

# About this course

This course is geared towards those who have some familiarity Dagster and want to learn more about ETL. You don't need to be an expert, but you should know your way around a Dagster project.

In this course, you'll learn how to orchestrate ETL pipelines. We will discuss common ETL situations and some of their pitfalls and how to effectively layer in Dagster to make managing these pipelines easier.

You'll load static files into a data warehouse using schedules, sensors, and partitions. You'll explore the nuances of extracting data from APIs, streamline your workflows with ETL frameworks (dlt), and replicate data across databases. Finally, you'll see how Dagster Components can help you build production-quality ETL solutions with just a few lines of code.

---

## Required experience

To successfully complete this course, you'll need:

- **Dagster familiarity** - You'll need to know the basics of Dagster to complete this course. **If you've never used Dagster before or want a refresher before getting started**, check out the [Dagster Essentials course](#).
- **Docker knowledge** - We will provide you with everything you need to run everything around Docker. But being able to navigate around the basics of Docker will be helpful.

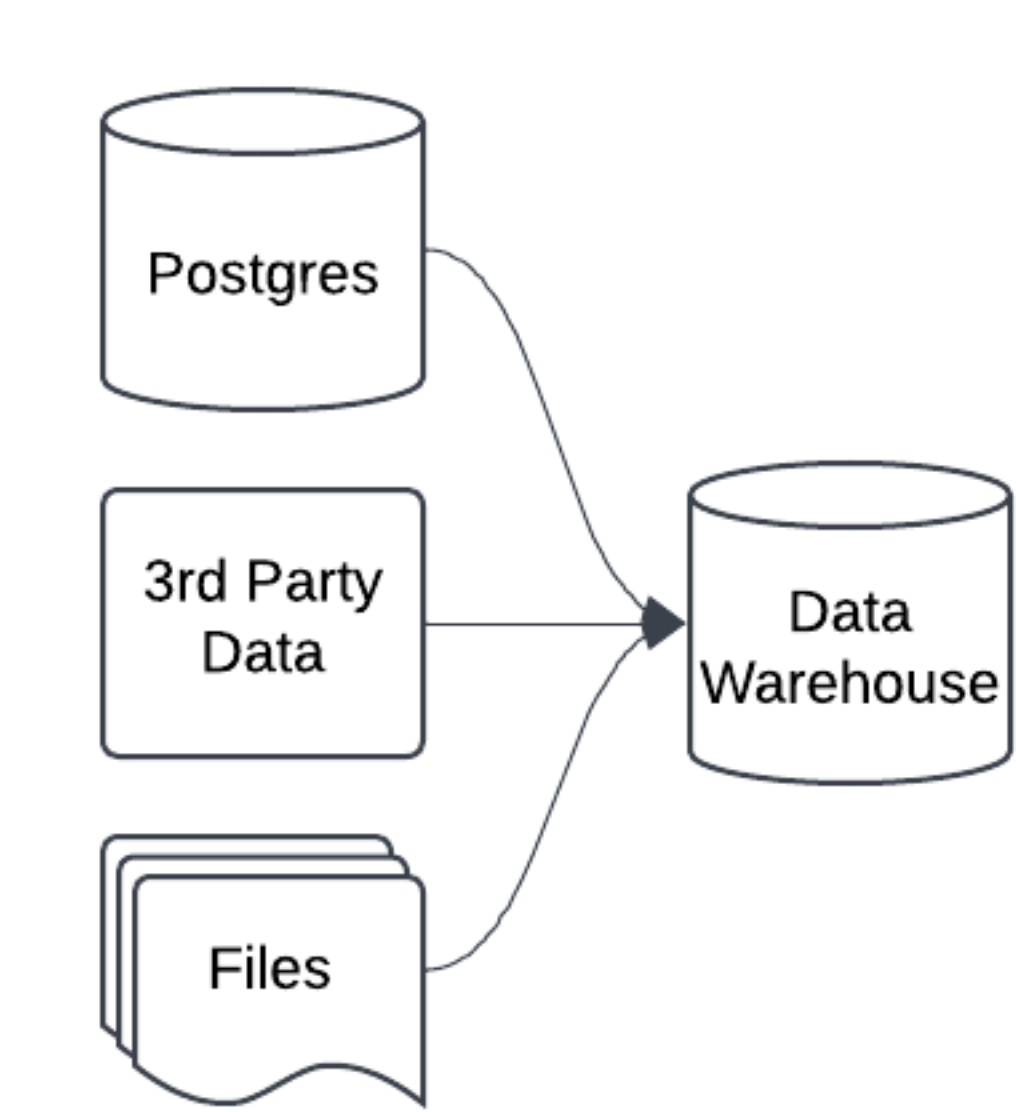
---

## Getting help

If you'd like some assistance while working through this course, reach out to the Dagster community on [Slack](#) in the `#dagster-university` channel. **Note:** The Dagster team is generally available during US business hours.

# What is ETL?

ETL stands for Extract, Transform, Load and is the process of consolidating data from various upstream sources into a single destination storage layer. These upstream sources often span multiple systems and data formats: including application databases, third-party services, and raw files. To fully leverage this data, it's typically best to bring everything into one centralized location, traditionally a data warehouse or data lake, where it can be standardized and made usable.



## ETL vs ELT

A quick note on definitions. If you're familiar with ETL, you may have also encountered ELT. The two approaches are very similar, but as the acronym suggests, the key difference is when the transformation happens. In ELT, data is loaded first into the destination system, and transformed afterward.

With the rise of modern data warehouses and lakes that support semi-structured and unstructured data, it's become less critical to transform data into a strict schema before loading. As a result, ETL and ELT are increasingly used interchangeably. Throughout this course, we'll refer to the process as ETL, even if some examples technically follow the ELT pattern.

Each stage of the ETL/ELT process can take many different forms:

Stage	Types
(E)xtract	Web scraping, external files (e.g., CSV, Excel), database replication, API ingestion, log parsing, message queues (Kafka, Kinesis)
(T)ransform	Data cleaning, normalization, filtering, aggregation, enrichment, joins, deduplication, business logic application, format conversion
(L)oad	Data warehouse ingestion (e.g., BigQuery, Snowflake, Redshift), database inserts/updates, data lake storage (e.g., S3, Delta Lake), streaming sinks, flat file outputs

A variety of tools are available to perform one or more of these stages. For example, Fivetran can handle both extraction and loading, while a tool like dbt is more commonly used for transformation.

## The Importance of ETL

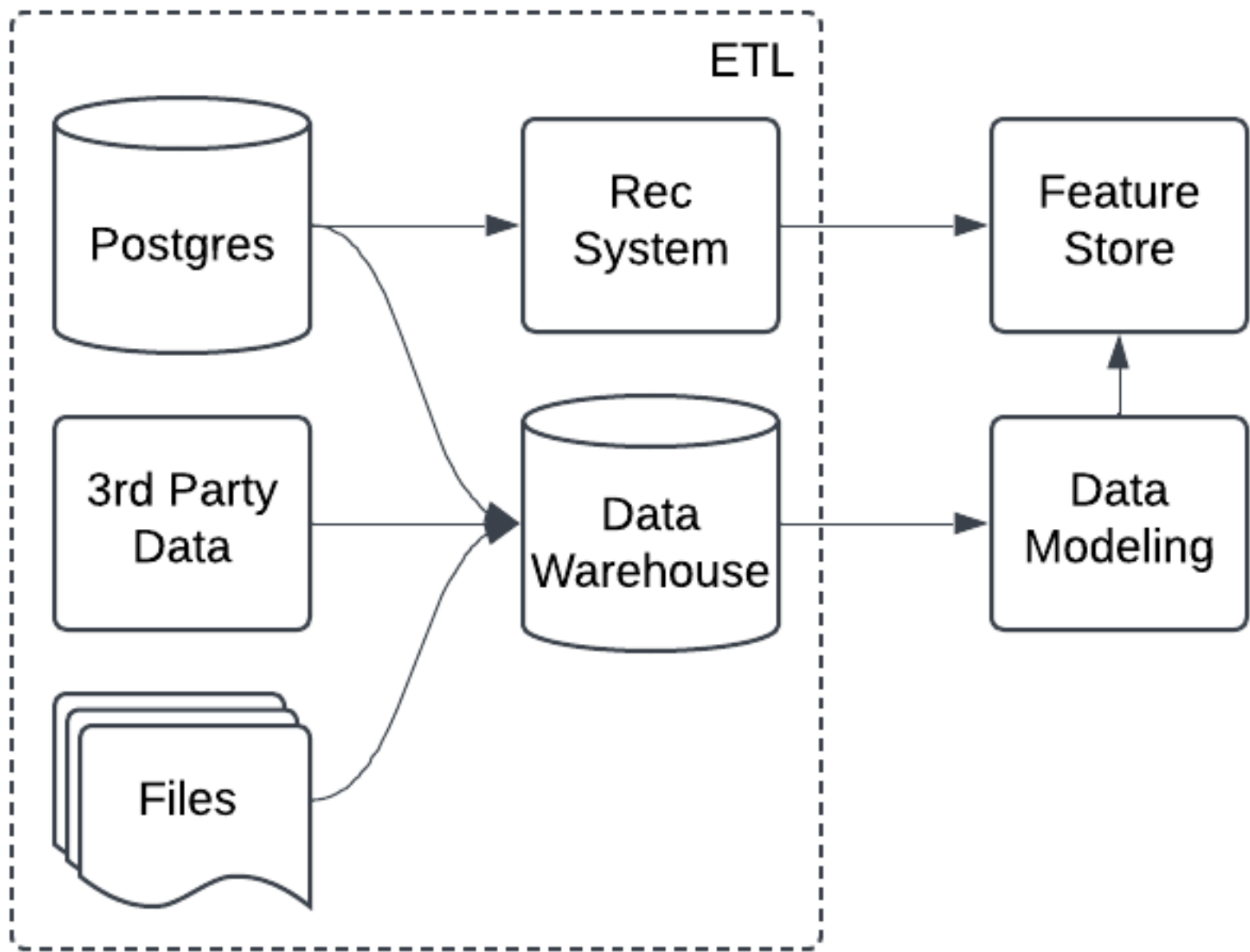
No matter the industry, ETL is foundational to data systems and applications. When implemented effectively, your data becomes a strategic moat that powers everything from operational dashboards to machine learning pipelines. Whether you're building classic BI reports or cutting-edge AI products, the value lies less in the tools and more in the quality and structure of your data.

Even in emerging areas like large language models (LLMs), it's not the model itself that defines success, but the clean, curated datasets used to generate embeddings and provide meaningful context. In short, great data makes great systems.

# ETL and Dagster

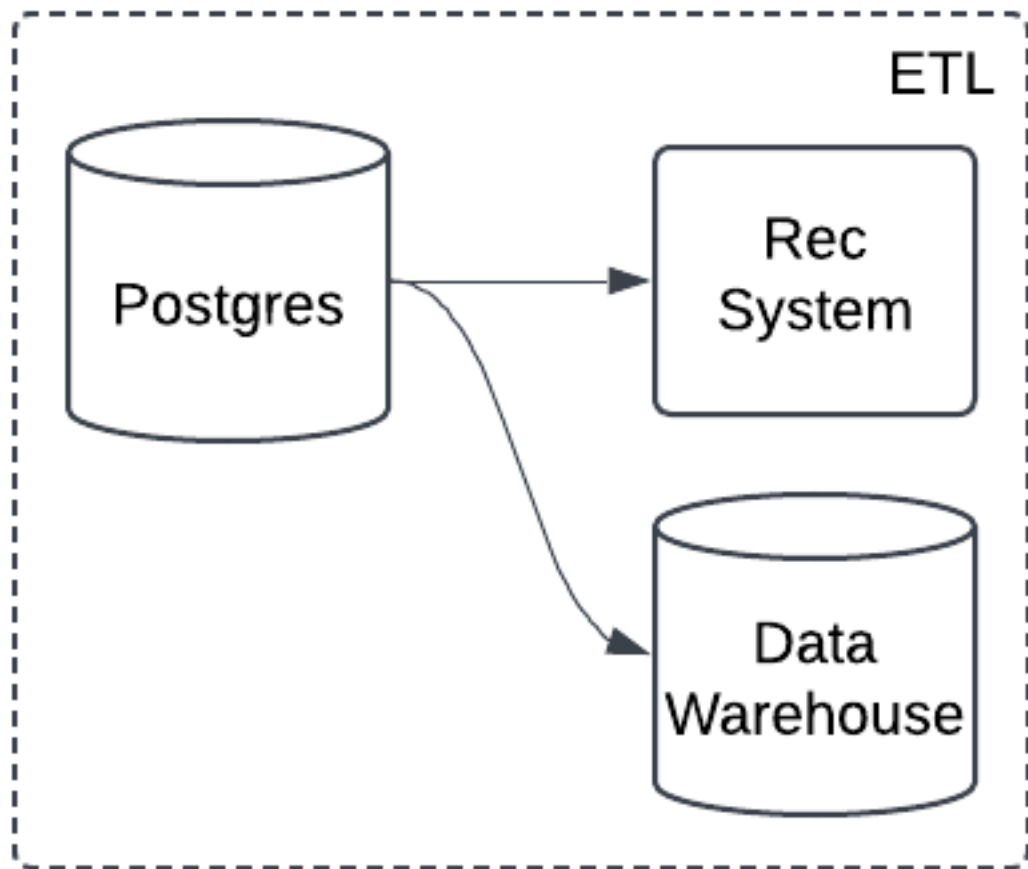
One of Dagster’s core strengths is breaking down data pipelines to their individual assets. This provides full lineage and visibility between different systems.

When visualizing your data stack, it’s often helpful to think of data flowing from left to right. On the far left, you typically find your raw data sources and the ETL processes that bring that data into your platform. This is a logical starting point for building out a data platform: focusing on ETL assets helps you concentrate on the most important datasets and avoid duplicating effort.



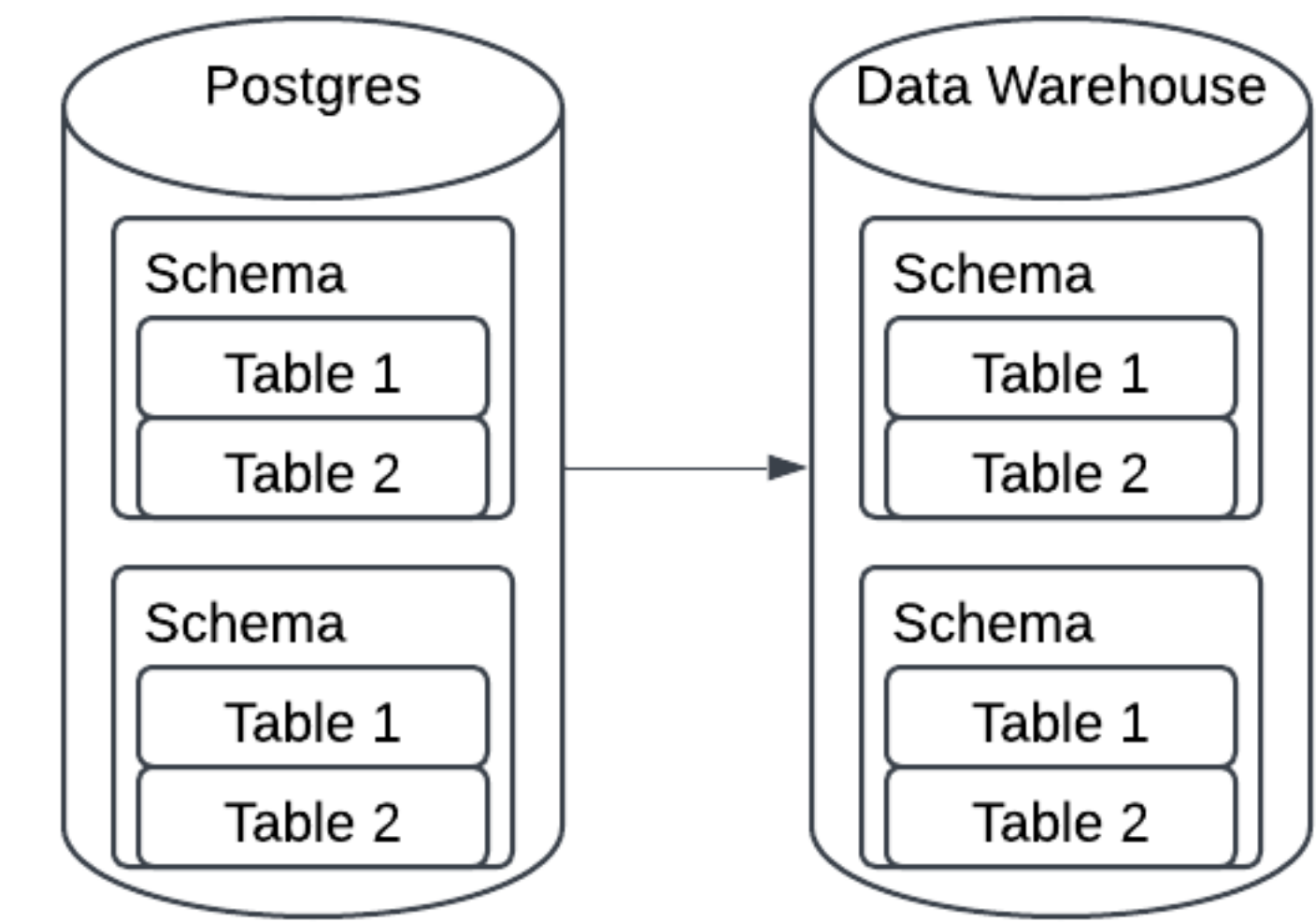
This asset based approach is what makes Dagster particularly well-suited for managing ETL pipelines. Because source assets are so fundamental to building with data, they tend to be reused across multiple projects. For example, if you're ingesting data from an application database, that data may feed into both analytics dashboards and machine learning workflows.

Without an asset based approach, it can get lost that multiple processes rely on the same sources.



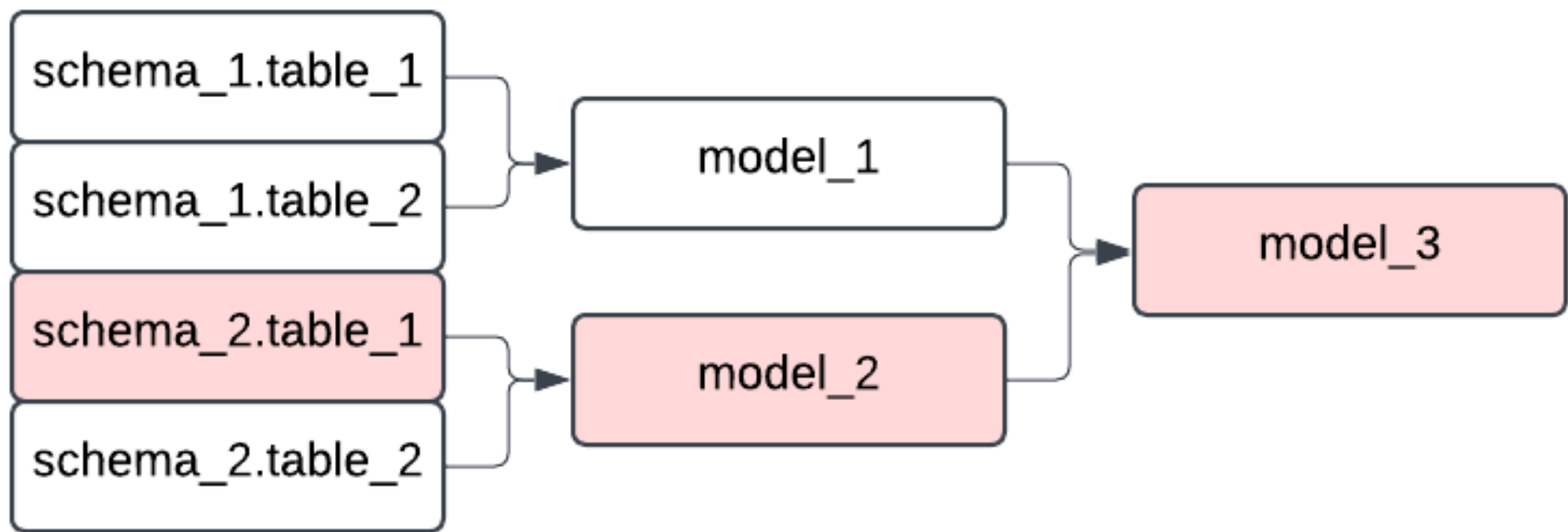
## Source asset granularity

Another reason an asset based approach tends to work well for ETL is that data sources tend to represent multiple individual entities. Consider a pipeline that ingests data from your application database, you’re likely pulling in multiple schema or tables, each used by specific data applications.



Each of these entities should be tracked as its own asset, so you can associate downstream processes with each one individually. That granularity gives you the ability to monitor, reason about, and recover from failures more effectively.

For example, if one source table fails to ingest, an asset based approach allows you to quickly understand which downstream assets and applications are impacted. This level of observability and control is what makes asset-based orchestration so powerful — especially in the context of managing critical ETL pipelines.



# Project preview

In this course, we'll focus on ETL and how to manage data ingestion using Dagster. All of the examples will walk through real-world ETL workflows you're likely to encounter, covering a variety of data sources and the unique challenges they present.

By the end of the course, you will:

- Create scheduled and event-driven pipelines to ingest files
- Build a custom resource to pull data from an external API
- Use Embedded ETL (with dlt) to build more resilient applications
- Replicate data across databases
- Refactor your code using Dagster Components for better modularity and reuse

If you get stuck or want to jump ahead, check out the [finished project here on GitHub](#).

## Set up requirements

This is an interactive class where you will be coding. In order to follow along you can either run the project locally on your own machine or work with the code in Github Codespaces where all requirements will be set.

***You will only need to follow the set up for one of these options, please skip the other:***

- [Local Development](#)
- [Github Codespaces](#)

# Set up local

This will set up Dagster for you local machine. If you would prefer to do this course in Github Codespaces, please follow [that guide](#).

- **To install git.** Refer to the [Git documentation](#) if you don't have this installed.
- **To have Python installed.** Dagster supports Python 3.9 - 3.12 (3.12 recommended).
- **To install a package manager.** To manage the python packages, we recommend `uv` which Dagster uses internally.

## Clone the Dagster University project

[Clone the Github repository](#). The command will depend on if you choose to clone with `ssh` or `https`:

Option	Command
ssh	<code>git clone git@github.com:dagster-io/project-dagster-university.git</code>
https	<code>git clone https://github.com/dagster-io/project-dagster-university.git</code>

After cloning the Dagster University project, navigate to the specific course directory within the repository. All courses are located inside the `dagster_university` folder.

For example, if you are completing "Dagster Essentials", change into that directory:

```
cd dagster_university/dagster_essentials
```

## Install the dependencies

### uv

If you do not have `uv` installed, you can do so in a [number of ways](#). To install the python dependencies with `uv`. While in the course specific directory run the following:

```
uv sync
```

This will create a virtual environment and install the required dependencies. To enter the newly created virtual environment:

OS	Command
MacOS	<code>source .venv/bin/activate</code>
Windows	<code>.venv\Scripts\activate</code>

### pip

To install the python dependencies with `pip`. While in the course specific directory run the following:

```
python3 -m venv .venv
```

This will create a virtual environment. To enter the newly created virtual environment:

OS	Command
MacOS	<code>source .venv/bin/activate</code>
Windows	<code>.venv\Scripts\activate</code>

To install the required dependencies:

```
pip install -e ".[dev]"
```



# Set up with Github Codespace

Instead of you setting up a local environment, you can use [Github Codespaces](#). This will allow you to work through this course and edit code in this repository in a cloud based environment.

## Creating a Github Codespace

There are unique Codespaces for the different courses in Dagster University. Be sure to select the create one creating a Codespace.

1. While logged into Github, go to the [Codespaces page](#).
2. In the top right, select "New Codespace"
3. Create a Codespace using the following.

Field	Value
Repository	dagster-io/project-dagster-university
Branch	main
Dev container configuration	Specific course name
Region	US East
Machine type	2-core

### Create a new codespace

Repository

To be cloned into your codespace

dagster-io/project-dagster-university

Codespace usage for this repository is paid for by dehume.

Branch

This branch will be checked out on creation

main

Dev container configuration

Your codespace will use this configuration

Default project configuration

Region

Your codespace will run in the selected region

Machine type

Resources for your codespace

Default project configuration

Dagster & dbt

Dagster Essentials

Testing with Dagster

Create codespace

4. Click "Create codespace"

The first time you create a codespace it may take a minute for everything to start. You will then be dropped in an interactive editor containing the code for the entire Dagster University repository.

## Working in the Codespace

In the terminal at the bottom of the Codespace IDE, navigate to the specific course directory. For example, if you are completing "Dagster Essentials", change into that directory:

```
cd dagster_university/dagster_essentials
```

To ensure everything is working you can launch the Dagster UI.

```
dg dev
```

After Dagster starts running you will be prompted to open the Dagster UI within your browser. Click "Open in Browser".

Your application (Dagster) running on port 3000 is available. [See all forwarded ports](#)

Open in Browser

Make Public

## Stopping your Github Codespace

Be sure to stop your Codespace when you are not using it. Github provides personal accounts [120 cores hours per month](#).

Your codespaces

Go to docs

New codespace

Explore quick start templates

Blank

By github

Start with a blank canvas or import any packages you need.

Use this template

React

By github

A popular JavaScript library for building user interfaces based on UI components.

Use this template

Jupyter Notebook

By github

JupyterLab is the latest web-based interactive development environment for notebooks, code, and data.

Use this template

.NET

By github

A full-stack framework for building web applications written in C# and .NET 8.

Use this template

Owned by dehume

dagster-io/project-dagster-university

fluffy xylophone

main No changes

2-core • 8GB RAM • 32GB

2.34 GB

Active

Rename

Export changes to a branch

Change machine type

Stop codespace

Auto-delete codespace

Open in Browser

Open in Visual Studio Code

Open in JupyterLab

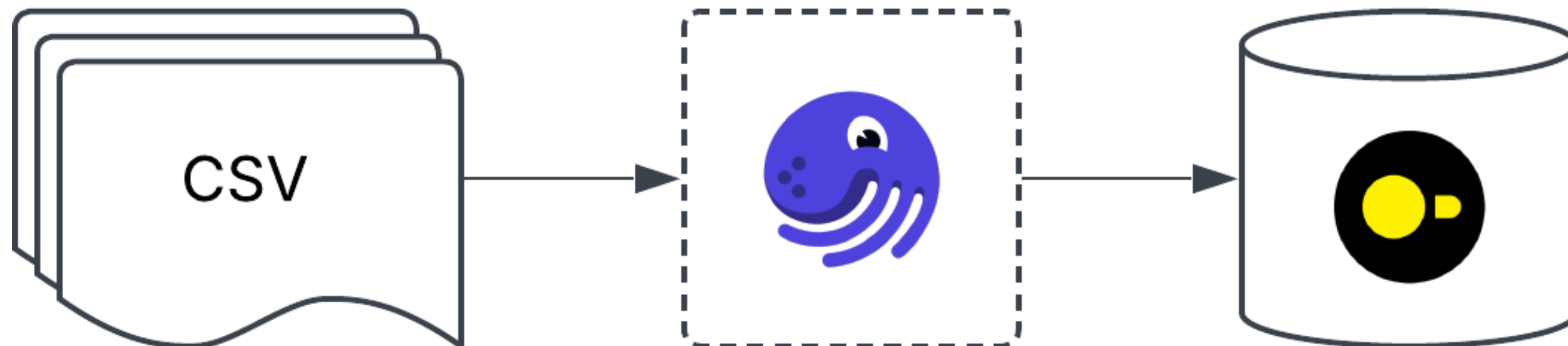
Delete

# Overview

We'll begin by loading data from an external system into a data warehouse. In production, this source might be something like S3 or Azure Blob Storage, but for now, we'll keep things simple by using the local filesystem.

This is one of the most common ETL use cases, and many data warehouses support it natively. For example, loading data from S3 can often be accomplished by executing a SQL command directly within the warehouse. However, there are also situations where some level of custom processing is needed before the data can be ingested.

In either case, this provides a solid starting point to explore the different ways ETL can be implemented in Dagster.





File import

As mentioned, we'll start by loading a file into a data warehouse. While we're using a local file and a local DuckDB instance for simplicity, the same principles apply regardless of the source and destination. You could easily adapt this pipeline to pull data from another storage layer and load it into a different data warehouse.

Config

Let's consider the first asset we need. As we discussed, the logical approach to designing ETL pipelines is to start as far left in the diagram as possible and follow the data as it progresses through our system.

If we're building a pipeline to load files into a data warehouse, we should start with the files we want to important. When we think about those files, it's likely that we'll have more than one file to ingest, with new files arriving over time. Because of this, we want to avoid hardcoding the pipeline for a single file.

Instead, we can use a [run configuration](#) to parameterize the process. This allows us to dynamically specify which file to load each time we execute the pipeline.

First, we'll define a run configuration so we can set the file path. In the `src/dagster_and_etl/defs/assets.py` file add the code for the run configuration:

```
# src/dagster_and_etl/defs/assets.py
import dagster as dg
from pathlib import Path

class IngestionFileConfig(dg.Config):
    path: str
```

Run configurations inherit from the Dagster `Config` class and allow us to define schemas that can be used in our Dagster assets.

Next, we will write an asset that uses this config. To make things easier, the asset will be designed to look within a specific directory relative to the asset file itself. This way, we only need to provide the file name, rather than the full path, when specifying the configuration:

```
@dg.asset()
def import_file(context: dg.AssetExecutionContext, config: IngestionFileConfig) -> str:
    file_path = (
        Path(__file__).absolute().parent / f"../../../data/source/{config.path}"
    )
    return str(file_path.resolve())
```

This asset will take in the run config and return the full file path string of a file in the `dagster_and_etl/data/source` directory.

Loading data

Now that we've identified the file we want to load, we can define the destination. In this case, we'll use [DuckDB](#). DuckDB is an in-process Online Analytical Processing (OLAP) database similar to Redshift and Snowflake, but designed to run locally with minimal setup.

In order to use DuckDB we need to establish a connection with our database. In Dagster we can do this with resources, so in the `resources.py` file, add the following:

```
# src/dagster_and_etl/defs/resources.py
import dagster as dg
from dagster_duckdb import DuckDBResource

@dg.definitions
def resources():
    return dg.Definitions(
        resources={
            "database": DuckDBResource(
                database="data/staging/data.duckdb",
            ),
        }
    )
```

The code above does the following:

- 1. Maps the `DuckDBResource` resource to the `Definitions` object.
- 2. Initializes the `DuckDBResource` to use the local file at `data/staging/data.duckdb` as the backend.

Defining our resource here means we can use it throughout out Dagster project.

DuckDB asset

Like most databases, DuckDB offers multiple ways to ingest data. One common way when working with files is using DuckDB's `COPY` statement. A typical `COPY` command follows this pattern:

```
COPY {table name} FROM {source}
```

This can include additional parameters to specify things like file type and formatting, but DuckDB often infers these settings correctly so we will not worry about it here.

As well as loading data into DuckDB, we need a destination table defined. We can design an asset that creates a table to match the schema of our file and load the file:

```
# src/dagster_and_etl/defs/assets.py
from dagster_duckdb import DuckDBResource

@dg.asset(
    kinds={"duckdb"},
)
def duckdb_table(
    context: dg.AssetExecutionContext,
    database: DuckDBResource,
    import_file,
):
    table_name = "raw_data"
    with database.get_connection() as conn:
        table_query = f"""
            create table if not exists {table_name} (
                date date,
                share_price float,
                amount float,
                spend float,
                shift float,
                spread float
            )
        """
        conn.execute(table_query)
        conn.execute(f"copy {table_name} from '{import_file}';")
```

The code above does the following:

- 1. Connects to DuckDB using the Dagster resource `DuckDBResource`.
- 2. Uses the DuckDB connection to create a table `raw_data` if that tables does not already exist. We define the schema of the table ourselves, knowing the structure of the data.
- 3. Runs a `COPY` for file path from the `import_file` asset into the `raw_data` table. `COPY` is an append command so if we run the command twice, the data will be loaded again.

Executing the assets

These two assets are all we need to ingest the data and there are multiple ways to do so in Dagster.

Command line ( `dg launch` )

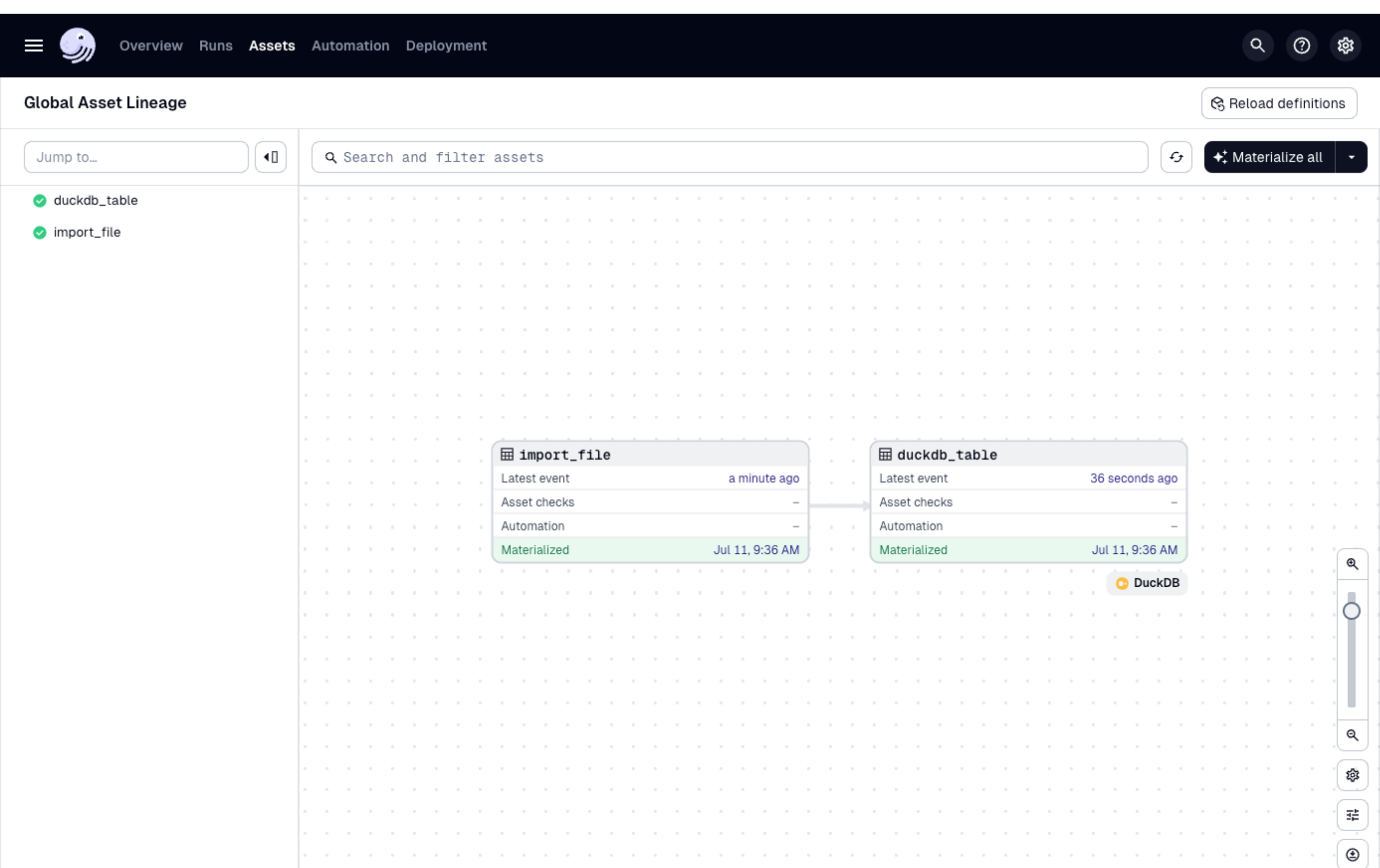
```
dg launch --assets import_file,duckdb_table --config-json '{"resources":{"database":{"config":{"database":
```

Web UI ( `dg dev` )

```
dg dev
```

Within the UI, materialize the assets while providing the following run execution:

```
resources:
  database:
    config:
      database: data/staging/data.duckdb
ops:
  import_file:
    config:
      path: 2018-01-22.csv
```



In either case (whether using the CLI or the UI) remember that we need to provide a value for the file path as a run configuration in orders to execute the assets.

Confirm data

To check that everything has loaded correctly, we can connect to the DuckDB database using the [DuckDB CLI](#) and run a query for the table we just made.

```
> duckdb data/staging/data.duckdb
D SELECT * FROM raw_data;
```

date	share_price	amount	spend	shift	spread
date	float	float	float	float	float
2018-09-28	264.77	33597288.0	270.26	278.0	260.555
2018-09-27	307.52	7337760.0	312.9	314.96	306.91
2018-09-26	309.58	7835863.0	301.91	313.89	301.1093
2018-09-25	300.99	4472287.0	300.0	304.6	296.5
2018-09-24	299.68	4834384.0	298.48	302.9993	293.58
2018-09-21	299.1	5038497.0	297.7	300.58	295.37
2018-09-20	298.33	7332477.0	303.56	305.98	293.33
2018-09-19	299.02	8264353.0	280.51	300.0	280.5

## Data integrity

Our ETL pipeline can already execute successfully. We provide a file, and it gets loaded into DuckDB. While it's encouraging to see data load without errors, there's actually something worse than a pipeline that fails to run, a pipeline that loads bad data.

Poor-quality data can cause more damage in an ETL workflow than a job that simply fails. Once bad data is ingested, it can have cascading effects, affecting all downstream consumers and applications. When fixing a bug around bad data, we not only have to fix the original issue, we're also tasked with cleaning up every process that depends on it.

In many cases, the data we ingest comes from systems outside our control. We often treat these sources as their own source of truth and assume they enforce some form of validation rules. But when we know there are specific characteristics that could disrupt our pipelines or outputs, it becomes critical to add validation checks as part of our ingestion process.

## Asset checks

In Dagster, one way to handle data validation is with asset checks. Asset checks let you define logic for your code to ensure they meet predefined quality standards. For example, we can attach an asset check to `import_file` to verify that the file structure and contents meet our expectations.

Let's start by writing an asset check to ensure the "share\_price" column in the file contains only valid, non-zero values. Before we think about the asset check, what might that logic look like if we were just reading the file:

```
import csv

with open("my_file.csv", mode="r", encoding="utf-8") as file:
    reader = csv.DictReader(file)
    data = (row for row in reader)

    for row in data:
        if float(row["share_price"]) <= 0:
            raise ValueError("Share price is invalid")
```

We would want to read the file, iterate through the rows and raise an exception if we find a row that does not meet our criteria.

Here is that similar logic within an asset check:

```
# src/dagster_and_etl/defs/assets.py
import csv

@dg.asset_check(
    asset=import_file,
    blocking=True,
    description="Ensure file contains no zero value shares",
)

def not_empty(
    context: dg.AssetCheckExecutionContext,
    import_file,
) -> dg.AssetCheckResult:
    with open(import_file, mode="r", encoding="utf-8") as file:
        reader = csv.DictReader(file)
        data = (row for row in reader)

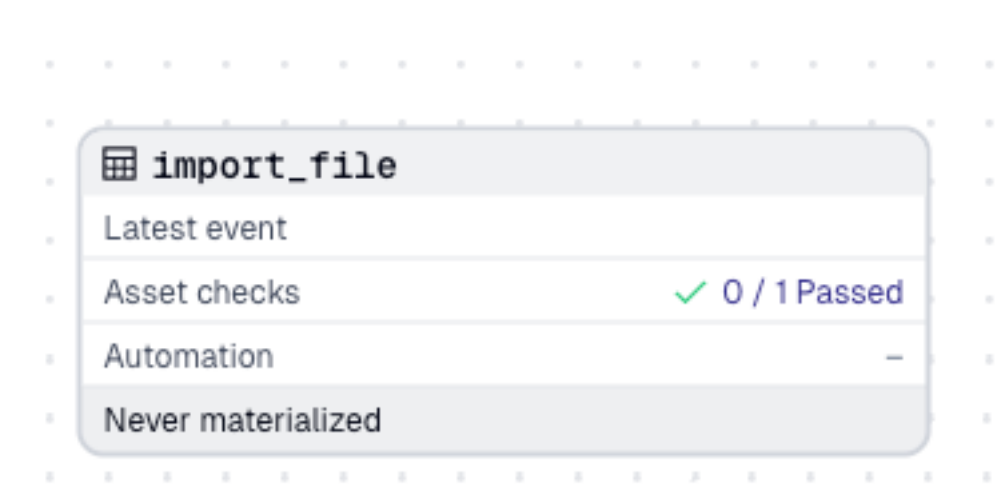
        for row in data:
            if float(row["share_price"]) <= 0:
                return dg.AssetCheckResult(
                    passed=False,
                    metadata={"'share' is below 0": row},
                )

    return dg.AssetCheckResult(
        passed=True,
    )
```

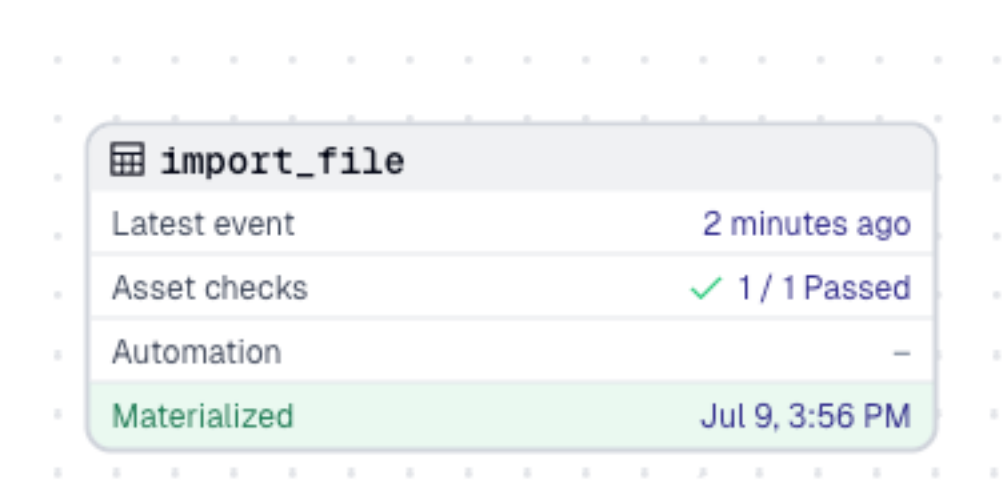
The code above does the following:

1. Uses the `@dg.asset_check` decorator to define an asset check. It references the `import_file` asset and sets `blocking=True`, which prevents downstream assets from executing if the check fails.
2. Reads the file path returned by the `import_file` asset and parses it as a CSV.
3. Iterates through each row to check whether the "share\_price" value is less than or equal to zero.
4. If a row fails the validation check, the `AssetCheckResult` will return as failed, otherwise this check will pass.

When you launch `dg dev` to view the assets in our pipeline, you won't see an additional node in the graph since the asset check is visually tied to the `import_file` asset.



Now, when you re-execute the pipeline, a green dot will appear on the `import_file` node if the asset check passes, indicating both successful materialization and validation.



If the check fails, the dot will appear red, and the downstream asset will not run. This helps catch data issues early in the process.

OverviewRunsAssetsAutomationDeployment

Assets > import\_file

Materialize

OverviewEventsChecksLineage

Checks (1)

Execute all

Filter checks

invalid\_share\_price

Succeeded

invalid\_share\_price

Overview

Execution history

Automation history

About

This is a **blocking** asset check.

Ensure file contains no zero value shares

Latest execution

Evaluation result	Timestamp	Target materialization
Passed	Jul 9, 3:56 PM	Jul 9, 3:56 PM

Plots

Plots are automatically generated by metadata

Include numeric metadata entries in your check metadata to see data graphed.

View documentation



Partitions

When running ETL pipelines in production, it's important to have a reliable way to track what data has been loaded, especially when a pipeline has matured and contains a longer execution history.

Without a structured approach, it can become difficult to manage which data has been processed. How do we know which files have yet to be processed? Dagster addresses this challenge with partitions. Partitions allow you to break down your data into smaller, logical segments: such as date, region, or category. Each segment can then be treated and tacked as an independent unit tied to a specific execution.

Partitions can then be tied to assets. Configured an asset with partitions allows you to:

- Materialize and update only specific partitions, avoiding unnecessary reprocessing of unchanged data.
- Launch targeted backfills to reprocess historical data or recover from failures without rerunning the entire pipeline.
- Track the status and lineage of each partition independently, giving you better visibility and control over your data workflows.

Creating partitions

Let's go back to our `import_file` asset. What is a logical way to divide that data if our three files look like this?

- 2018-01-22.csv
- 2018-01-23.csv
- 2018-01-24.csv

Based on these files, it would be safe to assume that each corresponds to a specific day. This lends itself to a daily partition which we can configure with Dagster like so:

```
partitions_def = dg.DailyPartitionsDefinition(  
    start_date="2018-01-21",  
    end_date="2018-01-24",  
)
```

This creates a partition specifically for the dates 2018-01-22 to 2018-01-24. If we removed the upper bound ( `end_date` ) this partition would include all days from 2018-01-22 to present. This can be useful for new incoming files but for this example we will limit our partition to only include the days contained in our local files.

Partitioned assets

Now that we've defined our partition, we can use the partition in a new asset called `import_partition_file` . This asset will rely on the partition key instead of the `FilePath` run configuration to determine which file should be processed.

The core logic of the asset remains the same but now you can run the pipeline for each day between 2018-01-21 and 2018-01-23, with each partition corresponding to a file for that date. This allows you to scale execution, track progress per partition, and reprocess specific days as needed:

```
# src/dagster_and_etl/defs/assets.py  
@dg.asset(  
    partitions_def=partitions_def,  
)  
def import_partition_file(context: dg.AssetExecutionContext) -> str:  
    file_path = (  
        Path(__file__).absolute().parent  
        / f"../../data/source/{context.partition_key}.csv"  
    )  
    return str(file_path.resolve())
```

Finally we can create a new downstream asset that relies on the partitioned data:

```
# src/dagster_and_etl/defs/assets.py  
@dg.asset(  
    kinds={"duckdb"},  
    partitions_def=partitions_def,  
)  
def duckdb_partition_table(  
    context: dg.AssetExecutionContext,  
    database: DuckDBResource,  
    import_partition_file,  
):  
    table_name = "raw_partition_data"  
    with database.get_connection() as conn:  
        table_query = f"""  
            create table if not exists {table_name} (  
                date date,  
                share_price float,  
                amount float,  
                spend float,  
                shift float,  
                spread float  
            )  
        """  
        conn.execute(table_query)  
        conn.execute(  
            f"delete from {table_name} where date = '{context.partition_key}';"  
        )  
        conn.execute(f"copy {table_name} from '{import_partition_file}';")
```

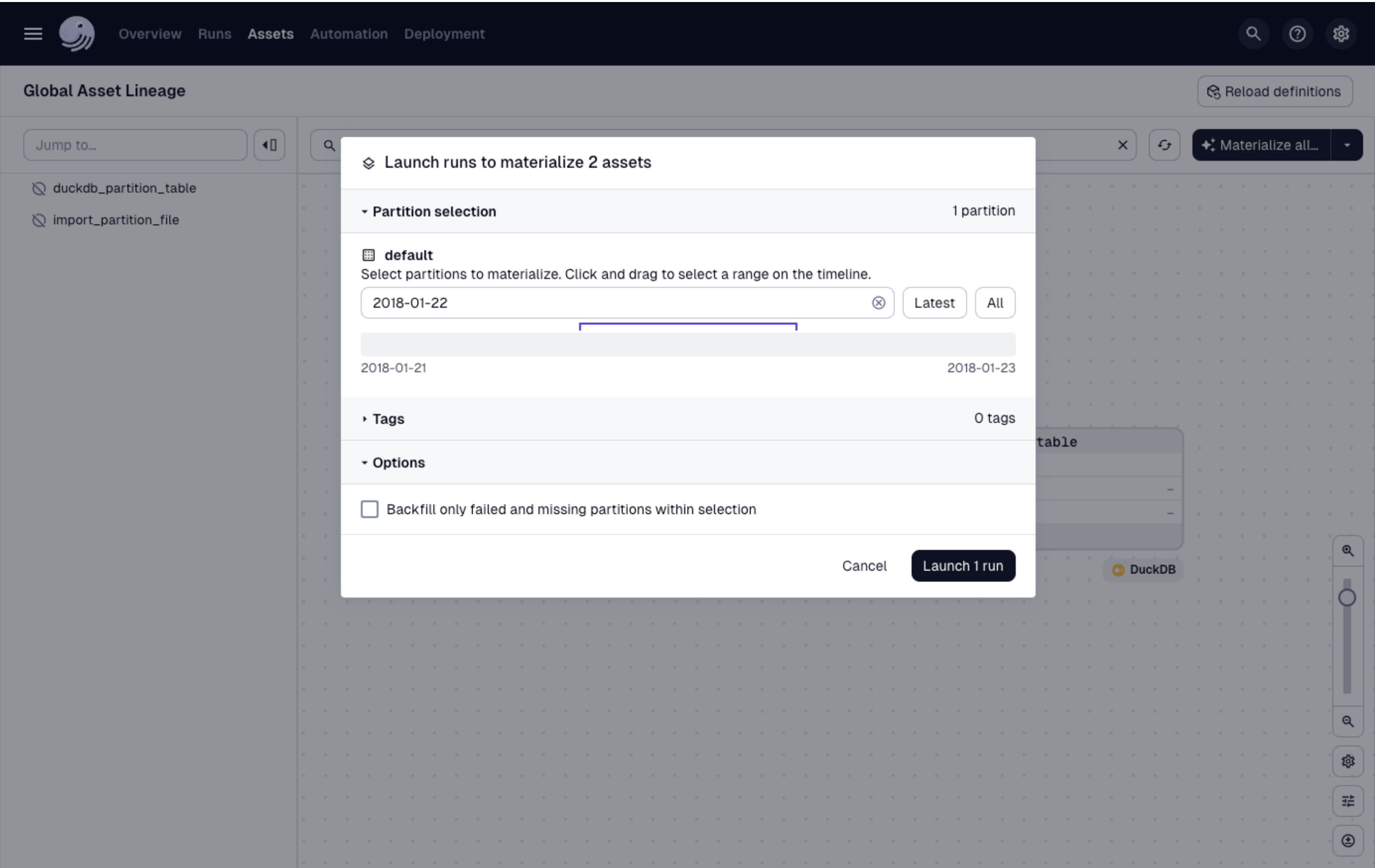
This is very similar to our original logic except for are two key differences:

1. We now include the partition ( `partitions_def` ) in the `@dg.asset` decorator .
2. We add a `delete from...` SQL statement targeting the table for the specific partition date. This ensures the pipeline is idempotent, allowing us to run backfills without the risk of data duplication.

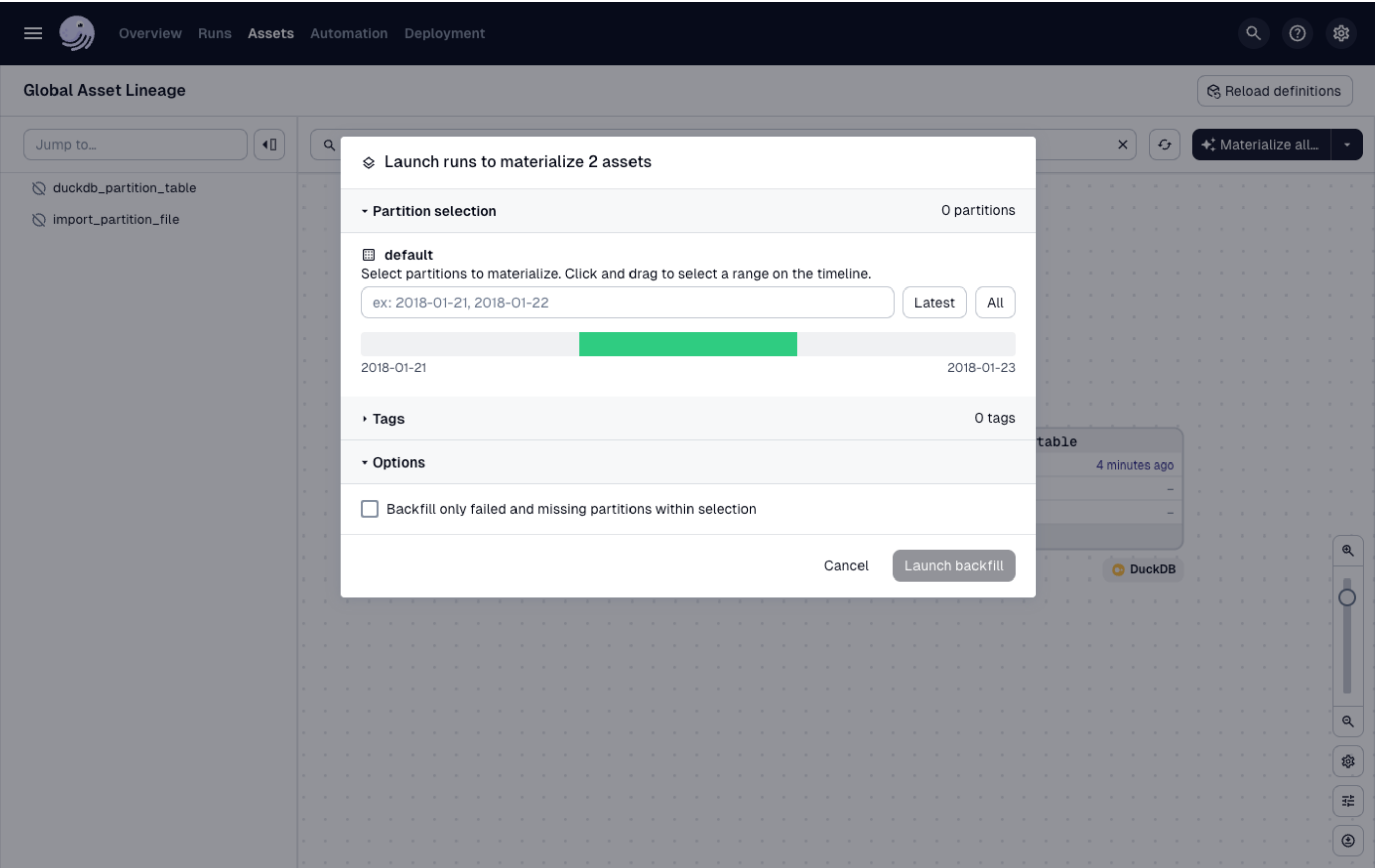
A quick not that deleting from the table before loading new data is one strategy for achieving idempotence. Another approach is to import the incoming data into a staging location and then upsert the new data into the final table. This method has the advantage of preserving existing data in the final table in case an error occurs during the copy process. And while the staging strategy is generally more resilient, we will use the simpler `delete from...` method for the purposes of this course.

Executing partitions

Similar to the file import, we can trigger executions for specific partitions. If we wanted to ingest the same data as previously, we would simply select the partition for "2018-01-22".



If we want to execute the assets again, we can see that the information is tracked.



# Complex partitions

Time based partitions are usually easy to implement. They follow consistent patterns, such as daily, weekly, or monthly, and have predictable boundaries.

But what about cases where the upstream data doesn’t follow a fixed pattern? For example, what if the data is grouped by something like customer name? In this case, the set of partitions may evolve over time as new customers are added.

How could we handle this with partitions to ensure we can still track our inputs if they follow an irregular pattern.

## Dynamic partitions

As well as time-based partitions, Dagster also supports dynamic partitions. These allow for more flexibility. Unlike time-based partitions, dynamic partitions let you control exactly what values are included and grow the partition set as new data categories (like customer names) appear.

Creating a dynamic partition is simple — all you need is a name:

```
dynamic_partitions_def = dg.DynamicPartitionsDefinition(name="dynamic_partition")
```

This defines a dynamic partition set named "dynamic\_partition", which you can configure with whatever partitions we want.

## Dynamic partitioned assets

Similar to our time based partition, let's create a new set of assets. Once again the the logic of our asset will remain the same except it uses the dynamic partition:

```
# src/dagster_and_etl/defs/assets.py
@dg.asset(
    partitions_def=dynamic_partitions_def,
)
def import_dynamic_partition_file(context: dg.AssetExecutionContext) -> str:
    file_path = (
        Path(__file__).absolute().parent
        / f"../../data/source/{context.partition_key}.csv"
    )
    return str(file_path.resolve())
```

Now, if we attempted to execute this asset in Dagster, you’d notice that it cannot be materialized. That is because no partitions have been added to `dynamic_partitions_def` . This makes sense as dynamic partitions are empty by default and have no built-in logic to determine which partition keys should exist. It’s up to us to register the partition values we want to execute.

Before we execute the pipeline, let’s define the downstream asset that loads data into DuckDB, using the values from our dynamic partition:

```
# src/dagster_and_etl/defs/assets.py
@dg.asset(
    kinds={"duckdb"},
    partitions_def=dynamic_partitions_def,
)
def duckdb_dynamic_partition_table(
    context: dg.AssetExecutionContext,
    database: DuckDBResource,
    import_dynamic_partition_file,
):
    table_name = "raw_dynamic_partition_data"
    with database.get_connection() as conn:
        table_query = f"""
            create table if not exists {table_name} (
                date date,
                share_price float,
                amount float,
                spend float,
                shift float,
                spread float
            )
        """
        conn.execute(table_query)
        conn.execute(
            f"delete from {table_name} where date = '{context.partition_key}';"
        )
        conn.execute(f"copy {table_name} from '{import_dynamic_partition_file}';")
```

This setup should look very similar to the daily partition we created earlier. But how does it play out in practice? To better understand the distinction, let’s dive into the differences in how partitioned pipelines are triggered. Each type requires a different approach to scheduling and orchestration, depending on how and when new data becomes available.



# Triggering partitions

The question of how to automate your pipelines largely depends on how your data arrives. At a high level, there are two main strategies: scheduled and event-driven execution.

## Scheduled ETL

Scheduled ETL pipelines run at fixed intervals. This works well for data that arrives consistently. For example, if a file is uploaded every morning at 5:00 AM, you might schedule your ETL process to run at 5:15 AM (supplying a small buffer).

Scheduling is a simple and very common with ETL. It's easy to set up, reliable, and integrates cleanly with most orchestration tools. However, it lacks nuance. What if the file arrives late, say at 5:45 AM? Unless you've implemented proper alerting, your pipeline might fail silently. Even with alerts, you'd need to manually verify when file has actually arrived and trigger an ad-hoc run to process it.

## Event-driven ETL

Event-driven pipelines are triggered by a change in state, such as the arrival of a new file itself. In this model, the timing of the file's arrival doesn't matter (5:00 AM, 5:45 AM, 8:00 AM...). The detecting a new file is what triggers the pipeline.

This approach is more flexible and responsive, but it comes with additional complexity. You need a way to track state, so the system knows which files have already been processed and which ones are new. Without this, you risk duplicate processing or missed data.

## Scheduling in Dagster

Our two partitioned pipelines can demonstrate the differences between these two strategies. The time-based partitioned assets fit a schedule while the dynamic partitioned assets will use a sensor (which is Dagster's way to handle event driven architectures).

## Implementing schedules

For our scheduled pipeline, we'll use the assets associated with the `DailyPartitionsDefinition` partition. As a reminder, this partition definition requires a specific date (e.g., "2018-01-22") for the asset to execute.

Because we're using Dagster's built-in `DailyPartitionsDefinition` class to generate a fixed pattern of daily partitions, Dagster can automatically create a corresponding schedule for us. All we need to do is provide the job we want to run and define the cadence at which it should run. Dagster will handle generating the appropriate partition key for each execution:

```
import dagster as dg

import dagster_and_etl.defs.jobs as jobs

asset_partitioned_schedule = dg.build_schedule_from_partitioned_job(
    jobs.import_partition_job,
)
```

This makes it simple to set up reliable, automated ETL for any use case where data arrives on a regular schedule.

## Implementing Event-driven

As mentioned earlier, event-driven pipelines are a bit more complex because they require maintaining state, specifically knowing which data has already been processed and which is new. The good news is that Dagster handles most of the complexity around state management through an abstraction called sensors.

Sensors in Dagster allow you to monitor external systems, like cloud storage, and trigger pipeline runs when new data is detected. They are particularly useful when working with dynamic partitions, where the set of valid partition keys is not always known.

Here's an example of what a sensor might look like for a dynamically partitioned asset:

```
# src/dagster_and_etl/defs/sensors.py
import json
import os
from pathlib import Path

import dagster as dg

import dagster_and_etl.defs.jobs as jobs
from dagster_and_etl.defs.assets import dynamic_partitions_def

@dg.sensor(target=jobs.import_dynamic_partition_job)
def dynamic_sensor(context: dg.SensorEvaluationContext):
    file_path = Path(__file__).absolute().parent / "../../data/source/"

    previous_state = json.loads(context.cursor) if context.cursor else {}
    current_state = {}
    runs_to_request = []
    dynamic_partitions_requests = []

    for filename in os.listdir(file_path):
        if filename not in previous_state:
            partition_key = filename.split(".")[0]
            last_modified = os.path.getmtime(file_path)
            current_state[filename] = last_modified

            dynamic_partitions_requests.append(partition_key)
            runs_to_request.append(
                dg.RunRequest(
                    run_key=filename,
                    partition_key=partition_key,
                )
            )

    return dg.SensorResult(
        run_requests=runs_to_request,
        cursor=json.dumps(current_state),
        dynamic_partitions_requests=[
            dynamic_partitions_def.build_add_request(dynamic_partitions_requests)
        ],
    )
```

The code above does the following:

1. Defines a sensor using the `dg.sensor` decorator for our `import_dynamic_partition_job` job.
2. Sets the current state from the `context` of the sensor. This determines the history of what the sensor has already processed.
3. Iterates through the files in the `data/sources` directory and determines if there are any new files since the last time the sensor ran.
4. Executes the `import_dynamic_partition_job` for any new files that have been added.

Now if we enable this sensor, it will trigger executions for all three files in the `data/sources` directory.

Event-driven pipelines like this can be more resilient and responsive, but they come with some important considerations.

First, you need access to the system where state is being checked. In our example, this isn't an issue since we're monitoring the local file system. But what if the files lived in a remote system where we don't have full read access? That could limit our ability to detect changes reliably.

You also need to ensure that your sensor logic is efficient. For example, if you're reading from an S3 bucket containing thousands of files, your sensor would need to query the entire bucket each time it runs. To mitigate this, it's often better to include logic that filters files by a specific prefix or folder path, reducing the scope of each scan.

Finally, consider what happens when a sensor is enabled for the first time. Because sensors typically detect anything that hasn't already been processed, the initial run can trigger a large number of events — potentially attempting to process everything at once.



You completed Knowledge check  
Your score

100%

RETAKE QUIZ

CONTINUE →

You answered 2 out of 2 questions correctly

1. When would a schedule be a good choice for your ETL process?



2. When would a sensor be a good choice for your ETL process?



# Cloud storage

What if you want to change the source of your ETL pipeline from local files to cloud storage? In most real-world scenarios, data is not stored locally but in a managed storage layer like Amazon S3. Let's walk through how to modify your Dagster assets to support S3 as the source for the ingestion pipeline.

**Note:** We won't run this code since we're not setting up an actual S3 bucket for testing. Instead, we'll focus on the structure and design changes required to support this integration.

## Redefining the run configuration

If you're new to AWS Simple Storage Service (S3): S3 is a cloud-based object storage service that lets you upload and retrieve data using "buckets" and "paths" (which are really just key prefixes).

```
s3://<bucket-name>/<optional-folder-path>/<object-name>
```

To support dynamic S3-based ingestion, we'll define a new config schema called IngestionFileS3Config. This configuration will have two fields:

Property	Type	Description
bucket	string	The name of the S3 bucket
path	string	The path within the S3 bucket of the object

What would this run configuration IngestionFileS3Config look like:

```
class IngestionFileS3Config(dg.Config):
    bucket: str
    path: str
```

This could be modeled as a single string (e.g. full S3 URI), but splitting bucket and path is often more practical, especially if your pipelines reuse the same bucket across different datasets.

## S3 assets

Next, let's update the import\_file and duckdb\_table assets to support S3-based ingestion. One key thing to note about DuckDB is that it can load data directly from S3 using a COPY statement, like so:

```
COPY my_table FROM 's3://my-bucket/my-file.csv' (FORMAT CSV, HEADER);
```

However, this approach only works if the S3 bucket is public. Most real-world buckets are private, but for the purpose of this example, we'll assume the S3 bucket is publicly accessible and does not require authentication.

With this background we can update our asset code:

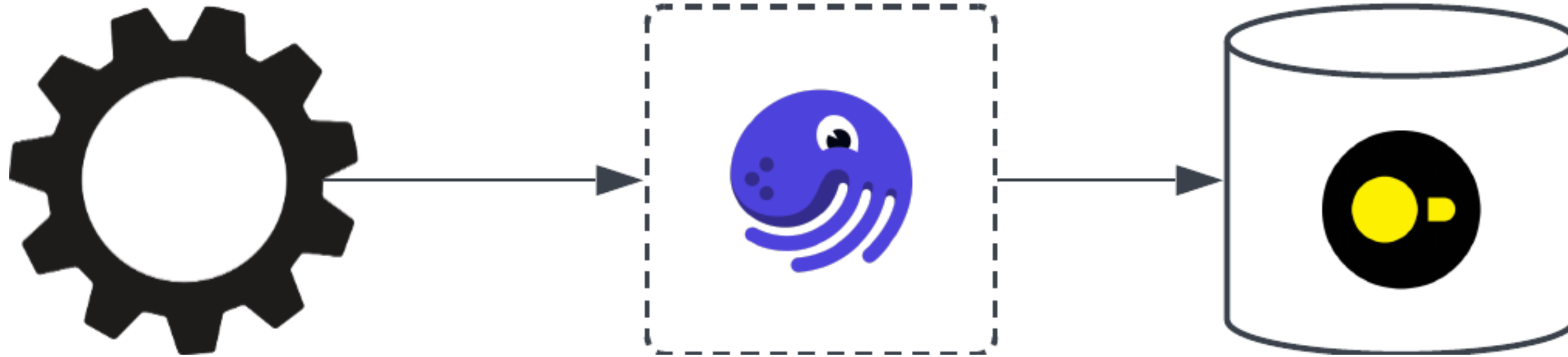
```
@dg.asset(
    kinds={"s3"},
)
def import_file_s3(
    context: dg.AssetExecutionContext,
    config: IngestionFileS3Config,
) -> str:
    s3_path = f"s3://{config.bucket}/{config.path}"
    return s3_path

@dg.asset(
    kinds={"duckdb"},
)
def duckdb_table_s3(
    context: dg.AssetExecutionContext,
    database: DuckDBResource,
    import_file_s3: str,
):
    table_name = "raw_s3_data"
    with database.get_connection() as conn:
        table_query = f"""
            create table if not exists {table_name} (
                date date,
                share_price float,
                amount float,
                spend float,
                shift float,
                spread float
            )
        """
        conn.execute(table_query)
        conn.execute(f"copy {table_name} from '{import_file_s3}' (format csv, header);")
```

Not much needs to change when moving from local files to cloud storage. The main takeaway is that loading static files from S3 closely mirrors loading them from the local filesystem. All the patterns and tools we've already applied (partitions, schedules, sensors) still apply. Whether the file is local or in S3 bucket, the logic for defining and orchestrating data assets remains largely the same.

# Overview

Now that we've covered CSV-based ingestion, let's move on to another common pattern: loading data from an API. We'll continue using the same destination (DuckDB) but introduce some new Dagster functionality to fetch data from an external API, apply light processing, and load the results into the database. This pattern is useful when working with third-party services, public datasets, or internal APIs that expose structured data for integration.



# APIs

Building ETL pipelines for APIs is highly dependent on the specific API, as not all APIs handle data access and delivery in the same way. Each API has its own structure, endpoints, authentication requirements, and query parameters which can all vary significantly.

Basic APIs often return data in JSON format as part of a GET request. When dealing with larger datasets, responses are typically paginated, meaning multiple requests must be made to retrieve the full set of results.

For APIs that manage large-scale or asynchronous data processing, direct responses may not be feasible. Instead, you may initiate a request that triggers a data export, with the resulting data landing in a separate storage layer (like S3) once it's ready. These types of APIs require polling or callback logic to track data readiness.

Because there's no universal standard for how APIs return data, the first step in building any ETL pipeline around an API is to carefully understand the API's structure and behavior.

## NASA API

For this lesson, we'll use one of the publicly available NASA APIs, specifically, the NeoWs (Near Earth Object Web Service), which provides information about asteroids. This is a relatively simple API that returns data in a JSON payload and doesn't require pagination or asynchronous processing, making it ideal for learning. It gives us a clean example of how to build an ETL pipeline from an external API into our DuckDB warehouse.

**Note:** This API requires an access key, but the process is quick and free. To get started, simply fill out a [short form](#) on NASA's API portal. Once submitted, you'll receive an API key that we'll use later in the course to authenticate our requests.

# API Resource

When working with external APIs in Dagster, it's often best to start by creating a resource. A resource provides a clean abstraction for external services, making it easy to reuse API logic across multiple assets. It also simplifies testing and long-term maintenance by isolating API-specific logic in a single, well-defined interface.

Before we write any code, let's review the characteristics of the NeoWs (Near Earth Object Web Service) API. The base URL for the endpoint is:

```
https://api.nasa.gov/neo/rest/v1/feed
```

This endpoint supports three query parameters:

Parameter	Type	Default	Description
start_date	YYYY-MM-DD	none	Starting date for asteroid search
end_date	YYYY-MM-DD	7 days after start_date	Ending date for asteroid search
api_key	string	DEMO_KEY	api.nasa.gov key for expanded usage

Given this structure, a full API request might look like:

```
https://api.nasa.gov/neo/rest/v1/feed?start_date=2015-09-07&end_date=2015-09-08&api_key=DEMO_KEY
```

The API will return a large JSON response that includes various metadata fields. To keep things simple, we'll focus only on the part we care about — the `near_earth_objects` field. This field contains the actual asteroid data, organized by date, and is all we need for our ETL pipeline.

## Coding our resource

Now that we know the API endpoint and the parameters required to make a call, let's write our resource. There are many ways to structure this, but we'll keep the implementation lean.

We'll create a resource called `NASAResource`, which is initialized from our API key. This resource will expose a single method: `get_near_earth_asteroids` with two parameters (`start_date`, `end_date`), which returns the parsed JSON response from the API.

Here's what that might look like added to the `resources.py`:

```
# src/dagster_and_etl/defs/assets.py
import dagster as dg
import requests

class NASAResource(dg.ConfigurableResource):
    api_key: str

    def get_near_earth_asteroids(self, start_date: str, end_date: str):
        url = "https://api.nasa.gov/neo/rest/v1/feed"
        params = {
            "start_date": start_date,
            "end_date": end_date,
            "api_key": self.api_key,
        }

        resp = requests.get(url, params=params)
        return resp.json()["near_earth_objects"][start_date]
```

Now that we have our resource defined, we can include it in the `Definitions` alongside the `DuckDBResource` resource in the `resources.py`:

```
# src/dagster_and_etl/defs/definitions.py
@dg.definitions
def resources():
    return dg.Definitions(
        resources={
            "nasa": NASAResource(
                api_key=dg.EnvVar("NASA_API_KEY"),
            ),
            "database": DuckDBResource(
                database="data/staging/data.duckdb",
            ),
        },
    )
```

**Note:** Remember you will need to set the environment variable `NASA_API_KEY` to the API key you created if you want to execute this pipeline.



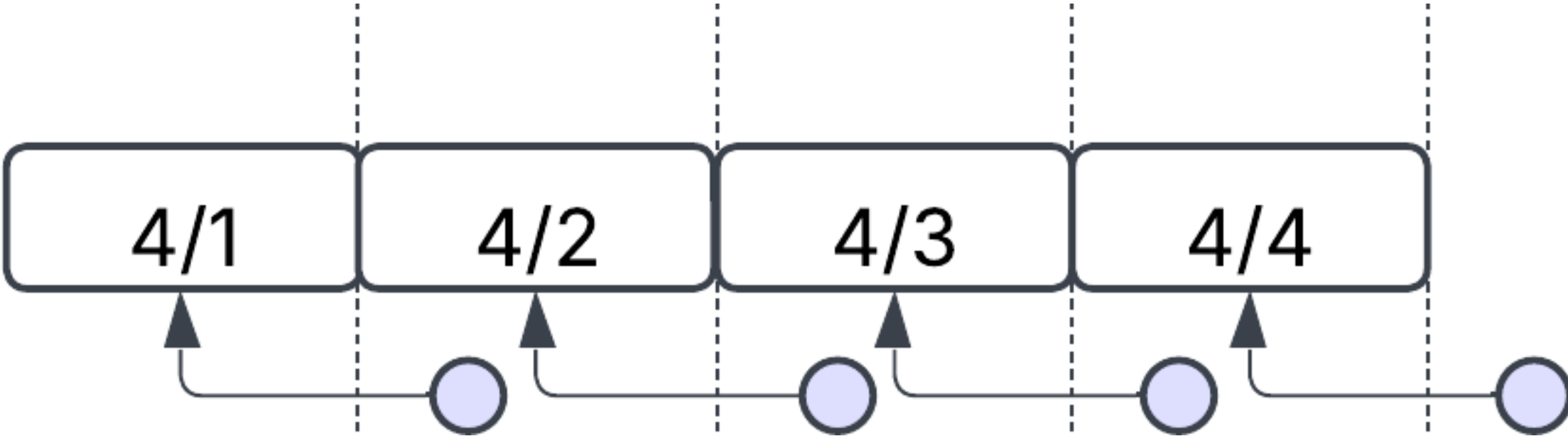
# ETL with APIs

The `NASAResource` gives us the ability to extract data, but it leaves us in a slightly more ambiguous position than when we were importing static files. With files, there's a clear boundary. One file ends, another begins. When extracting data from an API, those boundaries aren't always as obvious, making it less clear how to structure our assets.

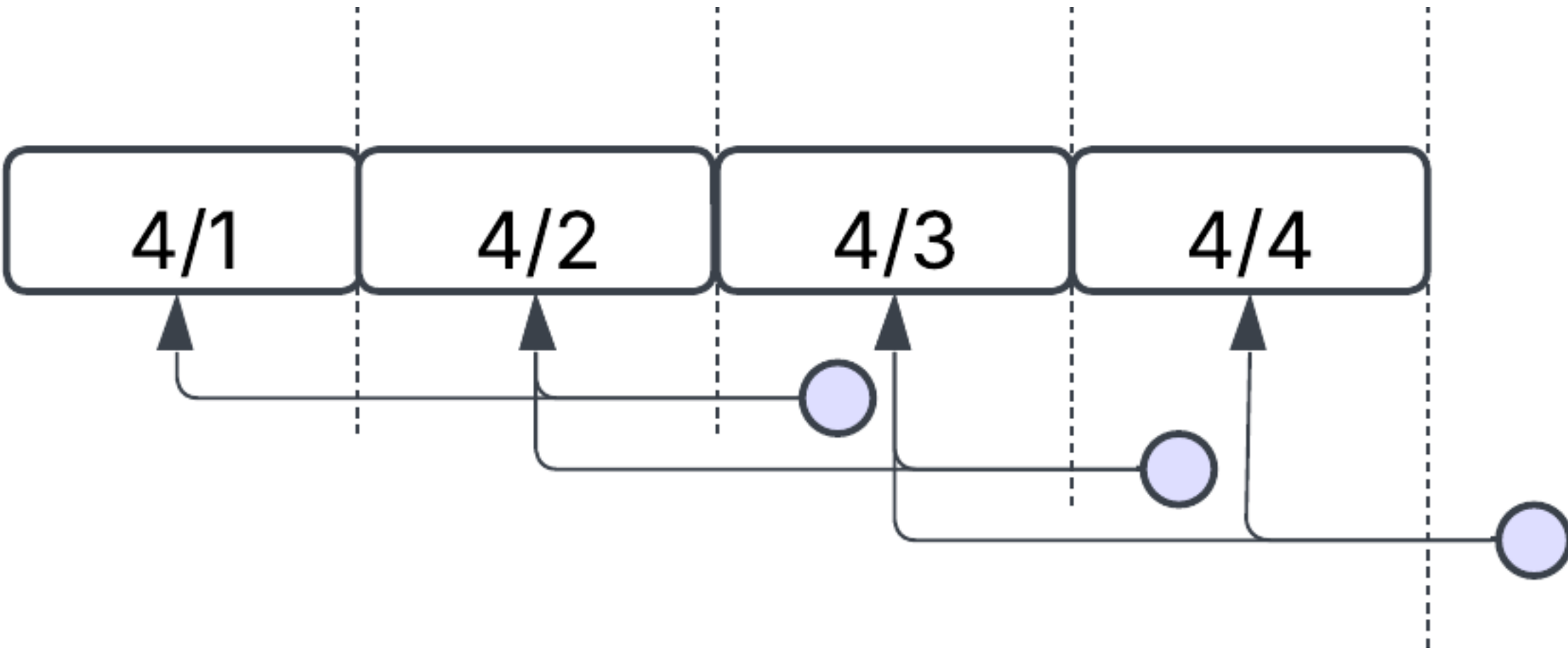
## Time bounding

With the NASA API and our resource, we need to provide a date range when pulling data. Some APIs allow filtering down to the minute or even the second, but the NASA API works at the day level. This makes it easy to structure our ingestion around pulling a full day's worth of data at a time.

But we do have the question of when should we run the process? If we try to fetch today's data, we can't be sure that all of it has been published yet. The safest approach is to query the API for the previous day's data by using a rolling, non-overlapping window.

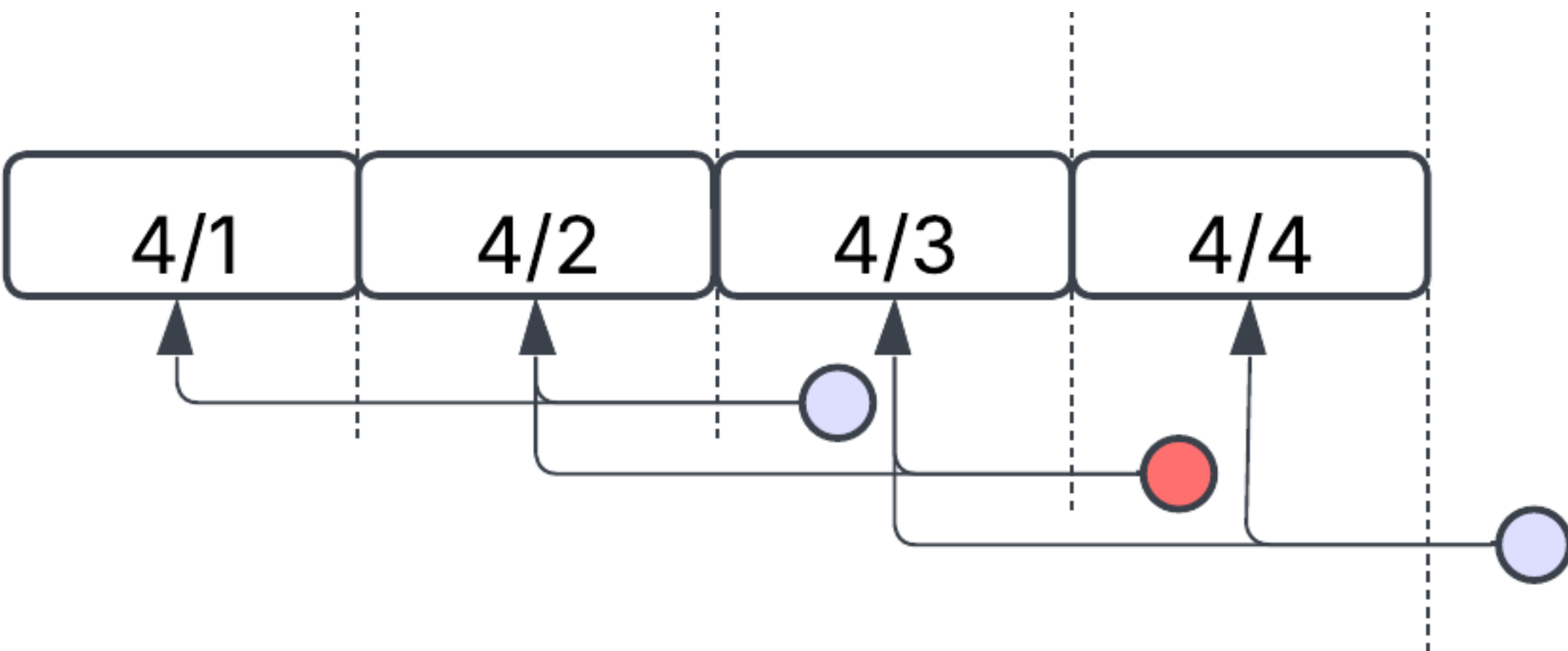


Alternatively, it's possible to implement a rolling window with overlaps:



This can be useful in two key scenarios:

- When the API allows for data corrections or updates, such as changes to past orders or late-arriving data.
- To add resiliency in the event of failures. If one run fails, the next overlapping window might successfully capture the missed data.



Rolling windows with overlaps do introduce added complexity. Since overlapping periods will process the same data more than once, this needs to be handled at the destination layer. This can be done using:

- Hard deletes, where old overlapping data is removed before new data is loaded.
- Soft deletes, where older records are marked as outdated and superseded by the newer ones.

For the purposes of this course, we'll stick to rolling windows without overlaps, which provide a simpler and more deterministic ingestion model.

# API Dagster assets

Now that we have a better understanding of how we want to structure data extraction from the API, we can start laying out our Dagster code. Since we want the pipeline to be runnable for any given date, a good first step is to create a run configuration (as we did in the previous lesson) that allows us to specify the target date at execution time. This provides the flexibility to manually trigger runs, automate daily ingestion, or even backfill historical data when needed:

```
import dagster as dg

class NasaDate(dg.Config):
    date: str
```

Since Dagster run configurations are built on top of Pydantic models, we can use a `field_validator` to ensure the provided date string is in the correct format (e.g., `YYYY-MM-DD` ). This adds a layer of validation that helps catch issues early, before the pipeline begins executing:

```
# src/dagster_and_etl/defs/assets.py
import datetime
from pydantic import field_validator

class NasaDate(dg.Config):
    date: str

    @field_validator("date")
    @classmethod
    def validate_date_format(cls, v):
        try:
            datetime.datetime.strptime(v, "%Y-%m-%d")
        except ValueError:
            raise ValueError("event_date must be in 'YYYY-MM-DD' format")
        return v
```

## Dagster asset

With our run configuration in place, we can now define our Dagster assets. Our first asset will be called `asteroids` , and it will use the `NASAResource` we defined earlier to extract asteroid data for a specific day. In this case, the previous day based on the date provided in the run configuration.

To make the correct API call, we'll need to compute both the `start_date` and `end_date` using that input date. The asset will return a list of dictionaries, each representing an asteroid observed during that time period:

```
# src/dagster_and_etl/defs/assets.py
from dagster_and_etl.defs.resources import NASAResource

@dg.asset(
    kinds={"nasa"},
)
def asteroids(
    context: dg.AssetExecutionContext,
    config: NasaDate,
    nasa: NASAResource,
) -> list[dict]:
    anchor_date = datetime.datetime.strptime(config.date, "%Y-%m-%d")
    start_date = (anchor_date - datetime.timedelta(days=1)).strftime("%Y-%m-%d")

    return nasa.get_near_earth_asteroids(
        start_date=start_date,
        end_date=config.date,
    )
```

Now that we've fetched the data from the API, the next step is getting that data into DuckDB. One approach would be to run an `INSERT` statement for each record, but that pattern isn't ideal for OLAP databases like DuckDB, which are optimized for bulk inserts rather than row-by-row operations.

A more efficient method, and one that aligns with what we did in the previous lesson, is to write the data to a file and then load that file into DuckDB using a `COPY` statement or a similar bulk load mechanism.

So, our next asset will take the list of dictionaries returned by the API and save it to a flat file (e.g., a CSV). To simplify the output and avoid dealing with deeply nested JSON structures, we'll narrow our focus to just the following fields:

- id
- name
- absolute\_magnitude\_h
- is\_potentially\_hazardous\_asteroid

```
# src/dagster_and_etl/defs/assets.py
@dg.asset
def asteroids_file(
    context: dg.AssetExecutionContext,
    asteroids,
) -> Path:
    filename = "asteroid_staging"
    file_path = (
        Path(__file__).absolute().parent / f"../../data/staging/{filename}.csv"
    )

    # Only load specific fields
    fields = [
        "id",
        "name",
        "absolute_magnitude_h",
        "is_potentially_hazardous_asteroid",
    ]

    with open(file_path, mode="w", newline="", encoding="utf-8") as file:
        writer = csv.DictWriter(file, fieldnames=fields)

        writer.writeheader()
        writer.writerows(
            {key: row[key] for key in fields if key in row} for row in asteroids
        )

    return file_path
```

The final asset will look very similar to the one we built in the previous lesson. We'll create a table in DuckDB with the four selected fields and use a `COPY` statement to load the data from the file we just wrote. This approach takes advantage of DuckDB's efficient bulk loading capabilities and keeps the asset clean and performant. By reusing the same pattern, we maintain consistency across our ETL pipelines, whether the source is a local file or an external API:

```
# src/dagster_and_etl/defs/assets.py
@dg.asset(
    kinds={"duckdb"},
)
def duckdb_table(
    context: dg.AssetExecutionContext,
    database: DuckDBResource,
    asteroids_file,
) -> None:
    table_name = "raw_asteroid_data"
    with database.get_connection() as conn:
        table_query = f"""
            create table if not exists {table_name} (
                id varchar(10),
                name varchar(100),
                absolute_magnitude_h float,
                is_potentially_hazardous_asteroid boolean
            )
        """
        conn.execute(table_query)
        conn.execute(f"copy {table_name} from '{asteroids_file}'")
```

We can now execute the pipeline for a specific date.

You completed Knowledge check  
Your score

100%

RETAKE QUIZ

CONTINUE →

You answered 1 out of 1 questions correctly

1. What type of window are we using for our NASA API example?



---

# Triggering API jobs

This development is following the same pattern as the previous lesson, we've built a pipeline that can be run ad hoc to load data into DuckDB. However, in many cases, we'll want to automate this process to keep data up to date without manual intervention.

Since everything we've built so far is time-based and revolves around daily extracts, using a schedule is the most appropriate way to automate the pipeline.

Let's start by creating a simple job that includes the three assets we've defined in this pipeline. Once the job is in place, we'll attach a daily schedule to trigger it automatically:

```
# src/dagster_and_etl/defs/jobs.py
asteroid_job = dg.define_asset_job(
    name="asteroid_job",
    selection=[
        assets.asteroids,
        assets.asteroids_file,
        assets.duckdb_table,
    ],
)
```

We'll use the job we just defined in our schedule. The first question to answer is when the schedule should execute. Since our assets are designed to process data for the previous day, it makes sense to run the schedule early in the morning, once the day has fully passed, let's say around 6:00 AM.

We avoid running the pipeline too close to midnight to ensure the data has had time to fully populate. As with all API-based workflows, you'll want to understand the behavior and latency of your specific source system before finalizing this timing.

This schedule will be similar to those we've written previously, with one key difference: it needs to supply a run configuration to the job. Fortunately, Dagster provides access to the execution time of the schedule, which we can use to dynamically generate the date for the run configuration.

Add the following schedule to the `schedules.py` :

```
# src/dagster_and_etl/defs/schedules.py
@dg.schedule(job=asteroid_job, cron_schedule="0 6 * * *")
def date_range_schedule(context):
    scheduled_date = context.scheduled_execution_time.strftime("%Y-%m-%d")

    return dg.RunRequest(
        run_config={
            "ops": {
                "asteroids": {
                    "config": {
                        "date": scheduled_date,
                    },
                },
            },
        },
    )
```

Now the pipeline is automated and will run every morning at fetch the previous days data.



# Backfilling from APIs

Our ETL pipeline is now set up to ingest new data from the API on an ongoing basis. But what about historical data? The NASA dataset goes back several years, and we may want to load that full history into our data warehouse.

The best strategy for this is to use a Dagster feature we've already worked with: partitions. By partitioning the data by date, we can use Dagster's built-in backfill functionality to launch jobs that process data across specific date ranges. This is a far more elegant and manageable approach than writing custom scripts or triggering a massive, monolithic run that pages through years of API results.

To begin, let's define a daily partition for our asteroid ingestion pipeline. This will allow us to backfill one day at a time and track progress across the full history:

```
nasa_partitions_def = dg.DailyPartitionsDefinition(  
    start_date="2025-04-01",  
)
```

Next we can update the `asteroids` asset to use a partition instead of a run configuration. We will create a separate asset for this.

```
# src/dagster_and_etl/defs/assets.py  
@dg.asset(  
    kinds={"nasa"},  
    partitions_def=nasa_partitions_def,  
)  
def asteroids_partition(  
    context: dg.AssetExecutionContext,  
    nasa: NASAResource,  
) -> list[dict]:  
    anchor_date = datetime.datetime.strptime(context.partition_key, "%Y-%m-%d")  
    start_date = (anchor_date - datetime.timedelta(days=1)).strftime("%Y-%m-%d")  
  
    return nasa.get_near_earth_asteroids(  
        start_date=start_date,  
        end_date=context.partition_key,  
    )
```

All of the downstream assets we've already built can remain unchanged, so for this example, we'll focus on a single partitioned asset to demonstrate how backfilling works with partitions.

## Rate limits

Another reason partitions are well-suited for API-based backfills is because of rate limiting. APIs, especially those that return large volumes of data, often enforce limits to prevent excessive load on their systems.

While partitions don't eliminate rate limits, they make it much easier to track progress and recover gracefully. Since each partition represents a discrete time window (like a single day), if a request fails due to a rate limit, only that partition is affected. This avoids losing the entire job and allows you to retry just the failed partitions, rather than restarting the entire ingestion process. It also makes it easy to throttle or space out runs as needed.

If we wanted to include rate limiting functionality into our `NASAResource` what might it look like?

```
# src/dagster_and_etl/defs/resources.py  
import dagster as dg  
import requests  
from requests.adapters import HTTPAdapter  
from urllib3.util.retry import Retry  
  
class NASAResource(dg.ConfigurableResource):  
    api_key: str  
  
    def get_near_earth_asteroids(self, start_date: str, end_date: str):  
        url = "https://api.nasa.gov/neo/rest/v1/feed"  
        params = {  
            "start_date": start_date,  
            "end_date": end_date,  
            "api_key": self.api_key,  
        }  
  
        # Retries  
        session = requests.Session()  
        retries = Retry(  
            total=5,  
            backoff_factor=0.5,  
            status_forcelist=[500, 502, 503, 504],  
            allowed_methods=["GET"]  
        )  
        adapter = HTTPAdapter(max_retries=retries)  
        session.mount("https://", adapter)  
  
        resp = session.get(url, params=params)  
        return resp.json()["near_earth_objects"][start_date]
```



# Overview

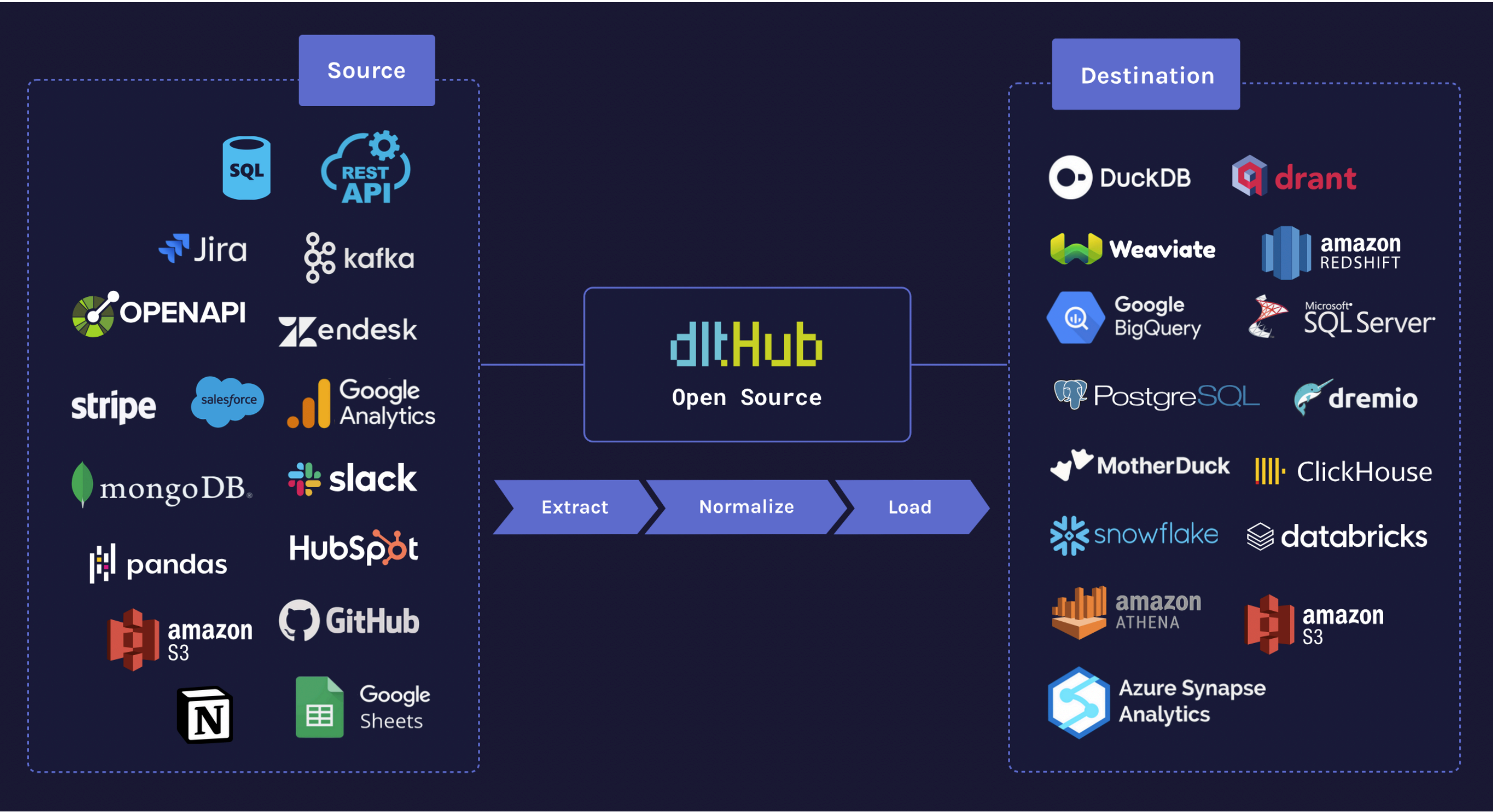
The pipelines we've built so far are great for learning the foundations of ETL pipeline construction and exploring how to apply various Dagster features. However, these aren't necessarily the kinds of pipelines you'd want to deploy directly to production. The main issue is that we're reinventing too many parts of the ETL process, developing logic and handling edge cases that could be handled more efficiently with existing tools.

The good news is that ETL is a universal problem, one that nearly every data-driven organization faces. As a result, a wide range of tools and frameworks have emerged to simplify and standardize ETL workflows — from data extraction and transformation to orchestration and monitoring. Leveraging these tools not only reduces boilerplate and bugs, but also allows you to focus on the parts of the pipeline that are unique to your business logic.

# dlt

One ETL framework we’re particularly excited about is [data load tool \(dlt\)](#). dlt is an open-source, lightweight Python library designed to simplify data loading. It takes care of many of the more tedious aspects of ETL — including schema management, data type handling, and normalization — so you can focus on what matters most.

dlt supports a wide range of popular sources and destinations, which means you can move data between systems without having to build and maintain all the supporting infrastructure yourself. While it still gives you the flexibility to handle custom or complex data workflows, it eliminates much of the boilerplate code you'd otherwise need to write — making your pipelines cleaner, more maintainable, and faster to develop.



Let's look at a simple dlt example.

# Basic dlt

When working with dlt, you should think in terms of two main components: your source and your destination. In our case, the source will be a simple list of dictionaries defined in our code, and the destination will be the same DuckDB database we've been using throughout this course.

This setup allows us to explore the basics of how a dlt pipeline works without adding complexity. Once you're comfortable with the mechanics, you can easily scale up to dynamic sources like APIs or cloud storage.

```
import os

import dlt

data = [
    {"id": 1, "name": "Alice"},
    {"id": 2, "name": "Bob"},
]

@dlt.source
def simple_source():
    @dlt.resource
    def load_dict():
        yield data

    return load_dict

pipeline = dlt.pipeline(
    pipeline_name="simple_pipeline",
    destination=dlt.destinations.duckdb(os.getenv("DUCKDB_DATABASE")),
    dataset_name="mydata",
)

load_info = pipeline.run(simple_source())
```

The code above does the following:

1. Creates a list containing two dicts called `data` .
2. Uses the `dlt.source` decorator to define a source function, inside of this source is a `dlt.resource` decorated function that yields the list defined above.
3. Creates a pipeline using `dlt.pipeline()` that sets the pipeline name, destination (DuckDB) and name of the dataset as it will appear in DuckDB.
4. Executes the pipeline with `pipeline.run` .

We can execute this code by running the file:

```
python dlt_quick_start.py
```

Since dlt is a pure Python framework, there are no additional services or heavy dependencies required, it runs natively in your Python environment.

## dlt Benefits

What stands out most about the dlt approach is how much more ergonomic and streamlined it is compared to the code we previously wrote by hand. Recall that when working directly with DuckDB, we had to manually manage several tedious steps:

- Stage the data by writing it to a CSV file.
- Define the target table and schema in DuckDB ahead of time.
- Load the data using a COPY statement.

With dlt, all of these responsibilities are abstracted away. Once you define your destination, dlt takes care of the rest. Its configuration based approach to REST APIs minimizes the amount of code required, reducing it to just the configuration object.

In addition to simplifying setup, dlt handles a number of complex API behaviors out of the box, including:

- Pagination
- Chained/multi-step API requests
- Other common API reliability issues

dlt also takes care of:

- Schema and table creation
- Data type inference
- Loading the data into your destination

This dramatically reduces boilerplate code and makes your ETL pipelines more maintainable, reliable, and adaptable.

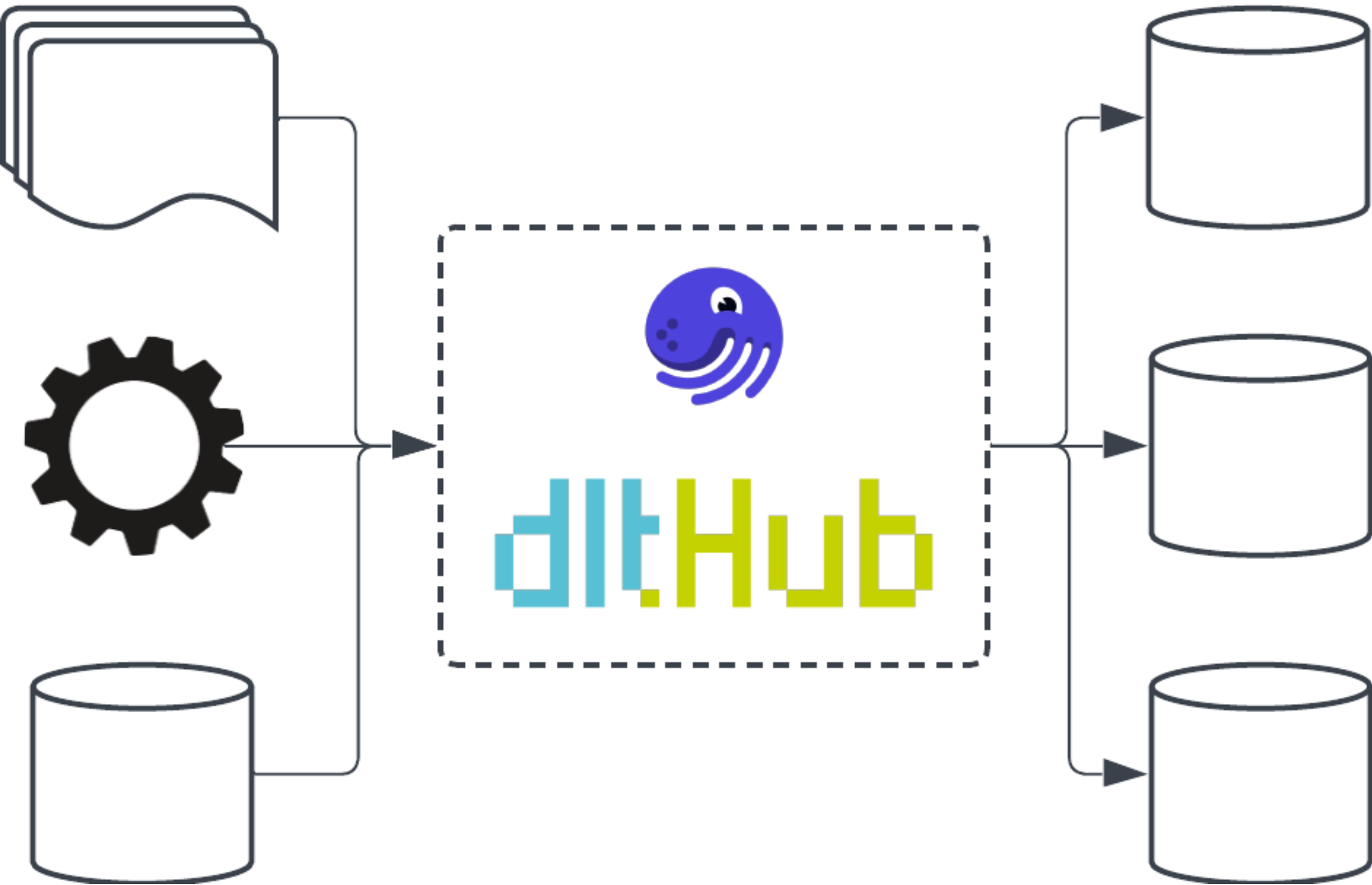
Finally, while we're building a custom API integration here, it's worth noting that dlt also provides out-of-the-box support for [many common verified sources](#).



# Dagster and dlt

Dagster is built to integrate seamlessly with the best tools in the modern data ecosystem — using each tool where it makes the most sense within your pipeline. That’s why we created specialized integrations for libraries like dlt. This feature is called Embedded ETL, and it’s designed to combine the simplicity and flexibility of lightweight ETL frameworks with the robustness and orchestration capabilities of Dagster.

These lightweight services pair well with Dagster because they can leverage Dagster’s existing architecture. Many standalone ETL tools require setting up separate infrastructure, like databases and task queues, to handle state tracking and orchestration. With Dagster, those responsibilities are already built-in. Instead of duplicating that architecture, tools like dlt can integrate directly and let Dagster manage scheduling and focus on what it does best around orchestration and observability.



## dlt assets

To see this in action, we’ll take the dlt quickstart pipeline we just created and convert it into a set of Dagster assets.

We’ll continue using the `simple_source` function decorated with `@dlt.source`. The only change is that we’ll now inject the `data` list directly within the function, so it behaves as a self-contained data source inside the pipeline:

```
@dlt.source
def simple_source():
    @dlt.resource
    def load_dict():
        data = [
            {"id": 1, "name": "Alice"},
            {"id": 2, "name": "Bob"},
        ]

        yield data

    return load_dict
```

Next, we’ll update our `resources.py` file to include the `DagsterDltResource`. This resource allows Dagster to execute dlt pipelines directly, bridging the gap between Dagster’s orchestration layer and dlt’s data loading capabilities. With this integration in place, we no longer need to define a separate `DuckDBResource` in Dagster — dlt will now be fully responsible for loading data into the database:

```
# src/dagster_and_etl/defs/definitions.py
import dagster as dg
from dagster_dlt import DagsterDltResource

@dg.definitions
def resources():
    return dg.Definitions(
        resources={
            "dlt": DagsterDltResource(),
        },
    )
```

Back in `assets.py`, we can define our dlt assets. This will resemble the `dlt.pipeline` code we used earlier, but much of the logic will now be embedded in the `@dlt_assets` decorator. The function itself simply yields the result of running the `dlt pipeline`:

```
# src/dagster_and_etl/defs/assets.py
@dlt_assets(
    dlt_source=simple_source(),
    dlt_pipeline=dlt.pipeline(
        pipeline_name="simple_pipeline",
        dataset_name="simple",
        destination="duckdb",
        progress="log",
    ),
)
def dlt_assets(context: dg.AssetExecutionContext, dlt: DagsterDltResource):
    yield from dlt.run(context=context)
```

Just like that, the same dlt pipeline we previously built can now be executed and tracked by Dagster! This integration doesn't just improve maintainability, it also cleans up our code and reduces boilerplate, making our ETL pipelines more modular and production-ready.

This works well for our simple example, but the real power comes next, we’ll refactor our earlier CSV and API pipelines to use dlt for extraction and loading, while continuing to orchestrate and monitor them through Dagster and represent each dlt resource as an asset.

You completed Knowledge check  
Your score

100%

RETAKE QUIZ

CONTINUE →

You answered 1 out of 1 questions correctly

1. What are some benefits from using an ETL framework compared to writing everything yourself?



---



# Refactoring static data with dlt

There are a couple of key differences between our original CSV pipeline and the dlt quickstart we just converted into Dagster assets. First, the CSV pipeline uses a run configuration to allow a specific file name to be passed in at runtime. Second, the dlt asset in this case is dependent on an upstream asset that stages the file.

We'll start by reusing the existing run configuration and upstream asset, both of which can remain unchanged. This allows us to maintain the same flexible file-based interface while swapping in dlt to handle the data loading:

```
# src/dagster_and_etl/defs/assets.py
class FilePath(dg.Config):
    path: str

@dg.asset
def import_file(context: dg.AssetExecutionContext, config: FilePath) -> str:
    file_path = (
        Path(__file__).absolute().parent / f"../../data/source/{config.path}"
    )
    return str(file_path.resolve())
```

Next, we need to define the `dlt.source` function for our CSV pipeline. This will look very similar to the `simple_source` function we used earlier, but with a few changes: it will read from a CSV file and take a file path as an input parameter. This allows us to dynamically pass in the file we want to process, while letting dlt handle the parsing and schema inference:

```
@dlt.source
def csv_source(file_path: str = None):
    def load_csv():
        with open(file_path, mode="r", encoding="utf-8") as file:
            reader = csv.DictReader(file)
            data = [row for row in reader]

        yield data

    return load_csv
```

This is very helpful. Before our pipelines had to be hardcoded to define the schema for the destination table. Offloading your schema management to a framework can make your pipelines much less error prone. Though it is still good to be aware of what data types it has selected. Just because a zip code looks like an integer does not mean that is how we want the data represented.

So far, things are looking very similar to our previous setup. We'll once again use the `@dlt_assets` decorator to generate our dlt-backed assets. The one key difference in this case is that our `csv_source` function accepts an input parameter for the file path.

Just like in our earlier Dagster pipeline, this parameter will be satisfied by the upstream `import_file` asset. This allows the dlt asset to consume the file path dynamically and run the ETL process based on the specific file returned by `import_file`:

```
@dlt_assets(
    dlt_source=csv_source(),
    dlt_pipeline=dlt.pipeline(
        pipeline_name="csv_pipeline",
        dataset_name="csv_data",
        destination="duckdb",
        progress="log",
    ),
)
def dlt_csv_assets(
    context: dg.AssetExecutionContext, dlt: DagsterDltResource, import_file
):
    yield from dlt.run(context=context, dlt_source=csv_source(import_file))
```

## dlt Translator

This setup might look complete, but there's one final step we need to take. Because dlt assets are generated differently, they also have a unique way of handling dependencies.

If you've taken the [Dagster & dbt course](#), you may recall the need for a Translator to map dbt assets to other assets in your Dagster project. Similarly, for dlt, we use a specialized translator — the `DagsterDltTranslator` — to accomplish the same thing.

In this case, we want to map our dlt asset to depend on the `import_file` asset, so that the file path returned by `import_file` is passed into the dlt source function. Fortunately, this is straightforward to implement. Our translator will look like this:

```
class CustomDagsterDltTranslator(DagsterDltTranslator):
    def get_asset_spec(self, data: DltResourceTranslatorData) -> dg.AssetSpec:
        default_spec = super().get_asset_spec(data)
        return default_spec.replace_attributes(
            deps=[dg.AssetKey("import_file")],
        )
```

With the translator set. We can include it within the `dlt_assets` decorator for the `dagster_dlt_translator` parameter:

```
@dlt_assets(
    ...
    dagster_dlt_translator=CustomDagsterDltTranslator(),
)
```

# Refactoring APIs with dlt

Next we can reconfigure our API pipeline with dlt. This is a much more custom implementation than loading data from a CSV so the `@dlt.source` will contain more code. However, we can simply reuse most of the logic from the previous lesson.

```
# src/dagster_and_etl/defs/assets.py
@dlt.source
def nasa_neo_source(start_date: str, end_date: str, api_key: str):
    @dlt.resource
    def fetch_neo_data():
        url = "https://api.nasa.gov/neo/rest/v1/feed"
        params = {
            "start_date": start_date,
            "end_date": end_date,
            "api_key": api_key,
        }

        response = requests.get(url, params=params)
        response.raise_for_status()

        data = response.json()

        for neo in data["near_earth_objects"][start_date]:
            neo_data = {
                "id": neo["id"],
                "name": neo["name"],
                "absolute_magnitude_h": neo["absolute_magnitude_h"],
                "is_potentially_hazardous": neo["is_potentially_hazardous_asteroid"],
            }

            yield neo_data

    return fetch_neo_data
```

This gives us the ability to pull in any date range from the NASA api using dlt. Rather than using the `dlt_assets` decorator. We can also nest this code directly in a dg asset. We can then update the `nasa_neo_source` function to use the values from the run configuration.

```
# src/dagster_and_etl/defs/assets.py
@dg.asset
def dlt_nasa(context: dg.AssetExecutionContext, config: NasaDate):
    anchor_date = datetime.datetime.strptime(config.date, "%Y-%m-%d")
    start_date = (anchor_date - datetime.timedelta(days=1)).strftime("%Y-%m-%d")

    @dlt.source
    def nasa_neo_source():
        @dlt.resource
        def load_neo_data():
            url = "https://api.nasa.gov/neo/rest/v1/feed"
            params = {
                "start_date": start_date,
                "end_date": config.date,
                "api_key": os.getenv("NASA_API_KEY"),
            }

            response = requests.get(url, params=params)
            response.raise_for_status()

            data = response.json()

            for neo in data["near_earth_objects"][config.date]:
                neo_data = {
                    "id": neo["id"],
                    "name": neo["name"],
                    "absolute_magnitude_h": neo["absolute_magnitude_h"],
                    "is_potentially_hazardous": neo[
                        "is_potentially_hazardous_asteroid"
                    ],
                }

                yield neo_data

            return load_neo_data

        pipeline = dlt.pipeline(
            pipeline_name="nasa_neo_pipeline",
            destination=dlt.destinations.duckdb(os.getenv("DUCKDB_DATABASE")),
            dataset_name="nasa_neo",
        )

        load_info = pipeline.run(nasa_neo_source())

    return load_info
```

Writing the function this way also makes it easy to include partitions as we would for any other asset.

```
# src/dagster_and_etl/defs/assets.py
@dg.asset(
    partitions_def=nasa_partitions_def,
    automation_condition=dg.AutomationCondition.on_cron("@daily"),
)
def dlt_nasa_partition(context: dg.AssetExecutionContext):
    anchor_date = datetime.datetime.strptime(context.partition_key, "%Y-%m-%d")
    start_date = (anchor_date - datetime.timedelta(days=1)).strftime("%Y-%m-%d")

    @dlt.source
    def nasa_neo_source():
        @dlt.resource
        def load_neo_data():
            url = "https://api.nasa.gov/neo/rest/v1/feed"
            params = {
                "start_date": start_date,
                "end_date": context.partition_key,
                "api_key": os.getenv("NASA_API_KEY"),
            }

            response = requests.get(url, params=params)
            response.raise_for_status()

            data = response.json()

            for neo in data["near_earth_objects"][context.partition_key]:
                neo_data = {
                    "id": neo["id"],
                    "name": neo["name"],
                    "absolute_magnitude_h": neo["absolute_magnitude_h"],
                    "is_potentially_hazardous": neo[
                        "is_potentially_hazardous_asteroid"
                    ],
                }

                yield neo_data

            return load_neo_data

        pipeline = dlt.pipeline(
            pipeline_name="nasa_neo_pipeline",
            destination=dlt.destinations.duckdb(os.getenv("DUCKDB_DATABASE")),
            dataset_name="nasa_neo",
        )

        load_info = pipeline.run(nasa_neo_source())

    return load_info
```

# Overview

We've explored several ways to ingest data from external sources, but one approach we haven't covered yet is moving data between databases. This is a broad category, as databases vary widely in type and behavior. However, the most common scenario and one that many people associate with traditional ETL, is replicating data from an OLTP database (like Postgres or MySQL) into a data warehouse (such as Snowflake or Redshift).

Although this flow is extremely common — and something nearly every data-driven organization performs to some extent — it's also a nuanced and potentially error-prone process. From schema drift and type mismatches to performance bottlenecks and data consistency challenges, moving data between systems requires careful handling and thoughtful architecture.



# Database replication

Despite being a very common workflow, database replication is nuanced and full of potential pitfalls.

## Understanding your source

Just like with APIs, replicating data from a database starts with understanding the system you're pulling from. Databases vary widely in structure, capabilities, and query patterns.

Database Type	Query	Examples
Relational	SQL	Postgres, MySQL, Oracle, SQLite
NoSQL	Varies	MongoDB, Redis, DynamoDB
Graph	Neo4j, Gremlin	Neo4j, Neptune, ArangoDB
Vector	Semantic	FAISS, Pinecone, Weaviate
Time-series	SQL-like	InfluxDB, Prometheus

Since it's too much to cover all of these in a single course, we'll focus on relational databases, specifically Postgres (it is also the most common form of replication). And even limiting it to just that, there are multiple strategies on how to sync data.

## Full refresh replication

Relational databases store data in tables. Imagine a customers table with the following data:

customer_id	first_name	last_name	email	created_at
1	Alice	Johnson	alice.johnson@example.com	2024-05-01 10:15:00
2	Bob	Smith	bob.smith@example.com	2024-05-02 08:42:00
3	Charlie	Lee	charlie.lee@example.com	2024-05-03 13:30:00
4	Dana	Martinez	dana.martinez@example.com	2024-05-04 09:50:00
5	Evan	Thompson	evan.thompson@example.com	2024-05-05 11:22:00

The most straightforward way to extract this data is to query the entire contents of the table:

```
SELECT * FROM customers;
```

This method is known as full refresh or snapshot replication. It's simple and works with just about any database (as long as the database is not append only) that supports `SELECT` queries. But as you can imagine, there's a major drawback: how do we keep the source and destination in sync?

If we rely solely on full refreshes, we have to run the entire extraction process every time. This can be prohibitively expensive for large tables and can strain the source database, impacting performance during the sync.

## Incremental replication

You can optimize full refreshes by filtering the data. Usually this involves querying based on time columns or incrementing ids. The checkpoint of the last query is then maintained by the ETL service so it knows where begin replication.

This helps limit the strain on ingestion because much less data needs to be queried and transferred. To perform incremental replication with our customers table, we could add in the created\_at column to only pull new records:

```
CREATE OR REPLACE ...
SELECT * FROM customers WHERE created_at > '2024-05-01';
```

This is helpful though adds some complexity and raises some additional questions:

- What happens if a row is updated, not just newly created?
- How do you track deletions?
- What if a row is delayed in appearing in the database?
- What if there's no timestamp column to filter on?

Most ETL services that do incremental replication can address these issues though they still require a table schema that has some way of tracking new records. If the customers table did not include the created\_at column, it would be much harder to perform incremental updates.

## Change data capture (CDC)

The final strategy we will cover is Change Data Capture (CDC). Relational databases like Postgres maintain a log of changes, in Postgres, this is the Write-Ahead Log (WAL), which records inserts, updates, and deletes for recovery and replication purposes.

CDC is more efficient and much less taxing on the source database. But it comes with trade-offs:

- CDC logs are not retained forever, typically only for a few days. This means that brings in new records must complete before those records are removed.
- CDC doesn't provide full historical context, so it can't be used alone to initialize a replica.

You need to apply the log events in order to reconstruct the current state of the data.

Here's what a stream of CDC events might look like:

Event Type	Timestamp	customer_id	first_name	last_name	email	created_at
INSERT	2024-05-01 10:00:00	101	Alice	Johnson	alice.johnson@example.com	2024-05-01 10:00:00
UPDATE	2024-05-02 14:30:00	101	Alicia	Johnson	alice.johnson@example.com	2024-05-01 10:00:00
UPDATE	2024-05-03 09:10:00	101	Alicia	Thompson	alicia.thompson@example.com	2024-05-01 10:00:00

## Building database replication systems

Most modern ETL tools that handle database replication use one or more of these approaches. For example a tool may perform a full refresh to establish the initial snapshot of the table and then switch to CDC to capture all changes moving forward.

This combined approach provides both completeness and efficiency but requires careful coordination during the cut over to ensure that no data is lost or duplicated.

If this sounds complex, that's because it is. Replicating data between databases is challenging and full of edge cases, which is why we strongly recommend using a dedicated framework instead of trying to build one from scratch.

You completed Knowledge check  
Your score

100%

RETAKE QUIZ

CONTINUE →

You answered 1 out of 1 questions correctly

1. What replication strategy uses logs, such as the Write-Ahead Log (WAL) in Postgres, to monitor database changes to construct replication?



# Sling

While we can use dlt for database replication, in this lesson we'll introduce another open-source ETL framework: [Sling](#). Sling is a modern tool designed to simplify both real-time and batch data replication. It helps teams move data from databases like Postgres and MySQL into cloud data warehouses such as Snowflake or Redshift with minimal setup. Exposing all configuration through a simple YAML interface.

Sling differs from dlt in that it is declarative by design. While dlt offers greater flexibility and can handle data ingestion from a wide variety of sources (such as our custom NASA API), Sling is purpose-built for database replication and excels at managing the complexities of moving data and schema evolution.



# Sling database replication set up

For this course, we'll keep things lightweight by using a simple Postgres database with a basic schema and a few rows of sample data, just enough to confirm that everything is working correctly.

One of the easiest ways to spin up a temporary database is with [Docker](#). We've provided a pre-configured Docker Compose file that starts a Postgres container and automatically seeds it with data.

To get started, run the following command:

```
docker compose -f dagster_and_etl_tests/docker-compose.yaml up -d
```

**Note:** The first time you run this command, Docker may need to download the required image, so it could take a few minutes.

This Docker Compose will launch a Postgres instance and populate it with a small schema containing a few sample tables and rows, perfect for development and testing.

## data.customers

Column	Type	Description
customer_id	SERIAL	Primary key, auto-incrementing ID
first_name	VARCHAR(100)	Customer's first name
last_name	VARCHAR(100)	Customer's last name
email	VARCHAR(255)	Unique email address

## data.products

Column	Type	Description
product_id	SERIAL	Primary key, auto-incrementing ID
name	VARCHAR(255)	Name of the product
description	TEXT	Product description
price	DECIMAL(10, 2)	Product price with two decimal places

## data.orders

Column	Type	Description
order_id	SERIAL	Primary key, auto-incrementing ID
customer_id	INTEGER	Foreign key referencing data.customers(customer_id)
product_id	INTEGER	Foreign key referencing data.products(product_id)
quantity	INTEGER	Quantity of product ordered, defaults to 1
total_amount	DECIMAL(10, 2)	Total price for the order
order_date	TIMESTAMP	Timestamp of the order, defaults to current time

The specifics of the schema aren't critical for our purposes, we just need some sample data to replicate and, most importantly, the connection details for the Postgres database running in Docker:

Field	Value
Host	localhost
Port	5432
Database	test_db
Username	test_user
Password	test_pass

# Dagster and Sling

When defining assets with Sling, you will first have to define your source and target. In this case our source is Postgres and the destination is still our DuckDB database.

To define the source and destination, we will use the `SlingConnectionResource`. The source POstgres instance will use the connection details from our Docker Compose.

```
# src/dagster_and_etl/defs/resources.py
import dagster as dg
from dagster_sling import SlingConnectionResource, SlingResource

source = SlingConnectionResource(
    name="MY_POSTGRES",
    type="postgres",
    host="localhost",
    port=5432,
    database="test_db",
    user="test_user",
    password="test_pass",
)
```

The destination has fewer attributes as it is just the file path to our local DuckDB database:

```
# src/dagster_and_etl/defs/resources.py
destination = SlingConnectionResource(
    name="MY_DUCKDB",
    type="duckdb",
    connection_string="duckdb:///var/tmp/duckdb.db",
)
```

With the source and destination defined, we can combine them together in the `SlingResource` and set it within the `dg.Definitions`:

```
# src/dagster_and_etl/definitions.py
defs = dg.Definitions(
    resources={
        "sling": sling,
    },
)
```

## Replication yaml

Our connections are set but we still need to set what we are replicating. Sling handles this with a replication yaml file. This determines which connection is our source and target:

```
source: MY_POSTGRES
target: MY_DUCKDB
```

More importantly we need to decide which of our tables to replicate between Postgres and DuckDB. We only need the `data.customers`, `data.products` and `data.orders` tables so can set those in the `streams`.

```
defaults:
  mode: full-refresh

  object: "{stream_schema}_{stream_table}"

streams:
  data.customers:
  data.products:
  data.orders:
```

This will replicate our 3 tables into DuckDB.

## Sling assets

Now we can generate our Dagster assets. Since we have already done most of the work in defining our source and destination in the `SlingResource` and the replication information in the `replication.yaml`, Dagster can automatically generate the assets with the `sling_assets` decorator:

```
# src/dagster_and_etl/defs/assets.py
import dagster as dg
from dagster_sling import SlingResource, sling_assets

replication_config = dg.file_relative_path(__file__, "sling_replication.yaml")

@sling_assets(replication_config=replication_config)
def postgres_sling_assets(context, sling: SlingResource):
    yield from sling.replicate(context=context).fetch_column_metadata()
```

That is everything we need. Now when we launch `dg dev` we will see six new assets in the asset graph.

First there are the three source assets representing the raw tables that exist in our Postgres database (customers, products, orders). Connected to each of these sources are our Dagster assets. Each of these individually.

# Executing the pipeline

We've defined all the assets we need, but we haven't yet discussed how we want to execute this pipeline. When running data replication in production, there are a few important considerations to keep in mind.

## Triggering our job

First, let's consider the best way to trigger our ETL assets. While some ETL tools, like Debezium, support continuous data loading, most are schedule-based.

Choosing the right schedule can be nuanced. Depending on how the data is extracted (which we'll cover later), running schedules too frequently may be inefficient. For example, if you attempt to run a job multiple times per minute, one execution might not finish before the next one starts—this can lead to a backup and degraded performance.

On the other hand, if your schedules are too far apart, you risk missing timely updates and working with stale data. In practice, running your schedules a few times per day, based on data volume and business needs, is often a good balance.

Creating a schedule for our Sling ETL assets is no different than defining any other schedule in Dagster. Suppose we want to run two schedules:

- One that refreshes all Sling assets once a day
- Another that refreshes the orders asset three times a day

We'd define two separate jobs:

```
# src/dagster_and_etl/defs/jobs.py
import dagster as dg

postgres_refresh_job = dg.define_asset_job(
    "postgres_refresh",
    selection=[
        dg.AssetKey(["target", "data", "customers"]),
        dg.AssetKey(["target", "data", "products"]),
        dg.AssetKey(["target", "data", "orders"]),
    ],
)

orders_refresh_job = dg.define_asset_job(
    "orders_refresh",
    selection=[
        dg.AssetKey(["target", "data", "orders"]),
    ],
)
```

Then we can create two distinct schedules with different cron expressions:

```
# src/dagster_and_etl/defs/schedules.py
postgres_refresh_schedule = dg.ScheduleDefinition(
    job=postgres_refresh_job,
    cron_schedule="0 6 * * *",
)

orders_refresh_schedule = dg.ScheduleDefinition(
    job=orders_refresh_job,
    cron_schedule="0 12,18 * * *",
)
```

With these schedules in place, all ETL assets will refresh at 6 UTC, and the orders asset will additionally refresh at 12 and 18 UTC.

## Replication Strategies

We haven't yet discussed how data is replicated with Sling. If you've looked at the `replication.yaml` file, you may have noticed that the default mode is set to `full-refresh` :

```
defaults:
  mode: full-refresh
```

This is the same as the full refresh strategy we discussed about earlier where the entire table is copied over every time. For the small database we're using, this is fine. However, as data volumes grow, you may want to switch to a more efficient replication strategy.

Sling does not offer a full CDC solution but we can still manage replication with fairly large databases by using incremental replication.

Let's think about our schema. Of the three tables, customers, products, and orders, which is most likely to grow the largest? The orders table. This makes sense: a single user may place many orders, and each order can involve multiple products.

Another important characteristic of the orders table is the presence of a time-based column, `order_date`, which tracks when an order was created. We can use this column to filter the records we need to process during each run.

To do this, we can configure the `replication.yaml` file to make orders an incremental asset, using the `order_date` column to track changes:

```
data.orders:
  mode: incremental
  primary_key: order_id
  update_key: order_date
```

With this configuration, Sling will only process new or updated records based on the `order_date`, reducing data load and improving performance.

Incremental replication is just one of [many strategies supported by Sling](#). Regardless of the tool you use, it's important to select a replication approach that fits your data's characteristics and your system's requirements.

# Overview

We've now covered a wide range of ETL use cases, from static files and APIs to database replication. Interestingly, as our pipelines have grown more complex, our code has actually become simpler. That's thanks to the thoughtful abstractions provided by tools like dlt, Sling and Dagster.

The last topic we'll cover is [Dagster Components](#). So far, we've seen how Dagster can work with frameworks like dlt and Sling to build robust pipelines. However, coordinating across frameworks can be difficult. You have to initialize configurations with different CLIs, manage configurations and know how everything connects.

In order to simplify this development process, Dagster unifies this all within components. Their goal is to make developing these types of integrations much easier and reduce the amount of code users are responsible for maintaining.

In this section we will redefine the Sling integration from the previous lesson with components.

# Dagster Components

Dagster Components provide a class-based interface ( **Component** ) for dynamically constructing Dagster definitions from arbitrary data sources, such as third-party integrations. They're designed to simplify orchestration by encapsulating complex logic into reusable, declarative building blocks.

To ground this in a practical example: a component can manage our dlt connection between Postgres and DuckDB. This eliminates the need for low-level setup—like CLI initialization with external libraries—and lets you focus solely on the relevant inputs and outputs for your use case.

This approach is especially effective for common workflows like database replication, which shouldn't require extensive custom development. Instead, users can provide a few configuration values, such as connection credentials or table filters, and let the component handle the orchestration. This lowers the barrier for non-engineers or new users, enabling them to integrate systems in Dagster without needing deep expertise in the framework's internals.

Let's walk through how to configure Postgres-to-DuckDB replication using a Dagster component. You'll see how much simpler and cleaner your setup becomes when you rely on components to manage orchestration and integration behind the scenes.



# Sling with Components

To navigate including Components with Dagster, we will be using the Dagster `dg` command line. `dg` is a useful to initializing and navigating Dagster projects. Because we have been working the virtual envionment and project structure of this course we have not needed `dg` as much.

We used `dg` before to lanuch the UI to view and execute our assets but now we will use it to scaffold our Component.

## Create Sling Component

To initialize our dlt Component, we will run the `dg` command to scaffold the Sling Component (`dagster_sling.SlingReplicationCollectionComponent`):

```
dg scaffold defs 'dagster_sling.SlingReplicationCollectionComponent' ingest_files
```

There will now be a new directory in `defs` containing two files:

```
.
├── src
│   ├── dagster_and_etl
│   │   └── defs
│   │       ├── ingest_files
│   │       │   ├── defs.yaml
│   │       │   └── replication.yaml
```

These are the only two files we will need to interact with in order to configure our Sling assets (no Python). Let's discuss what goes into each.

## Configure the YAML

The `replication.yaml` should look familiar. That is the yaml file that Sling uses to determine the replication characteristics. The `replication.yaml` that is generated by the `dg` command is pretty much empty:

```
source: {}
streams: {}
target: {}
```

Let's copy and paste all of the contents from our `sling_replication.yaml` from the previous lesson:

```
source: MY_POSTGRES
target: MY_DUCKDB

defaults:
  mode: full-refresh

  object: "{stream_schema}_{stream_table}"

streams:
  data.customers:
  data.products:
  data.orders:
    mode: incremental
    primary_key: order_id
    update_key: order_date
```

The next file is the `defs.yaml`. This contains two bits of information. Which component we are using, in this case Sling, and the location of the `replication.yaml`.

```
type: dagster_sling.SlingReplicationCollectionComponent

attributes:
  replications:
    - path: replication.yaml
```

The only thing we need to add are the connection details. Before we had defined these within the `SlingResource` but there is no need for that with Components. Instead we will enter the connection details in the `component.yaml`:

```
type: dagster_sling.SlingReplicationCollectionComponent

attributes:
  sling:
    connections:
      - name: MY_POSTGRES
        type: postgres
        host: localhost
        port: 5432
        database: test_db
        user: test_user
        password: test_pass
      - name: MY_DUCKDB
        type: duckdb
        instance: duckdb:///var/tmp/duckdb.db
    replications:
      - path: replication.yaml
```

Notice that the names of the connection in the `defs.yaml` still need to match up to the `replication.yaml` ( `MY_DUCKDB` and `MY_POSTGRES` ).

?

## Viewing the assets

Because we switching our assets over to components, we can cleanup a lot of our Python code. In `resources.py` you can remove everything related Sling. And in the `assets.py` file you can remove the assets generated by the `sling_assets`. We can still keep the assets downstream of the Component assets.

Now if we run `dg dev` we can see the same asset graph as before.

```
type:
dagster_sling.SlingReplicationCollectionComponent
```

```
attributes:
```

```
  sling:
```

```
    connections:
```

```
      - name: MY_POSTGRES
```

```
        type: postgres
```

```
        host: localhost
```

```
        port: 5432
```

```
        database: test_db
```

```
        user: test_user
```

```
        password: test_pass
```

```
      - name: MY_DUCKDB
```

```
        type: duckdb
```

```
        instance: /tmp/duckdb.db
```

```
replications:
```

```
  - path: replication.yaml
```

```
source = SlingConnectionResource(
    name="MY_POSTGRES",
    type="postgres",
    host="localhost",
    port=5432,
    database="test_db",
    user="test_user",
    password="test_pass",
)

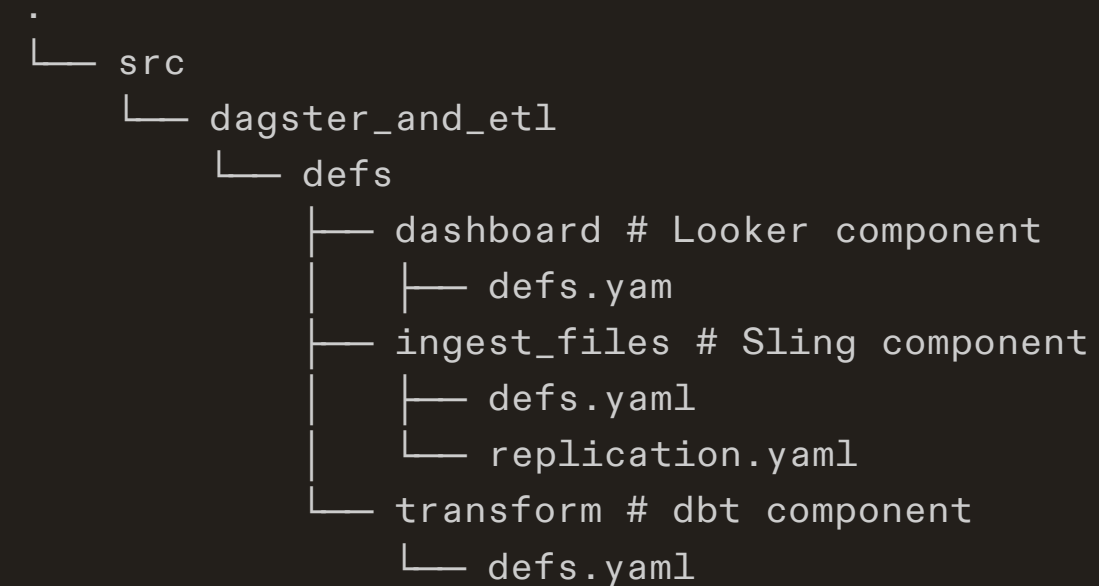
destination = SlingConnectionResource(
    name="MY_DUCKDB",
    type="duckdb",
    connection_string="duckdb:///var/tmp/duckdb.db",
)

sling = SlingResource(
    connections=[
        source,
        destination,
    ]
)

defs = dg.Definitions(
    resources={
        "sling": sling,
    },
)
```

# Using components

When using Dagster Components, one best practice is to treat each component as an isolated, testable, and versioned unit of logic. This means encapsulating not just asset definitions, but their dependencies, such as resources, config schemas, and even environment setup—within the component itself.

A terminal window with a dark background and light gray text. It shows a directory tree structure for a project. The root is '.', followed by 'src', then 'dagster\_and\_etl', and finally 'defs'. Under 'defs', there are three subdirectories: 'dashboard # Looker component', 'ingest\_files # Sling component', and 'transform # dbt component'. Each subdirectory contains a 'defs.yaml' file, and 'ingest\_files' also contains a 'replication.yaml' file. A clipboard icon is visible in the top right corner of the terminal window.

```
.
├── src
│   └── dagster_and_etl
│       └── defs
│           ├── dashboard # Looker component
│           │   └── defs.yaml
│           ├── ingest_files # Sling component
│           │   ├── defs.yaml
│           │   └── replication.yaml
│           └── transform # dbt component
│               └── defs.yaml
```

To ensure maintainability, avoid having components implicitly rely on external context or shared state unless explicitly passed in. It's also important to define clear interfaces. Use typed config models and input/output schemas so that consumers of a component understand how to configure and integrate it without digging into the internals.

## Reuse

Another best practice is to design for composability and reuse, much like you would with modules or packages in traditional software engineering. components should be small enough to be meaningful on their own (e.g., a set of assets to extract data from a specific source), but flexible enough to be combined with others in different DAGs or projects. Use version control and semantic naming to track changes, and maintain compatibility boundaries when updating shared components. When possible, validate components independently through unit tests or local runs before integrating them into larger pipelines, ensuring that failures are caught early and in isolation.

# Organizing our project

Organizing your Dagster project with components brings structure, scalability, and clarity to your data platform. Instead of building one large codebase where assets, resources, and configurations are tightly coupled, components allow you to break your project into self-contained modules. Each component bundles together related logic—such as a data source, transformation, or model training step—along with its resources and config schemas. This modular layout makes it easier to onboard new team members, reuse functionality across pipelines, and iterate on parts of your system without risking unrelated functionality.

A well-organized component-based project typically follows a pattern where each component lives in its own directory or package, complete with its own virtual environment, tests, and documentation. For example, you might have `components/snowflake_ingestion`, `components/ml_training`, and `components/reporting_pipeline`, each representing a logical slice of your platform. This structure encourages encapsulation and reduces dependency sprawl, allowing individual components to evolve at their own pace. By centralizing composition in your `definitions.py` file (or similar), you can declaratively stitch components together to build end-to-end workflows without compromising maintainability. As your team and projects grow, components provide the foundation for a scalable and collaborative development model.