Let’s dumb-proof data pipelines

Ganesh Chand
Staff Solutions Consultant, Databricks

databricks

Why make your data pipelines dumb-proof?

User error can cause production outages

- Missed SLAs
- Downstream impact
- Business impacts
- Data Decay

Common causes of ETL accidents due to human error

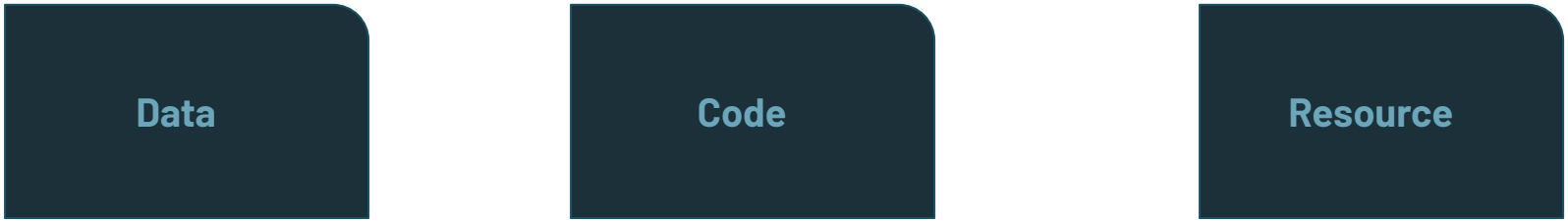
- Configuration error
- Resource misuse
- Operator error
- Defects

Typical Situations

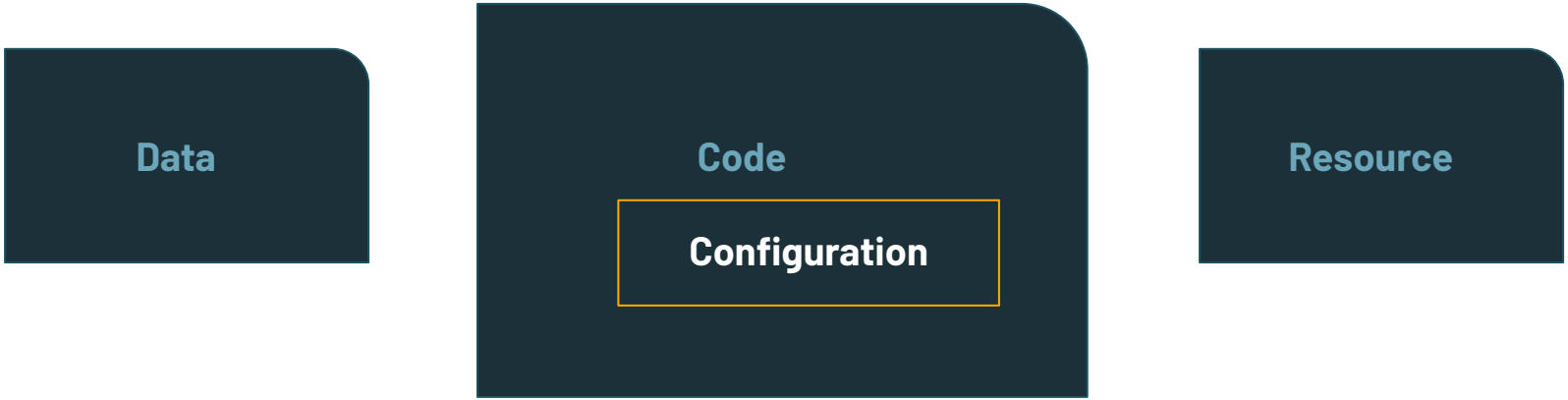
- Environment changes
- Emergency fixes
- Regular maintenance

Secure your loads and double check!

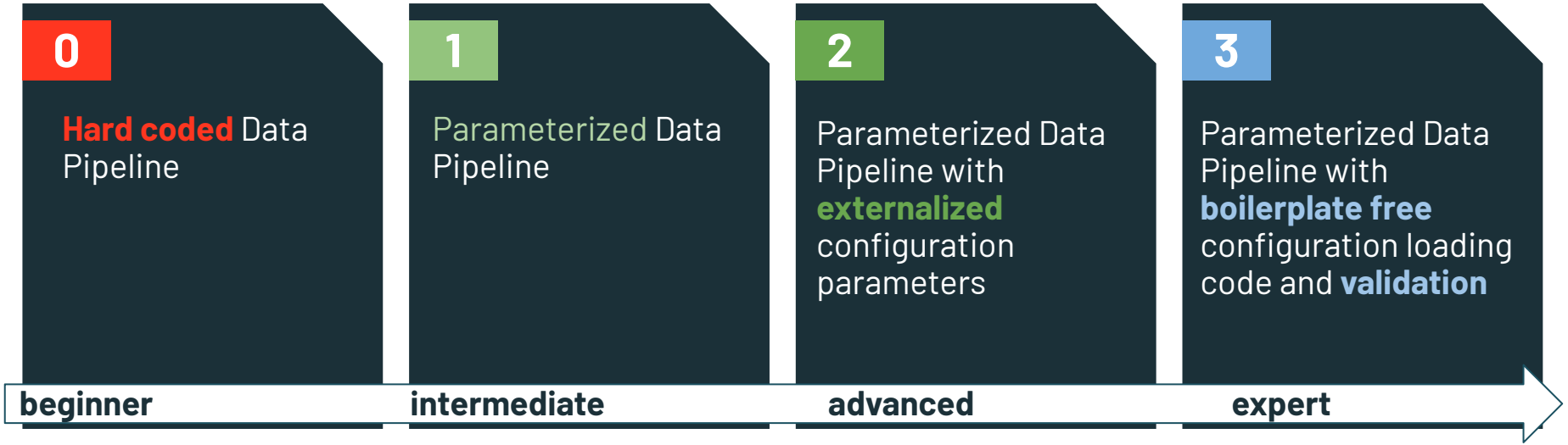
How to make your data pipelines dumb-proof?



How to make your data pipelines dumb-proof?



How to make your data pipelines dumb-proof?



Hard coded Data Pipelines

```
val input = spark.readStream
  .format("rate")
  .option("rowsPerSecond", 1)
  .load()

spark.conf.set("spark.sql.shuffle.partitions", 2)
spark.conf.set("spark.databricks.delta.properties.defaults.appendOnly," "true")

val output = input
 // add audit columns
  .withColumn("etl_insert_timestamp", current_timestamp)
  .withColumn("event_date", to_date('timestamp))

output.writeStream.format('delta")
  .option("mergeSchema", "true")
  .option("checkpointLocation", "/tmp/deltalake/dumbproof/bronze/rate/0/_checkpoint)"
  .partitionBy("event_date")
  .outputMode("append")
  .trigger(ProcessingTime('30 seconds"))
  .start("/tmp/deltalake/dumbproof/bronze/rate/0)'
```

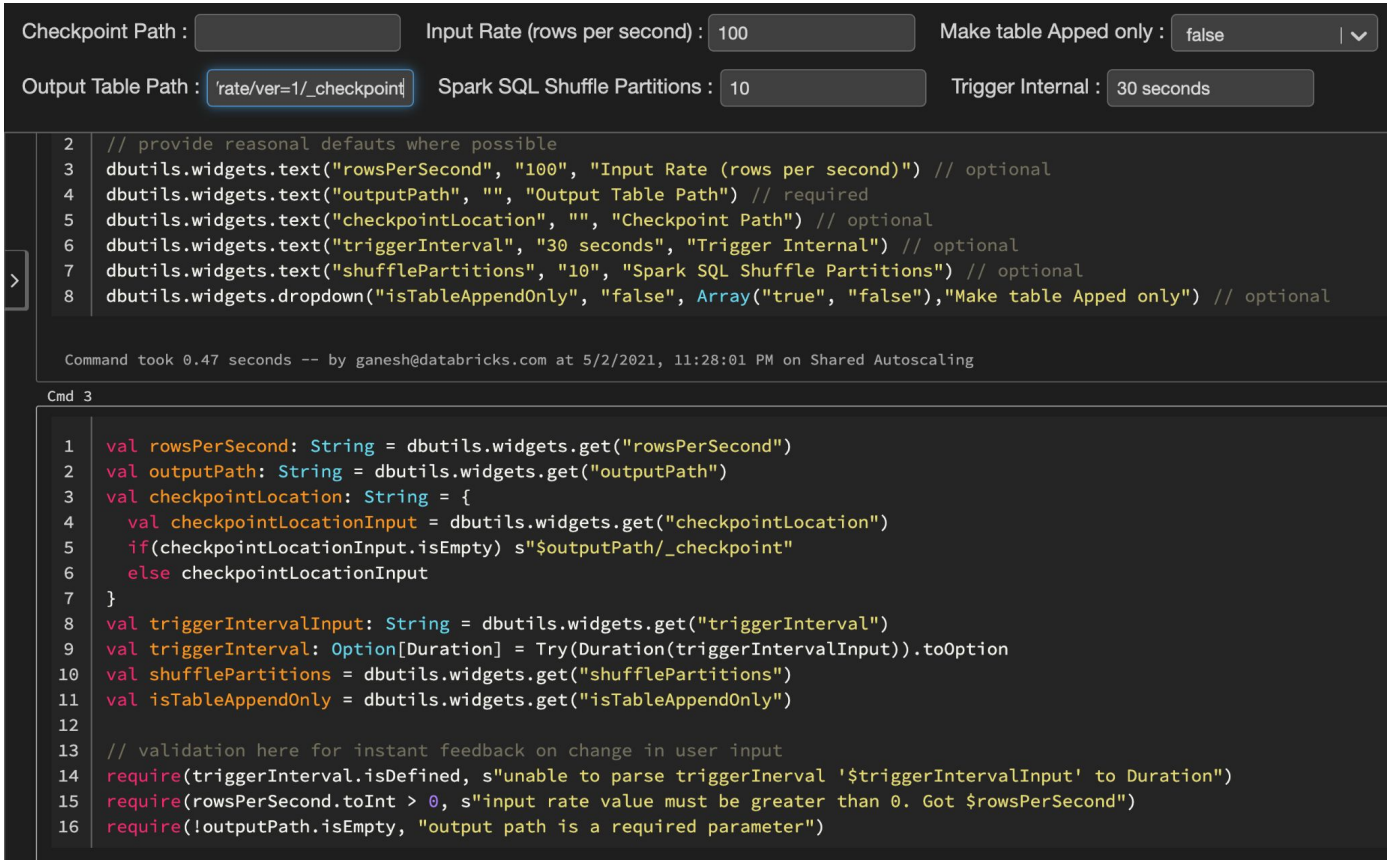
- Environment changes
- User error

Hard coding is the software development practice of embedding data directly into the source code of a program or other executable object, as opposed to obtaining the data from external sources or generating it at runtime. Wikipedia

Fixing Hard coded Data Pipelines

- **Parameterize** values that change across environments
- **Validate** user inputs
- Refactor code

Parameters & Input Validation



Define Parameters

Process User Inputs

Validate & Fail on invalid input

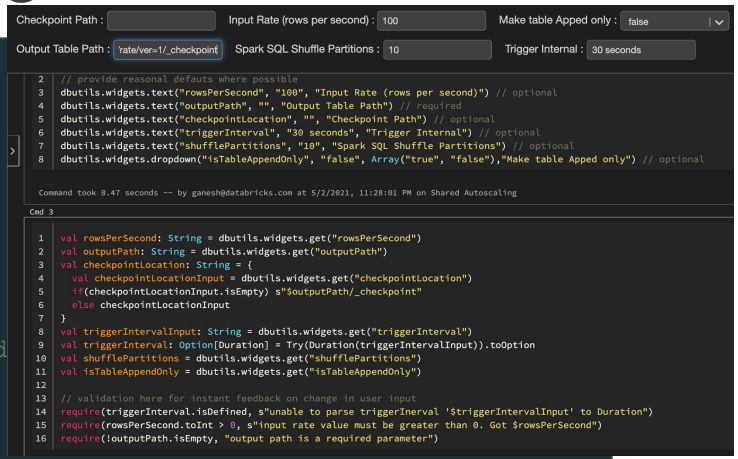
Parameterized Data Pipeline

```
val input = spark.readStream
  .format("rate")
  .option("rowsPerSecond", rowsPerSecond)
  .load()


spark.conf.set("spark.sql.shuffle.partitions", shufflePartitions)
spark.conf.set("spark.databricks.delta.properties.defaults.append

val output = input
    // add audit columns
    .withColumn("etl_insert_timestamp", current_timestamp)
    .withColumn("event_date", to_date('timestamp))


val writer = output.writeStream.format("delta")
  .queryName("ingest_rateSource_into_delta")
  .option("mergeSchema", "true")
  .option("checkpointLocation", checkpointLocation)
  .partitionBy("event_date")
  .outputMode("append")
  .trigger(ProcessingTime(triggerInterval.get))
  .start(outputPath)
```



- ✓ Environment changes
- ✓ User error

- ❑ Input parameters > 10
- ❑ Type casting parameter value
- ❑ Tracking parameter changes

databricks

Externalizing Configuration

- Choose configuration Format
- Configuration Library
- Refactor Code

Configuration Format

Many Options

- ❑ ENV VARIABLE
- ❑ INI
- ❑ XML
- ❑ JSON
- ❑ PROPERTIES
- ❑ HOCCON
- ❑ YAML
- ❑ TOML

Why not JSON (JavaScript Object Notation)

JSON isn't a Good Configuration Language

- Verbose
- Lack of comments
- Lack of multiline support
- Lack of substitutions and includes

Configuration Format

Many Options

- ❑ ENV VARIABLE
- ❑ INI
- ❑ XML
- ❑ JSON
- ❑ PROPERTIES
- ✓ **HOCON**
- ❑ YAML
- ❑ TOML

Why HOCON (Human-Optimized Config Object Notation)

HOCON is a purpose-built Configuration format

- Concise
- Supports comments
- Supports multiline String
- Supports substitutions and includes
- a JSON superset
- Python/Java/Scala libraries

Official HOCON Spec: https://github.com/lightbend/config/blob/master/HOCON.md

Configuration in **JSON** Format

```
{
 "dev": {
   "outputPath": "/tmp/deltalake/dev/sample/1/rate",
   "checkpointPath": "/tmp/deltalake/dev/sample/1/rate/_checkpoint",
   "rowsPerSecond": 1,
   "triggerInterval": "30s",
   "tuning": [
     "spark.sql.shuffle.partitions=10"
   ]
 },
 "prod": {
   "outputPath": "/tmp/deltalake/prod/sample/1/rate",
   "checkpointPath": "/tmp/deltalake/prod/sample/1/rate/_checkpoint",
   "rowsPerSecond": 1000,
   "triggerInterval": "1m",
   "tuning": [
     "spark.sql.shuffle.partitions=10"
   ]
 }
}
```

Configuration in **HOCON** format

```
// iteration 1: removed unwanted curly braces, colons and double quotes

dev {
 outputPath: /tmp/deltalake/dev/sample/1/rate
 checkpointPath: /tmp/deltalake/dev/sample/1/rate/_checkpoint
 rowsPerSecond: 1,
 triggerInterval: 30s,
 tuning: ["spark.sql.shuffle.partitions=10"]
},

prod {
 outputPath: /tmp/deltalake/prod/sample/1/rate
 checkpointPath: /tmp/deltalake/prod/sample/1/rate/_checkpoint
 rowsPerSecond: 1000,
 triggerInterval: 1m,
 tuning: ["spark.sql.shuffle.partitions=10"]
}
```

Optimized Configuration in **HOCON** format

```
// iteration 1: removed unwanted curly braces, colons and double quotes
// iteration 2: extracted common values to common config object and checkpoint path is set to relative to output path

common {
 rootPath = /tmp/deltalake
 tuning = ["spark.sql.shuffle.partitions=10"]
}

dev {
 outputPath: ${common.rootPath}/dev/sample/1/rate,
 checkpointPath: ${dev.outputPath}_checkpoint,
 rowsPerSecond: 1,
 triggerInterval: 30s,
 tuning: ${common.tuning}
},

prod {
 outputPath: ${common.rootPath}/prod/sample/1/rate,
 checkpointPath: ${prod.outputPath}/_checkpoint,
 rowsPerSecond: 1000,
 triggerInterval: 1m,
 tuning: ${common.tuning}
}
```

Readable and maintainable Configuration

```
{
  "dev": {
    "outputPath": "/tmp/deltalake/dev/sample/1/rate",
    "checkpointPath": "/tmp/deltalake/dev/sample/1/rate/_checkpoint",
    "rowsPerSecond": 1,
    "triggerInterval": "30s",
    "tuning": [
      "spark.sql.shuffle.partitions=10"
    ]
  },
  "prod": {
    "outputPath": "/tmp/deltalake/prod/sample/1/rate",
    "checkpointPath": "/tmp/deltalake/prod/sample/1/rate/_checkpoint",
    "rowsPerSecond": 1000,
    "triggerInterval": "1m",
    "tuning": [
      "spark.sql.shuffle.partitions=10"
    ]
  }
}
```

```
common {
  rootPath = /tmp/deltalake
  tuning = ["spark.sql.shuffle.partitions=10"]
}

dev {
  outputPath: ${common.rootPath}/dev/sample/1/rate,
  checkpointPath: ${dev.outputPath}_checkpoint,
  rowsPerSecond: 1,
  triggerInterval: 30s,
  tuning: ${common.tuning}
},

prod {
  outputPath: ${common.rootPath}/prod/sample/1/rate,
  checkpointPath: ${prod.outputPath}/_checkpoint,
  rowsPerSecond: 1000,
  triggerInterval: 1m,
  tuning: ${common.tuning}
}
```

Configuration Library

Typesafe - Configuration library for JVM languages by Lightbend

- Implemented in plain Java with **zero** dependencies
- Supports files in three formats: Java properties, JSON, and a human-friendly JSON superset (HOCON)
- Included in Databricks Runtime

```
import com.typesafe.config._

val conf: Config =
ConfigFactory.parseString(dbutils.fs.head("dbfs:/path/to/data-pipeline.conf")).resolve

val rowsPerSecond: Long = conf.getLong("rowsPerSecond")
val outputPath: String = conf.getString("outputPath")
val checkpointLocation: String = conf.getString("checkpointPath")
val triggerInterval: ScalaDuration =ScalaDuration(conf.getString("triggerInterval"))
val sparkTuningConfs: Seq[String] = conf.getStringList("tuning").asScala
```

Official Typesafe Config Docs:https://github.com/lightbend/config

Refactor Code - Loading and Parsing Configuration

Environment : dev Pipeline Config file : /dumbproof/deltalake/c

```
1  dbutils.widgets.removeAll()
2  dbutils.widgets.text("env", "dev", "Environment")
3  dbutils.widgets.text("configFilePath", "", "Pipeline Config file")
4
5  val env = dbutils.widgets.get("env")
6  val configFilePath = dbutils.widgets.get("configFilePath")
7
8
9  import com.typesafe.config._
10 val conf = ConfigFactory.parseString(dbutils.fs.head(configFilePath)).resolve.getConfig(env)
11
12
13 val rowsPerSecond: Long = conf.getLong("rowsPerSecond")
14 val outputPath: String = conf.getString("outputPath")
15 val checkpointLocation: String = conf.getString("checkpointPath")
16 val triggerInterval: ScalaDuration = ScalaDuration(conf.getString("triggerInterval"))
17 val sparkTuningConfs: Seq[String] = conf.getStringList("tuning").asScala
18 require(rowsPerSecond.toInt > 0, s"input rate value must be greater than 0. Got $rowsPerSecond")
```

Parameterized Data Pipeline

```
val input = spark.readStream
  .format("rate")
  .option("rowsPerSecond", rowsPerSecond)
  .load()

sparkTuningConfs
   .foreach(config => spark.sql(s"SET $config"))


val output = input
    // add audit columns
    .withColumn("etl_insert_timestamp", current_timestamp)
    .withColumn("event_date", to_date('timestamp))

val writer = output.writeStream.format("delta")
  .queryName("ingest_rateSource_into_delta")
  .option("mergeSchema", "true")
  .option("checkpointLocation", checkpointLocation)
  .partitionBy("event_date")
  .outputMode("append")
  .trigger(ProcessingTime(triggerInterval))
  .start(outputPath)
```

Environment : dev Pipeline Config file : /dumbproof/deltalake/c

```
1  dbutils.widgets.removeAll()
2  dbutils.widgets.text("env", "dev", "Environment")
3  dbutils.widgets.text("configFilePath", "", "Pipeline Config file")
4
5  val env = dbutils.widgets.get("env")
6  val configFilePath = dbutils.widgets.get("configFilePath")
7
8
9  import com.typesafe.config._
10 val conf = ConfigFactory.parseString(dbutils.fs.head(configFilePath)).resolve.getConfig(env)
11
12
13 val rowsPerSecond: Long = conf.getLong("rowsPerSecond")
14 val outputPath: String = conf.getString("outputPath")
15 val checkpointLocation: String = conf.getString("checkpointPath")
16 val triggerInterval: ScalaDuration = ScalaDuration(conf.getString("triggerInterval"))
17 val sparkTuningConfs: Seq[String] = conf.getStringList("tuning").asScala
18 require(rowsPerSecond.toInt > 0, s"input rate value must be greater than 0. Got $rowsPerSecond")
```

- ✓ Environment changes
- ✓ User error
- ✓ Input parameters > 10
- ✓ Type casting parameter value
- ✓ Tracking parameter changes

- ❑ Reduce boilerplate code
- ❑ Enhanced validation

databricks

Boilerplate free configuration code

PureConfig - Scala library for loading configuration files

- Reads HOCON, .properties, JSON to native Scala classes in a boilerplate-free way
- Supports sealed traits, case classes, collections, optional values out-of-the-box
- Config Writers to generate configuration files

Official Docs:https://pureconfig.github.io/

Sample Code

```scala
import scala.concurrent.duration.{Duration => ScalaDuration}
import org.apache.spark.sql.streaming.Trigger

sealed trait StreamingTrigger
case object TriggerOnce extends StreamingTrigger
case class TriggerWithInterval(interval: ScalaDuration) extends StreamingTrigger

case class PipelineConf(
                         outputPath: String,
                         checkpointPath: String,
                         rowsPerSecond: Long,
                         streamingTrigger: StreamingTrigger,
                         tuning: Seq[String]
                       ) {
  require(rowsPerSecond > 0, s"input rate value must be greater than 0. Got $rowsPerSecond")
  require(!outputPath.isEmpty, "table output path cannot be emppty")
  require(
    tuning.forall { sparkConfString =>
      sparkConfString.split("=").size == 2
    }, """Invalid spark configuration found. A valid spark spark configuration must be of type "<key>=<value>"""
  )
  val trigger: Trigger = streamingTrigger match {
    case TriggerOnce => Trigger.Once()
    case TriggerWithInterval(duration) => Trigger.ProcessingTime(duration)
  }
}

import pureconfig._
import pureconfig.generic.auto._

val configStr: String =
  """
    |  output-path = /table/output/path/,
    |  checkpoint-path = /table/checkpoint/path,
    |  rows-per-second = 1,
    |  streaming-trigger { type = "trigger-once" }
    |  tuning = ["spark.sql.shuffle.partitions=10"]
                """.stripMargin

val pipelineConf: PipelineConf = ConfigSource.string(configStr).load[PipelineConf].right.get

output                                                                                                  : sql.DataFrame
  .writeStream                                                                                          : DataStreamWriter[Row]
  .trigger(pipelineConf.trigger) // pipelineConf.trigger: org.apache.spark.sql.streaming.Trigger = OneTimeTrigger
  .format("delta")                                                                                      : DataStreamWriter[Row]
  .option("checkpointLocation", pipelineConf.checkpointPath)                                            : DataStreamWriter[Row]
  .start(pipelineConf.outputPath)                                                                       : StreamingQuery
```

- ✓ Environment changes
- ✓ User error
- ✓ Input parameters > 10
- ✓ Type casting parameter value
- ✓ Tracking parameter changes
- ✓ Reduce boilerplate code
- ✓ Enhanced validation

Summary

- User errors are inevitable. Make it hard, if not impossible, to make mistakes
- Catch errors and fail as early as you can
  - Compile Time **>** Unit Test **>** Integration Test **>** Runtime (before Spark Job starts)
- Externalize configuration to track changes
- Use libraries to reduce boilerplate

Source Code and examples available at

https://github.com/ganeshchand/lets-dumb-proof-data-pipeline

Resources