



DEVELOPER DOCUMENTATION

Groundzero DuckDB Runtime

Fast single-node analytics with DuckDB and Iceberg on Groundzero.

Developer documentation

Edition 2026-10-02

Runtime environments: Spark, DuckDB, Standard, Lakehouse

Contents

Guide - DuckDB Runtime

1	Overview	5
1.1	What the DuckDB runtime gives you	6
1.2	Choose the DuckDB runtime	6
1.3	How a run works	7
1.4	Versions	7
1.5	What changed recently	8
1.6	Machine-learning variants	8
2	Quickstart	9
2.1	Your first job	9
2.2	Your first dataset	11
2.3	Your first notebook	11
3	How-to guides	14
3.1	Create and configure a session	14
3.2	Read from connections	14
3.3	Query across sources	16
3.4	Query the Groundzero Lakehouse	17
3.5	Write results	19
3.6	Return output and previews	20
3.7	Use parameters and placeholders	21
3.8	Run jobs and schedule them	22
3.9	Use extra packages	23
3.10	Work with files	23
3.11	Tune performance	24
3.12	Log and debug	25
3.13	Handle secrets	25
3.14	Extract text from PDFs	26
3.15	Run Groundzero Agents over documents	26
3.16	Check data quality	27
4	API reference	28
4.1	Conventions	28
4.2	Session	28
4.3	Groundzero Lakehouse	31
4.4	Object storage	33
4.5	SQL databases	34
4.6	Cloud warehouses	35
4.7	Outputs and run context	36
4.8	Clients and connection details	37
4.9	Connectors	38
4.10	LLM providers	44
4.11	MCP servers	46
4.12	PDF extraction	47

4.13	Groundzero Agents	49
4.14	Data quality and record linkage	50
4.15	Talend migration helpers	50
5	Installed packages	52
5.1	Python and system	52
5.2	Connector and data libraries	52
5.3	LLM and AI libraries	54
5.4	Document extraction	54
5.5	DuckDB extensions	55
5.6	Packages added by the ML variants	55
6	Limits and troubleshooting	56
6.1	Limits and quotas	56
6.2	Things to know	56
6.3	Troubleshooting	57
6.4	FAQ	59
7	Examples gallery	60
7.1	Example 1: Partitioned daily export	60
7.2	Example 2: Rolling averages with window functions	60
7.3	Example 3: Flatten JSON events	61
7.4	Example 4: Churn scoring with scikit-learn	62
7.5	Example 5: Deduplicate and validate before loading	62
7.6	Example 6: Customer 360 across lakehouse, PostgreSQL and S3	63
7.7	Example 7: Incremental load from Snowflake	64
7.8	Example 8: Classify support tickets with an LLM	65
7.9	Example 9: Ingest a REST API	66
7.10	Example 10: Geospatial distance query	67
7.11	Example 11: Extract policy fields from PDFs with an Agent	68
7.12	Example 12: A two-step pipeline with executeJob	68

Appendix A - Choosing a Runtime and Shared Concepts

A.1	Choosing a runtime	69
A.1.1	At a glance	70
A.1.2	Decision guide	70
A.1.3	Where the choice is made	71
A.2	How a run starts	72
A.2.1	From Run to results	72
A.2.2	Datasets, jobs and notebooks compared	72
A.2.3	Sessions and warm starts	72
A.2.4	Timeouts and retries	73
A.2.5	Compute sizes, workers and GPU size	73
A.3	What your code can use during a run	75
A.3.1	Run details	75
A.3.2	Job placeholders	75
A.3.3	Imported models	75
A.4	Connections and secrets	76

A.5	The lakehouse (iceberg)	77
A.6	Custom runtimes	78
A.7	ML variants	80
A.8	Jobs: lifecycle, status and logs	81
A.8.1	Statuses	81
A.8.2	Logs	81
A.8.3	Scheduling	81
A.9	Glossary	82

GUIDE

DuckDB Runtime

Fast single-node analytics with DuckDB 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 DuckDB runtime runs your Python code next to the DuckDB analytics engine. It is the fastest way to join, filter and aggregate data that fits on one machine. You read from any Groundzero connection into DuckDB relations, combine them with SQL, and either return the result (datasets, previews) or write it back to storage and databases (jobs).

This guide shows how to run your first job, dataset and notebook, how to read and write data, and how to use the Groundzero helper functions. It also covers the installed packages, limits, troubleshooting and a gallery of runnable examples.

WHO THIS GUIDE IS FOR

Developers writing datasets, ETL jobs and notebooks on Groundzero. You should be comfortable with Python and SQL. All examples use fictional data and connection names such as `sales_s3` and `orders_pg`; replace them with the names of your own connections.

1 Overview

1.1 What the DuckDB runtime gives you

A Groundzero *runtime* is a software environment plus a compute size. When you run a dataset, job or notebook, Groundzero starts the runtime you selected with the connections your code needs, runs your code, and returns the output or logs.

The DuckDB runtime includes:

- Python 3.12 with pandas, NumPy and PyArrow.
- DuckDB 1.5.5 with the `iceberg`, `httpfs`, `aws`, `postgres`, `mysql`, `azure`, `json`, `parquet`, `spatial` and `motherduck` extensions already installed, so nothing is downloaded when your code runs.
- The Groundzero helper library (`GZ`). `duckSession()` returns a session object that wraps a DuckDB connection and adds one read and write method per connector (S3, JDBC databases, Snowflake, BigQuery, the Groundzero Lakehouse, REST APIs, vector stores, LLM providers and more).
- PDF text extraction (Docling with RapidOCR), the Qualipilot data quality library, and clients for Groundzero Agents and MCP servers.
- JupyterLab for notebooks, with `numpy`, `pandas`, `boto3`, `getClient` and `duckSession` already imported.

DuckDB is a columnar engine that runs inside your Python process. There is no cluster to size and no copying between Python and the engine. Relations, pandas DataFrames and Arrow tables are queried in place, and work larger than memory spills to local disk.

1.2 Choose the DuckDB runtime

Groundzero offers three runtime choices. Pick the one that matches the shape of your workload.

	Spark	DuckDB	Standard
Engine	Apache Spark 3.5.6	DuckDB 1.5.5	Platform default
Topology	Driver plus workers	One machine, one process	No compute to choose
Scale	Distributed	Single node, spills to disk	Single SQL connection
Lakehouse	Read and write	Read only	See the Standard guide
Code	PySpark and SQL	Python, SQL, pandas, Arrow	SQL
Best for	Very large or write-heavy ETL	Fast analytics and joins	Simple SQL datasets

Choose DuckDB when:

- the data you actually process fits one machine's memory and local disk;
- you join several sources (for example a lakehouse table, an S3 file and a PostgreSQL table) in one SQL statement;
- you want a quick start for datasets and previews, because DuckDB needs no cluster;
- you want to mix SQL with pandas, scikit-learn or Arrow code.

Choose another runtime when:

- you must write to the Groundzero Lakehouse. DuckDB reads the lakehouse only, so use the Spark runtime for lakehouse writes;
- the data does not fit on one machine even after filtering and projection;
- the dataset is a single SQL query against one SQL connection and needs no code, which the Standard option covers.

1.3 How a run works

Every dataset, job or notebook that uses the DuckDB runtime runs in its own isolated environment. Its size comes from the compute you choose in the runtime selector (see the Quickstart).

Workload	What starts	When it stops
ETL job	One run of your job file	When the job code finishes
Dataset run or preview	A session that serves runs	After idle time, see below
Notebook	A JupyterLab session	When you stop the session

A session handles one run at a time. For datasets and previews, Groundzero keeps idle sessions ready for a short while. A new run uses an idle session if one exists and otherwise starts another, so runs never queue behind each other. An idle session is released after about five minutes. Do not keep state in memory between runs, because any run may land on a new session.

Credentials

Your code only names a connection (`connName="sales_s3"`). It never handles keys or passwords. Secret values are hidden from your code: `os.environ` shows them as `****` .

Resources

The compute you pick sets the CPU and memory of the run. DuckDB is sized from it automatically:

- threads equal the CPU count of the compute (minimum two);
- the DuckDB memory limit is 80 percent of the compute's memory (minimum 512 MB), which leaves the rest for Python, Arrow and client libraries;
- work that does not fit in memory spills to disk under `/tmp/duckdb-spill` .

To override threads and memory for one session, call `duckSession(threads=..., memory_limit=...)` .

1.4 Versions

The runtime is updated continuously and has no numbered releases.

Component	Version
Python	3.12
DuckDB	1.5.5
NumPy	2.3.5

Component	Version
pandas	2.2 or newer, below 3
scikit-learn	1.5.2
SciPy	1.17.1
Qualipilot (<code>gz</code> , <code>linking</code>)	3.5.0
Docling / RapidOCR	2.116.0 / 3.9.2
MLflow	2.20.4
PyTorch (CPU build)	2.14.0, torchvision 0.29.0

Packages without a version in these tables install at their latest release when the runtime is built. The complete list is in Installed packages.

1.5 What changed recently

Recent changes that affect your code, newest first:

- Lakehouse reads in long-running sessions no longer fail with expired storage tokens.
- Lakehouse access policies, including column masking, apply to the user who runs the code. Cached rows are kept per user policy.
- Lineage records schemas for connector reads and writes.
- DuckDB threads and memory are sized from your compute, and each dataset session uses the full compute for one run at a time.
- Time zone data is included, so time-zone-aware timestamps (including `UTC`) work.
- Python 3.12 is the base for every runtime, including the ML variants.

1.6 Machine-learning variants

Three variants add machine learning libraries on top of the DuckDB runtime. They share the same `gz` library, extensions and behavior.

Variant	Adds
DuckDB scikit-learn	OpenCV (headless), XGBoost, LightGBM, CatBoost
DuckDB PyTorch	PyTorch 2.5.1 and torchvision 0.20.1 (CUDA 12.1), plus the scikit-learn additions, SHAP, LIME, Alumentations, Hugging Face Hub, Transformers, simsimd and fastai
DuckDB TensorFlow	TensorFlow, OpenCV, XGBoost, LightGBM, CatBoost, SHAP, LIME, NLTK, spaCy, h5py

scikit-learn, NumPy, SciPy and pandas are in every variant. The PyTorch variant uses the CUDA 12.1 build of PyTorch, so it can use an NVIDIA GPU when you set a GPU size.

If a library is in none of the variants, ask an administrator to create a custom runtime (see Use extra packages).

2 Quickstart

This section takes you from nothing to a working job, dataset and notebook. You need a Groundzero site with at least one connection to a data source. The examples use an Amazon S3 connection named `sales_s3` whose bucket, `acme-retail-data`, holds Parquet files of orders.

The orders files in these examples have the following columns.

Column	Type	Meaning
<code>order_id</code>	BIGINT	Unique order number
<code>customer_id</code>	BIGINT	Customer who placed the order
<code>region</code>	VARCHAR	Sales region, for example <code>EMEA</code>
<code>amount</code>	DOUBLE	Order value
<code>status</code>	VARCHAR	<code>open</code> , <code>shipped</code> or <code>returned</code>
<code>ordered_at</code>	TIMESTAMP	When the order was placed

2.1 Your first job

An ETL job is a Python file that runs once. This job reads one day of orders from S3, computes revenue per region and writes the result back to S3 as Parquet.

PYTHON

```

from GZ.DuckDBUtils import duckSession

session = duckSession()

orders = session.readS3File(
    connName="sales_s3",
    bucketName="acme-retail-data",
    filePath="raw/orders/2026-09-30.parquet",
    fileType="parquet",
)
session.register("orders", orders)

revenue_sql = """
    SELECT region,
           count(*)           AS orders,
           round(sum(amount), 2) AS revenue
    FROM orders
    WHERE status <> 'returned'
    GROUP BY region
    ORDER BY revenue DESC
"""

print(session.query(revenue_sql, returnFormat="pandas"))

session.writeS3File(
    connName="sales_s3",
    bucketName="acme-retail-data",
    filePath="curated/daily_revenue/2026-09-30.parquet",
    fileType="parquet",
    sourceQuery=revenue_sql,
)
session.close()

```

What happened:

1. `duckSession()` created a DuckDB connection and attached every connector method to the returned session object.
2. `readS3File` configured S3 access from the `sales_s3` connection and returned a lazy DuckDB relation. No data was downloaded yet.
3. `register` exposed the relation to SQL under the name `orders`.
4. `query` ran the aggregation. With `returnFormat="pandas"` the result is materialized as a DataFrame. The `print` output appears in the job's log.
5. `writeS3File` ran a `COPY ... TO` that streamed the result to S3.

To check it worked, open the job log for the printed table, then look for the new Parquet file under `curated/daily_revenue/` in the bucket.

KEEP DATA IN DUCKDB FOR AS LONG AS POSSIBLE

Relations are lazy. Convert to pandas or Arrow only for the final step or when a library requires it. Doing joins and filters in SQL avoids copying data into Python memory.

2.2 Your first dataset

A dataset runs your code and returns rows to the Groundzero UI, dashboards and APIs. Instead of writing files, you leave the rows you want to return in a variable called `outputDF` (or pass them to `gz_output()`).

PYTHON

```
from GZ.DuckDBUtils import duckSession

session = duckSession()

orders = session.readS3File(
    connName="sales_s3",
    bucketName="acme-retail-data",
    filePath="raw/orders/2026-09-30.parquet",
    fileType="parquet",
)
session.register("orders", orders)

outputDF = session.query(
    """
    SELECT region, status, count(*) AS orders, sum(amount) AS revenue
    FROM orders
    GROUP BY region, status
    ORDER BY region, status
    """
)
```

`outputDF` can be a DuckDB relation, a PyArrow table, record batch or reader, a pandas DataFrame, or a list of dictionaries. Groundzero converts it to JSON rows plus a schema, returns at most 100,000 rows, and pages the result for the UI.

2.3 Your first notebook

In a notebook the runtime pre-imports `numpy as np`, `pandas as pd`, `boto3`, `getClient` and `duckSession`, so the first cell can start working immediately:

PYTHON

```
session = duckSession()
orders = session.readS3File("sales_s3", "acme-retail-data",
                           "raw/orders/2026-09-30.parquet")

orders.limit(5).df()
```

The notebook session stays alive until you stop it, and you cannot change the runtime while it runs.

In the UI

You choose the runtime in the same control everywhere it appears. The control is a single-line field that shows the selected runtime name and, for non-Standard runtimes, a small badge with the compute name. Clicking it opens a panel with:

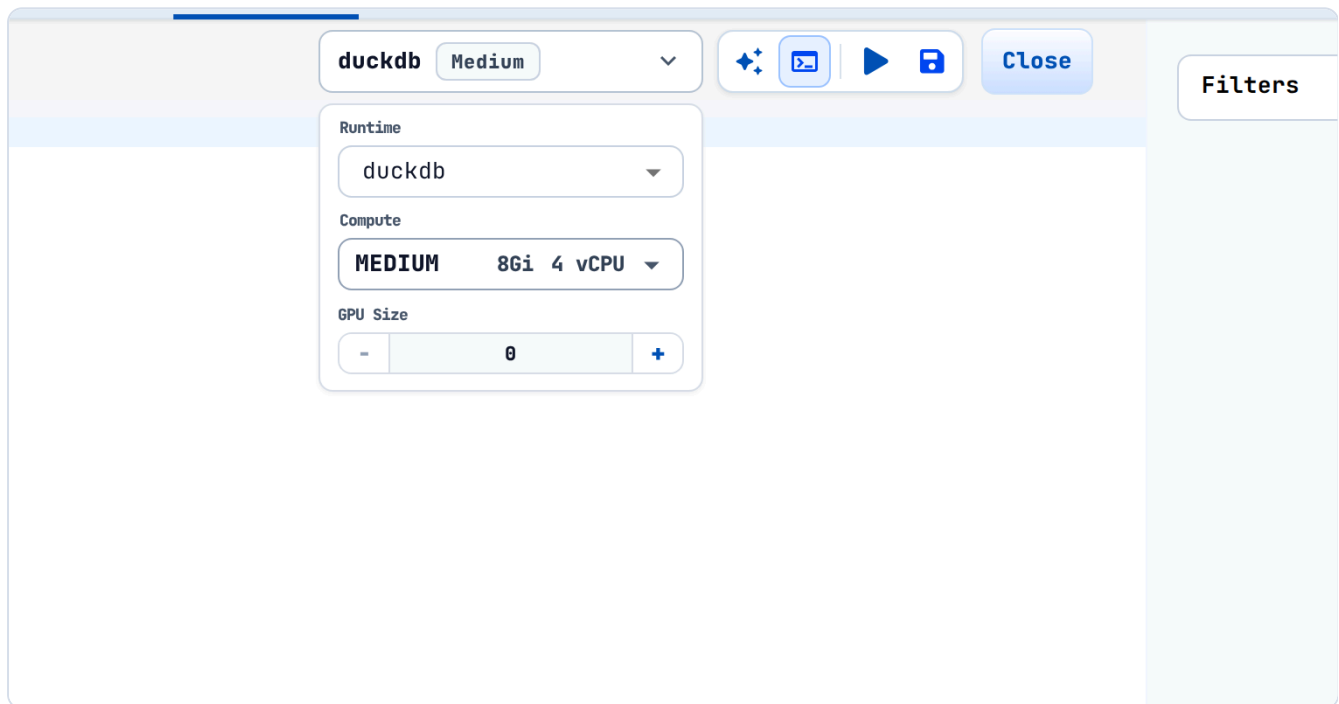
- **Runtime:** a dropdown listing the runtimes your administrators have enabled for the site. It shows `No RTEs Available` if none are enabled.
- **Compute:** a dropdown whose entries read `Name | memory | cpu`, one per compute size defined for the site. It shows `Insufficient Permission` if you cannot describe compute resources.

- **GPU Size** and, for Spark only, **Minimum Workers**: numeric steppers.

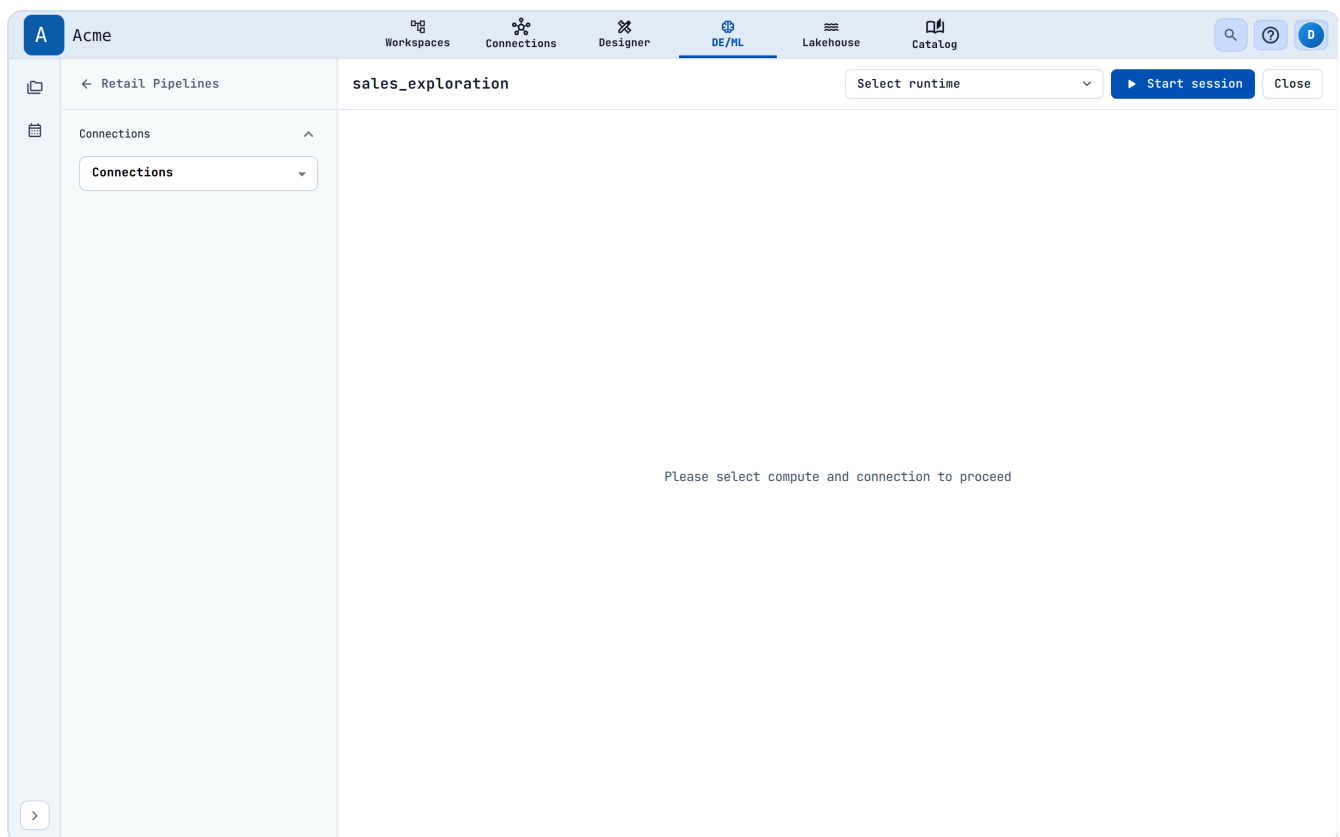
Where the control appears:

Place	Notes
Dataset designer	In the editor toolbar. Standard is offered only for no connection or a single SQL connection; DuckDB and Spark need a compute.
ETL job editor	In the toolbar next to Code . Choose DuckDB or Spark.
Notebook editor	Locked while the session runs; hovering shows <code>Stop the session to change compute</code> .
Job scheduler	A field labelled RTE that opens a Runtime and Compute menu; Save stays disabled until both are chosen.

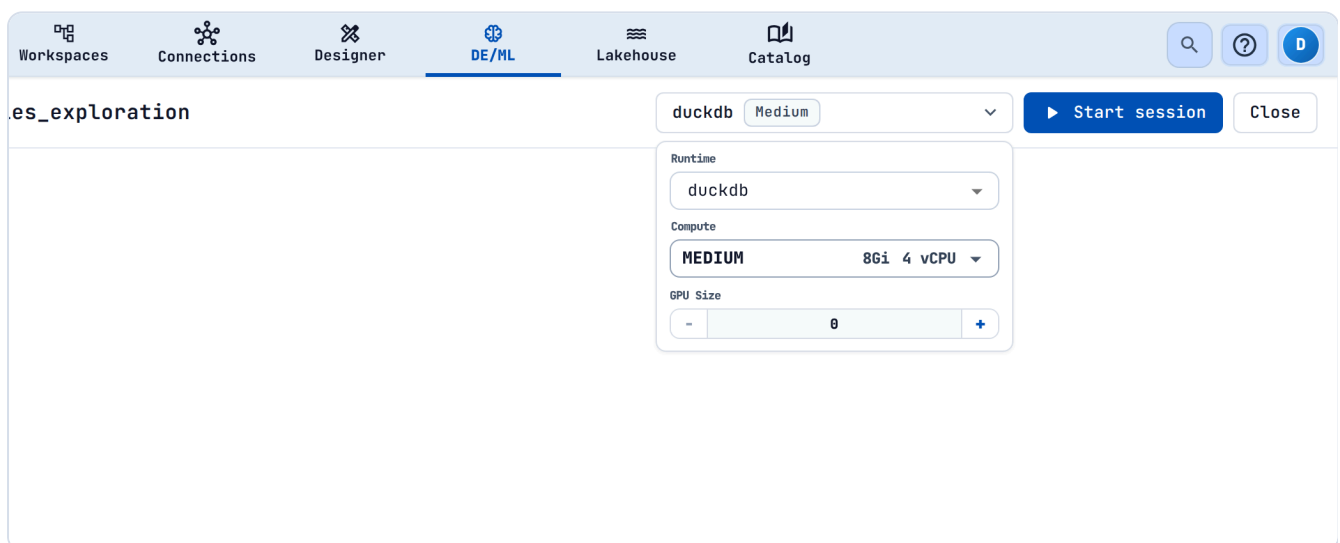
If you run a dataset without a compute, the designer shows `Please select Compute` or `Please select RTE and Compute`.



DuckDB selected: Compute and GPU Size, with no Minimum Workers control.



A notebook before its session starts: choose a runtime, then Start session.



Notebook runtime picker with duckdb and Medium compute.

3 How-to guides

3.1 Create and configure a session

`duckSession()` is the single entry point. It returns a `DuckSession`: a DuckDB connection plus every connector method (`readS3File`, `executeJdbc`, `readGzLakehouse` and so on).

PYTHON

```
from GZ.DuckDBUtils import duckSession

# Defaults: in-memory database, threads and memory sized from your compute.
session = duckSession()

# Persist the database to a file, pin resources, set DuckDB options.
session = duckSession(
    duckdbPath="/tmp/work.duckdb",
    logLevel="WARNING",
    threads=4,
    memory_limit="6GB",
    http_timeout="120",
)
```

Any extra keyword argument becomes a DuckDB `SET name='value'` statement, so every documented DuckDB setting is available. Settings that fail are logged as warnings and skipped.

A session is also a context manager, which closes the connection and releases connector clients when the block ends:

PYTHON

```
from GZ.DuckDBUtils import duckSession

with duckSession(logLevel="WARNING") as session:
    session.query("CREATE TABLE t AS SELECT range AS n FROM range(5)")
    print(session.query("SELECT sum(n) AS total FROM t", returnFormat="pandas"))
```

ONE SESSION PER RUN

Create the session once at the top of your code and reuse it. Each call to `duckSession()` opens a new DuckDB connection with its own tables, secrets and memory budget.

3.2 Read from connections

Every connector method follows the same conventions:

- the first argument is `connName`, the name of a Groundzero connection;
- `returnFormat` is `"duckdb"` (default, a lazy relation), `"pandas"` or `"arrow"`;
- `targetDuckdbTable`, when given, also registers the result as a table or view with that name so SQL can refer to it.

Object storage (S3, GCS, Azure Blob)

PYTHON

```
from GZ.DuckDBUtils import duckSession

session = duckSession()

# One Parquet file
orders = session.readS3File(
    connName="sales_s3",
    bucketName="acme-retail-data",
    filePath="raw/orders/2026-09-30.parquet",
)

# Many files with a glob, registered under a name
session.readS3File(
    connName="sales_s3",
    s3FullUrl="s3://acme-retail-data/raw/orders/*.parquet",
    targetDuckdbTable="orders_all",
)

# CSV with reader options; keys are DuckDB's own option names
customers = session.readS3File(
    connName="sales_s3",
    bucketName="acme-retail-data",
    filePath="raw/customers.csv",
    fileType="csv",
    readOptions={"delim": ",", "header": True},
)

print(session.query("SELECT count(*) AS n FROM orders_all",
    returnFormat="pandas"))
```

`readS3File` retries a failing read in two more modes before giving up: first with every column read as text (`all_varchar`), then with malformed rows ignored (`ignore_errors`). CSV reads default to a header row, a double quote character, null padding, relaxed strictness and a 100,000-row type-detection sample unless you override those options.

`readGcsFile(connName, bucketName, filePath, fileType, ...)` and `readAzureBlob(connName, containerName, blobName, fileType, ...)` take the same `fileType`, `readOptions`, `targetDuckdbTable` and `returnFormat` arguments.

SQL databases (JDBC)

`executeJdbc` runs a query on PostgreSQL, MySQL, Oracle, SQL Server, Db2, Redshift, CockroachDB, SAP HANA, TimescaleDB, Exasol and SQLite connections and loads the result into DuckDB. PostgreSQL, SQLite, Redshift and CockroachDB results stream in Arrow batches.

PYTHON

```
customers = session.executeJdbc(
    connName="orders_pg",
    sqlQuery="""
        SELECT customer_id, segment, signup_date
        FROM public.customers
        WHERE signup_date ≥ DATE '2025-01-01'
    """,
    targetDuckdbTable="customers",
)
```

Cloud warehouses

PYTHON

```
# Snowflake: results stream into a DuckDB table in Arrow batches.
n_rows = session.readSnowflake(
    conn_name="wh_snowflake",
    sql_query="SELECT * FROM ANALYTICS.PUBLIC.ORDERS",
    target_duckdb_table="orders_sf",
    returnFormat="row_count",
)

# BigQuery
events = session.readBigQuery(
    connName="analytics_bq",
    query="SELECT user_id, event FROM `acme.web.events` LIMIT 1000",
)
```

Note that `readSnowflake` uses snake_case argument names and requires the target table name. Because it streams one Arrow batch at a time, it can load tables far larger than Python memory.

APIs and other sources

`executeREST` calls an HTTP API configured as a REST connection and flattens the JSON records into a relation. `readKafka`, `readMongo`, `readDynamoDB`, `readSalesforce`, `readCassandra` and the rest of the connector table in the API reference follow the same pattern.

PYTHON

```
tickets = session.executeREST(
    connName="support_api",
    endpoint="/v2/tickets",
    params={"status": "open"},
    resultPath="data.items",
    maxPages=5,
)
```

3.3 Query across sources

Because every read returns a DuckDB relation, you can join data from different systems in one statement. Register each relation under a name, then use SQL.

PYTHON

```

from GZ.DuckDBUtils import duckSession

session = duckSession()
orders = session.readS3File(
    "sales_s3", "acme-retail-data", "raw/orders/2026-09-30.parquet"
)
customers = session.readS3File(
    "sales_s3", "acme-retail-data", "raw/customers.csv", fileType="csv"
)
session.register("orders", orders)
session.register("customers", customers)

top = session.query(
    """
    SELECT c.segment, count(*) AS orders, round(sum(o.amount), 2) AS revenue
    FROM orders o
    JOIN customers c USING (customer_id)
    GROUP BY c.segment
    ORDER BY revenue DESC
    """,
    returnFormat="pandas",
)
print(top)

```

`register` also accepts pandas DataFrames and Arrow tables, so you can mix DuckDB with pandas code:

PYTHON

```

import pandas as pd
from GZ.DuckDBUtils import duckSession

session = duckSession()
rates = pd.DataFrame({"region": ["EMEA", "APAC"], "fx": [1.08, 0.65]})
session.register("rates", rates)
print(session.query("SELECT * FROM rates ORDER BY region", returnFormat="pandas"))

```

3.4 Query the Groundzero Lakehouse

The Groundzero Lakehouse is an Apache Iceberg lakehouse. From the DuckDB runtime you read it through the lakehouse catalog. Table-level permissions, column masking and row policies are applied for the user running the code.

READ ONLY

Lakehouse access from DuckDB is read only. `writeGzLakehouse` raises `PermissionError`. To write lakehouse tables, use the Spark runtime.

Read a table or run SQL

PYTHON

```
from GZ.DuckDBUtils import duckSession

session = duckSession(siteName="acme")

# Whole table; the name is database.table
orders = session.readGzLakehouse(
    connName="lakehouse_main",
    tableName="retail.orders",
)

# SQL: always qualify tables as database.table
top = session.readGzLakehouse(
    connName="lakehouse_main",
    query="""
        SELECT region, sum(amount) AS revenue
        FROM retail.orders
        WHERE ordered_at ≥ DATE '2026-09-01'
        GROUP BY region
    """,
    returnFormat="pandas",
)
```

Rules the runtime enforces:

- Provide `tableName` or `query`. If you give neither, the call fails with `Either 'query' or 'tableName' must be provided`.
- Only read-only statements run (`SELECT`, `SHOW`, `DESCRIBE`, `EXPLAIN`). Anything else fails with `Only read-only queries are allowed`.
- Table references in a query must be fully qualified. An unqualified table raises `Use fully qualified table names`.
- A connection that points at an external lakehouse needs a site name: pass `siteName` to `duckSession()` (one site for all lakehouse connections) or `siteNames={"lakehouse_main": "acme"}` for a mapping, or pass `siteName` to `readGzLakehouse`. One session can use only one site per connection; to use another site, create a new session.

Filters from the UI

`readGzLakehouse` accepts structured `filters` that are added to the query as extra predicates, which keeps the query text stable while values change:

PYTHON

```
sales = session.readGzLakehouse(
    connName="lakehouse_main",
    query="SELECT * FROM retail.orders",
    filters=[
        {"column": "region", "op": "in", "values": ["EMEA", "APAC"]},
        {"column": "amount", "op": "gte", "values": [100]},
    ],
)
```

Supported operators are `in`, `eq`, `between`, `gte`, `gt`, `lte` and `lt`. A malformed filter raises an error; the query never runs unfiltered.

Keep a lakehouse table in the session

`registerLakehouseTable` reads a table once and keeps it for repeated queries. The table is named `lakehouse_<alias>`:

PYTHON

```
session.registerLakehouseTable(
    connName="lakehouse_main",
    tableName="retail.customers",
    alias="customers",
)
vip = session.query(
    "SELECT * FROM lakehouse_customers WHERE segment = 'vip' LIMIT 10"
)
```

By default the table is materialized as a physical DuckDB table. Pass `materialize=False` to register it as a view over the Arrow data instead.

Time travel and snapshots

Groundzero Lakehouse tables are Iceberg tables. Use DuckDB's Iceberg syntax in your `query` where the extension supports it.

3.5 Write results

DuckDB writes with `COPY`, so results stream to the destination without materializing in Python.

PYTHON

```
from GZ.DuckDBUtils import duckSession

session = duckSession()
session.readS3File(
    "sales_s3", "acme-retail-data", "raw/orders/2026-09-30.parquet",
    targetDuckdbTable="orders",
)
session.query("""
    CREATE TABLE summary AS
    SELECT region, count(*) AS orders FROM orders GROUP BY region
""")

# Write a DuckDB table
session.writeS3File(
    connName="sales_s3",
    bucketName="acme-retail-data",
    filePath="curated/summary.parquet",
    fileType="parquet",
    sourceTable="summary",
)

# Write a query result as CSV with options
session.writeS3File(
    connName="sales_s3",
    s3FullUrl="s3://acme-retail-data/curated/summary.csv",
    fileType="csv",
    sourceQuery="SELECT * FROM summary ORDER BY region",
    writeOptions={"header": True},
)
```

For databases use `writeJdbc(connName, targetTable, data=None, sourceTable=None, sourceQuery=None, mode="append")`, where `mode` is `append` or `replace`:

PYTHON

```
session.writeJdbc(
    connName="orders_pg",
    targetTable="reporting.region_summary",
    sourceTable="summary",
    mode="replace",
)
```

Most other connectors have a matching `write ...` method; see the connector table in the API reference. Supported file types for `writeS3File`, `writeGcs` and `writeAzureBlob` are `parquet`, `csv` and `json`; any other value raises `Unsupported file type`.

3.6 Return output and previews

When code runs as a dataset (or any run that returns rows), Groundzero looks for output in two places, in this order of priority:

1. a value passed to `gz_output(value)`; if called more than once, the last value wins;
2. a variable named `outputDF`.

If both exist, `gz_output()` is used and a warning is logged.

The accepted types and what happens to them:

Type	Handling
DuckDB relation	Limited to 100,000 rows, converted through Arrow
<code>pyarrow.Table</code> , <code>RecordBatch</code> , <code>RecordBatchReader</code>	Sliced to 100,000 rows
pandas DataFrame	<code>head(100000)</code> ; datetimes become <code>YYYY-MM-DD HH:MM:SS</code> strings, numeric NaN becomes <code>"NaN"</code>
list	First 100,000 items; dict items become rows

Any other type fails with `outputDF` must be `DuckDBPyRelation`, `pa.Table`, `pa.RecordBatch`, `pa.RecordBatchReader`, `DataFrame`, or `list`.

The schema returned with the rows uses these type names: `integer`, `long`, `short`, `byte`, `string`, `boolean`, `double`, `float`, `date`, `timestamp`, `time`, `binary`, `decimal`, `interval`, `array`, `json` and `null`.

Previews and dashboards page through the result: the response carries `totalRecords`, `totalPages`, `pageNumber`, `hasMore` and the `data` for the requested page. Asking for a page past the end fails with `Page N is out of range. Total pages: X for page size Y.`

IDENTICAL CODE RETURNS CACHED ROWS

Dataset output is cached by the exact text of your code. Re-running unchanged code returns the cached rows, without reading the source again, for up to 15 minutes. Results that read governed lakehouse tables are cached per user policy. Change the code (for example a comment) or wait for the cache to expire to force a refresh.

PYTHON

```
from GZ.DuckDBUtils import duckSession

session = duckSession()
orders = session.readS3File(
    "sales_s3", "acme-retail-data", "raw/orders/2026-09-30.parquet"
)
session.register("orders", orders)

# Either of these returns rows; do not use both.
gz_output(session.query("SELECT region, count(*) AS n FROM orders GROUP BY 1"))
```

3.7 Use parameters and placeholders

Groundzero substitutes values into your code text before it runs. There are two kinds.

Runtime placeholders

These describe the run and are available in every runtime. Write them as `${NAME}` or `{NAME}`.

Placeholder	Value
STRTTM	Run start time
ENDTM	Time the placeholder is evaluated
CLNTNM	Client name
SITENM	Site name
PRJNM	Project name
JOBNM	Job name
RTE	Runtime name
RGN	Region
USR	User running the code
USRGRPS	The user's groups, comma separated, each quoted

Each placeholder expands to a Python string literal that already contains single quotes, so it can be dropped straight into SQL text. For a project named `Sales`, the line

PYTHON

```
project = ${PRJNM}
```

becomes `project = "'Sales'"`, a string whose value is `'Sales'` including the quotes. Use it like this:

PYTHON

```
audit = session.query(f"SELECT * FROM audit WHERE project = {project}")
```

BRACE NAMES ARE REPLACED EVERYWHERE

`{PRJNM}` is replaced even inside an f-string or a SQL snippet, and the placeholder names are case sensitive. Do not name your own variables with these upper-case names.

Job parameters

When you start a job through the external job API you can pass your own parameters as `placeholders`. Each key is `$` followed by a name made of letters, digits and underscores. Values may be a string, number, boolean, null or a list of those. Every `$NAME` in the job's code is replaced by the matching Python literal before the run:

PYTHON

```
cutoff = $CUTOFF_DATE
regions = $REGIONS
orders = session.query(
    f"SELECT * FROM orders WHERE ordered_at ≥ DATE '{cutoff}'"
)
```

Request body (POST to the external job endpoint):

JSON

```
{
  "projectName": "Retail Analytics",
  "jobName": "daily_orders",
  "rteName": "duckdb",
  "computeName": "Medium",
  "placeholders": {
    "$CUTOFF_DATE": "2026-09-01",
    "$REGIONS": ["EMEA", "APAC"]
  }
}
```

Placeholder names that are not valid or values of unsupported types are rejected before anything runs (`Invalid placeholder name: ...`, `Invalid value for placeholder: ...`).

3.8 Run jobs and schedule them

An ETL job is a Python file saved in a project. To run it:

1. Open the job in the ETL job editor and choose the runtime and compute in the toolbar (see the Quickstart).
2. Run it once from the editor to check the output pane and logs.
3. Open the scheduler, pick the project and job, and set:
 - **Schedule type:** `Batch` or `Streaming`. A batch schedule adds a recurrence (cron-style) and an **Ends** choice of `Never` or `On` a date and time.

- **RTE:** the runtime and compute to use (both are required).
- **Name and Description.**

When a schedule has an end time, the run stops at that time. A job ends when your code finishes.

To chain jobs, call `executeJob` from a parent job. It runs the named sub-jobs of the current multi-job run in order and returns their results:

PYTHON

```
from GZ.JobUtils import executeJob

results = executeJob(["load_orders", "build_summary"])
```

`executeJob` raises `ValueError` if none of the names match a sub-job of the current multi-job run.

3.9 Use extra packages

Use this when your code needs a library the runtime does not include. You have three options, from lightest to heaviest.

1. **An ML variant:** choose the DuckDB scikit-learn, PyTorch or TensorFlow runtime when the library you need is listed there (see Machine-learning variants).
2. **A custom runtime:** administrators can build a runtime from any base runtime plus your own `requirements.txt`. Open **Custom RTE** in the admin area and choose **Add custom RTE**:
 - **Base Image:** pick the runtime to extend, for example the DuckDB scikit-learn runtime.
 - **Name:** a short name. The runtime kind is added as a prefix, so a name of `forecasting` on a DuckDB runtime is shown as `duckdb-forecasting`.
 - **Choose File:** a `.txt` file in `requirements.txt` format.

The page lists each custom runtime with its creator, creation time and status. The build can take a while, and the status changes when it finishes. Once it succeeds, choose the new runtime like any other.

3. **Install in a notebook:** `pip install` works for the current session. Packages installed this way last only until you stop the session, so move them to a custom runtime for jobs.

To check a package is available, import it in a notebook cell.

LEAVE THE PINNED PACKAGES ALONE

The runtime pins NumPy, pandas, scikit-learn and DuckDB. A custom `requirements.txt` that upgrades them can break the `GZ` library, so add new packages and leave those alone.

3.10 Work with files

Your code can use local scratch space. The sandbox allows reading and writing under `/tmp`, `/home/workspace` and `/home`, and rejects other locations:

PYTHON

```
import pandas as pd

df = session.query("SELECT 1 AS a, 2 AS b", returnFormat="pandas")
df.to_csv("/tmp/scratch.csv", index=False)
back = session.query("SELECT * FROM read_csv('/tmp/scratch.csv')")
```

Files written locally disappear when the run ends. Write anything you want to keep to an object store or database connection.

Job and dataset code runs with restrictions. Modules that give network or process access (`socket` , `urllib` , `http` and others) cannot be imported, `subprocess` is blocked, `eval` , `exec` and `compile` are not allowed, and secret values in `os.environ` read as `****` . Use connections for network access.

3.11 Tune performance

DuckDB is fast by default. These are the levers that matter.

1. **Select columns and filter early.** Parquet is columnar and DuckDB reads only the columns and row groups a query needs. Prefer `SELECT col_a, col_b` and `WHERE` clauses to `SELECT *` .
2. **Stay lazy.** Relations and SQL do not copy data into Python. Calling `.df()` or `returnFormat="pandas"` on a large table pulls it into memory.
3. **Read many files with a glob.** `s3FullUrl="s3://bucket/path/*.parquet"` lets DuckDB parallelize across files. Pass `readOptions={"hive_partitioning": True}` for partitioned layouts.
4. **Aggregate before returning.** A dataset returns at most 100,000 rows; do the grouping in SQL.
5. **Right-size the compute.** The DuckDB memory limit is 80 percent of the compute's memory. When a query needs more it spills to disk under `/tmp/duckdb-spill` , which is slower. If a run is slow and logs show spilling, choose a larger compute.
6. **Control threads and memory per session** when several heavy queries share a run:


```
duckSession(threads=4, memory_limit="6GB").
```
7. **Convert to Arrow, not pandas.** `relation.to_arrow_table()` is zero-copy friendly and works with Parquet writers and Polars.
8. **Materialize reused data.** If a relation is queried many times, create a table once with `CREATE TABLE x AS ...` (or `targetDuckdbTable`) so DuckDB does not re-read the source for every query.

PYTHON

```

from GZ.DuckDBUtils import duckSession

session = duckSession(threads=2, memory_limit="1GB")

# Materialize once, query many times
session.readS3File(
    "sales_s3", s3FullUrl="s3://acme-retail-data/raw/orders/*.parquet",
    targetDuckdbTable="orders_remote",
)
session.query("""
    CREATE TABLE orders_local AS
    SELECT order_id, region, amount FROM orders_remote
""")
print(session.query("SELECT region, avg(amount) AS avg_amount "
    "FROM orders_local GROUP BY region ORDER BY 1",
    returnFormat="pandas"))
print(session.query("SELECT current_setting('threads') AS threads",
    returnFormat="pandas"))

```

3.12 Log and debug

Anything your code prints is captured and shown in the job's log view with the job identifier prefix. The standard `logging` module writes to the same stream. Very long lines (over 200,000 characters) are split into numbered `LARGE_OUTPUT_PART_n_OF_m` chunks so the UI can reassemble them.

When your code raises an exception, the run fails and the error is reduced to the line in your code that raised it, for example:

TEXT

```
ValueError: Either s3FullUrl or both bucketName and filePath must be provided at line 7
```

For syntax errors the message ends with the offending line number. Use `print(session.query("EXPLAIN ...", returnFormat="pandas"))` to inspect query plans and `session.listTables()` to see what is registered.

To raise the verbosity of the helper library, pass `logLevel="DEBUG"` to `duckSession`.

3.13 Handle secrets

Never put keys or passwords in code. Create a connection in Groundzero and refer to it by name. Groundzero supplies the credentials to the helper library, and your code never sees them. In jobs and datasets, `os.environ` shows secret values as `****`. Reading Groundzero's own lakehouse storage buckets (names starting with `gz-lakehouse-`) directly is refused with `Direct access to gz-lakehouse-* buckets is restricted; use the GZLAKEHOUSE connector`.

If you need a native client for a connection, ask the helper library for one:

PYTHON

```
from GZ.Client import getClient

s3 = getClient().getS3Client("sales_s3")
resp = s3.list_objects_v2(Bucket="acme-retail-data", Prefix="raw/", MaxKeys=5)
```

3.14 Extract text from PDFs

`parsePDF` and `parsePDFCorpus` extract text and tables from PDFs (Docling with RapidOCR) and write results to S3.

PYTHON

```
result = session.parsePDFCorpus(
    s3ConnName="sales_s3",
    sourcePrefix="incoming/contracts/",
    outputPrefix="parsed/contracts",
    pageRangeSize=64,
)
print(result["status"], result["documentCount"], result["manifestUri"])
```

To check it worked, `result["status"]` reports the outcome and `manifestUri` points to the manifest of parsed documents.

Discovery is recursive and case insensitive. Results are restartable: running again with the same source and output skips page ranges that were already committed. OCR processes one document model at a time, so throughput is bounded by that. For distributed OCR over very large corpora use the Spark runtime.

3.15 Run Groundzero Agents over documents

`invokeAgent` calls an approved Groundzero Agent and waits for its result; `invokeAgentPDFCorpus` runs an Agent over every document in an S3 prefix and writes one result per document.

PYTHON

```
result = session.invokeAgentPDFCorpus(
    connName="agents",
    agentId="00000000-0000-4000-8000-000000000001",
    message="List each policy's effective date and exceptions.",
    pdfSourceConnName="sales_s3",
    pdfSourcePrefix="incoming/policies/",
    pdfSinkConnName="sales_s3",
    pdfSinkPrefix="agent-results/policies",
    summaryWorkers=4,
)
print(result["status"], result["resultsUri"])
```

The corpus run resolves the approved Agent version and its response schema before any OCR starts. With a JSON response schema configured the output mode is structured; without one the Agent returns cited facts. `summaryWorkers` runs between 1 and 32 documents concurrently (default 4), the document limit defaults to 10,000 and the Agent-call budget to 50,000. Unchanged documents are skipped on later runs.

3.16 Check data quality

`runDataQualityChecks`, `runTableDataQualityChecks` and `runQueryDataQualityChecks` run Qualipilot against a relation, a table or a query using the session's own DuckDB connection, and can add an LLM-written narrative through a Groundzero LLM connection:

PYTHON

```
report = session.runTableDataQualityChecks(  
    tableName="orders_local",  
    connectionName="llm_openai",  
    includeLlm=True,  
)  
session.saveDataQualityReport(report, "/tmp/orders_quality.html", "html")
```

Set `includeLlm=False` to run only the deterministic checks.

PYTHON

```
from GZ.DuckDBUtils import duckSession  
  
session = duckSession()  
session.query("""  
    CREATE TABLE orders_local AS  
    SELECT range AS order_id, 'EMEA' AS region, range * 1.5 AS amount  
    FROM range(100)  
""")  
  
report = session.runTableDataQualityChecks(  
    tableName="orders_local",  
    includeLlm=False,  
)  
session.saveDataQualityReport(report, "/tmp/orders_quality.json")
```

4 API reference

This reference documents the helper library available inside the DuckDB runtime. Everything below is importable in jobs, datasets and notebooks without installing anything.

4.1 Conventions

All connector methods are called on the session object returned by `duckSession()`. They share these conventions.

Convention	Meaning
<code>connName</code>	Name of a Groundzero connection. Credentials never appear in code.
<code>returnFormat</code>	"duckdb" (default), "pandas" or "arrow".
<code>targetDuckdbTable</code>	Optional name; the result is also registered under it so SQL can use it.
Lazy relations	A "duckdb" result is a <code>DuckDBPyRelation</code> ; it is evaluated when consumed.

WHAT ARROW RETURNS

`returnFormat="arrow"` returns a `pyarrow.RecordBatchReader` from `query()` and from most connector reads, but a `pyarrow.Table` from `readGzLakehouse`. Call `.read_all()` on a reader to get a table, or use `relation.to_arrow_table()`.

4.2 Session

duckSession

PYTHON

```
def duckSession(
    duckdbPath=":memory:", logLevel="INFO", threads=None, memory_limit=None,
    siteName=None, siteNames=None, **kwargs
) → DuckSession: ...
```

Creates a configured session: opens a DuckDB connection, applies performance settings, attaches Groundzero Lakehouse catalogs for lakehouse connections and returns the session.

Parameter	Type	Default	Description
<code>duckdbPath</code>	str	<code>":memory:"</code>	Database file path, or <code>:memory:</code>
<code>logLevel</code>	str	<code>"INFO"</code>	Python log level name
<code>threads</code>	int	from compute	Pins DuckDB threads for this session
<code>memory_limit</code>	str	from compute	Pins DuckDB memory, e.g. <code>"8GB"</code>
<code>siteName</code>	str or list	<code>None</code>	Lakehouse site for REST-backed connections

Parameter	Type	Default	Description
<code>siteNames</code>	dict or list	<code>None</code>	Per-connection lakehouse sites
<code>**kwargs</code>	any		Extra DuckDB <code>SET name='value'</code> options

`siteNames` is preferably a mapping `{"connectionName": "siteName"}`. A list of `{"connName": ..., "siteName": ...}` entries is also accepted. A single site name applies to every REST-backed lakehouse connection; a list of site names must have one entry per connection or the mapping is ignored with a warning.

The session sets `preserve_insertion_order=false`, turns off the progress bar, turns on metadata caches for objects, HTTP and Parquet, and spills to `/tmp/duckdb-spill`.

Returns a `DuckSession`.

Raises `AttributeError` if `logLevel` is not a valid level name. Failures applying individual settings are logged as warnings, not raised.

PYTHON

```
from GZ.DuckDBUtils import duckSession

session = duckSession(logLevel="WARNING", threads=2, memory_limit="2GB")
print(session.query("SELECT current_setting('memory_limit') AS m",
                    returnFormat="pandas"))
session.close()
```

query

PYTHON

```
def query(sqlQuery, params=None, returnFormat="duckdb"): ...
```

Runs SQL against everything registered in the session.

Parameter	Type	Default	Description
<code>sqlQuery</code>	str	required	SQL text
<code>params</code>	list or dict	<code>None</code>	Values for prepared-statement parameters
<code>returnFormat</code>	str	<code>"duckdb"</code>	<code>"duckdb"</code> , <code>"pandas"</code> or <code>"arrow"</code>

Returns a relation, a pandas DataFrame or an Arrow reader. A statement that returns no result set (for example `CREATE TABLE`) returns the underlying DuckDB connection object.

Raises whatever DuckDB raises for invalid SQL (`duckdb.ParserException`, `duckdb.CatalogException`, ...); the error and the query text are logged first.

PYTHON

```
rows = session.query(
    "SELECT * FROM orders WHERE region = $r AND amount > $min",
    params={"r": "EMEA", "min": 100},
    returnFormat="pandas",
)
```

register and unregister

PYTHON

```
def register(name, data): ...
def unregister(name): ...
```

`register` exposes a pandas DataFrame, Arrow table, relation or other object DuckDB can scan as a view called `name`. `unregister` removes it. Both return `None` and re-raise DuckDB errors.

listTables and describeTable

PYTHON

```
def listTables() → DataFrame: ...
def describeTable(tableName) → DataFrame: ...
```

`listTables` returns a DataFrame with a `name` column. `describeTable` returns DuckDB's `DESCRIBE` output (`column_name`, `column_type`, `null`, `key`, `default`, `extra`). It raises `ValueError("Invalid table name: ...")` if the name contains characters other than letters, digits, underscores and dots.

registerLakehouseTable

PYTHON

```
def registerLakehouseTable(connName, tableName, alias=None, materialize=True): ...
```

Reads a Groundzero Lakehouse table and keeps it in the session as `lakehouse_<alias>`.

Parameter	Type	Default	Description
<code>connName</code>	str	required	Lakehouse connection
<code>tableName</code>	str	required	<code>database.table</code> name
<code>alias</code>	str	table name	Suffix; dots and dashes become underscores
<code>materialize</code>	bool	True	True copies into a table; False registers a view

Returns `None`. **Raises** the errors of `readGzLakehouse`.

close

PYTHON

```
def close(): ...
```

Closes connector clients, unregisters session views and closes the DuckDB connection. Using the session as a `with` block calls it for you. Errors while closing are logged, not raised.

4.3 Groundzero Lakehouse

readGzLakehouse

PYTHON

```
def readGzLakehouse(
    connName, query=None, returnFormat="duckdb", tableName=None, siteName=None,
    filters=None
): ...
```

Aliases: `executeGzLakehouse`, `readGZLAKEHOUSE`.

Runs a read-only query on a lakehouse connection and returns the result.

Parameter	Type	Default	Description
<code>connName</code>	str	required	Groundzero Lakehouse connection
<code>query</code>	str	None	Read-only SQL; tables as <code>database.table</code>
<code>returnFormat</code>	str	"duckdb"	"duckdb", "pandas" or "arrow" (a Table)
<code>tableName</code>	str	None	Reads <code>SELECT *</code> from this table when no <code>query</code>
<code>siteName</code>	str	None	Site for this connection
<code>filters</code>	list of dict	None	Structured predicates, see below

Each filter is `{"column": str, "op": str, "values": list}` with an optional `"table"`. Operators: `in`, `eq`, `between`, `gte`, `gt`, `lte`, `lt`.

The call authorizes every referenced table for the caller, applies the caller's column masks and row policies by rewriting the query, runs it, and records an audit entry.

Returns a relation, DataFrame or Arrow table.

Raises

Exception	When
<code>ValueError</code>	Neither <code>query</code> nor <code>tableName</code> ; non-read-only SQL; unparsable SQL; missing site name; unknown warehouse; site already bound
<code>PermissionError</code>	Authentication fails, or the caller lacks <code>Select</code> on a table, or tables are not fully qualified
<code>ConnectionError</code>	The lakehouse cannot be reached
<code>RuntimeError</code>	Other lakehouse failures

PYTHON

```
df = session.readGzLakehouse(
    connName="lakehouse_main",
    query="SELECT region, count(*) AS n FROM retail.orders GROUP BY region",
    returnFormat="pandas",
)
```

Writing to the lakehouse

DuckDB reads the lakehouse only. `writeGzLakehouse` (alias `writeGZLAKEHOUSE`) raises `PermissionError` so code written for another runtime fails clearly. Use the Spark runtime to write lakehouse tables, or write results to S3 or a database from DuckDB.

getLakehouseCatalog

PYTHON

```
def getLakehouseCatalog(connName, siteName=None) → str: ...
```

Attaches the connection's REST catalog if needed and returns its DuckDB catalog alias (of the form `gzlakehouse__<connection>`), so you can write fully qualified SQL against `session.query`:

PYTHON

```
catalog = session.getLakehouseCatalog("lakehouse_main", siteName="acme")
rel = session.query(f'SELECT count(*) FROM "{catalog}".retail.orders')
```

Raises `ValueError` if the connection has no lakehouse URL (it is not REST-backed) or no site can be determined.

NOTE

Queries issued through `session.query` against an attached catalog do not go through the authorization, masking and audit steps of `readGzLakehouse`. Use `readGzLakehouse` for governed access.

setLakehouseSite and configureGzLakehouseCatalogs

PYTHON

```
def setLakehouseSite(connName, siteName): ...
def configureGzLakehouseCatalogs(connectionSiteNames=None, failFast=False): ...
```

`setLakehouseSite` binds a REST-backed connection to a site and attaches its catalog (`ValueError` if `siteName` is empty or the connection is not REST-backed). `configureGzLakehouseCatalogs` attaches the catalogs of every REST-backed connection; `duckSession` calls it for you. With `failFast=False` (default) an attach failure is deferred to first use.

4.4 Object storage

readS3File

PYTHON

```
def readS3File(
    connName, bucketName=None, filePath="", fileType="parquet", s3FullUrl=None,
    targetDuckdbTable=None, readOptions=None, returnFormat="duckdb"
): ...
```

Reads Parquet, CSV or JSON from Amazon S3 with DuckDB's native reader.

Parameter	Type	Default	Description
<code>connName</code>	str	required	S3 connection
<code>bucketName</code>	str	None	Bucket; use with <code>filePath</code>
<code>filePath</code>	str	""	Key or glob inside the bucket
<code>fileType</code>	str	"parquet"	parquet, csv or json (other values read as CSV)
<code>s3FullUrl</code>	str	None	Full <code>s3://</code> URL or glob; replaces bucket and path
<code>targetDuckdbTable</code>	str	None	Register the result under this name
<code>readOptions</code>	dict	None	Reader options, e.g. <code>{"delim": ";"}</code>
<code>returnFormat</code>	str	"duckdb"	"duckdb", "pandas" or "arrow"

`readOptions` keys are passed to the DuckDB reader as named arguments, so they must be DuckDB's own option names (for CSV: `delim`, `header`, `quote`, `dateformat`, `skip`, and so on; `delimiter` is not valid). Booleans, numbers and strings are accepted. The call retries failed reads with `all_varchar` and then `ignore_errors` before raising the last error.

Raises `ValueError` for Either `s3FullUrl` or both `bucketName` and `filePath` must be provided or incomplete connection details, and DuckDB errors (for example `duckdb.IOException`) when the object cannot be read.

writeS3File

PYTHON

```
def writeS3File(
    connName, bucketName=None, filePath="", fileType="parquet", s3FullUrl=None,
    sourceTable=None, sourceQuery=None, writeOptions=None
): ...
```

Alias: `writeS3`. Writes a table or query result with `COPY ... TO`.

Parameter	Type	Default	Description
<code>connName</code>	str	required	S3 connection
<code>bucketName</code> , <code>filePath</code>	str	None	Destination, or use <code>s3FullUrl</code>

Parameter	Type	Default	Description
<code>fileType</code>	str	"parquet"	parquet, csv or json
<code>sourceTable</code>	str	None	Name of a table or view in the session
<code>sourceQuery</code>	str	None	SQL whose result is written; wins over <code>sourceTable</code>
<code>writeOptions</code>	dict	None	COPY options, e.g. {"compression": "zstd"}

Returns `None`. **Raises** `ValueError` for a missing destination (Either `s3FullUrl` or both `bucketName` and `filePath` must be provided), a missing source (Either `sourceTable` or `sourceQuery` must be provided) or an unsupported type (`Unsupported file type: ...`).

4.5 SQL databases

executeJdbc

PYTHON

```
def executeJdbc(
    connName, sqlQuery, params=None, targetDuckdbTable=None, returnFormat="duckdb"
): ...
```

Runs a query on a SQL database connection and loads the result.

Parameter	Type	Default	Description
<code>connName</code>	str	required	A PostgreSQL, MySQL, Oracle, SQL Server, Db2, Redshift, CockroachDB, SAP, Exasol, TimescaleDB or SQLite connection
<code>sqlQuery</code>	str	required	Query in the database's own dialect
<code>params</code>	list or dict	None	Bound parameters
<code>targetDuckdbTable</code>	str	None	Register the result under this name
<code>returnFormat</code>	str	"duckdb"	"duckdb", "pandas" or "arrow"

Returns the result in the requested format. **Raises** `ValueError` (`Unsupported database type: ...`) for connection types it cannot map, and the driver's errors for failing SQL. If the ADBC driver for PostgreSQL-compatible databases fails, the call falls back to SQLAlchemy and pandas automatically.

writeJdbc

PYTHON

```
def writeJdbc(
    connName, targetTable, data=None, sourceTable=None, sourceQuery=None,
    mode="append"
): ...
```

Parameter	Type	Default	Description
<code>connName</code>	str	required	SQL database connection
<code>targetTable</code>	str	required	Destination table
<code>data</code>	relation, DataFrame or Table	None	Data to write
<code>sourceTable</code>	str	None	Session table to write
<code>sourceQuery</code>	str	None	Query to write
<code>mode</code>	str	"append"	append or replace

Provide one of `data`, `sourceTable` or `sourceQuery`. PostgreSQL-compatible and MySQL targets are written through DuckDB's `postgres` and `mysql` extensions; other databases go through SQLAlchemy in chunks of 10,000 rows.

Returns `True`. **Raises** `ValueError` (Either `data`, `sourceTable`, or `sourceQuery` must be provided) or the database's errors. `replace` drops and recreates the table.

getSqlEngine

PYTHON

```
def getSqlEngine(connName) → sqlalchemy.Engine: ...
```

Returns a SQLAlchemy engine for a JDBC connection, for code that needs the native API. Dispose of it when you are done.

4.6 Cloud warehouses

readSnowflake

PYTHON

```
def readSnowflake(
    conn_name, sql_query, target_duckdb_table, warehouse=None,
    force_microsecond_precision=False, client_prefetch_threads=4,
    returnFormat="duckdb"
): ...
```

Streams a Snowflake query into a DuckDB table in Arrow batches, so Python holds only one batch at a time.

Parameter	Type	Default	Description
<code>conn_name</code>	str	required	Snowflake connection
<code>sql_query</code>	str	required	Query to run
<code>target_duckdb_table</code>	str	required	Table to create or replace
<code>warehouse</code>	str	None	Overrides the connection's warehouse
<code>force_microsecond_precision</code>	bool	False	Cast timestamps to microseconds; use for dates outside 1677-2262

Parameter	Type	Default	Description
<code>client_prefetch_threads</code>	int	4	Download threads
<code>returnFormat</code>	str	"duckdb"	"duckdb", "pandas", "arrow" or "row_count"

Raises `ValueError` for an unsupported `returnFormat` (`Unsupported returnFormat. Use 'duckdb', 'pandas', 'arrow', or 'row_count'.`) and Snowflake connector errors for failing SQL.

4.7 Outputs and run context

gz_output and outputDF

PYTHON

```
def gz_output(value): ...
```

Available as a global in datasets and jobs run by the runtime (it is not defined in notebooks). Records `value` as the run's output. If called repeatedly the last value is used. A variable named `outputDF` is the alternative; when both exist `gz_output` wins. See How-to guides, Return output and previews, for the accepted types and limits.

Placeholders

`${NAME}` or `{NAME}` for `STRTTM`, `ENDTM`, `CLNTNM`, `SITENM`, `PRJNM`, `JOBNM`, `RTE`, `RGN`, `USR` and `USRGRPS`, and `$NAME` for job parameters passed through the external job API. See How-to guides, Use parameters and placeholders.

executeJob

PYTHON

```
def executeJob(jobNames: list[str]) → dict: ...
```

Import: `from GZ.JobUtils import executeJob`. Runs the named sub-jobs of the current multi-job run in sequence and returns `{jobName: result}`.

Parameter	Type	Description
<code>jobNames</code>	list of str	Names of sub-jobs to run

Raises `ValueError` if none of the names match the run's sub-jobs, and re-raises a sub-job's failure after recording it as failed.

sendEmail

PYTHON

```
def sendEmail(subject, body, to_email, from_email, **kwargs): ...
```

Import: `from GZ.SendEmail import sendEmail`. Sends an email through the Groundzero notification service.

Parameter	Type	Description
<code>subject</code>	<code>str</code>	Subject line
<code>body</code>	<code>str</code>	Message body
<code>to_email</code>	<code>str</code>	Recipient address
<code>from_email</code>	<code>str</code>	Sender address
<code>**kwargs</code>	<code>any</code>	Extra fields added to the message

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

PYTHON

```
from GZ.SendEmail import sendEmail

sendEmail(
    subject="Daily revenue ready",
    body="The 2026-09-30 revenue table was written.",
    to_email="data-team@example.com",
    from_email="noreply@example.com",
)
```

4.8 Clients and connection details

getClient

PYTHON

```
def getClient() → ClientSession: ...
```

Import: `from GZ.Client import getClient` . Returns an object with one factory per connector that builds the vendor's native client from a connection, for the cases where the session methods are not enough.

Method	Returns
<code>getS3Client(connName)</code>	<code>boto3</code> S3 client
<code>getS3FSCClient(connName)</code>	<code>s3fs.S3FileSystem</code>
<code>getSQLEngine(connName)</code>	SQLAlchemy engine
<code>getSnowflakeClient(connName)</code>	Snowflake connection
<code>getMongoClient(connName)</code>	<code>pymongo</code> client
<code>getBigQueryClient(connName)</code>	BigQuery client
<code>getAzureBlobClient(connName)</code>	Azure Blob service client
<code>getGCSCClient(connName)</code>	Google Cloud Storage client
<code>getDynamoDBClient(connName)</code>	DynamoDB client
<code>getKinesisClient(connName)</code>	Kinesis client

The class also provides `getSalesforceConnection`, `getCassandraSession`, `getSFTPClient`, `getRedshiftClient`, `getS3SelectClient`, `getCosmosDBClient`, `getElasticClient`, `getExasolConnection`, `getTimescaleClient`, `getPineconeClient`, `getTabularCatalog`, `getFireboltClient`, `getTeradataClient`, `getNeo4JClient`, `getInfluxDBClient`, `getDuckDBClient`, `getBusinessCentralClient`, `getShopifyClient`, `getMarketoClient`, `getAthenaClient`, `getCouchbaseClient`, `getClickhouseClient`, `getGoogleDriveClient`, `getOnedriveClient`, `getSplunkClient`, `getKDBAIClient`, `getDatabricksSQLClient`, `getAzureSynapseClient`, `getEventHubsClient`, `getSharepointClient`, `getScyllaDBClient`, `getAwsOpenSearchClient`, `getWeaviateClient`, `getFirestoreClient`, `getFirebaseDBClient`, `getOpenAIClient`, `getXAIClient`, `getAzureOpenAIClient`, `getAwsBedrockClient`, `getCohereClient`, `getTogetherAIClient`, `getFireworksAIClient`, `getHuggingFaceClient`, `getAnthropicClient`, `getGeminiClient` and `getMcpClient`, each taking `connName`.

Raises `ValueError` when the named connection does not exist or has the wrong type, for example `s3` connection details for 'x' not found or the connection type is incorrect.

4.9 Connectors

The signatures below list every connector method on the session object. session object. Reads return the result in `returnFormat`; most connectors register the result as `targetDuckdbTable` when given. Writes return `None`, a status value or a count depending on the system, and raise the vendor's errors. Many connectors cache a client per connection; `closeConnection(connName)` releases it and `session.close()` releases all of them.

Object storage

PYTHON

```
session.readS3File(
    connName, bucketName=None, filePath="", fileType="parquet", s3FullUrl=None,
    targetDuckdbTable=None, readOptions=None, returnFormat="duckdb"
)
session.writeS3File(
    connName, bucketName=None, filePath="", fileType="parquet", s3FullUrl=None,
    sourceTable=None, sourceQuery=None, writeOptions=None
)
session.writeS3(*args, **kwargs)

session.readGcsFile(
    connName, bucketName=None, filePath="", fileType="parquet", gcsFullUrl=None,
    targetDuckdbTable=None, readOptions=None, returnFormat="duckdb"
)
session.writeGcs(
    connName, bucketName=None, filePath="", fileType="parquet", gcsFullUrl=None,
    data=None, sourceTable=None, sourceQuery=None, writeOptions=None
)

session.readAzureBlob(
    connName, containerName, blobName, fileType="parquet", azureFullUrl=None,
    targetDuckdbTable=None, readOptions=None, returnFormat="duckdb"
)
session.writeAzureBlob(
    connName, containerName, blobName, fileType="parquet", azureFullUrl=None,
    data=None, sourceTable=None, sourceQuery=None, writeOptions=None
)
```

SQL databases and warehouses

PYTHON

```

session.getSqlEngine(connName)
session.executeJdbc(
    connName, sqlQuery, params=None, targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeJdbc(
    connName, targetTable, data=None, sourceTable=None, sourceQuery=None,
    mode="append"
)

session.getSnowflakeClient(conn_name, client_prefetch_threads=4)
session.readSnowflake(
    conn_name, sql_query, target_duckdb_table, warehouse=None,
    force_microsecond_precision=False, client_prefetch_threads=4,
    returnFormat="duckdb"
)

session.readBigQuery(
    connName, query, useLegacySql=False, targetDuckdbTable=None,
    returnFormat="duckdb", maxQueueSize=2, maxStreamCount=None
)
session.readBigQueryTable(
    connName, projectId, datasetId, tableId, selectedFields=None, rowFilter=None,
    targetDuckdbTable=None, returnFormat="duckdb", maxQueueSize=2,
    maxStreamCount=None
)
session.writeBigQuery(
    connName, tableName, data, writeDisposition="WRITE_APPEND", **kwargs
)

session.readAthena(
    connName, region, query, targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeAthena(connName, region, sqlQuery)

session.executeDatabricks(
    connName, sourceQuery, targetDuckdbTable=None, returnFormat="duckdb",
    fetchBatchSize=10000
)
session.writeDatabricks(connName, sqlQuery)

session.readAzureSynapse(
    connName, query, targetDuckdbTable=None, returnFormat="duckdb",
    fetchBatchSize=10000
)

session.executeTeradata(
    connName, sourceQuery, targetDuckdbTable=None, returnFormat="duckdb",
    fetchBatchSize=10000
)
session.writeTeradata(connName, tableName, data, **kwargs)

session.getExasolClient(connName)
session.readExasol(
    connName, sqlQuery, targetDuckdbTable=None, returnFormat="duckdb",
    fetchBatchSize=10000
)
session.writeExasol(connName, tableName, data, **kwargs)

```

```

session.executeFirebolt(
    connName, sqlQuery, parameters=None, targetDuckdbTable=None,
    returnFormat="duckdb", fetchBatchSize=10000
)
session.getFireboltSystemInfo(
    connName, infoType="tables", targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeFirebolt(connName, sqlQuery)

session.executeClickHouse(
    connName, sourceQuery, parameters=None, targetDuckdbTable=None,
    returnFormat="duckdb"
)
session.getSystemInfo(
    connName, table="tables", targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeClickhouse(connName, tableName, data, database=None, **kwargs)

session.executeSingleStore(
    connName, sourceQuery, targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeSingleStore(connName, tableName, data, **kwargs)

session.executeMotherDuck(
    connName, query=None, tableName=None, limit=None, alias="motherduck",
    targetDuckdbTable=None, returnFormat="duckdb"
)
session.readMotherDuck(
    connName, query=None, tableName=None, limit=None, alias="motherduck",
    targetDuckdbTable=None, returnFormat="duckdb"
)

```

NoSQL, search and graph

PYTHON

```

session.getMongoClient(connName)
session.readMongo(
    connName, databaseName, collectionName, mongoQuery=None, projection=None,
    targetDuckdbTable=None, returnFormat="duckdb", **kwargs
)
session.writeMongodb(connName, databaseName, collectionName, data, **kwargs)

session.readDynamoDB(
    connName, tableName, filters=None, targetDuckdbTable=None,
    returnFormat="duckdb", parallelSegments=4
)
session.queryDynamoDB(
    connName, tableName, keyConditionExpression, expressionAttributeValues,
    expressionAttributeNames=None, filterExpression=None, indexName=None,
    targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeDynamoDb(connName, tableName, data)

session.executeCosmos(
    connName, sourceQuery, enableCrossPartition=True, targetDuckdbTable=None,
    returnFormat="duckdb"
)
session.writeCosmosDb(connName, data, **kwargs)

session.executeCouchbase(

```

```

        connName, sourceQuery, parameters=None, targetDuckdbTable=None,
        returnFormat="duckdb"
    )
    session.writeCouchbase(connName, data, idField, **kwargs)

    session.getCassandraSession(connName)
    session.readCassandra(connName, cqlQuery, targetDuckdbTable=None, returnFormat="duckdb")

    session.executeScyllaDB(
        connName, keyspace, sourceQuery, targetDuckdbTable=None, returnFormat="duckdb"
    )

    session.readFirestore(
        connName, collection, filters=None, orderBy=None, limit=None,
        targetDuckdbTable=None, returnFormat="duckdb"
    )
    session.writeFirestore(connName, collection, data, idField=None)

    session.readRealtimeDatabase(
        connName, path, orderBy=None, limitToFirst=None, limitToLast=None,
        targetDuckdbTable=None, returnFormat="duckdb"
    )

    session.executeElasticsearch(
        connName, indexName, query=None, size=10000, targetDuckdbTable=None,
        returnFormat="duckdb"
    )
    session.executeESQL(connName, esqlQuery, targetDuckdbTable=None, returnFormat="duckdb")
    session.writeElasticsearch(connName, indexName, data, idField=None, **kwargs)

    session.searchOpenSearch(
        connName, index, query=None, size=1000, scroll=None, targetDuckdbTable=None,
        returnFormat="duckdb"
    )
    session.vectorSearch(
        connName, index, vector, vectorField, k=10, size=100, filter=None,
        targetDuckdbTable=None, returnFormat="duckdb"
    )
    session.getIndexMapping(connName, index, targetDuckdbTable=None, returnFormat="duckdb")
    session.writeAwsOpenSearch(connName, index, data, idField=None, **kwargs)

    session.executeSplunk(
        connName, searchQuery, earliestTime=None, latestTime=None, maxCount=0,
        targetDuckdbTable=None, returnFormat="duckdb"
    )

    session.streamGraphData(
        connName, graphName, nodeLabels=None, relationshipTypes=None,
        nodeProperties=None, relationshipProperties=None, targetDuckdbTable=None,
        returnFormat="duckdb"
    )
    session.executeCypherQuery(
        connName, cypherQuery, parameters=None, targetDuckdbTable=None,
        returnFormat="duckdb"
    )
    session.getGraphSchema(connName, targetDuckdbTable=None, returnFormat="duckdb")
    session.getNodeLabels(connName, targetDuckdbTable=None, returnFormat="duckdb")
    session.getRelationshipTypes(connName, targetDuckdbTable=None, returnFormat="duckdb")
    session.writeNeo4j(connName, label, data, idField, properties=None, **kwargs)

    session.executeInfluxQuery(
        connName, query, database=None, targetDuckdbTable=None, returnFormat="duckdb"
    )

```

```

)
session.writeToInflux(
    connName, dataframe, measurement, database=None, tagColumns=None,
    timeColumn="_time"
)

```

Streaming and lake formats

PYTHON

```

session.readKafka(
    connName, topic, maxMessages=1000, autoOffsetReset="earliest",
    consumerTimeout=10000, targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeKafka(connName, topic, data, keyField=None, **kwargs)

session.readEventHub(
    connName, consumerGroup="$Default", maxEvents=1000, startingPosition="-1",
    targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeEventHub(connName, data, partitionKey=None, **kwargs)

session.readKinesis(
    connName, streamName=None, shardIteratorType="LATEST", limit=1000,
    maxShards=None, targetDuckdbTable=None, returnFormat="duckdb"
)
session.readKinesisStream(
    connName, streamName=None, shardIteratorType="LATEST", limit=1000,
    maxShards=None, targetDuckdbTable=None, returnFormat="duckdb"
)

session.readHudi(
    connName, tablePath, filters=None, columns=None, asOfTimestamp=None,
    targetDuckdbTable=None, returnFormat="duckdb"
)
session.readHudiIncremental(
    connName, tablePath, startCommitTime, endCommitTime=None, columns=None,
    targetDuckdbTable=None, returnFormat="duckdb"
)

```

SaaS, files and APIs

PYTHON

```

session.executeREST(
    connName, endpoint=None, method=None, headers=None, params=None, body=None,
    resultPath=None, targetDuckdbTable=None, returnFormat="duckdb", timeout=60,
    maxPages=None, nextPath=None, pageParam=None, pageStart=1, maxRecords=None
)

session.getSalesforceConnection(connName)
session.readSalesforce(
    connName, soqlQuery, targetDuckdbTable=None, returnFormat="duckdb",
    useBulkApi=True, batchSize=10000, pkChunking=False, chunkSize=100000
)

session.readHubSpot(
    connName, objectType, properties=None, limit=100, after=None, filters=None,
    searchTerm=None, maxPages=None, rateLimit=10, targetDuckdbTable=None,
    returnFormat="duckdb"
)

```

```

session.readServiceNow(
    connName, table, query=None, limit=1000, displayValue=True, options=None,
    targetDuckdbTable=None, returnFormat="duckdb"
)
session.executeServiceNow(
    connName, sourceQuery, limit=1000, options=None, targetDuckdbTable=None,
    returnFormat="duckdb"
)

session.readZohoCRM(
    connName, module, perPage=200, maxPages=None, targetDuckdbTable=None,
    returnFormat="duckdb"
)
session.executeZohoCRM(connName, module, targetDuckdbTable=None, returnFormat="duckdb")

session.readShopify(
    connName, resource, limit=None, sinceId=None, fields=None,
    targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeShopify(connName, resource, data, **kwargs)

session.readM365BC(
    connName, entity, filter=None, select=None, top=None, targetDuckdbTable=None,
    returnFormat="duckdb"
)
session.writeM365BC(connName, entity, data, **kwargs)

session.readGoogleDrive(
    connName, fileId, fileType=None, targetDuckdbTable=None, returnFormat="duckdb",
    **kwargs
)
session.listFiles(
    connName, folderId=None, query=None, targetDuckdbTable=None,
    returnFormat="duckdb"
)
session.writeGoogleDriveFile(
    connName, fileName, data, folderId=None, fileType="csv", **writeOptions
)

session.readOneDriveFile(
    connName, driveId, filePath, fileType="csv", targetDuckdbTable=None,
    returnFormat="duckdb", **readOptions
)
session.listOneDriveFiles(
    connName, driveId, folderPath=None, targetDuckdbTable=None,
    returnFormat="duckdb"
)
session.writeOneDriveFile(
    connName, driveId, filePath, data, fileType="csv", **writeOptions
)

session.readSharePointList(
    connName, siteUrl, listName, filter=None, select=None, top=None,
    targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeSharePointList(connName, siteUrl, listName, data, idField=None)

session.readSFTPFile(
    connName, filePath, fileType="csv", targetDuckdbTable=None,
    returnFormat="duckdb", **readOptions
)

```

```
session.listSFTPFiles(
    connName, directory=".", targetDuckdbTable=None, returnFormat="duckdb"
)
session.writeSFTPFile(connName, filePath, data, fileType="csv", **writeOptions)
```

Vector stores

PYTHON

```
session.queryVectors(
    connName, vector, topK=10, includeMetadata=True, includeValues=False,
    namespace=None, filter=None, targetDuckdbTable=None, returnFormat="duckdb"
)
session.upsertVectors(connName, vectors, namespace=None)
session.deleteVectors(connName, ids, namespace=None)
session.getIndexStats(connName, targetDuckdbTable=None, returnFormat="duckdb")
session.writePinecone(connName, data, namespace=None, **kwargs)

session.queryWeaviate(
    connName, className, properties=None, nearVector=None, nearText=None,
    where=None, limit=100, targetDuckdbTable=None, returnFormat="duckdb"
)
session.hybridSearch(
    connName, className, query, alpha=0.5, properties=None, where=None, limit=100,
    targetDuckdbTable=None, returnFormat="duckdb"
)
session.getSchema(connName, targetDuckdbTable=None, returnFormat="duckdb")
session.writeWeaviate(connName, className, data, batchSize=100, **kwargs)

session.readKDBAI(
    connName, tableName, filter=None, limit=None, targetDuckdbTable=None,
    returnFormat="duckdb"
)
session.searchKDBAI(
    connName, tableName, vectors, nResults=10, filter=None, targetDuckdbTable=None,
    returnFormat="duckdb"
)
session.writeKdbai(connName, tableName, data, **kwargs)
```

4.10 LLM providers

Groundzero connections exist for Anthropic, Amazon Bedrock, Azure OpenAI, Cohere, Fireworks AI, Google Gemini, Hugging Face, OpenAI, Together AI and xAI. The generic methods below pick the provider from the connection's type, so code does not change when you switch providers.

complete

PYTHON

```
def 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, llmOptions=None, targetDuckdbTable=None, returnFormat="duckdb"
): ...
```

Adds an LLM completion to every row of a DataFrame or relation.

Parameter	Type	Default	Description
<code>connName</code>	str	required	LLM connection
<code>df</code>	DataFrame or relation	required	Input rows
<code>model</code>	str	None	Model name (optional)
<code>promptColumn</code>	str	"prompt"	Column holding the prompt
<code>promptTemplate</code>	str	None	Template using <code>{column}</code> fields instead of a prompt column
<code>systemPrompt</code> , <code>systemPromptColumn</code>	str	None	Fixed or per-row system prompt
<code>outputColumn</code>	str	"llm_response"	Name of the response column
<code>temperature</code> , <code>maxTokens</code>	number	0.0, None	Sampling controls
<code>responseFormat</code>	str	None	Provider response format hint
<code>continueOnError</code>	bool	False	Record row errors instead of raising
<code>timeout</code>	int	120	Seconds per call
<code>llmOptions</code>	dict	None	Provider-specific options

Besides `outputColumn` the result has `<outputColumn>_status`, `_error`, `_provider`, `_model`, `_prompt_tokens`, `_completion_tokens`, `_total_tokens` and `_latency_ms` columns.

Raises `ValueError` for an unknown connection (`Connection 'x' was not found.`), an unsupported connection type or missing input columns, and the provider's error for a failed call unless `continueOnError=True`.

PYTHON

```
import pandas as pd

tickets = pd.DataFrame({
    "ticket_id": [1, 2],
    "text": ["App crashes on login", "Love the new dashboard"],
})

scored = session.complete(
    connName="llm_openai",
    df=tickets,
    promptTemplate="Reply with one word, positive or negative: {text}",
    outputColumn="sentiment",
    continueOnError=True,
    returnFormat="pandas",
)
```

Fine-tuning and model management

These generic methods route to the provider-specific method for the connection type (for example `fineTuneOpenAI`):

PYTHON

```

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

```

If the provider has no such operation the call raises `NotImplementedError: <operation> is not available for connection type '<type>'`.

Provider-specific methods exist for each operation. The suffix is the provider name: `OpenAI`, `AzureOpenAI`, `AwsBedrock`, `Cohere`, `TogetherAI`, `HuggingFace`, `FireworksAI`, `Gemini`, `Anthropic` or `XAI`.

Operation	Methods (with provider suffix)
Fine-tune	<code>fineTune<P></code> (not XAI)
Distill	<code>distillOpenAI</code> , <code>distillAzureOpenAI</code> , <code>distillAwsBedrock</code>
Jobs	<code>listFineTuneJobs<P></code> , <code>getFineTuneJob<P></code> , <code>cancelFineTuneJob<P></code> (Bedrock: <code>stopFineTuneJobAwsBedrock</code>)
Models	<code>listModels<P></code> (not Azure OpenAI)
Batch	<code>createBatchJob<P></code> , <code>listBatchJobs<P></code> , <code>getBatchJob<P></code> , <code>cancelBatchJob<P></code> , <code>getBatchResults<P></code>
Deployments	<code>createDeployment<P></code> , <code>listDeployments<P></code> , <code>getDeployment<P></code> , <code>deleteDeployment<P></code>

Not every provider supports every operation: a method that does not apply to a provider fails with an explicit error (for example OpenAI has no deployment lifecycle). Fine-tuning methods take a pandas DataFrame with prompt and response columns and return a DataFrame describing the job:

PYTHON

```

job = session.fineTuneOpenAI(
    connName="llm_openai",
    df=training_df,
    model="gpt-4o-mini-2024-07-18",
    promptColumn="prompt",
    responseColumn="response",
)

```

4.11 MCP servers

MCP connections let your code call tools, read resources and fetch prompts from a Model Context Protocol server. Every method returns the result as a relation (or the requested `returnFormat`).

PYTHON

```

session.listMcpTools(connName, targetDuckdbTable=None, returnFormat="duckdb")
session.callMcpTool(
    connName, toolName, arguments=None, targetDuckdbTable=None,
    returnFormat="duckdb"
)
session.listMcpResources(connName, targetDuckdbTable=None, returnFormat="duckdb")
session.readMcpResource(
    connName, uri, targetDuckdbTable=None, returnFormat="duckdb"
)
session.listMcpPrompts(connName, targetDuckdbTable=None, returnFormat="duckdb")
session.getMcpPrompt(
    connName, promptName, arguments=None, targetDuckdbTable=None,
    returnFormat="duckdb"
)
session.getMcpLlmConfig(
    connName, provider="openai", serverName=None, requireApproval="always",
    includeSecrets=False, targetDuckdbTable=None, returnFormat="duckdb"
)

```

`completeWithMcpTool` calls an MCP tool for each row and then asks an LLM to complete a prompt that includes the tool result:

PYTHON

```

session.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, targetDuckdbTable=None, returnFormat="duckdb"
)

```

MCP CONNECTION SETUP

Create the connection of type MCP Server with transport `streamable_http`, a server URL that includes `http://` or `https://`, an authentication type (`bearer` with a token, or `none` for public servers), optional headers and allowed tools, and a timeout. A URL without a scheme fails before tool discovery.

4.12 PDF extraction

PYTHON

```

session.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
)

```

Alias: `extractPDF` . Parses one document from S3 and writes the results (CSV, JSON, XLSX or Markdown) under `outputPrefix` .

Parameter	Type	Default	Description
<code>s3ConnName</code>	str	required*	S3 connection holding the input (<code>connName</code> is an alias)
<code>inputUri</code>	str	required*	Document key or <code>s3://</code> URI (<code>inputFile</code> is an alias)
<code>outputPrefix</code>	str	generated	Destination prefix
<code>outputFormat</code> , <code>outputFormats</code>	str, list	"csv"	One or several output formats
<code>hybrid</code>	bool	False	Hybrid extraction; pairs with the <code>llm...</code> options
<code>ocrMode</code>	str	"auto"	OCR policy
<code>ocrStack</code>	str	"ppocrv6"	OCR stack
<code>strict</code>	bool	False	Strict extraction mode
<code>password</code>	str	None	Password for encrypted PDFs
<code>maxPages</code> , <code>maxFileBytes</code>	int	None	Safety limits
<code>returnAgentPages</code>	bool	False	Produce page JSON or Markdown for Agents
<code>pageRange</code>	tuple	None	(first, last) page; requires <code>returnAgentPages=True</code>

*One of `inputUri` or `inputFile` is required, and an S3 connection name.

Returns a dictionary that includes `result_uri` . **Raises** `ValueError` (`s3ConnName` is required , `inputUri` or `inputFile` is required , `pageRange` requires `returnAgentPages=True`) and `PdfLimitError` when a source is larger than the configured limit.

PYTHON

```

session.readPDF(
    s3ConnName, inputUri=None, outputPrefix=None, outputFormat="csv",
    outputFormats=None, targetDuckdbTable=None, readOptions=None,
    returnFormat="duckdb", outputS3ConnName=None, **options
)
session.readPDFResult(
    resultUri, outputFormat="csv", s3ConnName=None, targetDuckdbTable=None,
    readOptions=None, returnFormat="duckdb"
)
session.parsePDFCorpus(
    s3ConnName, sourcePrefix=None, inputUris=None, outputPrefix=None,
    outputS3ConnName=None, pageRangeSize=64, partitions=None, ocrMode="auto",
    language="eng", strict=False, password=None, maxFileBytes=None,
    maxDocuments=10000
)

```

`readPDF` parses a document and returns its table in one call (`pdfParser` , `readPDFExtraction` are aliases); `readPDFResult` (alias `readPDFExtractionResult`) loads an already parsed `csv` or `json` result and raises `ValueError` for other formats. `parsePDFCorpus` (alias `processPDFCorpus`) parses every PDF under `sourcePrefix` , or the listed `inputUris` , with restartable, content addressed output. `partitions` sets the number of I/O workers (1 to 8); OCR still processes one document model at a time.

4.13 Groundzero Agents

PYTHON

```
def listAgents(connName) → List[dict]: ...

def invokeAgent(
    connName, agentId, message, sessionId=None, *, requestId=None,
    expectedAgentVersion=None, disableTools=False, pdfSourceConnName=None,
    pdfSourceObject=None, pdfSinkConnName=None, pdfSinkPrefix=None
) → dict: ...

def invokeAgentPDFCorpus(
    connName, agentId, message, pdfSourceConnName, pdfSinkPrefix,
    pdfSourcePrefix=None, pdfSourceObjects=None, pdfSinkConnName=None,
    pageRangeSize=64, partitions=None, summaryWorkers=4,
    summaryInputChars=100000, maxDocuments=10000, maxAgentCalls=50000,
    ocrMode="auto", language="eng", strict=False, password=None,
    maxFileBytes=None
) → dict: ...
```

Method	Purpose
<code>listAgents</code>	Agents visible to the connection's service account
<code>invokeAgent</code>	Run an Agent in the background and wait for the final result (polls up to 15 minutes)
<code>invokeAgentPDFCorpus</code>	Extract JSON or cited facts from every complete PDF

Parameters worth noting:

Parameter	Description
<code>agentId</code>	Agent identifier (a UUID)
<code>message</code>	Instruction for the Agent; must be a non-empty string
<code>expectedAgentVersion</code>	Positive integer; the call fails if the approved version differs
<code>disableTools</code>	<code>True</code> runs without tools; it cannot be combined with PDF arguments
<code>pdfSourcePrefix</code> / <code>pdfSourceObjects</code>	Supply exactly one
<code>summaryWorkers</code>	Concurrent document workflows, 1 to 32
<code>maxDocuments</code> , <code>maxAgentCalls</code>	Bounds for one run, defaults 10,000 and 50,000

Raises `ValueError` for invalid arguments (`agentId` is required, `message` is required, `expectedAgentVersion` must be a positive integer), `RuntimeError` when an Agent run returns an invalid run identifier or status, and `TimeoutError` when polling exceeds 15 minutes. A timeout does not cancel the run: inspect it before starting another.

4.14 Data quality and record linkage

PYTHON

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

Parameter	Description
<code>data</code>	DuckDB relation, Arrow table or reader, pandas or Polars DataFrame
<code>tableName</code> , <code>query</code>	A session table, or SQL, to check without copying
<code>connectionName</code>	Groundzero LLM connection used for the narrative
<code>config</code>	A <code>qualipilot.QualipilotConfig</code>
<code>includeLlm</code>	<code>False</code> runs only deterministic checks
<code>references</code>	Optional reference data for referential checks
<code>reportFormat</code>	<code>json</code> , <code>html</code> or <code>markdown</code>

The run methods return a `Qualipilot QualityReport` . Checks reuse the session's DuckDB connection, and only a compact aggregate of the deterministic results is sent to the LLM, not row samples.

Local record linkage and consolidation collect inputs into Polars frames, so each requires `maxRows` :

PYTHON

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

`config` is a `qualipilot.linking.LinkConfig` (mode `dedupe` or `link`) and `consolidation` a `ConsolidationConfig` . **Raises** `TypeError` if `maxRows` is not an integer, `ValueError` if it is not positive, if an input has more than `maxRows` rows (input has more than `maxRows=N` rows), if `link` mode has no `rightData` , if `dedupe` mode has `rightData` , or if deduplication is not given a `dedupe` configuration. `maxRows` bounds rows, not width or candidate pairs: select only the columns you need and keep blocking rules tight.

4.15 Talend migration helpers

Import from `GZ.TalendCompilerUtils` . These helpers load an exported Talend project into a DuckDB manifest table, parse it and compile it into Groundzero job code. They take the session as their first argument.

Function	Purpose
<code>loadTalendProjectFromS3ToDuckDB(duck, connName, prefix, ...)</code>	Load project files from S3 into a manifest table
<code>loadTalendProjectFilesToDuckDB(duck, files, ...)</code>	Load files you already hold in memory
<code>parseTalendManifestToIR(duck, ...)</code>	Parse the manifest into an intermediate representation
<code>compileTalendManifestToGroundZeroJobs(duck, ...)</code>	Parse and compile to job code
<code>compileTalendIRToGroundZeroJobs(ir, ...)</code>	Compile an existing representation
<code>writeTalendCompileResultToDuckDB(duck, result, ...)</code>	Persist compiled jobs to a table
<code>renderTalendCompileReport(result)</code>	Render a compact report

5 Installed packages

Packages without a version install at their latest release when the runtime is built. To read the exact version in your session, run `pip show <name>` in a notebook.

5.1 Python and system

Component	Version or detail
Python	3.12
<code>numpy</code>	2.3.5
<code>pandas</code>	2.2.0 or newer, below 3
<code>pyarrow</code> , <code>SQLAlchemy</code> , <code>boto3</code> , <code>urllib3</code> , <code>matplotlib</code> , <code>pymysql</code>	unpinned
<code>jupyterlab</code> , <code>ipython</code> , <code>ipykernel</code>	unpinned
<code>mlflow</code>	2.20.4
<code>torch</code> , <code>torchvision</code> (CPU)	2.14.0, 0.29.0
<code>scipy</code>	1.17.1
<code>scikit-learn</code>	1.5.2
System tools	AWS CLI v2, <code>unixodbc</code> , PostgreSQL and MySQL client libraries, time-zone data

Time zone data is included, so time-zone-aware timestamps and the `UTC` zone work correctly in PyArrow.

5.2 Connector and data libraries

Package	Version	Purpose
<code>duckdb</code>	1.5.5	Query engine
<code>qualipilot[gz,linking]</code>	3.5.0	Data quality and record linkage
<code>pymongoarrow</code> , <code>pymongo</code>	unpinned	MongoDB
<code>google-cloud-bigquery-storage</code> , <code>google-cloud-bigquery</code>	unpinned	BigQuery
<code>pyodbc</code>	unpinned	ODBC databases
<code>psycopg2-binary</code> , <code>cx-Oracle</code>	unpinned	PostgreSQL, Oracle
<code>teradatasql</code>	unpinned	Teradata
<code>snowflake-connector-python</code> , <code>snowflake-sqlalchemy</code>	unpinned	Snowflake
<code>s3fs</code>	unpinned	S3 file system

Package	Version	Purpose
<code>simple-salesforce</code>	unpinned	Salesforce
<code>azure-storage-blob</code>	unpinned	Azure Blob Storage
<code>google-cloud-storage</code> , <code>gcsfs</code>	unpinned	Google Cloud Storage
<code>paramiko</code>	unpinned	SFTP
<code>cassandra-driver</code>	unpinned	Cassandra and ScyllaDB
<code>redshift-connector</code>	unpinned	Amazon Redshift
<code>requests</code> , <code>httpx</code> ($\geq 0.27, < 1$), <code>httpcore</code> ($\geq 1, < 2$)		HTTP clients
<code>azure-cosmos</code>	unpinned	Cosmos DB
<code>elasticsearch</code> , <code>opensearch-py</code>	unpinned	Elasticsearch, OpenSearch
<code>pyexasol</code>	unpinned	Exasol
<code>neo4j</code> , <code>graphdatascience</code>	unpinned	Neo4j
<code>pinecone</code>	unpinned	Pinecone
<code>weaviate-client</code>	unpinned	Weaviate
<code>kdbai-client</code>	$\geq 1.3.0$	KDB.AI
<code>firebolt-sdk</code>	unpinned	Firebolt
<code>influxdb3-python</code>	unpinned	InfluxDB 3
<code>ms-dynamics-business-central-sdk</code>	unpinned	Business Central
<code>shopifyapi</code> , <code>marketorestorepython</code>	unpinned	Shopify, Marketo
<code>azure-synapse</code>	unpinned	Azure Synapse
<code>google-api-python-client</code> , <code>google-auth</code> , <code>google-auth-http2</code> , <code>google-auth-oauthlib</code>	unpinned	Google Drive and authentication
<code>msal</code>	unpinned	Microsoft identity
<code>couchbase</code>	unpinned	Couchbase
<code>clickhouse-connect</code>	unpinned	ClickHouse
<code>databricks-sql-connector</code>	unpinned	Databricks SQL
<code>splunklib</code> , <code>splunk-sdk</code>	unpinned	Splunk
<code>firebase-admin</code>	unpinned	Firebase and Firestore
<code>adbc-driver-postgresql</code> , <code>adbc-driver-sqlite</code> , <code>adbc-driver-manager</code>	unpinned	Arrow database drivers
<code>pyiceberg[sql-postgres,s3fs,pyarrow]</code>	unpinned	Iceberg client
<code>hudi</code>	unpinned	Apache Hudi
<code>pyathena</code>	unpinned	Amazon Athena

Package	Version	Purpose
kafka-python	unpinned	Kafka
azure-eventhub	unpinned	Event Hubs
singlestoredb	unpinned	SingleStore
openpyxl	unpinned	Excel files

5.3 LLM and AI libraries

Package	Version	Purpose
openai	unpinned	OpenAI and Azure OpenAI
anthropic	unpinned	Anthropic
cohere	unpinned	Cohere
huggingface-hub	unpinned	Hugging Face
google-generativeai	unpinned	Gemini
mcp	>=1.29,<2	Model Context Protocol client

5.4 Document extraction

Package	Version	Purpose
docling[rapidocr]	2.116.0	Document conversion with OCR
docling-core	2.88.0	Docling core
docling-ibm-models	3.13.3	Docling models
docling-parse	7.8.1	Docling PDF parser
rapidocr	3.9.2	OCR
onnxruntime	1.28.0	Model inference
pdf2image	>=1.17	PDF rasterizing
pdfplumber	>=0.11	PDF text extraction
pillow	>=10.0	Images
pypdf	>=4.2	PDF utilities
pytesseract	>=0.3	OCR bridge

The layout, table-structure and OCR models come with the runtime, so PDF extraction works without internet access. The Groundzero helper packages (`gz` and others) are already importable.

5.5 DuckDB extensions

These extensions are installed. Connectors load the ones they need the first time you use them. For the others, such as `spatial`, run `session.query("LOAD spatial")`. Nothing is downloaded when your code runs.

Extension	Used for
<code>httpsfs</code>	Reading and writing S3, GCS and HTTP objects
<code>aws</code>	AWS credential handling
<code>iceberg</code>	Groundzero Lakehouse and Iceberg tables
<code>parquet</code> , <code>json</code>	File formats
<code>azure</code>	Azure Blob Storage
<code>postgres</code> , <code>mysql</code>	Attach and write PostgreSQL and MySQL databases
<code>spatial</code>	Geospatial types and functions
<code>motherduck</code>	MotherDuck

5.6 Packages added by the ML variants

Variant	Packages
scikit-learn	<code>opencv-python-headless</code> , <code>xgboost</code> , <code>lightgbm</code> , <code>catboost</code>
PyTorch	<code>torch 2.5.1</code> , <code>torchvision 0.20.1 (CUDA 12.1)</code> , <code>opencv-python-headless</code> , <code>xgboost</code> , <code>lightgbm</code> , <code>catboost</code> , <code>shap</code> , <code>lime</code> , <code>alumentations</code> , <code>huggingface_hub</code> , <code>transformers</code> , <code>simsimd</code> , <code>fastai</code>
TensorFlow	<code>tensorflow</code> , <code>opencv</code> , <code>xgboost</code> , <code>lightgbm</code> , <code>catboost</code> , <code>shap</code> , <code>lime</code> , <code>nltk</code> , <code>spacy</code> , <code>h5py</code>

6 Limits and troubleshooting

6.1 Limits and quotas

Limit	Value
Rows returned by a dataset or preview	100,000
Rows per page through the data API	up to 10,000
Runs per session at the same time	1
Wait for a dataset run	300 seconds
Idle session	released about 5 minutes after its last run
Output cache	up to 15 minutes
DuckDB memory	80 percent of the compute's memory, at least 512 MB
DuckDB threads	the compute's CPU count, at least 2
Spill location	<code>/tmp/duckdb-spill</code>
Agent run wait	15 minutes
Corpus run	10,000 documents, 50,000 Agent calls, 1 to 32 workers
REST connector request timeout	60 seconds by default
Log line length before splitting	200,000 characters

Compute (CPU and memory) comes from the compute sizes defined for your site. A job has no maximum duration. A schedule that has an end time stops the run at that time.

WHERE RESULTS ARE LIMITED

The 100,000 row limit applies to rows *returned* as output. Queries inside the run can process any amount of data that fits the compute's memory and local disk.

6.2 Things to know

- **DuckDB reads the lakehouse only.** Use the Spark runtime to write lakehouse tables.
- `returnFormat="arrow"` returns a reader in most places. `query()` and the file and database connectors return a `pyarrow.RecordBatchReader`; `readGzLakehouse` returns a `pyarrow.Table`. Call `.read_all()` on a reader.
- **One site per lakehouse connection per session.** To use a second site, start a new session.
- **Placeholders expand everywhere.** Upper-case names such as `{USR}` are replaced even in f-strings and comments.
- **pandas output turns numeric NaN into the text `NaN`** and datetimes into strings. Return a relation or Arrow table to keep types.
- **PDF OCR processes one document model at a time.** Use the Spark runtime for distributed OCR.

- **Identical dataset code is served from cache** for up to 15 minutes.
- **Some imports are blocked in jobs and datasets.** Modules such as `socket`, `urllib`, `http`, `shutil`, `pathlib`, `pickle`, `sqlite3` and `threading` cannot be imported. Use connections and the DuckDB engine instead.

6.3 Troubleshooting

Each row gives the message you see, the cause and the fix. Messages are shown exactly as the runtime prints them.

Symptom	Cause	Fix
Either <code>s3FullUrl</code> or both <code>bucketName</code> and <code>filePath</code> must be provided	A read or write was called without a location. The same form applies to <code>readGcsFile</code> and <code>readAzureBlob</code> .	Pass <code>s3FullUrl="s3:// ..."</code> , or both <code>bucketName</code> and <code>filePath</code> .
S3 connection details for 'x' not found or the connection type is incorrect.	The connection name does not exist in this site, is not an S3 connection, or was not attached to the run.	Check the spelling (names are case sensitive) and that the dataset or job has the connection selected.
S3 connection details for 'x' are incomplete	The S3 connection is missing a region, access key or secret.	Edit the connection in Groundzero.
GZLAKEHOUSE writes are not supported from DuckDB for connection 'x'. DuckDB lakehouse access is read-only.	<code>writeGzLakehouse</code> was called.	Switch the job to the Spark runtime, or write to S3 or a database from DuckDB.
Only read-only queries are allowed	<code>readGzLakehouse</code> accepts <code>SELECT</code> , <code>SHOW</code> , <code>DESCRIBE</code> and <code>EXPLAIN</code> only.	Remove <code>INSERT</code> , <code>UPDATE</code> , <code>DELETE</code> , <code>DDL</code> and similar statements.
Use fully qualified table names.	A lakehouse query referenced a table without its database.	Write <code>retail.orders</code> , not <code>orders</code> .
Either 'query' or 'tableName' must be provided	<code>readGzLakehouse</code> was called with neither.	Provide one.
databaseName is required when tableName is not qualified	You passed <code>tableName="orders"</code> and the connection has no default database.	Use <code>tableName="retail.orders"</code> .
GZLAKEHOUSE 'x' has <code>lakeHouseUrl</code> but no <code>siteName</code> for REST catalog setup.	The site for the connection cannot be worked out.	Pass <code>siteName="..."</code> or <code>siteNames={"x": "..."} to <code>duckSession</code>, or <code>siteName</code> to <code>readGzLakehouse</code>.</code>
GZLAKEHOUSE connection 'x' is already configured for site 'a'; create a new DuckDB session to use site 'b'.	One session can use one site per connection.	Create another <code>duckSession</code> for the second site.

Symptom	Cause	Fix
Authentication failed or You do not have permission (a <code>PermissionError</code>)	The lakehouse rejected the user.	Ask a site administrator for <code>Select</code> permission on the table or database.
Direct access to <code>gz-lakehouse-*</code> buckets is restricted; use the <code>GZLAKEHOUSE</code> connector	Your code tried to read Groundzero's lakehouse storage directly.	Read the table through <code>readGzLakehouse</code> .
Import of 'socket' is not allowed for security reasons (also <code>eval()</code> is not allowed for security reasons and Access to '/var/data.csv' outside sandbox-safe directories is not allowed)	The module, function or path is blocked in jobs and datasets.	Use connections for network access and <code>/tmp</code> for files.
<code>outputDF</code> must be <code>DuckDBPyRelation</code> , <code>pa.Table</code> , <code>pa.RecordBatch</code> , <code>pa.RecordBatchReader</code> , <code>DataFrame</code> , or <code>list</code> . Got X	The value in <code>outputDF</code> (or passed to <code>gz_output</code>) is not a supported type.	Convert with <code>.df()</code> , <code>.to_arrow_table()</code> or <code>list(...)</code> .
Page 3 is out of range. Total pages: 2 for page size 10.	A preview asked for a page past the end of the result.	Ask for a page within the total.
No executable code found.	The run had no code, or the job file was empty.	Save the code and run again.
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.
Connection not found: x	A dataset or job references a connection that no longer exists in the site.	Recreate the connection or select another one.
Invalid RTE	The runtime name was not recognised.	Choose the runtime again in the runtime selector.
Binder Error: Invalid named parameter "DELIMITER" for function <code>read_csv_auto</code>	A key in <code>readOptions</code> is not a DuckDB option name. The read is tried three times, so you see the last error.	Use the DuckDB name (<code>delim</code> or <code>sep</code> for the separator) and see the candidate list in the message.
Unsupported returnFormat. Use 'duckdb', 'pandas', 'arrow', or 'row_count'.	<code>readSnowflake</code> got another format. Other connectors accept <code>duckdb</code> , <code>pandas</code> and <code>arrow</code> .	Use one of the listed formats.
Out of Memory Error (<code>duckdb.OutOfMemoryException</code>)	The query needed more memory than the DuckDB limit and could not spill enough.	Filter and project earlier, avoid converting large relations to pandas, or choose a larger compute. Run <code>current_setting('memory_limit')</code> to see the limit.

Symptom	Cause	Fix
Could not parse query: ...	<code>readGzLakehouse</code> could not parse your SQL.	Check for dialect-specific syntax.
input has more than <code>maxRows=100,000</code> rows	A local record-linkage call received more rows than <code>maxRows</code> .	Filter the input, or raise <code>maxRows</code> knowingly.
<code>TimeoutError</code> from an Agent call	The wait for an Agent run exceeded 15 minutes. The run is not cancelled.	Look up the run identifier in the error message before you start another invocation.
Runtime is shutting down	A request reached a session that is closing after being idle.	Retry. A new session starts.

6.4 FAQ

Can DuckDB write to the Groundzero Lakehouse? No. DuckDB reads the lakehouse only. Write lakehouse tables with the Spark runtime, or write results to S3 or a database from DuckDB.

How do I choose between DuckDB and Spark? Use DuckDB when the data you process fits one machine and you want fast joins and a quick start. Use Spark for distributed processing and lakehouse writes. See Choose the DuckDB runtime.

Why did my dataset show old data? Dataset output is cached by the exact code text. Wait for the cache to expire or change the code. See Return output and previews.

How many rows can a dataset return? Up to 100,000. Aggregate or filter in SQL before returning.

Does state carry over between runs? Do not rely on it. A run may start in a new session.

Can I use `os.environ` and read credentials? No. In jobs and datasets, secret values in `os.environ` read as `****`. Use connections.

Which time zone does the runtime use? Time zone data is included, so named zones such as `UTC` and `Europe/Berlin` work with PyArrow and pandas.

How do I add a Python package? Choose an ML variant, ask an administrator for a custom runtime built from a `requirements.txt`, or `pip install` in a notebook. See Use extra packages.

How do I see why a job failed? Open the job's log view. The error shows the exception type, message and the line of your code that raised it. Printed output appears in the same view.

Are the method names the same as in the Spark runtime? Many connector methods have matching names, but the session object, return types and write support differ. Check the reference for each runtime.

Where do query results go when I use `returnFormat="pandas"`? Into memory as a DataFrame. For large results keep the relation lazy.

7 Examples gallery

The examples use fictional data for a retail company, Acme Retail. Connection names (`sales_s3` , `orders_pg` , `lakehouse_main` , ...) are placeholders for your own.

7.1 Example 1: Partitioned daily export

Aggregate all order files and write one Parquet folder per region, ready for downstream readers.

PYTHON

```
from GZ.DuckDBUtils import duckSession

session = duckSession()

session.readS3File(
    connName="sales_s3",
    s3FullUrl="s3://acme-retail-data/raw/orders/*.parquet",
    targetDuckdbTable="orders",
)

# The S3 call above configured access, so plain SQL can COPY to S3 too.
session.query("""
    COPY (
        SELECT region,
               CAST(ordered_at AS DATE) AS order_date,
               count(*)                AS orders,
               round(sum(amount), 2)    AS revenue
        FROM orders
        WHERE status <> 'returned'
        GROUP BY ALL
    )
    TO 's3://acme-retail-data/curated/revenue_by_region'
    (FORMAT PARQUET, PARTITION_BY (region), OVERWRITE_OR_IGNORE)
""")

check = session.query(
    "SELECT count(*) AS files FROM glob("
    "'s3://acme-retail-data/curated/revenue_by_region/*.parquet')",
    returnFormat="pandas",
)

print(check)
```

7.2 Example 2: Rolling averages with window functions

DuckDB's analytic SQL makes time-series features a one-liner. This computes a 7-day moving average of revenue and ranks days within each region.

PYTHON

```

from GZ.DuckDBUtils import duckSession

session = duckSession()
session.readS3File(
    "sales_s3", s3FullUrl="s3://acme-retail-data/raw/orders/*.parquet",
    targetDuckdbTable="orders",
)

outputDF = session.query("""
    WITH daily AS (
        SELECT region,
               CAST(ordered_at AS DATE) AS day,
               sum(amount) AS revenue
        FROM orders
        GROUP BY region, day
    )
    SELECT region, day, round(revenue, 2) AS revenue,
           round(avg(revenue) OVER (
               PARTITION BY region ORDER BY day
               ROWS BETWEEN 6 PRECEDING AND CURRENT ROW), 2) AS revenue_7d_avg,
           rank() OVER (PARTITION BY region ORDER BY revenue DESC) AS best_day_rank
    FROM daily
    ORDER BY region, day
""")

```

7.3 Example 3: Flatten JSON events

Semi-structured JSON is a first-class citizen. This reads newline-delimited event files and extracts nested fields without leaving SQL.

PYTHON

```

from GZ.DuckDBUtils import duckSession

session = duckSession()
session.readS3File(
    connName="sales_s3",
    bucketName="acme-retail-data",
    filePath="raw/events/events.json",
    fileType="json",
    targetDuckdbTable="events",
)

outputDF = session.query("""
    SELECT user_id,
           event,
           props.page AS page,
           CAST(props.duration_ms AS INTEGER) AS duration_ms,
           CAST(ts AS TIMESTAMP) AS ts
    FROM events
    WHERE event IN ('page_view', 'checkout')
    ORDER BY ts
""")

```

7.4 Example 4: Churn scoring with scikit-learn

Train a model on features computed in SQL, score every customer and write the scores back. Use the DuckDB scikit-learn (or PyTorch or TensorFlow) variant, or any runtime that includes scikit-learn.

PYTHON

```
from sklearn.linear_model import LogisticRegression

from GZ.DuckDBUtils import duckSession

session = duckSession()
session.readS3File(
    "sales_s3", s3FullUrl="s3://acme-retail-data/raw/orders/*.parquet",
    targetDuckdbTable="orders",
)

features = session.query("""
    SELECT customer_id,
           count(*)           AS n_orders,
           avg(amount)        AS avg_amount,
           max(amount)         AS max_amount,
           max(CASE WHEN status = 'returned' THEN 1 ELSE 0 END) AS had_return
    FROM orders
    GROUP BY customer_id
""", returnFormat="pandas")

x = features[["n_orders", "avg_amount", "max_amount"]]
y = features["had_return"]
model = LogisticRegression(max_iter=500).fit(x, y)

features["return_risk"] = model.predict_proba(x)[:, 1].round(4)
session.register("scores", features[["customer_id", "return_risk"]])

session.writeS3File(
    connName="sales_s3",
    s3FullUrl="s3://acme-retail-data/curated/return_risk.parquet",
    sourceTable="scores",
)

outputDF = session.query("SELECT * FROM scores ORDER BY return_risk DESC LIMIT 10")
```

7.5 Example 5: Deduplicate and validate before loading

A light quality gate that runs entirely in DuckDB: reject malformed rows, keep the latest record per order and report what was dropped.

PYTHON

```

from GZ.DuckDBUtils import duckSession

session = duckSession()
session.readS3File(
    "sales_s3", s3FullUrl="s3://acme-retail-data/raw/orders/*.parquet",
    targetDuckdbTable="raw_orders",
)

session.query("""
    CREATE TABLE clean_orders AS
    SELECT * EXCLUDE (rn)
    FROM (
        SELECT *, row_number() OVER (
            PARTITION BY order_id ORDER BY ordered_at DESC) AS rn
        FROM raw_orders
        WHERE amount IS NOT NULL AND amount ≥ 0 AND region IS NOT NULL
    )
    WHERE rn = 1
""")

stats = session.query("""
    SELECT (SELECT count(*) FROM raw_orders) AS raw_rows,
           (SELECT count(*) FROM clean_orders) AS clean_rows
""", returnFormat="pandas")
print(stats.to_dict("records")[0])

session.writeS3File(
    "sales_s3", s3FullUrl="s3://acme-retail-data/curated/clean_orders.parquet",
    sourceTable="clean_orders",
)

```

7.6 Example 6: Customer 360 across lakehouse, PostgreSQL and S3

One query joins three systems. The lakehouse read is governed for the calling user; the PostgreSQL and S3 reads stream into DuckDB.

PYTHON

```

from GZ.DuckDBUtils import duckSession

session = duckSession(siteName="acme")

orders = session.readGzLakehouse(
    connName="lakehouse_main",
    query="""
        SELECT customer_id, count(*) AS orders, sum(amount) AS revenue
        FROM retail.orders GROUP BY customer_id
    """,
)

profiles = session.executeJdbc(
    connName="orders_pg",
    sqlQuery="SELECT customer_id, segment, signup_date FROM public.customers",
)

tiers = session.readS3File(
    connName="sales_s3",
    bucketName="acme-retail-data",
    filePath="reference/loyalty_tiers.csv",
    fileType="csv",
)

session.register("orders", orders)
session.register("profiles", profiles)
session.register("tiers", tiers)

outputDF = session.query("""
    SELECT p.customer_id, p.segment, t.tier,
           o.orders, round(o.revenue, 2) AS revenue
    FROM profiles p
    LEFT JOIN orders o USING (customer_id)
    LEFT JOIN tiers t ON o.revenue BETWEEN t.min_revenue AND t.max_revenue
    ORDER BY revenue DESC NULLS LAST
""")

```

7.7 Example 7: Incremental load from Snowflake

Pull only the rows newer than the last load, append them to a Parquet history file set and report the row count.

PYTHON

```

from GZ.DuckDBUtils import duckSession

session = duckSession()

session.readS3File(
    connName="sales_s3",
    s3FullUrl="s3://acme-retail-data/history/orders/*.parquet",
    targetDuckdbTable="history",
)

last = session.query(
    "SELECT coalesce(max(order_ts), TIMESTAMP '1970-01-01') AS w FROM history",
    returnFormat="pandas",
)["w"].iloc[0]

rows = session.readSnowflake(
    conn_name="wh_snowflake",
    sql_query=f"""
        SELECT order_id, customer_id, amount, order_ts
        FROM ANALYTICS.PUBLIC.ORDERS
        WHERE order_ts > '{last:%Y-%m-%d %H:%M:%S}'
    """,
    target_duckdb_table="orders_new",
    returnFormat="row_count",
)

print(f"loaded {rows} new orders")

if rows:
    session.writeS3File(
        connName="sales_s3",
        s3FullUrl=(
            "s3://acme-retail-data/history/orders/"
            f"batch_{last:%Y%m%d%H%M%S}.parquet"
        ),
        sourceTable="orders_new",
    )

```

7.8 Example 8: Classify support tickets with an LLM

Read tickets from PostgreSQL, add a sentiment label from an LLM connection and save the result.

`continueOnError=True` keeps one bad row from failing the run.

PYTHON

```
from GZ.DuckDBUtils import duckSession

session = duckSession()

tickets = session.executeJdbc(
    connName="orders_pg",
    sqlQuery="SELECT ticket_id, body FROM support.tickets WHERE status = 'open'",
    returnFormat="pandas",
)

labelled = session.complete(
    connName="llm_openai",
    df=tickets,
    promptTemplate=(
        "Classify the sentiment of this support ticket as positive, "
        "neutral or negative. Reply with one word.\n\n{body}"
    ),
    outputColumn="sentiment",
    continueOnError=True,
    timeout=60,
)
session.register("labelled", labelled)

session.writeS3File(
    connName="sales_s3",
    s3FullUrl="s3://acme-retail-data/curated/ticket_sentiment.parquet",
    sourceQuery="""
        SELECT ticket_id, lower(trim(sentiment)) AS sentiment,
               sentiment_status, sentiment_total_tokens
        FROM labelled
    """,
)
```

7.9 Example 9: Ingest a REST API

Load paginated JSON from an API connection, keep the fields you need and write them as CSV.

PYTHON

```

from GZ.DuckDBUtils import duckSession

session = duckSession()

session.executeREST(
    connName="support_api",
    endpoint="/v2/tickets",
    params={"status": "open", "per_page": 100},
    resultPath="data.items",
    maxPages=20,
    maxRecords=2000,
    targetDuckdbTable="api_tickets",
)

session.writeS3File(
    connName="sales_s3",
    s3FullUrl="s3://acme-retail-data/raw/tickets/open.csv",
    fileType="csv",
    sourceQuery="SELECT id, subject, created_at FROM api_tickets",
    writeOptions={"header": True},
)

```

7.10 Example 10: Geospatial distance query

The `spatial` extension is installed. This finds the stores within 50 km of a point.

PYTHON

```

from GZ.DuckDBUtils import duckSession

session = duckSession()
session.query("LOAD spatial")

session.query("""
    CREATE TABLE stores AS
    SELECT * FROM (VALUES
        ('Berlin', 52.5200, 13.4050),
        ('Potsdam', 52.3906, 13.0645),
        ('Hamburg', 53.5511, 9.9937)
    ) AS t(store, lat, lon)
""")

outputDF = session.query("""
    SELECT store,
        round(ST_Distance_Sphere(
            ST_Point(lon, lat), ST_Point(13.4050, 52.5200)) / 1000, 1)
        AS km_from_berlin
    FROM stores
    WHERE ST_Distance_Sphere(
        ST_Point(lon, lat), ST_Point(13.4050, 52.5200)) ≤ 50000
    ORDER BY km_from_berlin
""")

```

7.11 Example 11: Extract policy fields from PDFs with an Agent

Parse a folder of PDFs and ask an approved Agent to fill your JSON schema for each one. The Agent's response schema, configured in Agent Builder, decides the output shape.

PYTHON

```
from GZ.DuckDBUtils import duckSession

session = duckSession()
try:
    result = session.invokeAgentPDFCorpus(
        connName="agents",
        agentId="00000000-0000-4000-8000-000000000001",
        message="Extract the policy number, holder name and effective date.",
        pdfSourceConnName="sales_s3",
        pdfSourcePrefix="incoming/policies/",
        pdfSinkConnName="sales_s3",
        pdfSinkPrefix="agent-results/policies",
        summaryWorkers=8,
    )
    print(result["status"], result["outputMode"])
    print(result["resultsUri"])
    print(result["manifestUri"])
finally:
    session.close()
```

7.12 Example 12: A two-step pipeline with executeJob

Put the heavy preparation in a sub-job and let the parent job orchestrate and notify.

PYTHON

```
from GZ.JobUtils import executeJob
from GZ.SendEmail import sendEmail

results = executeJob(["load_orders", "build_summary"])

sendEmail(
    subject="Retail pipeline finished",
    body=f"Completed sub-jobs: {' '.join(results)}",
    to_email="data-team@example.com",
    from_email="noreply@example.com",
)
```

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.

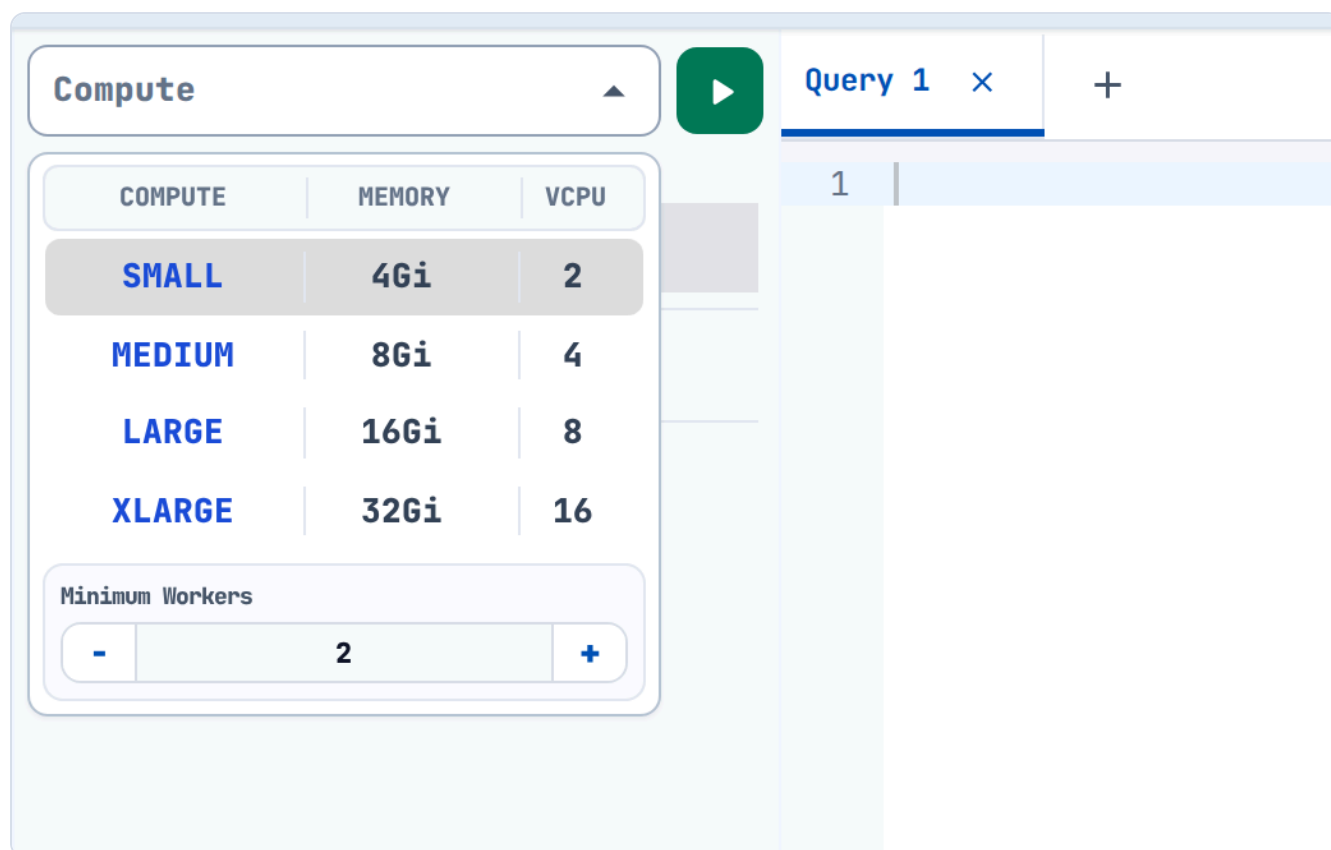
For Spark:

- 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.

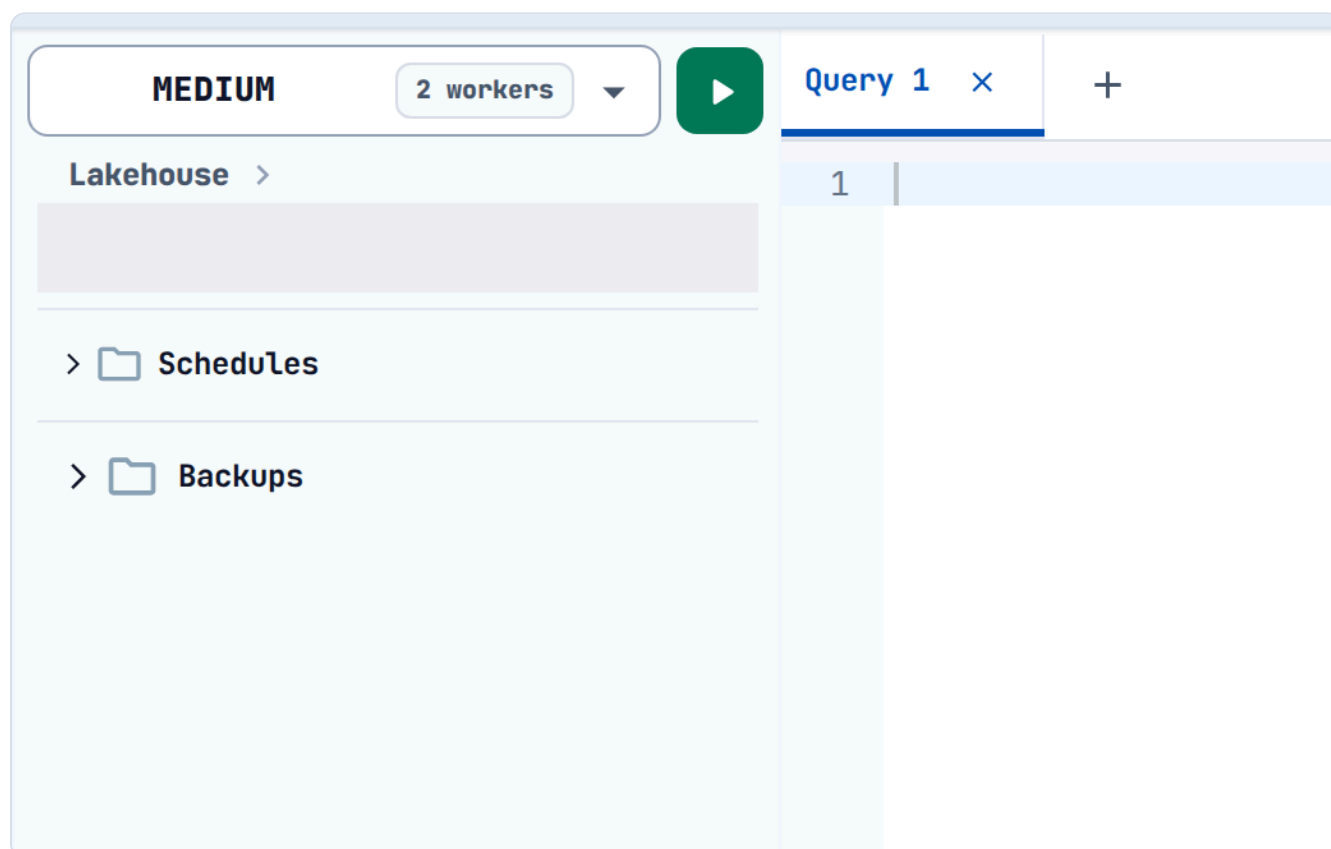


The screenshot shows the 'Compute' dropdown menu in the DuckDB Runtime interface. The menu is open, displaying a table of compute sizes. The 'SMALL' option is selected and highlighted. To the right of the menu is a green play button. The main panel shows 'Query 1' with a table containing one row with the value '1'.

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

Below the table is a 'Minimum Workers' section with a slider set to 2.

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



The screenshot shows the DuckDB Runtime interface with the 'MEDIUM' compute size selected. The '2 workers' dropdown is also visible. The main panel shows 'Query 1' with a table containing one row with the value '1'.

MEDIUM 2 workers

Lakehouse >

- > Schedules
- > Backups

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)

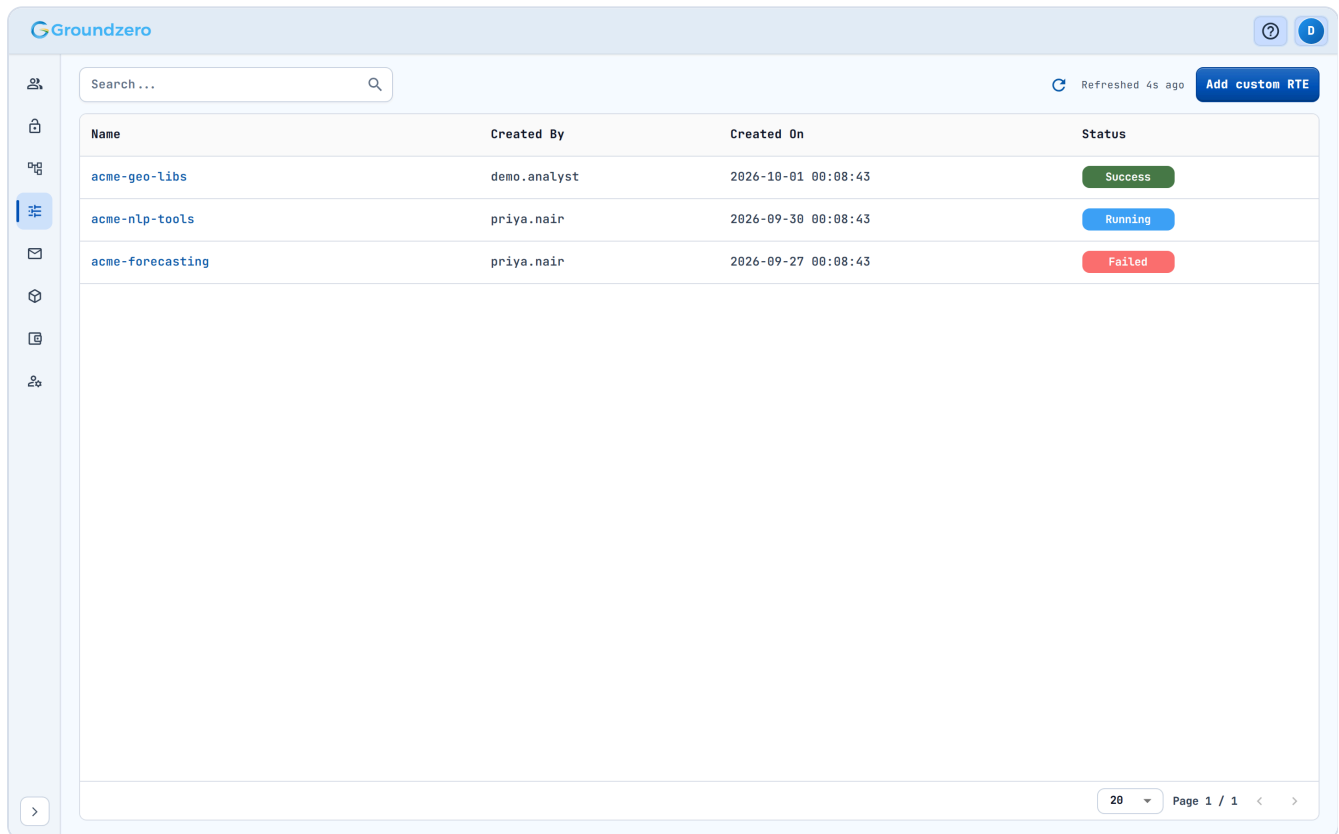
A.6 Custom runtimes

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 web interface. A modal dialog titled "Add Custom RTE" is open in the center. The dialog contains the following fields and controls:

- Base Image:** A dropdown menu with "spark" selected.
- Name:** A text input field containing "geo-libs".
- Preview:** The text "spark-geo-libs" is displayed below the name field.
- File Upload:** A button labeled "Choose File (.txt file only)" is present, with the filename "requirements.txt" shown below it.
- Actions:** "Cancel" and "Add" buttons are at the bottom right of the dialog.

The background interface shows a table of existing runtimes:

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: 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