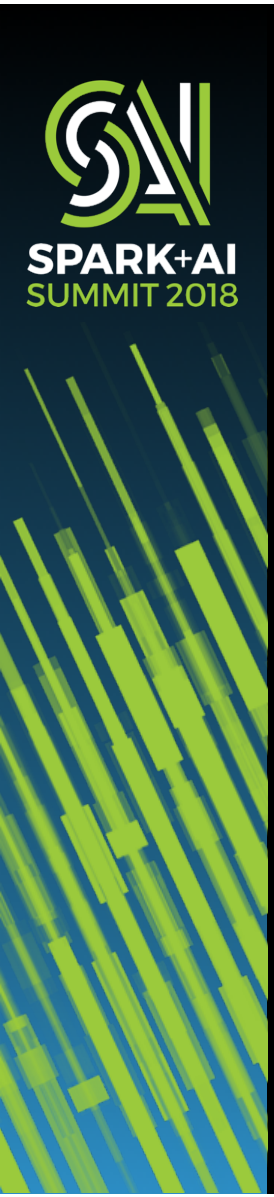


Streaming Trend Discovery:

real-time discovery in a sea of events

Scott Haines, Twilio

@newfront

#DDSAIS16

Agenda

- Intro and use cases for Spark at Twilio (5 min)
- Elements of a Streaming Discovery Engine (5 min)
- Principals of Valuable Data Sets (4 min)
- Refresher on Histograms and Percentiles (2 min)
- Building a Discovery Engine on Spark: Part One (15 min)
- Break?: 10 min
- Building a Discovery Engine on Spark: Part Two (15 min)
- Lessons learned (5 min)
- Questions (5 min)

Project Files on Github @ https://bit.ly/2ISI3BS

Scott Haines

Principal Engineer, Voice Insights @ Twilio

@newfront

- 6+ years working with Streaming Systems
  - Yahoo! Games
    - Recommendations, Personalization
  - Daily Fantasy Growth / Analytics
    - Mobile Campaigns, Fraud, Trends
  - Flurry Analytics
    - Statistics based notification system
- Stats / Data Science Enthusiast
- Drummer / Snowboarder / Hot Sauce Maker / Teacher

Twilio

Where I work

We are a public, large-scale, globally available & distributed communications as-a-service company.

- Founded in 2008. IPO'd in 2016
- Telecommunications is our bread and butter
- Along with Video / SMS / MMS / Wireless Sim / Authy / Sync
- Find out more and sign up @ twilio.com

twilio.com

Use cases for Spark @Twilio

- Real-Time Dashboards and Debugging
- Fraud Detection
- Carrier Behavior and Network Anomaly Detection
- Streaming ETL, Metrics and Aggregations
- ASR and Sentiment

Voice Insights @Twilio

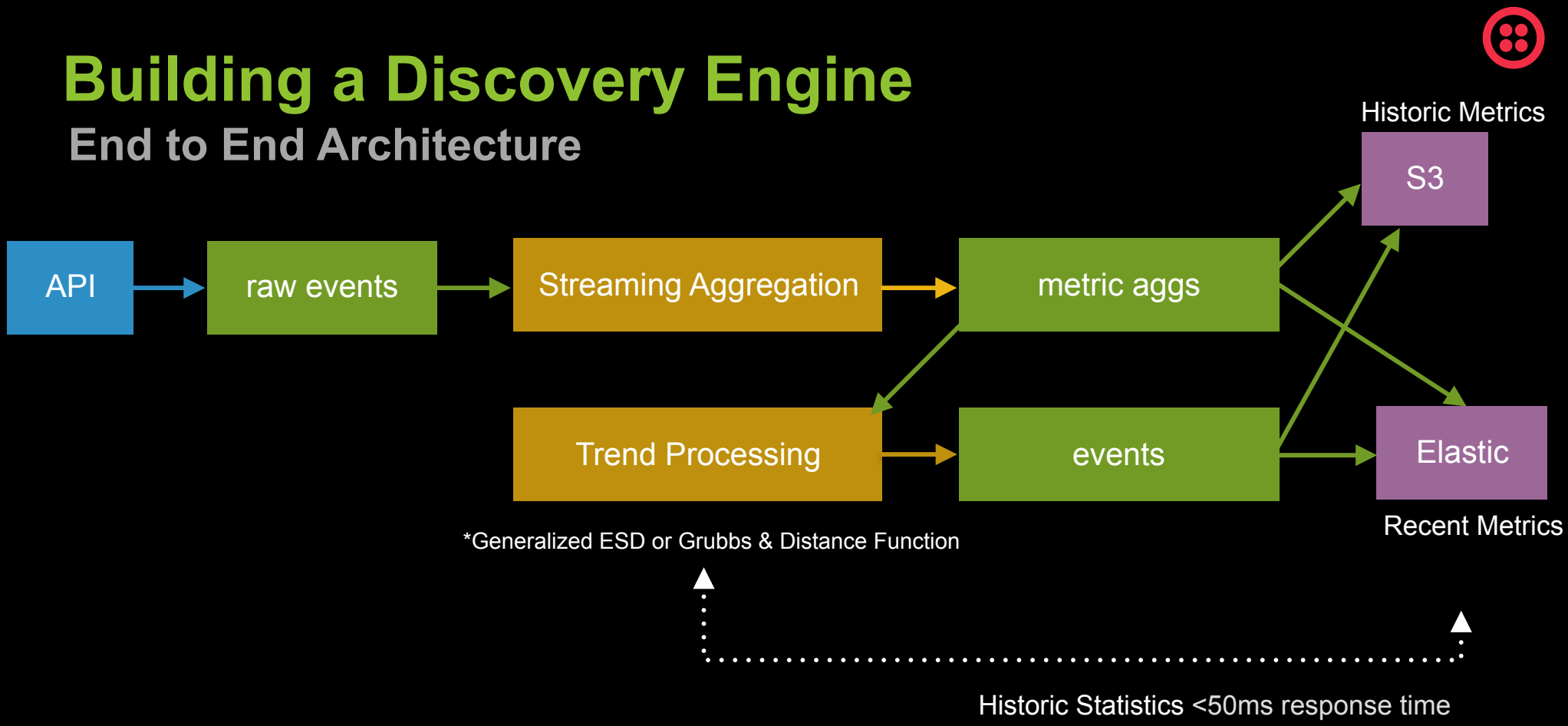
- Processing >3 TB /d
- >4 billion raw events /d
- >2 billion metric & sessions aggregations /d
- Powers Voice Insights / Carrier Ops / Support and more

Next up

- Elements of the Streaming Trend Discovery Architecture

Building a Discovery Engine

End to End Architecture



Core Principals of Valuable Data Sets

- Data must be Structured. No JSON!

Core Principals of Valuable Data Sets

- Data must be Structured. No JSON!
- JSON ~= Object

Core Principals of Valuable Data Sets

- Data must be Structured. No JSON!
- JSON ~= Object
- Avro, Protobuf, Parquet are widely used and can support backwards compatible updates.

Core Principals Establish a Data Contract

Structured Data has Types and are part of the Data Contract

- Properties of the Data must be documented.
- Data must be Accurate and Complete*
- Data must support backwards and forwards compatibility.
- Data should be easily flattened.

```
message Stats {
    optional double min      = 1;
    optional double p25      = 2;
    optional double median   = 3;
    optional double p75      = 4;
    optional double p90      = 5;
    optional double p95      = 6;
    optional double p99      = 7;
    optional double max      = 8;
    optional double mean     = 9;
    optional double sd       = 10;
    optional double variance = 11;
}
```

Core Principals Final Thoughts

- Structured Data allows for compile time guarantees
- Write once. Compile across languages
- Non-Java Object Serialization
- External Serialization simplifies deploys given you can simply use the last stored session state*

* don't corrupt your stored session data

Building our mental model

- Quick Refresher on Percentiles and Histograms

Building a Discovery Engine

Primer: What is a Percentile?

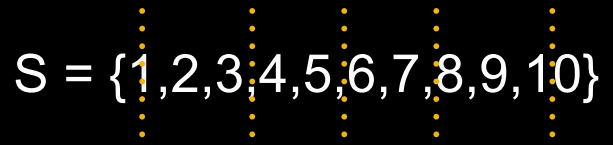
- Percentiles group numeric data in a set of observations.

S = {1,2,3,4,5,6,7,8,9,10}

Building a Discovery Engine

Primer: What is a Percentile?

- Percentiles group numeric data in a set of observations.
- Common splits are min, 25%,50% (median),75% and max

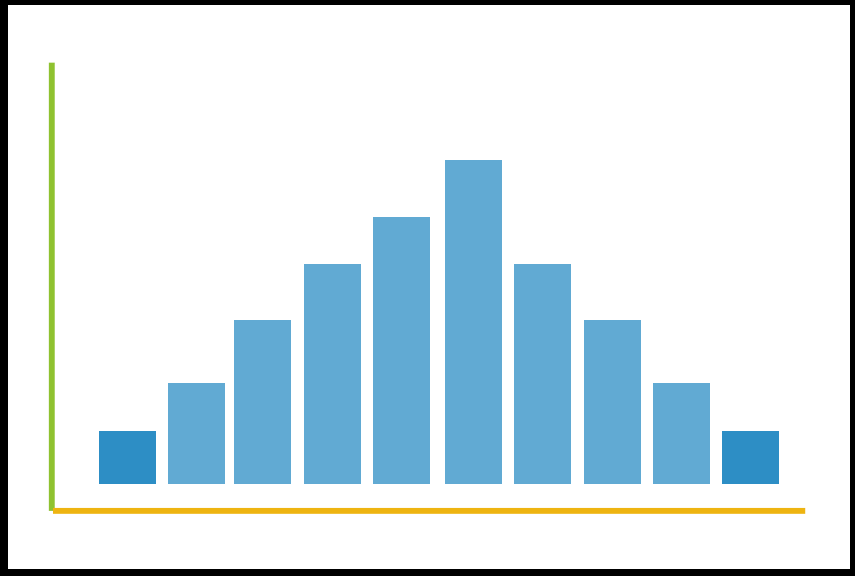


min:1
p25: 3.25
median: 5.50
p75: 7.75
max: 10.0

Building a Discovery Engine

Primer: What is a Histogram?

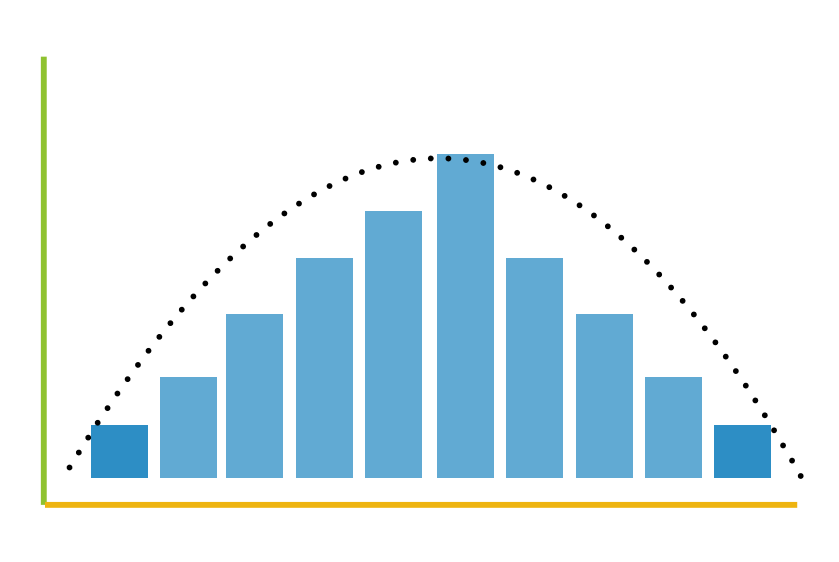
- Histograms are bucketed distributions of numeric datasets



Building a Discovery Engine

Primer: What is a Histogram?

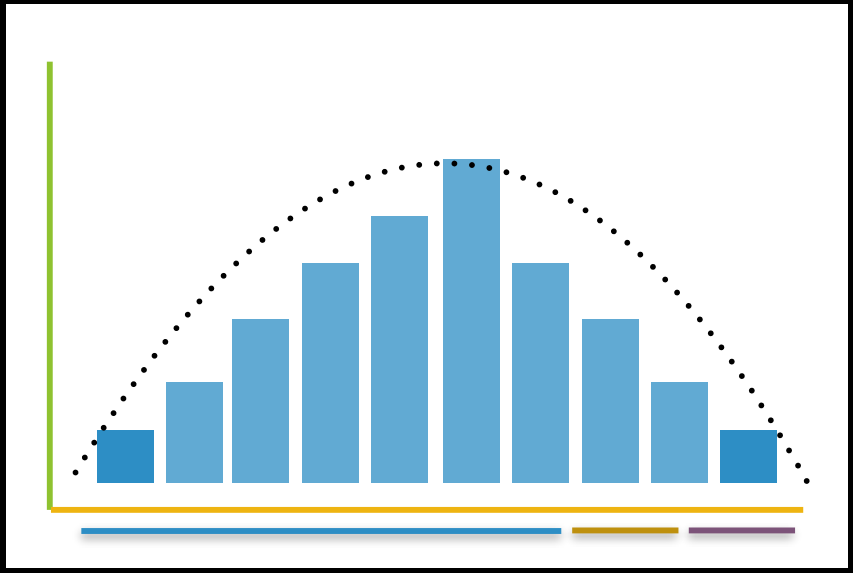
- Histograms are bucketed distributions of numeric datasets
- In a Normal Distribution, we talk about the Area Under the Curve



Building a Discovery Engine

Primer: What is a Histogram?

- Histograms are bucketed distributions of numeric datasets
- A Normal Distribution, fit into the Area Under the Curve
- Histograms allow us to understand the shape of a distribution

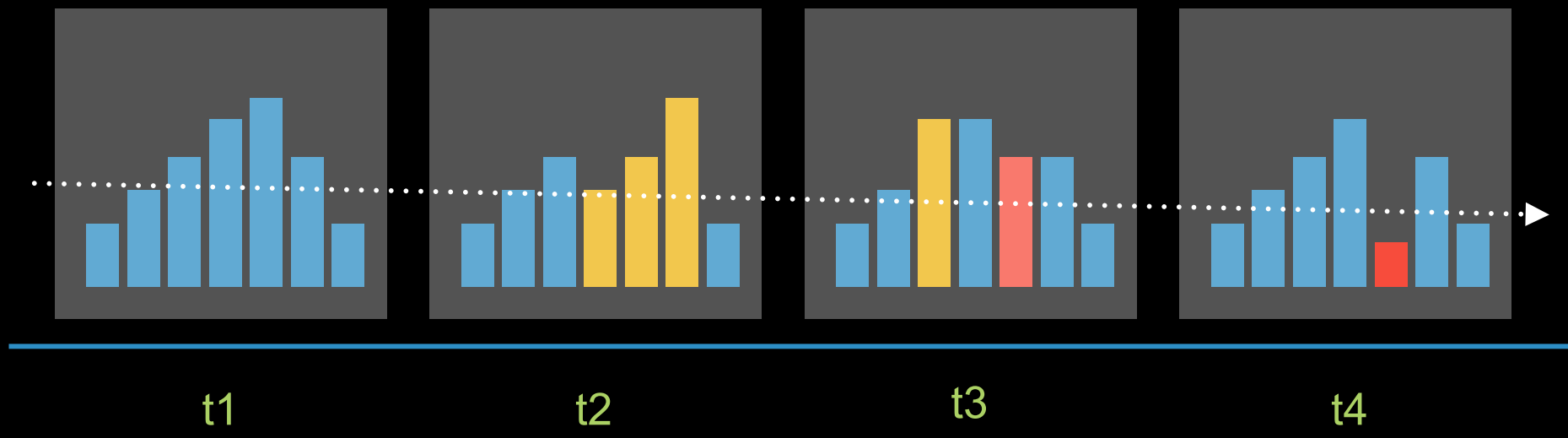


68-95-99.7 Rule

- 68%@1sd->mean,95@2x,99.7@3x)

Building a Discovery Engine

Goal: Time Series Histogram Analysis



Building a Discovery Engine

Deep Dive: Quantiles / Histograms in Spark

- Limitations: Not easily Aggregated or hard to extract

```
import org.apache.spark.sql.functions._
import spark.implicits._
case class SimpleData(name: String, value: Double)
val dataDF = spark.sparkContext.parallelize(Seq(SimpleData("a", 2.0),SimpleData("b", 3.0))).toDF
> dataDF.stat.approxQuantile("value", Array(0,0.25,0.5,0.75,1), 0.01)
res2: Array[Double] = Array(2.0, 2.0, 2.0, 3.0, 3.0)
> dataDF.select("value").map { _.getAs[Double](0) }.rdd.histogram(6)
res7: (Array[Double], Array[Long]) = (
  Array(2.0, 2.1666666666666665, 2.3333333333333335, 2.5, 2.6666666666666665, 2.8333333333333335, 3.0),
  Array(1, 0, 0, 0, 0, 1))
```

Building a Discovery Engine

Deep Dive: Aggregation Extension Options

```scala
abstract class InsightsAggregation extends UserDefinedAggregateFunction {
  type T<: Product
  val log: Logger

  def productEncoder: Encoder[T]
  def productSchema: StructType = productEncoder.schema

  lazy val fieldIndex: Map[String, Int] = productSchema.fieldNames.map { fieldName =>
    fieldName -> productSchema.fieldIndex(fieldName)
  }.toMap

  val fieldLength: Int

  override def inputSchema: StructType = InsightsAggregation.inputStringRow
  override def bufferSchema: StructType = productSchema
  override def dataType: DataType = productSchema
  override def deterministic: Boolean = true

  override def initialize(buffer: MutableAggregationBuffer): Unit = {
    (0 to fieldLength).foreach { num =>
      buffer(num) = 0L
    }
  }

  override def merge(cache: MutableAggregationBuffer, state: Row): Unit = {
    (0 to fieldLength).foreach { num =>
      cache(num) = cache.getLong(num) + state.getLong(num)
    }
  }
```

Define a new UDF based off of the MutableAggregationBuffer's in Spark

```scala
basicAgg[BasicEventAggregate](callEvent => {
    val records = callEvent
        .groupBy($"eventId", window($"timestamp", s"$windowIn
        .agg(
            min("pdd") as "minPdd",
            round(avg("pdd"), 2) as "avgPdd",
            max("pdd") as "maxPdd",
            min("duration") as "minDuration",
            round(avg("duration"), 2) as "avgDuration",
            max("duration") as "maxDuration",
            count("eventId") as "total",
            HistogramAggregation($"pdd") as "histogram"
        )
    // transform to final Dataset
    asBasicAggregates(records)
    })
```

Github https://bit.ly/2ISl3BS

Building a Discovery Engine

Deep Dive: Aggregation Extension Options

- Upside: Things are human readable

```
basicAgg[BasicEventAggregate](callEvent => {
    val records = callEvent
        .groupBy($"eventId", window($"timestamp", s"$windowInterval
        .agg(
            min("pdd") as "minPdd",
            round(avg("pdd"), 2) as "avgPdd",
            max("pdd") as "maxPdd",
            min("duration") as "minDuration",
            round(avg("duration"), 2) as "avgDuration",
            max("duration") as "maxDuration",
            count("eventId") as "total",
            HistogramAggregation($"pdd") as "histogram"
        )
      // transform to final Dataset
      asBasicAggregates(records)
      })
```

Building a Discovery Engine

Deep Dive: Aggregation Extension Options

- Upside: Things are human readable
- Downside: Performance* / Control*

```
basicAgg[BasicEventAggregate](callEvent => {
    val records = callEvent
        .groupBy($"eventId", window($"timestamp", s"$windowInterval
        .agg(
            min("pdd") as "minPdd",
            round(avg("pdd"), 2) as "avgPdd",
            max("pdd") as "maxPdd",
            min("duration") as "minDuration",
            round(avg("duration"), 2) as "avgDuration",
            max("duration") as "maxDuration",
            count("eventId") as "total",
            HistogramAggregation($"pdd") as "histogram"
        )
      // transform to final Dataset
      asBasicAggregates(records)
      })
```

Building a Discovery Engine

Deep Dive: Aggregation Extension Options : Final Thoughts

- Ditch the DataFrame's unstructured API and move over to Datasets typed API.
- This enables the use of flatMapGroupsWithState
- Now with a little effort you have full control over the aggregation strategy

Next up: after the break

Building a Discovery Engine: Part 2

1. Define the Problem (1m)
2. Build a plan of Attack (1m)
3. Model the Data (2m)
5. Build the Streaming Discovery Engine (~20m)
6. Hopefully time for lessons learned and Questions!


Global Telephony is tough.

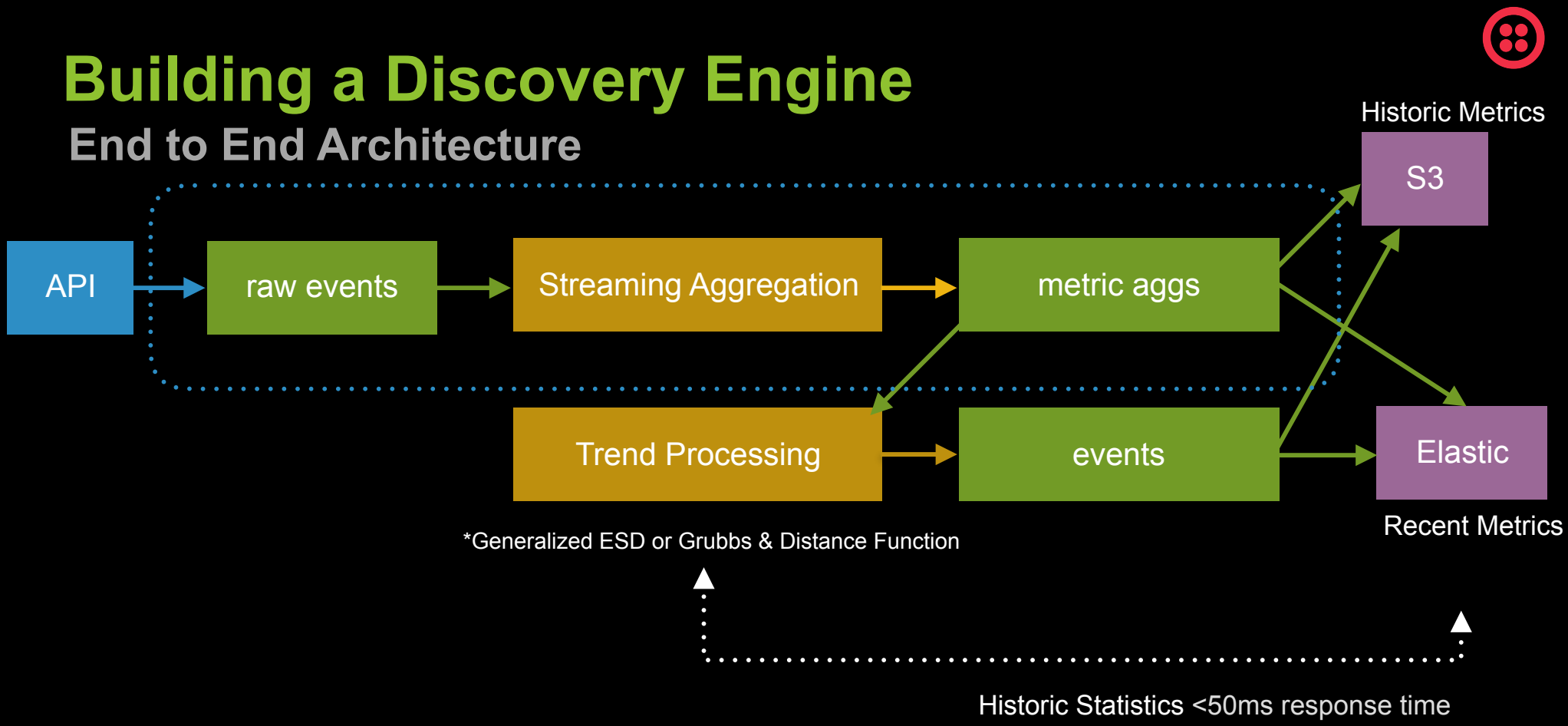
Simply Connecting two endpoints is ripe for failure.

Given Networks can go down at anytime.


We should build a system to
auto-discover trends in (~real-time) for our KPMs
and automatically report them...

Building a Discovery Engine

End to End Architecture



Building a Discovery Engine

Define the raw event

- Must provide the information necessary for aggregation

```
message CallEvent {
  // when the event occured utc
  optional uint64 event_time
  // when we received said event utc ***
  optional uint64 logged_event_time
  // unique identifier for said event - **
  optional string event_id
  // encapsulates a logical end to end ro
  optional string route_id
  // used to store multiple types in this
  optional CallEventType event_type
  // when event type is signaling_event t
  optional SignalingEvent signaling_event
  // use with the event_type and logical
  optional Dimensions event_dimensions
}
```

Building a Discovery Engine

Define the raw event

- Must provide the information necessary for aggregation
- id must be idempotent. Brownie points if it can be reused in Hash Partitioner

```
message CallEvent {
  // when the event occured utc
  optional uint64 event_time
  // when we received said event utc ***
  optional uint64 logged_event_time
  // unique identifier for said event - **
  optional string event_id
  // encapsulates a logical end to end ro
  optional string route_id
  // used to store multiple types in this
  optional CallEventType event_type
  // when event type is signaling_event t
  optional SignalingEvent signaling_event
  // use with the event_type and logical
  optional Dimensions event_dimensions
}
```

Building a Discovery Engine

Define the raw event

- Must provide the information necessary for aggregation
- id must be idempotent. *dedup key - Brownie points if it can be reused in Hash Partitioner in spark and kafka!
- event_dimensions + event_group can create great grouping hash

```
message CallEvent {
  // when the event occured utc
  optional uint64 event_time
  // when we received said event utc ***
  optional uint64 logged_event_time
  // unique identifier for said event - **
  optional string event_id
  // encapsulates a logical end to end ro
  optional string route_id
  // used to store multiple types in this
  optional CallEventType event_type
  // when event type is signaling_event t
  optional SignalingEvent signaling_event
  // use with the event_type and logical
  optional Dimensions event_dimensions
}
```

Building a Discovery Engine

Define the raw event

- Allows for event evolution while supporting backwards compatibility.

```
message SignalingEvent {
  optional string name                    = 1;
  optional SignalingEventType event_type = 2;
  optional PddEvent pdd                   = 3;
  optional CallStateEvent call_state      = 4;
}

enum SignalingEventType {
  unknown_event_type  = 0;
  pdd                 = 1;
  call_state          = 2;
}

message PddEvent {
  optional float pdd = 1;
}
```

Building a Discovery Engine

Define the raw event

- Allows for event evolution while supporting backwards compatibility.
- Enum ordinals can be used in ML jobs without the need for string encoding

```
message SignalingEvent {
  optional string name                    = 1;
  optional SignalingEventType event_type = 2;
  optional PddEvent pdd                   = 3;
  optional CallStateEvent call_state      = 4;
}

enum SignalingEventType {
  unknown_event_type  = 0;
  pdd                 = 1;
  call_state          = 2;
}

message PddEvent {
  optional float pdd = 1;
}
```

Building a Discovery Engine

Define the aggregation

- Includes logical time series Window
- Stores all Dimensional and Statistic Information

```
message MetricAggregation {
  optional string metric                = 1;
  optional uint64 window_start          = 2;
  optional uint64 window_end            = 3;
  optional string window_interval       = 4;
  optional uint32 samples               = 5;
  optional Stats stats                  = 6;
  optional Histogram histogram          = 7;
  optional Dimensions dimensions        = 8;
  optional string dimension_hash        = 10;
}
```

Building a Discovery Engine

Define the aggregation

- Includes logical time series Window
- Stores all Dimensional and Statistic Information
- Can be used together as a strong Binary Group Key

```
message MetricAggregation {
  optional string metric                = 1;
  optional uint64 window_start          = 2;
  optional uint64 window_end            = 3;
  optional string window_interval       = 4;
  optional uint32 samples               = 5;
  optional Stats stats                  = 6;
  optional Histogram histogram          = 7;
  optional Dimensions dimensions        = 8;
  optional string dimension_hash        = 10;
}
```

Building a Discovery Engine

Why hash the dimensions?

- Hashing the dimensions of the aggregation simplifies the groupBy or groupByKey mechanics.

- Use guava since it ships with Spark and works well at scale for Hashing.

```
def generateId(bytes: Array[Byte]): String = {
  val hf = Hashing.murmur3_128()
  val hc = hf.hashBytes(bytes)
  BaseEncoding.base64Url()
    .omitPadding()
    .encode(hc.asBytes())
}
```

Next up

- Putting together what we've learned with Spark
- Available in the content @ https://bit.ly/2ISI3BS
- Walk through of critical components of the Application

Building a Discovery Engine

```
override def run(): Unit = {
  // creates a streaming query that will aggregate events from kafka
  // and produce statistical histogram aggregations to be consumed downstream
  EventAggregation(config)
    .process(readKafkaStream())(spark)
    .writeStream
    .queryName("streaming.trend.discovery")
    .outputMode(eventProcessor.outputMode)
    .trigger(Trigger.ProcessingTime(Duration(config.triggerInterval)))
    .format("kafka")
    .option("topic", "spark.summit.call.aggregations")
    .options(Map(
      "checkpointLocation" -> config.checkpointPath,
      "kafka.producer.config1" -> "",
      "kafka.producer.config2" -> ""
    ))
    .start()
}
```

Creates the pipeline

* process(KDSR), inject SparkSession (implicit*)
* write back to Kafka
* checkpoint, etc

Building a Discovery Engine

1. Transform (protobuf -> Product)
2. Actual Aggregation

* groupedBy BinaryType
* fMGwS keyed by BinaryType
* EventTimeTimeout*

```
def process(df: DataFrame)(implicit spark: SparkSession): Dataset[Metrics.MetricAggregation] = {
  import spark.implicits._
  import EventAggregation.transforms._
  import EventAggregation.implicits._
  implicit val metricAggsEncoder: ExpressionEncoder[MetricAggregation] = EventAggregation.implicits.aggreationEnco
  val watermarkInterval = config.watermarkIntervalMinutes

  // 1. Convert from the kafka data frame to an instance of our Metric
  val groupedMetrics = df.mapPartitions(
    _.flatMap { kd =>
      bytesToMetricRow(kd.getAs[Array[Byte]]("value"), windowInterval)
    }
  )(metricEncoder)
  // 2.
  logger.info(s"aggregation outputMode=$outputMode windowIntervalMinutes=${config.windowIntervalMinutes} watermark
  groupedMetrics
    .withWatermark("timestamp", s"${config.watermarkIntervalMinutes} minutes")
    .groupByKey(_.metricAggregation)
    .flatMapGroupsWithState[Array[Byte], MetricAggregation](outputMode, GroupStateTimeout.EventTimeTimeout())(metr
}
```

Building a Discovery Engine

- StateSpec takes a StateFunc
- This is for Key/Value pair grouped data
- Key is [A]. It can be Binary. Cool
- Take an Iterator[B]
- Stores a GroupState[C]
- Emits Iterator[D]

```
def metricAggregator(
  metricAggregation: Array[Byte],
  metrics: Iterator[ProductMetric],
  state: GroupState[Array[Byte]]): Iterator[MetricAggregation] = {
```

Building a Discovery Engine

metricAggregation

```
val metricData = state.getOption match {
  case Some(lastState) =>
    logger.info(s"recovered the last state")
    // union the last and new sketches
    val lastSketch = DoubleSketch(lastState)
    val currentSketch = DoubleSketch.sketch(k, metrics.map { m => m.value }.toArray)
    val unionSketch = DoubleSketch.union(k, lastSketch, currentSketch)
    // store updated metrics
    logger.info(s"state=merge num.records=${unionSketch.getN}")
    unionSketch.toByteArray(true)
  case None =>
    val s = DoublesSketch.builder().setK(k).build()
    metrics.foreach { m =>
      s.update(m.value)
    }
    logger.info(s"state=new num.records=${s.getN}")
    s.toByteArray(true)
}
state.update(metricData)
val updateTimeout = aggregation.getWindowEnd
logger.info(s"updating.timeout.timestamp.to=$updateTimeout")
state.setTimeoutTimestamp(updateTimeout)
```

Building a Discovery Engine

- Data Sketches from Yahoo* provides serializable Quantile Sketches with Histogram capabilities.
- Data Sketches are a light weight representation of a complete set of data (on a diet - think HyperLogLog)
- We use this on our team to enable performant aggregation of complicated statistics without storing all rows in Spark memory.

```
object DoubleSketch {

  val k: Int = 512

  def apply(serialized: Array[Byte]): DoublesSketch = DoublesSketch.wrap(Memory.wrap(serialized))
  def sketch(data: Array[Double]): DoublesSketch = sketch(k, data)

  def sketch(k: Int, data: Array[Double]): DoublesSketch = {
    val s = DoublesSketch.builder().setK(k).build()
    data.foreach { s.update }
    s
  }

  def union(k: Int, a: Array[Double], b: Array[Double]): DoublesSketch = {
    val u = DoublesUnion.builder().setMaxK(k).build()
    u.update(sketch(k, a))
    u.update(sketch(k, b))
    u.getResult
  }

  def union(k: Int, a: DoublesSketch, b: DoublesSketch): DoublesSketch = {
    val u = DoublesUnion.builder().setMaxK(k).build()
    u.update(a)
    u.update(b)
    u.getResult
  }

}
```

Building a Discovery Engine

- Data Sketches from Yahoo* provides serializable Quantile Sketches with Histogram capabilities.
- Data Sketches are a light weight representation of a complete set of data (on a diet - think HyperLogLog)
- We use this on our team to enable performant aggregation of complicated statistics without storing all rows in Spark memory.

```
object DoubleSketch {

  val k: Int = 512

  def apply(serialized: Array[Byte]): DoublesSketch = DoublesSketch.wrap(Memory.wrap(serialized))
  def sketch(data: Array[Double]): DoublesSketch = sketch(k, data)

  def sketch(k: Int, data: Array[Double]): DoublesSketch = {
    val s = DoublesSketch.builder().setK(k).build()
    data.foreach { s.update }
    s
  }

  def union(k: Int, a: Array[Double], b: Array[Double]): DoublesSketch = {
    val u = DoublesUnion.builder().setMaxK(k).build()
    u.update(sketch(k, a))
    u.update(sketch(k, b))
    u.getResult
  }

  def union(k: Int, a: DoublesSketch, b: DoublesSketch): DoublesSketch = {
    val u = DoublesUnion.builder().setMaxK(k).build()
    u.update(a)
    u.update(b)
    u.getResult
  }

}
```

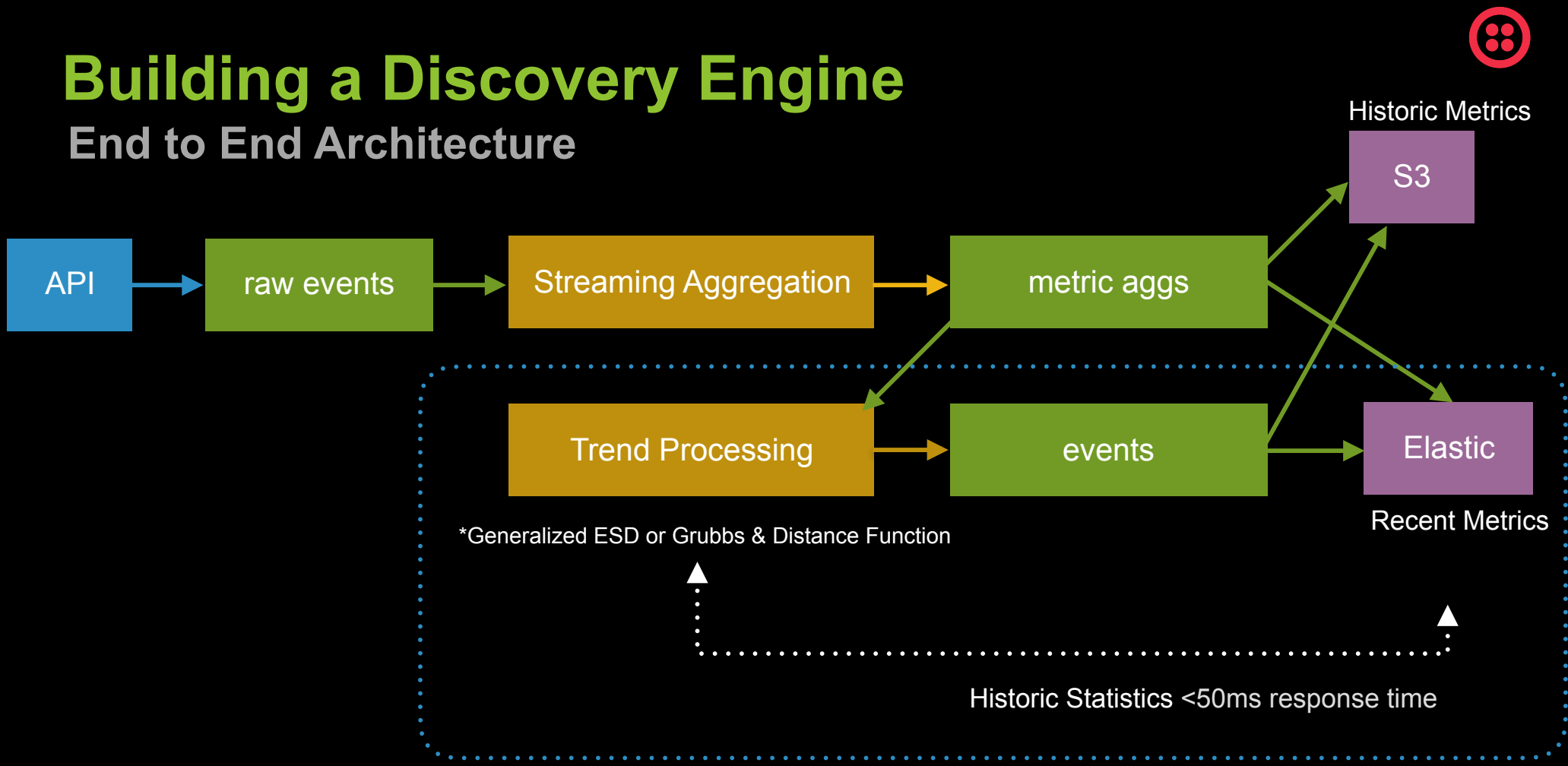
Building a Discovery Engine

- Testing should feel good.
- Ensure you solve Encoder issues early
- Make sure you don't add regression's into your 24/7/365 apps!

```
test("Should aggregate call events") {
  import session.implicits._
  val appConfig = appConfigForTest()
  val scenario = TestHelper.loadScenario[CallEvent](s"$pathToTestScenarios/pdd_events.json")
  val scenarioIter = scenario.toIterator
  scenario.nonEmpty shouldBe true

  testUtils.createTopic(kafkaTopic, partitions, overwrite = true)
  sendNextMessages(scenarioIter, 30, _.getEventId, _.getLoggedEventTime)

  val trendDiscoveryApp = new TrendDiscoveryApp(appConfigForTest(), session)
  val eventAggregation = EventAggregation(appConfig)

  eventAggregation.process(trendDiscoveryApp.readKafkaStream())(session)
    .writeStream
    .queryName("calleventaggs")
    .format("memory")
    .outputMode(eventAggregation.outputMode)
    .start()
    .processAllAvailable()

  val res = session
    .sql("select avg(stats.p99) from calleventaggs")
    .collect()
    .map { r =>
      r.getAs[Double](0) }
    .head

  DiscoveryUtils.round(res) shouldEqual 7.13
```

Building a Discovery Engine

End to End Architecture



Building a Discovery Engine

Anatomy of the application: Streaming Aggregation Query

Building the Continuous Trend Detector

1. Read from Kafka (raw events)
2. Look up *historic statistics from Elastic Search or Redis
3. Generate an anomaly stream based off of distance to statistical norm
4. Emit to Kafka for downstream use

Discovering trends in massive data sets doesn't come for free. There are core principals and ideologies that must be followed in order to extract interesting patterns from your data.

Questions? / Thanks

@newfront

github@newfront

shaines@twilio.com

Thanks to Marek Radonsky and Rohit Gupta from the Voice Insights team who helped make these patterns possible.