



DEVELOPER DOCUMENTATION

Groundzero Spark Runtime

Distributed processing with Apache Spark and Iceberg on Groundzero.

Developer documentation

Edition 2026-10-02

Runtime environments: Spark, DuckDB, Standard, Lakehouse

Contents

Guide - Spark Runtime

1	Overview	5
1.1	What the Spark runtime gives you	6
1.2	Choose the Spark runtime	6
1.3	How a run works	7
1.4	Versions	8
1.5	What changed recently	8
1.6	Machine-learning variants	9
2	Quickstart	10
2.1	Before you begin	10
2.2	Step 1: Create the job	10
2.3	Step 2: Run and read the result	11
2.4	Step 3: Try it in a notebook	12
2.5	Step 4: Schedule it	12
3	How-to guides	16
3.1	Read from a connection	16
3.2	Query and write Lakehouse (Iceberg) tables	17
3.3	Work with outputs and previews	19
3.4	Use parameters and placeholders	20
3.5	Chain jobs	21
3.6	Schedule through ETL jobs	21
3.7	Install and use extra packages	21
3.8	Tune performance	22
3.9	Log from your job	23
3.10	Check data quality	24
3.11	Parse PDFs and run Agents over documents	24
4	API reference	26
4.1	Conventions	26
4.2	Session	26
4.3	Lakehouse (Iceberg) methods	27
4.4	Object storage and files	31
4.5	Relational databases and warehouses	32
4.6	LLM methods	33
4.7	Data quality	35
4.8	PDF extraction	37
4.9	Agents	38
4.10	Jobs, mail and clients	40
4.11	Placeholders and outputs	42
4.12	Connector method index	43
5	Installed packages	50
5.1	Python	50
5.2	Data, SQL and cloud libraries	50

5.3	LLM and AI libraries	50
5.4	Document extraction	51
5.5	Notebook and plotting libraries	51
5.6	JVM libraries	51
5.7	System packages	51
6	Limits and troubleshooting	52
6.1	Limits and quotas	52
6.2	Restrictions in job scripts	52
6.3	Things to know	53
6.4	Troubleshooting	53
6.5	FAQ	55
7	Examples gallery	56
7.1	Daily revenue rollup in the lakehouse	56
7.2	Incremental load from PostgreSQL to Parquet	56
7.3	Deduplicate and enrich with window functions	57
7.4	Stream Kafka events into an Iceberg table	58
7.5	Classify support tickets with an LLM	59
7.6	Train and apply a scikit-learn model	60
7.7	Load a warehouse query into the lakehouse in bulk	61
7.8	Extract structured data from a folder of PDFs with an Agent	62
7.9	Quality gate with notification	62
7.10	Orchestrate sub-jobs	63

Appendix A - Choosing a Runtime and Shared Concepts

A.1	Choosing a runtime	64
A.1.1	At a glance	65
A.1.2	Decision guide	65
A.1.3	Where the choice is made	66
A.2	How a run starts	67
A.2.1	From Run to results	67
A.2.2	Datasets, jobs and notebooks compared	67
A.2.3	Sessions and warm starts	67
A.2.4	Timeouts and retries	68
A.2.5	Compute sizes, workers and GPU size	68
A.3	What your code can use during a run	70
A.3.1	Run details	70
A.3.2	Job placeholders	70
A.3.3	Imported models	70
A.4	Connections and secrets	71
A.5	The lakehouse (Iceberg)	72
A.6	Custom runtimes	73
A.7	ML variants	75
A.8	Jobs: lifecycle, status and logs	76
A.8.1	Statuses	76

A.8.2	Logs	76
A.8.3	Scheduling	76
A.9	Glossary	77

GUIDE

Spark Runtime

Distributed processing with Apache Spark and Iceberg on Groundzero.

-
- 1 Overview
 - 2 Quickstart
 - 3 How-to guides
 - 4 API reference
 - 5 Installed packages
 - 6 Limits and troubleshooting
 - 7 Examples gallery

The Groundzero Spark runtime runs Python jobs, dataset queries and notebooks on Apache Spark. A job is a Python script. It gets one ready-made `sparkSession`, a set of governed connectors, and a place to leave its result (`outputDF`). Each run starts its own Spark cluster with a driver and workers. You choose the size in Compute, so the same script runs on a small sample or on a large table.

This guide shows how to run your first job, read and write data, use the Groundzero helper functions, and handle the limits and errors you may meet. It ends with a gallery of working examples.

1 Overview

1.1 What the Spark runtime gives you

- Apache Spark with Python 3.12 (PySpark, pandas, NumPy, scikit-learn, DuckDB and more).
- The Groundzero helper library (`gZ`). It creates your Spark session and adds every connector as a method on that session.
- More than 50 connector families: databases, warehouses, object stores, streams, SaaS APIs, vector stores and LLM providers.
- Apache Iceberg support for Groundzero Lakehouse tables, with per-table permissions, audit logging and lineage.
- A paged preview of your result in the UI, and logs streamed while the job runs.
- JupyterLab notebooks on the same runtime.

You do not install or deploy anything. Pick the Spark runtime and a compute size when you create a job, dataset or notebook.

1.2 Choose the Spark runtime

Groundzero offers three runtime families. Use the table to decide which one fits your workload.

	Spark	DuckDB	Standard
Engine	Distributed Apache Spark	Single-node DuckDB	Managed query engine
Language	Python (PySpark)	Python (DuckDB)	SQL plus pandas Prep
Scales by	Adding workers	Larger compute size	Managed by Groundzero
Best for	Large joins, wide ETL, streaming, ML	Fast single-node analysis	Simple governed datasets
Chosen for	Jobs, notebooks, datasets	Jobs, notebooks, datasets	Datasets only
Compute picker	Yes, with workers	Yes	None

Choose Spark when:

- Your data does not fit comfortably in the memory of one machine.
- You need Structured Streaming (Kafka, Kinesis, Pulsar, Iceberg streaming).
- You need the broadest connector coverage, including bulk loads from Snowflake and Redshift straight into the lakehouse.
- You train or score models over a DataFrame, or call an LLM for every row.

Choose another runtime when the data is small (a few gigabytes) and start-up time matters more than scale. Spark starts a driver and at least one worker before your code runs, so a single-node runtime starts faster.

WHERE THE OTHER RUNTIMES ARE DOCUMENTED

The DuckDB and Standard runtimes have their own guides. This guide only covers Spark.

1.3 How a run works

Compute size and workers

You choose a named compute size in the runtime picker. Each size sets the CPU and memory of the driver and of every worker.

Setting	Value	Notes
Minimum workers	Default 1	Set in the runtime picker
Maximum workers	10	Fixed per session
Scale-up trigger	70 percent	Average CPU or memory use across workers
Executor memory	65 percent of worker memory	Never more than worker memory minus 1 GiB

Workers are added automatically when CPU or memory use passes 70 percent, up to a maximum of 10 workers. Workers added during a heavy workload stay until the session ends. Spark releases idle executors after 120 seconds. To get more memory per executor, choose a larger compute size.

What happens when a job runs

1. Groundzero loads your script.
2. It replaces placeholders such as `{STRITM}` and `{SITENM}` in the code (see Placeholders in the API reference).
3. It runs the script. Standard output and standard error stream to the job log.
4. If the script defined a variable named `outputDF`, Groundzero reads up to 100,000 rows of it for the preview and shows the first page. Later pages are served without recomputing the result.
5. Groundzero records the final status and lineage, and sends the completion notification if one is configured.

Notebooks skip these steps. They start JupyterLab with a kernel that has already imported `sparkSession`, `pandas`, `numpy`, `pyspark` and `executeJob`.

Sessions and datasets

A dataset that uses the Spark runtime runs its query in a Spark session that is shared between users of the same dataset. One session handles one run at a time. When every session for the dataset is busy, Groundzero starts another instead of queueing your run. A session stays available for 280 seconds after its last run and shuts down after 300 seconds of inactivity.

Lakehouse permissions are checked as the user who starts each run, and the session clears its views when the user changes. One user never sees another user's governed rows.

KEEP A NOTEBOOK OR DATASET WARM

If your next run starts within a few minutes, it reuses the running session and skips the Spark start-up. After the idle window, the next run starts a new session.

1.4 Versions

Component	Version
Python	3.12
Apache Spark (and PySpark)	3.5.6
Java	OpenJDK 17
Apache Iceberg	1.11.0
Delta Lake	3.3.2
Apache Hudi	1.1.1
NumPy	2.3.5
pandas	2.2 or later, below 3
DuckDB (Python)	1.5.5
scikit-learn	1.5.2
PyTorch (CPU)	2.14.0
MLflow	2.20.4
Qualipilot (data quality)	3.5.0
Docling (PDF extraction)	2.116.0

Connector driver versions you may need when you read vendor documentation:

Connector	Version
PostgreSQL JDBC	42.7.11
MySQL	8.4.0
SQL Server JDBC	13.4.0
Oracle JDBC	23.6.0.24.10
Snowflake Spark / JDBC	3.1.7 / 3.28.0
MongoDB Spark	10.5.0
Kafka clients	3.4.1
Elasticsearch Spark	9.4.1
BigQuery Spark	0.44.0

1.5 What changed recently

The Spark runtime is updated continuously and has no numbered releases. Recent changes that affect your code, newest first:

- Bulk loads from Snowflake and Redshift into lakehouse tables (`bulkLoadGzLakehouseFromSnowflake` , `bulkLoadGzLakehouseFromRedshift`).
- Lineage is recorded for every connector read and write.
- Dataset previews of wide nested columns no longer run out of memory.
- Faster start-up for Spark sessions.
- The `*Native` REST methods are now named `*Arrow` .
- Qualipilot 3.5.0 data quality, PDF extraction with OCR, Agent calls over documents, an xAI connector and LLM completion across providers.

1.6 Machine-learning variants

Three machine-learning variants add libraries for model work. They keep every Spark and Groundzero feature. Choose a variant in the runtime picker when you need it.

Variant	Adds
scikit-learn	<code>opencv-python-headless</code> , <code>xgboost</code> , <code>lightgbm</code> , <code>catboost</code>
PyTorch	PyTorch 2.5.1 and torchvision 0.20.1 (CUDA 12.1 build), <code>shap</code> , <code>lime</code> , <code>albumentations</code> , <code>transformers</code> , <code>huggingface_hub</code> , <code>fastai</code> , <code>simsimd</code> , plus the scikit-learn set
TensorFlow	<code>tensorflow</code> , <code>nlTK</code> , <code>spacy</code> , <code>h5py</code> , <code>shap</code> , <code>lime</code> , <code>scipy</code> , <code>opencv</code> , <code>xgboost</code> , <code>lightgbm</code> , <code>catboost</code>

All variants include scikit-learn 1.5.2, NumPy and pandas. The PyTorch variant uses the CUDA 12.1 build of PyTorch, so it can use an NVIDIA GPU when you set a GPU size.

CHECK THE EXACT VERSIONS

The TensorFlow variant installs its packages without version pins. To see a version, run `import tensorflow as tf; print(tf.__version__)` (or the equivalent for any package) in a notebook on the variant.

For LLM fine-tuning libraries (`unsloth` , `trl` , `peft` , `bitsandbytes`), use a GPU variant and install them with `%pip` in a notebook (see Install and use extra packages).

2 Quickstart

This quickstart creates a Spark job that builds a small DataFrame, aggregates it, writes the result to object storage and exposes it as the job output. It uses fictional data so you can run it as soon as you have an S3 connection.

2.1 Before you begin

You need:

- A project in which you can create ETL jobs.
- Permission to use a compute size (the runtime picker lists the sizes you may use).
- An S3 connection named `acme-s3` (any object-store connection works with the matching `read / write` method; this example uses S3).

2.2 Step 1: Create the job

1. Open the project and create a new ETL job.
2. In the editor toolbar, open the runtime picker, choose a Spark runtime and a compute size.
3. Paste the code below.

PYTHON

```

from GZ.SparkUtils import sparkSession
from pyspark.sql import functions as F

spark = sparkSession("QuickstartOrders", "WARN")

orders = spark.createDataFrame(
    [
        (1001, "emea", "widget", 3, 19.99),
        (1002, "emea", "gadget", 1, 49.50),
        (1003, "apac", "widget", 10, 19.99),
        (1004, "amer", "gizmo", 2, 120.00),
        (1005, "apac", "gadget", 4, 49.50),
    ],
    ["order_id", "region", "product", "quantity", "unit_price"],
)

revenue = (
    orders.withColumn("revenue", F.col("quantity") * F.col("unit_price"))
    .groupBy("region")
    .agg(
        F.count("*").alias("orders"),
        F.round(F.sum("revenue"), 2).alias("revenue"),
    )
    .orderBy(F.desc("revenue"))
)

spark.writeS3(
    df=revenue,
    connName="acme-s3",
    filePath="quickstart/revenue_by_region",
    format="parquet",
    mode="overwrite",
)

outputDF = revenue

```

2.3 Step 2: Run and read the result

Select **Run**. The log panel streams the job log: Spark start-up messages first, then anything your script prints. When the job completes, the preview shows the rows of `outputDF`:

TEXT

region	orders	revenue
apac	2	397.9
amer	1	240.0
emea	2	109.47

The same `revenue_by_region` data now exists as Parquet files under `quickstart/revenue_by_region` in the bucket of the `acme-s3` connection.

WHY OUTPUTDF ?

The runtime looks for a top-level variable named `outputDF` after your script finishes. A Spark DataFrame, a pandas DataFrame or a list of dictionaries can all be used. If the variable is missing, the job still succeeds, but there is no preview.

2.4 Step 3: Try it in a notebook

Create a notebook, choose the same runtime, and run the first cell. The kernel has already imported `sparkSession`, `pandas as pd`, `numpy as np`, `pyspark` and `executeJob`:

PYTHON

```
spark = sparkSession("NotebookScratch", "WARN")
df = spark.createDataFrame([(1, "a"), (2, "b")], ["id", "label"])
df.show()
```

2.5 Step 4: Schedule it

From the job page, add a schedule. The scheduler dialog has a single **RTE** field that opens the same runtime and compute picker. Pick a runtime, a compute size, and save. Scheduled runs use the runtime and compute stored with the schedule.

In the UI

The runtime is chosen in the **runtime picker**, a compact control that shows the selected runtime name and, next to it, a small badge with the compute name. Selecting it opens a panel with these fields:

Field	Meaning
Runtime	The runtime to use: <code>Standard</code> (datasets only) or one of the registered runtimes
Compute	A named size; each entry lists its name, memory and CPU
Minimum Workers	Shown for Spark runtimes; the starting worker count, never below 1
GPU Size	Shown for every runtime except <code>Standard</code> ; defaults to 0

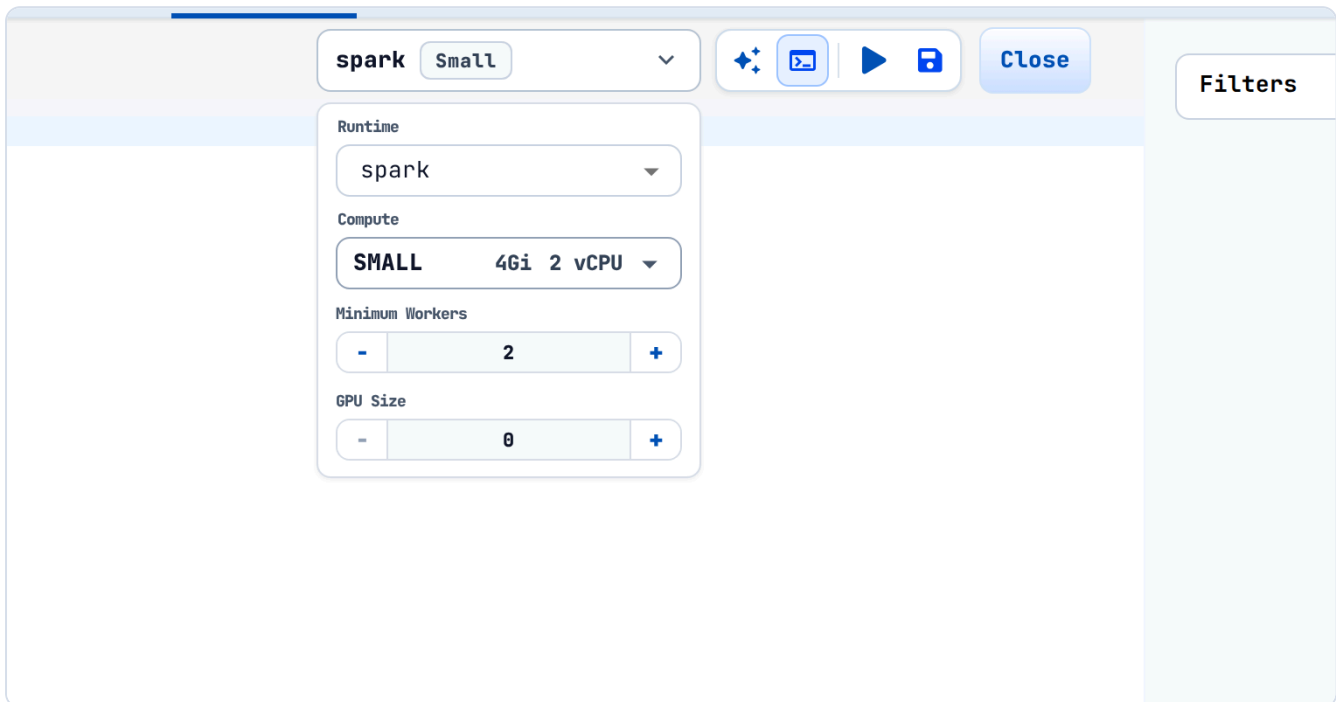
Where you find it:

- **ETL job editor.** The picker sits in the editor toolbar. `Standard` is not offered for jobs, so you choose a registered runtime directly.
- **Notebooks.** The picker is in the notebook header and also omits `Standard`. When the notebook's session is disconnected, the control turns red and shows "Session disconnected".
- **Datasets.** The dataset designer lists `Standard` first, followed by the registered runtimes. Choosing `Standard` hides the Compute field; any other runtime requires a compute size before **Save** is enabled.
- **Flow builder.** The picker sits in the top bar of the flow builder and, like the job editor, lists the registered runtimes.
- **Scheduler.** The scheduler dialog shows one **RTE** text field that displays the chosen runtime and compute together. Selecting it opens a menu with Runtime and Compute drop-downs and a **Save** button.

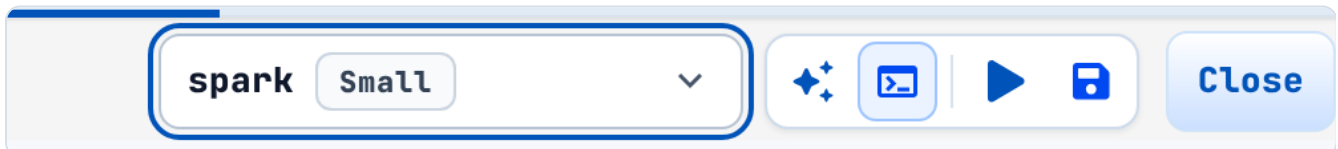
The Runtime list is loaded from your site, so it shows only the runtimes your administrator has registered. If the list is empty, the picker shows "No RTEs Available".

COMPUTE SIZES YOU CANNOT SEE

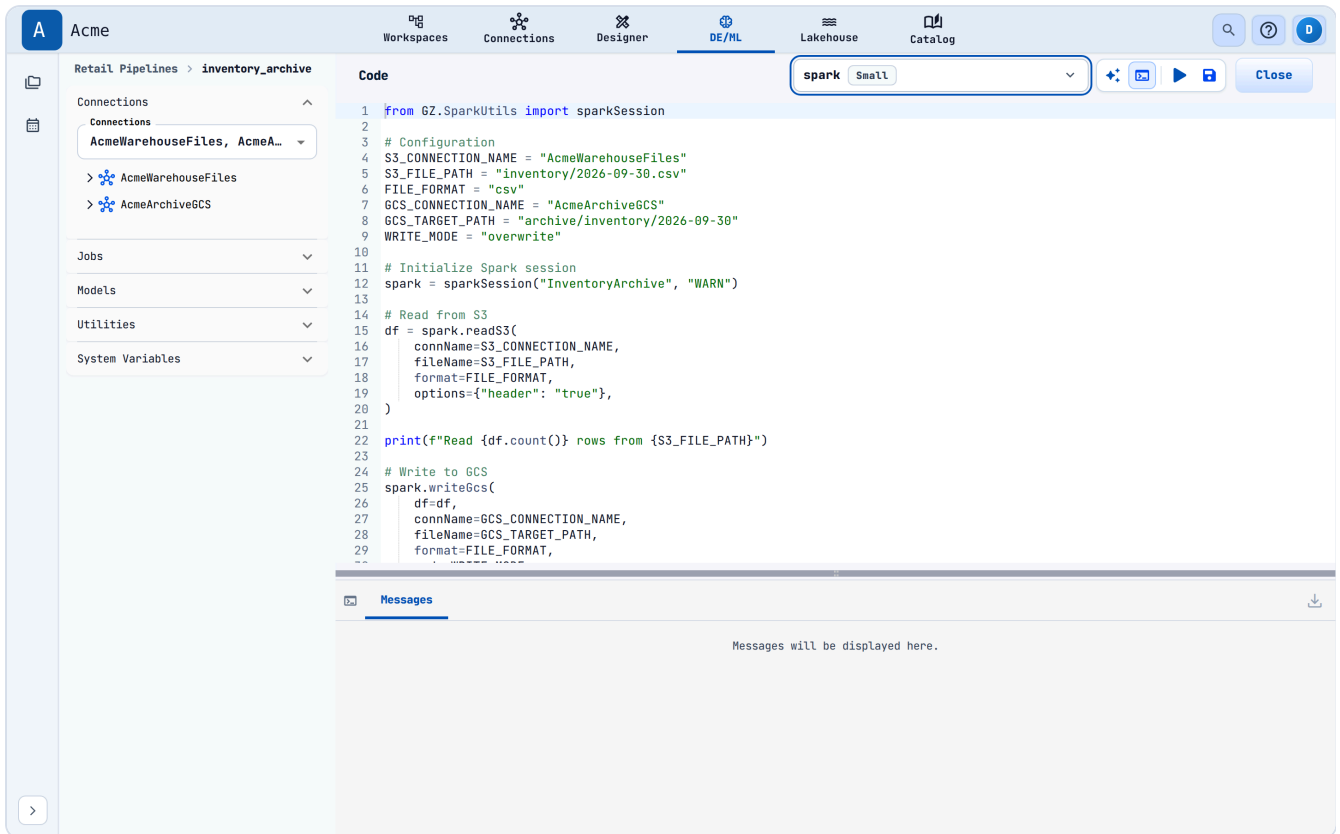
If the Compute field reads "Insufficient Permission", your user has no access to any compute size. Ask a site administrator to grant you access.



Runtime picker with Spark selected: Runtime, Compute, Minimum Workers and GPU Size.

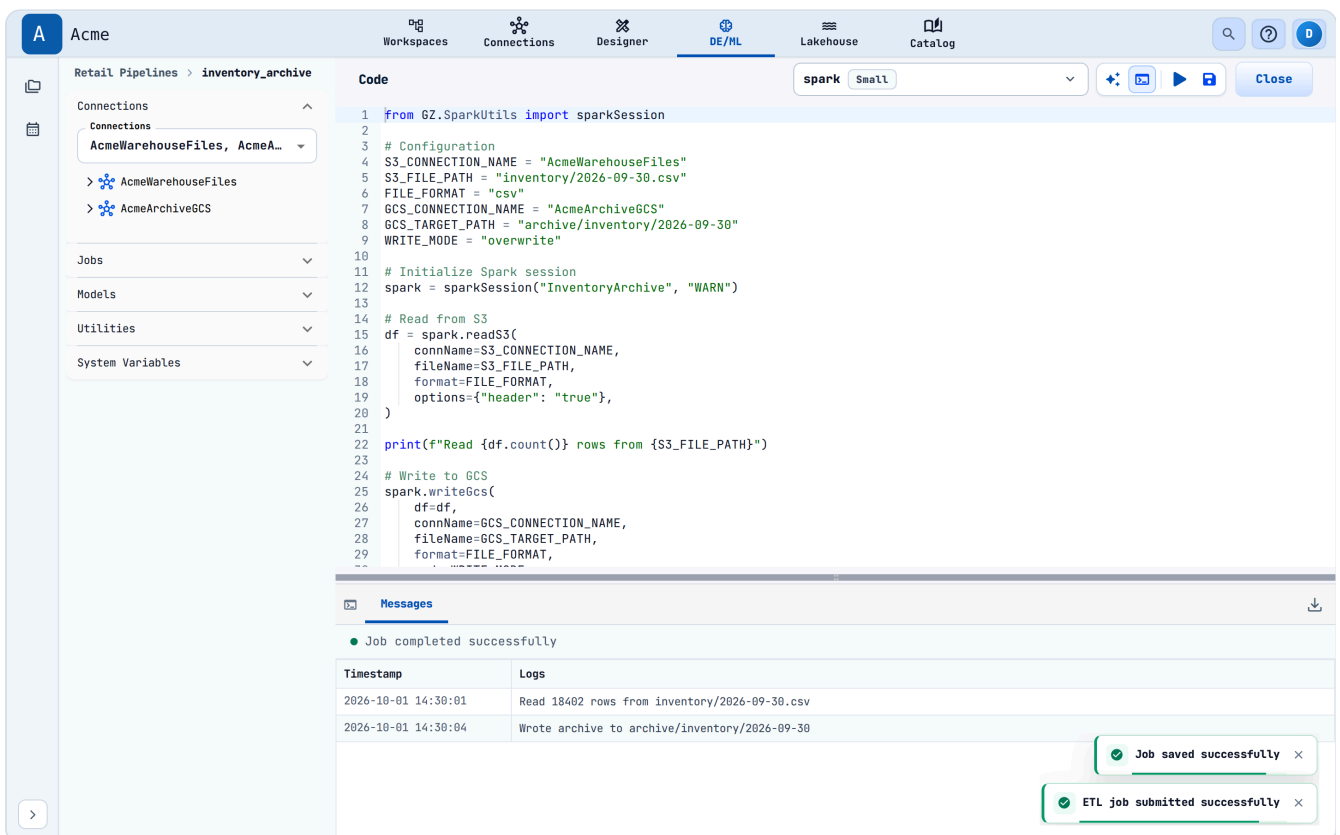


The closed picker shows the runtime name and a compute badge.



The screenshot shows the Groundzero job editor. On the left, the 'Connections' panel lists 'AcmeWarehouseFiles' and 'AcmeArchiveGCS'. The main editor displays a Python script for reading data from S3 and writing it to GCS. The 'DE/ML' tab is active, and the 'spark' runtime with 'Small' compute is selected. The 'Messages' tab at the bottom is currently empty, indicating no output has been generated yet.

The job editor with the Spark runtime and Small compute selected.



The screenshot shows the same job editor after a successful run. The 'Messages' tab now displays a 'Job completed successfully' message and a table of logs. Two toast notifications are visible: 'Job saved successfully' and 'ETL job submitted successfully'.

Timestamp	Logs
2026-10-01 14:30:01	Read 18402 rows from inventory/2026-09-30.csv
2026-10-01 14:30:04	Wrote archive to archive/inventory/2026-09-30

After a run, the Messages tab shows the job status and the timestamped logs.

New Task

×

Select Project*
Retail Pipelines ▼

Select Job*
inventory_archive ▼

RTE*
spark | Medium | 8Gi | 4

☒ Streaming ☐ Batch

Name*

Description

Start date
10/02/2026 

Start time
12:06 AM 

Cancel

Save

Scheduling a job: the RTE field records the runtime and compute as one value.

3 How-to guides

3.1 Read from a connection

Connections are created once in Groundzero (credentials stay in the connection) and referenced by name from code. Every connector adds `read*`, `execute*` and `write*` methods to the Spark session; the exact set depends on the connector (see the connector table in the API reference).

Pattern: the first argument to every read or execute method is `connName`, the name of the connection. For write methods the DataFrame comes first, followed by `connName`.

Object storage

PYTHON

```
from GZ.SparkUtils import sparkSession

spark = sparkSession("ReadSales")

raw = spark.readS3(
    connName="acme-s3",
    fileName="landing/sales/2026/",
    format="csv",
    options={"header": "true", "inferSchema": "true"},
)
raw.printSchema()
```

`readS3` builds the path `s3a://<bucket>/<fileName>` from the connection's bucket. Pass a schema as `schema=` to skip inference on large inputs.

Relational databases

All the JDBC-based connectors (PostgreSQL, MySQL, SQL Server, Oracle, Redshift, Teradata, ClickHouse, Databricks, Trino and others) share one reader:

PYTHON

```
orders = spark.readJDBC(
    connName="acme-postgres",
    query="SELECT order_id, region, amount FROM sales.orders "
    "WHERE order_date ≥ DATE '2026-01-01'",
)
```

Run DDL or DML (create, insert, update, delete) with `executeJDBC`:

PYTHON

```
spark.executeJDBC(
    connName="acme-postgres",
    query="DELETE FROM sales.staging_orders WHERE loaded = true",
)
```

PUSH WORK TO THE DATABASE

`readJDBC` sends your `query` to the database as a subquery, so filters and aggregations in the SQL run there. Select only the columns and rows you need before Spark sees them.

Snowflake, BigQuery and other warehouses

PYTHON

```
df = spark.executeSnowflake(
    connName="acme-snowflake",
    query="SELECT customer_id, SUM(total) AS spend FROM orders GROUP BY 1",
)
```

Most warehouse connectors follow the same `executeX(connName, query)` shape and return a `DataFrame`, plus a `writeX(df, connName, tableName, ...)` writer. See the connector table for the exact method names.

Streams

PYTHON

```
events = spark.readKafka(
    connName="acme-kafka",
    topic="orders.created",
    startingOffsets="earliest",
)
```

`readKafka` returns a streaming `DataFrame`. Streaming writers such as `writeKafka`, `writeStreamToS3` and `writeGzLakehouse(..., streaming=True)` require a `checkpointLocation`.

Other helpers for credentials

If you need the native client library rather than a Spark `DataFrame` (for example `boto3`, `pymongo` or a SQLAlchemy engine), use the client factory:

PYTHON

```
from GZ.Client import getClient

client = getClient()
s3 = client.getS3Client("acme-s3")
buckets = s3.list_objects_v2(Bucket="acme-data", MaxKeys=5)
```

Credentials are never visible to your script. The factory returns an object that is already authenticated.

3.2 Query and write Lakehouse (Iceberg) tables

Groundzero Lakehouse tables are Apache Iceberg tables. A **GZLAKEHOUSE** Groundzero Lakehouse tables are **Apache Iceberg tables**. A `GZLAKEHOUSE connection**` names the warehouse and the default database. Two methods cover most work: `executeQuery` to read with SQL, and `writeGzLakehouse` to append or overwrite. To check it worked, call `orders.show()` or read the table back with `executeQuery`.

Query with SQL

PYTHON

```
from GZ.SparkUtils import sparkSession

spark = sparkSession("LakehouseRead", siteName="acme")

orders = spark.executeQuery(
    connName="acme-lakehouse",
    sqlQuery="""
        SELECT region, SUM(amount) AS total
        FROM sales.orders
        GROUP BY region
    """,
)
orders.show()
```

Always qualify tables as `database.table`. The database name must be one you can see in the Lakehouse pages.

NAME THE SITE FOR LAKEHOUSE CONNECTIONS

A lakehouse connection routes every call to a site. Groundzero normally finds your site by itself. If it cannot, pass `siteName="your-site"` to `sparkSession` (applies to all such connections), pass `siteNames={"connection-name": "site"}` for one site per connection, or pass `siteName=` to the individual call. Otherwise the call raises `ValueError: siteName is required for GZLAKEHOUSE connection`.

Statements that only read data (`SELECT`, `SHOW`, `DESCRIBE`, `EXPLAIN`) need no extra check. Statements that change data are checked against your permission on each table first.

Write a table

PYTHON

```
daily = orders.withColumnRenamed("total", "daily_total")

spark.writeGzLakehouse(
    df=daily,
    connName="acme-lakehouse",
    tableName="orders_daily",
    database="sales",
    mode="append",
)
```

For connections with a lakehouse endpoint, `mode` must be `append` or `overwrite`. `overwrite` replaces the snapshot of an existing table. External (read-only) lakehouse connections reject all writes.

Stream into a table

PYTHON

```
query = spark.writeGzLakehouse(
    df=events_df,
    connName="acme-lakehouse",
    tableName="orders_stream",
    database="sales",
    streaming=True,
    checkpointLocation="s3://acme-data/checkpoints/orders_stream",
    outputMode="append",
    trigger={"availableNow": True},
)
```

Create the target table, including its partition specification, before streaming into it. `partitionBy` is rejected for streaming writes.

Check your connection first

`verifyLakehouseConnection` checks that the connection reaches the expected warehouse. It turns a routing problem into an immediate, readable error:

PYTHON

```
info = spark.verifyLakehouseConnection("acme-lakehouse", siteName="acme")
print(info["warehouse"], info["site_used"])
```

Bulk-load from Snowflake or Redshift

For large one-time loads, let the source warehouse unload Parquet files directly into the lakehouse table's storage and register them with one Iceberg commit. No row passes through Spark:

PYTHON

```
summary = spark.bulkLoadGzLakehouseFromSnowflake(
    connName="acme-snowflake",
    query="SELECT * FROM analytics.orders WHERE order_date ≥ '2026-01-01'",
    lakehouseConn="acme-lakehouse",
    tableName="orders_history",
    database="sales",
    mode="overwrite",
    siteName="acme",
)
print(summary)
```

3.3 Work with outputs and previews

The runtime turns your result into a preview that the UI pages through.

- Assign the DataFrame you want to show to the top-level variable `outputDF`.
- For a Spark DataFrame, the runtime reads up to 100,000 rows. If your result has more rows, the preview is the first 100,000.
- A pandas DataFrame is converted to a list of records. Any list of dictionaries also works.
- The schema shown in the preview is taken from `outputDF`.

- When the same code runs again with the same inputs within a short period, the runtime may serve the preview from its cache. Results from governed lakehouse tables are cached per caller policy, so users share cached rows only when their policies allow it.

PYTHON

```
import pandas as pd

summary = pd.DataFrame(
    {"metric": ["rows", "regions"], "value": [1_250_000, 4]}
)
outputDF = summary
```

KEEP PREVIEWS FAST

Order and limit the result before you assign it to `outputDF`. A deterministic plan without shuffles lets the runtime return the first page without reading the whole preview. Joins, aggregates and sorts need the whole result.

Dataset runs page the preview with a page size that defaults to 1,000 rows and is capped at 10,000.

3.4 Use parameters and placeholders

Groundzero replaces a fixed set of placeholders in your code with run-time values before the script executes. They are replaced as text, so write them where a Python string expression is valid.

PYTHON

```
from GZ.SparkUtils import sparkSession

spark = sparkSession("Parameters")

run_started = {STRTTM}
site = {SITENM}
print("Started:", run_started, "for site", site)
```

The line `run_started = {STRTTM}` becomes `run_started = "'2026-10-01 09:30:00'"`. The substituted value is already quoted for SQL, so it can be concatenated into a statement:

PYTHON

```
sql = "SELECT * FROM sales.orders WHERE created_at ≥ " + {STRTTM}
df = spark.executeQuery("acme-lakehouse", sql)
```

The placeholders are listed in the API reference. The `${NAME}` spelling is also accepted.

PLACEHOLDERS ARE NOT PYTHON F-STRING FIELDS

Write `"... ≥ " + {STRTTM}`, not `f"... ≥ {STRTTM}"`. Inside a string literal the replacement text would break the quoting.

3.5 Chain jobs

A job that was submitted with sub-jobs can run them in sequence from code using `executeJob` :

PYTHON

```
from GZ.JobUtils import executeJob

results = executeJob(["load_orders", "build_marts"])
print(list(results))
```

Each sub-job runs with the same restrictions as a normal job. If one fails, `executeJob` raises and the remaining jobs do not run.

3.6 Schedule through ETL jobs

1. Save the job with its runtime and compute.
2. Open the scheduler for the job and set the frequency, start and stop times.
3. Choose the RTE with the runtime picker.

If you set a stop time, the session ends at that time and aborts any job still running. Use a stop time as a safety cap for long-running streaming jobs.

When a job finishes, Groundzero sends the notification configured for the job's status (success or failure), if there is one. Its subject and body can use the same placeholders. From code, you can also send your own mail (see `sendEmail`).

3.7 Install and use extra packages

Use this when your code needs a library that the runtime does not include. Check Installed packages first, because most common libraries are already there.

Need	Route
Try a package in a notebook	<code>%pip install name</code> in a notebook cell
Machine-learning libraries	Pick the scikit-learn, PyTorch or TensorFlow variant
Packages for every job	A custom runtime that extends a Spark runtime

Notebook installs

In a notebook you can install a package for the life of the session:

TEXT

```
%pip install rapidfuzz
```

To check it worked, `import rapidfuzz` in the next cell. The package disappears when the session ends. Job scripts run with restrictions, so do not install packages from inside a job script.

Custom runtimes

A custom runtime is a Spark runtime with extra Python packages added. Administrators create it on the **Custom RTE** page:

1. Prepare a `requirements.txt` with one pip requirement per line.
2. Select **Add custom RTE**, choose the **Base Image** to extend (for example a Spark variant), give the runtime a **Name**, and upload the file.
3. Wait for the build. The table on the page shows each runtime's name, creator, creation date and build **Status**.
4. When the status shows success, choose the new runtime in the runtime picker.

An example `requirements.txt`:

TEXT

```
splink
sqlparse
s3fs
```

PICK THE CLOSEST BASE

Extend the Spark variant that already has most of what you need (for example the scikit-learn variant for model work) so you only add the difference.

If the build fails, the **Status** column shows the failure. Check that each package name and version in `requirements.txt` exists and supports Python 3.12.

3.8 Tune performance

Partitions and shuffles

Spark runs one task per partition. Match the partition count to the cores you have: too few leaves workers idle, too many wastes time on scheduling.

PYTHON

```
from GZ.SparkUtils import sparkSession

spark = sparkSession("Tuned", **{"spark.sql.shuffle.partitions": "64"})

df = spark.executeQuery("acme-lakehouse", "SELECT * FROM sales.orders")
df = df.repartition(64, "region")
```

Any keyword argument you pass to `sparkSession` is applied as a Spark setting. Setting names contain dots, so pass them with `**{ ... }` as shown. Settings apply only when the session is first created. A reused session ignores them.

Defaults the runtime already sets:

Setting	Value
<code>spark.serializer</code>	Kryo, buffer up to 1 GB

Setting	Value
<code>spark.driver.maxResultSize</code>	0 (unlimited)
<code>spark.network.timeout</code>	600 seconds
<code>spark.sql.broadcastTimeout</code>	600 seconds
<code>spark.sql.adaptive.enabled</code>	true when a lakehouse connection exists
<code>spark.dynamicAllocation.enabled</code>	true, executors idle out after 120 seconds

Memory

- Executor memory is set from the compute size (65 percent of worker memory, at most worker memory minus 1 GiB). To get more, pick a larger compute size.
- Avoid `collect()` and `toPandas()` on large DataFrames: the data lands in the driver's memory. Aggregate or limit first.
- Cache a DataFrame (`df.cache()`) only when several actions reuse it, and release it with `df.unpersist()` .

THE DRIVER HAS NO RESULT LIMIT

`spark.driver.maxResultSize` is 0, so an oversized `collect()` fails with an out-of-memory error instead of a friendly limit message.

Arrow

Arrow speeds up transfers between Spark and Python.

- Use the `*Arrow` connector methods (for example `readRESTArrow` , `readSalesforceArrow` , `readSFTPArrow`) for Python-side API connectors. They fetch and write through Arrow tables, and many accept a `columns` list to prune data.
- Enable Arrow for `toPandas()` and pandas UDFs with `spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", "true")` .
- Dataset previews of nested columns are read through Arrow for you, so you need no setting for that.

Read less

- Select columns and filter early; Iceberg and Parquet push both down.
- Prefer Parquet or Iceberg over CSV for intermediate data.
- Pass an explicit `schema=` to `readS3` to skip schema inference.

3.9 Log from your job

Everything your script writes to standard output or standard error is captured and shown in the job log, tagged with the job's execution identifier. Use `print` or the standard `logging` module.

PYTHON

```
import logging

log = logging.getLogger("orders-job")
log.warning("Loaded %d rows", orders.count())
print("done")
```

`sparkSession(appName, logLevel)` sets the Spark log level (`ALL` , `DEBUG` , `INFO` , `WARN` , `ERROR` , `FATAL` , `OFF` , `TRACE`). The default is `WARN` . Errors raised by your script are reported with the exception type, message and the line number in your code, for example `ValueError: bad region at line 14` .

The runtime also collects Spark metrics for each run (tasks, stages, shuffle and spill, GC time, input and output records) and attaches them to the execution so you can inspect them with the run details. No code is needed for that.

3.10 Check data quality

Qualipilot runs deterministic checks inside Spark and, optionally, sends a compact aggregate to an LLM connection for a narrative summary. Row samples and top values are never sent.

PYTHON

```
from GZ.SparkUtils import sparkSession
from qualipilot import CheckConfig, QualipilotConfig
from qualipilot.models.config import ColumnRange

spark = sparkSession("QualityCheck")
orders = spark.createDataFrame(
    [(1001, "ana@example.com", 125.0), (1002, None, -25.0)],
    ["order_id", "customer_email", "amount"],
)
config = QualipilotConfig(
    checks=CheckConfig(
        required_columns=["order_id", "customer_email", "amount"],
        column_ranges={"amount": ColumnRange(min=0, max=10_000)},
    )
)
report = spark.runDataQualityChecks(
    df=orders, connectionName="acme-openai", config=config
)
print(report.to_json())
spark.saveDataQualityReport(report, "reports/orders.html", "html")
```

3.11 Parse PDFs and run Agents over documents

`parsePDF` extracts one document from S3, `parsePDFCorpus` extracts a folder restartably, and `invokeAgentPDFCorpus` runs a Groundzero Agent over every document in a prefix. Corpus runs are content-addressed: re-running the same source and output prefix skips page ranges that are already committed.

PYTHON

```
result = spark.parsePDFCorpus(  
    s3ConnName="acme-s3",  
    sourcePrefix="incoming/policies/",  
    outputS3ConnName="acme-s3",  
    outputPrefix="pdf-results/policies",  
    pageRangeSize=64,  
)  
print(result["status"], result["documentCount"], result["manifestUri"])
```

See the [API reference](#) for the full parameter list and the [examples gallery](#) for an end-to-end Agent run.

4 API reference

This section documents the Groundzero helper API that is available in the Spark runtime. Standard PySpark (DataFrame, functions, types, MLlib) is documented by Apache Spark and is not repeated here.

4.1 Conventions

- Import the session factory once: `from GZ.SparkUtils import sparkSession`.
- The object it returns (called `spark` in this guide) is the **Groundzero session**. Every connector method is a method on it.
- Connections are always referenced by `connName`, the connection name in Groundzero. Credentials never appear in code.
- Read and execute methods return a Spark DataFrame. Write methods take the DataFrame as the first argument (`df`) and generally return `None`.
- Method and parameter names use camelCase and are case sensitive.
- Argument names in the signatures below are the real parameter names; pass them by keyword for readability.

PARAMETERS NOT DESCRIBED HERE

Parameters that a method accepts but that this guide does not describe are passed straight through to the underlying connector.

4.2 Session

sparkSession

PYTHON

```
sparkSession(appName, logLevel="WARN", siteName=None, siteNames=None, **kwargs)
```

Creates (or returns) the Groundzero session for the current run.

Parameter	Type	Default	Description
<code>appName</code>	<code>str</code>	required	Application name shown in Spark and the logs
<code>logLevel</code>	<code>str</code>	<code>"WARN"</code>	Spark log level, such as <code>INFO</code> or <code>ERROR</code>
<code>siteName</code>	<code>str</code>	<code>None</code>	Site used for every REST-backed lakehouse connection
<code>siteNames</code>	<code>dict</code>	<code>None</code>	Mapping of connection name to site
<code>**kwargs</code>	<code>any</code>	<code>none</code>	Extra Spark settings, for example <code>**{"spark.x": "y"}</code>

`siteNames` also accepts a list of site names (a single entry applies to every REST-backed connection; a list of the same length maps by connection order) or a list of dictionaries with `connName` (or `connectionName`, `connection`, `name`) and `siteName` (or `site`).

Returns. The Groundzero session object.

Raises. `ValueError` if you try to move a lakehouse connection to a different site in the same session:

`GZLAKEHOUSE connection 'x' is already configured for site 'a'; cannot reconfigure it to 'b' in the same Spark session.`

Behavior:

- Calling `sparkSession` again in the same notebook or reused session returns the existing session. Extra Spark settings in `**kwargs` take effect only on the first call.
- Your settings are applied on top of the Groundzero defaults (see Tune performance).
- Lakehouse catalogs are registered for every GZLAKEHOUSE connection in the run.

PYTHON

```
spark = sparkSession(
    "DailyOrders",
    "INFO",
    siteNames={"acme-lakehouse": "acme", "partner-lakehouse": "partner"},
)
```

Session object

The session is a restricted wrapper around a Spark session.

Member	Behavior
<code>spark.sql(sqlQuery, *args, **kwargs)</code>	Runs Spark SQL; catalog and policy checks apply
<code>spark.createDataFrame(...)</code>	Passed through to Spark
<code>spark.table(...)</code> , <code>spark.range(...)</code>	Passed through to Spark
<code>spark.conf</code> , <code>spark.catalog</code> , <code>spark.udf</code>	Passed through to Spark
<code>spark.read</code> , <code>spark.write</code>	Blocked; use the connector methods
<code>spark.sparkContext</code>	Blocked

Accessing a blocked member raises `AttributeError: 'sparkSession' object has no attribute 'read'`. Use the connector's `read*` method instead. On a `DataFrame`, `df.write` works as in Spark, but connectors (which attach credentials, authorization, audit and lineage) are the supported route.

4.3 Lakehouse (Iceberg) methods

executeQuery

PYTHON

```
spark.executeQuery(connName, sqlQuery, siteName=None)
```

Runs SQL against a GZLAKEHOUSE connection and returns the result.

Parameter	Type	Default	Description
<code>connName</code>	<code>str</code>	required	GZLAKEHOUSE connection name
<code>sqlQuery</code>	<code>str</code>	required	SQL; reference tables as <code>database.table</code>
<code>siteName</code>	<code>str</code>	None	Site to route to; falls back to the session site

Returns. A Spark DataFrame.

Raises.

- `PermissionError`: authentication failed, the user is not authorized for a referenced table, or the connection is external (read-only) and the statement is not `SELECT`, `SHOW`, `DESCRIBE` or `EXPLAIN`.
- `ConnectionError`: the lakehouse endpoint could not be reached.
- `RuntimeError`: the endpoint returned a non-success status, or routed to a warehouse that does not match the connection.
- `ValueError`: the site is unknown or the connection has no warehouse name.

Every call is recorded in the lakehouse audit log (success or failure, with the duration), and lineage is reported for the tables it touches.

PYTHON

```
df = spark.executeQuery(
    connName="acme-lakehouse",
    sqlQuery="SELECT id, status FROM sales.orders WHERE status = 'open'",
    siteName="acme",
)
```

writeGzLakehouse

PYTHON

```
spark.writeGzLakehouse(
    df, connName, tableName, database=None, mode="overwrite",
    partitionBy=None, options=None, siteName=None, streaming=False,
    checkpointLocation=None, outputMode="append", triggerTime=None,
    trigger=None,
)
```

Writes a DataFrame to a lakehouse table.

Parameter	Type	Default	Description
<code>df</code>	DataFrame	required	Data to write; streaming frames start a stream
<code>connName</code>	<code>str</code>	required	GZLAKEHOUSE connection name
<code>tableName</code>	<code>str</code>	required	Target table
<code>database</code>	<code>str</code>	None	Target database; defaults to the connection's
<code>mode</code>	<code>str</code>	"overwrite"	<code>append</code> or <code>overwrite</code> (endpoint connections)
<code>partitionBy</code>	<code>list</code>	None	Partition columns; batch writes on legacy connections only

Parameter	Type	Default	Description
<code>options</code>	<code>dict</code>	<code>None</code>	Extra writer options, such as <code>fanout-enabled</code> for streams
<code>siteName</code>	<code>str</code>	<code>None</code>	Site to route to
<code>streaming</code>	<code>bool</code>	<code>False</code>	Start a Structured Streaming write
<code>checkpointLocation</code>	<code>str</code>	<code>None</code>	Required for streaming writes
<code>outputMode</code>	<code>str</code>	<code>"append"</code>	<code>append</code> or <code>complete</code> for streams
<code>triggerTime</code>	<code>str</code>	<code>None</code>	Processing-time trigger such as <code>"1 minute"</code>
<code>trigger</code>	<code>dict</code>	<code>None</code>	Trigger arguments such as <code>{"availableNow": True}</code>

Returns. For a batch write to a lakehouse endpoint, a `dict` with `status`, `table`, `warehouse`, `catalog`, `site`, `mode` and `via`. For a stream, the running streaming query.

Raises.

- `ValueError`: `mode` is not `append` or `overwrite`; `database` is missing; the site is missing; `partitionBy` is used with a stream.
- `PermissionError`: the user may not write the table, or the connection is external.
- `ConnectionError`, `RuntimeError`: the lakehouse endpoint is unreachable or rejected the commit.

A `DataFrame` whose `isStreaming` flag is true is written as a stream even when `streaming` is not set. The write is authorized per table, audited, and its schema and lineage are captured.

getLakehouseCatalog

PYTHON

```
spark.getLakehouseCatalog(connName, siteName)
```

Returns the Spark catalog alias for a REST-backed connection and site, so you can write fully qualified SQL for `spark.sql`.

Parameter	Type	Default	Description
<code>connName</code>	<code>str</code>	required	GZLAKEHOUSE connection name
<code>siteName</code>	<code>str</code>	required	Site that fixes the catalog

Returns. `str`, the catalog alias. **Raises.** `ValueError` if the connection has no lakehouse endpoint.

PYTHON

```
catalog = spark.getLakehouseCatalog("acme-lakehouse", "acme")
spark.sql(f"SELECT COUNT(*) FROM {catalog}.sales.orders").show()
```

setLakehouseSite

PYTHON

```
spark.setLakehouseSite(connName, siteName)
```

Registers the catalog for a site ahead of use. It does not change the active catalog. **Returns.** `None`. **Raises.** `ValueError` if the connection has no warehouse name or no lakehouse endpoint.

verifyLakehouseConnection

PYTHON

```
spark.verifyLakehouseConnection(connName, siteName=None)
```

Probes the lakehouse endpoint and checks that it routes to the connection's warehouse. The probe is anonymous: it checks reachability and routing, not authorization.

Returns. `dict` with the keys `provider_url`, `warehouse`, `site_used` and `prefix`. **Raises.** `ConnectionError`, `RuntimeError` or `ValueError` with a descriptive message.

Bulk loads into the lakehouse

PYTHON

```
spark.bulkLoadGzLakehouseFromSnowflake(
    connName, query, lakehouseConn, tableName, database=None,
    mode="overwrite", warehouseSize=None, storageIntegration=None,
    stagingLocation=None, maxFileSize=DEFAULT_MAX_FILE_SIZE_BYTES,
    siteName=None,
)
spark.bulkLoadGzLakehouseFromRedshift(
    connName, query, lakehouseConn, tableName, database=None,
    mode="overwrite", iamRole=None, stagingLocation=None,
    maxFileSize=DEFAULT_MAX_FILE_SIZE_BYTES, siteName=None,
)
```

The warehouse unloads the query result as Parquet under the target table's location, and the files are registered with one Iceberg commit. No row passes through Spark. The default for `maxFileSize` is 256 MiB.

Parameter	Type	Default	Description
<code>connName</code>	<code>str</code>	required	Source Snowflake or Redshift connection
<code>query</code>	<code>str</code>	required	Query whose rows are loaded
<code>lakehouseConn</code>	<code>str</code>	required	GZLAKEHOUSE connection of the target
<code>tableName</code>	<code>str</code>	required	Target table; created from the file schema if missing
<code>database</code>	<code>str</code>	<code>None</code>	Target database; defaults to the connection's
<code>mode</code>	<code>str</code>	<code>"overwrite"</code>	<code>overwrite</code> replaces the rows, <code>append</code> adds
<code>warehouseSize</code>	<code>str</code>	<code>None</code>	Snowflake only; warehouse size during the unload
<code>storageIntegration</code>	<code>str</code>	<code>None</code>	Snowflake only; storage integration to write with
<code>iamRole</code>	<code>str</code>	<code>None</code>	Redshift only; role to unload with
<code>stagingLocation</code>	<code>str</code>	<code>None</code>	Prefix inside the table location for the files
<code>maxFileSize</code>	<code>int</code>	256 MiB	Upper bound in bytes per unloaded file

Parameter	Type	Default	Description
siteName	str	None	Site to route to

Returns. A summary of the rows and files registered, or a summary with `via: "spark_write"` when the files had to be written with Spark. **Raises.** `ValueError` if the Redshift method is called with a connection that is not a Redshift connection.

4.4 Object storage and files

readS3

PYTHON

```
spark.readS3(connName, fileName, format="parquet", schema=None, options=None)
```

Parameter	Type	Default	Description
connName	str	required	S3 connection name
fileName	str	required	Key or prefix inside the connection's bucket
format	str	"parquet"	Any Spark format: <code>csv</code> , <code>json</code> , <code>parquet</code> , <code>orc</code>
schema	schema	None	Spark schema; skips inference
options	dict	None	Reader options, for example <code>{"header": "true"}</code>

Returns. A DataFrame. **Raises.** `Exception("Error reading file from S3: ...")` wrapping the underlying cause.

writeS3

PYTHON

```
spark.writeS3(df, connName, filePath, format, mode, options=None, coalesce=False)
```

Parameter	Type	Default	Description
df	DataFrame	required	Data to write
connName	str	required	S3 connection name
filePath	str	required	Target key or prefix
format	str	required	Output format
mode	str	required	<code>overwrite</code> , <code>append</code> , <code>ignore</code> or <code>errorifexists</code>
options	dict	None	Writer options
coalesce	bool	False	Write a single file (<code>df.coalesce(1)</code>)

Returns. `None`.

COALESCE FUNNELS ALL DATA THROUGH ONE TASK

`coalesce=True` makes one executor write the entire result. Use it only for small outputs.

writeStreamToS3

PYTHON

```
spark.writeStreamToS3(df, connName, filePath, format, outputMode,
                      checkpointLocation, coalesce=False, options=None)
```

Writes each micro-batch of a streaming DataFrame to S3 in overwrite mode. **Returns.** The running streaming query.

4.5 Relational databases and warehouses

readJDBC

PYTHON

```
spark.readJDBC(connName, query, options=None)
```

Reads the result of `query` from any JDBC connection (PostgreSQL, MySQL, SkySQL, Oracle, SQL Server, Redshift, Teradata, IBM Db2, SAP and SAP HANA, Synapse, CockroachDB, TimescaleDB, ClickHouse, Databricks, Trino, Presto, Vertica, Dremio).

Parameter	Type	Default	Description
<code>connName</code>	<code>str</code>	required	JDBC connection name
<code>query</code>	<code>str</code>	required	SQL sent to the database
<code>options</code>	<code>dict</code>	<code>None</code>	Extra Spark JDBC options, such as <code>fetchsize</code>

Returns. A DataFrame. **Raises.** `ValueError("Connection x not found or is not a JDBC type.")`. TLS options for Redshift are enforced by the runtime.

writeJDBC

PYTHON

```
spark.writeJDBC(df, connName, tableName, mode, options=None)
```

Parameter	Type	Default	Description
<code>df</code>	DataFrame	required	Data to write
<code>connName</code>	<code>str</code>	required	JDBC connection name
<code>tableName</code>	<code>str</code>	required	Target table
<code>mode</code>	<code>str</code>	required	<code>append</code> , <code>overwrite</code> , <code>ignore</code> or <code>errorifexists</code>
<code>options</code>	<code>dict</code>	<code>None</code>	Extra Spark JDBC options

Raises. `ValueError` for an unknown connection, and `TypeError("Use writeSnowflake() for connection x.")` for Snowflake connections.

executeJDBC

PYTHON

```
spark.executeJDBC(connName, query)
```

Runs a DDL or DML statement through the JDBC driver and commits it. Prints `DDL statement executed successfully!` and returns `None`. Errors are printed and re-raised.

Snowflake

PYTHON

```
spark.executeSnowflake(connName, query, options=None)
spark.writeSnowflake(df, connName, tableName, mode="append", extraOptions=None)
spark.executeSnowflakeStatement(connName, statement)
```

Method	Returns
<code>executeSnowflake</code>	A <code>DataFrame</code> with the query result
<code>writeSnowflake</code>	<code>None</code> ; repartitions to at least 200 partitions first
<code>executeSnowflakeStatement</code>	The string <code>Snowflake statement executed successfully.</code>

`options` and `extraOptions` are merged into the connector options. Use `executeSnowflakeStatement` for DDL and session statements.

4.6 LLM methods

complete

PYTHON

```
spark.complete(connName, df, model=None, promptColumn="prompt",
               promptTemplate=None, systemPrompt=None, systemPromptColumn=None,
               outputColumn="llm_response", temperature=0.0, maxTokens=None,
               responseFormat=None, continueOnError=False, timeout=120,
               **providerOptions)
```

Adds a completion to every row of a `DataFrame` using the provider of the connection (OpenAI, Azure OpenAI, Anthropic, AWS Bedrock, Cohere, Gemini, Fireworks AI, Together AI, Hugging Face and xAI). The calls run inside the executors, so the work is distributed and no rows are collected to the driver.

Parameter	Type	Default	Description
<code>connName</code>	<code>str</code>	required	LLM connection name
<code>df</code>	<code>DataFrame</code>	required	Input rows
<code>model</code>	<code>str</code>	<code>None</code>	Model; for Azure the deployment name; for Bedrock the model ID

Parameter	Type	Default	Description
<code>promptColumn</code>	<code>str</code>	<code>"prompt"</code>	Column holding the prompt text
<code>promptTemplate</code>	<code>str</code>	<code>None</code>	Template with <code>{{column}}</code> or <code>\${column}</code> fields
<code>systemPrompt</code>	<code>str</code>	<code>None</code>	System prompt applied to every row
<code>systemPromptColumn</code>	<code>str</code>	<code>None</code>	Column with a per-row system prompt
<code>outputColumn</code>	<code>str</code>	<code>"llm_response"</code>	Name of the response column
<code>temperature</code>	<code>float</code>	<code>0.0</code>	Sampling temperature
<code>maxTokens</code>	<code>int</code>	<code>None</code>	Response limit
<code>responseFormat</code>	<code>str</code>	<code>None</code>	Provider response format, for example <code>json</code>
<code>continueOnError</code>	<code>bool</code>	<code>False</code>	Record per-row errors instead of failing
<code>timeout</code>	<code>int</code>	<code>120</code>	Request timeout in seconds (not used by Bedrock)

`providerOptions` accepts provider-only settings, such as `useConverse` for Bedrock or `returnFullText` for Hugging Face.

Returns. The input DataFrame plus these columns, where `<out>` is `outputColumn : <out>`, `<out>_status` (`success` or `error`), `<out>_error`, `<out>_provider`, `<out>_model`, `<out>_prompt_tokens`, `<out>_completion_tokens`, `<out>_total_tokens` and `<out>_latency_ms`.

Raises. `ValueError` if the connection is unknown (`Connection 'x' was not found.`), has an unsupported type (`Connection 'x' has unsupported LLM type 'y'. Supported types: ...`), a referenced column is missing (`Missing columns: [...]. Available: [...]`), or a Bedrock call has no model (`AWS Bedrock model must be set in the connection or call.`). With `continueOnError=False` the first failing row fails the job.

Requests are retried with exponential backoff on HTTP 429 and 5xx responses.

PYTHON

```
tickets = spark.createDataFrame(
    [(1, "Printer jams on page two"), (2, "Cannot reset my password")],
    ["id", "text"],
)
labeled = spark.complete(
    connName="acme-openai",
    df=tickets,
    promptTemplate="Label the ticket as hardware or account: {{text}}",
    outputColumn="label",
    maxTokens=8,
    continueOnError=True,
)
```

Fine-tuning and batch operations

Provider-neutral entry points route by connection type:

PYTHON

```
spark.fineTune(connName, df, **kwargs)
spark.distill(connName, df, **kwargs)
spark.listFineTuneJobs(connName, **kwargs)
spark.getFineTuneJob(connName, **kwargs)
spark.cancelFineTuneJob(connName, **kwargs)
spark.listModels(connName, **kwargs)
```

Each forwards its arguments to the provider method named by the operation plus the provider suffix, for example `fineTuneOpenAI`, `fineTuneAwsBedrock` or `listModelsTogetherAI`. For AWS Bedrock, cancel maps to `stopFineTuneJobAwsBedrock`. If the provider does not implement the operation, a `NotImplementedError` is raised: `fineTune is not available for connection type 'x'`.

Provider suffix	Typical methods
OpenAI, AzureOpenAI	<code>fineTune</code> , <code>distill</code> , jobs, batch jobs, deployments
AwsBedrock	<code>fineTune</code> , <code>distill</code> , jobs (<code>stop</code>), batch jobs, deployments
TogetherAI, FireworksAI	<code>fineTune</code> , jobs, batch jobs, deployments
Cohere, Gemini, HuggingFace, Anthropic	<code>complete</code> , models, batch jobs, deployments
Unsloth	<code>fineTuneUnsloth</code> , <code>listModelsUnsloth</code> (GPU runtime)
XAI	<code>completeXAI</code> , <code>listModelsXAI</code>

Direct fine-tuning is unavailable for Anthropic, Cohere, the Gemini API and the generic Hugging Face connector. Use AWS Bedrock for eligible Claude models, Vertex AI for Gemini tuning, and Unsloth or AutoTrain for Hugging Face models. OpenAI and Azure OpenAI require at least 10 training examples.

The batch method families follow the same naming: `createBatchJob<Provider>`, `listBatchJobs<Provider>`, `getBatchJob<Provider>`, `cancelBatchJob<Provider>` and `getBatchResults<Provider>`. Deployment methods follow `createDeployment<Provider>`, `listDeployments<Provider>`, `getDeployment<Provider>` and `deleteDeployment<Provider>`. Which ones exist for a provider is listed in the table above; the method listing for each connector is in the connector reference below.

4.7 Data quality

PYTHON

```
spark.createDataQualityChecker(df, connectionName=None, config=None,
                               references=None)
spark.runDataQualityChecks(df, connectionName=None, config=None,
                           includeLlm=True, references=None)
spark.runTableDataQualityChecks(tableName, connectionName=None, config=None,
                                includeLlm=True, references=None)
spark.runQueryDataQualityChecks(query, connectionName=None, config=None,
                                 includeLlm=True, references=None)
spark.saveDataQualityReport(report, path, reportFormat="json")
```

Parameter	Type	Default	Description
<code>df</code>	<code>DataFrame</code>	<code>required</code>	Data to check (streaming frames are not allowed for local work)
<code>tableName</code>	<code>str</code>	<code>required</code>	Catalog table to load and check
<code>query</code>	<code>str</code>	<code>required</code>	Spark SQL whose result is checked
<code>connectionName</code>	<code>str</code>	<code>None</code>	LLM connection used for the narrative
<code>config</code>	<code>QualipilotConfig</code>	<code>None</code>	Checks to run
<code>includeLlm</code>	<code>bool</code>	<code>True</code>	Ask the LLM to narrate the results
<code>references</code>	<code>dict</code>	<code>None</code>	Reference data forwarded to Qualipilot
<code>report</code>	<code>QualityReport</code>	<code>required</code>	Report returned by a run method
<code>path</code>	<code>str</code>	<code>required</code>	Output path
<code>reportFormat</code>	<code>str</code>	<code>"json"</code>	Output format, for example <code>html</code>

Returns. `runDataQualityChecks`, `runTableDataQualityChecks` and `runQueryDataQualityChecks` return a `QualityReport` with the check results and `llm_report`. `createDataQualityChecker` returns a checker you can use as a context manager and call `run(include_llm=...)` on.

Only a compact aggregate of the deterministic results goes to the LLM. Row samples and top values are excluded.

Local record linkage

TEXT

```
spark.runLocalRecordLinkage(data, config, rightData=None, *, maxRows)
spark.runLocalRecordDeduplication(data, config, consolidation, *, maxRows)
spark consolidateLocalRecords(data, idColumn, clusters, config, *, maxRows)
```

Run Qualipilot's native linkage on small datasets, materialized directly as driver-local Polars DataFrames. `maxRows` is mandatory and must be positive; the runtime applies `limit(maxRows + 1)` and fails if the data exceeds it.

Raises. `ValueError` with messages such as `maxRows must be a positive integer`, `data exceeds maxRows=N`, `rightData is required in link mode`, `rightData is only valid in link mode`, and `driver-local Qualipilot operations reject streaming data`. Nested, variant, duplicate or case-conflicting columns fail before collection. `maxRows` limits rows, not width or candidate pairs, so select only the columns you need.

4.8 PDF extraction

parsePDF

PYTHON

```
spark.parsePDF(s3ConnName=None, inputUri=None, outputPrefix=None,
               outputFormat="csv", outputFormats=None, inputFile=None,
               connName=None, hybrid=False, ocrMode="auto",
               ocrStack="ppocrv6", strict=False, password=None,
               language="eng", maxPages=None, maxFileBytes=None,
               llmProvider=None, llmConnName=None, llmModel=None,
               llmBaseUrl=None, llmApiKey=None, outputS3ConnName=None,
               returnAgentPages=False, contentAddressed=False,
               inlineAgentPages=True, pageRange=None)
```

Parses one PDF stored in S3 and writes the result back to S3. The `llm*` parameters configure the LLM-assisted hybrid mode and are used together with `hybrid=True`.

Parameter	Type	Default	Description
<code>s3ConnName</code>	<code>str</code>	<code>None</code>	S3 connection of the source (or use <code>connName</code>)
<code>inputUri</code>	<code>str</code>	<code>None</code>	PDF location; <code>inputFile</code> is an alias
<code>outputPrefix</code>	<code>str</code>	<code>generated</code>	Output prefix; defaults to <code>pdf-extractor/<uuid></code>
<code>outputFormat</code>	<code>str</code>	<code>"csv"</code>	<code>json</code> , <code>md</code> , <code>txt</code> , <code>csv</code> or <code>xlsx</code>
<code>outputFormats</code>	<code>list</code>	<code>None</code>	Several formats at once
<code>ocrMode</code>	<code>str</code>	<code>"auto"</code>	<code>off</code> , <code>auto</code> or <code>all</code>
<code>ocrStack</code>	<code>str</code>	<code>"ppocrv6"</code>	<code>classic</code> or <code>ppocrv6</code>
<code>language</code>	<code>str</code>	<code>"eng"</code>	OCR language
<code>strict</code>	<code>bool</code>	<code>False</code>	Fail instead of returning partial results
<code>password</code>	<code>str</code>	<code>None</code>	Password for encrypted PDFs
<code>maxPages</code>	<code>int</code>	<code>None</code>	Page limit
<code>maxFileBytes</code>	<code>int</code>	<code>None</code>	File size limit
<code>outputS3ConnName</code>	<code>str</code>	<code>None</code>	S3 connection for output; defaults to the source

Returns. A `dict` with `status`, `input_uri`, `output_prefix`, `output_format`, `result_uri`, `artifacts` (including a `manifest`), `page_count` and `sourceSha256`.

Raises. `ValueError("s3ConnName is required")` and `ValueError("inputUri or inputFile is required")`; `ValueError("pageRange requires returnAgentPages=True")`.

readPDF and readPDFResult

PYTHON

```
spark.readPDF(s3ConnName, inputUri=None, outputPrefix=None, outputFormat="csv",
              outputFormats=None, readOptions=None, outputS3ConnName=None,
              **options)
spark.readPDFResult(resultUri, outputFormat="csv", s3ConnName=None,
                    readOptions=None)
```

`readPDF` parses a PDF and returns the result as a DataFrame. `readPDFResult` reads an existing result. Spark reads `csv` (without a header by default), `json` and the text formats `txt`, `text`, `md`, `markdown`. Other formats raise `ValueError("Spark cannot read 'x' directly")`.

parsePDFCorpus

PYTHON

```
spark.parsePDFCorpus(s3ConnName, sourcePrefix=None, inputUri=None,
                     outputPrefix=None, outputS3ConnName=None,
                     pageRangeSize=64, partitions=None, ocrMode="auto",
                     language="eng", strict=False, password=None,
                     maxFileBytes=None, maxDocuments=10_000)
```

Parses an S3 prefix, or an explicit list, as restartable distributed work. Discovery is recursive and finds `.pdf` files regardless of case. Results are content-addressed: re-running with the same source and output prefix skips page ranges that were already committed.

Parameter	Type	Default	Description
<code>s3ConnName</code>	<code>str</code>	required	S3 connection of the source
<code>sourcePrefix</code>	<code>str</code>	<code>None</code>	Folder to scan (give this or <code>inputUri</code>)
<code>inputUri</code>	<code>list</code>	<code>None</code>	Explicit PDF list
<code>outputPrefix</code>	<code>str</code>	<code>None</code>	Where results go
<code>pageRangeSize</code>	<code>int</code>	<code>64</code>	Pages per unit of work
<code>partitions</code>	<code>int</code>	<code>None</code>	Parallel parsers, capped (default cap 2)
<code>maxDocuments</code>	<code>int</code>	<code>10_000</code>	Document limit

Returns. A `dict` including `status`, `documentCount` and `manifestUri`.

The cap on `partitions` exists because every parser worker loads a complete document-model set; size executor memory before raising it.

4.9 Agents

listAgents

PYTHON

```
spark.listAgents(connName)
```

Returns a `list[dict]` of the Agents visible to the connection's service account.

invokeAgent

TEXT

```
spark.invokeAgent(connName, agentId, message, sessionId=None, *,
                  requestId=None, expectedAgentVersion=None,
                  disableTools=False, pdfSourceConnName=None,
                  pdfSourceObject=None, pdfSinkConnName=None,
                  pdfSinkPrefix=None)
```

Submits the message in the background and waits for the final result, polling for up to 15 minutes so no HTTP request stays open for the whole run.

Parameter	Type	Default	Description
<code>connName</code>	<code>str</code>	required	Agent connection
<code>agentId</code>	<code>str</code>	required	Agent identifier (UUID)
<code>message</code>	<code>str</code>	required	Non-empty message
<code>sessionId</code>	<code>str</code>	None	UUID of an existing conversation
<code>requestId</code>	<code>str</code>	None	Idempotency identifier
<code>expectedAgentVersion</code>	<code>int</code>	None	Fail if the live version differs; must be ≥ 1
<code>disableTools</code>	<code>bool</code>	False	Run without tools; not allowed with PDFs
<code>pdfSourceConnName</code>	<code>str</code>	None	S3 connection of a PDF to bind
<code>pdfSourceObject</code>	<code>str</code>	None	PDF location
<code>pdfSinkConnName</code>	<code>str</code>	None	S3 connection for extracted pages
<code>pdfSinkPrefix</code>	<code>str</code>	None	Prefix for extracted pages

Returns. The final invocation result as a `dict`, including `pdfArtifacts` when a PDF was bound.

Raises. `ValueError` (message must be a non-empty string, `expectedAgentVersion` must be a positive integer, `disableTools` must be a boolean, `disableTools` cannot be used with PDF tool bindings), and `TimeoutError` with the run ID if polling times out. A timeout does not cancel or resubmit the run; inspect the existing run before starting another.

invokeAgentPDFCorpus

TEXT

```
spark.invokeAgentPDFCorpus(connName, agentId, message, pdfSourceConnName,
                           pdfSinkPrefix, *, pdfSourcePrefix=None,
                           pdfSourceObjects=None, pdfSinkConnName=None,
                           pageRangeSize=64, partitions=None,
                           summaryWorkers=4, summaryInputChars=100_000,
                           maxDocuments=10_000, maxAgentCalls=50_000,
                           ocrMode="auto", language="eng", strict=False,
                           password=None, maxFileBytes=None)
```

Runs an approved Agent over every complete PDF under a prefix. Pass exactly one of `pdfSourcePrefix` or `pdfSourceObjects`.

Parameter	Type	Default	Description
<code>connName</code>	<code>str</code>	required	Agent connection
<code>agentId</code>	<code>str</code>	required	Agent identifier
<code>message</code>	<code>str</code>	required	Instruction applied to every document
<code>pdfSourceConnName</code>	<code>str</code>	required	S3 connection of the documents
<code>pdfSinkPrefix</code>	<code>str</code>	required	Prefix for results
<code>summaryWorkers</code>	<code>int</code>	4	Concurrent document workflows, 1 to 32
<code>maxDocuments</code>	<code>int</code>	10_000	Document limit
<code>maxAgentCalls</code>	<code>int</code>	50_000	Shared Agent-call budget

Returns. A `dict` with `status`, `outputMode`, `resultsUri`, `manifestUri`, `processedDocumentCount` and `skippedDocumentCount`. Each row at `resultsUri` links to a JSON object with the validated `data`. The older names `summaryResultsUri`, `summaryManifestUri`, `summaryUri` and `summarizedDocumentCount` remain aliases.

The corpus resolves the approved Agent version and response schema before any PDF work. With a JSON response schema, the output mode is `structured`; without one, the Agent returns cited facts. Unchanged results are reused on later runs. Documents that fail are recorded in the run index and do not stop the others.

4.10 Jobs, mail and clients

executeJob

PYTHON

```
from GZ.JobUtils import executeJob

executeJob(jobNames)
```

Runs the named sub-jobs of the current job in sequence, each under the same restrictions as a normal job.

Parameter	Type	Default	Description
<code>jobNames</code>	<code>list[str]</code>	required	Names of sub-jobs to run, in order

Returns. `dict` mapping each job name to its output. **Raises.** `ValueError` when none of the names match a sub-job, and any exception from a sub-job (the remaining jobs do not run).

sendEmail

PYTHON

```
from GZ.SendEmail import sendEmail

sendEmail(subject, body, to_email, from_email, **kwargs)
```

Sends an email through the Groundzero notification service. Extra keyword arguments are added to the message.

Parameter	Type	Default	Description
<code>subject</code>	<code>str</code>	required	Subject line
<code>body</code>	<code>str</code>	required	Message body
<code>to_email</code>	<code>str</code>	required	Recipient address
<code>from_email</code>	<code>str</code>	required	Sender address

Returns. `None` . **Raises.** Any error from the notification service.

getClient

PYTHON

```
from GZ.Client import getClient

client = getClient()
```

Returns a client factory. Its methods build authenticated native clients from a connection name; each takes `connName` (the SQL engine method takes `conn_name`).

Method	Returns
<code>getSQLEngine(conn_name)</code>	SQLAlchemy engine (PostgreSQL, MySQL, Oracle, Db2, SQL Server, SAP, Redshift, Exasol, CockroachDB)
<code>getS3Client(connName)</code>	boto3 S3 client
<code>getS3FSCClient(connName)</code>	s3fs file system
<code>getSnowflakeClient(connName)</code>	Snowflake connection
<code>getMongoClient(connName)</code>	MongoDB client
<code>getAzureBlobClient(connName)</code>	Azure Blob client
<code>getKinesisClient</code> , <code>getDynamoDBClient</code> , <code>getAthenaClient</code>	AWS clients
<code>getBigQueryClient</code> , <code>getGCSCClient</code> , <code>getGoogleDriveClient</code>	Google clients
<code>getCassandraSession</code> , <code>getRedshiftClient</code> , <code>getElasticClient</code>	Database clients

Method	Returns
<code>getOpenAIClient</code> , <code>getAzureOpenAIClient</code> , <code>getAwsBedrockClient</code>	LLM clients
<code>getCohereClient</code> , <code>getTogetherAIClient</code> , <code>getHuggingFaceClient</code>	LLM clients
<code>getFireworksAIClient</code> , <code>getXAIClient</code> , <code>getMcpClient</code>	LLM and MCP clients

Further methods exist for Salesforce, SFTP, CosmosDB, Exasol, Timescale, Pinecone, Firebolt, Teradata, Neo4j, InfluxDB, DuckDB, Business Central, Shopify, Marketo, Couchbase, ClickHouse, OneDrive, Splunk, Databricks SQL, Synapse, Event Hubs, SharePoint, ScyllaDB and Weaviate, plus `read_rest` and `write_rest` helpers for REST connections.

4.11 Placeholders and outputs

Placeholders

Placeholder	Replaced with
<code>{STRTTM}</code>	Run start time, <code>YYYY-MM-DD HH:MM:SS</code>
<code>{ENDTM}</code>	Timestamp taken just before your script starts; in notifications, the end time
<code>{CLNTNM}</code>	Client name
<code>{SITENM}</code>	Site name
<code>{PRJNM}</code>	Project name
<code>{JOBNM}</code>	Job name
<code>{STS}</code>	Status
<code>{ERR}</code>	Error message
<code>{RTE}</code>	Runtime name
<code>{RGN}</code>	Region
<code>{USR}</code>	User name of the caller
<code>{USRGRPS}</code>	Groups of the caller

`{STS}` and `{ERR}` are meant for notification messages. In job code they are empty. In code, each placeholder becomes a Python string literal that already includes single quotes (for example `"'2026-10-01 09:30:00'"`). `{USRGRPS}` expands to a comma-separated list of such literals. A name that is not in the table is left unchanged, so ordinary braces in strings and dictionaries are safe unless they match a name exactly. The `${NAME}` spelling is equivalent.

outputDF

After your script ends, the runtime reads the global `outputDF`:

Value	Handling
Spark DataFrame	Up to 100,000 rows are collected; schema from the DataFrame
pandas DataFrame	Converted with <code>to_dict(orient="records")</code>
Any other iterable	Converted with <code>list(...)</code>
Not defined	No preview; the job still succeeds

4.12 Connector method index

Every connector below is available as methods on the session. `df` is a Spark DataFrame; `connName` is the Groundzero connection name; `mode` is a Spark save mode unless noted. Methods ending in `Arrow` exchange data through Apache Arrow tables.

LOOK UP SYNTAX

The Groundzero assistant can look up the Spark syntax of any connector method, so you do not need to memorize this list.

Object storage and files

TEXT

```
# S3
spark.readS3(connName, fileName, format='parquet', schema=None, options=None)
spark.writeS3(df, connName, filePath, format, mode, options=None, coalesce=False)
spark.writeStreamToS3(df, connName, filePath, format, outputMode, checkpointLocation,
    coalesce=False, options=None)
# GCS
spark.getCredentialsFileGCS(credentialsFile)
spark.readGcs(connName, fileName, format, options)
spark.writeGcs(df, connName, fileName, format, mode, options=None, coalesce=False)
# AZUREBLOB
spark.readAzureBlob(connName, filePath, options=None)
spark.writeAzureBlob(df, connName, filePath, format, mode, options=None)
# SFTP
spark.readSFTP(connName, fileType, filePath)
spark.writeSFTP(df, connName, fileType, filePath, mode='overwrite')
spark.listSFTP(connName, remotePath='.')
spark.deleteSFTP(connName, remotePath)
spark.readSFTPArrow(connName, fileType, filePath, **options)
spark.writeSFTPArrow(df, connName, fileType, filePath, mode='overwrite', **options)
spark.listSFTPArrow(connName, remotePath='.', recursive=False)
spark.deleteSFTPArrow(connName, remotePath, recursive=False)
spark.mkdirSFTPArrow(connName, remotePath)
spark.renameSFTPArrow(connName, srcPath, dstPath)
spark.existsSFTPArrow(connName, remotePath)
# GOOGLEDRIVE
spark.readGoogleDriveFiles(connName, folderId=None, query=None, limit=None)
spark.readGoogleDriveFileContent(connName, fileId)
spark.writeGoogleDriveFiles(df, connName, parentFolderId=None, batchSize=10,
    maxPartitions=None)
spark.deleteGoogleDriveFile(connName, fileId)
spark.createGoogleDriveFolder(connName, folderName, parentFolderId=None)
spark.readGoogleDriveFilesArrow(connName, folderId=None, query=None, limit=None,
    columns=None, options=None)
```

```
# ONEDRIVE
spark.readOneDriveFiles(connName, folderPath=None, fileExtensions=None, limit=None,
    batchSize=1000, userPrincipalName=None, driveId=None)
spark.writeOneDriveFiles(df, connName, folderPath=None, batchSize=10,
    maxPartitions=None, userPrincipalName=None, driveId=None)
spark.deleteOneDriveFile(connName, fileId=None, filePath=None, userPrincipalName=None,
    driveId=None)
spark.readOneDriveFilesArrow(connName, folderPath=None, limit=None, batchSize=100,
    userPrincipalName=None, driveId=None, columns=None, options=None)
# SHAREPOINT
spark.readSharePoint(connName, listName, select=None, filter=None, orderBy=None,
    expand=None, top=None, maxResults=5000)
spark.writeSharePoint(df, connName, listName, operation='create', batchSize=100)
spark.readSharePointArrow(connName, listName, select=None, filter=None, orderBy=None,
    expand=None, top=None, maxResults=None, columns=None, options=None)
spark.writeSharePointArrow(df, connName, listName, operation='create', batchSize=100,
    options=None)
```

Table formats

TEXT

```
# DELTA
spark.readDelta(connName, path=None, tableName=None, options=None, **cloudOptions)
spark.writeDelta(df, connName, path=None, tableName=None, mode='append', options=None,
    **cloudOptions)
spark.executeDelta(query)
# HUDI
spark.readHudi(connName, tableName, options=None)
spark.writeHudi(df, connName, tableName, writeOperation='upsert', options=None)
```

Relational databases and warehouses

TEXT

```
# JDBC
spark.readJDBC(connName, query, options=None)
spark.writeJDBC(df, connName, tableName, mode, options=None)
spark.executeJDBC(connName, query)
spark.bulkLoadGzLakehouseFromRedshift(connName, query, lakehouseConn, tableName,
    database=None, mode='overwrite', iamRole=None, stagingLocation=None,
    maxFileSize=DEFAULT_MAX_FILE_SIZE_BYTES, siteName=None)
# SNOWFLAKE
spark.executeSnowflakeStatement(connName, statement)
spark.executeSnowflake(connName, query, options=None)
spark.writeSnowflake(df, connName, tableName, mode='append', extraOptions=None)
spark.bulkLoadGzLakehouseFromSnowflake(connName, query, lakehouseConn, tableName,
    database=None, mode='overwrite', warehouseSize=None, storageIntegration=None,
    stagingLocation=None, maxFileSize=DEFAULT_MAX_FILE_SIZE_BYTES, siteName=None)
# GOOGLEBIGQUERY
spark.getCredentialsFileBQ(credentialsFile)
spark.executeBigQuery(connName, tableName, query=None)
spark.writeBigQuery(df, connName, tableName, writeMethod, mode)
# ATHENA
spark.readAthena(connName, region=None, query=None, options=None, workGroup=None,
    catalog=None, sessionToken=None, accessKey=None, secretAccessKey=None)
spark.executeAthena(connName, region=None, query=None, options=None, workGroup=None,
    catalog=None, sessionToken=None, accessKey=None, secretAccessKey=None)
spark.writeAthena(df, connName, tableName, mode, options=None, region=None,
    workGroup=None, catalog=None, sessionToken=None, accessKey=None,
    secretAccessKey=None, path=None, createTable=None, replaceTable=None,
```

```

        repairTable=None)
# DATABRICKS
spark.executeDatabricks(connName, query, options=None)
spark.writeDatabricks(df, connName, tableName, mode='append', options=None)
spark.executeDatabricksStatement(connName, query)
# TRINO
spark.executeTrino(connName, query, options=None)
spark.writeTrino(df, connName, tableName, mode='append', options=None)
spark.executeTrinoStatement(connName, query)
# PRESTO
spark.executePresto(connName, query, options=None)
spark.writePresto(df, connName, tableName, mode='append', options=None)
spark.executePrestoStatement(connName, query)
# VERTICA
spark.executeVertica(connName, query, options=None)
spark.writeVertica(df, connName, tableName, mode='append', options=None)
spark.executeVerticaStatement(connName, query)
# EXASOL
spark.executeExasol(connName, query, fingerprint=None)
spark.writeExasol(df, connName, tableName, fingerprint=None, mode='append',
    options=None)
# DORIS
spark.readDoris(connName, tableIdentifier=None, options=None)
spark.executeDoris(connName, tableIdentifier=None, filterQuery=None, options=None)
spark.writeDoris(df, connName, tableIdentifier=None, mode='append', options=None)
# DREMIO
spark.executeDremio(connName, query, options=None)
spark.writeDremio(df, connName, tableName, mode='append', options=None)
spark.executeDremioStatement(connName, query)
# DREMIOFLIGHT
spark.executeDremioFlight(connName, query, options=None)
spark.writeDremioFlight(df, connName, tableName, mode='append', options=None)
spark.executeDremioFlightStatement(connName, query)
# FIREBOLT
spark.executeFirebolt(connName, sqlQuery)
spark.readFirebolt(connName, sqlQuery)
spark.writeFirebolt(df, connName, tableName, mode='append', options=None)
spark.executeFireboltSQL(connName, sqlStatement)
spark.listFireboltTables(connName)
# SINGLESTORE
spark.readSingleStore(connName, query, options=None)
spark.readSingleStoreTable(connName, tableName, options=None)
spark.writeSingleStore(df, connName, tableName, mode, options=None)
spark.executeSingleStore(connName, query, options=None)
# MOTHERDUCK
spark.executeMotherDuck(connName, query)
spark.readMotherDuck(connName, query)
spark.executeMotherDuckSQL(connName, statement)
spark.writeMotherDuck(df, connName, tableName, mode='append')

```

NoSQL, search and caches

TEXT

```

# MONGODB
spark.readMongodb(connName, databaseName, collectionName, options=None)
spark.writeMongodb(df, connName, databaseName, collectionName, mode)
# DYNAMODB
spark.readDynamoDB(connName, tableName, options=None, schema=None)
spark.readDynamoDBIndex(connName, tableName, indexName, options=None, schema=None)
spark.queryDynamoDB(connName, tableName, hashKeyName, hashKeyValue, rangeKeyName=None,

```

```

    rangeKeyValue=None, rangeKeyOperator='=', options=None, schema=None)
spark.writeDynamoDB(df, connName, tableName, mode='append', options=None)
spark.updateDynamoDB(df, connName, tableName, mode='append', options=None)
spark.deleteDynamoDB(df, connName, tableName, mode='append', options=None)
# COSMOSDB
spark.readCosmosDb(connName)
spark.writeCosmosDb(df, connName, mode)
# COUCHBASE
spark.readCouchbase(connName, scope=None, collection=None, service='query', **kwargs)
spark.writeCouchbase(df, connName, scope=None, collection=None, service='query',
    mode='overwrite', **kwargs)
# CASSANDRA
spark.executeCassandra(connName, tableName, query, isQuery=False)
spark.writeCassandra(df, connName, table, mode)
spark.executeCassandraStatement(connName, statement)
# SCYLLADB CLOUD
spark.readScyllaDbCloud(connName, tableName, keyspace=None, options=None)
spark.executeScyllaDbCloud(connName, query, keyspace=None)
spark.writeScyllaDbCloud(df, connName, tableName, keyspace=None, mode='append',
    options=None)
spark.executeScyllaDbCloudCQL(connName, cqlStatement, keyspace=None)
spark.listScyllaDbCloudTables(connName, keyspace=None)
spark.createScyllaDbCloudTable(connName, tableName, columns, keyspace=None)
# ELASTICSEARCH
spark.readElasticsearch(connName, indexName)
spark.writeElasticsearch(df, connName, indexName, mode)
# REDIS
spark.readRedis(connName, table=None, options=None)
spark.writeRedis(df, connName, table=None, mode='append', options=None)
# NEO4J
spark.executeNeo4j(connName, query, options=None)
spark.writeNeo4j(df, connName, nodeLabel=None, relationship=None, mode='overwrite',
    options=None)
# INFLUXDB
spark.executeInfluxDB(connName, query, database=None)
spark.writeInfluxDB(df, connName, measurement, timeColumn='time', tagColumns=None,
    fieldColumns=None, batchSize=5000, timeUnit='ns', useForeachPartition=True,
    maxPartitions=None)
spark.writeInfluxDBLineProtocol(df, connName, batchSize=5000, maxPartitions=None)

```

Streams

TEXT

```
# KAFKA
spark.readKafka(connName, topic, startingOffsets=None, options=None)
spark.writeKafka(df, connName, topic, checkpointLocation, outputMode, triggerTime,
    options=None, s3AccessKey=None, s3SecretKey=None, s3SessionToken=None,
    s3Region=None, s3Endpoint=None)

# KINESIS
spark.readKinesisStream(connName, startingPosition, consumerType, options=None)
spark.writeKinesisStream(connName, df, checkpointLocation, outputMode, triggerTime,
    outputType, options=None, forEachBatch=None)

# EVENTHUBS
spark.readAzureEventHubs(connName, consumerGroup, **kwargs)
spark.writeAzureEventHubs(df, connName, **kwargs)

# PULSAR
spark.readPulsar(connName, topic=None, options=None)
spark.readPulsarStream(connName, topic=None, options=None)
spark.writePulsar(df, connName, topic=None, mode='append', options=None)
spark.writePulsarStream(df, connName, checkpointLocation, topic=None,
    outputMode='append', triggerTime='1 seconds', options=None)
```

SaaS and APIs

TEXT

```
# REST
spark.readREST(connName, endpoint=None, method=None, params=None, data=None,
    options=None)
spark.writeREST(df, connName, endpoint=None, method='POST', chunkSize=1000,
    options=None)
spark.readRESTArrow(connName, endpoint=None, method=None, params=None, data=None,
    options=None, columns=None)
spark.writeRESTArrow(df, connName, endpoint=None, method='POST', chunkSize=1000,
    options=None)

# SALESFORCE
spark.executeSalesforce(connName, sqlQuery, includeDeleted=False)
spark.readSalesforce(connName, sqlQuery, includeDeleted=False)
spark.writeSalesforce(df, connName, objectName, operation='insert', batchSize=2000,
    useBulk=False, externalIdField='Id', dropNulls=False, useSerial=False,
    raiseOnError=True)
spark.executeSalesforceArrow(connName, sqlQuery, includeDeleted=False, columns=None,
    options=None)
spark.readSalesforceArrow(connName, sqlQuery, includeDeleted=False, columns=None,
    options=None)
spark.writeSalesforceArrow(df, connName, objectName, operation='insert', batchSize=200,
    externalIdField='Id', options=None, **legacyOpts)

# HUBSPOT
spark.readHubSpot(connName, objectType, properties=None, limit=None, offset=None,
    filters=None, searchTerm=None, rateLimit=10)
spark.writeHubSpot(df, connName, objectType, operation='create', batchSize=100,
    rateLimit=10)
spark.readHubSpotArrow(connName, objectType, properties=None, limit=None, filters=None,
    searchTerm=None, rateLimit=None, columns=None, options=None)
spark.writeHubSpotArrow(df, connName, objectType, operation='create', batchSize=100,
    rateLimit=None, idProperty=None, options=None)

# SERVICENOW
spark.readServiceNow(connName, table, query=None, limit=1000, displayValue=True,
    options=None)
spark.executeServiceNow(connName, sourceQuery, limit=1000, options=None)
spark.writeServiceNow(df, connName, table, method='POST', batchSize=100)
```

```

spark.readServiceNowArrow(connName, table, query=None, limit=None, displayValue=None,
    columns=None, options=None)
spark.executeServiceNowArrow(connName, sourceQuery, columns=None, options=None)
spark.writeServiceNowArrow(df, connName, table, operation='create', idColumn='sys_id',
    options=None)
# SHOPIFY
spark.readShopify(connName, resource, params=None, maxResults=None,
    shopifyTier='enterprise', rateLimit=None, batchSize=250)
spark.writeShopify(df, connName, resource, operation='create', batchSize=50,
    shopifyTier='enterprise', rateLimit=None)
spark.readShopifyArrow(connName, resource, params=None, maxResults=None, rateLimit=None,
    batchSize=250, recordsPath=None, columns=None, options=None)
spark.writeShopifyArrow(df, connName, resource, operation='create', batchSize=50,
    wrapperKey=None, options=None)
# ZOHOCRm
spark.readZohoCRM(connName, module, perPage=200, maxPages=None)
spark.executeZohoCRM(connName, module)
spark.readZohoCRMArrow(connName, module, perPage=None, maxPages=None, columns=None,
    options=None)
spark.executeZohoCRMArrow(connName, module, columns=None, options=None)
# M365BC
spark.readM365BC(connName, entity, operation='get_all', params=None)
spark.writeM365BC(df, connName, entity, operation='create', batchSize=100)
spark.readM365BCArrow(connName, entity, params=None, columns=None, options=None)
spark.writeM365BCArrow(df, connName, entity, operation='create', batchSize=100,
    options=None)
# SPLUNK
spark.executeSplunk(connName, searchQuery, operation='get_all', params=None,
    requestTimeout=30)
spark.writeSplunk(df, connName, batchSize=100, index=None, source=None, sourcetype=None,
    hecUrl=None, hecHost=None, hecPort=None, hecPath='/services/collector/event',
    requestTimeout=30, requestChannel=None)
spark.executeSplunkArrow(connName, searchQuery, earliestTime=None, latestTime=None,
    requestTimeout=None, columns=None, options=None)
spark.writeSplunkArrow(df, connName, index=None, source=None, sourcetype=None,
    hecUrl=None, hecHost=None, hecPort=8088, hecPath='/services/collector/event',
    requestChannel=None, options=None)

```

Vector stores

TEXT

```

# PINECONE
spark.readPinecone(connName, ids_to_fetch=None, schema=None, options=None)
spark.writePinecone(df, connName=None, options=None, mode='upsert')
# QDRANT
spark.readQdrant(connName, collectionName=None, schema=None, options=None)
spark.writeQdrant(df, connName, collectionName=None, options=None)
# WEAVIATECLOUD
spark.createWeaviateCloudClass(connName, className, properties, vectorizer='none',
    moduleConfig=None)
spark.readWeaviateCloud(connName, className, properties=None, where_filter=None,
    limit=None, batchSize=10000)
spark.writeWeaviateCloud(df, connName, className, batchSize=100, maxPartitions=None,
    idColumn=None, vectorColumn=None, mode='append', options=None, writer='spark')
spark.listWeaviateCloudClasses(connName)
spark.deleteWeaviateCloudObject(connName, className, objectId)

```

Model Context Protocol

TEXT

```
# MCP
spark.listMcpTools(connName)
spark.callMcpTool(connName, toolName, arguments=None)
spark.listMcpResources(connName)
spark.readMcpResource(connName, uri)
spark.listMcpPrompts(connName)
spark.getMcpPrompt(connName, promptName, arguments=None)
spark.completeWithMcpTool(llmProvider, llmConnName, mcpConnName, df, toolName,
    arguments=None, argumentsColumn=None, promptColumn='prompt', promptTemplate=None,
    mcpPromptTemplate=None, systemPrompt=None, systemPromptColumn=None,
    outputColumn='llm_response', mcpResultColumn='mcp_tool_result',
    mcpErrorColumn='mcp_tool_error', model=None, temperature=0.0, maxTokens=None,
    responseFormat=None, continueOnError=False, timeout=120, llmOptions=None)
spark.getMcpLlmConfig(connName, provider='openai', serverName=None,
    requireApproval='always', includeSecrets=False)
```

5 Installed packages

Packages marked "pinned" have a fixed version. The others install at their latest version when the runtime is built. To read the exact version in your session, run `pip show <name>` in a notebook.

5.1 Python

Package	Version
Python	3.12
numpy	2.3.5 (pinned)
pandas	2.2 or later, below 3
pyspark	3.5.6 (pinned)
scikit-learn	1.5.2 (pinned)
duckdb	1.5.5 (pinned)
torch, torchvision (CPU)	2.14.0, 0.29.0 (pinned)
mlflow	2.20.4 (pinned)

5.2 Data, SQL and cloud libraries

Package	Purpose
<code>SQLAlchemy</code> , <code>psycopg2-binary</code> , <code>pymysql</code> , <code>pyodbc</code> , <code>cx-Oracle</code>	Database access
<code>jaydebeapi</code>	JDBC from Python
<code>snowflake-connector-python</code> , <code>simple-salesforce</code> , <code>hubspot-api-client</code>	Warehouse and SaaS clients
<code>cassandra-driver</code> , <code>redis</code> , <code>azure-eventhub</code> , <code>influxdb3-python</code>	Store and stream clients
<code>pinecone</code> , <code>msgpack</code> , <code>paramiko</code>	Vector store, serialization, SFTP
<code>boto3</code> , <code>urllib3</code> , <code>requests</code> , <code>httpx</code> , <code>httpcore</code>	AWS and HTTP
<code>pyiceberg[s3fs,sql-postgres,sql-sqlite,pyarrow]</code>	Iceberg from Python
<code>sparkmeasure</code>	Spark metrics

5.3 LLM and AI libraries

Package	Purpose
<code>openai</code> , <code>anthropic</code> , <code>cohere</code> , <code>google-generativeai</code>	Provider SDKs
<code>huggingface-hub</code>	Model hub client
<code>mcp</code>	Model Context Protocol, 1.29 up to 2.0

Package	Purpose
<code>qualipilot[gz,linking]</code>	Data quality and record linkage, 3.5.0 (pinned)

5.4 Document extraction

Package	Version
<code>docling[rapidocr]</code>	2.116.0 (pinned)
<code>rapidocr</code>	3.9.2 (pinned)
<code>onnxruntime</code>	1.28.0 (pinned)
<code>pdfplumber</code> , <code>pypdf</code> , <code>pdf2image</code> , <code>pytesseract</code> , <code>pillow</code> , <code>openpyxl</code>	0.11, 4.2, 1.17, 0.3, 10.0, 3.1 or later

The layout, table-structure and OCR models come with the runtime, so PDF extraction works without internet access.

5.5 Notebook and plotting libraries

`matplotlib`, `jupyterlab`, `ipython`, `ipykernel`. The Groundzero helper packages (`GZ` and others) are already importable.

5.6 JVM libraries

Library	Version
Spark	3.5.6
Iceberg Spark runtime and AWS bundle	1.11.0
Delta Lake	3.3.2
Hudi Spark bundle	1.1.1
AWS SDK v2 bundle	2.25.70
AWS SDK v1 bundle	1.12.781

Connector drivers (Snowflake, MongoDB, Kafka, Cassandra, Elasticsearch, BigQuery, JDBC drivers and others) are preinstalled and loaded for the connection types your run uses.

5.7 System packages

OpenJDK 17, the AWS CLI, ODBC (`unixodbc`) and time-zone data, so timestamps with a time zone work with Arrow.

6 Limits and troubleshooting

6.1 Limits and quotas

Limit	Value
Workers per session	1 minimum (default), 10 maximum
Worker scale-up threshold	70 percent CPU or memory
Executor memory	65 percent of worker memory, at most worker memory minus 1 GiB
Idle executor release	120 seconds
Rows collected for a preview	100,000
Dataset page size	Default 1,000, maximum 10,000
Dataset run wait	300 seconds
Session kept after its last run	280 seconds
Session inactivity shutdown	300 seconds
Network and broadcast timeout	600 seconds
Agent invocation wait	15 minutes
Agent corpus documents, calls	10,000, 50,000 by default
Agent corpus workers	1 to 32, default 4
LLM request timeout	120 seconds by default
LLM retries	3, for HTTP 429 and 5xx

What the limits mean in practice:

- **Dataset runs are bounded.** A dataset run must answer within 300 seconds. Move long transformations into a scheduled job and let the dataset read the result.
- **Sessions end.** A session is released after about 5 minutes of inactivity. Never keep state in the driver's memory that you cannot rebuild.
- **Previews are samples.** The preview holds at most 100,000 rows. Write the complete result to a table or file when you need all rows.
- **Stop times end the session.** When a schedule's stop time is reached, the session ends and aborts running work.

6.2 Restrictions in job scripts

Job scripts run with restrictions. Notebook kernels do not apply these rules.

Area	Rule
Imports	<code>socket</code> , <code>urllib</code> , <code>http</code> , <code>ssl</code> , <code>threading</code> , <code>multiprocessing</code> , <code>pickle</code> , <code>shutil</code> , <code>pathlib</code> , <code>glob</code> , <code>gzip</code> , <code>zipfile</code> , <code>sqlite3</code> , <code>signal</code> , <code>asyncio</code> and similar modules cannot be imported
Processes	<code>subprocess</code> , <code>os.system</code> , <code>os.popen</code> , <code>os.exec*</code> , <code>os.spawn*</code> and <code>os.fork</code> are blocked
Files	<code>open</code> works under <code>/tmp</code> , <code>/home/workspace</code> and <code>/home</code> ; other system paths are refused
Credentials	Secret values in <code>os.environ</code> show as <code>****</code> ; changes stay private to your script
Dynamic code	<code>eval</code> , <code>exec</code> , <code>compile</code> , <code>input</code> and <code>breakpoint</code> are blocked
AWS	<code>boto3</code> clients cannot be created; use the connectors or <code>getClient()</code>
Exit	<code>sys.exit()</code> is blocked

Use connectors for network and storage access, and finish a script by assigning `outputDF` rather than calling `exit`.

6.3 Things to know

- **Spark settings apply once.** In a reused session (notebook, shared dataset session) `sparkSession(..., **settings)` returns the existing session and ignores new settings. Restart the session to change them.
- **`collect()` has no result limit.** A very large `collect()` exhausts the driver's memory.
- **`coalesce=True` writes through a single task.** Use it for small outputs only.
- **A lakehouse catalog belongs to one run.** Do not keep a catalog name from an earlier run (see `belongs to another execution` below).
- **Unpinned packages move.** The TensorFlow variant and any package without a pinned version can change when the runtime is rebuilt.
- **Streaming into a lakehouse table needs its partitioning first.** Create the table with its partition spec, then stream into it.

6.4 Troubleshooting

Each entry gives the message exactly as you see it, then the cause and the fix.

Symptom	Cause	Fix
<code>AttributeError: 'sparkSession' object has no attribute 'read'</code>	You used <code>spark.read</code> , <code>spark.write</code> or <code>spark.sparkContext</code> .	Use the connector methods, for example <code>spark.readS3(...)</code> , or DataFrame operations.
<code>ImportError: Import of 'x' is not allowed for security reasons</code>	The module is blocked in job scripts.	Use a connector for the same task, or run the code in a notebook.
<code>PermissionError: os.system() is not allowed (and similar for subprocess)</code>	Job code cannot start processes.	Use library calls instead.

Symptom	Cause	Fix
PermissionError: Access to '/data/x' outside sandbox-safe directories is not allowed	The script wrote outside the allowed folders.	Write temporary files under /tmp or /home/workspace . Use S3 or a lakehouse table for anything that must outlive the run.
ImportError: Direct access to 'boto3' is restricted	A job script created a boto3 client.	Use spark.readS3 , spark.writeS3 or getClient() .
ValueError: No executable code found.	The job has no code.	Save the script and run again.
ValueError: siteName is required for GZLAKEHOUSE connection 'x'.	Groundzero could not work out the site.	Pass siteName='<site>' to the method or to sparkSession .
ValueError: REST catalog writes only support mode='append' or mode='overwrite'; got 'x'.	The write mode is not allowed for lakehouse tables.	Change mode to append or overwrite .
ValueError: database is required for REST catalog writes	No database was given.	Pass database= or set a default database on the connection.
PermissionError: GZLAKEHOUSE connection 'x' is external and read-only; ...	External lakehouse connections only allow SELECT , SHOW , DESCRIBE and EXPLAIN .	Write to an internal connection.
ValueError: partitionBy is not supported for streaming GZLAKEHOUSE writes; ...	partitionBy was passed to a streaming write.	Create the table with its partition spec, then stream into it.
RuntimeError: GZLAKEHOUSE catalog 'x' belongs to another execution.	You used a catalog name saved from an earlier run.	Get the current one with spark.getLakehouseCatalog(...) or use executeQuery .
ValueError: Connection x not found or is not a JDBC type.	The name is wrong or the connection is not a JDBC type.	Check the connection name and type.
TypeError: Use writeSnowflake() for connection x.	writeJDBC was used on a Snowflake connection.	Call writeSnowflake .
Exception: Error reading file from S3: ...	The text after the colon is the cause (missing key, permission, malformed file).	Check the path and the connection's bucket.
ValueError: Missing columns: [...]. Available: [...]	A column named in promptColumn , promptTemplate or systemPromptColumn is not in the DataFrame.	Fix the column name.
ValueError: AWS Bedrock model must be set in the connection or call.	No model was given.	Pass model= (the model ID) or set a model on the connection.

Symptom	Cause	Fix
The dataset session stopped responding before it took the run. Run the dataset again to start a new session.	The session ended before your run started.	Run the dataset again. A new session starts.
<code>TimeoutError</code> from <code>invokeAgent</code>	The wait for the Agent ended after 15 minutes. The Agent run is not cancelled.	Look up the run ID in the error and inspect that run before you submit again.
<code>ErrorType: message at line N</code>	An error in your script. <code>N</code> is the line in your code, and the failing line is printed below.	Fix that line.
A job is slow to start	A new session starts a driver and at least one worker.	Keep a notebook or dataset warm, raise the minimum workers for large jobs, or schedule heavy work ahead of time.

6.5 FAQ

Do I need to create the SparkSession myself? Call `sparkSession(appName)`. It returns the shared session with all connectors attached. A bare PySpark session has no connectors, credentials or governance.

How do I see the result of my job? Assign a DataFrame to `outputDF`. The preview and its schema appear in the UI. Anything you `print` appears in the log.

Can I use pandas? Yes. Assign a pandas DataFrame to `outputDF`. `toPandas()` works for small results. For large data, stay in Spark.

Where do my credentials come from? From the connection. Code uses only the connection name, and secret values are hidden from your script.

How do I choose how many workers I get? Set Minimum Workers in the runtime picker. Groundzero adds workers, up to 10, when CPU or memory use passes 70 percent.

Why did my Spark settings not apply? The session already existed. Settings passed to `sparkSession` are used only when the session is created.

Can two users share a dataset session? Yes. Runs on the same dataset share a pool of sessions, one run per session at a time. Lakehouse permissions apply to each user, and the session clears the previous user's views.

How do I add a library? Use `%pip install` in a notebook for a trial. For permanent packages, ask an administrator to create a custom runtime based on a Spark runtime.

Which variant do I need for machine learning? For scikit-learn or boosting libraries, the scikit-learn variant. For deep learning with PyTorch, the PyTorch variant. For TensorFlow or spaCy, the TensorFlow variant.

Can I read a table I cannot write? Yes. External lakehouse connections are read-only, and per-table permissions decide who can query or write a table.

What happens to my notebook files? Your notebook is saved each time you save it.

How are runs recorded? Each run records status, logs, Spark metrics, audit entries for lakehouse access and lineage from every connector read and write.

7 Examples gallery

All examples use fictional names and data. Replace connection names (`acme-*`) with your own. Each example is a complete job script.

7.1 Daily revenue rollup in the lakehouse

Reads yesterday's orders from a lakehouse table, aggregates them by region and appends the result to a summary table. The aggregate is also shown as the job preview.

PYTHON

```
from GZ.SparkUtils import sparkSession
from pyspark.sql import functions as F

spark = sparkSession("DailyRevenueRollup", siteName="acme")

orders = spark.executeQuery(
    connName="acme-lakehouse",
    sqlQuery="""
        SELECT region, product, quantity, unit_price
        FROM sales.orders
        WHERE order_date = date_sub(current_date(), 1)
    """,
)

daily = (
    orders.withColumn("revenue", F.col("quantity") * F.col("unit_price"))
    .groupBy("region")
    .agg(
        F.sum("revenue").alias("revenue"),
        F.countDistinct("product").alias("products"),
    )
    .withColumn("report_date", F.date_sub(F.current_date(), 1))
)

spark.writeGzLakehouse(
    df=daily,
    connName="acme-lakehouse",
    tableName="revenue_daily",
    database="sales",
    mode="append",
)

outputDF = daily.orderBy(F.desc("revenue"))
```

7.2 Incremental load from PostgreSQL to Parquet

Pulls rows changed in the last day, stamps them with the run's name and start time through placeholders, and appends them to Parquet in S3.

PYTHON

```
from GZ.SparkUtils import sparkSession
from pyspark.sql import functions as F

spark = sparkSession("IncrementalCustomers")

run_job = {JOBNM}.strip("'")
run_started = {STRTTM}.strip("'")

changes = spark.readJDBC(
    connName="acme-postgres",
    query="SELECT customer_id, email, country, updated_at "
    "FROM crm.customers WHERE updated_at ≥ now() - interval '1 day'",
)

landed = (
    changes.withColumn("loaded_by", F.lit(run_job))
    .withColumn("loaded_at", F.lit(run_started))
)

spark.writeS3(
    df=landed,
    connName="acme-s3",
    filePath="landing/crm/customers",
    format="parquet",
    mode="append",
)

print("Loaded", landed.count(), "changed customers")
outputDF = landed.groupBy("country").count()
```

The placeholders are substituted as quoted string literals, so `.strip("'")` removes the quotes that are meant for SQL.

7.3 Deduplicate and enrich with window functions

Keeps the latest record per customer and joins a country reference table read from CSV.

PYTHON

```
from GZ.SparkUtils import sparkSession
from pyspark.sql import Window
from pyspark.sql import functions as F

spark = sparkSession("DedupeCustomers")

customers = spark.readS3(
    connName="acme-s3",
    fileName="landing/crm/customers_raw/",
    format="csv",
    options={"header": "true", "inferSchema": "true"},
)
countries = spark.readS3(
    connName="acme-s3",
    fileName="reference/countries.csv",
    format="csv",
    options={"header": "true"},
)

latest = Window.partitionBy("customer_id").orderBy(F.desc("updated_at"))
deduped = (
    customers.withColumn("rank", F.row_number().over(latest))
    .where(F.col("rank") == 1)
    .drop("rank")
    .join(F.broadcast(countries), on="country_code", how="left")
)

spark.writeS3(
    df=deduped,
    connName="acme-s3",
    filePath="curated/customers",
    format="parquet",
    mode="overwrite",
)
outputDF = deduped
```

7.4 Stream Kafka events into an Iceberg table

Consumes a topic and appends micro-batches to a lakehouse table. With the `availableNow` trigger the job processes everything that has arrived and then finishes, which suits scheduled runs.

PYTHON

```
from GZ.SparkUtils import sparkSession
from pyspark.sql import functions as F

spark = sparkSession("OrderEventsStream", siteName="acme")

raw = spark.readKafka(
    connName="acme-kafka",
    topic="orders.created",
    startingOffsets="earliest",
)

events = raw.select(
    F.col("key").cast("string").alias("order_key"),
    F.get_json_object(F.col("value").cast("string"), "$.region").alias("region"),
    F.get_json_object(F.col("value").cast("string"), "$.total")
    .cast("double")
    .alias("total"),
    F.col("timestamp").alias("event_time"),
)

query = spark.writeGzLakehouse(
    df=events,
    connName="acme-lakehouse",
    tableName="order_events",
    database="sales",
    streaming=True,
    checkpointLocation="s3://acme-data/checkpoints/order_events",
    outputMode="append",
    trigger={"availableNow": True},
)

query.awaitTermination()
```

CREATE THE TABLE FIRST

The target `sales.order_events` table must already exist with the partition specification you want, because the streaming writer does not apply `partitionBy`.

7.5 Classify support tickets with an LLM

Calls an LLM connection once per row, keeps going when individual rows fail, and saves successes and failures separately.

PYTHON

```
from GZ.SparkUtils import sparkSession
from pyspark.sql import functions as F

spark = sparkSession("TicketTriage")

tickets = spark.executeQuery(
    connName="acme-lakehouse",
    sqlQuery="SELECT ticket_id, subject, body FROM support.tickets LIMIT 500",
    siteName="acme",
)

scored = spark.complete(
    connName="acme-openai",
    df=tickets,
    promptTemplate=(
        "Classify this support ticket as billing, technical or other. "
        "Reply with one word.\nSubject: {{subject}}\nBody: {{body}}"
    ),
    systemPrompt="You are a precise ticket router.",
    outputColumn="category",
    maxTokens=5,
    continueOnError=True,
)

good = scored.where(F.col("category_status") == "success")
bad = scored.where(F.col("category_status") == "error")

spark.writeS3(good, "acme-s3", "triage/ok", "parquet", "overwrite")
spark.writeS3(bad, "acme-s3", "triage/errors", "parquet", "overwrite")

outputDF = good.groupBy(F.lower(F.trim("category")).alias("category")).count()
```

7.6 Train and apply a scikit-learn model

Trains on a sample collected to the driver, then scores the full DataFrame in parallel with a pandas UDF. Run it on the scikit-learn (or another ML) variant, or on the base runtime, since scikit-learn is part of the base runtime.

PYTHON

```

import pandas as pd
from GZ.SparkUtils import sparkSession
from pyspark.sql import functions as F
from pyspark.sql.functions import pandas_udf
from sklearn.ensemble import GradientBoostingClassifier

spark = sparkSession("ChurnModel")

features = ["tenure_months", "monthly_spend", "support_tickets"]
customers = spark.executeQuery(
    connName="acme-lakehouse",
    sqlQuery="SELECT customer_id, tenure_months, monthly_spend, "
    "support_tickets, churned FROM crm.customer_features",
    siteName="acme",
)

sample = customers.sample(fraction=0.2, seed=7).limit(50_000).toPandas()
model = GradientBoostingClassifier(random_state=7)
model.fit(sample[features], sample["churned"])

@pandas_udf("double")
def churn_probability(*cols: pd.Series) → pd.Series:
    frame = pd.concat(cols, axis=1)
    frame.columns = features
    return pd.Series(model.predict_proba(frame)[: , 1])

scored = customers.withColumn(
    "churn_probability", churn_probability(*[F.col(c) for c in features])
)
outputDF = scored.select("customer_id", "churn_probability").orderBy(
    F.desc("churn_probability")
)

```

7.7 Load a warehouse query into the lakehouse in bulk

Verifies the lakehouse route, lets Snowflake unload the rows directly into the table, then counts them.

PYTHON

```

from GZ.SparkUtils import sparkSession

spark = sparkSession("OrdersBulkLoad", siteName="acme")

check = spark.verifyLakehouseConnection("acme-lakehouse", siteName="acme")
print("Routing to warehouse", check["warehouse"], "at site", check["site_used"])

summary = spark.bulkLoadGzLakehouseFromSnowflake(
    connName="acme-snowflake",
    query="SELECT * FROM analytics.orders WHERE order_date ≥ '2026-01-01'",
    lakehouseConn="acme-lakehouse",
    tableName="orders_history",
    database="sales",
    mode="overwrite",
    warehouseSize="X-Large",
    siteName="acme",
)
print("Bulk load summary:", summary)

outputDF = spark.executeQuery(
    connName="acme-lakehouse",
    sqlQuery="SELECT COUNT(*) AS loaded_rows FROM sales.orders_history",
)

```

7.8 Extract structured data from a folder of PDFs with an Agent

Runs an approved Groundzero Agent over every PDF in a prefix and reads the results back as a DataFrame.

PYTHON

```

from GZ.SparkUtils import sparkSession

spark = sparkSession("InvoiceExtraction")

result = spark.invokeAgentPDFCorpus(
    connName="acme-agents",
    agentId="123e4567-e89b-42d3-a456-426614174000",
    message="Extract the invoice number, vendor and total into the schema.",
    pdfSourceConnName="acme-s3",
    pdfSourcePrefix="incoming/invoices/",
    pdfSinkConnName="acme-s3",
    pdfSinkPrefix="agent-pdf-results/invoices",
    summaryWorkers=8,
)
print(result["status"], result["outputMode"])
print("Processed", result["processedDocumentCount"], "documents")

outputDF = spark.readS3(
    connName="acme-s3",
    fileName="agent-pdf-results/invoices/",
    format="json",
)

```

7.9 Quality gate with notification

Runs checks on a table, writes the HTML report, and mails the team when a column is out of range.

PYTHON

```

from GZ.SendEmail import sendEmail
from GZ.SparkUtils import sparkSession
from qualipilot import CheckConfig, QualipilotConfig
from qualipilot.models.config import ColumnRange

spark = sparkSession("OrdersQualityGate")

orders = spark.executeQuery(
    connName="acme-lakehouse",
    sqlQuery="SELECT order_id, customer_email, amount FROM sales.orders",
    siteName="acme",
)
config = QualipilotConfig(
    checks=CheckConfig(
        required_columns=["order_id", "customer_email", "amount"],
        column_ranges={"amount": ColumnRange(min=0, max=100_000)},
    )
)
report = spark.runDataQualityChecks(
    df=orders, connectionName="acme-openai", config=config
)
spark.saveDataQualityReport(report, "reports/orders_quality.html", "html")

negative = orders.where("amount < 0").count()
if negative:
    sendEmail(
        subject="Orders quality gate failed",
        body=f"{negative} orders have a negative amount.",
        to_email="data-team@example.com",
        from_email="alerts@example.com",
    )
outputDF = orders.where("amount < 0")

```

7.10 Orchestrate sub-jobs

A parent job runs two sub-jobs in order and then reports their outputs.

PYTHON

```

from GZ.JobUtils import executeJob
from GZ.SparkUtils import sparkSession

spark = sparkSession("NightlyPipeline")

results = executeJob(["load_orders", "build_marts"])
for name, output in results.items():
    print("Finished", name, "with output", output)

outputDF = spark.createDataFrame(
    [(name,) for name in results], ["completed_job"]
)

```

APPENDIX A

Choosing a Runtime and Shared Concepts

Compare the runtimes and learn the concepts they share.

- A.1 Choosing a runtime
- A.2 How a run starts
- A.3 What your code can use during a run
- A.4 Connections and secrets
- A.5 The lakehouse (Iceberg)
- A.6 Custom runtimes
- A.7 ML variants
- A.8 Jobs: lifecycle, status and logs
- A.9 Glossary

Groundzero offers four ways to run your data work: the **Standard** runtime, the **DuckDB** runtime, the **Spark** runtime and the **Lakehouse** SQL engine. This chapter covers what they have in common, so the per-runtime guides can stay focused. It explains how to choose a runtime, how a run starts, how to size compute, what your code can use during a run, how connections and the lakehouse work, how custom and machine-learning runtimes are added, and how to follow a run to the end.

A.1 Choosing a runtime

A.1.1 At a glance

	Standard	DuckDB	Spark	Lakehouse
Engine	Managed query engine, SQL plus pandas Prep	DuckDB 1.5.5 on one machine	Apache Spark 3.5.6 cluster	Spark SQL 4.0 with Iceberg 1.10
Compute size	None	You choose	You choose, plus minimum workers	You choose, plus minimum workers
Start	Nothing to start	Quick start, then kept warm for datasets	Cluster start, then kept warm for datasets	Start a session, then run many queries
Connections	One SQL source or the lakehouse	Many, in one query	Many, in one job	The warehouse you open
Writes	No (read queries)	Yes, with the helper library	Yes, with the helper library	Yes, with SQL on Iceberg tables
Scale	Pages of rows	Fits in one machine's memory, or streams from storage	Distributed across workers	Distributed across workers, results paged
Languages	SQL and Python (pandas)	Python and SQL	Python (PySpark) and SQL	SQL (Spark dialect)
Helper library	None	<code>GZ.DuckDBUtils</code>	<code>GZ.SparkUtils</code>	None
ML variants	None	scikit-learn, PyTorch, TensorFlow	scikit-learn, PyTorch, TensorFlow	None
Best for	Charts, dashboards, quick lookups	Fast analytics, ETL, joins across sources	Large data, wide shuffles, distributed ETL	Exploring and maintaining Iceberg tables with SQL

A.1.2 Decision guide

Work down this list and stop at the first match.

1. Is it one query on one SQL source or the lakehouse, feeding a chart, dashboard or API read? Use **Standard**.
2. Is the work interactive SQL on Iceberg tables: exploring, creating or altering tables, `MERGE`, time travel, table maintenance, policies? Use the **Lakehouse**: open a warehouse, start a session and write SQL.
3. Does it join several connections, write results, run on a schedule, or run Python beyond a few lines of pandas? Continue.
4. Does the data fit comfortably in the memory of one machine, or can DuckDB stream it from object storage (Parquet, Iceberg)? Use **DuckDB**. It starts quickly and needs no cluster.

5. **Is the data too large for one machine, or does the job need distributed shuffles, joins of very large tables or Spark-native libraries?** Use **Spark**.
6. **Do you need scikit-learn, PyTorch or TensorFlow?** Pick the matching ML variant of DuckDB or Spark.
7. **Do you need packages that are not installed?** Ask an administrator for a custom runtime built on the runtime you picked.

START SMALL

Prototype on Standard or DuckDB. Move to Spark when a job outgrows one machine, not before. The Groundzero helper libraries of DuckDB and Spark follow the same shape, which keeps a port short.

A.1.3 Where the choice is made

You choose a runtime in three places: when you create a dataset, when you create or schedule an ETL job, and when you start a notebook. The runtime control lists the runtimes available to you and, except for Standard, a compute size. Standard appears only in the dataset designer. You choose the Lakehouse by opening a warehouse in **Lakehouse**, where you pick a compute size and start a session.

A.2 How a run starts

A.2.1 From Run to results

The steps are the same everywhere you can start a run.

1. You press Run, a schedule fires, or an API client submits a job.
2. Groundzero checks that you may use the chosen compute size and the dataset or job.
3. It prepares the connections you selected for the run.
4. It records the run with a unique identifier and starts lineage tracking.
5. It starts your runtime with the compute size you chose, or reuses a warm session (see Sessions and warm starts).
6. Your code runs, and its status, output and logs appear as it goes.
7. For a dataset run, the first page of rows is returned to the caller. For a job, the run page follows status and logs until the run ends.

A.2.2 Datasets, jobs and notebooks compared

	Dataset run	ETL job run	Notebook
Starts from	Dataset designer, chart, dashboard	Job page, schedule, API	Notebook page
Output	A page of rows	Whatever the code writes, plus logs	Interactive
Session	Kept warm and shared (DuckDB, Spark)	Ends when the run ends	Stays up until stopped
Default page size	1000 rows	not applicable	not applicable

In a Spark or DuckDB notebook the Groundzero helper library is already imported (`sparkSession` and `executeJob` for Spark; `duckSession` for DuckDB), together with NumPy, pandas and boto3. Notebooks are saved to your notebook storage each time you save.

A.2.3 Sessions and warm starts

Starting a runtime takes a moment, so Groundzero keeps a **warm session** after a dataset run on DuckDB or Spark. Your next run starts faster if it arrives within a few minutes.

- A session handles one run at a time. A new run takes an idle session that matches its dataset, compute size and connections, and prefers the one you used last.
- If every matching session is busy, Groundzero starts another one. Runs never wait in a queue behind each other.
- A warm session stays available for about 280 seconds after a run and ends after about 300 seconds without a run. After that, the next run starts a new session.
- Lakehouse permissions follow the **user who runs**, not the session, so sharing a session never widens anyone's access.

- Do not keep state in memory between runs. Any run can start in a new session.

Jobs do not share sessions. Each job run gets its own session and ends it when the job finishes. A schedule can also set a stop time, at which the run ends.

A.2.4 Timeouts and retries

Setting	Value
Wait for a dataset run's output	300 seconds by default
Wait for a new session to become ready	60 seconds
Spark workers	1 minimum by default, 10 maximum

If a run request fails before it reaches the runtime, Groundzero retries it automatically. If it may already have reached the runtime, Groundzero does not send it again, so a run never executes twice by accident.

A.2.5 Compute sizes, workers and GPU size

A **compute size** is a named CPU and memory pair that your administrators define. The runtime control lists each as `Name | memory | cpu`. The size you choose is the CPU and memory your code gets. You need Describe permission on a compute size to use it.

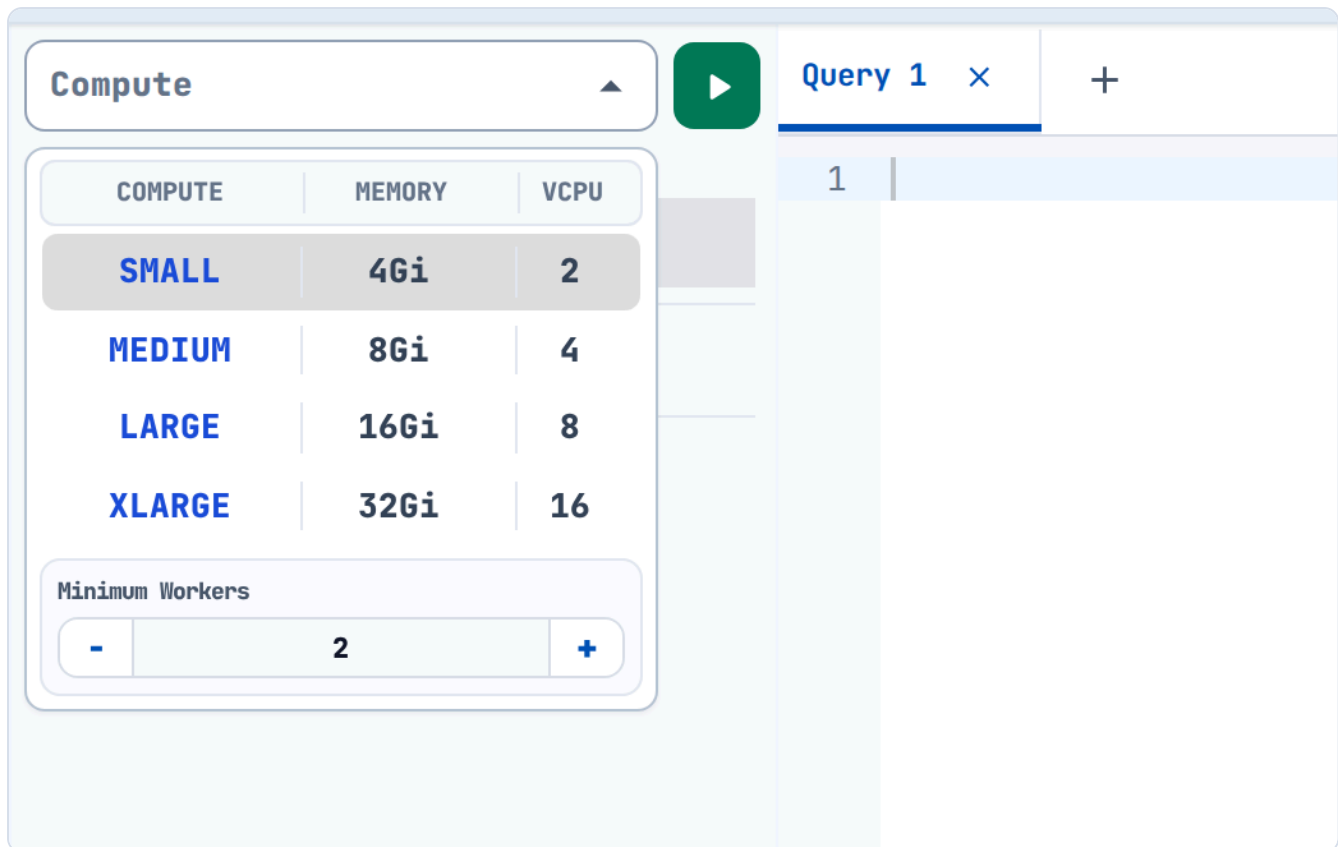
For Spark:

- The compute size applies to the driver and to each worker.
- **Minimum workers** sets how many workers start. If you leave it empty, one starts. Workers are added, up to a maximum of 10, when average CPU or memory use passes 70 percent.
- Each executor gets about 65 percent of its worker's memory, and never more than the worker's memory minus 1 GiB.
- Spark starts with one executor and releases idle executors after 120 seconds.

The runtime control also shows a **GPU Size** field for every runtime except Standard.

CHOOSING MEMORY

DuckDB and the Standard runtime hold working data in the memory of one machine. Spark spreads it over workers and spills to disk. When a run stops because it ran out of memory, pick a larger compute size first, then reduce the data you read.



Compute

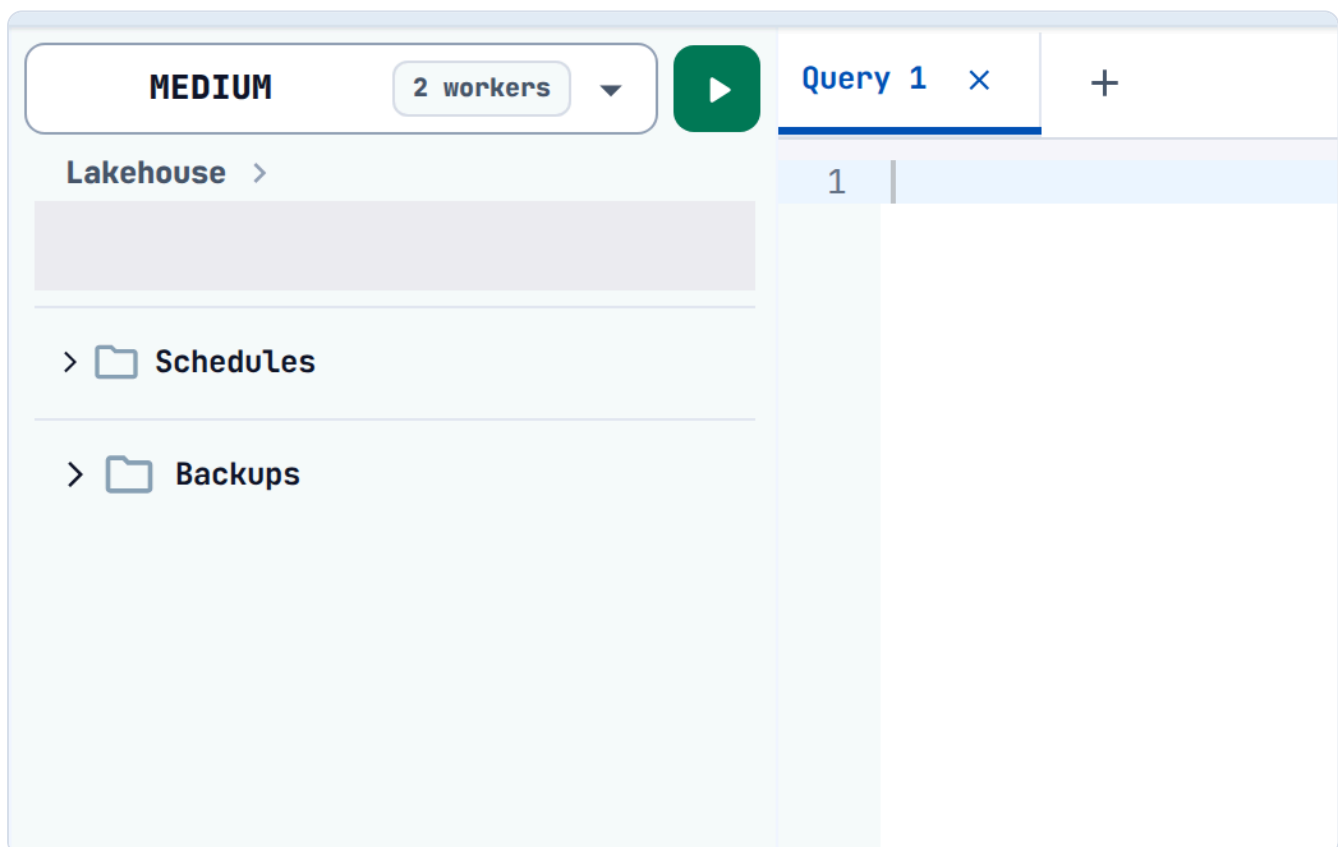
COMPUTE	MEMORY	VCPU
SMALL	4Gi	2
MEDIUM	8Gi	4
LARGE	16Gi	8
XLARGE	32Gi	16

Minimum Workers

- 2 +

Query 1

Compute sizes are chosen from a table of name, memory and vCPU.



MEDIUM 2 workers

Lakehouse >

> Schedules

> Backups

Query 1

A chosen compute size with its worker count.

A.3 What your code can use during a run

A.3.1 Run details

Every run gives your code a fixed set of values. You can read them from `os.environ` in notebooks and, where listed, in job code.

Name	Meaning
<code>USERNAME</code>	User who started the run
<code>USERGROUPS</code>	The user's groups, comma-separated
<code>SITE_NAME</code>	Site (tenant) name
<code>REGION</code>	Cloud region
<code>JOB_IDENTIFIER</code>	Id of this run
<code>SESSION_ID</code>	Id of the session
<code>WORKLOAD</code>	<code>ETL_JOB</code> or <code>NOTEBOOK</code>
<code>PAGE_NUMBER</code> , <code>PAGE_SIZE</code>	Paging of a dataset run

Do not rely on any other values or try to override these.

A.3.2 Job placeholders

Text in a job can contain `${NAME}` placeholders that the runtime fills in when the job runs.

Placeholder	Meaning
<code>\${STRTTM}</code>	Run start time
<code>\${ENDTM}</code>	Run end time
<code>\${CLNTNM}</code> , <code>\${SITENM}</code>	Client and site name
<code>\${PRJNM}</code> , <code>\${JOBNM}</code>	Project and job name
<code>\${STS}</code> , <code>\${ERR}</code>	Run status and error text
<code>\${RTE}</code> , <code>\${RGN}</code>	Runtime name and region
<code>\${USR}</code> , <code>\${USRGRPS}</code>	User and groups

The Standard runtime uses a smaller set; see the Standard chapter.

A.3.3 Imported models

A run can be given registered ML models. Groundzero finds each model's latest stored version and makes it available to your code.

A.4 Connections and secrets

A **connection** stores how to reach a source (host, database, credentials or role) under a name. Your code refers to the connection by name and never contains credentials.

- Groundzero prepares your connections when a run starts.
- Only the connections you selected for the job or dataset are available to its code.
- Lakehouse connections use short-lived access that Groundzero refreshes for you, so long-running sessions keep working.
- Do not print connection details or credentials to run logs. Logs are stored and shown on the run page.

NEVER HARD-CODE SECRETS

Put credentials in a connection. Do not paste keys or passwords into job code, parameters or notebooks.

A.5 The Lakehouse (Iceberg)

The Groundzero lakehouse stores tables in Apache Iceberg format on object storage, with one warehouse per site. The lakehouse console and its SQL editor run lakehouse SQL for you. Runtimes read and write the table data directly, with temporary access that applies your Groundzero roles and policies.

Runtime	Engine versions	How it reaches the lakehouse
Standard	DuckDB 1.5.5	Reads Iceberg tables (read-only)
DuckDB	DuckDB 1.5.5, iceberg extension	Reads through the helper library (read-only)
Spark	Spark 3.5.6, Iceberg 1.11.0	Reads and writes through the helper library
Lakehouse	Spark SQL 4.0.0, Iceberg 1.10.0	Your SQL session in the Lakehouse pages (see the Lakehouse guide)

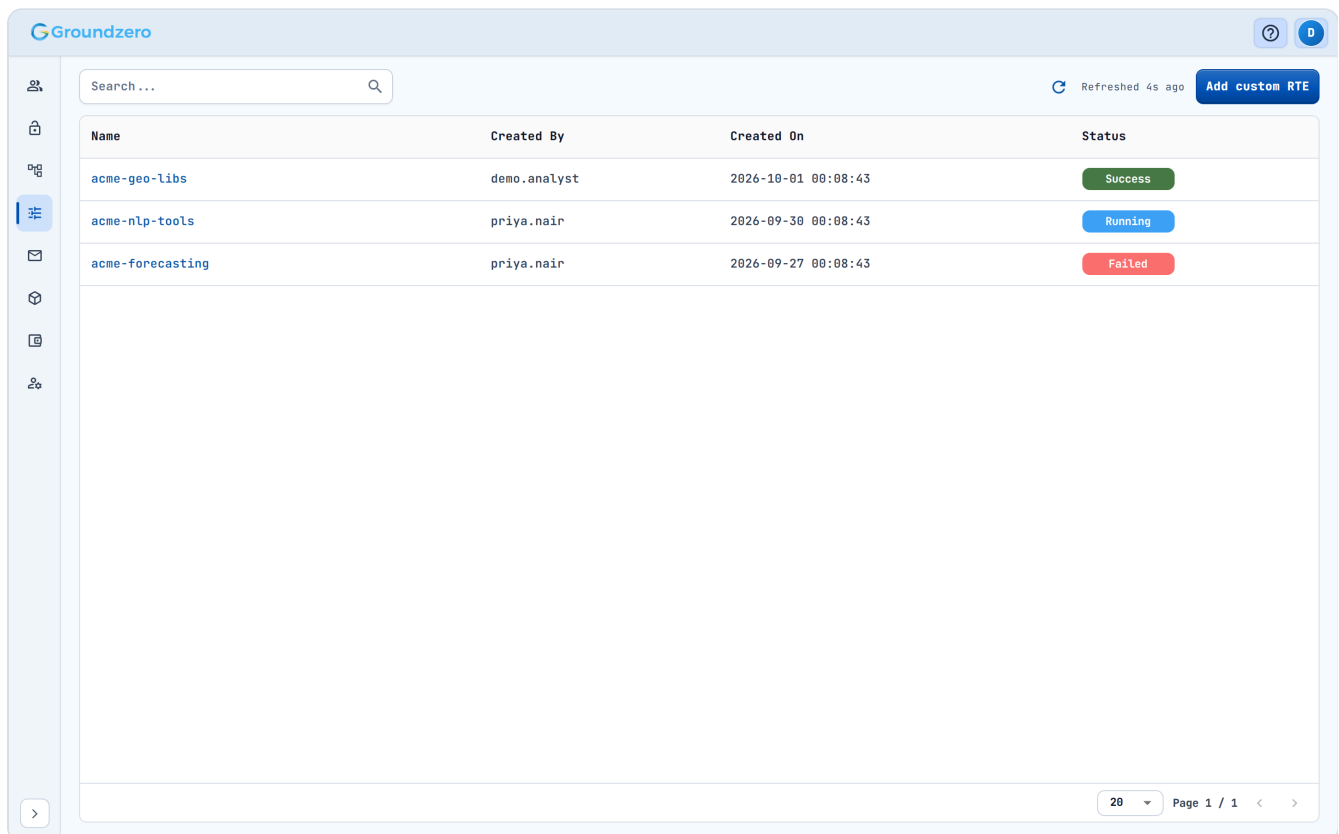
A.6 Custom runtimes

A custom runtime is an existing runtime plus your own pip packages. Administrators create it on the **Custom RTE** page.

1. Upload a `requirements.txt` and choose the **Base Image** (the runtime to extend) and a name.
2. Groundzero builds the runtime with the packages from `requirements.txt`.
3. The build status shows as QUEUED, IN PROGRESS, then SUCCESS or FAILURE.
4. When the build succeeds, the runtime appears in the runtime list.

Guidelines:

- Extend the runtime you actually run on: a Spark base for Spark jobs, a DuckDB base for DuckDB jobs.
- Pin versions. Do not change packages the base already includes (NumPy, pandas, scikit-learn) unless you need a different version.
- Pure-Python packages install most reliably. A package that needs extra system libraries may fail to build. Check the FAILURE status and the package's requirements.



The screenshot shows the Groundzero admin console interface. At the top, there's a search bar and a refresh button. Below is a table with columns: Name, Created By, Created On, and Status. The table lists three custom runtimes: 'acme-geo-lib' (Success), 'acme-nlp-tools' (Running), and 'acme-forecasting' (Failed). The status is indicated by colored buttons: green for Success, blue for Running, and red for Failed. At the bottom right, there's a pagination control showing 'Page 1 / 1'.

Name	Created By	Created On	Status
acme-geo-lib	demo.analyst	2026-10-01 00:08:43	Success
acme-nlp-tools	priya.nair	2026-09-30 00:08:43	Running
acme-forecasting	priya.nair	2026-09-27 00:08:43	Failed

The Custom RTE page in the admin console lists each custom runtime and its build status.

The screenshot displays the Groundzero Spark Runtime interface. At the top, there is a search bar and a refresh button labeled 'Refreshed 8s ago'. A table lists custom RTEs with columns for Name, Created By, Created On, and Status. A modal dialog titled 'Add Custom RTE' is open in the center, allowing users to add a new custom runtime environment. The dialog includes a 'Base Image' dropdown menu set to 'spark', a 'Name' input field containing 'geo-libs', and a 'Choose File (.txt file only)' button. Below the file selection, the text 'spark-geo-libs' and 'requirements.txt' are visible. The dialog has 'Cancel' and 'Add' buttons at the bottom right.

Name	Created By	Created On	Status
acme-geo-libs	demo.analyst	2026-10-01 00:08:43	Success
acme-nlp-tools	priya.nair	2026-09-30 00:08:43	Running
acme-forecasting			Failed

Add Custom RTE

Base Image: spark

Name: geo-libs

spark-geo-libs

Choose File (.txt file only)

requirements.txt

Cancel Add

Add custom RTE: choose a base image, name the runtime and upload a requirements file.

A.7 ML variants

Spark and DuckDB each come as a base runtime and three ML variants. A variant is the base plus extra libraries.

Variant	Adds
Base	Spark or DuckDB, Groundzero helper libraries, scikit-learn 1.5.2, CPU PyTorch 2.14.0 and torchvision 0.29.0, document parsing
sklearn	OpenCV (headless), XGBoost, LightGBM, CatBoost
pytorch	sklearn extras, PyTorch 2.5.1 and torchvision 0.20.1 (GPU build), SHAP, LIME, Albumentations, Hugging Face Hub and Transformers, simsimd, fastai
tensorflow	TensorFlow, OpenCV, XGBoost, LightGBM, CatBoost, SHAP, LIME, NLTK, spaCy, h5py

Notes:

- The pytorch variant uses the GPU build of PyTorch, so GPU-sized runs can train on GPUs.
- Variants appear in the runtime list with names such as `gz-rte-duckdb-sklearn` and a version tag such as `1.0.0`.
- The Standard runtime has no ML variants.

A.8 Jobs: lifecycle, status and logs

A.8.1 Statuses

Status	Meaning
Queued	Accepted, waiting to start
Running	Your code is executing
Success	Finished without error
Failed	The code or Groundzero reported an error
Aborted	Stopped before it could finish

Queued and Running are active. Success, Failed and Aborted are final.

A.8.2 Logs

Run logs stream to the run page while the run works.

- Lines never repeat when the page reloads the stream.
- If a run stops without reporting a final state, Groundzero marks it Failed rather than leaving it Running.
- If the page is asked for logs too often, it retries after a short delay.

Print and logging output from your code appears in the run log as the run works.

A.8.3 Scheduling

A schedule names the job, its runtime and its compute size. It may also set a start and stop time.

A.9 Glossary

Term	Meaning
Runtime (RTE)	The environment and engine your code runs in
Compute size	A named CPU and memory pair for a runtime
Session	A running runtime that can accept runs
Warm session	A dataset session kept ready for a few minutes and shared between runs
Connection	A named, stored way of reaching a source
Dataset	A saved query (and code) that returns rows
Prep	The pandas code that reshapes a Standard dataset's rows
Lakehouse	Groundzero's Iceberg tables on object storage
Warehouse	A site's lakehouse storage location
Custom runtime	A runtime extended with your pip packages
ML variant	A runtime with machine-learning libraries added
Lineage	The record of which assets a run read and wrote