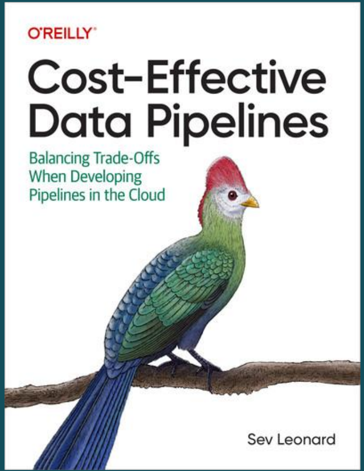
# Schemas: The Key to Data Happiness

Sev Leonard
PyCon 2025

# About

PYCON 2016
ROSE CITY
PORTLAND, OREGON
MAY 28TH - JUNE 5TH

The Data Scout




OHSU

FLETCH

OREGON HEALTH AUTHORITY

# In this talk I will…

- What is a schema?
- Awesome things you can do with schemas
  - **Validate** data
  - **Orchestrate** pipelines
  - **Communicate** about data
  - **Generate** data and code
  - **Automate** schema extraction and maintenance
- Tips

# What do you mean by schema?

A blueprint for data

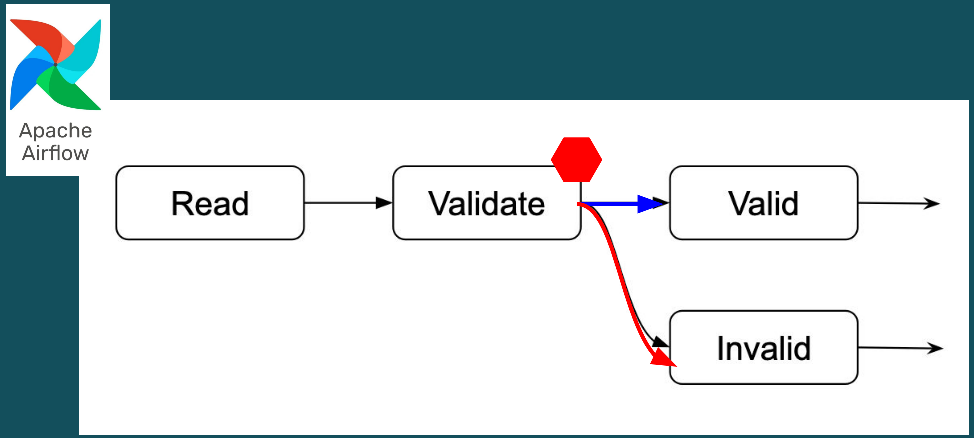
- Type
- Nullable
- Accepted values

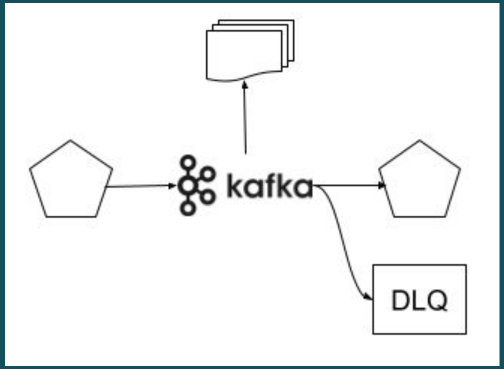
```
cat_schema = {
    "items": [
        {"name": {"type": "string"}}
        {"breed": {"enum": ["dsh", "persian", "siamese"]}},
        {"age": {
            "type": "integer",
            "minimum": 0
        }},

        "required": ["name", "age"]
    ]
}
```

# Validate!

```
>>> from jsonschema import validate
>>> validate([{"name": "Princess", "breed": "siamese"}], cat_schema)

Traceback (most recent call last):
  ...
jsonschema.exceptions.ValidationError: 'age' is a required property

Failed validating 'required' in schema['items']:
    {'properties': {'age': {...},
                    'breed': {...},
                    'name': {...}},
     'required': ['name', 'age'],
     'type': 'object'}

On instance[0]:
    {'breed': 'siamese', 'name': 'Princess'}
```

# Orchestrate!




*Prevent failures and preserve data quality*

[Ch 4 - Data Validation]    [Data validation code examples]

# Communicate!

```
cat_schema = {
    "items": [
        {"name": {"type": "string"}}
        {"breed": {"enum": ["dsh", "persian", "siamese"]}},
        {"age": {
            "type": "integer",
            "minimum": 0
        }},

        "required": ["name", "age"]
    ]
}
```

# Generate!

- Data - Ch 9 Cost-Effective Data Pipelines
- Code

# Faker

- Faker is a Python library for generating fake data
- Built-in and custom data providers [code example]
- Discussion & PR to implement distributors

# Generate data with Faker

```json
"name": {
    "type": "string", "faker": "name"
},
"breed": {
    "type": "string",
    "enum": ["dsh", "persian", "siamese"]
},
"age": {
    "type": "integer",
    "minimum": 0
}
```

```python
from faker import Faker

record = {}
for field, definition in cat_schema["items"]["properties"].items():
    if "faker" in definition:
        faker_method = getattr(fake, definition["faker"])
        record[field] = faker_method()
    else:
        if definition.get("enum"):
            record[field] = fake.random_element(definition["enum"])
        elif definition["type"] == "integer":
            record[field] = fake.random_int(
                definition.get("minimum", 0),
                definition.get("maximum", 30))
```

# Fake cats

```
def generate_test_data(num_recs=10):
    fake = Faker()
    test_data = []
    for _ in range(num_recs):
        record = {}
        for field, definition in cat_schema["items"]["properties"].items():
            ...
    return test_data

>>> generate_test_data(4)
[
    {'name': 'Dr. Gregory Thomas', 'breed': 'dsh', 'age': 20},
    {'name': 'Brian Gomez', 'breed': 'persian', 'age': 6},
    {'name': 'Mary Hubbard', 'breed': 'dsh', 'age': 27},
    {'name': 'Judith Espinoza', 'breed': 'persian', 'age': 8}
]
```

# AI cats

“Generate 4 json records that conform to the following json schema. If a property has a "faker" attribute, use the named method in the python faker library”


```json
    {
        "name": "John Doe",
        "breed": "persian",
        "age": 5
    },
    {
        "name": "Jane Smith",
        "breed": "dsh",
        "age": 3
    },
    {
        "name": "Alice Johnson",
        "breed": "siamese",
        "age": 7
    },
    {
        "name": "Bob Brown",
        "breed": "dsh",
        "age": 2
    }
```

# AI cats take 2

“Given the following json schema <u>and sample records</u>, create 10 json records that match this dataset”

```json
    {
        "name": "Michael Smith",
        "breed": "persian",
        "age": 4
    },
    {
        "name": "Emma Davis",
        "breed": "dsh",
        "age": 8
    },
    {
        "name": "William Garcia",
        "age": 10
    }
]
```

# Generated data applications

- Unit testing
- Load testing
- Creating anomalous events
- ML & AI training

# Code generation

“Create sqlalchemy table definitions for the following schema”

```python
# Define the Cat model
class Cat(Base):
    __tablename__ = 'cats'  # Name of the table in the database

    id = Column(Integer, primary_key=True, autoincrement=True)  # Primary key
    name = Column(String, nullable=False)  # Name of the cat (required)
    breed = Column(Enum("dsh", "persian", "siamese"), nullable=True)  # Breed of the cat (optional)
    age = Column(Integer, nullable=False)  # Age of the cat (required, must be >= 0)
```

## “Using the same schema, how would you model this using dbt?”

```sql
-- models/cats.sql

with source_data as (
    select
        name,
        breed,
        age
    from {{ ref('source_table') }}  -- Replace 'source_table' w
)

select
    name,
    case
        when breed in ('dsh', 'persian', 'siamese') then breed
        else null  -- Handle invalid breeds
    end as breed,
    age
from source_data
where age >= 0  -- Ensure age is non-negative
```

```sql
-- tests/cats_age_test.sql

select *
from {{ ref('cats') }}
where age < 0
```

# Automate!

- Schema extraction
- Detecting drift

# Schema extraction

```
df = (spark
        .read
        .option("inferSchema", True)
        .json('sample_data.json'))

>>> df.printSchema()
root
 |-- age: long (nullable = true)
 |-- breed: string (nullable = true)
 |-- name: string (nullable = true)
```

```
from genson import SchemaBuilder

builder = SchemaBuilder()

for record in sample_data:
    builder.add_object(record)

builder.to_schema()
{
    '$schema': 'http://json-schema.org/schema#',
    'type': 'object',
    'properties': {
        'name': {'type': 'string'},
        'breed': {'type': 'string'},
        'age': {'type': 'integer'}
    },
    'required': ['age', 'name']
}
```

# Detecting schema drift

```
[{"name": "Party Cat", "breed": "dsh", "age": 20, "food": "crunchy bits"}]
```

```
from pyspark.testing import assertSchemaEqual

df = (spark
        .read
        .option("inferSchema", True)
        .json('party_cat.json'))

assertSchemaEqual(df.schema, cat_schema)

PySparkAssertionError: [DIFFERENT_SCHEMA] Schemas do not match.
--- actual
+++ expected
- StructType([StructField('age', LongType(), True), StructField('food', StringType(), True), ...])
?                           ^    ^^          ^^^                    ^^^^   ^  ---         ------

+ StructType([StructField('age', IntegerType(), False), StructField('name', StringType(), False), ...])
?                          ^^^   ^^ +++          ^ ^                    + ^    ^^^^           ^^^^
```

Spark schema comparison code

# Applying a schema != validation

```
[{"name": "Party Cat", "breed": "dsh", "age": 20, "food": "crunchy bits"}]
```

```
>>> df = (spark
            .read
            .schema(cat_schema)
            .json('party_cat.json'))

>>> df.show()
+---------+-----+---+
|     name|breed|age|
+---------+-----+---+
|Party Cat|  dsh| 20|
+---------+-----+---+
```

# Validating data with extra fields

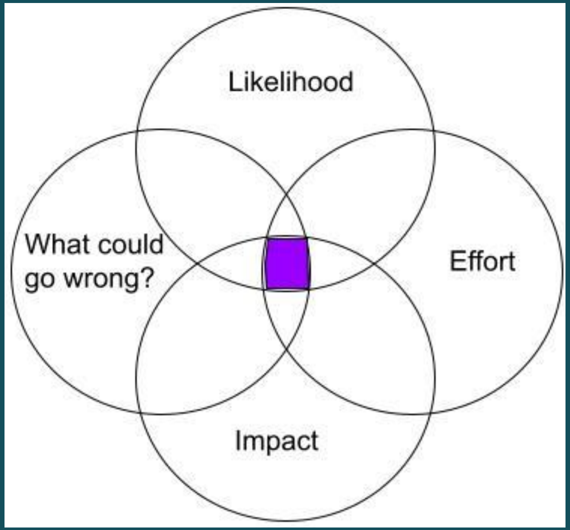
```
>>> from jsonschema import validate
>>> party_cat = [{"name": "Party Cat",
                  "breed": "dsh",
                  "age": 20,
                  "food": "crunchy bits"}]
>>> validate(party_cat, cat_schema)
>>>
```

Extra fields are allowed by default

```
cat_schema = {
    "items": {
        "properties": { ... },
        "required": ["name", "age"],
        "additionalProperties": False
    }
}

>>> validate(party_cat, cat_schema)

ValidationError: Additional properties are not allowed ('food' was unexpected)
```

# When to use schemas

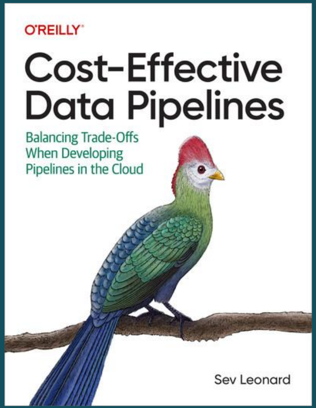
- Data source and target maturity
- Product / Organization maturity
- Prioritize



# Summary

- **Validate** to prevent pipeline failures and data downtime
- **Orchestrate** to control invalid data handling and monitoring
- **Communicate** data characteristics to humans and machines
- **Generate** testing and training data and to reduce coding overhead
- **Automate** to track schema drift and minimize manual schema maintenance
- Confirm assumptions about schema application and validation
- Assess the right level of schematization for your situation

# Thanks!


O'REILLY
Cost-Effective Data Pipelines
Balancing Trade-Offs When Developing Pipelines in the Cloud
Sev Leonard

# Links

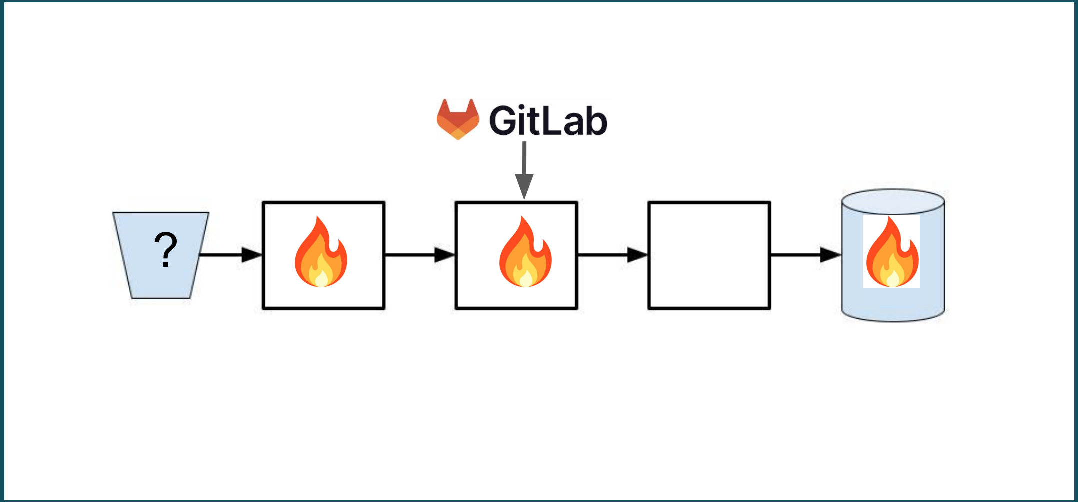
- Ch 9 Cost-Effective Data Pipelines - Data Generation
- Confluent Schema Registry
- JSON Schema
- Apache Avro
- Generate JSON schemas from JSON
- Data Contracts
- pyspark testing documentation
- JSON schema additionalFields property
- Code repo - Cost Effective Data Pipelines

# Miscellany

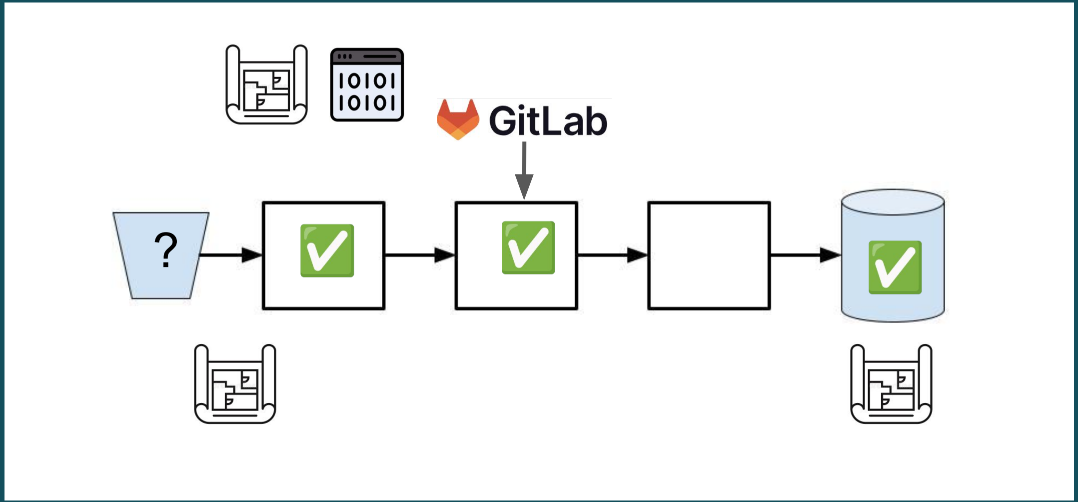
# How would you like a solution for these problems?

- Failing pipelines
- Missing or malformed data
- Bugs in data transformation code
- Inadequate training data
- Lack of understanding about data structure
- Coding overhead
- Out of date test data due to manual maintenance
- Demonstrating to your boss that you used AI for something

# How can schemas help?



# How can schemas help?



# Data formats and schemas

Formats with embedded schemas

- Parquet
- Avro
- Protobuf

Formats without embedded schemas

- CSV
- JSON