapy.admin.AstraDBAdmin.cal
lers"), debug=True
    )
)
```

```
callers (attribute)

path:
        astrapy.admin.AstraDBAdmin.callers

callers = callers_param

parent: astrapy.admin.AstraDBAdmin
```

At this point, we've created a Vector Store with all the documents from the AstraPy documentation. Each document contains metadata about the module, class, attribute, or function, and the page content contains the description of the item.

In the next section we'll see how to build relationships from the metadata in order to traverse through the documentation in a similar way to how a human would.

Part 2: Graph Traversal

The GraphRAG library allows us to traverse through the documents in the Vector Store. By changing the `Strategy`, we can control how the traversal is performed.

Basic Traversal

We'll start with the default `Eager` strategy, which will traverse the graph in a breadth-first manner. In order to do this we need to set up the relationships between the documents. This is done by defining the "edges" between the documents.

In our case we will connect the "references", "gathered_types", "parent", "implemented_by", and "bases" fields in the metadata to the "id" field of the document they reference.

In [ ]:
```
edges = [
    ("gathered_types", "$id"),
    ("references", "$id"),
    ("parent", "$id"),
    ("implemented_by", "$id"),
    ("bases", "$id"),
]
```

Note that edges are directional, and indicate metadata fields by default. The magic string `$id` is used to indicate the document's id.

In the above `edges` list, any document id found in `gathered_types` will be connected to documents with the corresponding id. The other edges will work in a similar way.

Lets use these edges to create a LangChain retriever and documents for our query.

In [ ]:
```
from langchain_graph_retriever import GraphRetriever

default_retriever = GraphRetriever(store=store,
edges=edges)

print_doc_ids(default_retriever.invoke(query,
select_k=6, start_k=3, max_depth=2))
```

```
`astrapy.core.db.AsyncAstraDB.collection` has example: Fals
e
`astrapy.core.db.AstraDB.collection` has example: False
`astrapy.admin.DataAPIDatabaseAdmin.list_keyspaces` has exa
mple: True
`astrapy.admin.DataAPIDatabaseAdmin` has example: True
`astrapy.core.db.AsyncAstraDB` has example: False
`astrapy.core.db.AstraDBCollection` has example: False
```

Notes on the extra keyword args:

- `select_k` in GraphRAG is equivalent to `k` in LangChain. It specifies the number of nodes to select during retrieval.
- `start_k` indicates the number of nodes to select using standard vector retrieval before moving onto graph traversal.
- `max_depth` is the maximum depth to traverse in the graph.

With this configuration, we were only able to find 2 documents with example code.

Custom Strategy

Now we will create a custom strategy that will traverse a larger portion of the graph and return the documents that contain code examples or descriptive text.

To do this, we need to implement a class that inherits from the base `Strategy` class and overrides `iteration` method:

In [ ]:
```
import dataclasses
from collections.abc import Iterable

from graph_retriever.strategies import NodeTracker,
Strategy
from graph_retriever.types import Node


@dataclasses.dataclass
class CodeExamples(Strategy):
    # internal dictionary to store all nodes found during
the traversal
    _nodes: dict[str, Node] =
dataclasses.field(default_factory=dict)

    def iteration(self, *, nodes: Iterable[Node],
tracker: NodeTracker) -> None:
        # save all newly found nodes to the internal node
dictionary for later use
        self._nodes.update({n.id: n for n in nodes})
        # traverse the newly found nodes
        new_count = tracker.traverse(nodes=nodes)

        # if no new nodes were found, we have reached the
end of the traversal
        if new_count == 0:
            example_nodes = []
            description_nodes = []

            # iterate over all nodes and separate nodes
with examples from nodes with
            # descriptions
            for node in self._nodes.values():
                if "example" in node.metadata:
                    example_nodes.append(node)
                elif node.content != "":
                    description_nodes.append(node)

            # select the nodes with examples first and
descriptions second
            # note: the base `finalize_nodes` method
will truncate the list to the
            #   `select_k` number of nodes
            tracker.select(example_nodes)
            tracker.select(description_nodes)
```

As described in the comments above, this custom strategy will first try to select documents that contain code examples, and then will use documents that contain descriptive text.

We can now use this custom strategy to build a custom retriever, and ask the query again:

```
In [ ]: custom_retriever = GraphRetriever(store=store,
        edges=edges, strategy=CodeExamples())

        print_doc_ids(custom_retriever.invoke(query, select_k=6,
        start_k=3, max_depth=2))
```

```
`astrapy.admin.DataAPIDatabaseAdmin.list_keyspaces` has exa
mple: True
`astrapy.admin.DataAPIDatabaseAdmin` has example: True
`astrapy.client.DataAPIClient` has example: True
`astrapy.database.AsyncDatabase` has example: True
`astrapy.database.Database` has example: True
`astrapy.authentication.UsernamePasswordTokenProvider` has
example: True
```

Now we have found 6 documents with code examples! That is a significant improvement over the default strategy.

Step 3: Using GraphRAG to Generate Code

We now use the `CodeExamples` strategy inside a Langchain pipeline to generate code snippets.

We will also use a custom document formatter, which will format the document in a way that makes it look like standard documentation. In particular, it will format all the extra details stored in the metadata in a way that is easy to read. This will help the LLM use the information in the documents to generate code.

In [ ]:

```
from graph_rag_example_helpers.examples.code_generation
import format_docs
from langchain.chat_models import init_chat_model
from langchain_core.output_parsers import
StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough

llm = init_chat_model("gpt-4o-mini",
model_provider="openai")

prompt = ChatPromptTemplate.from_template(
    """Generate a block of runnable python code using
the following documentation as
    guidance. Return only the code. Don't include any
example usage.

    Each documentation page is separated by three dashes
(---) on its own line.
    If certain pages of the provided documentation
aren't useful for answering the
    question, feel free to ignore them.

    Question: {question}

    Related Documentation:

    {context}
    """
)

graph_chain = (
    {"context": custom_retriever | format_docs,
"question": RunnablePassthrough()}
    | prompt
    | llm
    | StrOutputParser()
)

print(graph_chain.invoke(query))
```

~~~
```python
import os
from astrapy.client import DataAPIClient
from astrapy.collection import Collection

def connect_and_retrieve_rows(num_rows):
    api_endpoint = os.getenv('ASTRA_DB_API_ENDPOINT')
    application_token = os.getenv('ASTRA_DB_APPLICATION_TOK
EN')
~~~

```
    keyspace = os.getenv('ASTRA_DB_KEYSPACE')
    collection_name = os.getenv('ASTRA_DB_COLLECTION')

    client = DataAPIClient(token=application_token)
    database = client.get_database(api_endpoint)
    collection = Collection(database=database, name=collect
ion_name, keyspace=keyspace)

    rows = collection.find(limit=num_rows)
    return list(rows)
```
```

We can try running this generated code to see if it works:

In [ ]:
```
import os

from astrapy.client import DataAPIClient
from astrapy.collection import Collection

def connect_and_retrieve_rows(num_rows):
    api_endpoint = os.getenv("ASTRA_DB_API_ENDPOINT")
    application_token =
os.getenv("ASTRA_DB_APPLICATION_TOKEN")
    keyspace = os.getenv("ASTRA_DB_KEYSPACE")
    collection_name = os.getenv("ASTRA_DB_COLLECTION")

    client = DataAPIClient(token=application_token)
    database = client.get_database(api_endpoint)
    collection = Collection(database=database,
name=collection_name, keyspace=keyspace)

    rows = collection.find(limit=num_rows)
    return list(rows)
```

In [ ]:
```
for row in connect_and_retrieve_rows(5):
    print(row)
```

```
{'_id': 'astrapy.info.EmbeddingProviderAuthentication', 'co
ntent': 'A representation of an authentication mode for usi
ng an embedding model,\nmodeling the corresponding part of
the response returned by the\n\'findEmbeddingProviders\' Da
ta API endpoint (namely "supportedAuthentication").', 'meta
data': {'kind': 'class', 'name': 'EmbeddingProviderAuthenti
cation', 'path': 'astrapy.info.EmbeddingProviderAuthenticat
ion', 'parameters': [{'name': 'enabled', 'type': 'bool'},
{'name': 'tokens', 'type': 'list[EmbeddingProviderToke
n]'}], 'attributes': [{'name': 'enabled', 'type': 'bool', '
description': 'whether this authentication mode is availabl
e for a given model.'}, {'name': 'tokens', 'type': 'list[Em
```
```

```
beddingProviderToken]', 'description': 'a list of `Embeddin
gProviderToken` objects,\ndetailing the secrets required fo
r the authentication mode.'}], 'gathered_types': ['Embeddin
gProviderToken'], 'parent': 'astrapy.info'}}
{'_id': 'astrapy.defaults.DEV_OPS_RESPONSE_HTTP_CREATED', '
content': '', 'metadata': {'kind': 'attribute', 'name': 'DE
V_OPS_RESPONSE_HTTP_CREATED', 'path': 'astrapy.defaults.DEV
_OPS_RESPONSE_HTTP_CREATED', 'value': 'DEV_OPS_RESPONSE_HTT
P_CREATED = 201', 'parent': 'astrapy.defaults'}}
{'_id': 'astrapy.info.CollectionInfo.full_name', 'content':
'', 'metadata': {'kind': 'attribute', 'name': 'full_name',
'path': 'astrapy.info.CollectionInfo.full_name', 'value': '
full_name: str', 'parent': 'astrapy.info.CollectionInfo'}}
{'_id': 'astrapy.collection.Collection.full_name', 'content
': 'The fully-qualified collection name within the databas
e,\nin the form "keyspace.collection_name".', 'metadata':
{'kind': 'attribute', 'name': 'full_name', 'path': 'astrapy
.collection.Collection.full_name', 'value': 'full_name: str
', 'example': ">>> my_coll.full_name\n'default_keyspace.my_
v_collection'", 'parent': 'astrapy.collection.Collection'}}
{'_id': 'astrapy.exceptions.DataAPIErrorDescriptor', 'conte
nt': 'An object representing a single error returned from t
he Data API,\ntypically with an error code and a text messa
ge.\nAn API request would return with an HTTP 200 success e
rror code,\nbut contain a nonzero amount of these.\n\nA sin
gle response from the Data API may return zero, one or more
of these.\nMoreover, some operations, such as an insert_man
y, may partally succeed\nyet return these errors about the
rest of the operation (such as,\nsome of the input document
s could not be inserted).', 'metadata': {'kind': 'class', '
name': 'DataAPIErrorDescriptor', 'path': 'astrapy.exception
s.DataAPIErrorDescriptor', 'parameters': [{'name': 'error_d
ict', 'type': 'dict[str, str]'}], 'attributes': [{'name': '
error_code', 'type': 'str | None', 'description': 'a string
code as found in the API "error" item.'}, {'name': 'message
', 'type': 'str | None', 'description': 'the text found in
the API "error" item.'}, {'name': 'attributes', 'type': 'di
ct[str, Any]', 'description': 'a dict with any further key-
value pairs returned by the API.'}], 'parent': 'astrapy.exc
eptions'}}
```

Conclusion

The results clearly demonstrate that **GraphRAG leads to functional code generation, while standard vector-based retrieval fails**.

In contrast, attempts using **only an LLM** or **standard vector-based RAG** resulted in **incomplete or non-functional outputs**. The appendix includes examples illustrating these limitations.

By structuring document relationships effectively, **GraphRAG improves retrieval quality, enabling more reliable LLM-assisted code generation**.

Appendix

LLM Alone

Here we show how to use the LLM alone to generate code for the query. We will use the same query as before, but modify the prompt to not include any context.

In [ ]:
```
llm_only_prompt = ChatPromptTemplate.from_template(
    """Generate a block of runnable python code. Return only the code.
    Don't include any example usage.

    Question: {question}
    """
)

llm_only_chain = (
    {"question": RunnablePassthrough()} | llm_only_prompt | llm | StrOutputParser()
)

print(llm_only_chain.invoke(query))
```

~~~
```python
import os
from astra import AstraClient

def fetch_rows_from_astra_db(num_rows):
    # Retrieve environment variables
    api_endpoint = os.getenv("ASTRA_DB_API_ENDPOINT")
    application_token = os.getenv("ASTRA_DB_APPLICATION_TOK
~~~

```
EN")
    keyspace = os.getenv("ASTRA_DB_KEYSPACE")
    collection = os.getenv("ASTRA_DB_COLLECTION")

    # Initialize the Astra DB client
    client = AstraClient(api_endpoint, application_token)

    # Retrieve rows from the specified collection
    query = f'SELECT * FROM {keyspace}.{collection} LIMIT
{num_rows}'
    response = client.execute_statement(query)

    # Return the rows retrieved
    return response['rows']
```
```

This code is not functional. The package `astra` and the class `AstraClient` do not exist.

## Standard RAG

Here we show how to use the LLM with standard RAG to generate code for the query. We will use the same query and prompt as we did with GraphRAG.

In [ ]:
```
rag_chain = (
    {
        "context": store.as_retriever(k=6) |
format_docs,
        "question": RunnablePassthrough(),
    }
    | prompt
    | llm
    | StrOutputParser()
)

print(rag_chain.invoke(query))
```

````
```python
import os
from astra import AstraClient

def fetch_rows_from_astradb(num_rows):
    endpoint = os.getenv('ASTRA_DB_API_ENDPOINT')
    token = os.getenv('ASTRA_DB_APPLICATION_TOKEN')
    keyspace = os.getenv('ASTRA_DB_KEYSPACE')
    collection = os.getenv('ASTRA_DB_COLLECTION')

    client = AstraClient(
        endpoint=endpoint,
````

~~~
        token=token
    )

    query = f'SELECT * FROM {keyspace}.{collection} LIMIT
{num_rows}'
    response = client.execute(query)
    return response['data']
```
~~~

This code is also not functional.

Converting AstraPy Documentation

The AstraPy documentation was converted into a JSONL format via some custom code that is not included in this notebook. However, the code is available in the `graph-rag-example-helpers` package here.