

Gowtham SB

Linkedin - <https://www.linkedin.com/in/sbgowtham/>

The Big Data Fix Book

About the Book

Everyone talks about building ETL pipelines in interviews.
But guess what? So did the 10 people before you.

This book isn't about just building pipelines — it's about fixing them.

Real production issues. Real bugs. Real headaches. And real fixes that got people hired.

"The Interview Cracker: 100 Real Big Data Engineering Issues That Impress Instantly" is your interview edge. It's a curated collection of challenges in Spark, Kafka, BigQuery, Airflow, and more — with the exact configs, code snippets, and storylines to talk through confidently.

Each issue comes with:

- A real **story**
- The actual **problem**
- The **fix**
- And how to **add it to your resume**

✨ Why it works: Interviewers have seen a hundred cookie-cutter projects.
But talk about a real challenge you fixed — and they'll sit up.
Who knows, they might even *add your fixes to their own resume* 😊.

Use this book as a **launchpad for deeper learning**, or as your **pre-interview booster**. Either way — you'll walk in with stories that stand out.

Disclaimer:




While I've used tools like the **internet and language models** to help structure this book, all the concepts, challenges, and insights shared here are based on my own real-world experience and understanding.

How to Use This Book in Interviews

This book isn't meant to be read cover to cover like a novel — and definitely not just memorized.

Each issue here is simplified into a crisp, structured format: **story, challenge, fix, config/code, and how to explain it in your resume or interview**. No fluff — just sharp, real-world content designed to impress.

Here's how to make it work for you:

-  **Do a deep-dive on 2 or 3 issues:** You don't need to explain all 100 problems in an interview. Just pick **2 or 3 that truly resonate with your experience or interest**, research them in detail, understand the config/code, and make them your go-to stories.
-  **Integrate them into your project explanation:** Instead of just saying, *"I built an ETL pipeline,"* bring in one of these challenges and how you solved it. That **one story will make you stand out** from every other candidate who said the same generic answer before you.
-  **Stretch it smartly:** A good project walkthrough with 2–3 real issues can take **20 minutes of a 1-hour interview**, showcasing your problem-solving mindset, depth, and hands-on experience.


 **Pro tip:** These are the kinds of stories that **interviewers themselves may add to their resume** — yes, really.

Table of Contents

1. **Spark Core & Hive Issues (1–30)**
 2. **Spark SQL Issues**
 - Part 1: Issues 31–40
 - Part 2: Issues 51–60
 3. **Spark Streaming Issues (61–70)**
 4. **Kafka Issues (71–80)**
 5. **BigQuery Issues (81–90)**
 6. **Airflow Issues (91–100)**
-

Issue 1: Spark + Hive ACID

Story: During a migration from Hive to Spark, our team encountered errors when trying to query Hive ACID tables directly from Spark jobs.

Challenge: Spark does not natively support Hive ACID table read operations and shows corrupted or duplicated data.

Fix / Solution:

- Use Hive Warehouse Connector (HWC) to read ACID tables
- Alternatively, use Hive to create external non-transactional views that Spark can read safely.

Code / Configuration:

```
spark.sql.hive.hwc.execution.mode=spark  
spark.datasource.hive.warehouse.load.staging.dir=/tmp
```

Description (Resume): Implemented Hive Warehouse Connector integration in Spark to safely access transactional (ACID) Hive tables and maintain consistency.

Issue 2: Spark Join Skew

Story: A batch ETL job took hours due to a skewed join on country codes, where 'US' had 70% of the data.

Challenge: Skewed keys in joins cause tasks to hang and waste resources.

Fix / Solution:

- Introduce salting on skewed keys before join
- Enable Adaptive Query Execution (Spark 3+)

Code / Configuration:

```
val saltedDF = df.withColumn("salted_key", concat(col("country"), lit("_"), (rand() *  
10).cast("int")))  
spark.conf.set("spark.sql.adaptive.skewJoin.enabled", true)
```

Description (Resume): Resolved Spark join skew issues in large datasets using salting and adaptive query execution, improving job speed by 80%.

Issue 3: Streaming Lag in Structured Streaming

Story: A Kafka → Spark Streaming job consistently lagged behind incoming events and triggered alerts.

Challenge: The streaming job couldn't process data in real-time, causing backlogs.

Fix / Solution:

- Enable backpressure and batch limit
- Optimize batch duration and memory

Code / Configuration:

```
spark.streaming.backpressure.enabled=true  
spark.streaming.kafka.maxRatePerPartition=1000
```

Description (Resume): Tuned structured streaming pipeline using Kafka with Spark to enable real-time processing with backpressure and throughput control.

Issue 4: Too Many Small Files on S3

Story: A daily Spark write created 30,000+ tiny Parquet files, making Hive reads and S3 listing very slow.

Challenge: Small file problem leads to performance issues and high S3 costs.

Fix / Solution:

- Coalesce files before write
- Optimize parallelism and partitioning

Code / Configuration:

```
df.coalesce(50).write.mode("overwrite").partitionBy("event_date").parquet("s3://bucket/path")
```

Description (Resume): Solved small file issue in S3-based Spark pipeline by implementing data coalescing and write optimizations for Parquet.

Issue 5: Spark Job OOM on Driver (collect())

Story: A developer used `.collect()` on a 1GB+ DataFrame, crashing the driver node in dev and prod.

Challenge: Large `.collect()` operations load data into driver memory, causing OOM.

Fix / Solution:

- Avoid `collect()` unless limited row count
- Use `.limit()` and `.toPandas()` for sampling

Code / Configuration:

```
df.limit(1000).toPandas()
```

Description (Resume): Prevented production Spark driver OOM errors by replacing heavy `.collect()` calls with efficient DataFrame sampling techniques.

Issue 6: Python UDF Slowing Down Spark Jobs

Story: A UDF written in Python for string cleaning caused a 10x slowdown in our ETL pipeline.

Challenge: Python UDFs don't scale as efficiently as Spark-native functions.

Fix / Solution:

- Replaced Python UDF with `F.regexp_replace`, a native Spark SQL function.

Code / Configuration:

```
df.withColumn("cleaned_col", F.regexp_replace("col", "\\W", ""))
```

Description (Resume): Boosted Spark performance by refactoring slow Python UDFs into native Spark SQL expressions.

Issue 7: Spark SQL Ignored Partition Pruning

Story: A query scanning a partitioned Hive table still scanned all partitions.

Challenge: Improper filtering disabled partition pruning.

Fix / Solution:

- Avoid functions like `date_format()` in WHERE clause

Code / Configuration:

-- Wrong:

```
WHERE date_format(event_date, 'yyyy-MM-dd') = '2023-10-01'
```

-- Right:

```
WHERE event_date = '2023-10-01'
```

Description (Resume): Improved query performance by ensuring proper partition filter usage in Spark SQL on partitioned Hive tables.

Issue 8: Data Duplication During Reprocessing

Story: A reprocessing job resulted in duplicate rows in the target Delta table.

Challenge: No deduplication logic during upserts.

Fix / Solution:

- Use `MERGE INTO` with deduplication conditions.

Code / Configuration:

```
MERGE INTO target USING source ON target.id = source.id  
WHEN MATCHED THEN UPDATE SET *  
WHEN NOT MATCHED THEN INSERT *
```


Description (Resume): Ensured idempotent Spark job design using Delta Lake MERGE logic to avoid duplicate records.

Issue 9: S3 Write Intermittent Failures in Spark

Story: Random write failures to S3 occurred under high load.

Challenge: S3 consistency issues and network throttling.

Fix / Solution:

- Enabled S3A fast upload
- Tuned connection parameters

Code / Configuration:

```
spark.hadoop.fs.s3a.fast.upload=true  
spark.hadoop.fs.s3a.connection.maximum=100  
spark.hadoop.fs.s3a.threads.max=50
```

Description (Resume): Achieved reliable Spark-to-S3 writes by optimizing S3 connector configurations for concurrency and performance.

Issue 10: Spark Dynamic Allocation Not Scaling Executors

Story: Despite DRA enabled, only 2 executors were allocated for a large job.

Challenge: Shuffle service not enabled.

Fix / Solution:

- Enabled external shuffle service

Code / Configuration:

```
spark.dynamicAllocation.enabled=true  
spark.shuffle.service.enabled=true  
spark.dynamicAllocation.minExecutors=2  
spark.dynamicAllocation.maxExecutors=100
```

Description (Resume): Configured dynamic resource allocation in Spark to optimize executor scaling based on workload demand.

Spark Core & Hive Issues (11–20)

Issue 11: Spark Job Crashed Due to Too Many Tasks

Story: A team ran a Spark job with 10,000 partitions, and the cluster crashed due to too many concurrent tasks.

Challenge: Uncontrolled task parallelism overwhelmed cluster resources.

Fix / Solution:

- Limited number of shuffle partitions

Code / Configuration:

```
spark.sql.shuffle.partitions=500
```

Description (Resume): Improved cluster stability by controlling Spark shuffle partition count to avoid task explosion.

Issue 12: Misuse of repartition() Caused Long Execution

Story: A developer added `.repartition(2000)` before a write, which caused a huge shuffle and job slowdown.

Challenge: Over-repartitioning leads to expensive shuffles.

Fix / Solution:

- Used `.coalesce()` instead

Code / Configuration:

```
df.coalesce(200).write.parquet("s3://output")
```

Description (Resume): Reduced job runtime by replacing expensive repartitioning with coalesce to minimize shuffle.

Issue 13: Spark SQL Query Failed on Large Broadcast Table

Story: A join with a large dimension table marked for broadcast failed with memory error.

Challenge: Broadcast threshold exceeded executor memory.

Fix / Solution:

- Increased broadcast threshold or removed hint

Code / Configuration:

```
spark.sql.autoBroadcastJoinThreshold=104857600 # 100MB
```

Description (Resume): Optimized join strategies by adjusting broadcast settings to avoid executor memory failures.

Issue 14: Legacy Hive Table Format Incompatible with Spark

Story: Spark failed to read an old Hive table stored in TextFile format.

Challenge: Format incompatibility and missing schema.

Fix / Solution:

- Used schema inference and custom read options

Code / Configuration:

```
spark.read.option("delimiter", "    ").csv("path").toDF("col1", "col2")
```

Description (Resume): Enabled legacy Hive table integration with Spark by implementing schema inference and format compatibility.

Issue 15: Spark Writes Ignored Partition Overwrite Mode

Gowtham SB

Linkedin - <https://www.linkedin.com/in/sbgowtham/>

Story: `.mode("overwrite")` erased entire Hive table instead of just one partition.

Challenge: Partition overwrite not dynamic.

Fix / Solution:

- Enabled dynamic partition overwrite

Code / Configuration:

```
spark.sql.sources.partitionOverwriteMode=dynamic
```

Description (Resume): Prevented full table overwrite in Hive by enabling dynamic partition overwrite mode in Spark.

Issue 16: Data Type Mismatch Broke Write to Hive

Story: Spark failed to write because of decimal mismatch (e.g., `Decimal(10,5)` vs `Decimal(18,2)`).

Challenge: Strict schema enforcement in Hive.

Fix / Solution:

- Casted columns explicitly

Code / Configuration:

```
df.withColumn("amount", col("amount").cast("decimal(18,2)"))
```

Description (Resume): Ensured compatibility in Hive writes by performing data type casting within Spark transformations.

Issue 17: Cache Did Not Persist Across Spark Sessions

Story: Cached table was missing when reusing it in another job.

Challenge: Cache is session-scoped.

Fix / Solution:

- Used checkpointing or persisted table to disk

Code / Configuration:

```
df.write.mode("overwrite").parquet("/tmp/checkpoint")
```

Description (Resume): Implemented persistent storage in Spark for reuse across job stages using checkpointing.

Issue 18: Spark Job Hangs When Reading Too Many Small Files

Story: Reading millions of tiny JSON files caused Spark to hang or crash.

Challenge: File listing and scheduling overload.

Fix / Solution:

- Combined files using Hadoop FileInputFormat
- Used recursive file reading and `maxPartitionBytes`

Code / Configuration:

```
spark.hadoop.mapreduce.input.fileinputformat.input.dir.recursive=true  
spark.sql.files.maxPartitionBytes=134217728
```

Description (Resume): Solved small file input bottlenecks in Spark by tuning file merge and read configurations.

Issue 19: Parquet Read Skipped Columns Silently

Story: Spark read Parquet file but silently skipped newly added columns.

Challenge: Schema evolution disabled.

Fix / Solution:

Gowtham SB

Linkedin - <https://www.linkedin.com/in/sbgowtham/>

- Enabled schema merge

Code / Configuration:

```
spark.sql.parquet.mergeSchema=true
```

Description (Resume): Ensured complete schema evolution support in Parquet reads using Spark config tuning.

Issue 20: Spark SQL Failed to Push Filters on Nested Fields

Story: A query on nested JSON failed to push filters, resulting in full scan.

Challenge: Spark needs explicit extraction of nested fields.

Fix / Solution:

- Used `get_json_object` or `explode` to flatten fields

Code / Configuration:

```
SELECT * FROM table WHERE get_json_object(json_col, '$.type') = 'event'
```

Description (Resume): Improved performance in Spark JSON queries by restructuring nested field access for filter pushdown.

Spark Core & Hive Issues (21–30)

Issue 21: Incorrect Partition Strategy Caused File Explosion

Story: Partitioned a table by `user_id` (high cardinality) and generated millions of files on S3.

Challenge: Poor partitioning leads to metadata bloat and slow queries.

Fix / Solution:

Gowtham SB

Linkedin - <https://www.linkedin.com/in/sbgowtham/>

- Use low-cardinality fields like `event_date`, `region`
- Apply Z-Ordering for optimized reads

Code / Configuration:

```
df.write.partitionBy("event_date", "country").parquet("s3://.../")
```

Description (Resume): Improved Spark data lake performance by re-designing partition strategy and eliminating small file issues.

Issue 22: Spark Structured Streaming Query Didn't Progress

Story: Streaming query started but never progressed beyond first microbatch.

Challenge: Watermarking not applied on event-time column

Fix / Solution:

- Apply watermark to define lateness tolerance

Code / Configuration:

```
df.withWatermark("event_time", "10 minutes")
```

Description (Resume): Enabled robust event-time streaming in Spark using watermarking for late data handling.

Issue 23: Join Performed Full Shuffle Despite Small Table

Story: A fact-dim join triggered massive shuffle even though the dim table was small.

Challenge: Spark failed to auto-broadcast

Fix / Solution:

- Use manual broadcast hint

Code / Configuration:

```
from pyspark.sql.functions import broadcast
df1.join(broadcast(df2), "id")
```

Description (Resume): Reduced job duration by applying broadcast join hint on small-dimension tables in Spark.

Issue 24: Data Corruption Due to Overlapping Writes

Story: Two Spark jobs writing to the same path caused partial/corrupted outputs.

Challenge: No write-locking mechanism in place

Fix / Solution:

- Write to separate staging paths
- Move to final destination after success

Code / Configuration:

```
write_path = "s3://bucket/tmp/job_ts=" + job_ts
```

Description (Resume): Prevented data corruption by isolating writes into staging directories and using atomic copy on completion.

Issue 25: Spark Job Not Failing on Data Errors

Story: Corrupt records silently skipped in production without alerts.

Challenge: Default mode skips bad records

Fix / Solution:

- Set fail-fast mode for early detection

Code / Configuration:

```
spark.read.option("mode", "FAILFAST").csv("s3://...")
```

Description (Resume): Enabled fail-fast ingestion in Spark to ensure immediate detection of bad input records.

Issue 26: Delta Table Reads Returned Old Snapshot

Story: Delta reads missed the latest inserts from a concurrent writer

Challenge: Stale snapshot due to caching

Fix / Solution:

- Refresh cache before read

Code / Configuration:

```
spark.catalog.refreshTable("delta_table")
```

Description (Resume): Ensured data freshness in Delta reads by programmatically refreshing table cache.

Issue 27: Long Lineage Caused Spark Job Stack Overflow

Story: 100s of transformations led to DAG explosion and stack overflow.

Challenge: Spark failed on long lineage

Fix / Solution:

- Checkpoint intermediate results

Code / Configuration:

```
df.checkpoint()
```

Description (Resume): Stabilized large Spark DAGs using checkpointing to reduce lineage size and memory pressure.

Issue 28: Spark SQL Performance Degraded Due to Missing Statistics

Story: Optimizer couldn't generate efficient plans

Challenge: Table stats missing for Catalyst

Fix / Solution:

- Run ANALYZE TABLE to collect statistics

Code / Configuration:

```
ANALYZE TABLE my_table COMPUTE STATISTICS
```

Description (Resume): Improved Spark SQL optimization by enabling cost-based planning with statistics collection.

Issue 29: Spark Failed on Too Many Output Files (TooManyFilesException)

Story: High cardinality output caused file listing to fail

Challenge: S3 has listing limitations

Fix / Solution:

- Use coalesce or dynamic partition pruning

Code / Configuration:

```
df.coalesce(200).write...
```

Description (Resume): Resolved output bottlenecks in Spark jobs by limiting file generation through coalescing strategy.

Issue 30: MLlib Model Training Fails on Sparse Data

Gowtham SB

Linkedin - <https://www.linkedin.com/in/sbgowtham/>

Story: Model training failed with NaN and 0 vector warnings

Challenge: Data not preprocessed properly

Fix / Solution:

- Impute missing values, remove low variance columns

Code / Configuration:

```
from pyspark.ml.feature import Imputer
imputer = Imputer().setInputCols(["col1"]).setOutputCols(["col1"])
```

Description (Resume): Improved Spark ML model accuracy by implementing data cleansing and imputation strategies.

Spark Core & Hive Issues (31–40)

Issue 31: Checkpoint Location in Streaming Not Set

Story: A streaming job failed to resume state after restart, leading to data duplication.

Challenge: No checkpoint directory was defined for structured streaming.

Fix / Solution:

- Always set a persistent checkpoint location.

Code / Configuration:

```
query.writeStream.option("checkpointLocation", "/path/to/checkpoint").start()
```

Description (Resume): Enabled fault-tolerant Spark Structured Streaming by integrating durable checkpointing mechanisms.

Issue 32: Spark Job Failed Due to JVM PermGen Error

Story: Job crashed on large joins with **PermGen space** error.

Challenge: High number of UDFs and classes loaded exceeded JVM memory limit.

Fix / Solution:

- Tuned JVM memory settings and consolidated UDF usage.

Code / Configuration:

```
--conf "spark.executor.extraJavaOptions=-XX:MaxPermSize=512m"
```

Description (Resume): Solved JVM PermGen failures by optimizing Spark UDFs and tuning executor memory parameters.

Issue 33: UDF Results Inconsistent Across Partitions

Story: A UDF using datetime without timezone caused inconsistent results across nodes.

Challenge: UDFs without deterministic behavior created non-repeatable outputs.

Fix / Solution:

- Set timezone explicitly and avoid non-deterministic logic inside UDFs.

Code / Configuration:

```
spark.conf.set("spark.sql.session.timeZone", "UTC")
```

Description (Resume): Ensured consistent distributed processing in Spark by standardizing timezone and removing non-determinism from UDFs.

Issue 34: Temporary Views Not Accessible in Other Sessions

Story: Created a temp view in one session, but it wasn't visible in another.

Challenge: Temp views are scoped to current session only.

Fix / Solution:

- Use global temp views or persist as intermediate tables.

Code / Configuration:

```
df.createOrReplaceGlobalTempView("global_table")
```

Description (Resume): Enabled cross-session data sharing in Spark by switching to global temp views for intermediate datasets.

Issue 35: spark.read.json() Fails on Schema Mismatch

Story: Ingesting semi-structured logs failed when a new field was introduced.

Challenge: Spark failed to infer schema from inconsistent JSON records.

Fix / Solution:

- Used schema evolution and permissive mode.

Code / Configuration:

```
spark.read.option("mode", "PERMISSIVE").json("path")
```

Description (Resume): Enabled resilient JSON ingestion by configuring Spark for permissive schema handling in dynamic sources.

Issue 36: Too Many Output Files Created During Write

Story: A write job produced thousands of tiny output files even on coalesce.

Challenge: Each task wrote its own file due to lack of file compaction.

Fix / Solution:

- Use repartition before write and apply file compaction downstream.

Code / Configuration:

```
df.repartition(50).write.mode("overwrite").parquet("s3://path")
```

Description (Resume): Minimized storage fragmentation in Spark writes by tuning partitioning and file compaction techniques.

Issue 37: Spark Job Took 5x Longer After Migration to GCS

Story: Migrating data from HDFS to GCS caused slower reads.

Challenge: Default connector and lack of tuning slowed performance.

Fix / Solution:

- Used optimized GCS connector and tuned IO parallelism.

Code / Configuration:

```
spark.hadoop.fs.gs.inputstream.support.gzip.encoding=false  
spark.hadoop.fs.gs.status.parallel.enable=true
```

Description (Resume): Optimized Spark performance on GCS by configuring parallel file access and using cloud-native connectors.

Issue 38: Date Functions Behaved Inconsistently on Cluster

Story: `current_date()` produced different outputs on executor vs driver.

Challenge: Timezone inconsistency.

Fix / Solution:

- Set default timezone in Spark config.

Code / Configuration:

```
spark.sql.session.timeZone=UTC
```

Description (Resume): Ensured time-based accuracy in Spark jobs by enforcing global timezone configurations.

Issue 39: Spark SQL CTE Evaluated Multiple Times

Story: Common Table Expressions (CTEs) caused redundant scans.

Challenge: Spark evaluates CTEs multiple times unless optimized.

Fix / Solution:

- Cache intermediate results explicitly or convert to temp views.

Code / Configuration:

```
df.createOrReplaceTempView("stage1")
spark.sql("SELECT * FROM stage1 WHERE condition")
```

Description (Resume): Improved query efficiency by materializing CTEs into temp views to prevent redundant Spark scans.

Issue 40: Data Missing After Delta Merge

Story: A Delta Lake upsert skipped inserting unmatched records.

Challenge: Incorrect merge condition filtered out rows.

Fix / Solution:

- Validated merge logic and handled null-safe equality.

Code / Configuration:

```
MERGE INTO target USING source ON target.id <=> source.id
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT *
```

Description (Resume): Ensured correctness of Delta Lake upserts by validating merge conditions and applying null-safe comparisons.

Spark SQL Issues (41–50)

Issue 41: SELECT * Caused Huge Query Cost in BigQuery

Story: A BigQuery job ran with `SELECT *` on a wide table and scanned terabytes of data, incurring high cost.

Challenge: Wide table scans cause unnecessary processing and billing.

Fix / Solution:

- Explicitly select only required columns.

Code / Configuration:

```
SELECT id, timestamp, status FROM dataset.table WHERE event_date = '2024-01-01'
```

Description (Resume): Reduced query cost and improved performance by avoiding `SELECT *` and implementing column-level projection.

Issue 42: Kafka Consumer Lag Never Reduced

Story: Kafka consumer showed lag increasing over time despite sufficient resources.

Challenge: Message processing rate slower than consumption rate.

Fix / Solution:

- Increased `max.poll.records` and used parallel processing.

Code / Configuration:

```
max.poll.records=1000
```

Description (Resume): Reduced Kafka lag by tuning consumer config and enabling multi-threaded message processing in Spark Streaming.

Issue 43: COUNT(DISTINCT) Slowed Down Spark SQL Job

Story: A daily report with `COUNT(DISTINCT user_id)` took 40+ minutes to run.

Challenge: DISTINCT operations are expensive in large-scale joins.

Fix / Solution:

- Used approximate count function.

Code / Configuration:

```
SELECT approx_count_distinct(user_id) FROM table
```

Description (Resume): Improved query performance by applying approximate aggregation techniques for large-scale analytics.

Issue 44: Streaming Buffer Delay in BigQuery

Story: Real-time dashboard didn't show new data due to BigQuery streaming buffer delay.

Challenge: Streaming data not immediately visible in base table.

Fix / Solution:

- Query `_STREAMING_BUFFER` table to check pending data.

Code / Configuration:

```
SELECT * FROM dataset.table__STREAMING_BUFFER
```

Description (Resume): Enabled real-time monitoring of BigQuery ingestion by integrating buffer check logic into ETL pipeline.

Issue 45: Spark SQL Failed Due to Missing File Format Metadata

Story: Reading a Hive table failed because metadata pointed to a removed SerDe.

Challenge: Inconsistent SerDe and input format.

Fix / Solution:

- Re-created table with supported format (Parquet/ORC).

Code / Configuration:

```
CREATE TABLE new_table STORED AS PARQUET AS SELECT * FROM old_table
```

Description (Resume): Resolved Hive table compatibility issue by migrating to supported file formats for Spark SQL interoperability.

Issue 46: Duplicate Events in Streaming Due to At-Least-Once Semantics

Story: Replayed messages caused duplicate inserts in downstream sinks.

Challenge: Structured streaming retries cause duplication.

Fix / Solution:

- Deduplicate using watermark and event ID.

Code / Configuration:

```
df.withWatermark("event_time", "10 minutes").dropDuplicates(["event_id"])
```

Description (Resume): Enabled idempotent writes in streaming jobs by implementing watermark-based deduplication.

Issue 47: spark.read.csv() Failed on Missing Headers

Story: Schema mismatch occurred while reading a CSV without header row.

Challenge: Default behavior expects header.

Fix / Solution:

- Set `header=false` and define schema manually.

Code / Configuration:

```
spark.read.option("header", "false").schema(custom_schema).csv("path")
```

Description (Resume): Enabled controlled ingestion of CSV files in Spark by disabling header inference and using predefined schema.

Issue 48: Spark History Server Not Showing Completed Jobs

Story: Completed applications weren't visible in Spark History UI.

Challenge: Event logs weren't properly configured or saved.

Fix / Solution:

- Enabled Spark event logging.

Code / Configuration:

```
spark.eventLog.enabled=true  
spark.eventLog.dir=hdfs:///spark-logs
```

Description (Resume): Enabled Spark job auditing by configuring event logging and history server integration.

Issue 49: Delta Table Grows Rapidly Due to No VACUUM

Story: Delta Lake table size kept increasing even after overwrites.

Challenge: Old versions retained indefinitely.

Fix / Solution:

- Run VACUUM with retention policy.

Code / Configuration:

```
VACUUM delta_table RETAIN 168 HOURS
```

Description (Resume): Managed Delta Lake storage growth by scheduling automated VACUUM operations to clean stale data files.

Issue 50: Spark SQL Job Failed Due to Field Name Case Sensitivity

Story: A join failed because of mismatched column name casing (`UserID` vs `userid`).

Challenge: Column case mismatch led to unresolved reference.

Fix / Solution:

- Enabled strict case sensitivity or normalized columns.

Code / Configuration:

```
spark.sql.caseSensitive=true
```

Description (Resume): Prevented join errors in Spark SQL by enforcing consistent column casing and enabling case sensitivity during transformations.

Spark SQL Issues (51–60)

Issue 51: IN Clause Slowed Down Join Execution

Story: A report query using `WHERE id IN (...)` was extremely slow on large tables.

Challenge: IN clause creates list scanning and disables broadcast.

Fix / Solution:

- Replaced with a join using a small DataFrame.

Code / Configuration:

```
broadcast_df = spark.createDataFrame([1,2,3], "int").toDF("id")
main_df.join(broadcast(broadcast_df), "id")
```

Description (Resume): Optimized Spark SQL by rewriting IN clause to broadcast join for performance boost.

Issue 52: CAST Failed When Column Contained Nulls

Gowtham SB

Linkedin - <https://www.linkedin.com/in/sbgowtham/>

Story: A CAST to decimal failed due to presence of nulls in the column.

Challenge: Improper casting causes job failure.

Fix / Solution:

- Used `when/otherwise` to handle nulls safely.

Code / Configuration:

```
from pyspark.sql.functions import when
new_df = df.withColumn("amount", when(df.amount.isNotNull(),
df.amount.cast("decimal(10,2)").otherwise(0.0))
```

Description (Resume): Ensured stable Spark transformations by applying null-safe casting logic for numeric columns.

Issue 53: Caching Didn't Improve Performance

Story: `.cache()` had no effect — job still took same amount of time.

Challenge: Cache wasn't triggered by any action.

Fix / Solution:

- Used `.count()` or `.show()` after `.cache()`

Code / Configuration:

```
df.cache()
df.count() # Action to materialize cache
```

Description (Resume): Improved Spark query performance by ensuring proper cache materialization and memory reuse.

Issue 54: Using UDF Blocked Catalyst Optimizer

Story: A Python UDF disabled optimizations and led to slower queries.

Gowtham SB

Linkedin - <https://www.linkedin.com/in/sbgowtham/>

Challenge: UDFs break query planning and pushdown.

Fix / Solution:

- Rewrote logic using built-in functions

Code / Configuration:

```
from pyspark.sql.functions import regexp_extract
cleaned_df = df.withColumn("domain", regexp_extract("email", "@(.+)$", 1))
```

Description (Resume): Enhanced Spark SQL optimization by replacing UDFs with native Catalyst-compatible functions.

Issue 55: Overwrite on Partitioned Table Deleted All Data

Story: Intended to overwrite one partition, but the full table got deleted.

Challenge: Default overwrite mode unsafe for partitioned writes.

Fix / Solution:

- Enabled dynamic partition overwrite.

Code / Configuration:

```
spark.sql.sources.partitionOverwriteMode=dynamic
```

Description (Resume): Prevented data loss in Spark table overwrites by enabling dynamic partition-aware write mode.

Issue 56: Slow Performance on COUNT Aggregations

Story: Spark job was bottlenecked on simple `COUNT(*)` aggregations.

Challenge: Lack of statistics or skewed data.

Fix / Solution:

- Repartitioned and used pre-aggregated tables.

Code / Configuration:

```
df.repartition(50).groupBy("status").count()
```

Description (Resume): Optimized groupBy aggregations in Spark SQL by repartitioning and applying data summarization techniques.

Issue 57: Case When Logic Failed for Null Conditions

Story: CASE WHEN logic failed to capture NULLs properly.

Challenge: Nulls require explicit checks in Spark SQL.

Fix / Solution:

- Used `isNull()` function in WHEN clause.

Code / Configuration:

```
CASE WHEN col IS NULL THEN 'Unknown' ELSE col END
```

Description (Resume): Increased query reliability by handling null branches explicitly in Spark SQL CASE WHEN logic.

Issue 58: Performance Issue in GROUP BY Multiple Columns

Story: GROUP BY on 5+ columns slowed down the entire job.

Challenge: High cardinality groupings create large shuffle.

Fix / Solution:

- Reduced number of grouping columns
- Added bucketing for frequent aggregations

Code / Configuration:

GROUP BY region, event_date

Description (Resume): Improved aggregation performance by restructuring Spark SQL groupings for better scalability.

Issue 59: DATE_TRUNC Failed Due to Inconsistent Input Types

Story: A job using `DATE_TRUNC` failed when fed a timestamp instead of date.

Challenge: Incompatible function input.

Fix / Solution:

- Cast timestamp to date before truncation.

Code / Configuration:

`DATE_TRUNC('MONTH', CAST(event_time AS DATE))`

Description (Resume): Ensured consistent date manipulation by type casting before applying Spark SQL date functions.

Issue 60: Temp View Overwritten Silently in Session

Story: A temp view got silently replaced mid-pipeline causing wrong results.

Challenge: No isolation or tracking of temp view updates.

Fix / Solution:

- Used meaningful unique view names per transformation stage

Code / Configuration:

`df.createOrReplaceTempView("cleaned_stage1")`

Description (Resume): Improved pipeline reliability by using distinct view names and avoiding accidental view overwrites in Spark sessions.

Spark Streaming Issues (61–70)

Issue 61: Streaming Job Fails Without Checkpointing

Story: After a driver crash, the job restarted from scratch and reprocessed all data.

Challenge: Lack of checkpointing leads to no fault tolerance.

Fix / Solution:

- Configure a persistent checkpoint directory.

Code / Configuration:

```
query.writeStream.option("checkpointLocation", "/path/to/checkpoint").start()
```

Description (Resume): Enabled fault-tolerant Spark Streaming by integrating durable checkpointing mechanisms.

Issue 62: Kafka Offsets Not Committed After Failure

Story: Spark Streaming job crashed mid-batch and reprocessed messages after restart.

Challenge: Offset commits were not tied to processing completion.

Fix / Solution:

- Enabled checkpointing and ensured `awaitTermination()`

Code / Configuration:

```
query.awaitTermination()
```

Description (Resume): Achieved reliable Kafka streaming consumption in Spark by handling offset commits through checkpointing.

Issue 63: Watermarking Didn't Filter Late Data

Story: Spark processed events from 3 hours ago, breaking real-time SLA.

Challenge: Watermark logic was missing.

Fix / Solution:

- Applied watermark on event_time column.

Code / Configuration:

```
df.withWatermark("event_time", "10 minutes")
```

Description (Resume): Enhanced streaming timeliness by introducing watermark-based event time filtering in Spark.

Issue 64: Structured Streaming Writes Were Duplicated

Story: Downstream system received duplicate rows.

Challenge: Streaming retries caused idempotency failure.

Fix / Solution:

- Applied `dropDuplicates()` on a unique identifier.

Code / Configuration:

```
df.dropDuplicates(["event_id"])
```

Description (Resume): Implemented deduplication strategy in Spark Structured Streaming to ensure idempotent sink writes.

Issue 65: Trigger Once Didn't Work as Expected

Story: Expected single micro-batch execution, but job kept running.

Gowtham SB

Linkedin - <https://www.linkedin.com/in/sbgowtham/>

Challenge: Trigger settings not configured correctly.

Fix / Solution:

- Used correct `.trigger(once=True)` API.

Code / Configuration:

```
query.writeStream.trigger(once=True).start()
```

Description (Resume): Designed micro-batch Spark jobs using trigger-once mode for efficient incremental batch processing.

Issue 66: Streaming Job Lag Due to Slow Sinks

Story: Sink to PostgreSQL couldn't keep up, causing processing delays.

Challenge: Blocking writes slowed microbatch processing.

Fix / Solution:

- Wrote to staging files and used async write thread.

Code / Configuration:

```
df.writeStream.format("parquet")...
```

Description (Resume): Improved Spark Streaming throughput by decoupling heavy sinks with file staging strategy.

Issue 67: Streaming Job Failed on Output File Collision

Story: A streaming job wrote to a static path, and multiple writes collided.

Challenge: Same checkpoint + output paths cause corruption.

Fix / Solution:

- Used unique paths with partitioning or timestamp-based output dirs.

Code / Configuration:

```
path = f"/output/stream_{datetime.now()}"
```

Description (Resume): Prevented file write conflicts in Spark Streaming by using timestamped output directories.

Issue 68: Memory Leak from Stateful Aggregations

Story: Long-running job caused GC overhead due to state build-up.

Challenge: No TTL (time-to-live) on streaming state.

Fix / Solution:

- Configured watermark + timeout.

Code / Configuration:

```
.groupBy(window("event_time", "10 mins")).withWatermark("event_time", "5 minutes")
```

Description (Resume): Managed memory growth in streaming apps by configuring state TTL and periodic cleanup.

Issue 69: Unbounded Kafka Stream Filled Memory

Story: Spark job reading from Kafka with no bounds consumed unthrottled data.

Challenge: No control over ingestion rate.

Fix / Solution:

- Set max rate per partition.

Code / Configuration:

```
spark.streaming.kafka.maxRatePerPartition=500
```

Gowtham SB

Linkedin - <https://www.linkedin.com/in/sbgowtham/>

Description (Resume): Added backpressure control in Kafka-Spark streaming jobs for safe and scalable data ingestion.

Issue 70: Streaming Joins Skipped Matches Due to Watermark Timing

Story: Inner joins between two streams lost rows due to late watermark.

Challenge: Join timing mismatch dropped valid rows.

Fix / Solution:

- Aligned watermark logic and extended retention window.

Code / Configuration:

```
left.withWatermark("time", "10 mins").join(right.withWatermark("time", "10 mins"))
```

Description (Resume): Improved completeness of Spark streaming joins by aligning watermark policies across streams.

Kafka Issues (71–80)

Issue 71: Kafka Consumer Lag Never Reduced

Story: The consumer group always showed increasing lag even when there were no errors.

Challenge: Message processing was slower than ingestion rate.

Fix / Solution:

- Increased consumer parallelism and `max.poll.records`

Code / Configuration:

```
max.poll.records=1000  
fetch.min.bytes=1
```

Description (Resume): Improved Kafka consumer performance by tuning poll size and adding multi-threaded consumption logic.

Issue 72: Kafka Producer Failed With Buffer Exhausted Error

Story: High-throughput producer crashed with `BufferExhaustedException`.

Challenge: The producer buffer filled up before it could flush.

Fix / Solution:

- Increased buffer and batch size; added linger

Code / Configuration:

```
buffer.memory=67108864  
batch.size=65536  
linger.ms=20
```

Description (Resume): Scaled Kafka producer throughput by optimizing memory buffer and batching configs.

Issue 73: Duplicate Messages After Restart

Story: Consumers reprocessed messages after failure.

Challenge: Offset was committed before processing finished.

Fix / Solution:

- Manually commit offset after successful processing

Code / Configuration:

```
consumer.commitSync();
```

Description (Resume): Achieved reliable processing in Kafka by ensuring commit offset occurs only after processing success.

Issue 74: Topic Partition Imbalance Across Consumers

Gowtham SB

Linkedin - <https://www.linkedin.com/in/sbgowtham/>

Story: One consumer was overloaded while others were idle.

Challenge: Uneven partition assignment.

Fix / Solution:

- Increased partitions and enabled **StickyAssignor**

Code / Configuration:

```
partition.assignment.strategy=org.apache.kafka.clients.consumer.StickyAssignor
```

Description (Resume): Balanced load across Kafka consumers by increasing partitions and tuning assignment strategy.

Issue 75: Kafka Messages Lost Due to Retention Settings

Story: Messages were deleted before they could be consumed.

Challenge: Short **retention.ms** and slow consumer lag.

Fix / Solution:

- Increased topic retention time

Code / Configuration:

```
kafka-configs.sh --alter --topic my-topic --add-config retention.ms=86400000
```

Description (Resume): Prevented message loss in Kafka by adjusting topic-level retention to handle slow consumers.

Issue 76: High CPU Usage on Kafka Brokers

Story: Kafka brokers spiked to 90–100% CPU on ingestion burst.

Challenge: I/O and compression bottleneck.

Fix / Solution:

- Tuned socket buffers and enabled compression

Code / Configuration:

```
compression.type=snappy  
socket.send.buffer.bytes=102400
```

Description (Resume): Optimized Kafka broker performance during high-load bursts through compression and I/O tuning.

Issue 77: Kafka Streams Application Didn't Scale

Story: Kafka Streams topology handled only 1 partition and couldn't scale.

Challenge: Source topic had too few partitions.

Fix / Solution:

- Increased topic partitions to match parallelism

Code / Configuration:

```
kafka-topics.sh --alter --topic my-topic --partitions 12
```

Description (Resume): Enabled parallelism in Kafka Streams by matching topic partitioning with thread count.

Issue 78: Delayed Log Compaction Caused Old Data

Story: Compact topics retained stale key-value data longer than needed.

Challenge: Log compaction wasn't aggressive enough.

Fix / Solution:

- Tuned compaction settings

Code / Configuration:

Gowtham SB

Linkedin - <https://www.linkedin.com/in/sbgowtham/>

min.cleanable.dirty.ratio=0.1

log.cleaner.enable=true

Description (Resume): Accelerated Kafka log compaction cycles by tuning cleaner thresholds for up-to-date data.

Issue 79: Kafka Security TLS Failed in Multi-Region Setup

Story: Kafka client failed to connect cross-region with TLS enabled.

Challenge: Incomplete certificate chain or wrong DNS config.

Fix / Solution:

- Set endpoint identification and imported CA properly

Code / Configuration:

```
ssl.endpoint.identification.algorithm=  
security.protocol=SASL_SSL
```

Description (Resume): Secured Kafka multi-region connections by configuring TLS properties and certificate trust chain.

Issue 80: Kafka Topic Grew Uncontrollably Due to Null Keys

Story: A null key in producer caused no compaction and high disk usage.

Challenge: Log compaction skipped null-keyed records.

Fix / Solution:

- Enforced key generation on all producer messages

Code / Configuration:

```
producer.send(new ProducerRecord<>("topic", generatedKey, value))
```

Gowtham SB

Linkedin - <https://www.linkedin.com/in/sbgowtham/>

Description (Resume): Improved Kafka data retention by ensuring all messages had keys to support log compaction.

BigQuery Issues (81–90)

Issue 81: SELECT * Caused Expensive Query Costs

Story: Developer used `SELECT *` on a wide table which scanned multiple TBs of data.

Challenge: Scans unneeded columns and spikes cost.

Fix / Solution:

- Select only required columns

Code / Configuration:

```
SELECT id, name FROM project.dataset.table WHERE event_date = '2024-01-01'
```

Description (Resume): Reduced BigQuery costs by implementing column projection and avoiding wide table scans.

Issue 82: Partition Filter Not Used – Full Table Scan

Story: Query over partitioned table still scanned the entire dataset.

Challenge: Applied function on partition column in WHERE clause.

Fix / Solution:

- Use partition column directly in condition

Code / Configuration:

```
WHERE event_date = '2024-01-01'
```

Description (Resume): Improved BigQuery performance by using direct partition filtering to enable partition pruning.

Issue 83: Cross Join Triggered Unexpectedly

Story: JOIN without ON clause created billions of rows unexpectedly.

Challenge: Unintentional Cartesian product.

Fix / Solution:

- Always use proper ON clause with JOIN

Code / Configuration:

```
SELECT * FROM a JOIN b ON a.id = b.id
```

Description (Resume): Prevented data explosion in BigQuery by applying strict join keys to avoid Cartesian joins.

Issue 84: COUNT(DISTINCT) Very Slow

Story: A query with COUNT(DISTINCT) took 30+ mins on large dataset.

Challenge: DISTINCT on massive records is expensive.

Fix / Solution:

- Use APPROX_COUNT_DISTINCT

Code / Configuration:

```
SELECT APPROX_COUNT_DISTINCT(user_id) FROM project.dataset.table
```

Description (Resume): Optimized BigQuery aggregations by applying approximate distinct functions for speed.

Issue 85: Materialized View Not Updating

Story: Materialized view didn't show recent data after table updates.

Gowtham SB

Linkedin - <https://www.linkedin.com/in/sbgowtham/>

Challenge: Materialized views are only refreshed when source table is updated.

Fix / Solution:

- Use scheduled queries or replace with manual refresh

Code / Configuration:

REFRESH MATERIALIZED VIEW project.dataset.view_name

Description (Resume): Maintained up-to-date reporting in BigQuery by scheduling materialized view refresh jobs.

Issue 86: Streaming Buffer Data Not Visible in Query

Story: Recent data from Pub/Sub was not reflected in live dashboards.

Challenge: Streaming buffer data is delayed in visibility.

Fix / Solution:

- Check buffer table and alert if lag crosses threshold

Code / Configuration:

SELECT * FROM dataset.table__STREAMING_BUFFER

Description (Resume): Ensured data completeness in BigQuery streaming ingestion by monitoring buffer table status.

Issue 87: Temporary Tables Deleted Prematurely

Story: Team used temp table for joins, but it disappeared mid-pipeline.

Challenge: Temporary tables last only during session.

Fix / Solution:

- Use persistent staging tables with expiration policy

Code / Configuration:

```
CREATE TABLE project.dataset.temp_table  
OPTIONS(expiration_timestamp=TIMESTAMP_ADD(CURRENT_TIMESTAMP(), INTERVAL 1  
HOUR)) AS SELECT ...
```

Description (Resume): Built robust pipelines in BigQuery by replacing temp tables with time-bound persistent tables.

Issue 88: BigQuery Join Failed Due to Mismatched Types

Story: Join failed because of INT vs STRING key mismatch.

Challenge: Schema mismatch between sources.

Fix / Solution:

- Cast keys explicitly before join

Code / Configuration:

```
SELECT * FROM a JOIN b ON CAST(a.id AS STRING) = b.id
```

Description (Resume): Fixed schema mismatch issues in BigQuery joins using explicit casting to align data types.

Issue 89: Data Duplication in Append-Only Tables

Story: Scheduled jobs reprocessed and inserted duplicate records.

Challenge: No deduplication strategy.

Fix / Solution:

- Use **MERGE** with unique keys or **ROW_NUMBER** filtering

Code / Configuration:

```
MERGE target USING staging ON target.id = staging.id
```

WHEN NOT MATCHED THEN INSERT ROW

Description (Resume): Prevented duplicate inserts in BigQuery by implementing MERGE logic based on unique keys.

Issue 90: Table Scan Too Expensive in BI Tools

Story: Dashboarding tool triggered full scan every time.

Challenge: Poor design with SELECT * + no filters

Fix / Solution:

- Used pre-aggregated summary tables

Code / Configuration:

```
CREATE TABLE dataset.summary AS SELECT region, COUNT(*) FROM raw_table GROUP BY region
```

Description (Resume): Reduced BI dashboard costs in BigQuery by creating pre-aggregated summary layers.

Airflow Issues (91–100)

Issue 91: DAG Failed Due to Missing Package in Worker

Story: DAG ran fine locally but failed on Airflow server due to missing Python dependency.

Challenge: Workers lacked required pip packages.

Fix / Solution:

- Installed required packages using requirements.txt or custom Docker image.

Code / Configuration:

```
pip install -r requirements.txt
```

Description (Resume): Improved DAG portability by containerizing Airflow dependencies and managing Python packages centrally.

Issue 92: Tasks Not Retrying After Failure

Story: One task failed due to network glitch and didn't retry.

Challenge: DAG was not configured with retries.

Fix / Solution:

- Added retry logic and delay.

Code / Configuration:

```
default_args = {  
    'retries': 3,  
    'retry_delay': timedelta(minutes=5)  
}
```

Description (Resume): Increased Airflow task resilience by configuring automatic retries for transient failures.

Issue 93: Task Execution Overlapping Caused Data Corruption

Story: A scheduled DAG started before previous run completed.

Challenge: Concurrency control missing.

Fix / Solution:

- Set `max_active_runs` to 1

Code / Configuration:

```
DAG(..., max_active_runs=1)
```

Description (Resume): Ensured data integrity in Airflow DAGs by limiting concurrent executions.

Issue 94: XCom Data Too Large for Backend

Story: Pushed a large Pandas DataFrame into XCom, causing serialization failure.

Challenge: XCom size limit exceeded.

Fix / Solution:

- Saved data to S3/GCS and passed path in XCom

Code / Configuration:

```
xcom_push(key='path', value='s3://bucket/output.csv')
```

Description (Resume): Avoided Airflow XCom limitations by externalizing large data references to cloud storage.

Issue 95: DAG Not Triggering Due to Schedule Misalignment

Story: DAG expected hourly runs but only executed once daily.

Challenge: Incorrect `schedule_interval` setting.

Fix / Solution:

- Updated schedule using cron syntax

Code / Configuration:

```
schedule_interval='0 * * * *'
```

Description (Resume): Fixed Airflow scheduling issues by aligning DAG run intervals with business requirements.

Issue 96: Parallel Tasks Failed Due to Shared Temp File

Gowtham SB

Linkedin - <https://www.linkedin.com/in/sbgowtham/>

Story: Two parallel tasks overwrote each other's temp output file.

Challenge: Shared temp location caused conflict.

Fix / Solution:

- Used unique file names per task instance

Code / Configuration:

```
filename = f"/tmp/output_{ds_nodash}_{task_instance.task_id}.csv"
```

Description (Resume): Enabled safe parallel execution in Airflow by isolating file outputs across tasks.

Issue 97: DAGs Not Visible After Uploading to DAGS Folder

Story: Newly added DAGs didn't show up in Airflow UI.

Challenge: Syntax error or missing DAG object in file.

Fix / Solution:

- Verified DAG object instantiation and logs

Code / Configuration:

```
dag = DAG('my_dag', ...) # Must be declared
```

Description (Resume): Ensured DAG discoverability in Airflow by validating DAG object definitions and Python syntax.

Issue 98: Airflow Webserver Crashed on Large Log Files

Story: DAG logs grew too large and crashed the web UI.

Challenge: Unbounded log file size.

Fix / Solution:

- Configured log rotation and retention

Code / Configuration:

```
[logging]
max_log_file_size = 5MB
log_filename_template = {{ dag_id }}/{{ task_id }}/{{ execution_date }}.log
```

Description (Resume): Stabilized Airflow UI by implementing log rotation and retention policy for task logs.

Issue 99: Sensor Task Stuck for Hours

Story: External sensor ran indefinitely, holding up DAG progress.

Challenge: No timeout or poke interval set.

Fix / Solution:

- Added timeout and interval

Code / Configuration:

```
ExternalTaskSensor(..., timeout=600, poke_interval=30)
```

Description (Resume): Increased pipeline efficiency by configuring proper timeouts for long-running Airflow sensor tasks.

Issue 100: BashOperator Command Failed Without Logging

Story: BashOperator task failed silently due to missing `set -e`

Challenge: Exit codes ignored in shell script.

Fix / Solution:

- Prefixed bash script with `set -e`

Code / Configuration:

```
bash_command="set -e; ./my_script.sh"
```

Description (Resume): Improved Airflow error detection by enforcing strict command execution using `set -e` in BashOperator.

Gowtham SB

Linkedin - <https://www.linkedin.com/in/sbgowtham/>

About the Author

Gowtham SB is a **Data Engineering Expert, Educator, and Content Creator** with a passion for **Big Data technologies**. With years of experience in the field, he has worked extensively with **cloud platforms, distributed systems, and data pipelines**, helping professionals and aspiring engineers master the art of data engineering.

Beyond his technical expertise, Gowtham is a **renowned mentor and speaker**, sharing his insights through engaging content on **YouTube and LinkedIn**. He has built one of the **largest Tamil Data Engineering communities**, guiding thousands of learners to excel in their careers.

Through his deep industry knowledge and hands-on approach, Gowtham continues to **bridge the gap between learning and real-world implementation**, empowering individuals to build **scalable, high-performance data solutions**.

Socials

 **YouTube** - <https://www.youtube.com/@dataengineeringvideos>

 **Instagram** - <https://instagram.com/dataengineeringtamil>


 **Instagram** - <https://instagram.com/thedatatech.in>

 **Connect for 1:1** - <https://topmate.io/dataengineering/>

 **LinkedIn** - <https://www.linkedin.com/in/sbgowtham/>

 **Website** - <https://codewithgowtham.blogspot.com>

 **GitHub** - <http://github.com/Gowthamdataengineer>

 **Whats App** - <https://lnkd.in/g5JrHw8q>

 **Email** - atozknowledge.com@gmail.com

 **All My Socials** - <https://lnkd.in/gf8k3aCH>