

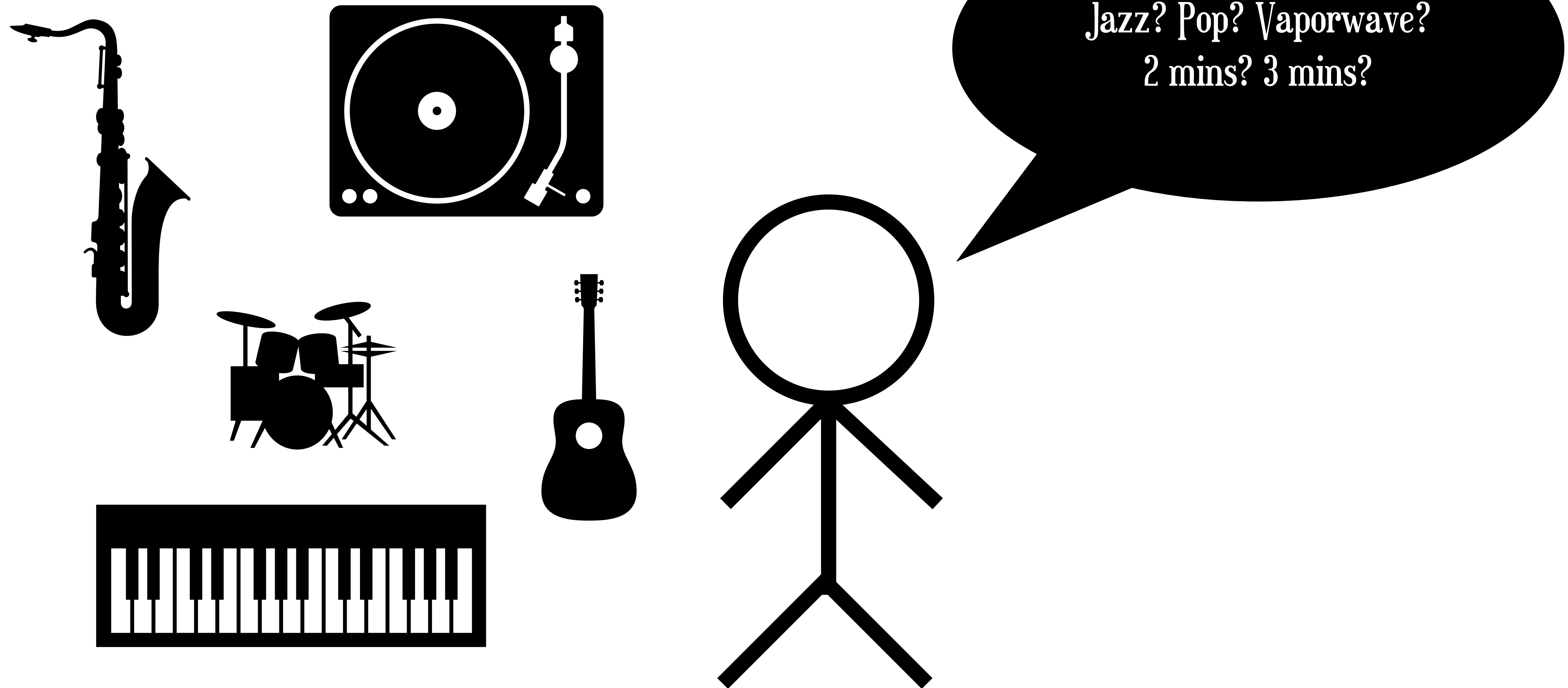
# Music Popularity Prediction

CSYE7200 – Final Project

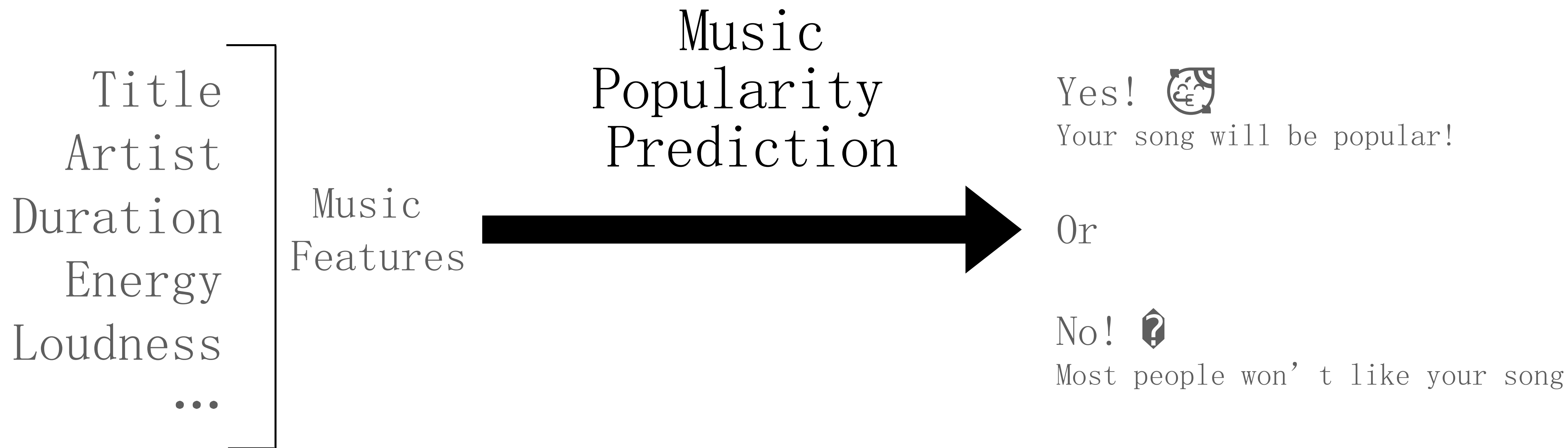
# Our team

3 members!

- Yiqing Huang ( Jackie / 黄以清 )
- Qinyun Lin ( Niro / 林沁昀 )
- Zhilue Wang ( Harry / 王之略 )



(Music producer who is trying to write next song)



# Data sources

- Million Song Dataset
  - <http://millionsongdataset.com/>
- ~1,000,000 rows of data
- Has “**song\_hotttnesss**” attribute for popularity

Because the data size is too large (more than 300 GB), we finally used a subset of the source data (about 380,000 rows)


Some attributes

end\_of\_fade\_in  
start\_of\_fade\_out  
loudness  
tempo  
title  
year  
**song\_hotttnesss**  
...

# Use cases

- User calls the API with music features, and receives prediction value.
- User calls the API and system validates user input, informs user if there is any error.
- User calls the API and system processes the input data, runs ML model on it and returns the prediction value.

# Scala in our project

- Planning to write all codes in Scala 
- Data loading and pre-processing — Scala & Spark
- Machine learning — Spark build-in ML library
- API web service — Scala Play! Framework

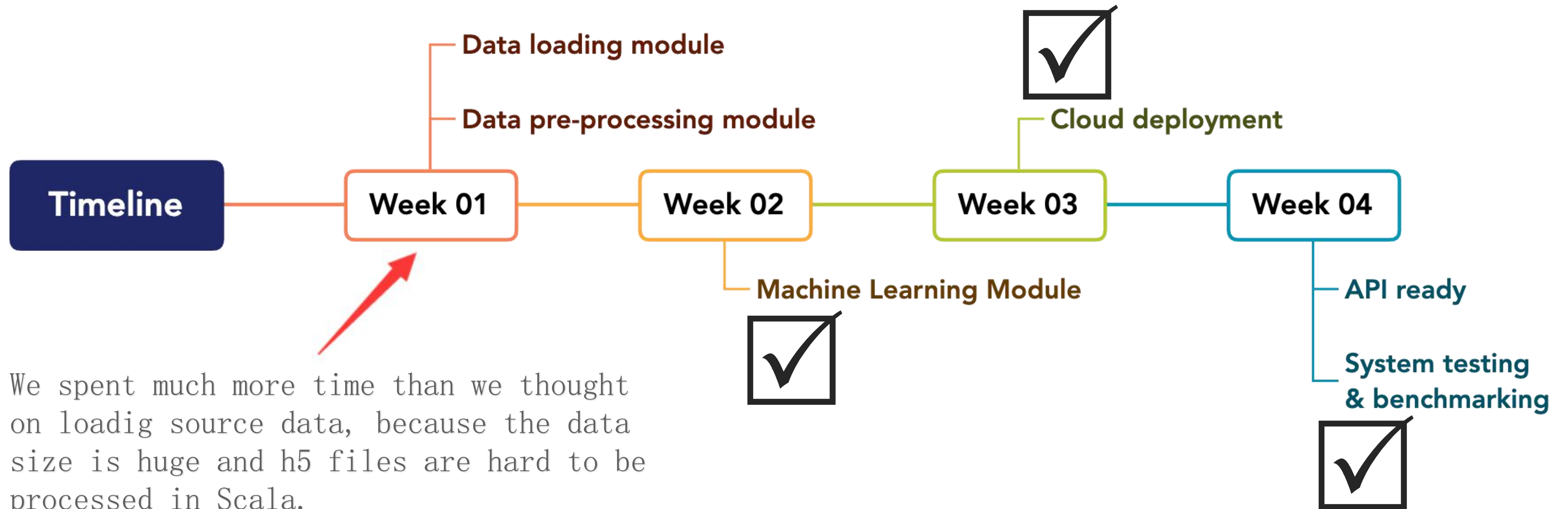
## Languages



Library used for reading h5 file  
in scala:  
<https://github.com/jamesmudd/jhdf>

- Our repo: <https://github.com/NiftyMule/csye7200-bigdata-project>

# Milestones



We spent much more time than we thought on loading source data, because the data size is huge and h5 files are hard to be processed in Scala.

Fortunately, we finished this milestone at Week 02 and didn't break our timeline



# Methodology

## Big data

- Whole application is a big-data application
- 2 loops
  - **Training:** Ingest -> Feature extraction -> Machine learning
  - **Inference:** Ingest -> Feature extraction -> Model inference -> Return

# Ingesting – Implementation

```
def loadRowFromH5(h5FilePath: String): Try[Option[Row]] = Try (...)

def loadDataFromFolder(folderPath: String, spark: SparkSession): Try[DataFrame] = Try (...)

def loadCsv(filepath: String, spark: SparkSession): Try[DataFrame] = Try {
  spark.read
    .option("delimiter", ",")
    .schema(getSchema())
    .csv(filepath)
}

def dfFromJson(json: JsValue, spark: SparkSession): Try[DataFrame] = Try {
  spark.read
    .schema(getSchema(isTrainData = false))
    .json(
      spark.createDataset(List(json.toString()))(Encoders.STRING)
    )
}
```


# Ingesting – Unit Test

```
▶ behavior of "loadDataFromH5"  
  
▶ ⊞ it should "successfully load sample .h5 file" in {...}  
  
▶ ⊞ it should "skip the row if hotness field is missing" in {...}  
  
▶ ⊞ it should "return failure when given invalid filepath" in {...}  
  
▶ behavior of "loadDataFromFolder"  
  
▶ ⊞ it should "successfully load data from h5 folders" in {...}  
  
▶ ⊞ it should "fail when given wrong filepath" in {...}  
  
▶ behavior of "loadCSV"  
  
▶ ⊞ it should "successfully load data from CSV file" in {...}  
  
▶ behavior of "Json converter"  
  
▶ ⊞ it should "successfully convert JsValue to DataFrame" in {...}
```

# Methodology

## Machine learning

- Spark built-in ML library
- Algorithms planning to be used:
  - Logistic regression
  - Random forest
  - Bag of Words



We didn' t use this technic because we end out dropping all StringType features and the accuracy still get up to the Acceptance criteria.



# Preprocessing

```
def columnProcessing(df: DataFrame, isTrainData: Boolean = true): Try[DataFrame] = Try {  
  // drop songs before 1920 and those with nan values  
  logger.info(message = s"raw dataframe size: ${df.count()}")  
  val df1 = df.filter(df("year") > 1920)  
    .na.drop(List("artist_latitude", "artist_longitude"))  
    .drop(colNames = "artist_id", "artist_name", "title", "artist_terms", "artist_terms_freq", "artist_terms_weight")  
    .withColumn(colName = "year", df("year") - 1920)  
  // logger.info(s"Batch year = ${df1.select("year").head(4).map(x => x.getInt(0)).mkString("Array(", ", ", ", ")")}")  
  
  if (isTrainData) {  
    val avgHotness = df1.select(expr(expr = "AVG(song_hotness)"))  
      .collect().head.getDouble(0)  
  
    // set training label & shift years  
    df1.withColumn(colName = "label", when(df1("song_hotness") >= avgHotness, value = 1).otherwise(value = 0))  
  } else df1  
}
```

Drop invalid data and useless features

Calculate Avg hotness for deciding if a song is popular

Generate labels for training



# Assembler & Standard Scaler

```
def assembleScalePipeline(df: DataFrame): Pipeline = {  
  val vectorAssembler = new VectorAssembler()  
    .setInputCols(df.dtypes  
      .filter(x => !List("song_hotness", "label").contains(x._1) && List("DoubleType", "IntegerType").contains(x._2))  
      .map(_._1)  
    )  
    .setOutputCol("raw_features")  
  
  val standardScaler = new StandardScaler()  
    .setInputCol("raw_features")  
    .setOutputCol("features") // MUST set here "features", Model will find this col by specific name to train  
  
  new Pipeline().setStages(Array(vectorAssembler, standardScaler))  
}
```

Choose Double and Int type of data as training features

Perform Stand Scale

Construct as a pipeline

# Model training

```
def fit(df: DataFrame,
      modelName: String,
      evaluate: Boolean = false
    ): Try[PipelineModel] = Try {
  val dataSplit = df.randomSplit(Array(0.8, 0.2), seed = 11L)
  val trainSet = dataSplit(0).cache()

  val assemble_scale_pipeline = assembleScalePipeline(df)

  val model = modelName match {
    case ModelName_LR => new LogisticRegression()
    case ModelName_RF => new RandomForestClassifier()
    case _ => throw new Exception("Invalid model name")
  }

  val whole_pipeline = new Pipeline().setStages(Array(assemble_scale_pipeline, model))

  logger.info(message = s"Fitting with $modelName...")
  val trainedPipelineModel = whole_pipeline.fit(trainSet)
  logger.info(message = s"Fitting complete! [$modelName]")

  // evaluate model performance
  if (evaluate) {...}

  trainedPipelineModel
}
```

80% for training, 20% for validation

Construct whole pipeline with assemble, scale and model

Evaluation part

# Machine Learning – Unit test

- ▶ *behavior* of "ML Pipeline"
- ▶ *it* should "successfully preprocess dataframe" taggedAs Slow in {...}
- ▶ *it* should "successfully train and predict" taggedAs Slow in {...}



# Methodology

## Cloud

A Master with 2 Slaves on AWS

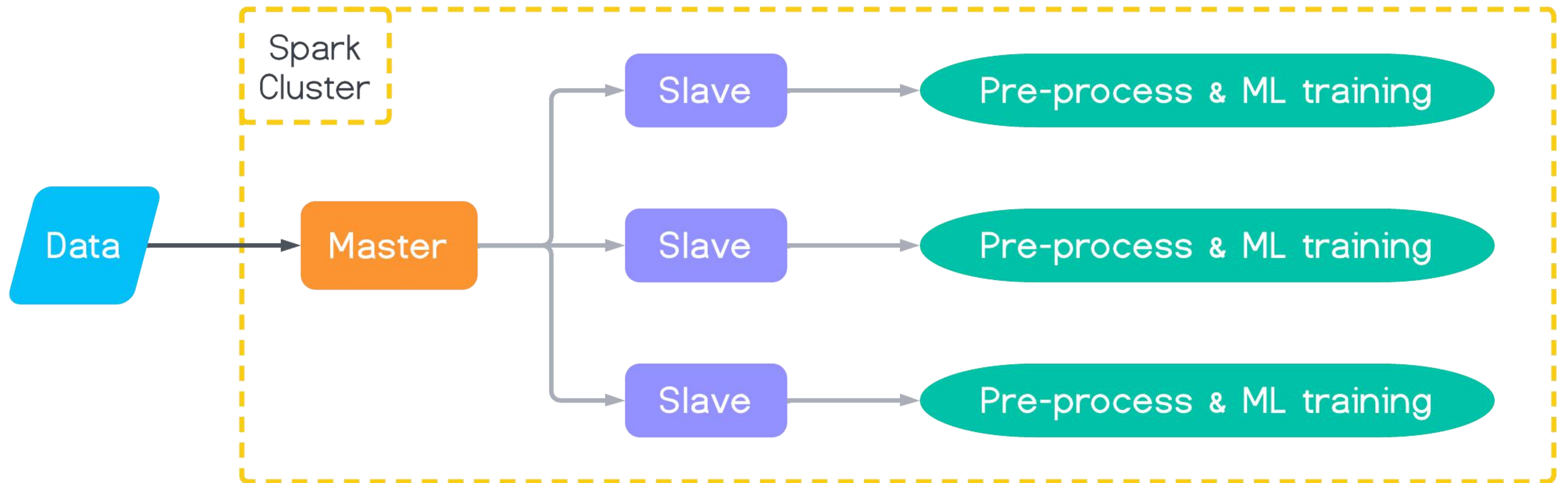
- Deploy Spark cluster on AWS
- To leverage the power of parallel computing
- Expose an API

We finally designed 5 APIs exposed to users using Play Framework:

```
# Spark
GET      /spark/train
POST     /spark/infer/lr
POST     /spark/infer/rf
POST     /spark/infer_batch/lr
POST     /spark/infer_batch/rf
```

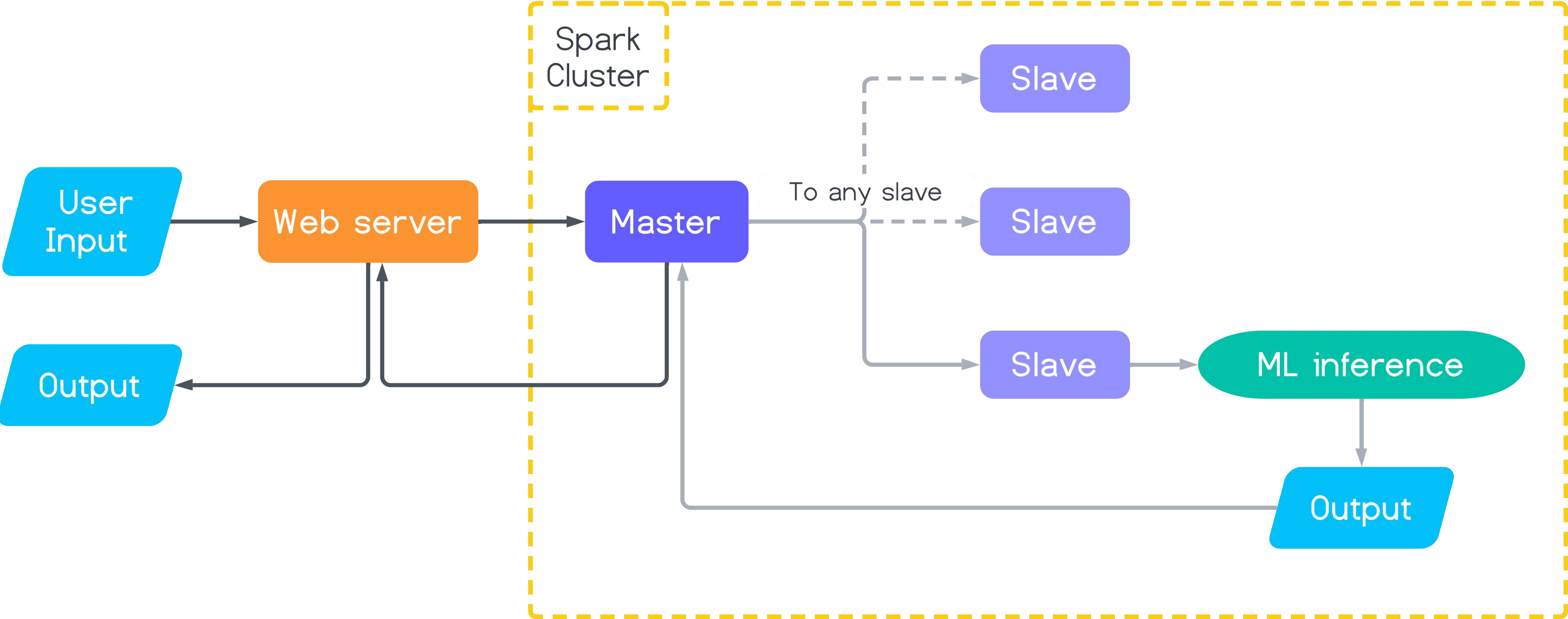
# Methodology

Training pipeline ← Made use of spark.ml.Pipeline to achieve



# Methodology

Inference pipeline  Made use of `spark.ml.PipelineModel` to achieve




# Acceptance criteria

- User queries responding time:
  - Single record: <5s ☒
  - Batch records: <1s per record (on average) ☒
    - (batch has >5 records)
- Model predicting time: <4s ☒
- Precision & recall of the ML model: >60% ☒

	Resonding time (Single)	Resonding time (per record on Avg) (Batch size=50)	Resonding time (per record on Avg) (Batch size=500)	Validation Accuracy	Recall
Logistic Regression	417.28ms	12.32ms	3.23ms	78.29%	79.12%
Random Forest	542.39ms	10.46ms	4.26ms	79.07%	78.02%



# Our goal

- Learn how to:
  - Process & load data in Spark and Scala ☒
  - Train a machine learning model and use it for prediction ☒
  - Deploy a Spark cluster to cloud environment ☒
  - Implement a simple web server in Scala ☒
- Apply these knowledges to build a real-world application! 

Front web page...

High concurrent request...

Larger Data...

To be continued...

DEMO

