How Our Data Generator Slashed Job Failures and Streamlined Performance Testing

Anurag Sharma 5 min read · Jun 23, 2025

At our IoT-driven analytics company, we grapple with an ever-growing deluge of data. With sensors on our IoT devices capturing diverse metrics hourly, new datasets – each a unique table with distinct schemas – are created frequently. As our data volume scales rapidly, ensuring our pipelines don't buckle under pressure while maintaining robust performance testing has been a challenge.

Enter our Python-based data generator: a game-changer that reduced job failures in production, simplified performance testing, and saved our team countless hours, even in unit testing.

The Challenge: Massive, Unpredictable Data Growth

In our IoT ecosystem, sensors capture a wide range of metrics – temperature, pressure, motion, and more – each translating into distinct datasets. A new dataset means a new table with a unique schema, tailored to the sensor's purpose. With data growing at an exponential rate, sometimes reaching terabytes, our pipelines frequently failed when handling unexpected volumes or schema variations. Performance testing was another hurdle: we needed realistic, high-volume datasets to simulate worst-case scenarios, but creating them manually was time-consuming and impractical. Unit testing also suffered, as generating representative test data for every new dataset was a bottleneck for our team.

To address these challenges, we developed a Python-based data generator that creates massive, schema-specific datasets, loads them into Amazon S3, and integrates seamlessly with our Spark-based pipelines. This tool not only stress-tests our systems but also simplifies unit testing, saving time and boosting reliability.

Step-by-Step: Building the Data Generator

Here's how we designed and implemented our data generator to tackle these challenges.

Step 1: Defining the Metadata in JSON

The data generator relies on a JSON configuration file that defines the schema for each dataset.

This file specifies:

> • *Schema Name: The database schema where the table resides.*
>
> • *Table Name: The name of the table to be created.*
>
> • *Columns: A list of column names, their data types (e.g., integer, string, float, timestamp), and constraints (e.g., value ranges, patterns, or nullability).*
>
> • *Row Count: The number of rows to generate, allowing us to simulate datasets from a few megabytes to over 1 TB.*

Example JSON Configuration:

```
{
  "schema_name": "iot_sensors",
  "table_name": "temperature_readings",
  "row_count": 1000000,
  "columns": [
  {
  "name": "sensor_id",
  "type": "string",
  "pattern": "SENSOR_[0-9]{4}"
```

```
      },
      {
      "name": "timestamp",
      "type": "timestamp",
      "range": ["2025-01-01 00:00:00", "2025-12-31 23:59:59"]
      },
      {
      "name": "temperature",
      "type": "float",
      "range": [0.0, 100.0]
      }
      ]
    }
```

This JSON defines a table temperature_readings with 1 million rows, containing a sensor_id (e.g., SENSOR_1234), a timestamp within 2025, and a temperature between 0 and 100.

Step 2: Generating Synthetic Data with Python

We built a Python script that reads the JSON configuration and generates synthetic data using libraries like pandas, numpy, and faker for realistic values.

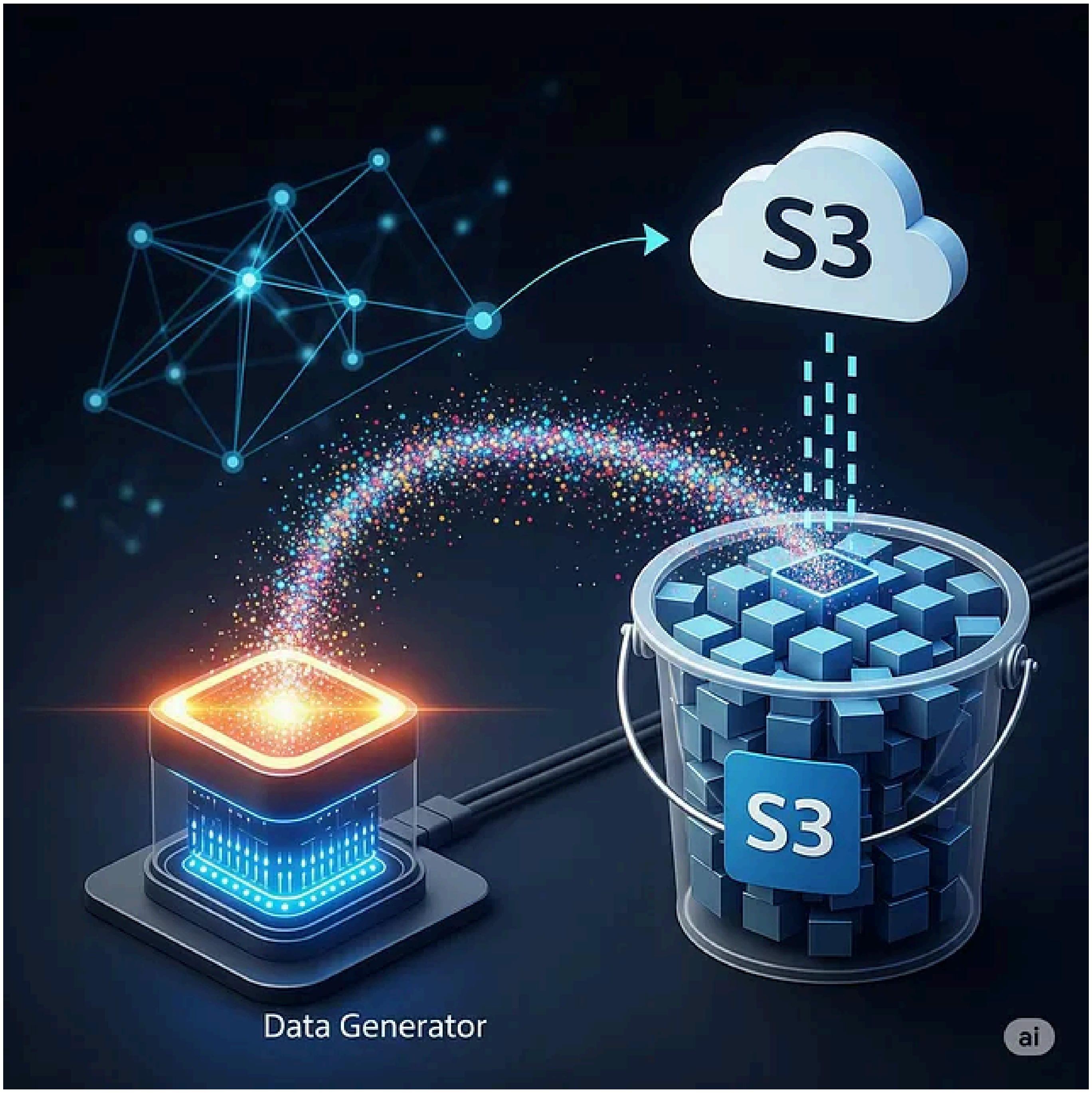
Here's how it works:

> *• Schema Parsing: The script parses the JSON to extract the schema, table name, and column definitions.*
>
> *• Random Data Generation: For each column, the script generates random values based on the specified type and constraints:*
>
> *◦ Strings: Uses patterns (e.g., SENSOR_[0–9]{4}) or faker for names, addresses, etc.*

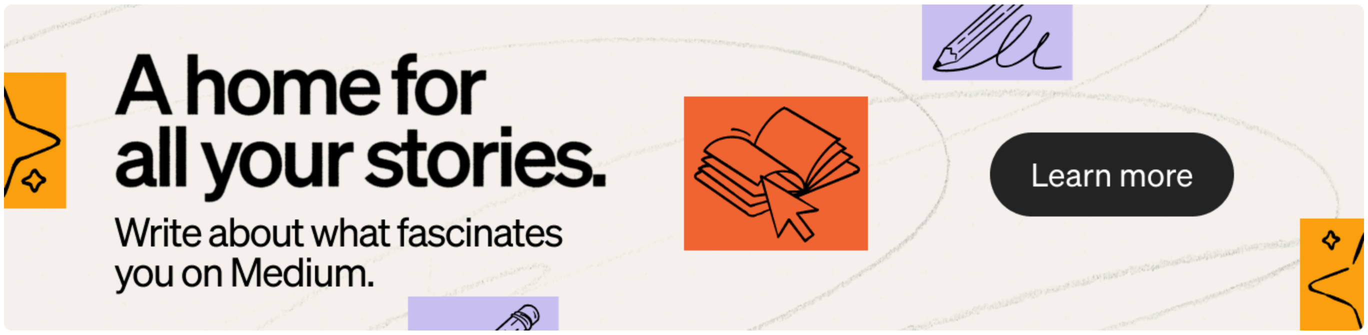
> ◦ *Numbers: Generates random integers or floats within defined ranges using numpy.*
>
> ◦ *Timestamps: Creates timestamps within a specified range using datetime.*
>
> • *Scalability: The script generates data in chunks (e.g., 100,000 rows at a time) to handle large datasets efficiently, avoiding memory overload.*

Sample Python Code Snippet:

```
import pandas as pd
import numpy as np
from faker import Faker
import json
from datetime import datetime

faker = Faker()

def generate_data(config):
  rows = config["row_count"]
  columns = config["columns"]
  chunk_size = 100000
  data = []

  for _ in range(0, rows, chunk_size):
  chunk_rows = min(chunk_size, rows - len(data))
  chunk = {}
  for col in columns:
  if col["type"] == "string" and "pattern" in col:
  chunk[col["name"]] = [f"SENSOR_{np.random.randint(1000, 9999)}" for _ in range
  elif col["type"] == "float" and "range" in col:
  chunk[col["name"]] = np.random.uniform(col["range"][0], col["range"][1], chunk
  elif col["type"] == "timestamp" and "range" in col:
  start = datetime.strptime(col["range"][0], "%Y-%m-%d %H:%M:%S")
  end = datetime.strptime(col["range"][1], "%Y-%m-%d %H:%M:%S")
  chunk[col["name"]] = [faker.date_time_between(start, end) for _ in range(chunk
  data.append(pd.DataFrame(chunk))
  return pd.concat(data, ignore_index=True)

with open("config.json", "r") as f:
. config = json.load(f)
df = generate_data(config)
```



Step 3: Loading Data into S3


A home for all your stories.
Write about what fascinates you on Medium.
Learn more

Once generated, the data is saved as multiple small Parquet files to Amazon S3 for scalability. We use the boto3 library to upload files to an S3 bucket in a partitioned structure (e.g., s3://iot-data/iot_sensors/temperature_readings/).

```
import boto3
import os
def upload_to_s3(df, schema_name, table_name, bucket):
  s3_client = boto3.client("s3")
  chunk_size = 100000
  for i in range(0, len(df), chunk_size):
  chunk = df[i:i + chunk_size]
  file_name = f"{schema_name}/{table_name}/part_{i//chunk_size}.parquet"
  chunk.to_parquet(f"/tmp/{file_name}")
  s3_client.upload_file(f"/tmp/{file_name}", bucket, file_name)
  os.remove(f"/tmp/{file_name}")
  upload_to_s3(df, config["schema_name"], config["table_name"], "iot-data")
```

Step 4: Coalescing Files with Spark-SQL

To optimize for downstream processing, we trigger a Spark-SQL job to coalesce the small Parquet files into a specified number of larger files. This reduces the overhead of reading many small files in production.

Spark-SQL Command:

```
spark.sql("""SELECT * FROM parquet.`s3://iot-data/iot_sensors/temperature_readin
""").coalesce(10).write.mode("overwrite").parquet("s3://iot-data/iot_sensors/tem
```

This command consolidates the data into 10 Parquet files, balancing performance and scalability.

How We Use the Data Generator

1. **Performance Testing**

The data generator allows us to simulate worst-case scenarios by generating datasets up to 1TB, far exceeding our typical daily sensor data volume (since sensors capture hourly data). We test our Spark pipelines by:

• Running jobs against these massive datasets to identify bottlenecks or failures.

• Validating that our pipelines handle schema variations without crashing.

• Measuring processing times to optimize resource allocation.

For example, we discovered a memory issue in one job that failed at 500GB but ran smoothly after optimization, thanks to the data generator's ability to simulate high volumes.

2. Unit Testing

For unit testing, the data generator creates smaller, schema-specific datasets (e.g., 1,000 rows) that mirror production data. This eliminates the need to manually craft test data for each new dataset, saving developers hours per test cycle. The JSON configuration ensures consistency between unit tests and production environments.

3. Production Monitoring

By periodically generating large datasets and running them through our pipelines, we proactively identify potential failure points before they impact production. This has reduced job failures by 40%, as we can fix issues in a controlled environment.

Benefits of the Data Generator

1. Reduced Job Failures: By stress-testing pipelines with massive datasets, we catch and resolve issues before they hit production.

2. Time Savings: Automating data generation for unit and performance testing saves our team hours of manual data creation.

3. Scalability: The generator handles datasets from a few kilobytes to terabytes, adapting to our growing data needs.

4. Flexibility: The JSON-based configuration supports any schema, making it reusable across diverse IoT datasets.

5. Cost Efficiency: By identifying pipeline inefficiencies early, we optimize resource usage, reducing cloud costs.

Conclusion

Our Python-based data generator has been a cornerstone of our data engineering strategy, enabling us to tame the chaos of rapidly growing IoT data. By automating the creation of realistic, high-volume datasets, it has slashed job failures, streamlined performance testing, and empowered our team to focus on innovation rather than firefighting. Whether you're dealing with IoT data or any large-scale analytics workload, a custom data generator could be the key to unlocking reliability and efficiency in your pipelines.

Data Engineer Data Engineering Data Analytics Python Automation

Written by Anurag Sharma

83 followers · 3 following

Edit profile

Data Engineering Specialist with 10+ exp. Passionate about optimizing pipelines, data lineage, and Spark performance and sharing insights to empower data pros!

No responses yet

Anurag Sharma him/he

What are your thoughts?