Amazon Kinesis Data Streams

- Collect and store streaming data in **real-time**

Kinesis Data Streams

- Retention between up to 365 days
- Ability to reprocess (replay) data by consumers
- Data can’t be deleted from Kinesis (until it expires)
- Data up to 1MB (typical use case is lot of “small” **real-time data**)
- Data ordering guarantee for data with the same “Partition ID”
- At-rest KMS encryption, in-flight HTTPS encryption
- Kinesis Producer Library (KPL) to write an optimized producer application
- Kinesis Client Library (KCL) to write an optimized consumer application

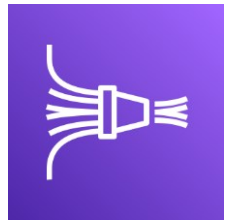
Kinesis Data Streams – Capacity Modes

- **Provisioned mode:**
  - Choose number of shards
  - Each shard gets 1MB/s in (or 1000 records per second)
  - Each shard gets 2MB/s out
  - Scale manually to increase or decrease the number of shards
  - You pay per shard provisioned per hour

- **On-demand mode:**
  - No need to provision or manage the capacity
  - Default capacity provisioned (4 MB/s in or 4000 records per second)
  - Scales automatically based on observed throughput peak during the last 30 days
  - Pay per stream per hour & data in/out per GB

Amazon Data Firehose

Amazon Data Firehose



- *Note: used to be called "Kinesis Data Firehose"*
- Fully Managed Service
  - Amazon Redshift / Amazon S3 / Amazon OpenSearch Service
  - 3rd party: Splunk / MongoDB / Datadog / NewRelic / …
  - Custom HTTP Endpoint
- Automatic scaling, serverless, pay for what you use
- **Near Real-Time** with buffering capability based on size / time
- Supports CSV, JSON, Parquet, Avro, Raw Text, Binary data
- Conversions to Parquet / ORC, compressions with gzip / snappy
- Custom data transformations using AWS Lambda (ex: CSV to JSON)

Kinesis Data Streams vs Amazon Data Firehose

Kinesis Data Streams

- Streaming data collection
- Producer & Consumer code
- Real-time
- Provisioned / On-Demand mode
- Data storage up to 365 days
- Replay Capability

Amazon Data Firehose

- Load streaming data into S3 / Redshift / OpenSearch / 3rd party / custom HTTP
- Fully managed
- Near real-time
- Automatic scaling
- No data storage
- Doesn't support replay capability

Troubleshooting Kinesis Data Stream Producers: Performance

- Writing is too slow
  - Service limits may be exceeded. Check for throughput exceptions, see what operations are being throttled. Different calls have different limits.
  - There are shard-level limits for writes and reads
  - Other operations (ie, CreateStream, ListStreams, DescribeStreams) have stream-level limits of 5-20 calls per second
  - Select partition key to evenly distribute puts across shards
- Large producers
  - Batch things up. Use Kinesis Producer Library, PutRecords with multi-records, or aggregate records into larger files.
- Small producers (i.e. apps)
  - Use PutRecords or Kinesis Recorder in the AWS Mobile SDKs

Other Kinesis Data Stream Producer Issues

- Stream returns a 500 or 503 error
  - This indicates an AmazonKinesisException error rate above 1%
  - Implement a retry mechanism
- Connection errors from Flink to Kinesis
  - Network issue or lack of resources in Flink's environment
  - Could be a VPC misconfiguration
- Timeout errors from Flink to Kinesis
  - Adjust RequestTimeout and #setQueueLimit on FlinkKinesisProducer
- Throttling errors
  - Check for hot shards with enhanced monitoring (shard-level)
  - Check logs for "micro spikes" or obscure metrics breaching limits
  - Try a random partition key or improve the key's distribution
  - Use exponential backoff
  - Rate-limit

Troubleshooting Kinesis Data Stream Consumers

- Records get skipped with Kinesis Client Library
  - Check for unhandled exceptions on processRecords
- Records in same shard are processed by more than one processor
  - May be due to failover on the record processor workers
  - Adjust failover time
  - Handle shutdown methods with reason “ZOMBIE”
- Reading is too slow
  - Increase number of shards
  - maxRecords per call is too low
  - Your code is too slow (test an empty processor vs. yours)
- GetRecords returning empty results
  - This is normal, just keep calling GetRecords
- Shard Iterator expires unexpectedly
  - May need more write capacity on the shard table in DynamoDB
- Record processing falling behind
  - Increase retention period while troubleshooting
  - Usually insufficient resources
  - Monitor with GetRecords.IteratorAgeMilliseconds and `MillisBehindLatest`

Troubleshooting Kinesis Data Stream Consumers

- Lambda function can't get invoked
  - Permissions issue on execution role
  - Function is timing out (check max execution time)
  - Breaching concurrency limits
  - Monitor `IteratorAge` metric; it will increase is this is a problem
- `ReadProvisionedThroughputExceeded` exception
  - Throttling
  - Reshard your stream
  - Reduce size of GetRecords requests
  - Use enhanced fan-out
  - Use retries and exponential backoff

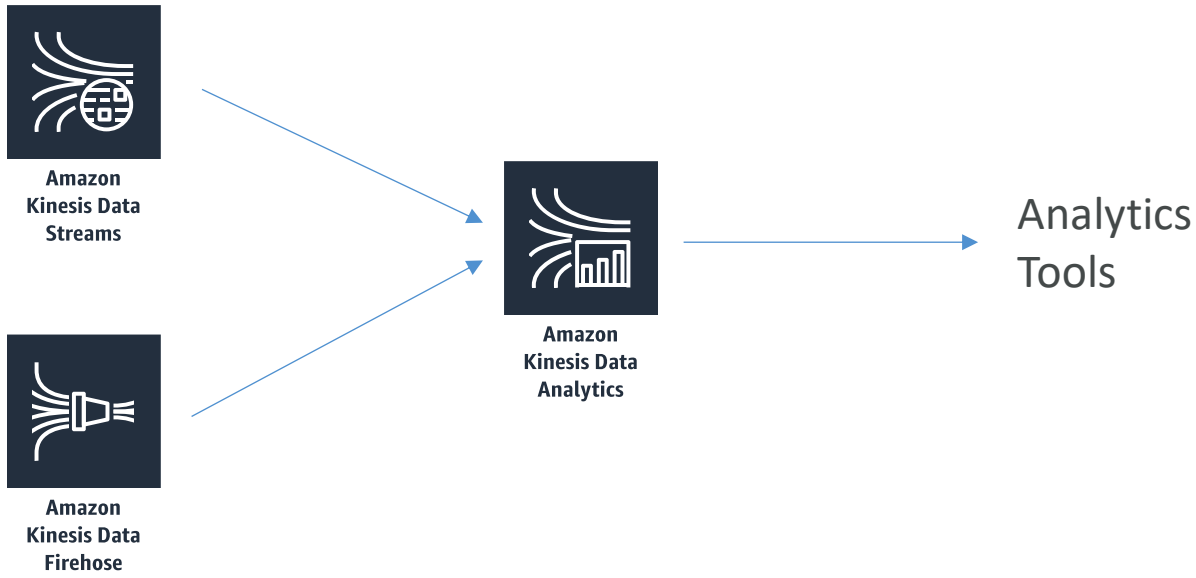
Troubleshooting Kinesis Data Stream Consumers

- High latency
  - Monitor with GetRecords.Latency and IteratorAge
  - Increase shards
  - Increase retention period
  - Check CPU and memory utilization (may need more memory)
- 500 errors
  - Same as producers – indicates a high error rate (>1%)
  - Implement a retry mechanism
- Blocked or stuck KCL application
  - Optimize your `processRecords` method
  - Increase maxLeasesPerWorker
  - Enable KCL debug logs

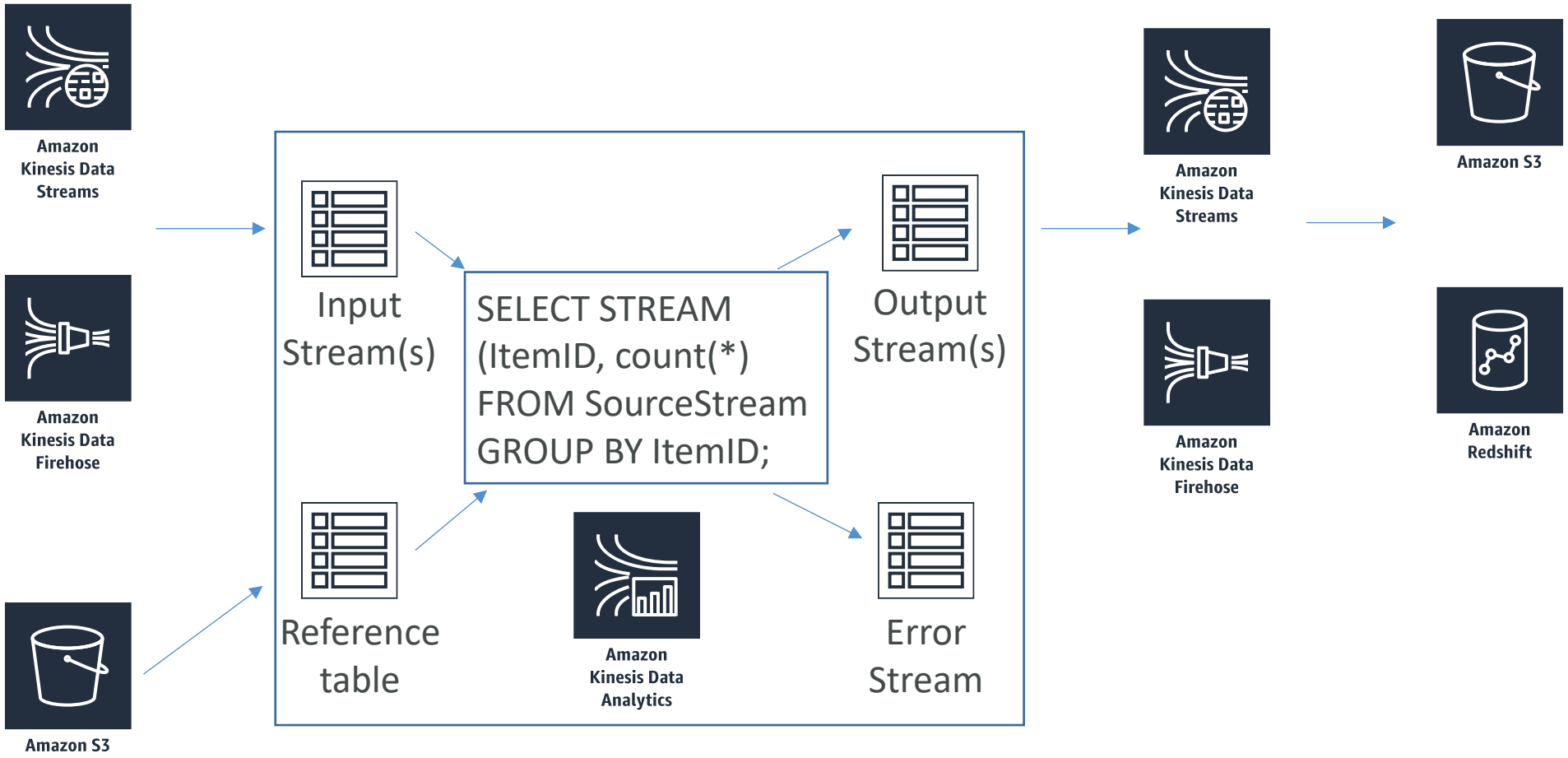
Kinesis Data Analytics / Managed Service for Apache Flink

Querying streams of data

Amazon Kinesis Data Analytics for SQL Applications

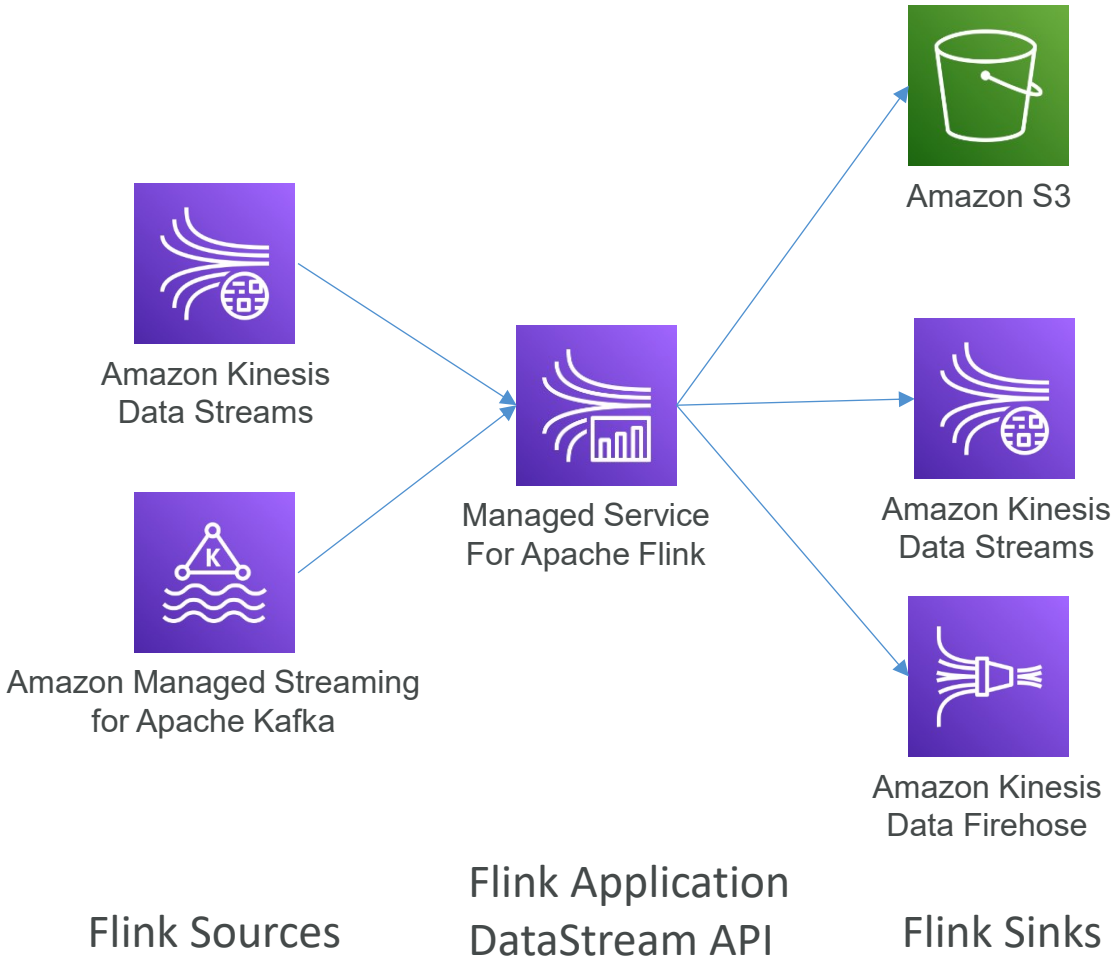


In more depth…



Managed Service for Apache Flink

- Formerly Kinesis Data Analytics for Apache Flink or for Java
  - Kinesis Data Analytics always used Flink under the hood
  - But now supports Python and Scala
  - Flink is a framework for processing data streams
- MSAF integrates Flink with AWS
  - Instead of using SQL, you can develop your own Flink application from scratch and load it into MSAF via S3
- In addition to the DataStream API, there is a Table API for SQL access
- Serverless



Common use-cases

- Streaming ETL
- Continuous metric generation
- Responsive analytics

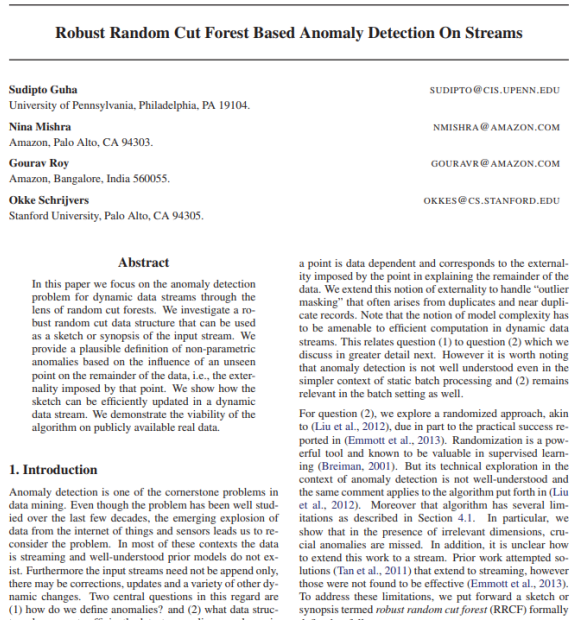
Amazon Kinesis Data Analytics

Kinesis Analytics

- Pay only for resources consumed (but it's not cheap)
  - Charged by Kinesis Processing Units (KPU's) consumed per hour
  - 1 KPU = 1 vCPU + 4GB
- Serverless; scales automatically
- Use IAM permissions to access streaming source and destination(s)
- Schema discovery

RANDOM_CUT_FOREST

- SQL function used for anomaly detection on numeric columns in a stream
- They're especially proud of this because they published a paper on it
- It's a novel way to identify outliers in a data set so you can handle them however you need to
- Example: detect anomalous subway ridership during the NYC marathon



Robust Random Cut Forest Based Anomaly Detection On Streams

Sudipto Guha SUDIPTO@CIS.UPENN.EDU
University of Pennsylvania, Philadelphia, PA 19104.

Nina Mishra NMISHRA@AMAZON.COM
Amazon, Palo Alto, CA 94303.

Gourav Roy GOURAVR@AMAZON.COM
Amazon, Bangalore, India 560055.

Okke Schrijvers OKKES@CS.STANFORD.EDU
Stanford University, Palo Alto, CA 94305.

Abstract

In this paper we focus on the anomaly detection problem for dynamic data streams through the lens of random cut forests. We investigate a robust random cut data structure that can be used as a sketch or synopsis of the input stream. We provide a plausible definition of non-parametric anomalies based on the influence of an unseen point on the remainder of the data, i.e., the externality imposed by that point. We show how the sketch can be efficiently updated in a dynamic data stream. We demonstrate the viability of the algorithm on publicly available real data.

1. Introduction

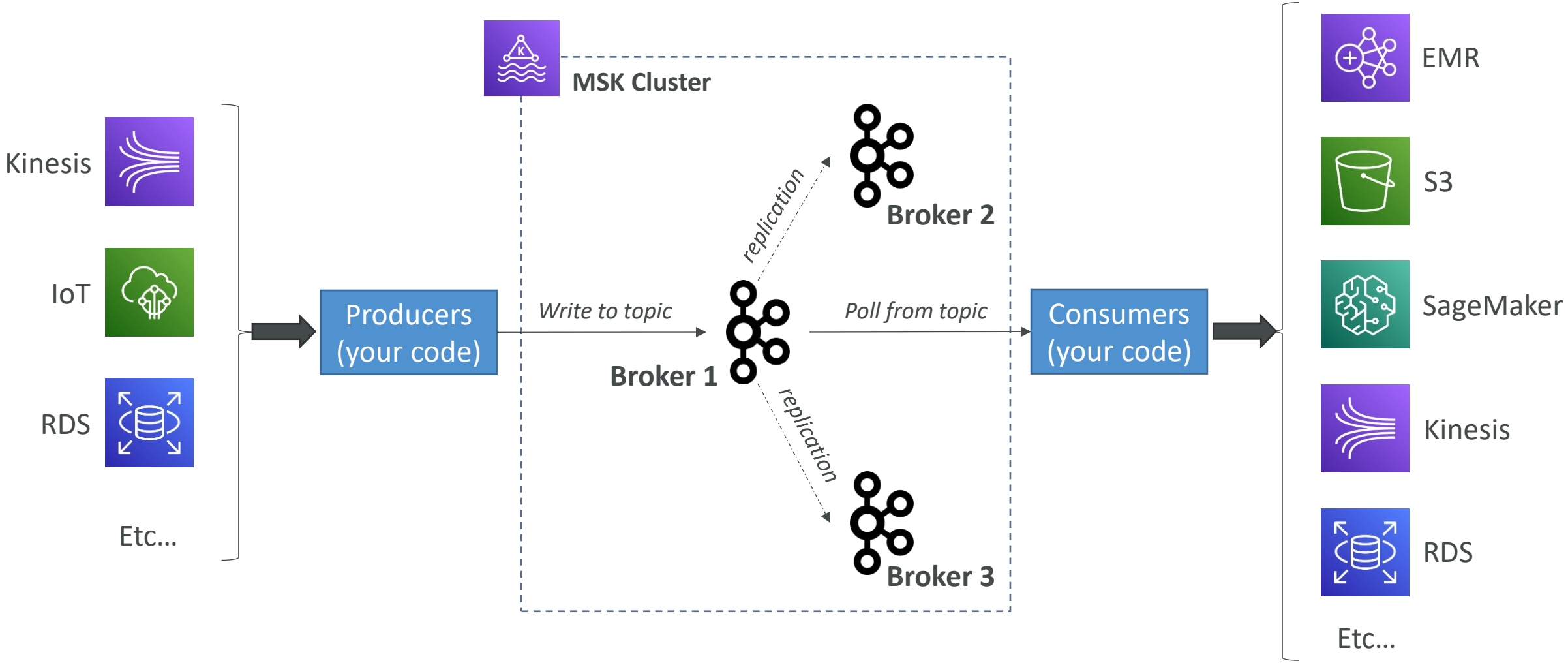
Anomaly detection is one of the cornerstone problems in data mining. Even though the problem has been well studied over the last few decades, the emerging explosion of data from the internet of things and sensors leads us to reconsider the problem. In most of these contexts the data is streaming and well-understood prior models do not exist. Furthermore the input streams need not be append only, there may be corrections, updates and a variety of other dynamic changes. Two central questions in this regard are (1) how do we define anomalies? and (2) what data structure a point is data dependent and corresponds to the externality imposed by the point in explaining the remainder of the data. We extend this notion of externality to handle "outlier masking" that often arises from duplicates and near duplicate records. Note that the notion of model complexity has to be amenable to efficient computation in dynamic data streams. This relates question (1) to question (2) which we discuss in greater detail next. However it is worth noting that anomaly detection is not well understood even in the simpler context of static batch processing and (2) remains relevant in the batch setting as well.

For question (2), we explore a randomized approach, akin to (Liu et al., 2012), due in part to the practical success reported in (Emmott et al., 2013). Randomization is a powerful tool and known to be valuable in supervised learning (Breiman, 2001). But its technical exploration in the context of anomaly detection is not well-understood and the same comment applies to the algorithm put forth in (Liu et al., 2012). Moreover that algorithm has several limitations as described in Section 4.1. In particular, we show that in the presence of irrelevant dimensions, crucial anomalies are missed. In addition, it is unclear how to extend this work to a stream. Prior work attempted solutions (Tan et al., 2011) that extend to streaming, however those were not found to be effective (Emmott et al., 2013). To address these limitations, we put forward a sketch or synopsis termed *robust random cut forest* (RRCF) formally

Amazon Managed Streaming for Apache Kafka (Amazon MSK)

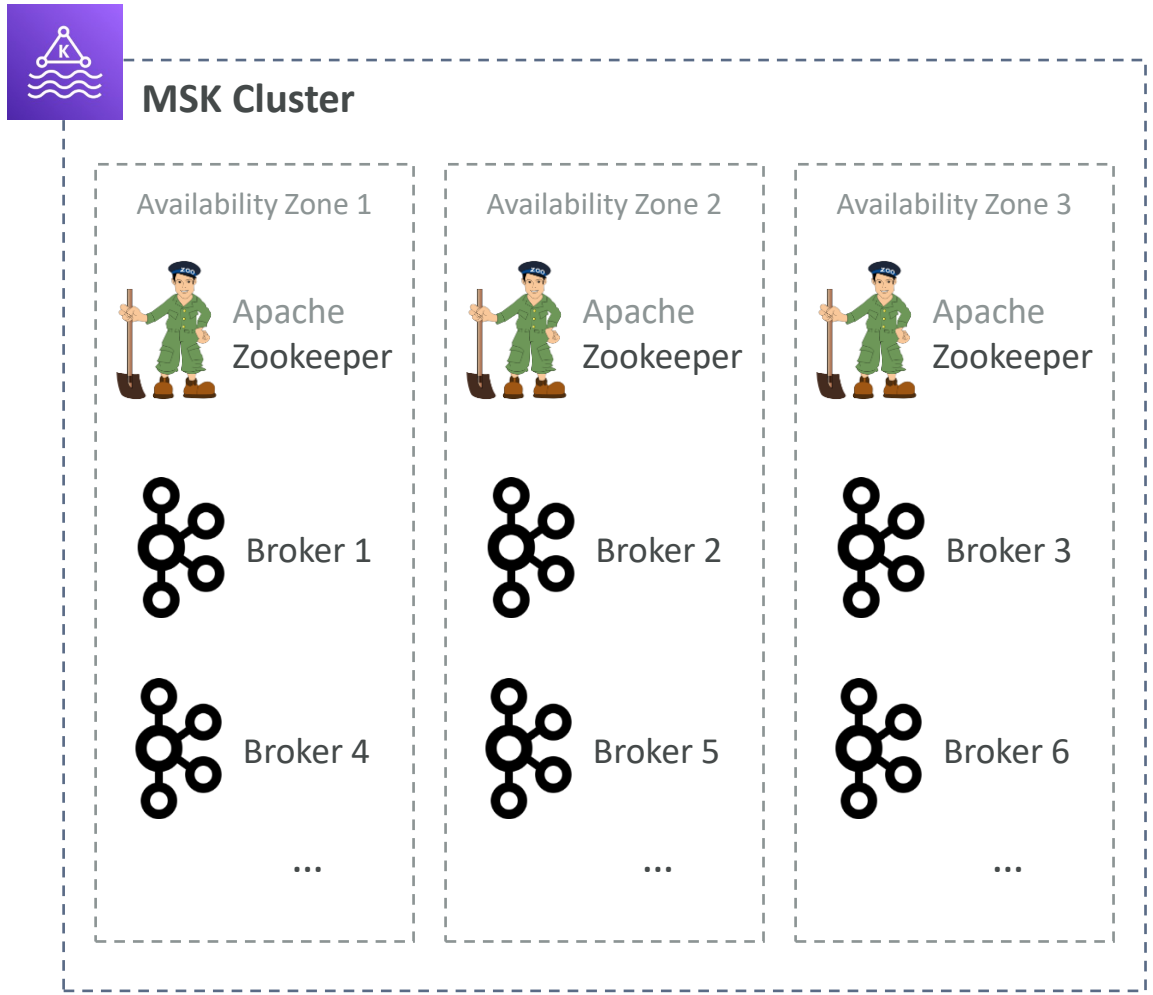
- Alternative to Kinesis (Kafka vs Kinesis next lecture)
- Fully managed Apache Kafka on AWS
  - Allow you to create, update, delete clusters
  - MSK creates & manages Kafka brokers nodes & Zookeeper nodes for you
  - Deploy the MSK cluster in your VPC, multi-AZ (up to 3 for HA)
  - Automatic recovery from common Apache Kafka failures
  - Data is stored on EBS volumes
- You can build producers and consumers of data
- Can create custom configurations for your clusters
  - Default message size of 1MB
  - **Possibilities of sending large messages (ex: 10MB) into Kafka after custom configuration**

Apache Kafka at a high level



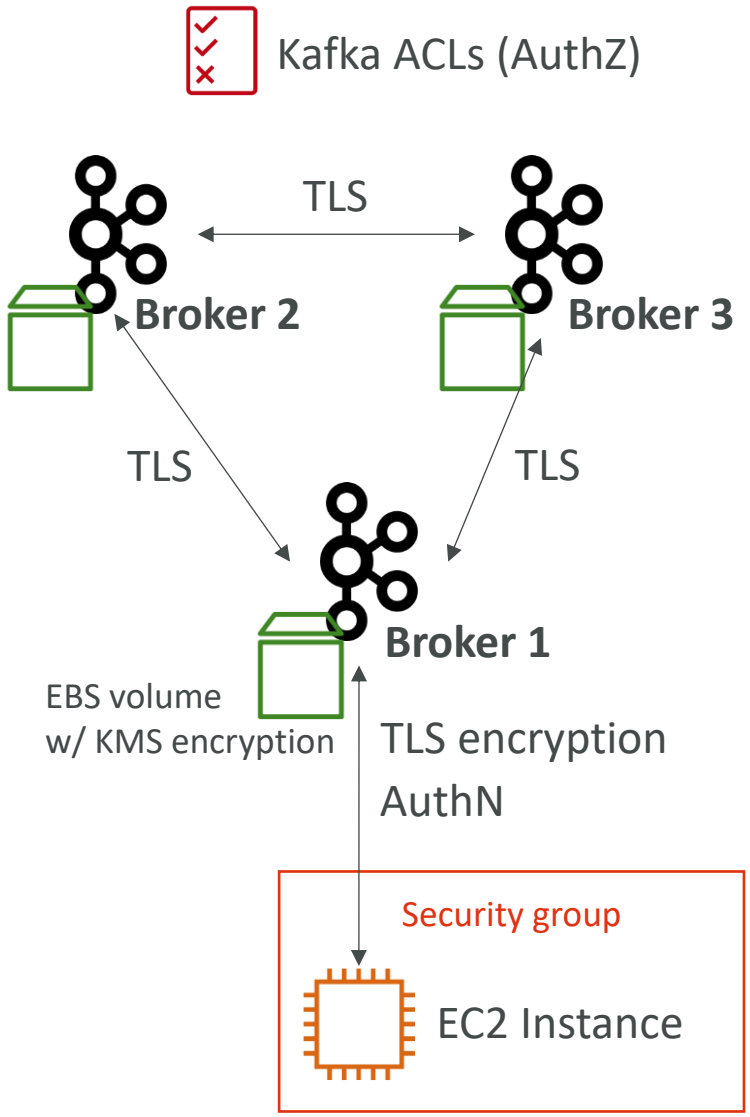
MSK – Configurations

- Choose the number of AZ (3 – recommended, or 2)
- Choose the VPC & Subnets
- The broker instance type (ex: kafka.m5.large)
- The number of brokers per AZ (can add brokers later)
- Size of your EBS volumes (1GB – 16TB)



MSK – Security

- **Encryption:**
  - Optional in-flight using TLS between the brokers
  - Optional in-flight with TLS between the clients and brokers
  - At rest for your EBS volumes using KMS
- **Network Security:**
  - Authorize specific security groups for your Apache Kafka clients
- **Authentication & Authorization (important):**
  - Define who can read/write to which topics
  - Mutual TLS (AuthN) + Kafka ACLs (AuthZ)
  - SASL/SCRAM (AuthN) + Kafka ACLs (AuthZ)
  - IAM Access Control (AuthN + AuthZ)

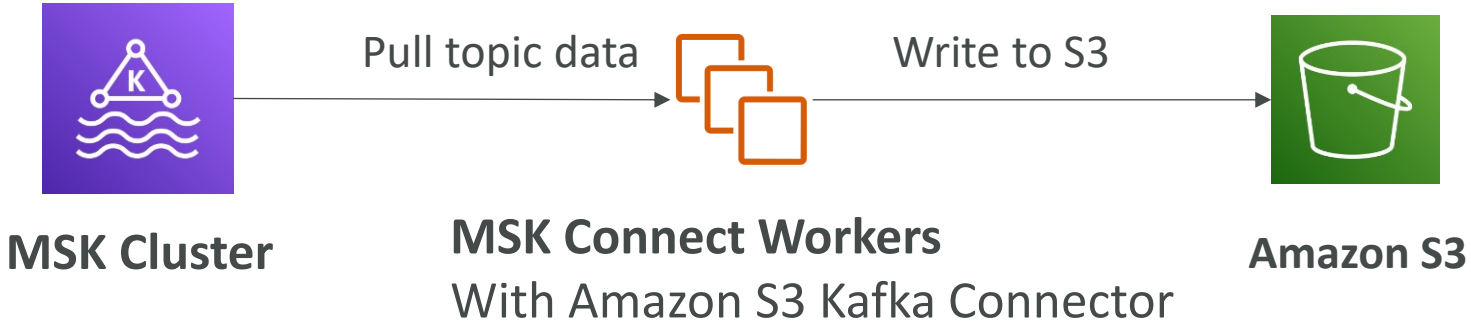


MSK – Monitoring

- **CloudWatch Metrics**
  - Basic monitoring (cluster and broker metrics)
  - Enhanced monitoring (++enhanced broker metrics)
  - Topic-level monitoring (++enhanced topic-level metrics)
- **Prometheus (Open-Source Monitoring)**
  - Opens a port on the broker to export cluster, broker and topic-level metrics
  - Setup the JMX Exporter (metrics) or Node Exporter (CPU and disk metrics)
- **Broker Log Delivery**
  - Delivery to CloudWatch Logs
  - Delivery to Amazon S3
  - Delivery to Kinesis Data Streams

MSK Connect

- Managed Kafka Connect workers on AWS
- Auto-scaling capabilities for workers
- You can deploy any Kafka Connect connectors to MSK Connect as a **plugin**
  - Amazon S3, Amazon Redshift, Amazon OpenSearch, Debezium, etc…
- Example pricing: Pay $0.11 per worker per hour



MSK Serverless

- **Run Apache Kafka on MSK without managing the capacity**
- **MSK automatically provisions resources and scales compute & storage**
- You just define your topics and your partitions and you're good to go!
- Security: IAM Access Control for all clusters

- Example Pricing:
  - $0.75 per cluster per hour = $558 monthly per cluster
  - $0.0015 per partition per hour = $1.08 monthly per partition
  - $0.10 per GB of storage each month
  - $0.10 per GB in
  - $0.05 per GB out

Kinesis Data Streams vs Amazon MSK

Kinesis Data Streams

- 1 MB message size limit
- Data Streams with Shards
- Shard Splitting & Merging
- TLS In-flight encryption
- KMS At-rest encryption
- Security:
  - IAM policies for AuthN/AuthZ

Amazon MSK

- 1MB default, configure for higher (ex: 10MB)
- Kafka Topics with Partitions
- Can only add partitions to a topic
- PLAINTEXT or TLS In-flight Encryption
- KMS At-rest encryption
- Security:
  - Mutual TLS (AuthN) + Kafka ACLs (AuthZ)
  - SASL/SCRAM (AuthN) + Kafka ACLs (AuthZ)
  - IAM Access Control (AuthN + AuthZ)

Data Transformation, Integrity, and Feature Engineering

EMR

Elastic MapReduce

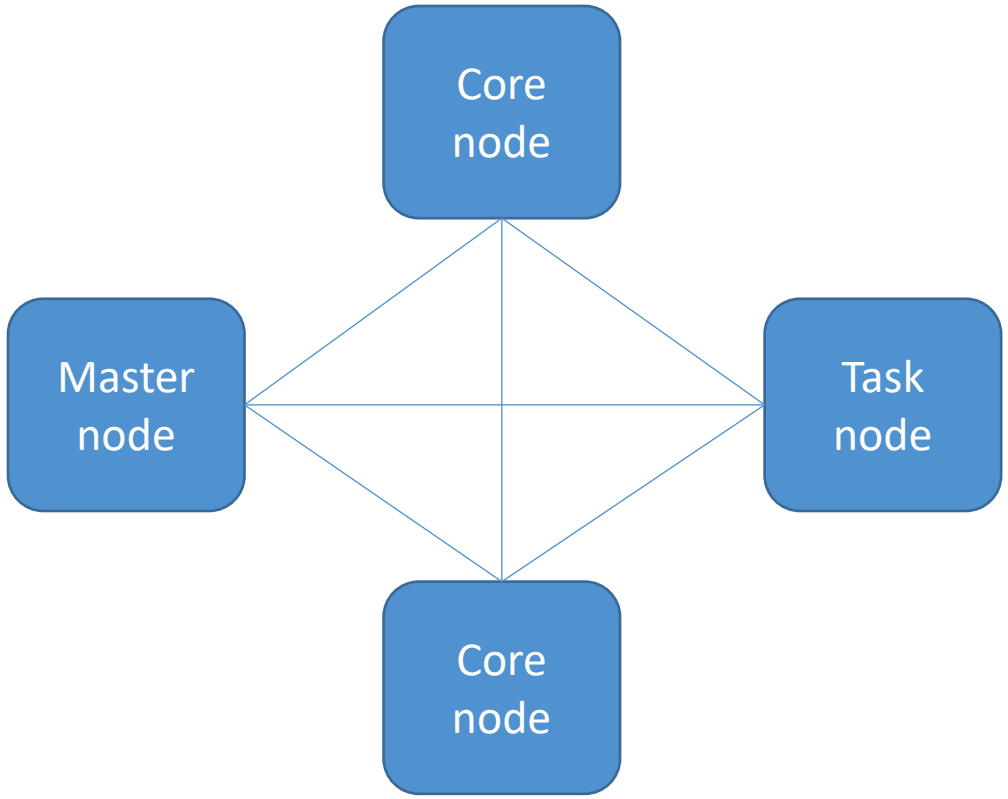
What is EMR?

- Elastic MapReduce
- Managed Hadoop framework on EC2 instances
- Includes Spark, HBase, Presto, Flink, Hive & more
- EMR Notebooks
- Several integration points with AWS

Amazon EMR

DataCumulus Sundog Education

An EMR Cluster



- **Master node**: manages the cluster
  - Single EC2 instance
- **Core node**: Hosts HDFS data and runs tasks
  - Can be scaled up & down, but with some risk
- **Task node**: Runs tasks, does not host data
  - No risk of data loss when removing
  - Good use of **spot instances**

EMR Usage

- Transient vs Long-Running Clusters
  - Can spin up task nodes using Spot instances for temporary capacity
  - Can use reserved instances on long-running clusters to save $
- Connect directly to master to run jobs
- Submit ordered steps via the console
- EMR Serverless lets AWS scale your nodes automatically

EMR / AWS Integration

- Amazon EC2 for the instances that comprise the nodes in the cluster
- Amazon VPC to configure the virtual network in which you launch your instances
- Amazon S3 to store input and output data
- Amazon CloudWatch to monitor cluster performance and configure alarms
- AWS IAM to configure permissions
- AWS CloudTrail to audit requests made to the service
- AWS Data Pipeline to schedule and start your clusters

EMR Storage

- HDFS
- EMRFS: access S3 as if it were HDFS
  - **EMRFS Consistent View** – Optional for S3 consistency
  - Uses DynamoDB to track consistency
- Local file system
- EBS for HDFS

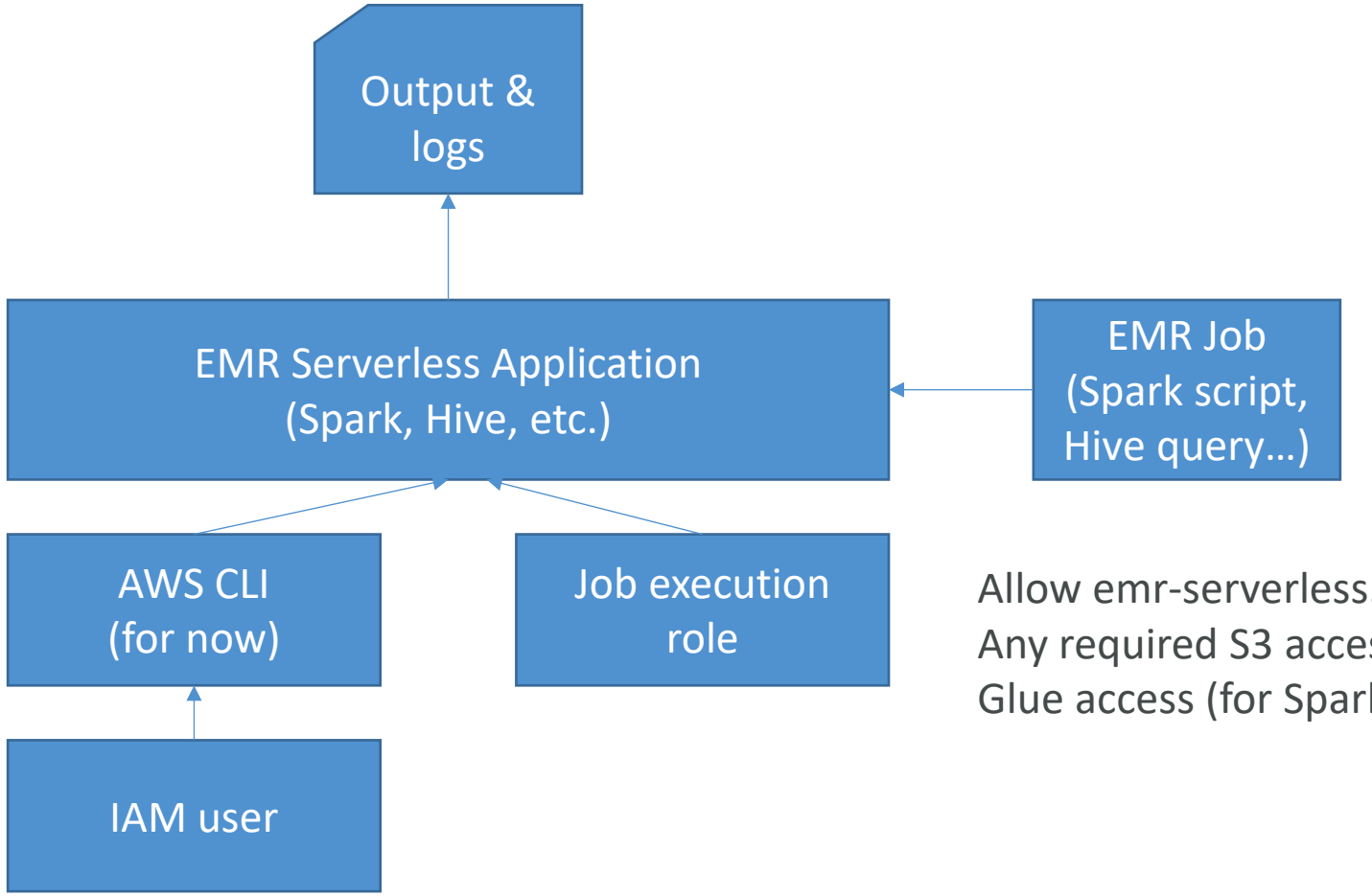
EMR promises

- EMR charges by the hour
  - Plus EC2 charges
- Provisions new nodes if a core node fails
- Can add and remove tasks nodes on the fly
- Can resize a running cluster's core nodes

EMR Serverless

- Choose an EMR Release and Runtime (Spark, Hive, Presto)
- Submit queries / scripts via job run requests
- EMR manages underlying capacity
  - But you can specify default worker sizes & pre-initialized capacity
  - EMR computes resources needed for your job & schedules workers accordingly
  - All within one region (across multiple AZ's)
- Why is this a big deal?
  - You no longer have to estimate how many workers are needed for your workloads – they are provisioned as needed, automatically.
- Serverless? Really?
  - TBH you still need to think about worker nodes and how they are configured.

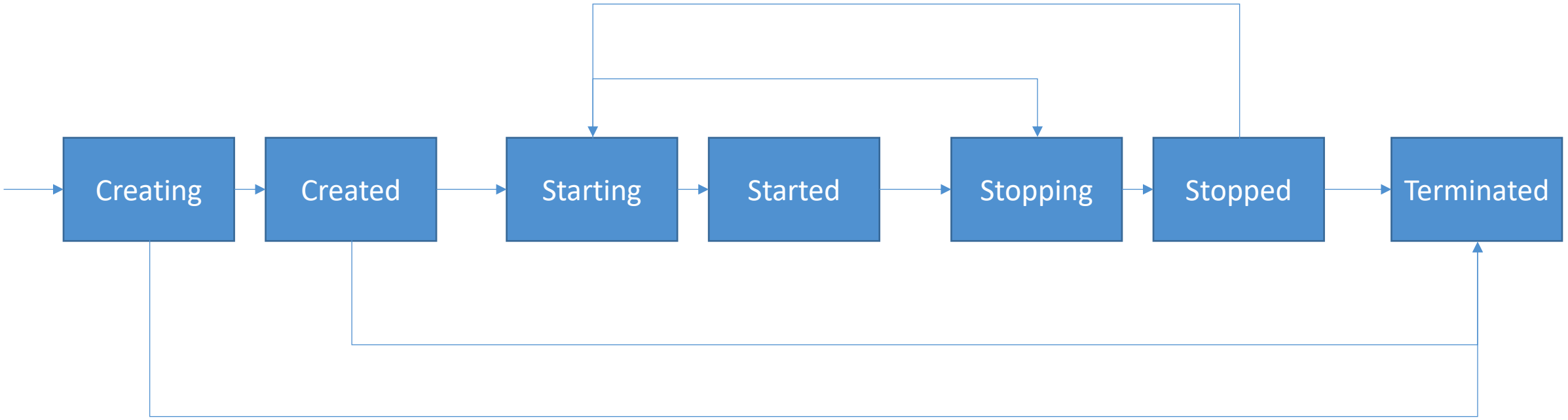
Using EMR Serverless




```
aws emr-serverless start-job-run \
   --application-id <application_id> \
   --execution-role-arn <execution_role_arn> \
   --job-driver '{
      "sparkSubmit": {
        "entryPoint": "s3://us-east-1.elasticmapreduce/emr-
containers/samples/wordcount/scripts/wordcount.py",
        "entryPointArguments": ["s3://DOC-EXAMPLE-
BUCKET/output"],
        "sparkSubmitParameters": "--conf spark.executor.cores=1 --
conf spark.executor.memory=4g --conf spark.driver.cores=1 --conf
spark.driver.memory=4g --conf spark.executor.instances=1"
      }
   }' \
   --configuration-overrides '{
      "monitoringConfiguration": {
        "s3MonitoringConfiguration": {
          "logUri": "s3://DOC-EXAMPLE-BUCKET/logs"
        }
      }
   }'
```

Allow emr-serverless.amazonaws.com service
Any required S3 access for scripts & data
Glue access (for SparkSQL), KMS keys

EMR Serverless Application Lifecycle



This isn't all automatic!
Must call API's such as CreateApplication, StartApplication, StopApplication, and importantly DeleteApplication to avoid excess charges.

Pre-Initialized Capacity

- Spark adds 10% overhead to memory requested for drivers & executors
- Be sure initial capacity is at least 10% more than requested by the job

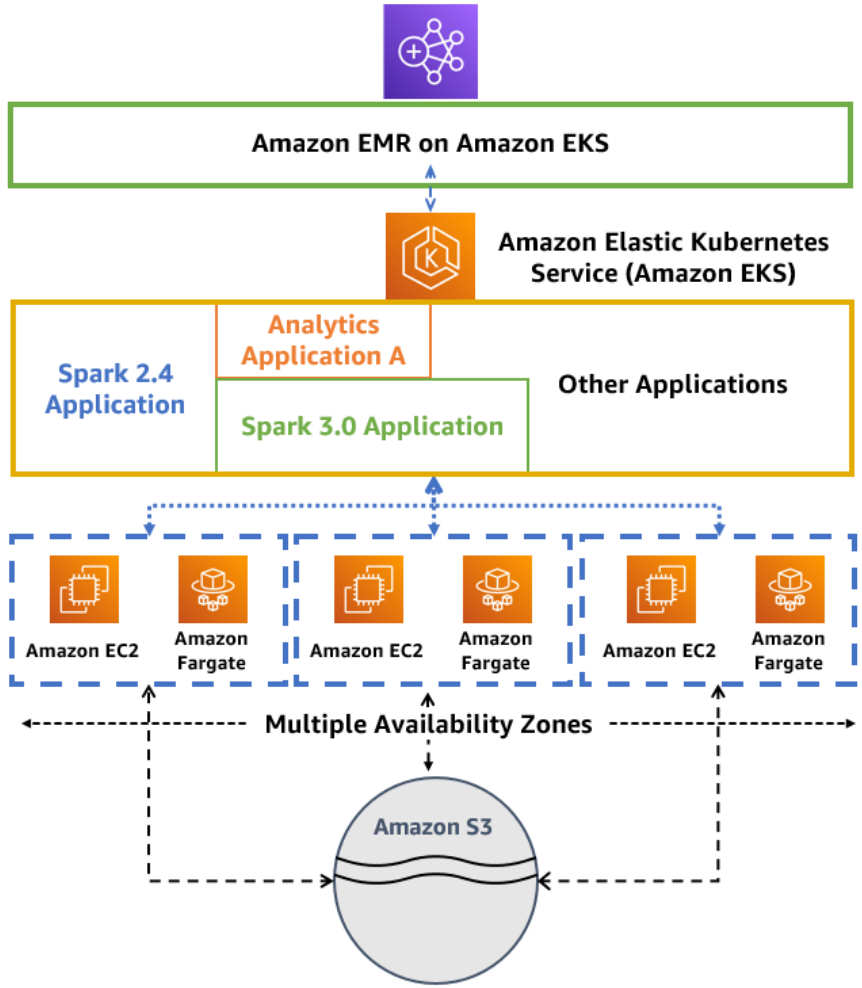
```
aws emr-serverless create-application \
 --type "SPARK" \
 --name <"my_application_name"> \
 --release-label "emr-6.5.0-preview" \
 --initial-capacity '{
  "DRIVER": {
     "workerCount": 5,
     "resourceConfiguration": {
        "cpu": "2vCPU",
        "memory": "4GB"
     }
  },
  "EXECUTOR": {
     "workerCount": 50,
     "resourceConfiguration": {
        "cpu": "4vCPU",
        "memory": "8GB"
     }
  }
 }' \
 --maximum-capacity '{
  "cpu": "400vCPU",
  "memory": "1024GB"
 }'
```

EMR Serverless Security

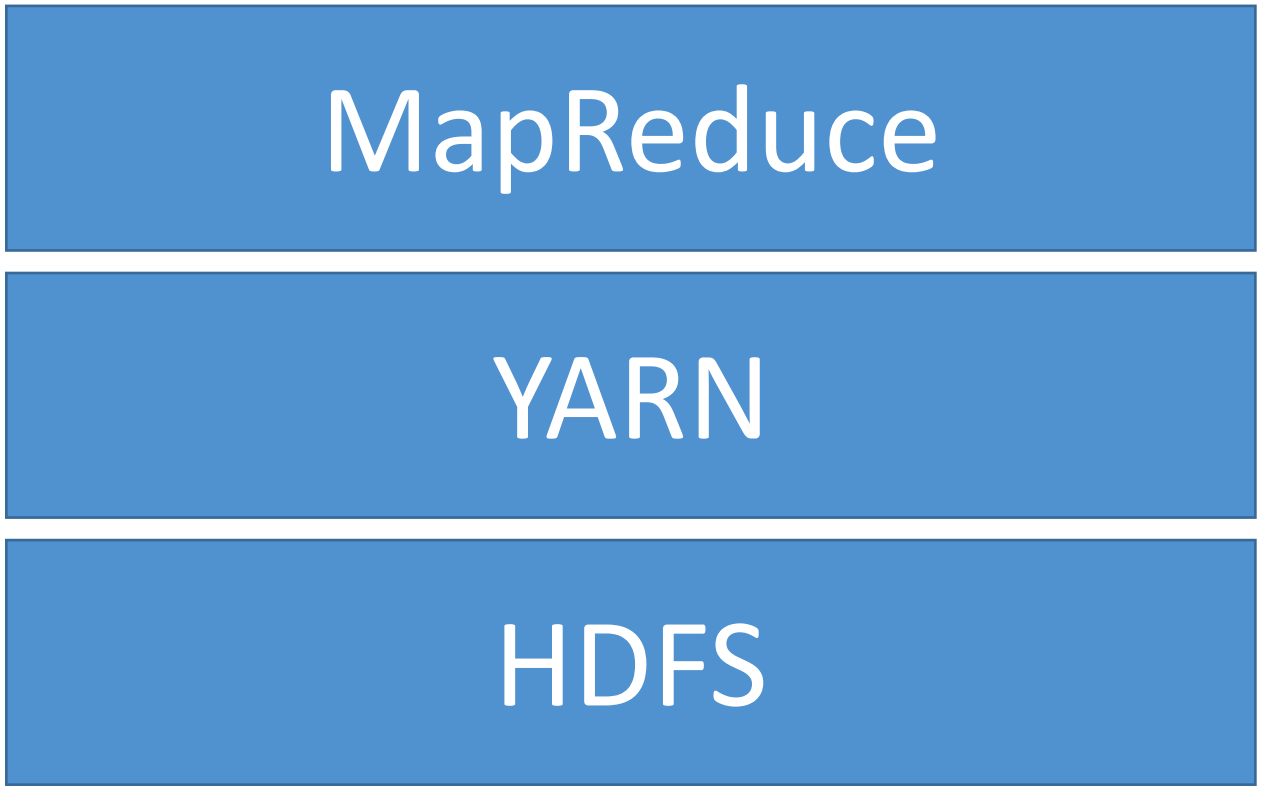
- Basically the same as EMR
- EMRFS
  - S3 encryption (SSE or CSE) at rest
  - TLS in transit between EMR nodes and S3
- S3
  - SSE-S3, SSE-KMS
- Local disk encryption
- Spark communication between drivers & executors is encrypted
- Hive communication between Glue Metastore and EMR uses TLS
- Force HTTPS (TLS) on S3 policies with aws:SecureTransport

EMR on EKS

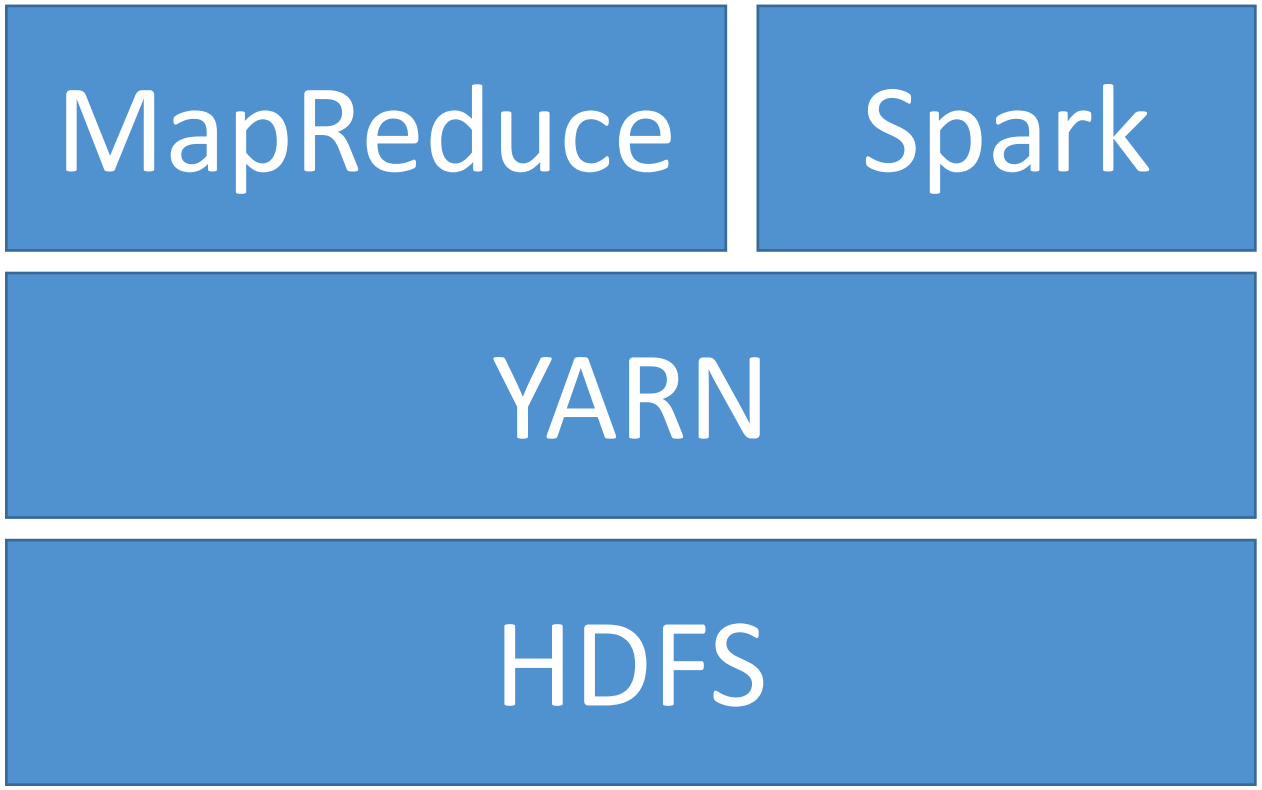
- Allows submitting Spark job on Elastic Kubernetes Service without provisioning clusters
- Fully managed
- Share resources between Spark and other apps on Kubernetes



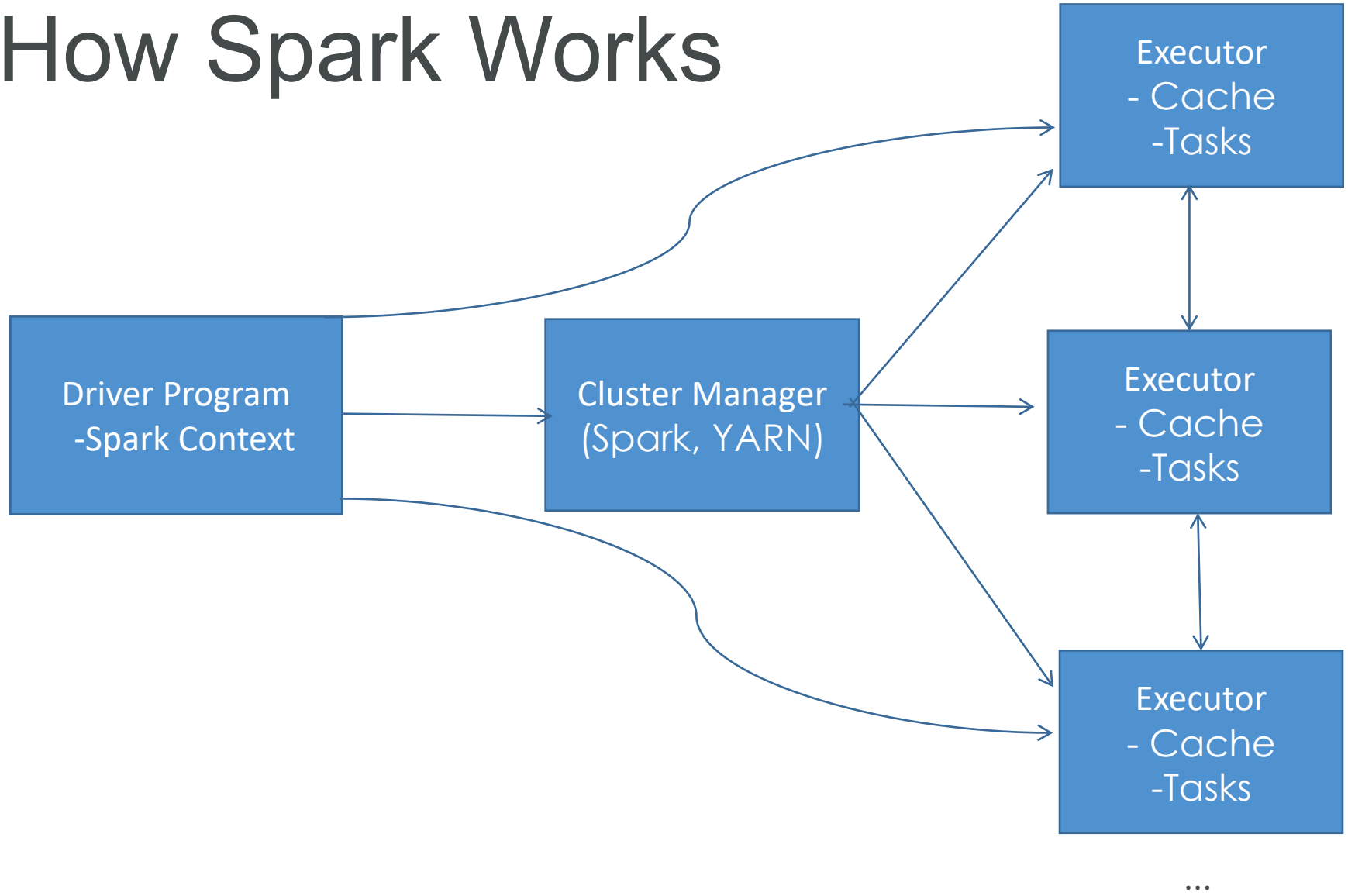
So… what's Hadoop?



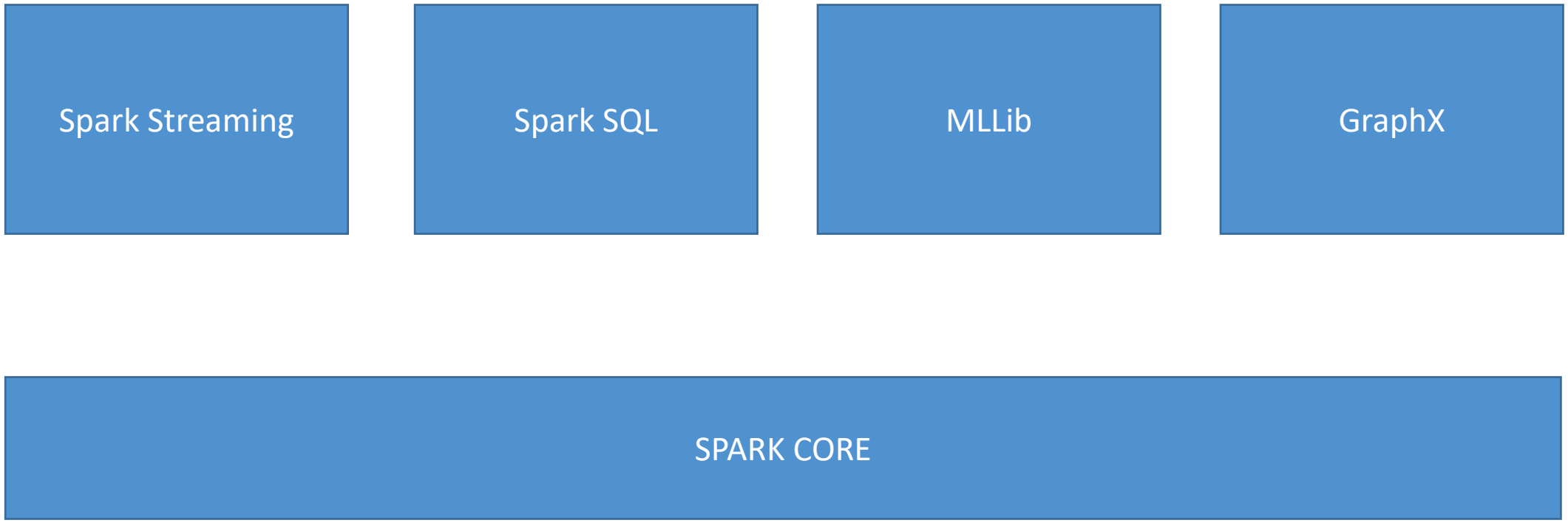
Apache Spark



How Spark Works



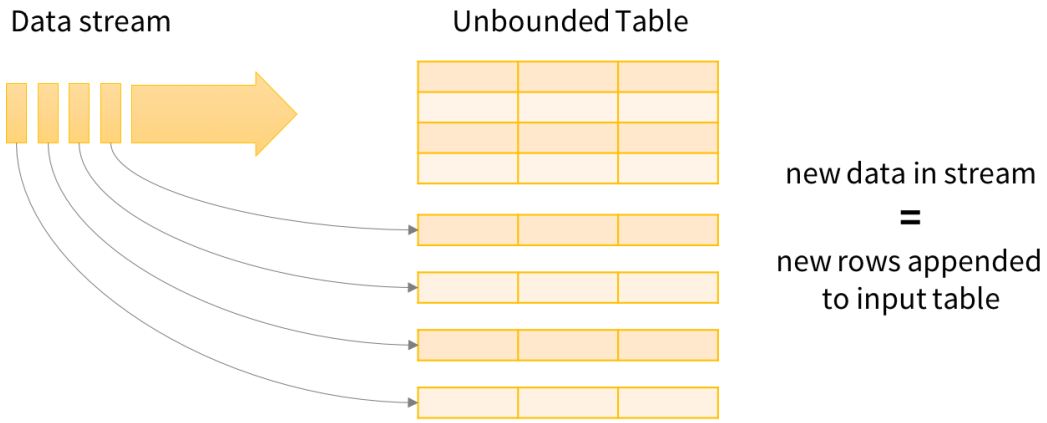
Spark Components



Spark MLLib

- Classification: logistic regression, naïve Bayes
- Regression
- Decision trees
- Recommendation engine (ALS)
- Clustering (K-Means)
- LDA (topic modeling)
- ML workflow utilities (pipelines, feature transformation, persistence)
- SVD, PCA, statistics

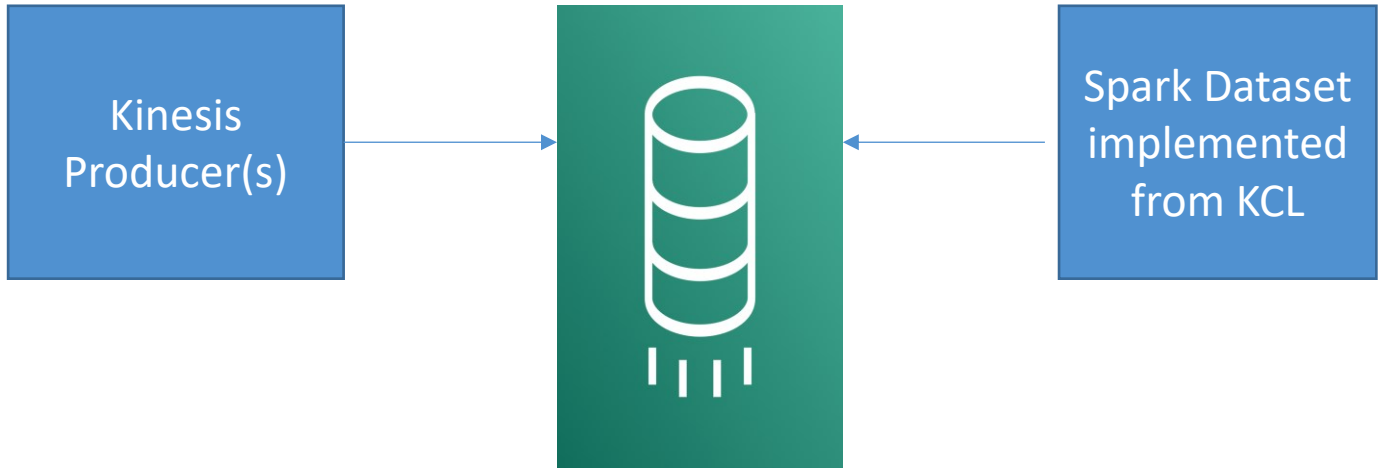
Spark Structured Streaming



Data stream as an unbounded Input Table

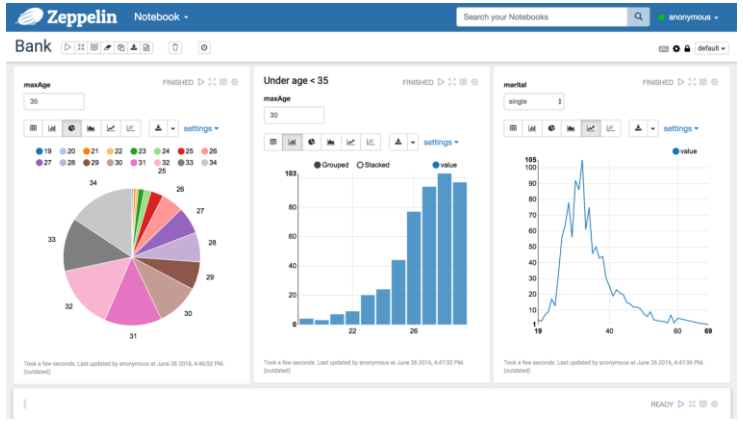
```
val inputDF = spark.readStream.json("s3://logs")
inputDF.groupBy($"action", window($"time", "1 hour")).count()
       .writeStream.format("jdbc").start("jdbc:mysql//...")
```

Spark Streaming + Kinesis



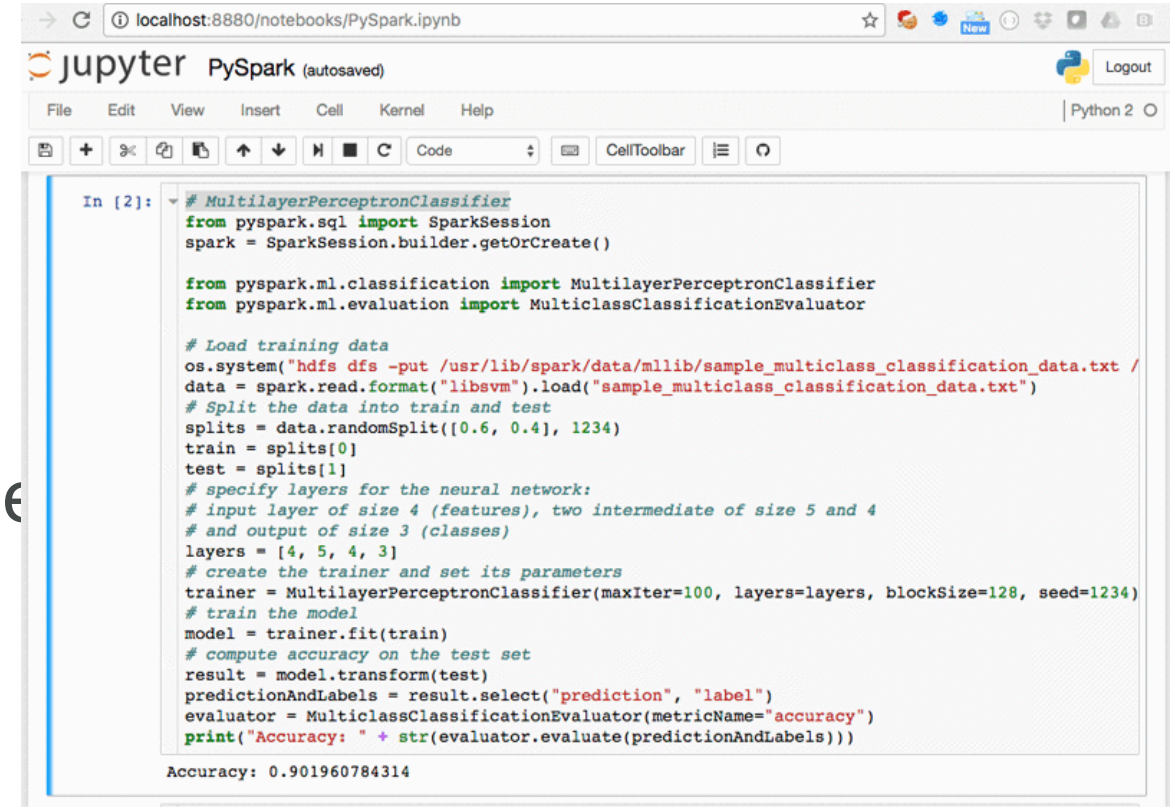
Zeppelin + Spark

- Can run Spark code interactively (like you can in the Spark shell)
  - This speeds up your development cycle
  - And allows easy experimentation and exploration of your big data
- Can execute SQL queries directly against SparkSQL
- Query results may be visualized in charts and graphs
- Makes Spark feel more like a data science tool!



EMR Notebook

- Similar concept to Zeppelin, with more AWS integration
- Notebooks backed up to S3
- Provision clusters from the notebook!
- Hosted inside a VPC
- Accessed only via AWS console



EMR Security

- IAM policies
- Kerberos
- SSH
- IAM roles
- Security configurations may be specified for Lake Formation
- Native integration with Apache Ranger
  - For data security on Hadoop / Hive

EMR: Choosing Instance Types

- Master node:
  - m4.large if < 50 nodes, m4.xlarge if > 50 nodes
- Core & task nodes:
  - m4.large is usually good
  - If cluster waits a lot on external dependencies (i.e. a web crawler), t2.medium
  - Improved performance: m4.xlarge
  - Computation-intensive applications: high CPU instances
  - Database, memory-caching applications: high memory instances
  - Network / CPU-intensive (NLP, ML) – cluster computer instances
  - Accelerated Computing / AI – GPU instances (g3, g4, p2, p3)
- Spot instances
  - Good choice for task nodes
  - Only use on core & master if you're testing or very cost-sensitive; you're risking partial data loss