Data Streaming Analytics using PySpark

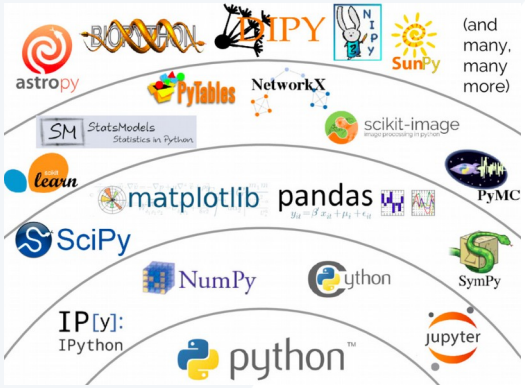
Laurynas Stašys, Data Analytics Lead
@Arxan Technologies

About me

- MSc in Software Engineering /Data Science
- Working with data analytics since 2015. Analyzing 100s million entities and 100TBs of data:
  - NLP, Time-series data analysis
- Previous experience in more “standard” software development fields

Technology stack:

- Pandas, NumPy, NTLK
- PySpark
- AWS ecosystem



Things I will talk about

- Apache Spark
- Spark Streaming
- Streaming analytics in Arxan Technologies

Things I will **not** talk about

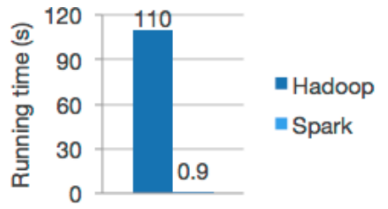
- Compare Spark with other frameworks (extensively)
- Low level Spark stuff
  - Check Apache project for more information
- Spark ML
  - Next time(?!)

Apache Spark

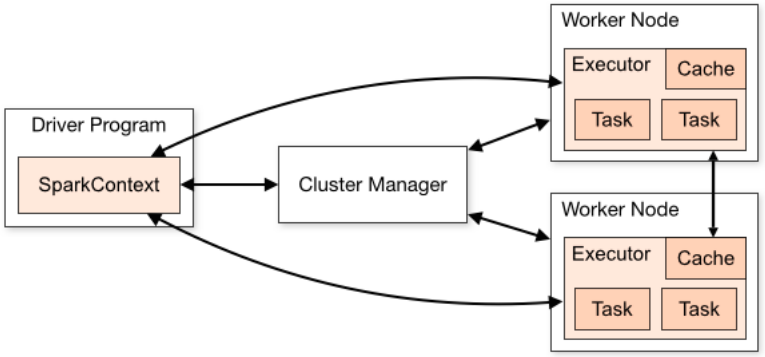
What is Spark

In-memory based data processing framework which main features are:

- RDDs (resilient distributed datasets) – fault-tolerant collection of elements that can be operated in parallel.
  - Can parallelize existing datasets or reference datasets in external storage system
  - If any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it (DAG)
- Performs computations in cluster which consists of the following components (each running their own Java processes):
  - Driver – is a JVM process that hosts SparkContext for a Spark application in Spark Master Node and hosts Web UI Application
  - Executor – performs data processing tasks and keeps data in memory
  - Cluster manager – YARN application master which handles data transfer
- Two types of operations:
  - Transformations - creates a new dataset from an existing one
  - Actions return a value to the driver program after running a computation on the dataset

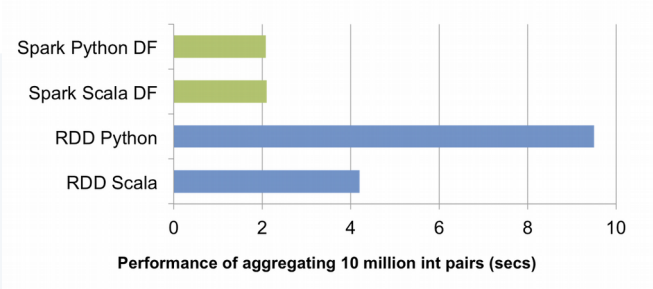


Logistic regression in Hadoop and Spark



Spark DataFrame API

- Think of named RDD with metadata
- Conceptually equivalent to a table in a relational database or a data frame in R/ Python
- Features built-in SQL-like functions
  - Check DataFrame API for more information
- ”Go-to” programming model as of Spark v2.1
  - Better programming model than operating with RDDs
    - i.e. RDD has no schema
- Better performance compared with RDD

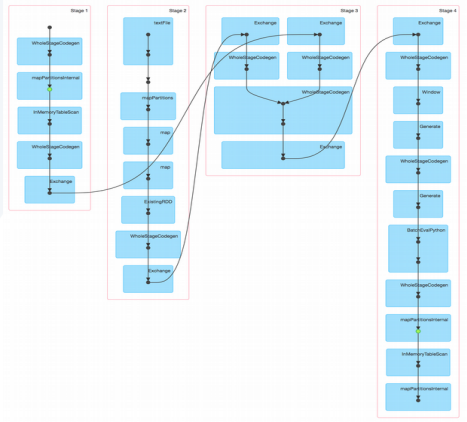


Directed acyclic graphs (DAG)

Transformations in Spark are lazy, in a way that they are not computed right away (until data is sent to driver).
Spark itself creates DAGs – chain of operations - which is graph of of tasks –

- *map* tasks: tasks that organizes data
- *reduce* tasks: tasks that aggregates data

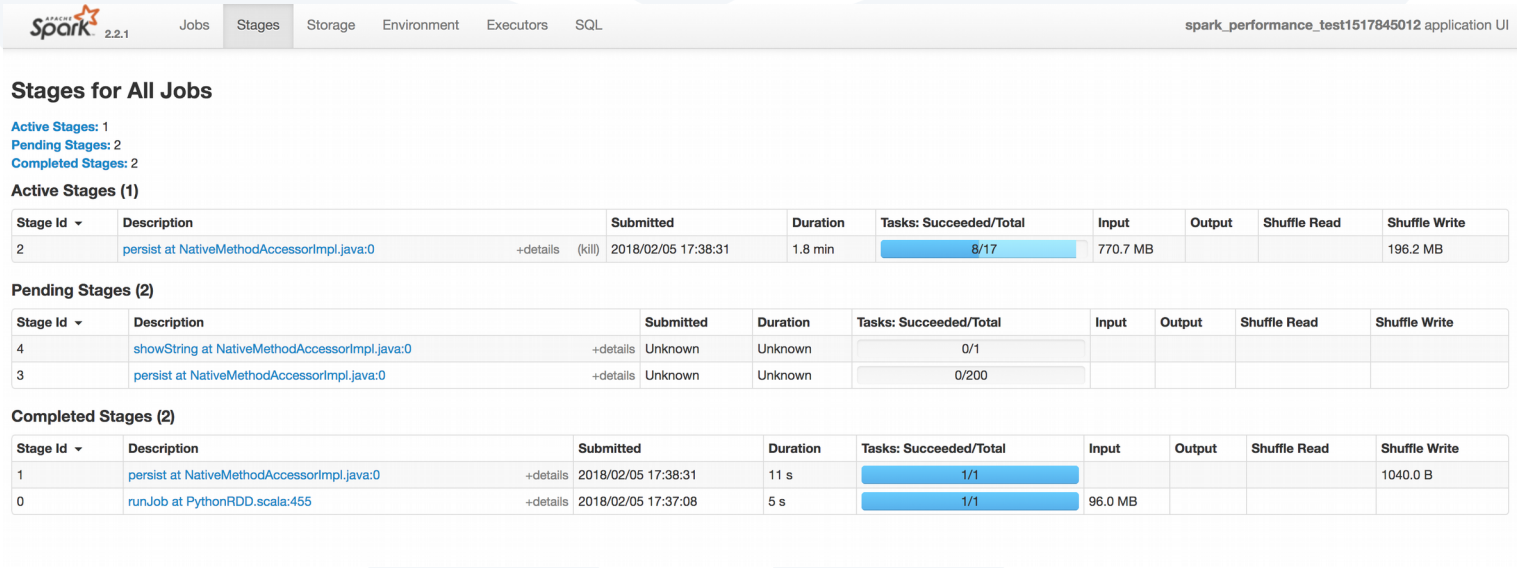
```
data_frame = data_frame \
    .drop('app_name', 'authorization_token') \
    .withColumn('request_id', F.row_number().over(Window.partitionBy('device_uniqueness').orderBy('device_uniqueness'))) \
    .withColumn('records', F.explode('records')) \
    .withColumn('guard_type', F.col('records.Type')) \
    .withColumn('guard_id', F.col('records.id')) \
    .select('*', F.explode(F.col('records.Severities')).alias('event_severity', 'value')) \
    .drop('records') \
    .withColumn('event_count', F.col('value.Occurences')) \
    .withColumn('last_event', F.col('value.LastEvent')) \
    .withColumn('first_event', F.col('value.FirstEvent')) \
    .withColumn('first_event_timestamp', parse_timestamp_to_datetime_udf(F.col('value.FirstEvent'))) \
    .drop('value') \
    .withColumn('year_id', F.year(F.col('first_event_timestamp'))) \
    .withColumn('month_id', F.month(F.col('first_event_timestamp'))) \
    .withColumn('day_id', F.dayofmonth(F.col('first_event_timestamp'))) \
    .withColumn('hour_id', F.hour(F.col('first_event_timestamp'))) \
    .withColumn('week_id', F.weekofyear(F.col('first_event_timestamp'))) \
    .drop('first_event_timestamp')
```



Spark UI

Spark master node will host Spark UI (usually on port 4040) which will allow users to see:

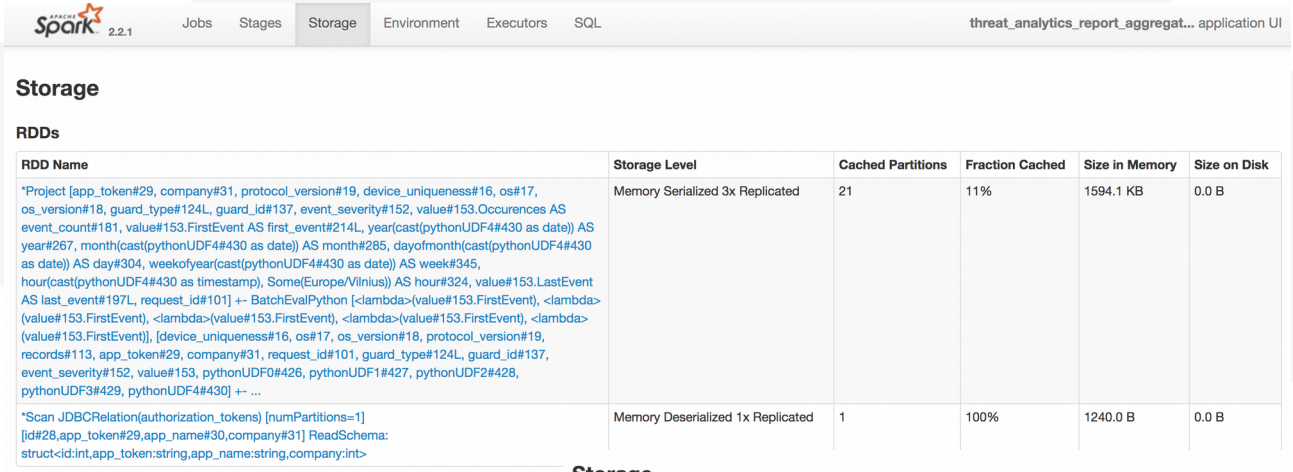
- Running jobs
- Job stages
  - Each stage will run single task for each partition
  - Detailed DAG information for each task
- Persisted RDDs



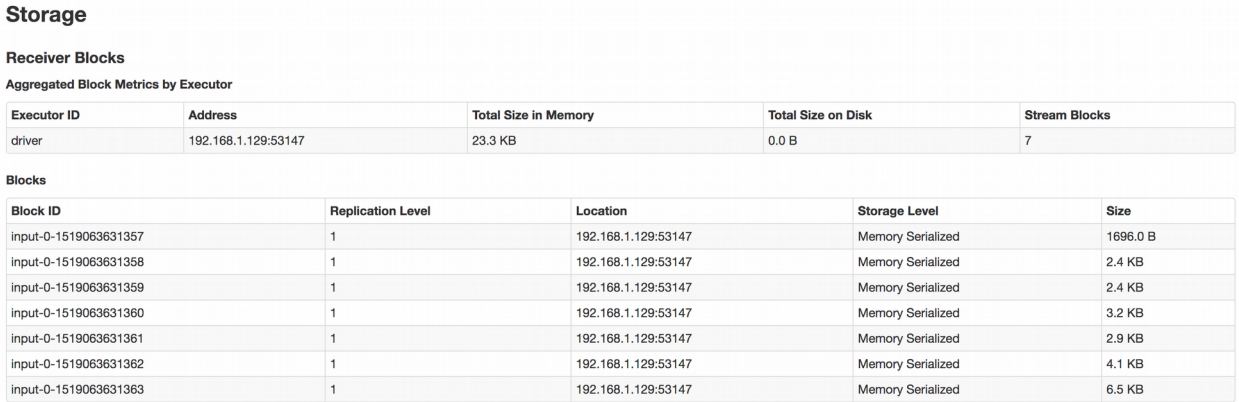
Spark UI. Storage

/storage page:

- Gives information about each of the persisted RDDs
  - Good tool to debug your programs



Also for streaming:



Pyspark Streaming

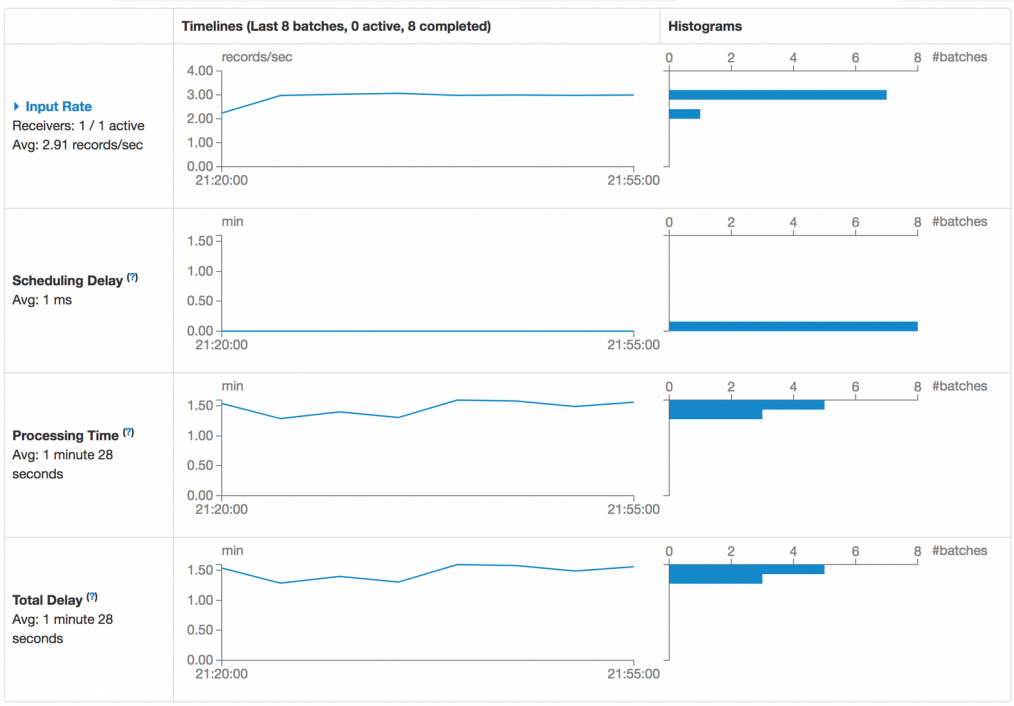
- Built-in support for many popular MQs
- Python API has some features missing compared to other APIs
- StreamingContext – main object for streaming functionality
- Stream can be treated as sequence of incoming RDDs


```
ssc = StreamingContext(spark.sparkContext, config["WindowIntervals"]["WindowDurationSeconds"])

streamName = config["KinesisConfig"]["StreamName"]
appName = config["KinesisConfig"]["StreamAppName"] + uuid.uuid4().__str__()
endpointUrl = config["KinesisConfig"]["EndpointUrl"]
regionName = config["KinesisConfig"]["RegionName"]

dataStream = KinesisUtils.createStream(ssc, appName, streamName, endpointUrl, regionName,
                                       InitialPositionInStream.LATEST, 3600,
                                       awsSecretKey=config["AWSCredentials"]["AccessKey"],
                                       awsAccessKeyId=config["AWSCredentials"]["AccessKeyId"])

dataStream.foreachRDD(perform_window_aggregation_pipeline)

ssc.start()
ssc.awaitTermination()
```

- Processing is performed for collection of RDDs (DStream) which runs *foreachRDD* method to run data processing every interval

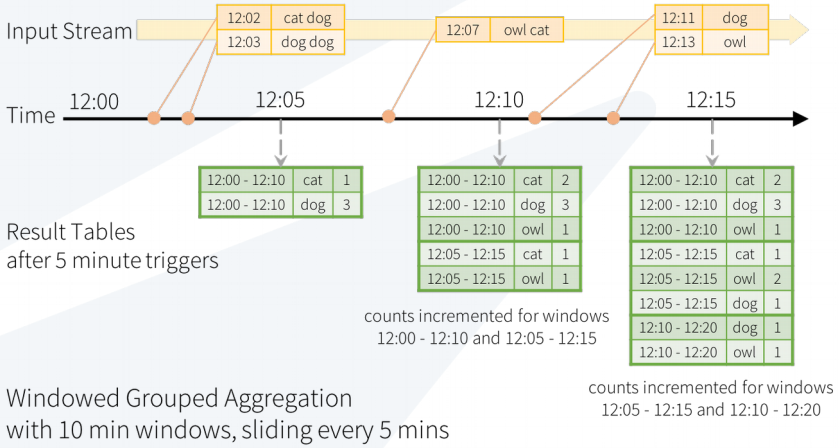
Pyspark Streaming UI

- Each batch is processed as separate stage
  - Spark Streaming Context is continuous stage that should not be killed!
- Provides stats on processing jobs



Pyspark structured streaming

- Spark provides API to process incoming data as DataFrame without needing to convert incoming RDDs to DataFrame object
- Window functions allows to run data analysis/aggregations for specified time interval
- Data is treated as single RDD which is collected in interval



Data analytics in Arxan

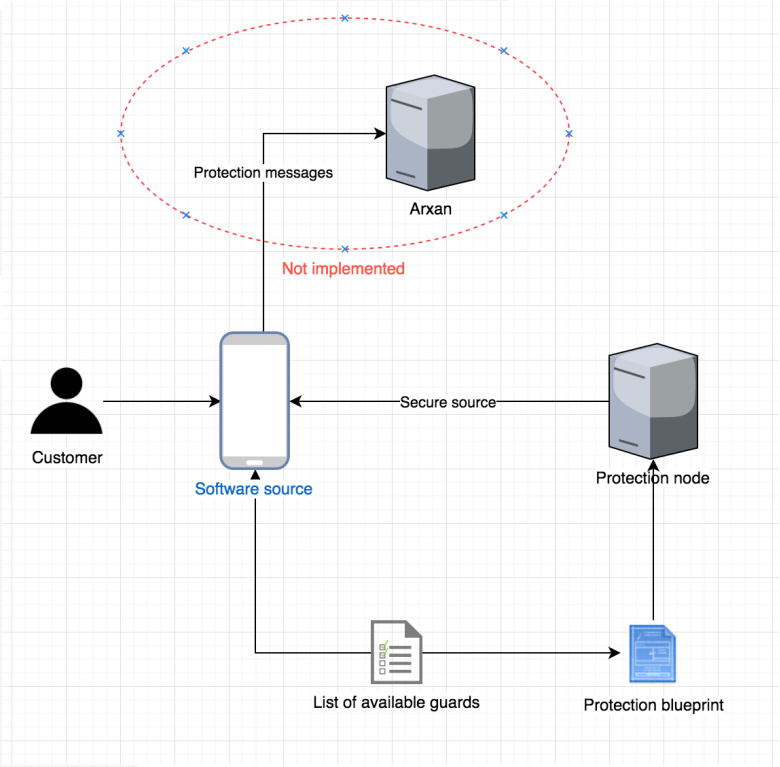
What do we do in Arxan

- Develop application security solutions for mobile and IOT
- Code obfuscation and protection
- Application protection uses guards which helps to identify application hacking activities
- Used by many financial institutions, game developers and basically everyone who wants to protect their users
- Securing different platforms:
  - Android
  - iOS
  - Web
  - Java
  - Etc.
- Only handful hacker activities among huge installation base

Protection workflow

Data workflow is defined as following:

- Customer owns an application source which he wants to protect
- Arxan provides guards which can be applied to protect the application
- Protection blueprint is generated which can be treated as a build template
- Protection node protects app and returns secured source
- When application is run it reports if any guard was tampered

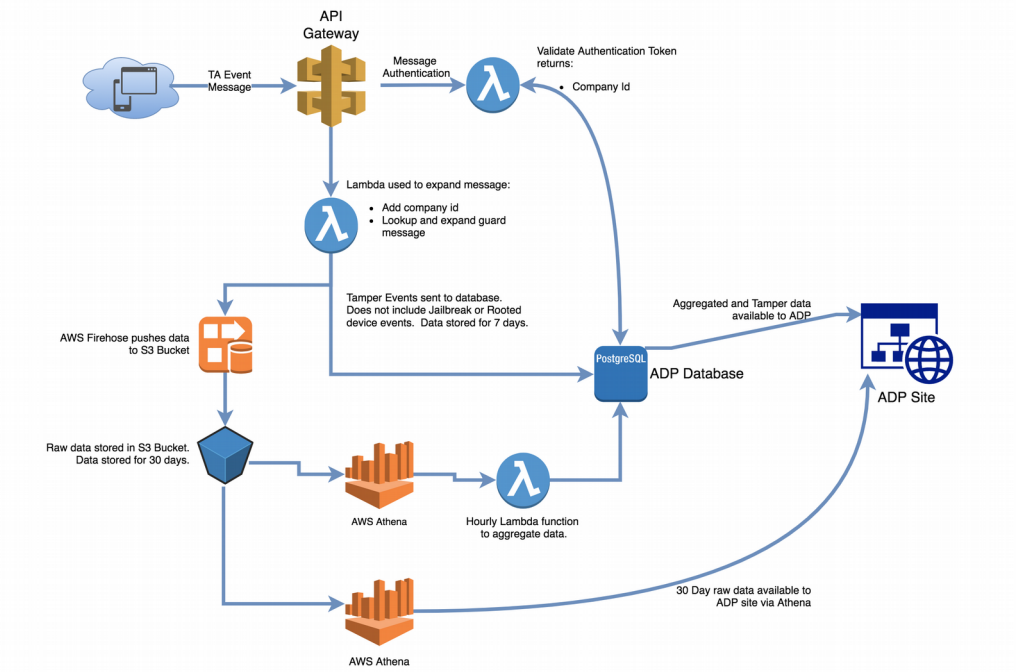


Challenge

- At the moment, customers does not know whether our guards work or not
  - We are sure if something happened only once something really bad happen or our customer reports hacking activity
  - Applications are not reporting if they are hacked
- Customer are not able to see benefit of our protection
  - no visibility until something really bad happens
  - no way of identifying deeper causes of issues (no causality analysis)

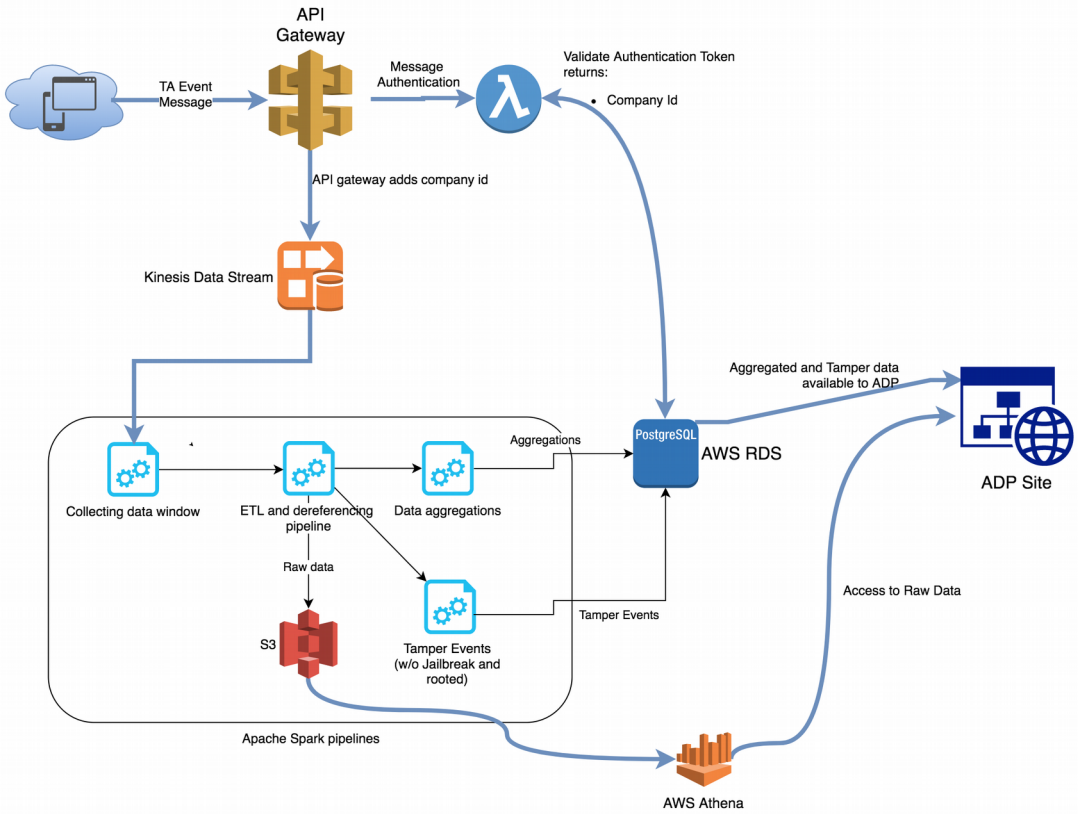
Current architecture. Phase 1

- Sticking to legacy frameworks and technologies
- Lambda functions performs ETL operations
- Operations are performed as each request comes in:
  - DB overhead when dereferencing data
- JavaScript APIs
- Using scheduled lambdas to generate reports
- AWS Kinesis Firehose streams have predefined input/output sources
  - Unable to define more advanced options – data partitioning, limited output formats



Spark solution architecture

- Accessing request data from AWS Kinesis
  - Don’t need to use additional servers to host Kafka, Rabbit or etc.
- Processing data using Spark Streaming API
- Writing raw data to S3 as Parquet files (using snappy compression)
  - Using distcp to copy data from HDFS to S3
- Writing reports to AWS RDS
  - Using Apache Sqoop to copy data from HDFS to RDS



Streaming data using Kinesis

ARXAN

- Python Kinesis API does not support structured streaming therefore we need to manually define schema
- Must use *spark-streaming-kinesis-asl* library

```
schema = T.StructType([
    T.StructField('field_1', T.StringType(), True,
                  metadata={'description': 'field descr'}),
    T.StructField('field_2', T.StringType(), True,
                  metadata={'description': 'field descr'}),
    T.StructField('field_3',
                  T.LongType(), True,
                  metadata={'description': 'field descr'}),
    T.StructField('field_4', T.StringType(), True,
                  metadata={'description': 'field descr'}),
    T.StructField('field_5', T.StringType(), True,
                  metadata={'description': 'field descr'}),
    T.StructField('field_6', T.IntegerType(), True,
                  metadata={'description': 'field descr'}),
    T.StructField('field_7', T.ArrayType(
        T.StructType([
            T.StructField('field_8', T.LongType(), True,
                          metadata={'description': 'field descr'}),
            T.StructField('field_9', T.StringType(), True,
                          metadata={'description': 'field descr'}),
            T.StructField('field_10', T.MapType(T.StringType(), T.StructType(
                [
                    T.StructField('field_11', T.IntegerType(), True,
                                  metadata={'description': 'field descr'}),
                    T.StructField('field_11', T.LongType(), True,
                                  metadata={'description': 'field descr'}),
                    T.StructField('field_12', T.LongType(), True,
                                  metadata={'description': 'field descr'})
                ]
            )), False, metadata={'description': 'field_13'})
        ])), True, metadata={'description': 'field_14'})
])

dstream = dstream.map(lambda x: json.loads(x))
df = spark.createDataFrame(dstream, schema, verifySchema=False)

# Easiest way of renaming variables. Renaming in order to have consistent naming in DB tables
df = df.selectExpr('field_1 as new_field_name', 'field_2 as new_field_name', ...)

return df
```

Data aggregations

- Using built-in DataFrame SQL functions to aggregate and transform data (check SQL manual)
  - Have to use custom UDFs to implement more complicated calculations
  - Custom UDFs are not running on Scala interpreter therefore is much slower
  - API not yet matured to calculate functions on multiple columns
- Since memory size is finite :
  - Aggregating only on hour basis
  - Requirement for distinct devices stats:
    - We can’t persist data for a month or week therefore we need to store raw data and run scheduled jobs every month, week

```
report = df \
    .groupBy('year', 'month', 'day', 'hour', 'company', 'app_token', 'os', 'device_uniqueness', 'request_id') \
    .agg(F.size(F.collect_list(F.col('event_severity'))).alias('guard_counts'))\
    .groupBy('year', 'month', 'day', 'hour', 'company', 'app_token', 'os')\
    .agg(F.countDistinct(F.col('device_uniqueness')).alias('installation_count'),
         F.collect_list('guard_counts').alias('guard_counts')) \
    .withColumn('guards_fired_max', max_value_udf(F.col('guard_counts'))) \
    .withColumn('guards_fired_mean', mean_value_udf(F.col('guard_counts'))) \
    .withColumn('guards_fired_median', median_value_udf(F.col('guard_counts'))) \
    .withColumn('report_granularity', F.lit(1)) \
    .withColumn('time_id', F.concat_ws(' ',
                                       F.concat_ws('-', F.col('year'), F.col('month'), F.col('day')),
                                       F.concat_ws(':', F.col('hour'), F.lit('0:0.0')))) \
    .withColumn('time_id', F.to_timestamp(F.col("time_id"), "yyyy-M-d HH:m:s.SSS")) \
    .drop('guard_counts')
```

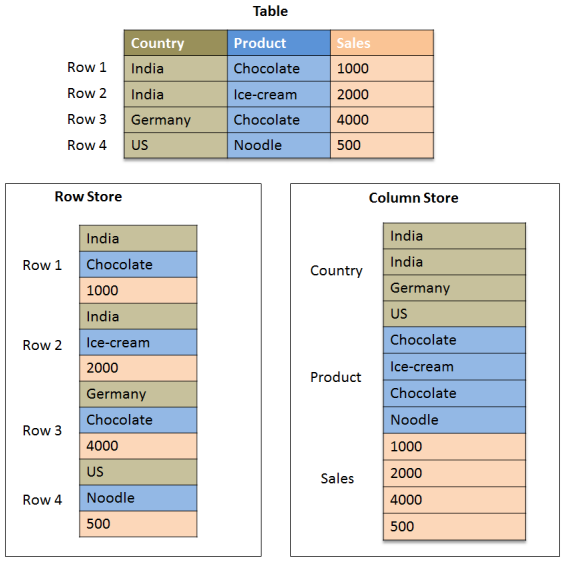
```
report = df \
    .groupBy('year', 'month', 'day', 'hour', 'os', 'company', 'app_token') \
    .agg(F.countDistinct('device_uniqueness').alias('unique_installations')) \
    .withColumn('report_granularity', F.lit(1))\
    .withColumn('time_id', F.concat_ws(' ',
                                       F.concat_ws('-', F.col('year'), F.col('month'), F.col('day')),
                                       F.concat_ws(':', F.col('hour'), F.lit('0:0.0')))) \
    .withColumn('time_id', F.to_timestamp(F.col("time_id"), "yyyy-M-d HH:m:s.SSS"))
```

```
window = Window.partitionBy('device_uniqueness').orderBy('device_uniqueness')
data_frame = data_frame \
    .drop('app_name', 'authorization_token') \
    .withColumn('request_id', F.row_number().over(window)) \
    .withColumn('records', F.explode('records')) \
    .withColumn('guard_type', F.col('records.Type')) \
    .withColumn('guard_id', F.col('records.id')) \
    .select('*', F.explode(F.col('records.Severities')).alias('event_severity', 'value')) \
    .drop('records') \
    .withColumn('event_count', F.col('value.Occurences')) \
    .withColumn('last_event', F.col('value.LastEvent')) \
    .withColumn('first_event', F.col('value.FirstEvent')) \
    .withColumn('first_event_timestamp', parse_timestamp_to_datetime_udf(F.col('value.FirstEvent'))) \
    .drop('value') \
    .withColumn('year', F.year(F.col('first_event_timestamp'))) \
    .withColumn('month', F.month(F.col('first_event_timestamp'))) \
    .withColumn('day', F.dayofmonth(F.col('first_event_timestamp'))) \
    .withColumn('hour', F.hour(F.col('first_event_timestamp'))) \
    .withColumn('week', F.weekofyear(F.col('first_event_timestamp'))) \
    .drop('first_event_timestamp')

data_frame = data_frame.select('app_token', 'company', 'protocol_version', 'device_uniqueness', 'os',
                               'os_version', 'guard_type', 'guard_id', 'event_severity', 'event_count',
                               'first_event', 'year', 'month', 'day', 'week', 'hour',
                               'last_event', 'request_id')
```

```
report = df\
    .groupBy('year', 'month', 'day', 'hour', 'company', 'app_token', 'os', 'os_version', 'guard_type')\
    .agg(F.countDistinct('device_uniqueness').alias('unique_installations'),
         calculate_dict_sum_list_of_array_udf(F.collect_list(F.array('event_severity', 'event_count'))).alias('severities'))\
    .withColumn('severities', dump_json_udf(F.col('severities'))).drop('guard_info')\
    .withColumn('report_granularity', F.lit(1))\
    .withColumn('time_id', F.concat_ws(' ',
                                       F.concat_ws('-', F.col('year'), F.col('month'), F.col('day')),
                                       F.concat_ws(':', F.col('hour'), F.lit('0:0.0')))) \
    .withColumn('time_id', F.to_timestamp(F.col("time_id"), "yyyy-M-d HH:m:s.SSS"))
```

Data compression

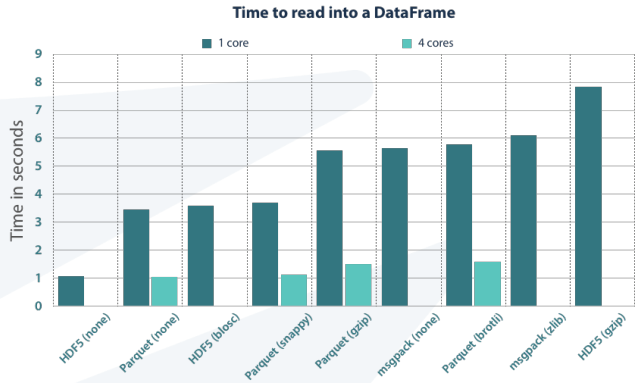
- Using Apache Parquet columnar storage format
  - Stores references to values rather than “raw” data as in CSV
  - Binary data with metadata
- Does not support complex data structures such as Dictionaries, Vectors, etc.
  - Need to flatten data to be able to store in Parquet

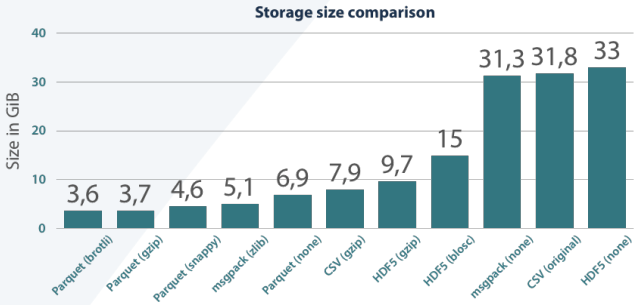


Data compression performance

- Larger datasets reach larger compression ratios
- Compressed data also achieves better performance as there are less data to read when performing calculations
- As of 2.2.1 API version *none, snappy, gzip,* and *lzo* compression algorithms are supported
  - Can be set by following config: *spark.sql.parquet.compression.codec*
  - Pandas also provide *brotli* compression which achieves even better ratios

- Benchmark application to test Parquet indicates that Parquet definitely adds performance gains





	Dimensions	Data size	Read performance	Count performance	GroupBy perfomance	Filter Performance
Parquet regular	$1.4*10^7$ rows x 11 columns	0.18 GB	0.54 sec	0.21 sec	5.94 sec	1.76 sec
Parquet large	$1.4*10^8$ rows x 11 columns	1.8 GB	0.25 sec	0.17 sec	28.69 sec	6.33 sec
CSV regular	$1.4*10^7$ rows x 11 columns	1.69 GB	3.77 sec	10.67 sec	23.10 sec	12.38 sec
CSV large	$1.4*10^8$ rows x 11 columns	16.87 GB	3.15 sec	83.01 sec	88.06 sec	75.05 sec

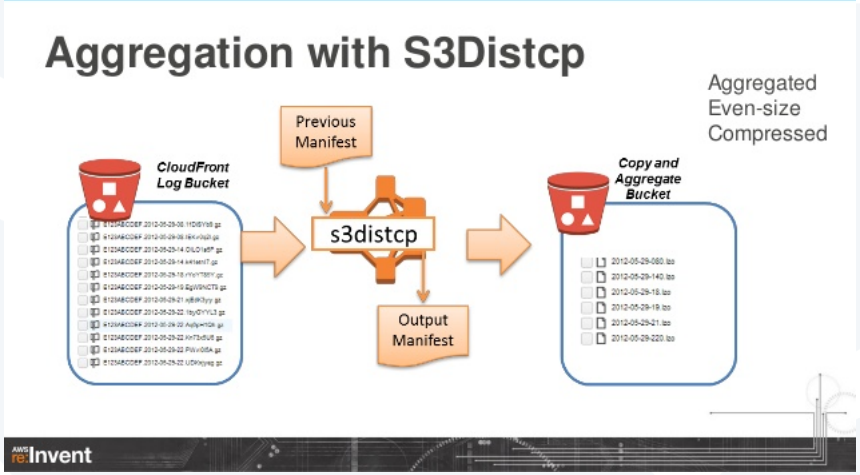
Writing data to HDFS

```
df \
    .write \
    .partitionBy('company', 'app_token', 'year_id', 'month_id', 'day_id', 'hour_id') \
    .parquet('hdfs://' + hdfs_path, mode='append')
```

```
df = spark\
    .read\
    .parquet(parquet_path)
```

- Spark integrates well with HDFS being one of the primary data sources
  - MongoDB
  - Neo4j
  - Traditional RDS
  - Many more available…
- Reading/writing Parquet is very simple due to built-in API support
- Using partitions when writing data to HDFS so that all reports/raw data extracts are stored in logically structured directories
  - Improves AWS Athena performance and reduce billing costs up to 95%

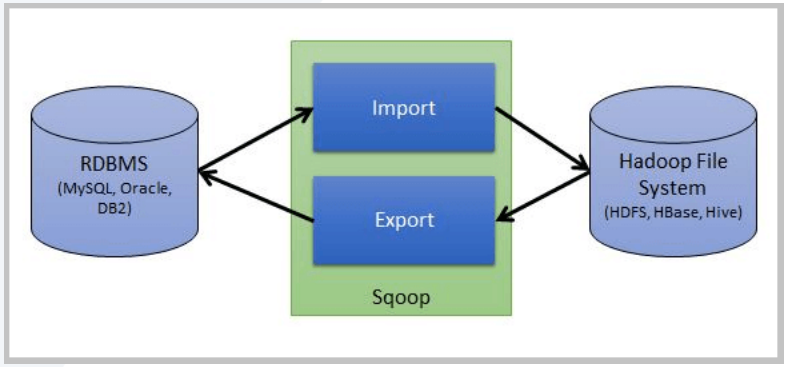
Transferring data to 3rd party storage



- For better performance initially writing data to HDFS
- Using AWS native S3Distcp component to move data from AWS EMR HDFS to AWS S3
- hadoop distcp
  -Dfs.s3a.access.key={KEY_ID}
  -Dfs.s3a.fast.upload=true
  -Dfs.s3a.secret.key={KEY_SECRET}
  hdfs://{DATA_PATH} s3a://{BUCKET_NAME}
- AWS EMR has its own distcp implementation which can be run as additional step in data processing pipeline

- Apache Sqoop is a component built to efficiently move data HDFS <-> RDS
- Using AWS Postgres RDS solution
- sqoop export
  --connect
  jdbc:postgresql://{HOST}:{PORT}/{DATABASE}?user={USER_ID}&password={PASSWORD}

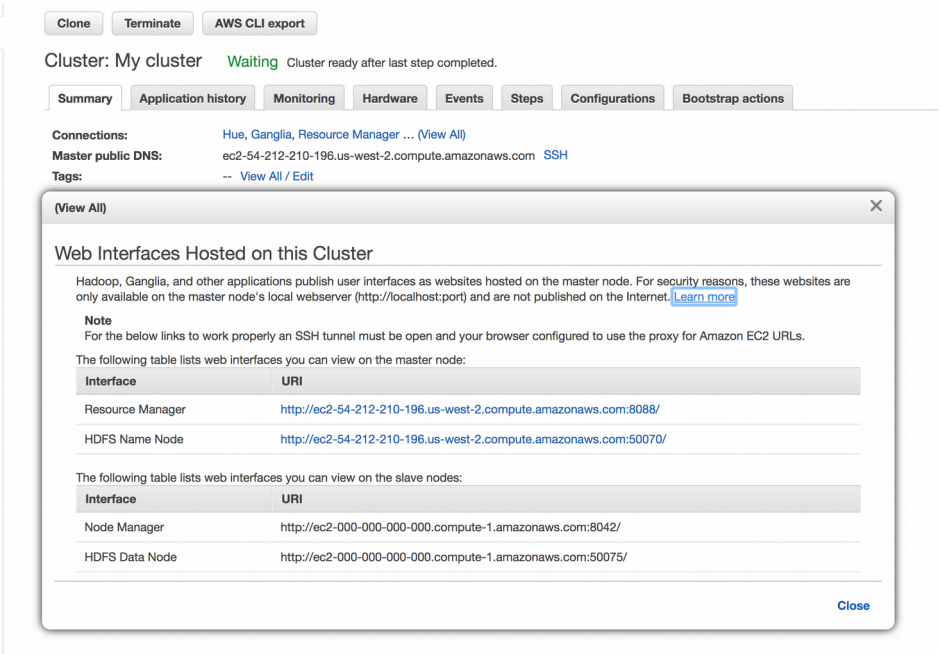
  --table {TABLE_ID}
  --export-dir {HDFS_PATH}



AWS EMR

Special kind of AWS EC2 instances that are built to be used for big data applications:

- Possibility to use MapReduce, Spark, Flink and many many more big data components to build solutions
- Provides many favorite components to manage cluster, data and etc.
- Spot instance availability (good for non production/POC projects)
- Use bootstrap actions (ending up writing bash files :/) to initiate environment in each EMR cluster server
- Provides list of running steps in cluster



Deploying application in AWS EMR

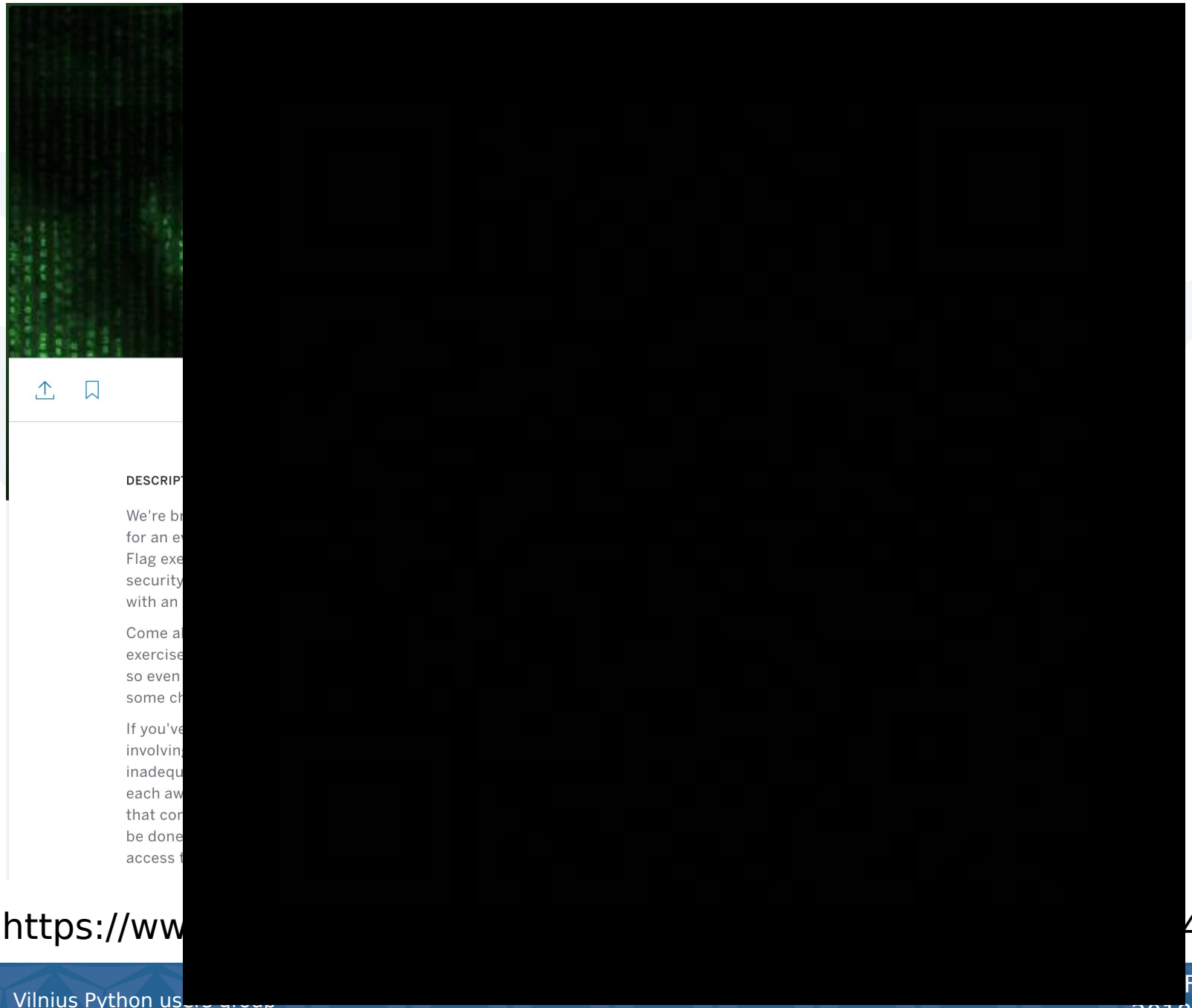
To run application in AWS EMR it's best to use AWS CLI component
Applications are treated as "steps" and are submitted by following command:

```
aws emr add-steps
--cluster-id {CLUSTER_ID}
--steps Type=spark,Name={STEP_NAME},
Args=[
--master,yarn,
--deploy-mode,cluster,
--conf,PYSPARK_PYTHON=/usr/bin/python3,
--packages,{PACKAGES_IDs},
--py-files,{PATH_TO_SOURCE.ZIP_IN_S3},
{PATH_TO_MAIN_SCRIPT_IN_S3},
--ConfigPath,{CONFIG_IN_S3_BUCKET_PATH}],
ActionOnFailure=CONTINUE
```

--conf contains settings such as python version, CLASS_PATH and etc.
--packages defines 3rd party dependencies from maven repos (i.e. org.apache.spark:spark-streaming-kinesis-asl_2.11:2.2.0 for Kinesis ACL library)
--py-files refers to the .py files that belong to the solution.
{PATH_TO_MAIN_SCRIPT_IN_S3} .py with main() function which is entry point of the processing
Other arguments following main script will be treated as command line arguments (sys.argv)

ARXAN

Questions



DESCRIPT

We're br
for an e
Flag exe
security
with an

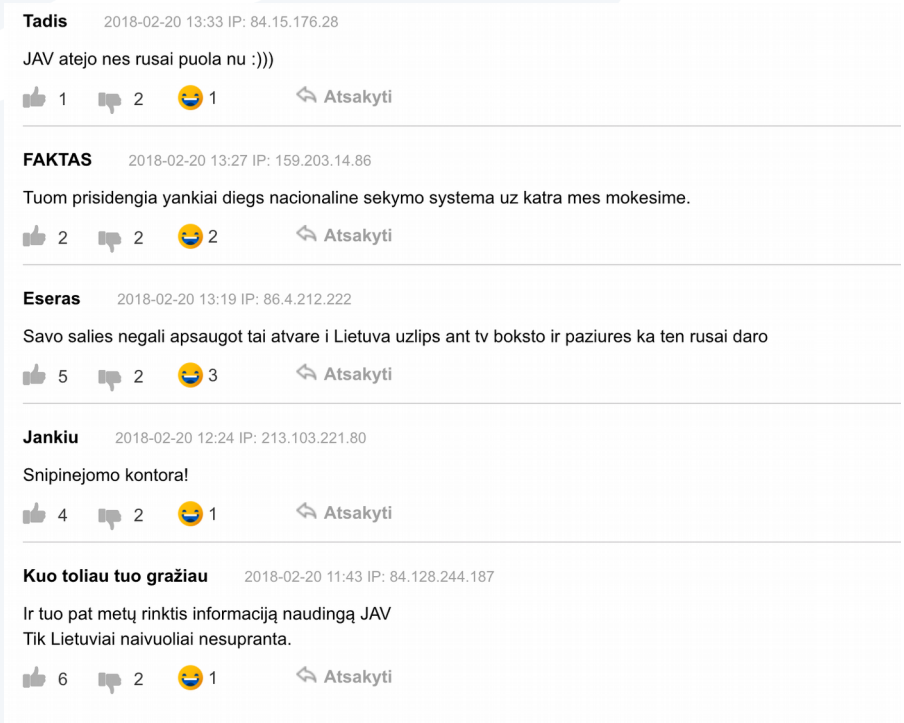
Come a
exercise
so even
some ch

If you've
involvin
inadequ
each aw
that cor
be done
access t

https://ww 42993444

We’re hiring

- C / Cryptography Engineer
- C++ Software Engineer
- C++ Software Engineer - Android
- Full Stack JavaScript Engineer
- Test Automation Engineer - Vilnius



Tadis 2018-02-20 13:33 IP: 84.15.176.28

JAV atejo nes rusai puola nu :)))

1 2 1 Atsakyti

FAKTAS 2018-02-20 13:27 IP: 159.203.14.86

Tuom prisidengia yankiai diegs nacionaline sekymo systema uz katra mes mokesime.

2 2 2 Atsakyti

Eseras 2018-02-20 13:19 IP: 86.4.212.222

Savo salies negali apsaugot tai atvare i Lietuva uzlips ant tv boksto ir paziures ka ten rusai daro

5 2 3 Atsakyti

Jankiu 2018-02-20 12:24 IP: 213.103.221.80

Snipinejomo kontora!

4 2 1 Atsakyti

Kuo toliau tuo gražiau 2018-02-20 11:43 IP: 84.128.244.187

Ir tuo pat metų rinktis informaciją naudingą JAV
Tik Lietuviai naivuoliai nesupranta.

6 2 1 Atsakyti