`ai_parse_document` function

09/23/2025

Applies to: ✓ Databricks SQL ✓ Databricks Runtime

> **Important**
>
> This feature is in Beta.

The `ai_parse_document()` function invokes a state-of-the-art generative AI model from Databricks Foundation Model APIs to extract structured content from unstructured documents.

Requirements

> **Important**
>
> The model powering this function is made available using Mosaic AI Model Serving Foundation Model APIs. See Applicable model developer licenses and terms for information about which models are available on Databricks and the licenses and policies that govern the use of those models.
>
> If models emerge in the future that perform better according to Databricks's internal benchmarks, Databricks may change the models and update the documentation.

- A workspace in a US region that supports AI Functions optimized for batch inference.
  - If your workspace is not in the US, but is in a region that supports AI Functions optimized for batch inference, then cross geography routing must be enabled on your workspace.
- Mosaic AI Agent Bricks Beta enabled.
- Databricks Runtime 17.1 or above.
- If you are using serverless compute, the following is also required:
  - The serverless environment version must be set to 3 or above, as this enables features like `VARIANT`.
  - Must use either Python or SQL. For additional serverless features and limitations, see Serverless compute limitations.
- The `ai_parse_document` function is available using Databricks notebooks, SQL editor, Databricks workflows, jobs, or Lakeflow Declarative Pipelines.

- See the Beta products pricing page for billing details.

Data security

Your document data is processed within the Databricks security perimeter. Databricks does not store the parameters that are passed into the `ai_parse_document function` calls, but does retain metadata run details, such as the Databricks Runtime version used.

Supported input file formats

Your input data files must be stored as blob data in bytes, meaning a binary type column in a DataFrame or Delta table. If the source documents are stored in a Unity Catalog volume, the binary type column can be generated using Spark `binaryFile` format reader.

The following file formats are supported:

- PDF
- JPG / JPEG
- PNG
- DOC/DOCX
- PPT/PPTX

Syntax

```
ai_parse_document(content)
```

```
ai_parse_document(content, Map("version" -> "2.0"))
```

Arguments

- `content`: A `BINARY` expression representing the input byte array data.
- `version`: The version of the output schema, supported: "1.0", "2.0".

- `'imageOutputPath'`: Optional. Save rendered page images to a Unity Catalog volume for reference or multi-modal RAG applications.
- `'descriptionElementTypes'`: AI-generated descriptions. Only descriptions for `figures` are supported for version 2.0, so `'*'` and `'figure'` produce the same behavior.
  - '' (empty string): No descriptions are generated.
  - `'figure'`: Generate descriptions for figures only. Only supports AI-generated descriptions.
  - `'*'` (default): Generate descriptions for all supported element types.

Returns

The `ai_parse_document` function extracts the contextual layout metadata from the document, like `page_number`, `header`, `footer`. It also extracts the content of the document such as text paragraphs and represents them in markdown. For version 2.0, tables are represented in HTML. The output is of `VARIANT` type.

Important

The function output schema is versioned using a major.minor format like, "1.0". Databricks might upgrade the supported or default version to reflect improved representations based on ongoing research.

- Minor version upgrades are backward-compatible and might only introduce new fields.
- Major version upgrades might include breaking changes such as field additions, removals, or renamings.

The following is the output schema:

Note

As of September 22, 2025, the output schema is on version "2.0" and has been updated to include:

- `descriptions` for AI-generated figure descriptions.
- `bbox` for bounding box coordinates.

To migrate your existing workloads to use the updated schema, see Migrate workloads to updated schema.

```
{
  "document": {
    "pages": [
      {
        "id": INT,                // 0-based page index
        "image_uri": STRING       // Path to saved page image (if enabled)
      }
    ],
    "elements": [
      {
        "id": INT,                 // 0-based element index
        "type": STRING,            // Supported: text, table, figure, table, title,
caption, section_header,
                                   // page_footer, page_header, page_number, footnote
        "content": STRING,         // Text content (markdown) of the target element
        "bbox": [                  // Bounding box coordinates
          {
            "coord": [ INT ],
            "page_id": INT
          }
        ],
        "description": STRING      // AI-generated description for figures
      }
    ]
  },
  "error_status": [
    {
      "error_message": STRING       // The detailed error message
      "page_id": INT                // 0-based page index
    }
  ],
  "metadata": {
    "id": STRING,
    "version": STRING               // The version of the output schema
  }
}
```

Migrate workloads to updated schema

The steps in this section describe how to migrate workloads that were created before September 22, 2025 to use the updated output schema.

1. In your SQL request, specify a specific schema version using the `version` parameter.

```SQL
SELECT
ai_parse_document(
  content,
  map('version', '2.0')
) AS parsed
FROM READ_FILES('/path/to/documents', format => 'binaryFile');
```

1. Modify your code to read content from the `elements` array instead of the `pages` array.
2. Re-evaluate metadata. For example, if you were using `page` metadata like headers and footers, you need to develop an alternative approach for extracting this information from the `elements`.
3. Validate your updated logic with sample documents before migrating your full workload.
4. Consider enabling figure descriptions or image persistence if they are relevant to your use case.
5. Check permissions. For example, if you plan to use image persistence, ensure you have the correct permissions set up for the target Unity Catalog volume.

Examples

The following example uses `ai_parse_document` to extract document layouts as `VARIANT` output for a single file and specifies,

- Where to save rendered images.
- Pins an output schema version.
- Enables AI-generated descriptions for figures.

```SQL
SELECT
  path,
  ai_parse_document(
    content,
    map(
      'version', '2.0',
      'imageOutputPath', '/Volumes/catalog/schema/volume/directory/',
      'descriptionElementTypes', '*'
    )
  ) as parsed_doc
FROM READ_FILES('/Volumes/data/documents/', format => 'binaryFile');
```

The following example uses `ai_parse_document` to extract document layouts as `VARIANT` output for files in a Unity Catalog volume.

SQL

SQL

```
SELECT
  path,
  ai_parse_document(content)
FROM READ_FILES('/Volumes/path/to/your/directory', format => 'binaryFile');
```

Python

Python

```
from pyspark.sql.functions import *


df = spark.read.format("binaryFile") \
  .load("/Volumes/path/to/your/directory") \
  .withColumn(
    "parsed",
    expr("ai_parse_document(content)"))
display(df)
```

Scala

Scala

```
import org.apache.spark.sql.functions._


val df = spark.read.format("binaryFile")
  .load("/Volumes/path/to/your/directory")
```

```
    .withColumn(
      "parsed",
      ai_parse_document($"content"))
  display(df)
```

The following example uses `ai_parse_document` to separate each top-level field of the output. For example, `document.pages`, `document.elements`, `error_status`, and `metadata` into individual columns.

SQL

SQL

```sql
WITH corpus AS (
  SELECT
    path,
    ai_parse_document(content) AS parsed
  FROM
    READ_FILES('/Volumes/path/to/source/file.pdf', format => 'binaryFile')
)
SELECT
  path,
  parsed:document:pages,
  parsed:document:elements,
  parsed:error_status,
  parsed:metadata
FROM corpus;
```

Python

Python

```python
from pyspark.sql.functions import *


df = spark.read.format("binaryFile") \
 .load("/Volumes/path/to/source/file.pdf") \
 .withColumn(
   "parsed",
   ai_parse_document("content")) \
 .withColumn(
   "parsed_json",
   parse_json(col("parsed").cast("string"))) \
 .selectExpr(
    "path",
    "parsed_json:document:pages",
    "parsed_json:document:elements",
    "parsed_json:error_status",
    "parsed_json:metadata")
display(df)
```

Scala

Scala

```
import com.databricks.sql.catalyst.unstructured.DocumentParseResultV2_0
import org.apache.spark.sql.functions._


val df = spark.read.format("binaryFile")
 .load("/Volumes/path/to/source/file.pdf")
 .withColumn(
   "parsed",
   ai_parse_document($"content").cast(DocumentParseResultV2_0.SCHEMA))
 .select(
   $"path",
   $"parsed.*")
display(df)
```

Limitations

- While Databricks continuously works to improve all of its features, LLMs are an emerging technology and may produce errors.
- The `ai_parse_document` function can take time to extract document content while preserving structural information, especially for documents that contain highly dense content or content with poor resolution. In some cases, the function may take a while to run or ignore content. Databricks is continuously working to improve latency.
- See Supported input file formats. Databricks welcomes feedback on which additional formats are most important for your organization.
- Customizing the model that powers `ai_parse_document` or using a customer-provided model for `ai_parse_document` is not supported.
- The underlying model may not perform optimally when handling images using text of non-Latin alphabets, such as Japanese or Korean.

- Documents with digital signatures may not be processed accurately.