Manually testing a data pipeline

INTRODUCTION TO DATA PIPELINES

Testing data pipelines

Data pipelines should be thoroughly tested

- Validate that data is extracted, transformed, and loaded as expected

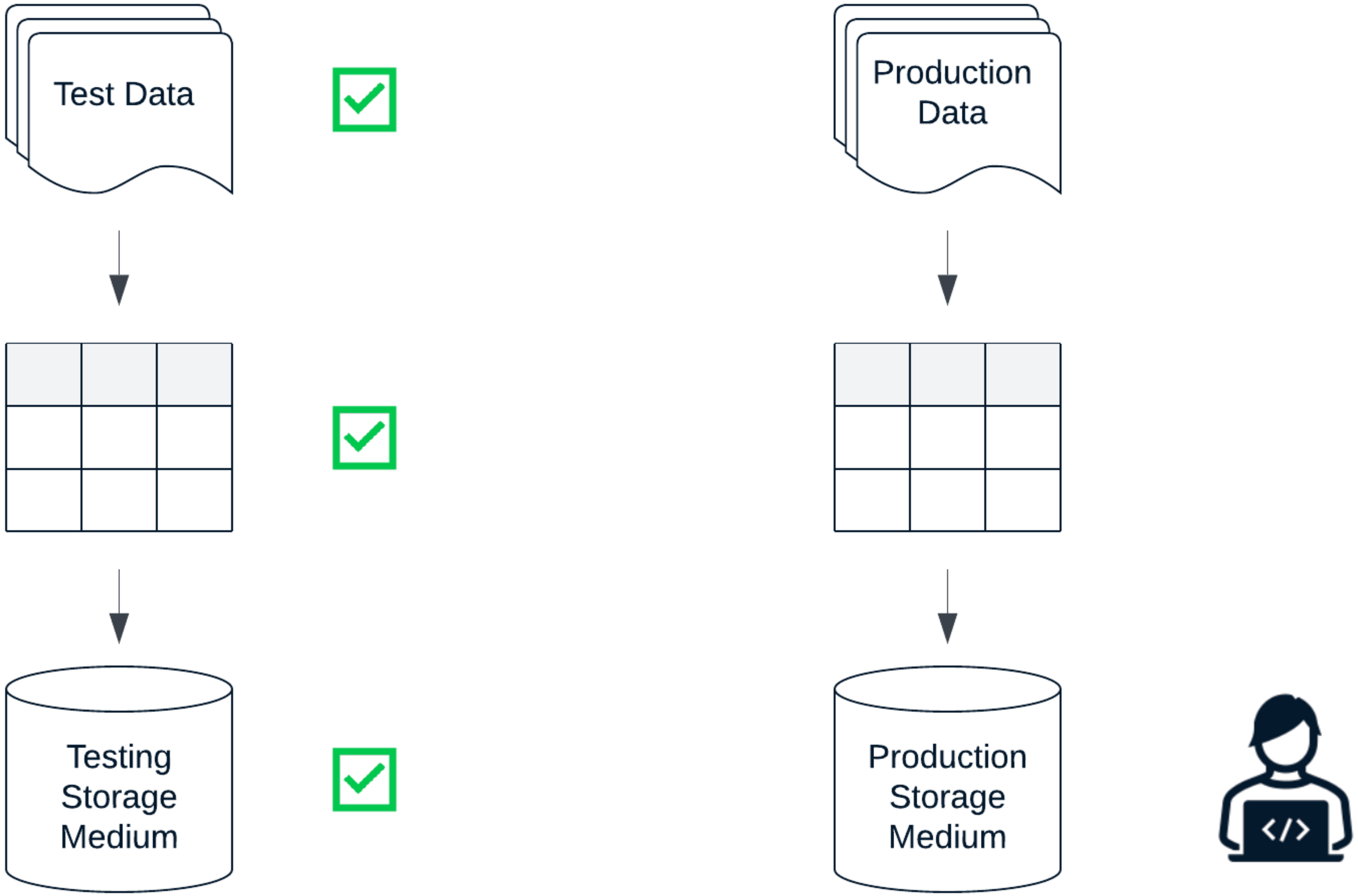
Validating pipelines' limits maintenance efforts after deployment

- Identify and fix data quality issues
- Improves data reliability

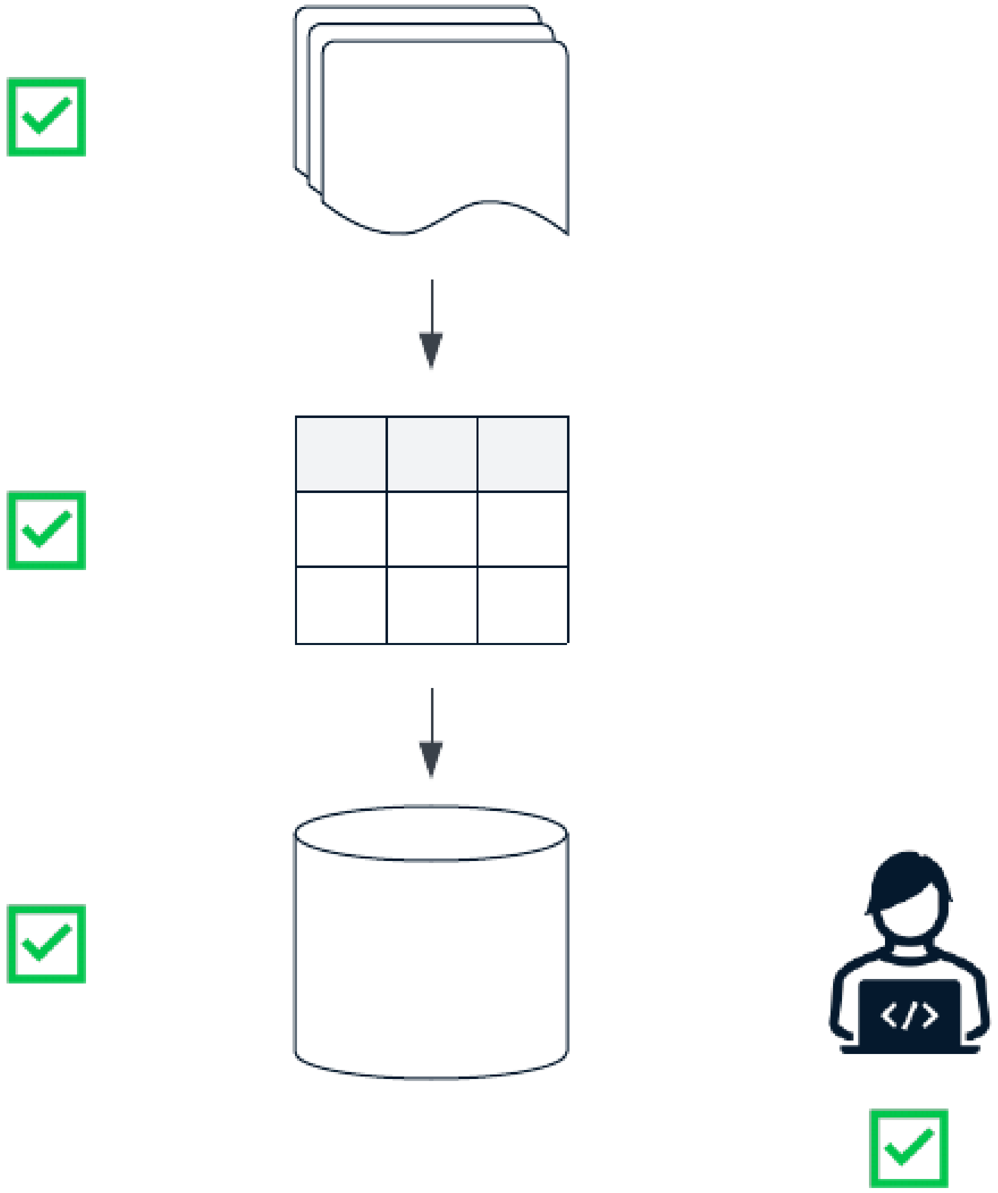
Tools and techniques to test data pipelines

- End-to-end testing
- Validating data at "checkpoints"
- Unit testing

Testing and production environments



Testing a pipeline end-to-end



End-to-end testing

- Confirm that pipeline runs on repeated attempts
- Validate data at pipeline checkpoints
- Engage in peer review, incorporate feedback
- Ensure consumer access and satisfaction with solution

Validating pipeline checkpoints

```
# Extract, transform, and load data as part of a pipeline
...

# Take a look at the data made available in a Postgres database
loaded_data = pd.read_sql("SELECT * FROM clean_stock_data", con=db_engine)
print(loaded data.shape)
```

```
(6438, 4)
```

```
print(loaded_data.head())
```

```
          timestamps      volume      open     close
1997-05-15 13:30:00  1443120000  0.121875  0.097917
1997-05-16 13:30:00   294000000  0.098438  0.086458
1997-05-19 13:30:00   122136000  0.088021  0.085417
```

Validating DataFrames

```
# Extract, transform, and load data, as part of a pipeline
...

# Take a look at the data made available in a Postgres database
loaded_data = pd.read_sql("SELECT * FROM clean_stock_data", con=db_engine)

# Compare the two DataFrames.
print(clean_stock_data.equals(loaded_data))
```

```
True
```

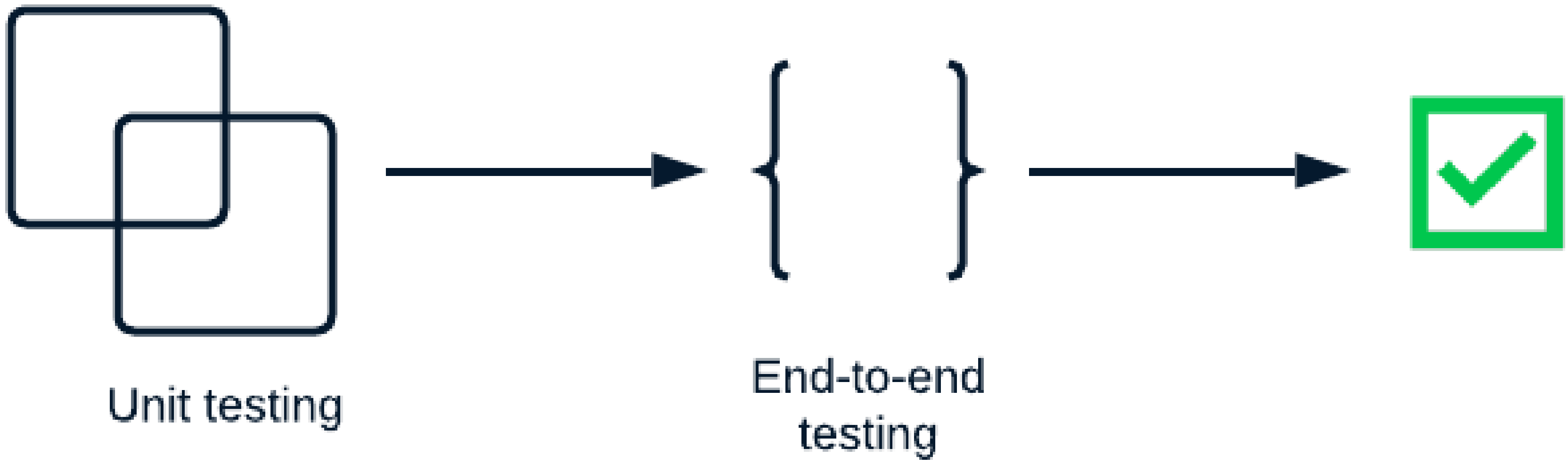
Unit-testing a data pipeline

INTRODUCTION TO DATA PIPELINES

Validating a data pipeline with unit tests

Unit tests:

- Commonly used in software engineering workflows
- Ensure code works as as expected
- Help to validate data



pytest for unit testing

```python
from pipeline import extract, transform, load

# Build a unit test, asserting the type of clean_stock_data
def test_transformed_data():
    raw_stock_data = extract("raw_stock_data.csv")
    clean_stock_data = transform(raw_data)
    assert isinstance(clean_stock_data, pd.DataFrame)
```

```
> python -m pytest

test_transformed_data .                                                  [100%]
================================ 1 passed in 1.17s ================================
```

assert and isinstance

```python
pipeline type = "ETL"


# Check if pipeline_type is an instance of a str
isinstance(pipeline type, str)
```

```
True
```

```python
# Assert that the pipeline does indeed take value "ETL"
assert pipeline_type == "ETL"
```

```python
# Combine assert and isinstance
assert isinstance(pipeline_type, str)
```

AssertionError

```
pipeline type = "ETL"


# Create an AssertionError
assert isinstance(pipeline type, float)
```

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
AssertionError
```

Mocking data pipeline components with fixtures

```
import pytest


@pytest.fixture()
def clean_data():
    raw stock data = extract("raw stock data.csv")
    clean stock data =  transform(raw data)
    return clean stock data
```

```
def test_transformed_data(clean_data):
    assert isinstance(clean_data, pd.DataFrame)
```

Unit testing DataFrames

```python
def test_transformed_data(clean_data):
    # Include other assert statements here
    ...

    # Check number of columns
    assert len(clean_data.columns) == 4

    # Check the lower bound of a column
    assert clean_data["open"].min() >= 0

    # Check the range of a column by chaining statements with "and"
    assert clean_data["open"].min() >= 0 and clean_data["open"].max() <= 1000
```

Running a data pipeline in production

INTRODUCTION TO DATA PIPELINES

Data pipeline architecture patterns

```
# Define ETL function
...
def load(clean_data):
...

# Run the data pipeline
raw_stock_data = extract("raw_stock_data.csv")
clean_stock_data = transform(raw_data)
load(clean_stock_data)
```

```
> ls
 etl_pipeline.py
```

```
# Import extract, transform, and load functions
from pipeline utils import extract, transform, load

# Run the data pipeline
raw_stock_data = extract("raw_stock_data.csv")
clean_stock_data = transform(raw_stock_data)
load(clean_stock_data)
```

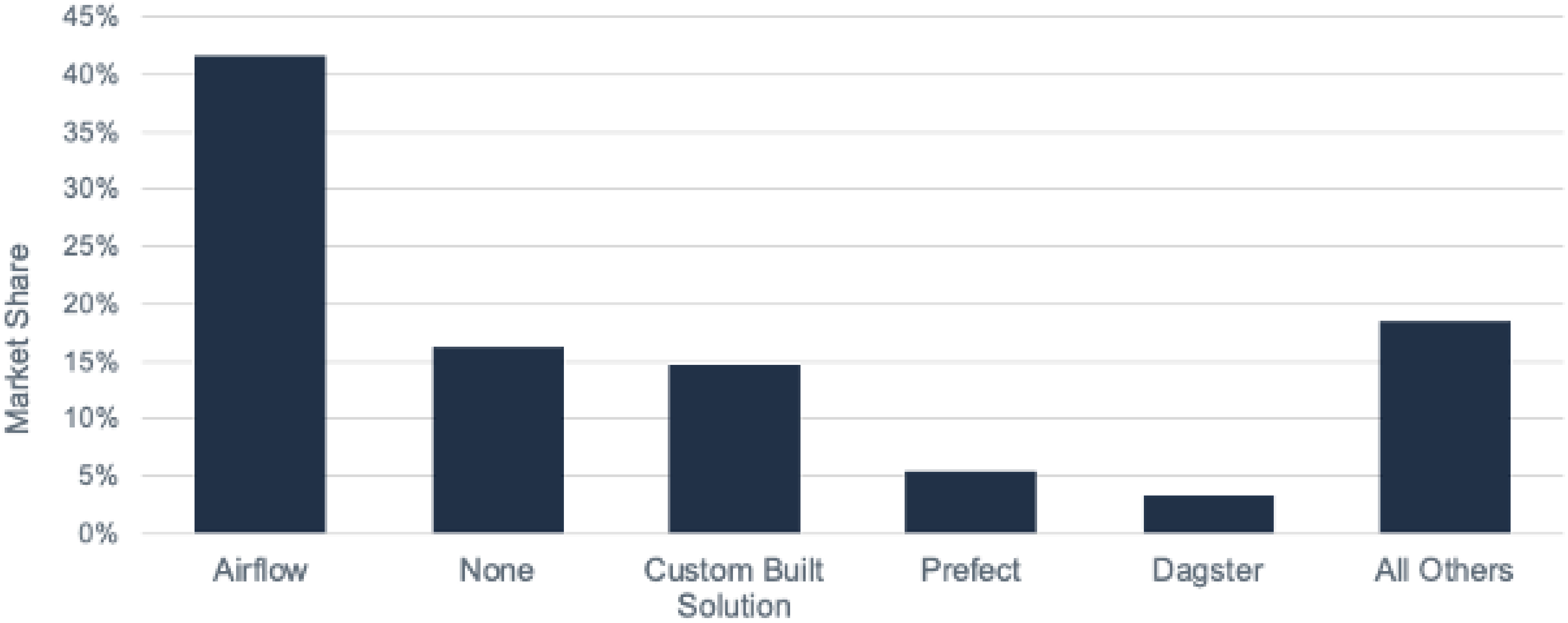
```
> ls
 etl_pipeline.py
 pipeline_utils.py
```

Running a data pipeline end-to-end

```python
import logging
from pipeline_utils import extract, transform, load

logging.basicConfig(format='%(levelname)s: %(message)s', level=logging.DEBUG)
```

```python
try:
    # Extract, transform, and load data
    raw_stock_data = extract("raw_stock_data.csv")
    clean_stock_data = transform(raw_stock_data)
    load(clean_stock_data)

    logging.info("Successfully extracted, transformed and loaded data.")  # Log success message

# Handle exceptions, log messages
except Exception as e:
    logging.error(f"Pipeline failed with error: {e}")
```

Orchestrating data pipelines in production



[1] https://open.substack.com/pub/seattledataguy/p/the-state-of-data-engineering-part?r=1po78c&utm_campaign=post&utm_medium=web