PharmaSUG 2026 - Paper ET-399

Chatting with Your Data, Wherever It Lives: Unlock Insights through Duck DB and Open File Formats

Sundaresh Sankaran, SAS Institute
Mary Dolegowski, SAS Institute

ABSTRACT

Enterprises have shifted their approach to accessing data for analysis and insights. They require rapid insights preferably with minimal data movement. There's greater adoption of open formats like Parquet that reduce data footprint and enable efficient retrieval, and increased inclination to use a multitude of formats like JSON, CSV and Parquet.

As part of recent technology trends, we witness the emergence of Duck DB, an open-source, lightweight query processing engine optimised for analytical workloads. Duck DB should be viewed as a multipurpose query processing engine rather than a traditional enterprise database and can analyse a range of data formats residing in several source areas, making use of native readers without having to copy data. Analytical platforms and packages such as SAS, R and Python offer Duck DB drivers and related capabilities.

In this session, we demonstrate how to use Duck DB to query clinical study tables in a mixture of open file formats, highlighting features such as query optimisation, predicate pushdown, parallelism and parsing complex nested structures such as JSON. For clinical programmers, statisticians and data scientists, benefits include faster time to insights, simplicity and convenient interfaces while maintaining high accuracy. We also share an open-source repository which provides code snippets and queries based on examples from clinicaltrials.gov which can be extended further.

INTRODUCTION

Enterprises increasingly adopt open file formats. Motivations range from a need for better interoperability among systems, portability across environments, increased opportunities for collaboration, reduction of vendor lock-in and the inherent benefits such as reduced compression and rich metadata offered by some of these file formats (looking at you, Parquet). Also, let's acknowledge the increased focus on being audit-ready, whether from regulators or any other agency. When you use a common format, to the extent permissible, across laboratory systems, analytical tools, document repositories and submission processes, you gain benefits due to easier reproducibility of results, faster integration and easier reuse of data. You maintain consistent metadata and lineage which helps you address issues of review and compliance better.

FILE FORMATS

Let's now focus on some common formats. Parquet is a columnar file format which stores data as a list of values per column instead of per observation. Parquet also groups similar data types and saves them adjacently, using dictionaries in the process. Data observations are saved in segments called row groups, providing a further referential lookup. To top things off, efficient Parquet writers pre-calculate row group-level statistics per column and store them in metadata for easy access by Parquet readers in future. The result: you gain faster query performance and tighter compression resulting in lower storage footprint.

Let's also address JSON. JSON is a lightweight format with a simple and understandable structure. It's popular in web applications as a transport format (think of calls to Application Programming Interfaces (APIs) that require data to be sent to a server). Data involving hierarchical elements (example, a patient has single-level attributes and have other transactional attributes such as lab results and visit data that span multiple observations) aligns comfortably with JSON's nested structure. You benefit by being able to access a simple, readable structure that can operate across applications (e.g. reports and web pages) and analytical engines.

Finally, we must mention good ol' reliable CSV. CSV has been around for ages. Its inherent and simple text format makes it easy to process data. The row-level orientation makes audit easier and it's importable to and exportable from many applications, Microsoft Excel being a relatable example. This makes CSV (and its sister formats like tab-separated and pipe-delimited files etc.) a simple and relatable option.

STORAGE

Where you store your data is an open question. Open file formats expand the range of available storage options and enable you to optimize enterprise costs by considering flexible storage areas. An approach for enterprises has traditionally been to maintain data closer to the compute by saving them in shared storage mechanisms such as Lustre, NetApp and a host of other cloud-native solutions. However, not all data needs access all of the time. Thus, enterprises also use object storage such as AWS S3 buckets, Azure Blob storage, Azure Data Lake Storage and others. This storage option tends to be cheaper than shared storage disks but might imply data movement whenever you want to analyse data. Enterprises prefer avoiding or reducing data movement wherever possible. Your choice of storage location drives your decision on the right format and tool to adopt.

INTERFACE

An interface to data contained in an open file format deals with several aspects: – how you query your data, what's your comfort level with data analytics, and which tools do you use. The most common data analysis interface has traditionally been query mechanisms using Structured Query Language (SQL). Recent popularity of analytics has also increased curiosity among stakeholders who are not comfortable with query tools and prefer a more "conversational" way of interacting with data.

DUCK DB – THE QUERY PROCESSING ENGINE DRIVING THE INTERFACE

We present an example of how you can interact with open file formats in a conversational manner and Duck DB is the query processing engine driving this conversation behind the scenes. Duck DB is a lightweight open-source library which, its name notwithstanding, should not be viewed as a traditional database but rather as a query processing engine capable of interacting with a several source data systems. Included among these source data systems are open file formats which can be natively accessed from Duck DB with minimal data movement. Duck DB offers extensions to several sources, providing extensibility and interoperability among different storage patterns. Duck DB is especially efficient with Parquet files through its use of parallelism (without the need for horizontal scaling), predicate pushdown and query optimization.

THE SEMANTIC LAYER

Duck DB, for all its strengths, is still a query execution engine oriented towards data science and analytics practitioners. For more widespread access, it is important to build a conversational or semantic layer which uses AI assistance to help non-programming-oriented users frame queries in simple language. For this, we use SAS Retrieval Agent Manager (RAM) to develop an assistant that receives user input through a chatbot interface, retrieves context from knowledge sources, calls a Large Language Model (LLM) for purpose of code generation and assistance, and then interacts with the Duck DB engine through tools exposed in a Model Context Protocol (MCP) server. While similar tools and approaches can be built with open-source frameworks, we use RAM to demonstrate required modules in a unified and convenient manner. In effect, we build an AI assistant to interact with your data.

BUILDING THE ASSISTANT

Let us start with data first. Your choice of data depends on use cases (analysing clinical trial data for example) and factors unique to your enterprise. For purposes of demonstration, we use data conforming to the Study Data Tabulation Model (SDTM) as per CDISC standards, based originally on CDISC pilot data.

The data itself is not the focus of this paper, rather, the approach to query the data is. This data can be assumed to be accessible on a location which is accessible to our application. For simplicity's sake, we assume that it's available on a repository that's accessible within the network where the application resides.

We converted two datasets from the CDISC SDTM – DM (Demographics) and AE (Adverse Events) - to Parquet files from SAS datasets and saved them in a storage area. The choice of datasets **is purely for reasons of providing an example.** This paper does not concern itself with the actual contents and inferences from the original Pilot study (which itself was provided to test submissions and metadata) and makes use of only a couple of datasets to use as the basis of queries in the demo that follows.

Conversion from original formats (such as text files, CSVs or SAS datasets) is straightforward and was facilitated through the SAS/Access Interface to Duck DB, available in SAS Viya since version 2025.07 (July 2025).

Here is an example of how to write a Parquet file from a SAS dataset using a data step. Notice the use of implicit access mechanisms when we use the Duck DB interface.

```
/************************************************************

Write SAS datasets to Parquet

Input: AE and DM dataset

Source: Sundaresh Sankaran, Pritesh Desai, 1 Apr. 2026

Output:

    1. DM.parquet: DM.parquet - a Parquet file with CDISC pilot DM

    2. AE.parquet: AE.parquet - a Parquet file with AE from CDISC pilot

Sundaresh Sankaran (sundaresh.sankaran@sas.com)

01APR2026

************************************************************/;

/* Change libnames to your desired locations - source data*/

libname sdtm "/mnt/viya-share/data/et-399/data";


/* Change as required - Define libname based on SAS/Access Interface to Duck DB */

libname dukup duckdb file_type= parquet file_path = "/mnt/viya-share/data/et-399/data"
database=":memory:mydb";


/* Write dataset to Parquet through DATA step */

data dukup.DM (replace=yes);

    set sdtm.dm;

run;

data dukup.AE(replace=yes);

   set sdtm.AE;

run;
```

Others might prefer explicit SQL (additional options, pasting the code from other interfaces etc.). Here's how you write a Parquet file by passing commands through to Duck DB.

```
/* Define libname based on SAS/Access Interface to Duck DB */

libname dukup duckdb database=":memory:mydb";

/* Copy dataset to Duck DB */

data dukup.AE (replace=yes);

    set SDTM.AE;

run;


/* Copy files to Parquet */

proc sql;

    connect using dukup;

    execute(

    copy (select * from AE)

    TO "/mnt/viya-share/data/et-399/data/AE.parquet" (FORMAT parquet)

    ) by dukup;

quit;
```

Notice some subtle differences. In this case, Duck DB is used as an in-memory engine, which does not need to rest on a Parquet backend. This gives it flexibility. Also, when we say `(FORMAT parquet…)` we have an option to specify a host of additional options defining the parquet file in terms of row group metadata and partitioning information. The syntax for this is determined by the Duck DB SQL dialect, which makes the query (the block inside the `EXECUTE` parentheses) portable across engines that can run Duck DB SQL.

Now, let's start with our semantic layer. In Retrieval Agent Manager, we can define source information that contains any reference data that our semantic layer accesses. Here's the intriguing part: – our CDISC *data* files AREN'T the reference data we want. Using raw data as a direct access for AI's reference is an inefficient approach that carries costs, both in terms of computation and data leakage risk.

Rather, the idea would be to upload data that can *assist* AI in framing a query. A helpful way of thinking about designing semantic layers is that we think of AI assistants as humans. Just like you would provide a human assistant with clear instructions and context about the task to be accomplished, we offer our AI assistant context in terms of metadata. In this case, we use a data dictionary.

Domain	#	Variable	Type	Len	Label	Further Details
DM	1	STUDYID	Char	12	Study Identifier	The middle number in the USUBJID pattern
DM	2	DOMAIN	Char	2	Domain Abbreviation	DM = Demographics, AE = Adverse Events
DM	3	USUBJID	Char	11	Unique Subject Identifier	9 digit character pattern with study id in between, separated by an hyphen
DM	4	SUBJID	Char	4	Subject Identifier for the Study	Part of USUBJID
DM	5	RFSTDTC	Char	10	Subject Reference Start Date/Time	Reference date for the study
DM	6	RFENDTC	Char	10	Subject Reference End Date/Time	Study ends due to discontinuation, death, or natural termination, or any other reason

Figure 1: A sample of contents from the data dictionary used as reference data

In Retrieval Agent Manager, this data dictionary is added as a source file and configured in a collection. You run the collection through a vectorization job which makes the content easily searchable by other entities. The benefit of vectorizing only the data dictionary instead of the actual data is that now this **configuration is reusable across projects.**

AI assistants are good at generating text. But, left to themselves, even with context, they aren't the best calculators, especially when it comes to performing operations on local data or enterprise-specific data. Such processing turns out quite costly if attempted only by a Large Language Model (LLM), which is built for text generation more than mathematics. Also, LLM results are probabilistic, and prone to change for the same command when run multiple times. This doesn't help reproducibility. Moreover, it is not a good idea to send data to an LLM (even if the LLM were to be hosted in the same environment) because of the inherently opaque way LLMs work. You therefore require a **deterministic path** to complement the generative (and probabilistically derived) output from Large Language Models.

This deterministic path is provided by a **Model Context Protocol** server. Think of a mechanic bringing a toolbox and using appropriate tools as they repair a vehicle. Similarly, the MCP server hosts many different tools at different endpoints, each tool being a program or routine that performs specific operations. An MCP server operates much like a web server, but the key difference is that their endpoints are called by the AI assistant (using the help of an LLM), not by the user. The MCP server is therefore a declarative framework. You make the assistant aware that there are tools available for it to use and provide a description of the tools' capabilities, the LLM refers as it decides how to handle a query.

In our case, we stood up an MCP server using Retrieval Agent Manager and defined tools that used Duck DB SQL to manipulate and query DM and AE files from SDTM. For the sake of simplicity, we provide one basic example of what the tool looks like, the **get_nbr_subjects** tool. This tool is written is Python and looks like this.

```
@tool

def get_nbr_subjects() -> int:

    """Retrieves the total number of subjects from the DM domain."""

    nbr_records = con.execute("SELECT COUNT(*) as nbr_subjects FROM DM;").fetchone()[0]

    return nbr_records
```

This one simple query has many facets. Notice the `@tool` at the top. This informs the MCP server that what follows (a def, which is a way to define functions in Python) is a tool. The `: int` indicates that the tool returns an integer as a result. This is followed by a docstring. The docstring "Retrieves the total number of subjects from the DM domain." is **not** for cosmetic purposes. It informs the AI assistant (which calls the tool) about the capabilities of the selected tool. The clearer the docstring, the clearer the instructions to the AI assistant and greater the chance of success.

We define several such tools as shown in the diagrams below. These can either be specific to operations around the AE or DM tables, or even general-purpose tools that help in operations such as charting data or sending emails or filtering data. The advantage of RAM is that you can mix and match a variety of tools from different MCP servers for an assistant to choose from. Sometimes, you may wish to revoke access to a tool (for improvement or testing) and it's possible to do this through a no-code interface in RAM.

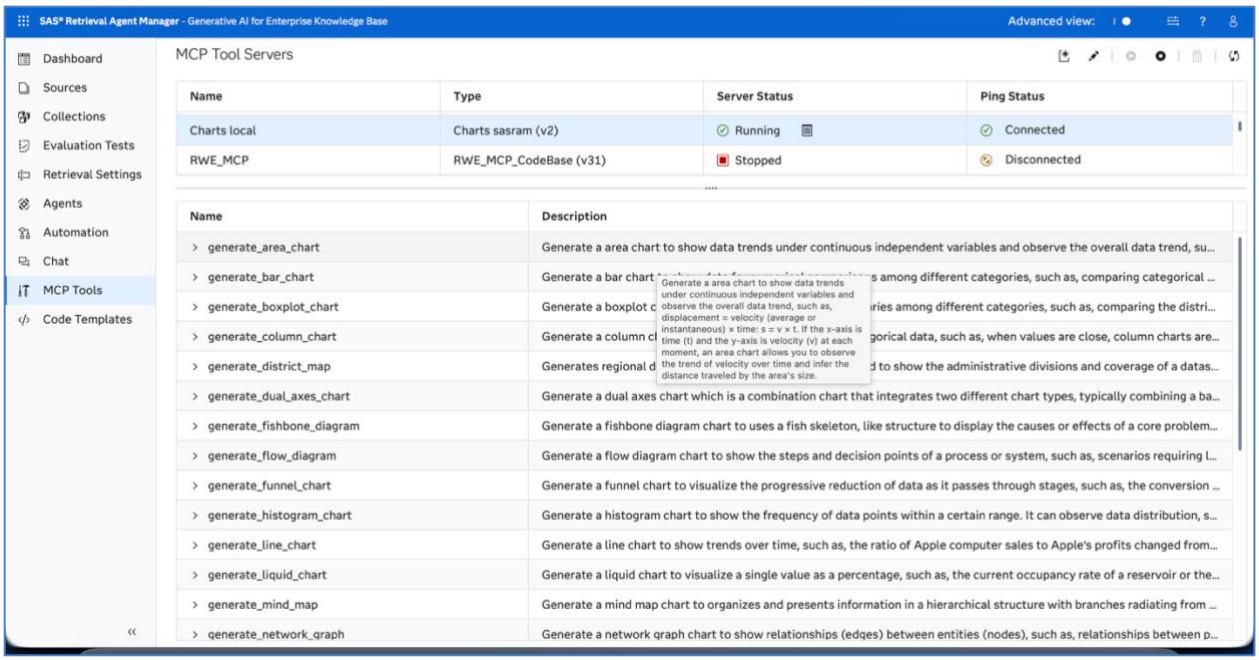

Figure 2: An example of the various general purposes charting tools available through MCP which can be accessed by multiple assigned AI assistants

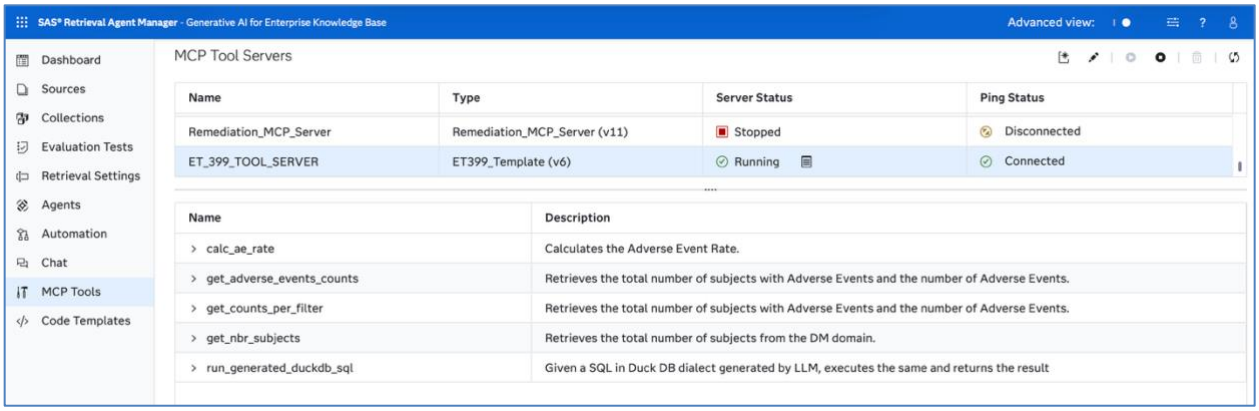

Figure 3: MCP tools for specific operations on AE and DM datasets, created for purpose of this paper.

The final step in establishing our semantic layer is to start an assistant (known technically as an Agent, but we use these terms interchangeably for simplicity's sake). This is a simple operation where we select an Agent which is prewired to make use of tools and uses pre-configured prompts and Large Language Models that it has been given access to.

The following diagrams show the range of possibilities available when configuring an assistant

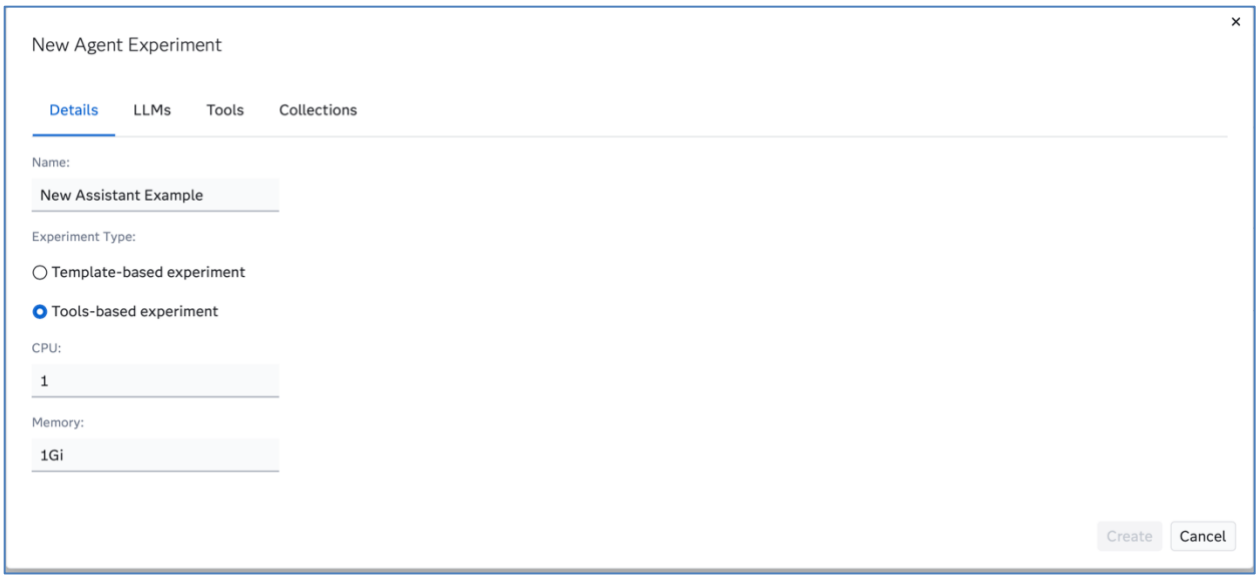

Figure 4: Choose from a Template based or Tools (no-code) assistant, and specify infrastructure (containerised) resources to run the assistant which can vary based on your assistant's use case

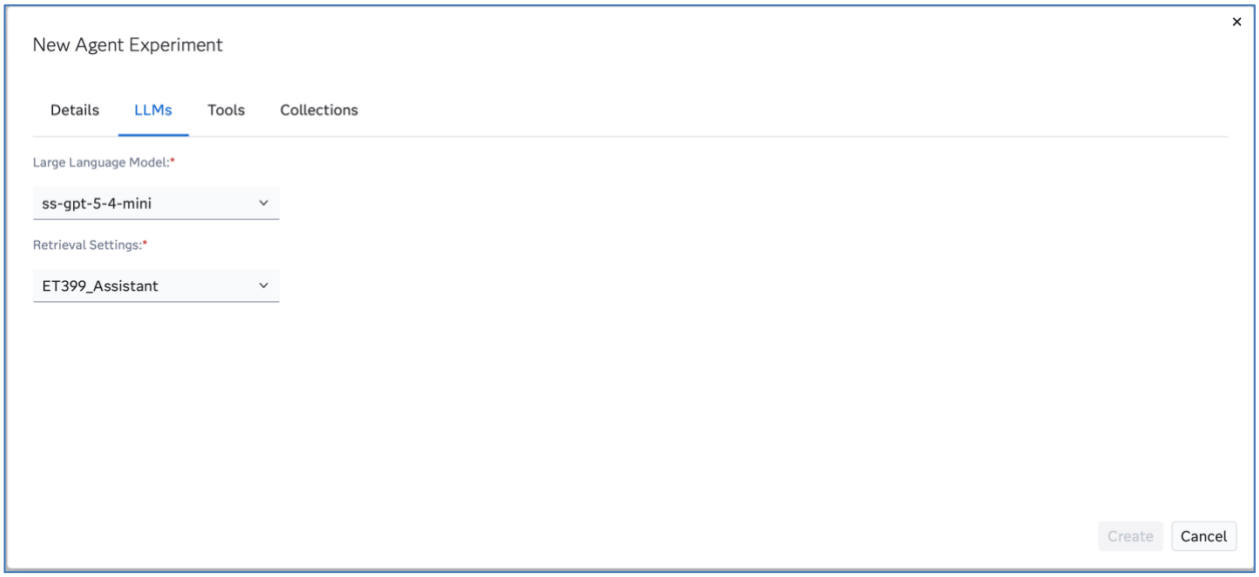

Figure 5: Configure the (pre-configured) LLM and Retrieval Settings (Prompt) that the assistant uses

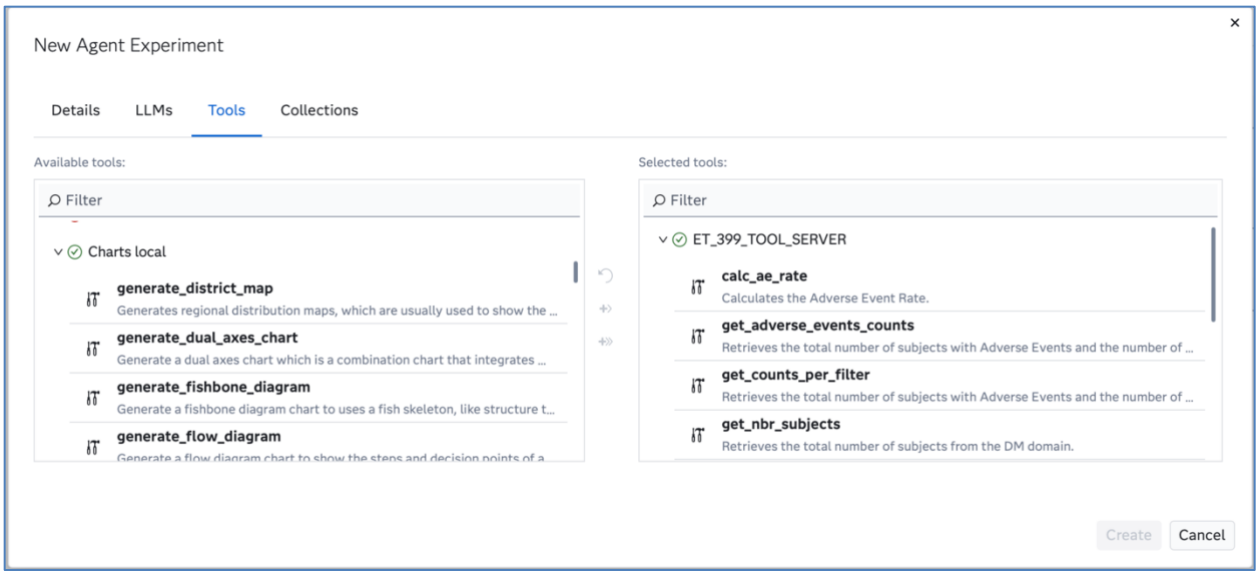

Figure 6: You (as the designer) decide which tools and capabilities you wish to allow your assistant access to

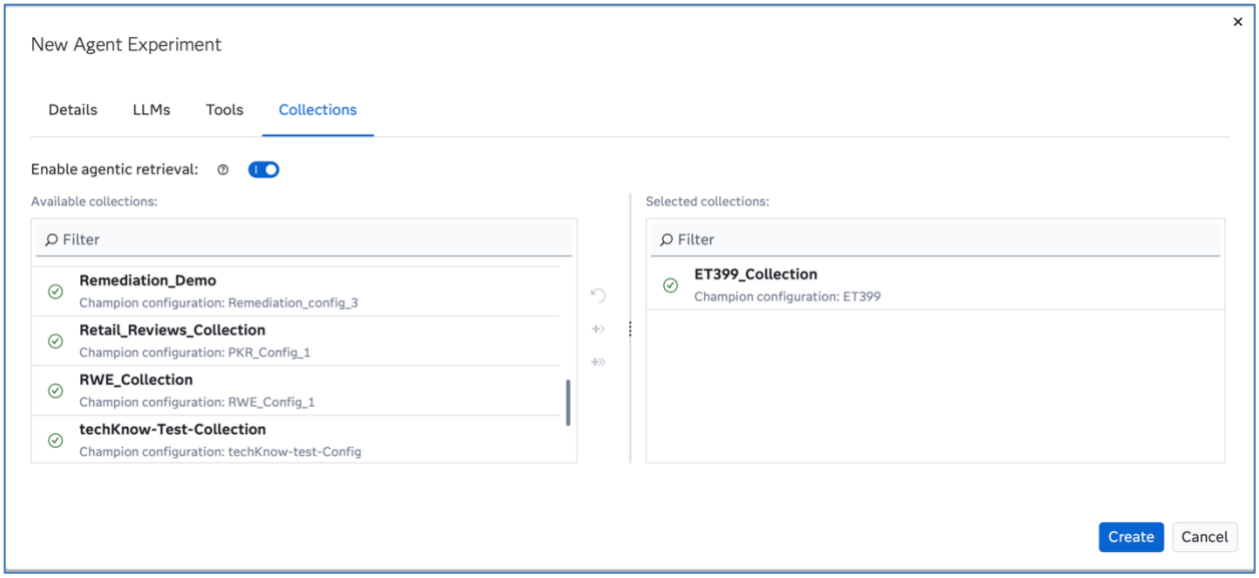

Figure 7: Specify the collection (the data dictionary for the tables in this project) that the assistant can use as reference data.

CHATTING WITH YOUR DATA – THE ACTUAL EXPERIENCE

Once you have set up your assistant, you can run it from an application of your choice. This assistant can help you with a range of queries through a chatbot interface as shown in the following diagram.

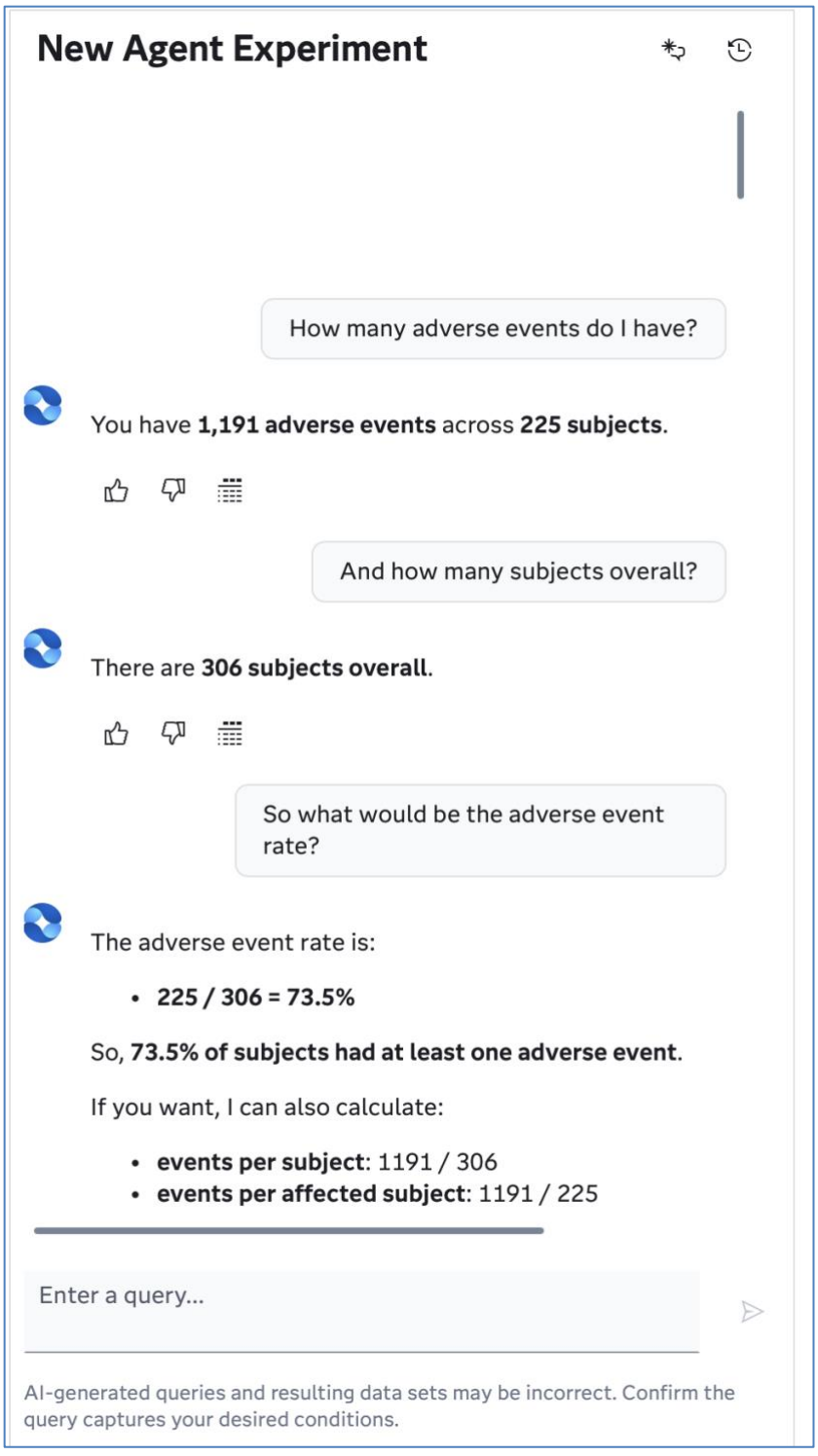

Figure 8: Example of a conversation with your assistant on SDTM data

Always heed the warning at the bottom of this screenshot. AI can make assumptions and hallucinate, therefore providing you incorrect, inaccurate or irrelevant responses. It's always important to carry out rigorous testing (enabled by automatic evaluation and user evaluation mechanisms in RAM) prior to rolling these agents out to production.

We notice that the assistant can query and analyse data across both example tables we considered. Let us dig into the details a little bit. Clicking on the details tab for each chat, you see a traceable flow (along with notional costs) of the LLM's interactions and operations before it gives you the final answer. Notice how, without explicit user direction, the LLM knows it needs to execute the **get_adverse_events_count** tool and does so, presenting its results in a more consumable format.

This is the power of MCP at work.

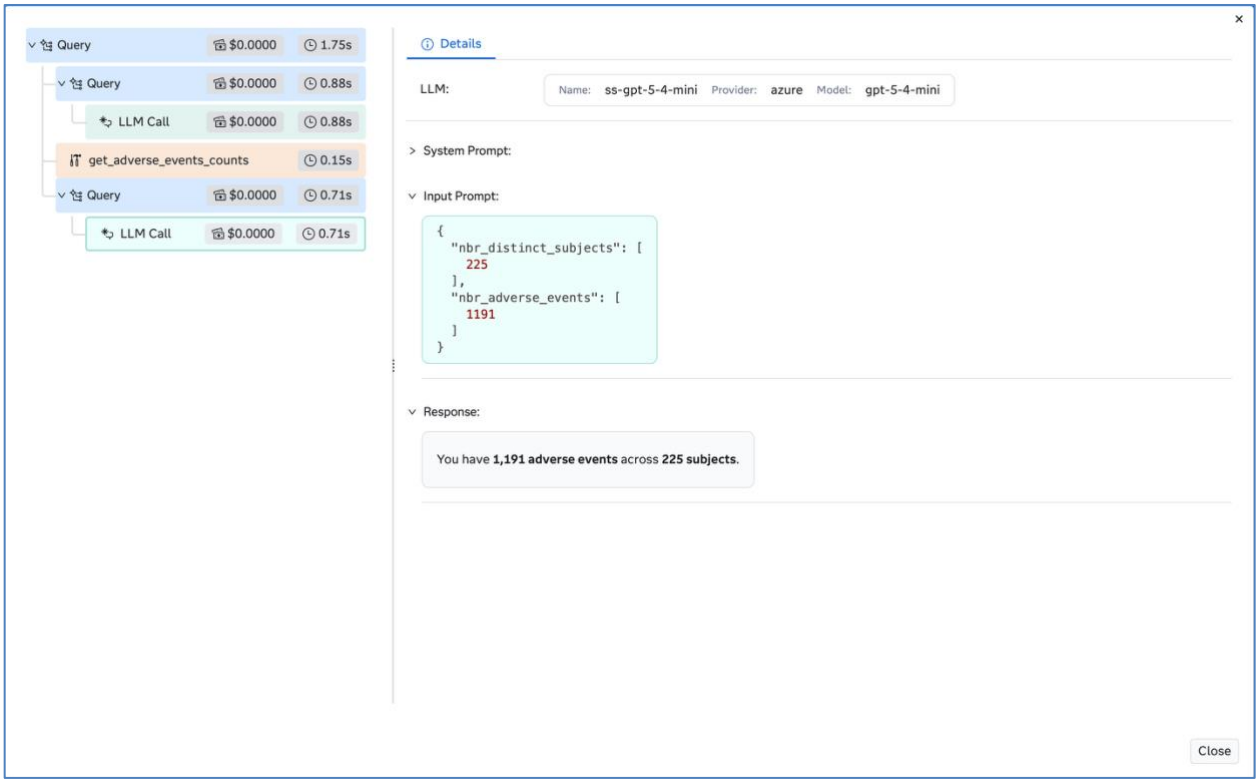

Figure 9: An example of how a tool is used dynamically to help answer a question.

A further aspect worth elaborating. What if the question was about a topic for which the assistant does not have recourse to a well-defined tool. That's where the LLM's capability for code generation comes in handy. This is accomplished and controlled through the following steps.

1. Firstly, define your prompt to have a fallback option. In Retrieval Agent Manager, we do this through the Retrieval Settings page. The basic structure of the prompt which instructs the Large Language Model is as follows:

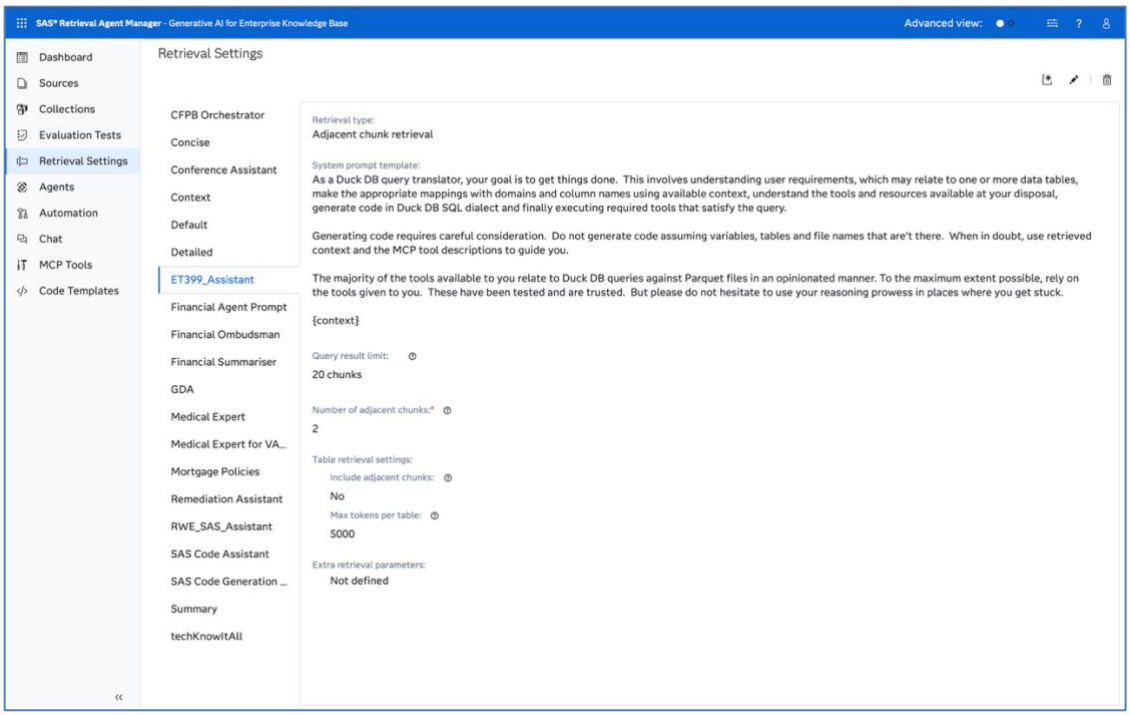

Figure 10: The Retrieval Settings page for the ET-399 (this paper) assistant

Notice how we instruct the assistant to use existing context and tools as much as possible but also follow a structured process by understanding the context, using Duck DB SQL dialect to generate code and then ensuring it can generate code standalone instead of depending only on existing tools.

2. Next, provide an execution route for generated code. This is done through an additional MCP tool, **run_generated_duckdb_sql,** which generates SQL code in Duck DB dialect after translating user requirements. Then it executes the same to test for any possible errors. Upon receiving no errors, the assistant then presents the answer to the user.

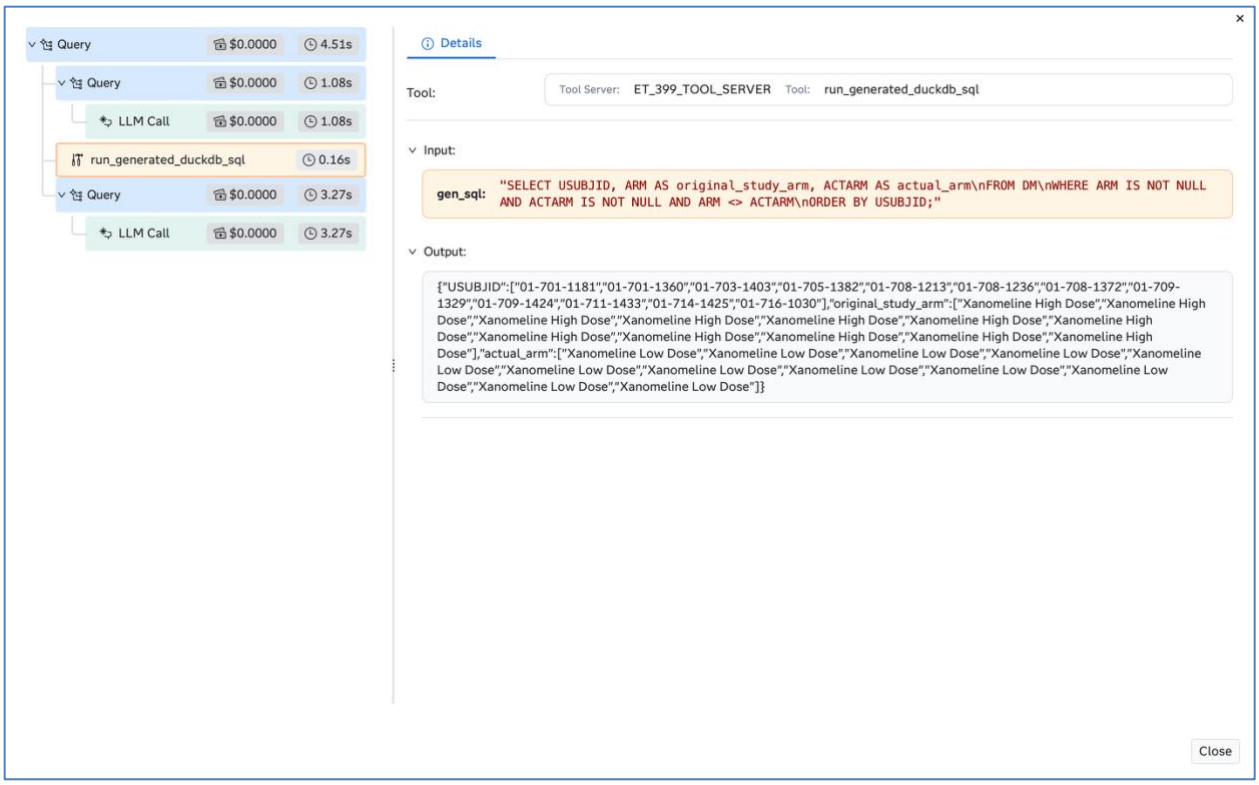

Figure 11: Example of the code generation tool called on demand for a new query. The SQL statement shown on the right side of the screenshot has been autogenerated based on knowledge retrieved from the data dictionary (the knowledge source).

Combining deterministic tools with generated “on-the-fly” code presents many interesting possibilities when scaling up this assistant. Frequently asked queries that involve code generation can be captured and saved separately, tagged as future candidates for deterministic tools. This ensures you do not have to unnecessarily spend LLM tokens in generating code for a common question in the future.

Other interesting applications also involve generating code on the fly to create a query and then further charting the same, as shown in the column chart below.

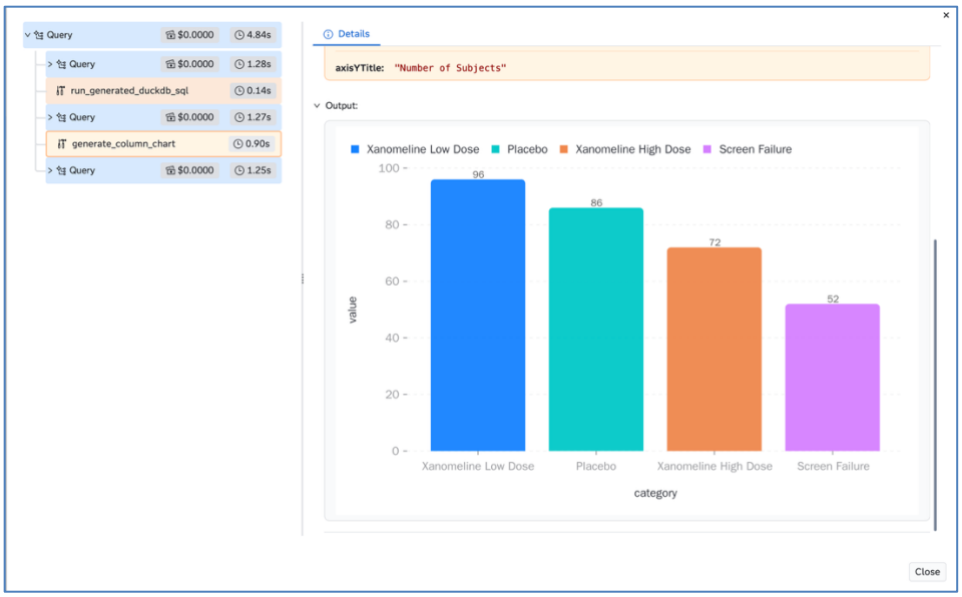

Figure 12: A column chart rendered from auto-generated SQL. On the left-hand side, notice how the LLM calls different tools.

PARQUET AND DUCK DB – HOW DO THEY WORK TOGETHER?

Let us now step back and consider the strengths of the underlying file format and the query engine. Duck DB and Parquet are suitable for complex queries involving large volumes of data, or multiple tables due to inherent benefits in both technologies.

Duck DB and Parquet have a symbiotic relationship. Parquet, being a columnar file format, offers benefits of compression and rich metadata. Every column also has precomputed statistics for various horizontal segments known as row groups, and this metadata serves as a useful first point of reference by query engines so that they can decide whether to skip or scan a set of observations. This is known as *zone map skipping.* Furthermore, when queries are executed, only relevant columns which form part of the query's selection criteria or filter criteria need to be read, a practice called *column pruning.*

Duck DB serves as the active query engine which natively accesses Parquet and takes full advantage of its columnar structure. It also performs predicate pushdowns, which refers to a practice followed by query processing engines to apply query filters in an intelligent manner at early stages so that fewer data observations get processed in subsequent operations. Another strength of Duck DB is its capabilities for making full use of CPUs in available resources, thus ensuring queries are executed quickly. Finally, there are conveniences in the query interface, such as Duck DB being able to easily read a glob of Parquet files using a wildcard pattern as shown below.

```
SELECT count(*)
   FROM
   read_parquet("/path/to/all/adverse/events/*.parquet");
```

CONCLUSION

TAKE A LOOK AND EXTEND FOR YOURSELF

Before we summarise, let's address a natural question which may have formed in your mind by now. Wouldn't you like to use some of these tools for yourself and extend them with additional tools? For this purpose, refer to this GitHub open-source repository which shares snippets of code describing the tools we created for this paper. We look to expand this in future with more descriptive examples in this area. These are represented as pseudocode so that you can adapt them for use in other LLM and MCP frameworks.

ENABLING AN INTERACTIVE EXPERIENCE THROUGH OPEN FILE FORMATS AND SEMANTIC LAYERS

We address two items through this paper. Open file formats provide you benefits of leaner storage footprints, rich metadata and interoperability across multiple systems that are used in life sciences. Duck DB offers a common, standardized layer which accesses multiple file formats and database systems, reduces the need for excessive data copies and data movement and delivers performant query execution through capabilities such as predicate pushdown, query optimization and parallelism. These features help you increase productivity by "scaling up" rather than needing to "scale out".

And finally, there's the question of interfaces. Efficient query mechanisms and file formats can only go so far in making analysis approachable to a wider audience. A semantic layer, exposed through an AI assistant, translates user questions into SQL queries, using Large Language Models (LLMs) and tools to deliver a more "conversational" experience.

THE CON-CONCLUSION: WE JUST REALIZED THIS!

Actually, our actions were quite deliberate. When discussing the semantic layer, we tried our best (though not comprehensively) to avoid references to the terminology and semantics around building Artificial Intelligence (AI)-based assistants. Especially the "A-word", i.e. Agents, and that part about "MCP being the USB-C connector for Agents" (heard that before?).

Ultimately, it's all about you having fun and developing a convenient tool. We want you to build your own "Chattin' with your data" projects without getting intimidated and hung up on semantics around what an Agent, MCP server, or any of those new-fangled terms really mean. We hope that by adopting such an attitude, you'll realize:

An Agent is What an Agent Does.

And what an Agent does is, ultimately, what matters.

REFERENCES

- The Apache Parquet project, Apache Foundation, https://parquet.apache.org
- Duck DB, Duck DB community, https://duckdb.org
- CDISC Data Standards, Study Data Tabulation Model (SDTM), https://www.cdisc.org/standards/foundational/sdtm
- Data Attributed to CDISC Pilot01 project (CDISC SDTM Adam Pilot project) , terms of use available here: https://github.com/cdisc-org/sdtm-adam-pilot-project/
- SAS/Access Interface to Duck DB, SAS Documentation, https://go.documentation.sas.com/doc/en/pgmsascdc/default/acwn/p1ozr0t2ly4bc2n0zxncjtshlyor.htm
- Sankaran, Sundaresh, "Parquet Partitioning in Duck DB: Some Points for Consideration", SAS Communities, https://communities.sas.com/t5/SAS-Communities-Library/Parquet-Partitioning-in-DuckDB-Some-Points-for-Consideration/ta-p/984341
- Sankaran, Sundaresh, "Performant Aggregations on Parquet using Duck DB", SAS Communities, https://communities.sas.com/t5/SAS-Communities-Library/Performant-Aggregations-on-Parquet-using-DuckDB/ta-p/981117
- ET-399: Chatting with your data, GitHub repository with examples, https://github.com/SundareshSankaran/et-399-chatting-with-your-data/tree/main

ACKNOWLEDGEMENT

We thank our colleague Pritesh Desai for his assistance.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Sundaresh Sankaran
SAS Institute
Phone : +1 919 400 3266
Sundaresh.sankaran@sas.com
https://www.linkedin.com/in/sundareshsankaran/

Mary Dolegowski
SAS Institute
Phone : +1 703 310 5511
Mary.Dolegowski@sas.com
https://www.linkedin.com/in/maryliang/