Advanced RAG Solution Accelerator

A new solution accelerator that supports advanced techniques for ingesting, formatting and intent extraction from structured and non-structured data and querying the data through simple web interface to achieve improved accuracy and performance rates than baseline RAG.

Contents

Overview

What is RAG and Why Advanced RAG?

Retrieval-Augmented Generation (RAG) is a natural language processing technique that combines the strengths of retrieval-based and generation-based models. It uses search algorithms to retrieve relevant data from external sources such as databases, knowledge bases, document corpora, and web pages. This retrieved data, known as "grounding information," is then input into a large language model (LLM) to generate more accurate, relevant, and up-to-date outputs.

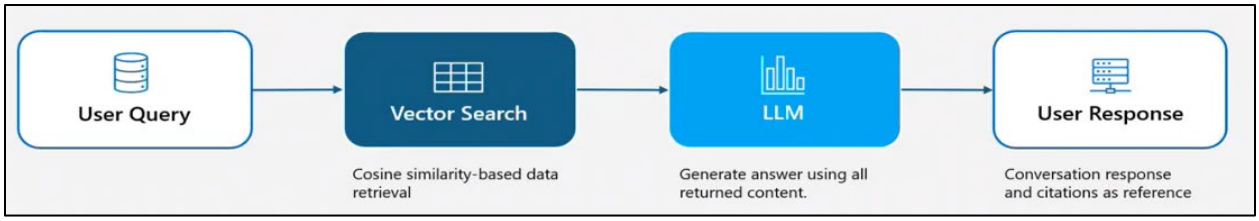


Figure1: High level Retrieval Augmented Flow

Usage Patterns

Here are some horizontal use cases where customers have used Retrieval Augmented Generation based systems:

1. **Conversational Search and Insights:** Summarize large volumes of information for easier consumption and communication.
2. **Content Generation:** Tailor interactions with individualized information to produce personalized output and recommendations.
3. **AI Assistant, Q&A, and Decisioning:** Analyze and interpret data to uncover patterns, identify trends, gain valuable insights, and answer questions.

Also, below are a few examples of vertical use cases where Retrieval Augmented Generation have been beneficial.

1. **Public Sector Knowledge Base:** A government agency needs a system to provide citizens with information about public services, such as permits, licenses, and local regulations.
2. **Compliance Document Retrieval:** A regulatory body must assist organizations in understanding compliance requirements through a database of guidelines and policies.
3. **Healthcare Patient Education:** A health department aims to provide patients with educational resources about common conditions and treatments.

Challenges with Baseline RAG:

1. **Ability to cover complex data:** RAG on plain text content seems to do fine. However, when the content becomes more complex like financial reports with images and complex tables and document sections spanning pages, being able to parse and index them isn't straightforward.
2. **Context Window Limitations:** As the dataset scales, the performance of RAG systems can degrade, particularly due to the "lost in the middle" phenomenon, making it challenging to retrieve specific information from large datasets.
3. **Search Limitations:** Though there have been advancements in Search technology to be able to perform vector-based searches, however searching over vector embedding alone may not be sufficient for achieving high accuracy.
4. **Groundedness:** When the search context is not enough, sometimes RAG systems can generate incorrect or misleading information that is not grounded in the customer's data. Careful evaluations may be necessary to catch these and fix them.
5. **Latency and User Experience:** Balancing performance and latency is crucial, as high latency can negatively impact the user experience. Optimizing this balance is a key challenge.
6. **Quality Improvement Levers:** Identifying and effectively utilizing the right levers for quality improvements, such as accuracy, latency, and relevance, can be difficult.

Advanced RAG aims to address the challenges faced with Baseline RAG by incorporating advanced techniques for ingestion, formatting, and intent extraction from both structured and unstructured data. It provides improved baseline architecture to build a more scalable solution that meets the accuracy and performance requirements of the business.

By implementing advanced methodologies in data ingestion, search optimization, and evaluation, Advanced RAG enhances the overall effectiveness and reliability of Retrieval-Augmented Generation systems. This ensures that the business value of RAG systems is maximized, aligning technological capabilities with business needs.

RAG Quality Improvement

Background

Our implementation uses default configurations from document indexing services to ingest financial data. We use Azure AI search for indexing it. The content was also vectorized in the index. The search index covered a few years of financial reports for the company.

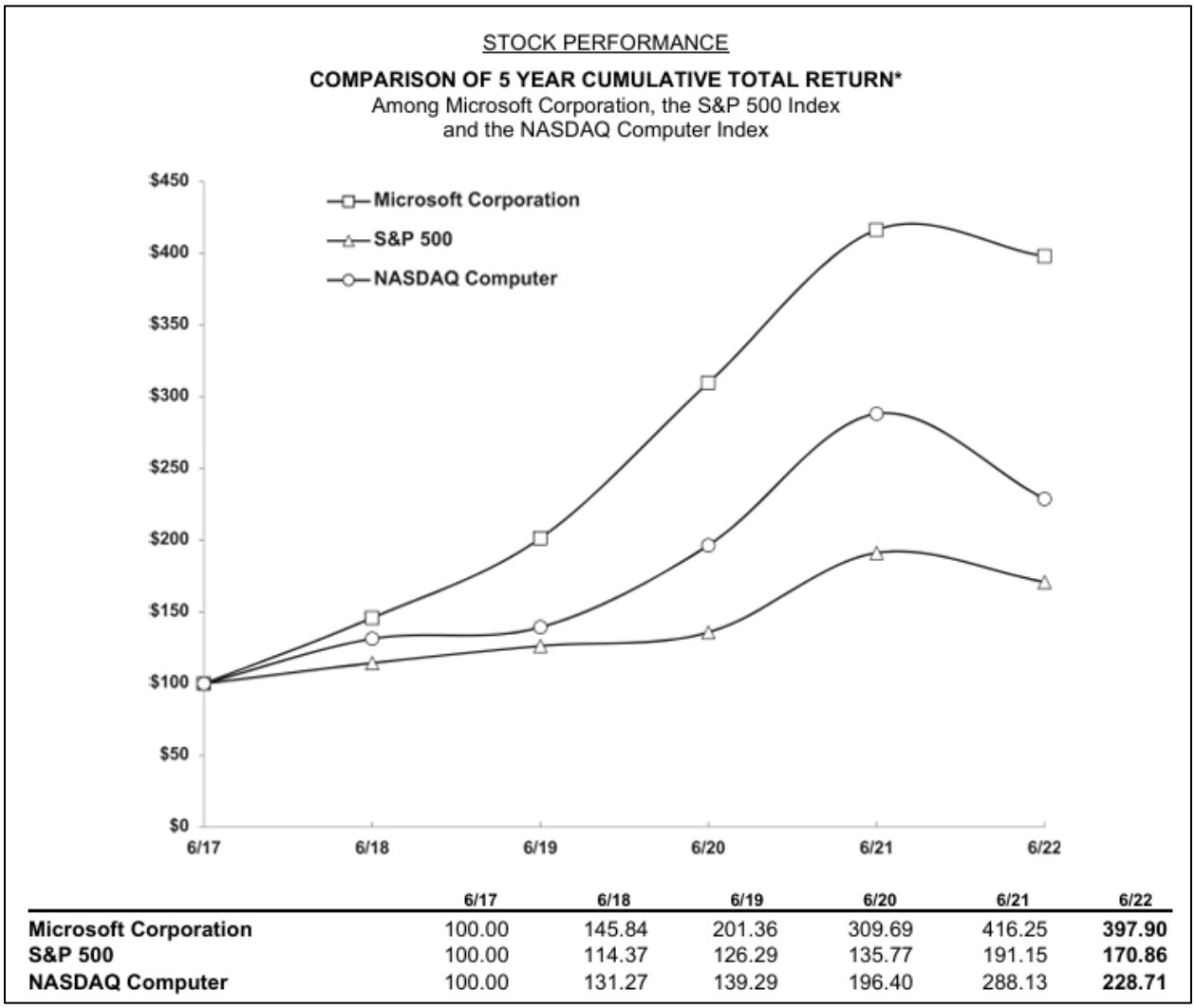
Once the RAG solution was implemented, overall accuracy was measured using the GPT similarity metric, which evaluates the similarity between user-provided ground truth answers and the model's predicted answers on a scale of 1 to 5, where 5 represents that the system produced answers that perfectly matched the ground truth answers.

Accuracy Improvement Efforts

To improve the accuracy of the Retrieval-Augmented Generation (RAG) system, several strategies were implemented that could be grouped under three different categories; Ingestion improvements, search improvements and improvements in tooling and evaluation.

Ingestion Improvements:

1. **Improve Parsing:** Efforts were made to minimize information loss during ingestion by handling data in images and complex tables. Image descriptions were generated, and various techniques were used to handle complex tables, including converting them into Markdown, HTML, and paragraph formats to ensure accurate parsing of tabular data.
    - **Information in Images:** The image below shows the performance of Microsoft Stock compared to the rest of the market (S&P 500 and NASDAQ). Efficient parsing techniques can eliminate the need for additional tables and supporting text content by extracting key insights from images and storing them in the form of text.



	6/17	6/18	6/19	6/20	6/21	6/22
Microsoft Corporation	100.00	145.84	201.36	309.69	416.25	**397.90**
S&P 500	100.00	114.37	126.29	135.77	191.15	**170.86**
NASDAQ Computer	100.00	131.27	139.29	196.40	288.13	**228.71**

Figure 1 Example of information in images

- **Complex Tables:** The image below shows an example of a table with financial data represented in a complex structure in the financial report. In this particular example, the table contains multiple sub columns (years) within a top-level column along with rows spanning over multiple lines.

(In millions, except percentages and per share amounts)	Three Months Ended December 31,		Percentage Change	Six Months Ended December 31,		Percentage Change
	2023	2022		2023	2022	
Gross margin	$ 42,397	$ 35,259	20%	$ 82,612	$ 69,929	18%
Severance, hardware-related impairment, and lease consolidation costs	0	152	*	0	152	*
Adjusted gross margin (non-GAAP)	$ 42,397	$ 35,411	20%	$ 82,612	$ 70,081	18%
Operating income	$ 27,032	$ 20,399	33%	$ 53,927	$ 41,917	29%
Severance, hardware-related impairment, and lease consolidation costs	0	1,171	*	0	1,171	*
Adjusted operating income (non-GAAP)	$ 27,032	$ 21,570	25%	$ 53,927	$ 43,088	25%
Net income	$ 21,870	$ 16,425	33%	$ 44,161	$ 33,981	30%
Severance, hardware-related impairment, and lease consolidation costs	0	946	*	0	946	*
Adjusted net income (non-GAAP)	$ 21,870	$ 17,371	26%	$ 44,161	$ 34,927	26%
Diluted earnings per share	$ 2.93	$ 2.20	33%	$ 5.92	$ 4.54	30%
Severance, hardware-related impairment, and lease consolidation costs	0	0.12	*	0	0.13	*
Adjusted diluted earnings per share (non-GAAP)	$ 2.93	$ 2.32	26%	$ 5.92	$ 4.67	27%

* *Not meaningful.*

Figure 2 Example of a complex table in financial reports

2. **Optimal Chunk Size:** The impact of chunk size on search results was analyzed. Parsed content was split into paragraphs, and a small percentage of these paragraphs were used to generate questions. Custom scripts created a question-to-paragraph mapping dataset. Different indexes with varying chunk sizes (e.g., 3k, 8k) were created, and search results were evaluated for different values of top_k.

- **Recall Values with Different Chunk Sizes:** The image and table below show recall values with different top_k search results on different indexes with different chunk sizes. For example, with a chunk size of 3k characters, the recall was 78.5% for top_k = 7 and 91% for top_k = 25.

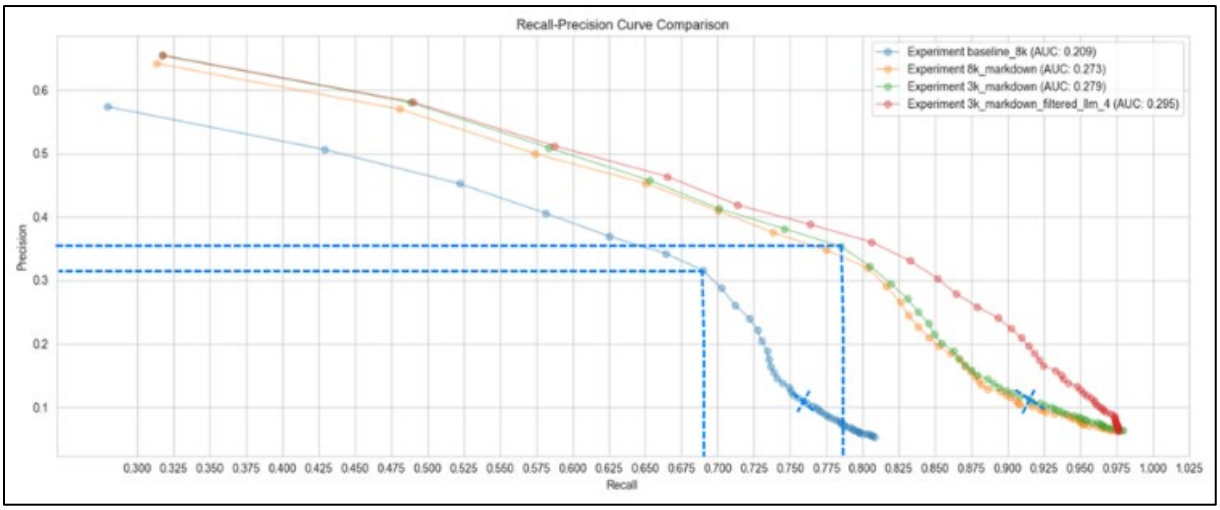


Figure 3 Recall on different chunk sizes

Recall	Chunk Size	
	8K (chars)	3K (chars)
top_k = 7	69%	78.5%
top_k = 25	76%	91%

Table 1 Search Recall on different chunk sizes

Based on the search recall value, for our content, it seems chunks of size 3K characters would work best for our content and we could use top_k of 25 to get most of the relevant search results

3. **Index Enrichment:** Additional metadata was added to chunks during ingestion to aid in better retrieval. This included metadata in additional fields used during search (like headings, section topics etc.) and other fields used for filtering (report year etc.).

Search Improvements:

1. **Pre-processing of User Input:** Techniques such as rephrasing and query expansion were used to enhance the quality of user input.
2. **Advanced Search Features:** The use of vector, semantic, and hybrid search features was implemented to increase the number of relevant results.
3. **Filtering and Reranking:** Filters were dynamically extracted from user queries, and search results were reranked to improve relevance.
    - **Example of a rephrased user query**: Below is an example of a user prompt that is then rephrased and fanned out into various smaller (more focused) search queries, produced from GPT-4o using a custom prompt

      User Query:

      Can you explain the difference between the gross profit for Microsoft in 2023 and 2024?

Rephaser Output*:*


```
"search_requests": [
    {
        "search_query": "Please provide the gross profit for the year 2023 for Microsoft",
        "subsidiary_filter": "Microsoft",
        "earliest_year_filter": 2023,
        "reasoning": "This is the original value the user is asking about."
    },
    {
        "search_query": "Please provide the revenue for the year 2023 for Microsoft",
        "subsidiary_filter": "Microsoft",
        "earliest_year_filter": 2023,
        "reasoning": "The user is asking for the gross profit for the year to 2023. In order to calculate the gross profit we need the value of the revenue."
    },
    {
        "search_query": "Please provide the cost of goods sold (COGS) for the year 2023 for Microsoft",
        "subsidiary_filter": "Microsoft",
        "earliest_year_filter": 2023,
        "reasoning": "The user is asking for the gross profit for the year to 2023. In order to calculate the gross profit we need the value of the COGS."
    },
    {
        "search_query": "Please provide the gross profit for the year 2024 for Microsoft",
        "subsidiary_filter": "Microsoft",
        "earliest_year_filter": 2024,
        "reasoning": "This is the original value the user is asking about."
    },
    {
        "search_query": "Please provide the revenue for the year 2024 for Microsoft",
        "subsidiary_filter": "Microsoft",
        "earliest_year_filter": 2024,
        "reasoning": "The user is asking for the gross profit for the year to 2024. In order to calculate the gross profit we need the value of the revenue."
    },
    {
        "search_query": "Please provide the cost of goods sold (COGS) for the year 2024 for Microsoft",
        "subsidiary_filter": "Microsoft",
        "earliest_year_filter": 2024,
        "reasoning": "The user is asking for the gross profit for the year to 2024. In order to calculate the gross profit we need the value of the COGS."
    }
]
```


Evaluation and Tooling:

1. **Standardizing Datasets:** As the project progressed, soon we had too many datasets which started to result in inconsistent ways of measuring the quality of the bot response. To resolve that issue, we standardized our dataset and used AML to store, document and version them. When any updates were made to the dataset (say some inconsistency was found in the golden data set and that user prompt was ignored from accuracy computation the dataset was updated and a new version created). This way everyone was using a known dataset for evaluations.

2. **Standardizing Accuracy Calculation:** To calculate accuracy of bot's answers, similarity score is used, which is a rating between 1 to 5 based on how similar bot's answer is to the golden dataset. Initially, the similarity score metric included in the Prompt Flow was used to calculate this, but soon we realized that from the produced scores it wasn't easy to understand why certain things were scored in a certain way. So, the team created its own prompt and calibrated it against how humans had done the evaluations. The tuned prompt was then used in prompt flow to run evaluations. The prompt, along with scoring the bot's result also provides reason why it gave that score, which useful in analyzing the results.

   The following image shows a snippet of that prompt:

```
System:
You are an expert in evaluating financial reports. Given  a Question, Generated, Ground truth. Eval
The questions are about financial data in some financial reports.

User:
# INSTRUCTIONS
Focus on the presence of the financial numbers that are mentioned in the ground truth.
You can neglect some precision, like if the generated has 117.53% while the GT is 117.5% that's fir
You also consider units, like 7,240,853,000 can be the same as 7.24b or 7.24B. Neglect citations ir
It is okay is the answer contains some extra information like calculations and reasoning.

You are to examine each question step by step give
1 if the generated answer is completely irrelevant
2 if the generated answer has the right direction but there is no further detail provided
3 if the generated answer is only contains parts  of the information requested by the question
4 if conclusion in the Ground truth is not in the Generated answer but all other supporting points
5 if the information that is contained in the Ground truth is also contained in generated.


# EXAMPLE:
For example if the generated is
```

Figure 4 Custom prompt for scoring bot response

3. **Automating Accuracy Calculations:** Tools were also developed to automate the generation of predictions and the evaluation of accuracy in a more repeatable and consistent way. More details on Analysis can be found in the Eval Tool section

4. **Analyzing problematic queries:** Running evaluations and just looking at the overall / average score wasn't enough to analyze the cause of the issue. So, we took a first pass at categorizing the user queries into certain buckets. These categories became:
    - Queries that are direct hit on some content in the report – like revenue for a year
    - Queries where we need to perform some calculations - like Gearing Ratio
    - Queries that do compare and contrast across some KPI – largest two segments by revenue
    - Open ended queries where we need to perform analysis – like why and what?

    Later, LLM was leveraged to auto categorize ground truth questions as more ground truth questions were updated.

    Once the questions were categorized, evaluations were broken down by these categories to ease analysis and understanding problematic queries. Figure below shows a snapshot of the spread of the user queries in the ground truth by the following categories:

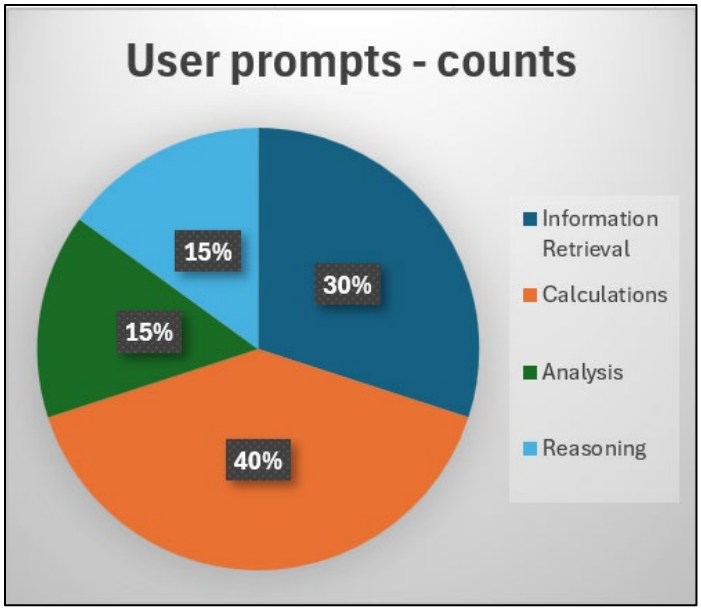


Figure 5 Spread of user prompt by categories

The figure below shows a snapshot of similarity scores across these categories

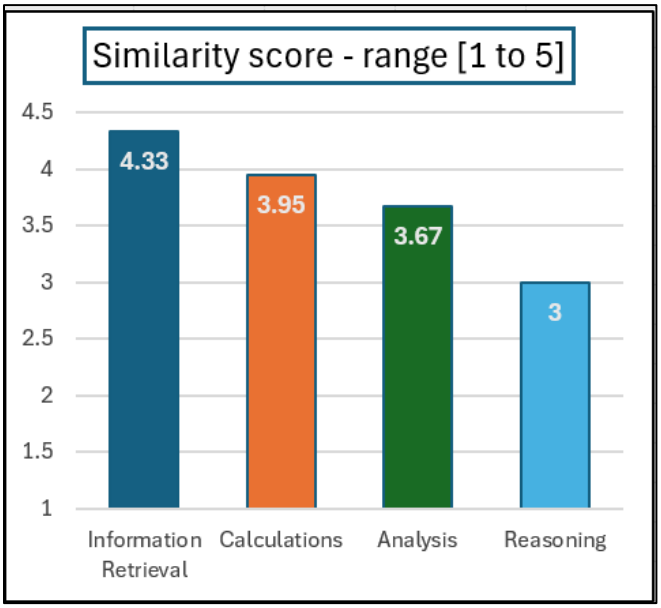


Figure 6 Average similarity score by category

Later, another category was added (difficulty level) with below values. The final results were reported across these categories.

- **Easy**: If the search context had the answer user was looking for
- **Medium**: If there was no direct hit and required some calculations to get to the final result
- **Hard**: If the question required some analysis to be performed on the retrieved/calculated data like a financial analyst does.

Results after the Accuracy Improvement Efforts

After multiple iterations of accuracy improvement efforts and stabilizing the solution the overall accuracy of the system came around 4.3, making the overall solution more acceptable to the end user. The solution was also scaled up to cover content across multiple years, with over 15 financial reports and roughly 1300 pages in total.

Another important metric to consider – the pass rate based on question type (% of time some answers were scored a value of 4 or a 5) to ensure the copilot was consistently passing these ground truths. The table below lists the pass rate by difficulty:

Pass rate by difficulty			
Difficulty	Easy	Medium	Hard
Pass Rate	95	79	72

Table 2: Accuracy Improvement Analysis by Difficulty

Solution architecture

The RAG solution is designed to handle various tasks using robust and scalable architecture. The architecture includes the following key aspects:

Security

- **User Authentication**: The solution uses Microsoft Entra ID for user authentication, ensuring secure access to the system.
- **Network Security**: All runtime components are locked behind a Virtual Network (VNet) to ensure that traffic does not traverse public networks. This enhances security by isolating the components from external threats.
- **Managed Identities**: The solution leverages managed identities where possible to simplify the management of secrets and credentials. This reduces the risk of credential exposure and makes it easier to manage access to Azure resources.

Composability

- **Modular Design**: The solution is broken down into smaller, well-defined core microservices and skills that act as plug-and-play components. This modular design allows you to use existing services or bring in new ones to meet your specific needs.
- **Core Microservices**: Backend services handle different aspects of the solution, such as session management, data processing, runtime configuration, and orchestration.
- **Skills**: Specialized services provide specific capabilities, such as cognitive search and image processing. These skills can be easily integrated or replaced as needed.

Iterability

- **Configuration Service**: The solution includes a configuration service that allows you to create runtime configurations for each microservice. This enables you to make changes, such as updating prompts or search indexes, without redeploying the entire solution.
- **Per-User Prompt Configuration:** Configuration service can be used to apply different configurations for each user prompt, allowing for rapid experimentation and iteration. This flexibility helps to quickly adapt to changing requirements and improve the overall system.
- **Testing and Evaluation**: The solution also comes with the ability to run dummy/simulated conversations in the form of nightly runs, end-to-end integration tests on demand, and an evaluation tool to perform end-to-end evaluation of the solution.

Logging and Instrumentation

- **Application Insights**: The solution integrates with Azure Application Insights in Azure Monitor for logging and instrumentation, making it easy to debug by reviewing logs.
- **Traceability**: One can easily trace what is happening in the backend using the conversation_id and dialog_id (unique GUIDs generated by the frontend) for each user session and interaction. This helps in identifying and resolving issues quickly.

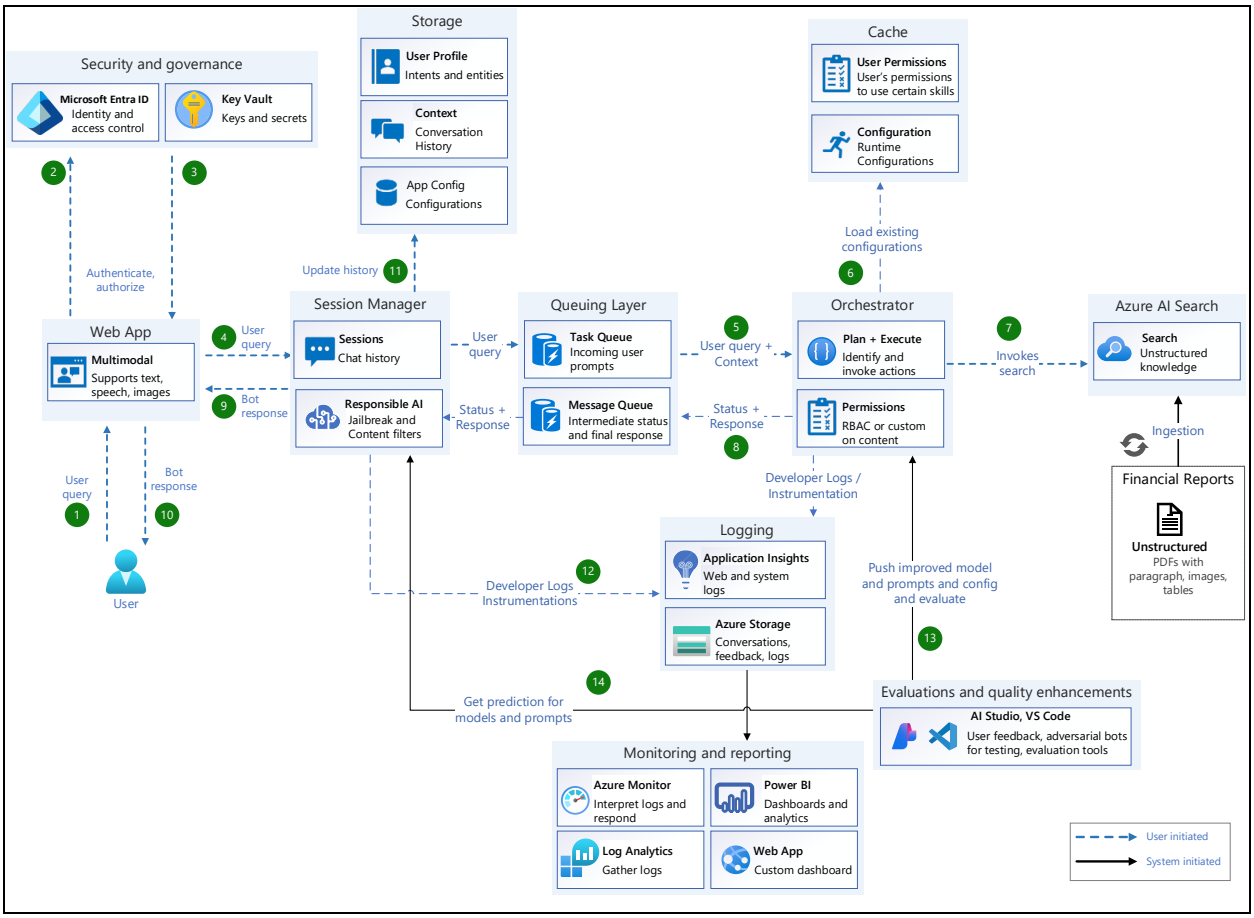


Figure 7 Solution Architecture

Before exploring the data flow, we begin with the Ingestion process, crucial for preparing the solution. This involves creating and populating the search index with relevant content (corpus). Detailed instructions on parsing, chunking, and indexing can be found in the Solution capabilities section of the document.

User Query Processing Flow

1. **User Authentication**:
    a. Users interact with the bot via a web application and must authenticate using Microsoft Entra ID to ensure secure access.
2. **User Interaction**:
    a. Once authenticated, users can submit requests through text or voice:
        i. The web app establishes a WebSocket connection with the backend session manager.
        ii. For voice interactions, Microsoft Speech Services are utilized for live transcription. The web app requests a speech token from the backend, which is then used in the Speech SDK for transcription.
3. **Token Management**:
    a. The backend retrieves secrets from Key Vault to generate tokens necessary for front end operations.
4. **Transcription and Submission**:

    a. After transcription, the web app submits the transcribed text to the backend.
5. **Session Management**:
    a. The session manager assigns unique connection IDs for each WebSocket connection to identify clients. User prompts are then pushed into a message queue, implemented using Azure Cache for Redis.
6. **Orchestrator:** The orchestrator plays a critical role in managing the flow of information. It reads the user query from the message queue and performs several actions:
    a. Plan & Execute: It identifies the required actions based on the user query and context.
    b. Permissions: It checks user permissions using Role-Based Access Control (RBAC) or custom permissions on the content. NOTE: The current implementation doesn't do it, however Orchestrator could easily be updated to do so.
    c. Invoke Actions: It triggers the appropriate actions, such as invoking the Azure AI Search for retrieving relevant information.
7. **Azure AI Search:** The orchestrator interacts with Azure AI Search to query the unstructured knowledge base. This involves searching through financial reports or other content to find the information the user requested.
8. **Status & Response:** The orchestrator processes the search results and formulates a response. It updates the queue with the status and the final response, which includes any necessary predictions or additional information.
9. **Session Manager:** The response from the orchestrator is sent back to the session manager. This component is responsible for maintaining the session's integrity and ensuring that each client receives the correct response. It uses the unique connection ID to route the response back to the appropriate client.
10. **Web App:** The web app receives a response from the session manager. It then delivers the bot's response back to the user, completing the interaction cycle. This response can be in text and /or speech format, depending on the user's initial input method.
11. **Update History:** On successful completion of bot response, the session manager updates the user profile and conversation history in the storage component. This includes details about user intents and entities, ensuring that the system can provide personalized and context-aware responses in future interactions.
12. **Developer Logs / Instrumentation:** Throughout the process, logs and instrumentation data are collected. These logs are essential for monitoring and debugging the system, as well as for enhancing its performance and reliability.
13. **Evaluations and Quality Enhancements:** The collected data along with golden datasets, manual feedback is utilized for ongoing evaluations and quality enhancements. Tools like Azure AI Foundry and VS Code along with the configuration service are used to test the bots, develop and evaluate different prompts and models.
14. **Monitoring and Reporting:** The system is continuously monitored using Azure Monitor and other analytics tools. Power BI dashboards provide insights into system performance, user interactions, and other key metrics. This ensures that the solution remains responsive and effective over time.

Solution capabilities

The solution will support the following capabilities:

Document Ingestion Pipeline

Document ingestion in a Retrieval-Augmented Generation (RAG) application is a critical process that ensures efficient and accurate retrieval of information. Currently, the ingestion service supports the following scenarios:

1. Large financial documents containing complex tables, graphs, charts and other figures
2. Large retail product catalogs containing images and descriptions

The overall process can be broken down into three primary stages:

1. **Document Loading**

   The Document Loader is the first stage in the document ingestion pipeline. Its primary function is to load documents into memory and extract text and metadata. One can configure to use either Azure AI Document Intelligence service or **LangChain** with Azure AI Document Intelligence for text extraction.

2. **Document Parsing**

   Document Parser is the second stage in the document ingestion pipeline. Its role is to process the loaded text and metadata, splitting the document into manageable chunks and cleaning the text for indexing. One can use either a **Fixed-size chunking** with overlap or go with **Layout-based chunking,** where with the use of LLMs chunking is done based on whether certain paragraphs should be kept together. The solution used layout-based chunking and sections and subsections were extracted and maintained as metadata for the chunked paragraphs.

3. **Document Indexing**

   Document Indexer is the final stage in the document ingestion pipeline. Its purpose is to upload the parsed chunks into a search index, enabling efficient retrieval based on user queries. As part of document parsing additional metadata (section and subsection names and titles) are also passed along with the text to be indexed. Main content and certain metadata fields are also stored as vectors to enable better retrieval.

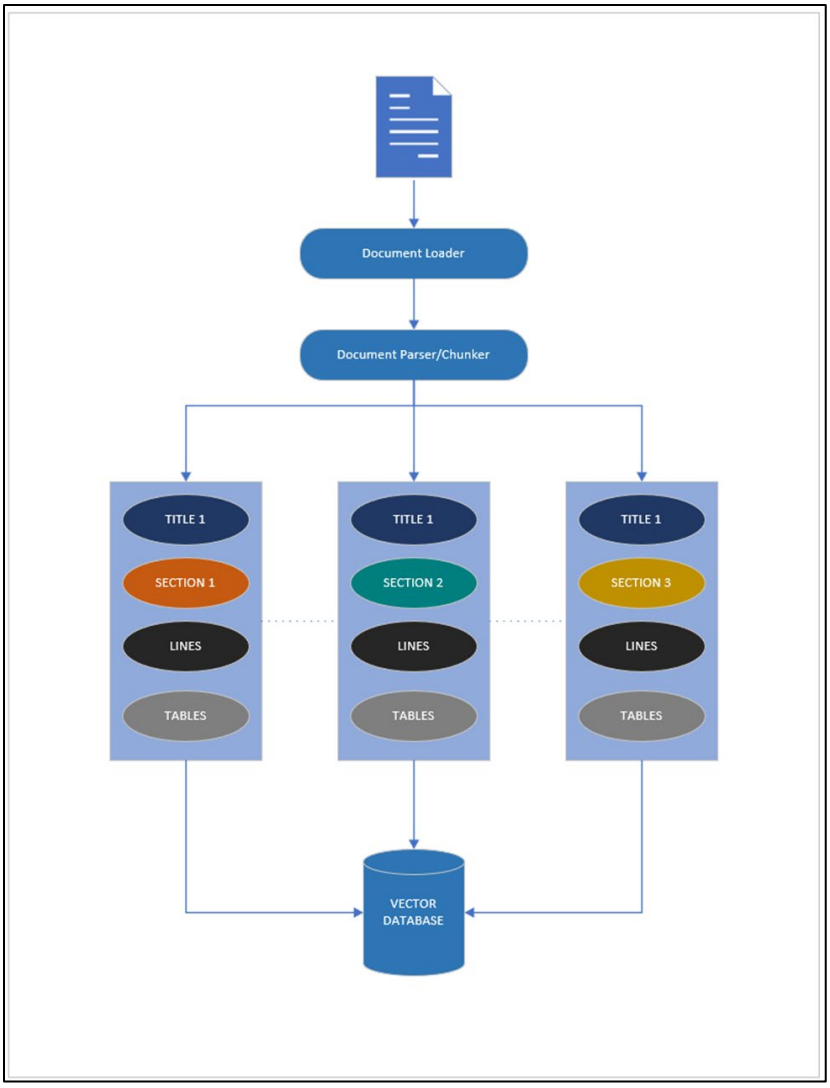


Figure 8: Indexing by document

Search

Once the Ingestion pipeline is executed successfully resulting in a valid, queryable Search Index, the Search service can be configured and integrated into the end-to-end RAG application.

The Search Service exposes an API that enables users to query a search index in Azure AI Search. It processes natural language queries, applies requested filters, and invokes search requests against the preconfigured search configuration using the Azure AI Search SDK.

Search Index Configuration: The search index configuration defines the schema and the type of search to apply, including simple text search, vector search, hybrid search, and hybrid with additional semantic understanding. This is done as part of index creation and document ingestion.

User Query: The process starts with a user query, a natural language input from the user.

Query Embeddings Generation: Using an LLM, the query is vectorized so hybrid search could be performed on the user query.

Search Filter Generation: From the user query, filters, based on criteria such as equality, range conditions, and substring matches, are generated to refine the search results.

Search Invocation: The search service constructs a query using the embedding and filters, sends it to Azure AI Search via the Azure AI Search SDK, and receives the search results.

Pruning: Pruning refines these results further to ensure relevance based on additional semantic filtering and ranking.

Search Results: The final output represents the items from the search index that best match the user's query, after all filters and pruning have been applied.

Query Reprocessing

One of the first steps we approach when we receive a chat message is preprocessing to make sure that we have better search results that will enable our RAG system to answer the question accurately. We perform the following steps as part of the preprocessing:

1- **Message Rephrasing:** When the chatbot receives a new message, we need to make sure that we rephrase this message based on the chat history as this new message may depend on the previous context. For example, when we ask, "Which team won the Premier League in 2023?" and then we ask a follow-up question "What about the following year?" we will need to rephrase this follow-up question to "Which team won the premier League in 2024?"

2- **Fanout**: If the query is asking about complex data that does not exist in the indexed documents, it can be calculated by other simpler data that already exists in the document. For example, if the indexed documents are financial reports and the query is asking about the gross profit margin, if we search for gross profit margin, we may not find it in the indexed documents. But to calculate the gross profit margin, we can use both the Revenue and the Cost Of Goods (COGS) which exist in the indexed documents. If we can break down the original question about gross profit margin to sub questions for Revenue and COGS, then that would help the model to calculate the gross profit margin given these values.

Check out the new service Rewrite queries with semantic ranker in Azure AI Search (Preview).

AI Skills

To ensure modularity and ease of maintenance, our solution designates any service capable of providing data as a "skill." This approach allows for seamless plug-and-play integration of components. For instance, Azure AI Search is treated as a skill within our architecture. Should the solution require additional data sources, such as a relational database, these can be encapsulated within an API and similarly integrated as skills.

Wrapping content providers as skills serves two primary purposes:

1. **Enhanced Logging and Debugging**:

Skills can be configured to incorporate logging and instrumentation fields, ensuring that all generated logs include relevant context. This uniformity greatly facilitates efficient debugging by providing comprehensive log insights.

2. **Dynamic Configuration**:
   Skills can leverage the configuration service to expose runtime configurations. This flexibility is particularly beneficial during evaluations, allowing for adjustments such as modifying the number of top-k results or switching to a different search index to accommodate improvements in data ingestion.

By adopting this skill-based approach, the architecture remains adaptable and scalable, supporting ongoing enhancements and diverse data integration.

Sharing Intermediate Results

Sharing intermediate results from the RAG process provides the user with details about what is happening once a query is sent to the bot. This is especially useful when the query takes a long time to return. This also helps to see how the query was broken down into smaller queries, so if something goes wrong (especially for harder queries), the user could have the ability to rephrase and get a better response.

Once the user sends the query to the bot, the orchestrator emits intermediate updates like “Searching for ...”, “Retrieved XX results...” before the final answer is delivered.

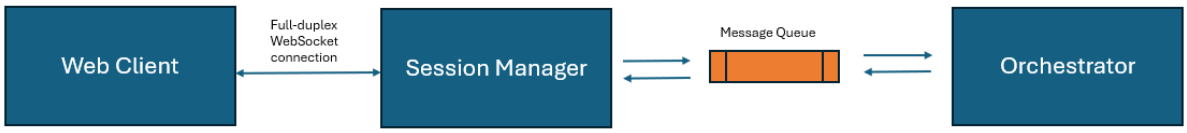


Figure 9: Messaging Framework

Architecture to support this:

1. **WebSocket connection (Client <> Sessions Manager)** - When the client connects to the session manager a persistent WebSocket connection is created, all communication between the client and session manager is handled through this connection. This also allows queueing up of multiple messages from the client. The session manager listens to the incoming messages and queues them up in a message queue. Then requests are handled one by one. Meanwhile, intermediate messages and final answers of the previously submitted messages are sent asynchronously back to the client.
2. **Message Queue (Session Manager <> Orchestrator)** - once the session manager receives a request its enqueued into a task queue. Since there can be multiple orchestrator instances running in the cluster, the task queue ensures only one instance receives a particular request. The orchestrator then begins the RAG process. As the RAG process continues, the orchestrator sends intermediate messages by publishing them to a message queue. All instances of the session

manager subscribe to this message queue. The instance handling the client relevant to the incoming message forwards it to the client.

Runtime Configuration

The runtime configuration service enhances the architecture's dynamicity and flexibility. It enables core services and AI skills to decouple and parameterize various components, such as prompts, search data settings, and operational parameters. These services can easily override default configurations with new versions at runtime, allowing for dynamic behavior adjustments during operation.

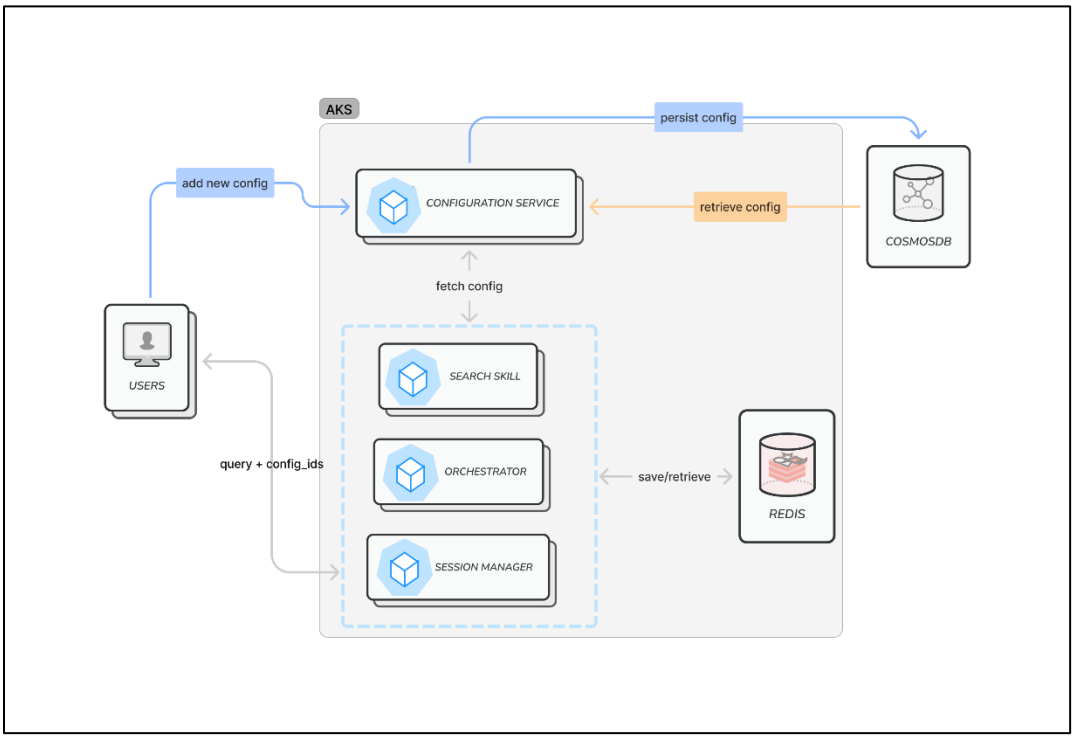


Figure 10: Runtime Configuration

- **Core Services and AI Skills:** define unique identifiers for their individual configurations. At runtime, they check if the payload consists of a configuration override. If yes, they attempt to retrieve it from the cache. In a scenario where it is not present in cache memory, i.e., first time fetch, they read from the configuration service and save it in cache memory for future references.
- **Configuration Service:** facilitates Create, Read and Delete operations for a new configuration. Validates the incoming config against a Pydantic model and generates a unique version for the configuration upon successful save.
- **Cosmos DB**: persists the new config version.
- **Redis**: high availability memory store for storing and quick retrievals of configurations for subsequent queries.

Evaluation Tool

Improving accuracy of RAG based solution is a continuous process that requires experimenting with different changes, running predictions with those changes (running user query through the bot), evaluating the bot produced result against the ground truth and analyzing the issues and then repeating these steps again.

All this required a consistent way of evaluating the end-to-end results. Initially the team did the evaluation and scoring of the results manually but as the search index grew (ingested a few thousand financial reports) and the golden dataset grew, doing it manually was very time-consuming. So, the team developed a custom prompt and used LLM to do the scoring. The prompt was calibrated against the human scores. Once the prompt was stabilized Evaluation tool was built to do two things:

1. For each golden question call the bot endpoint and generate the prediction (bot answer)
2. Then take the ground truth and predicted results to run evaluation of them and produce some metrics.

Implementation Guide

Please refer to GitHub repo.

Additional Resources

- Get started on Azure AI Foundry
- Evaluation of generative AI applications
- Generate adversarial simulations for safety evaluation
- Generate synthetic data and simulate non-adversarial tasks
- AI architecture guidance to build AI workloads on Azure
- Responsible AI Tools and Practices