Amazon Web Services

Kinesis

Contents

- ✓ Streaming Data Analytics - Basics
- ✓ Amazon Kinesis Services
- ✓ Kinesis Data Streams – Concepts & Terminology
- ✓ Working with Kinesis Data Streams - Demos
- ✓ APIs for Kinesis Scaling – Demos
- ✓ Kinesis Agent
- ✓ Kinesis Producer and Client Libraries (KPL & KCL)
- ✓ KPL & KCL – Architecture & Demos
- ✓ Kinesis Data Firehose – Concepts
- ✓ Kinesis Data Firehouse Demo

Streaming Data Analytics

Data that is generated continuously by thousands of data sources, which typically send in data records simultaneously, and in small sizes is called **Streaming data**. .

- Streaming data include a wide variety of data such as:
  - log files generated by web sites and mobile apps
  - e-commerce purchases
  - in-game player activity
  - social network feeds
  - Financial trading floors etc.

Streaming Data Use Cases

Use Case 1	Sensors in transportation vehicles, industrial equipment, and farm machinery send data to a streaming application. The application monitors performance, detects any potential defects in advance, and places a spare part order automatically preventing equipment down time.
Use Case 2	A financial institution tracks changes in the stock market in real time, computes value-at-risk, and automatically rebalances portfolios based on stock price movements.
Use Case 3	A real-estate website tracks a subset of data from consumers' mobile devices and makes Real-time property recommendations of properties to visit based on their geo-location.

Amazon Kinesis

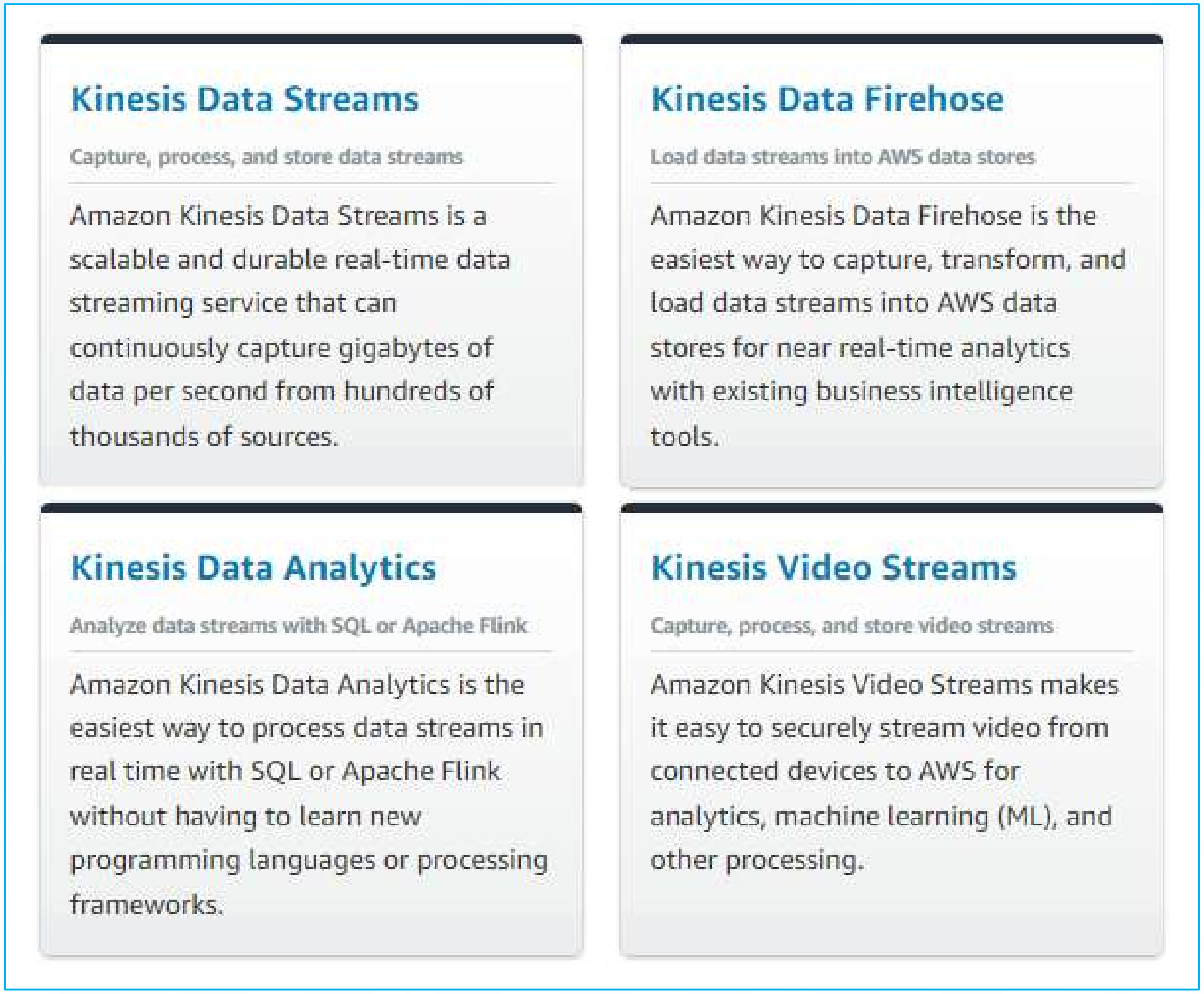
Amazon Kinesis makes it easy to collect, process, and analyze real-time, streaming data so you can get timely insights and react quickly to new information.

- Amazon Kinesis offers key capabilities to cost-effectively process streaming data at any scale, along with the flexibility to choose the tools that best suit the requirements of your application.

- With Amazon Kinesis, you can ingest real-time data such as video, audio, application logs, website clickstreams, and IoT telemetry data for machine learning, analytics, and other applications.

- Amazon Kinesis enables you to process and analyze data as it arrives and respond instantly instead of having to wait until all your data is collected before the processing can begin.

Amazon Kinesis Services



Kinesis Data Streams

Capture, process, and store data streams

Amazon Kinesis Data Streams is a scalable and durable real-time data streaming service that can continuously capture gigabytes of data per second from hundreds of thousands of sources.

Kinesis Data Firehose

Load data streams into AWS data stores

Amazon Kinesis Data Firehose is the easiest way to capture, transform, and load data streams into AWS data stores for near real-time analytics with existing business intelligence tools.

Kinesis Data Analytics

Analyze data streams with SQL or Apache Flink

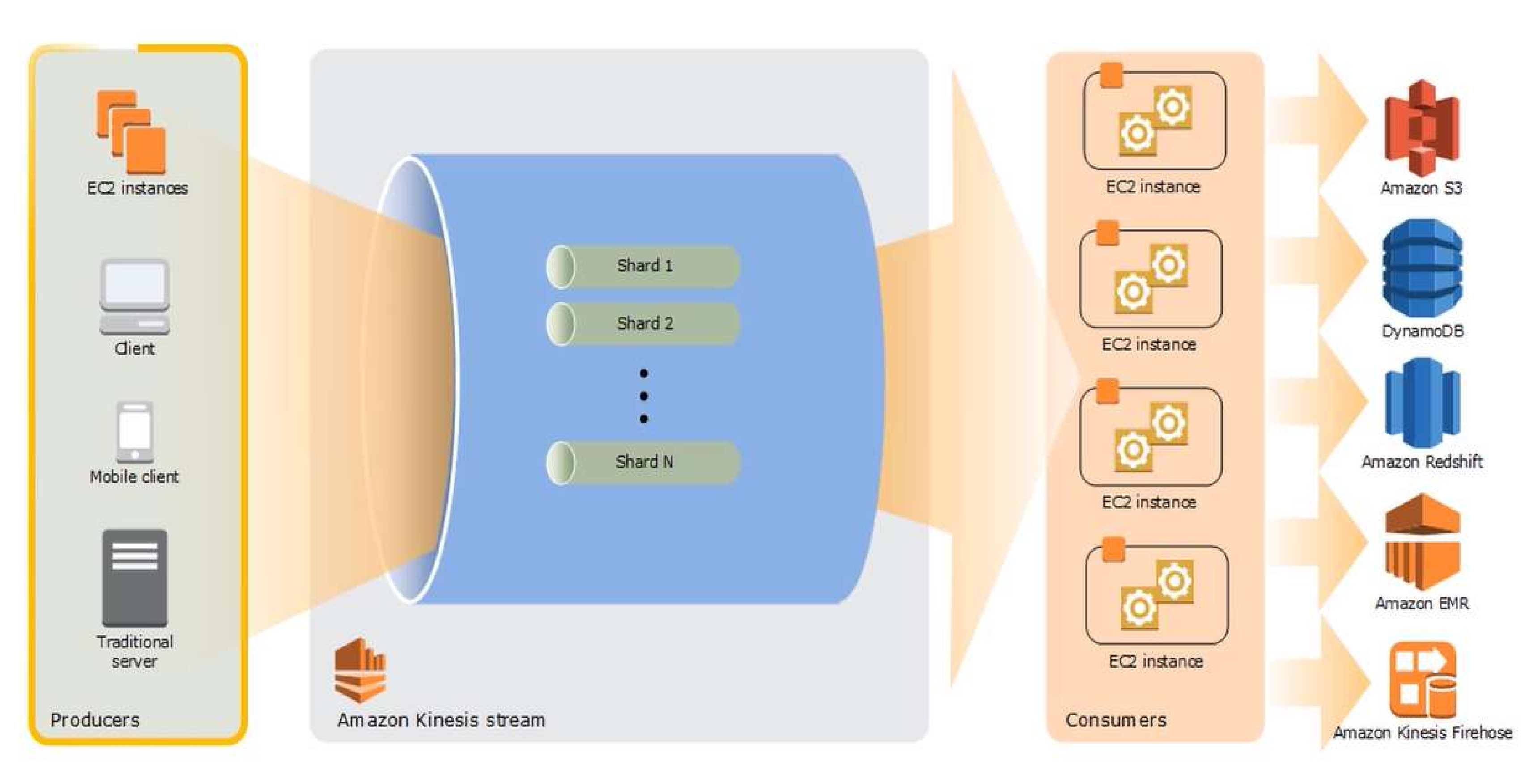
Amazon Kinesis Data Analytics is the easiest way to process data streams in real time with SQL or Apache Flink without having to learn new programming languages or processing frameworks.

Kinesis Video Streams

Capture, process, and store video streams

Amazon Kinesis Video Streams makes it easy to securely stream video from connected devices to AWS for analytics, machine learning (ML), and other processing.

Kinesis Data Streams High-Level Architecture



Kinesis Data Streams – Producers & Consumers

- The producers continually push data to Kinesis Data Streams, and the consumers process the data in real time.

- Consumers (such as a custom application running on Amazon EC2 or an Amazon Kinesis Data Firehose delivery stream) can store their results using an AWS service such as Amazon DynamoDB, Amazon Redshift, or Amazon S3.

Kinesis Data Streams - Features

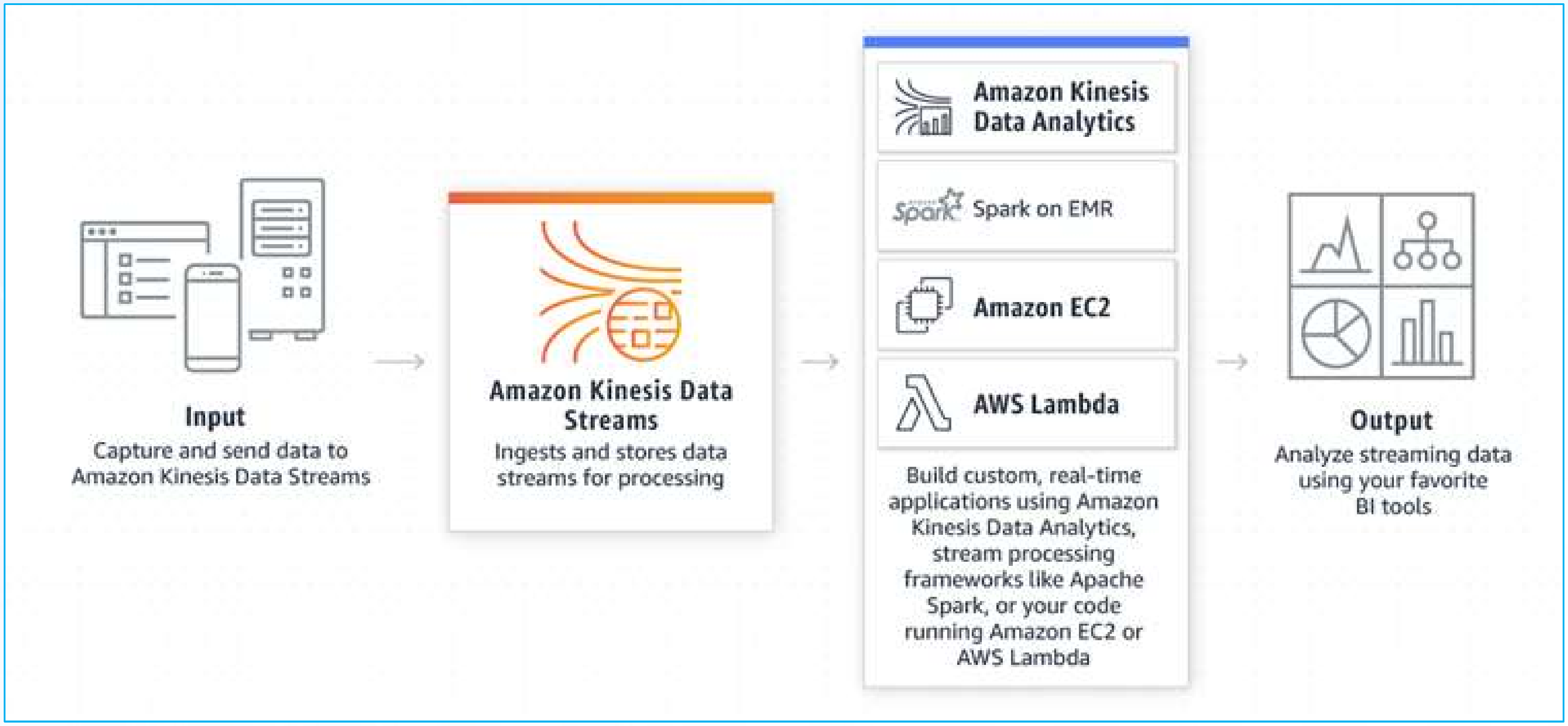
- Kinesis Data Streams store the data for 24 hours (retention period is configurable). Kinesis deletes the data after 24 hours.

- Consumers can not delete the data from Kinesis Streams (unlike SQS). The data is read-only for consumers.

- The delay between put and retrieve is less than 1 sec. (latency < 1 sec)

- Multiple consumers can consume data concurrently.

Kinesis Data Streams - Applications



Capture and send data to Amazon Kinesis Data Streams

Ingests and stores data streams for processing

Build custom, real-time applications using Amazon Kinesis Data Analytics, stream processing frameworks like Apache Spark, or your code running Amazon EC2 or AWS Lambda

Analyze streaming data using your favorite BI tools

When can I use Kinesis Data Streams?

- You can use Kinesis Data Streams for rapid and continuous data intake and aggregation.

- The type of data used can include IT infrastructure log data, application logs, social media, market data feeds, and web clickstream data.

- Because the response time for the data intake and processing is in real time, the processing is typically lightweight.

- The following are typical scenarios for using Kinesis Data Streams:
  - Accelerated log and data feed intake and processing
  - Real-time metrics and reporting
  - Real-time data analytics
  - Complex stream processing

Terminology - Data Record

- **Data Record**
  - Is a unit of data stored in Kinesis data stream
  - Data records are composed of a sequence-number, partition-key and a data blob, which is an immutable sequence of bytes.

Terminology - Shard

- **Shard**
  - Collection of similar data records that can be easily identified.
  - A Data Stream is made of many shards.

- 1 shard write: 1 MB/sec write (1000 messages/sec)
  read: 2 MB/sec read

- N shards write: N MB/sec write (N*1000 messages/sec)
  read: 2N MB/sec read

- 1 Shard has a capacity of:
  - 5 transactions/sec for reads
  - Max. total data read rate of 2 MB/sec
  - 1000 records/sec for writes
  - Max. total data write rate of 1 MB/sec

- Total capacity of the stream is the sum of the capacities of all shards.

Terminology - Sequence Number

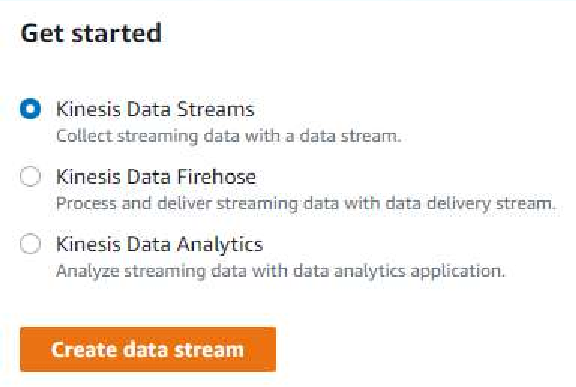
- **Sequence Number**
  - Each data record has a sequence number that is unique per partition-key within its shard.
  - Kinesis Data Streams assigns the sequence number after you write to the stream with **`client.putRecords`** or **`client.putRecord`**.
  - Sequence numbers for the same partition key generally increase over time.
  - The longer the time period between write requests, the larger the sequence numbers become.

Terminology - Partition Key

- **Partition Key**
  - A partition-key is used to group data by shard within a stream.
  - The Kinesis data streams service segregates the data records belonging to a stream into multiple shards, using the partition-key associated with each data record to determine which shard a given data record belongs to.

Demo - Create a Kinesis Data Stream

- Open the “Kinesis” service from the management console
- Select a region (ex: us-east-1) in which you want to create the stream.
- Select “**Kinesis Data Streams**” and click on “**Create Data Stream**”
- Enter a Data stream name (ex: demo-stream)
- Select a Data stream capacity (ex: Provisioned)



Get started

● Kinesis Data Streams
Collect streaming data with a data stream.

○ Kinesis Data Firehose
Process and deliver streaming data with data delivery stream.

○ Kinesis Data Analytics
Analyze streaming data with data analytics application.

Kinesis CLI Commands

https://docs.aws.amazon.com/cli/latest/reference/kinesis/index.html

NOTE:

- We will be using CLI to work with the streams.
- Make sure the user account you created the stream with has “KinesisFullAccess” policy attached.

Demo - Work with the Stream

- Describe a Stream

```
aws kinesis describe-stream
        --stream-name demo-stream
        --region us-east-1
```

- Write some records to the stream

```
aws kinesis put-record
        --stream-name demo-stream
        --partition-key 1
        --data “hello”
        --region us-east-1
        --cli-binary-format raw-in-base64-out
```

Demo - Work with the Stream

- Get the shard iterator
  - shard-id can be see from the describe command or when you put a record.
  - shard-iterator-type TRIM_HORIZON reads data from the beginning.
  - command returns shard-iterator. we need this to read from the stream.

```
aws kinesis get-shard-iterator
        --stream-name demo-stream
        --shard-iterator-type TRIM_HORIZON
        --shard-id shardId-000000000000
```

Demo - Work with the Stream

- Read records from the shard (using the shard-iterator)

```
 aws kinesis get-records
         --limit 2
         --shard-iterator [PUT_SHARD_ITERATOR_HERE]

aws kinesis get-records --limit 2 --shard-iterator
AAAAAAAAAAFFpxzQatUEMZAgp+mRN/C80VeNLIgRzMIl0uepyffChcFUAAUZW9/hWw3LxWGn4m
VL4w12Y/3D5ZopX0v3aQtC56zp3CH8NnT5E9tSlZ3YXlyWXO5xazcTAMlyj7X8v8x+ZAmZksrq
lnxm3wvwBn3aXcs2KOYvdcOs58wS3YCcQ8ftzkTQLAp/9cCzale2y5ad9lT/8paTAXoVmznOO+
7rMH+GTYWQSU10yhLI0pYGLQ==
```

NOTE: Shard Iterator expires after 300 seconds

The above command returns a JSON with 2 records data and also the NextShardIterator which you have you to further iterate the shard.

The output of the data is base64 encoded. You can decode the data using the following command.

$echo MQ== | base64 --decode (here MQ== is the base64 encoded data)

APIs of Kinesis Scaling

- SplitShard
  - Replaces a shard with two shards
- MergeShards
  - Replace two shards with a single shard
- UpdateShardCount
  - Scale up or down to a specific number of shards

Demo: split-shard

- Split the shard using **split-shard** command

```
aws kinesis split-shard
        --stream-name demo-stream
        --shard-to-split shardId-000000000000
        --new-starting-hash-key 170141183460469231731687303715884105727
```

- Here the **--new-starting-hash-key** value is a new hash-key to split the shard. This can be half of the **EndingHashKey** of your shard (you can get this from describe command) to split into two equal shards.

- Now write several messages to the stream with different partition-ids and observe that they are written to different shards based on the partition-id.

- Now you read both the shards by getting their corresponding shard-iterators. (use get-shard-iterator and get-records commands described earlier).

Demo: merge-shards

- Merge two adjacent shards as a new shard

```
aws kinesis merge-shards
        --stream-name demo-stream
        --shard-to-merge shardId-000000000001
        --adjacent-shard-to-merge shardId-000000000002
```

Kinesis Agent

https://docs.aws.amazon.com/streams/latest/dev/writing-with-agents.html

- Kinesis Agent is a stand-alone Java software application that offers an easy way to collect and send data to Kinesis Data Streams.
  - The agent continuously monitors a set of files and sends new data to your stream.
  - It also emits Amazon CloudWatch metrics to help you better monitor and troubleshoot the streaming process.
- You can install the agent on Linux-based server environments such as web servers, log servers, and database servers.
- After installing the agent, configure it by specifying the files to monitor and the stream for the data. After the agent is configured, it durably collects data from the files and reliably sends it to the stream.

Demo: Kinesis Agent

Step 1: Launch an EC2 instance (with Linux AMI)

- Attach a Role that has "AWSKinesisFullAccess“
- Connect to the EC2 via SSH.

Step 2: Install Kinesis Agent

- `sudo yum install -y aws-kinesis-agent`

Step 3: Create and run a shell script to continuously write some data to a log file (Kinesis agent will read from this file and send the data to kinesis stream)

- `sudo vi generatelogs.sh`
- Add the following script to the file.

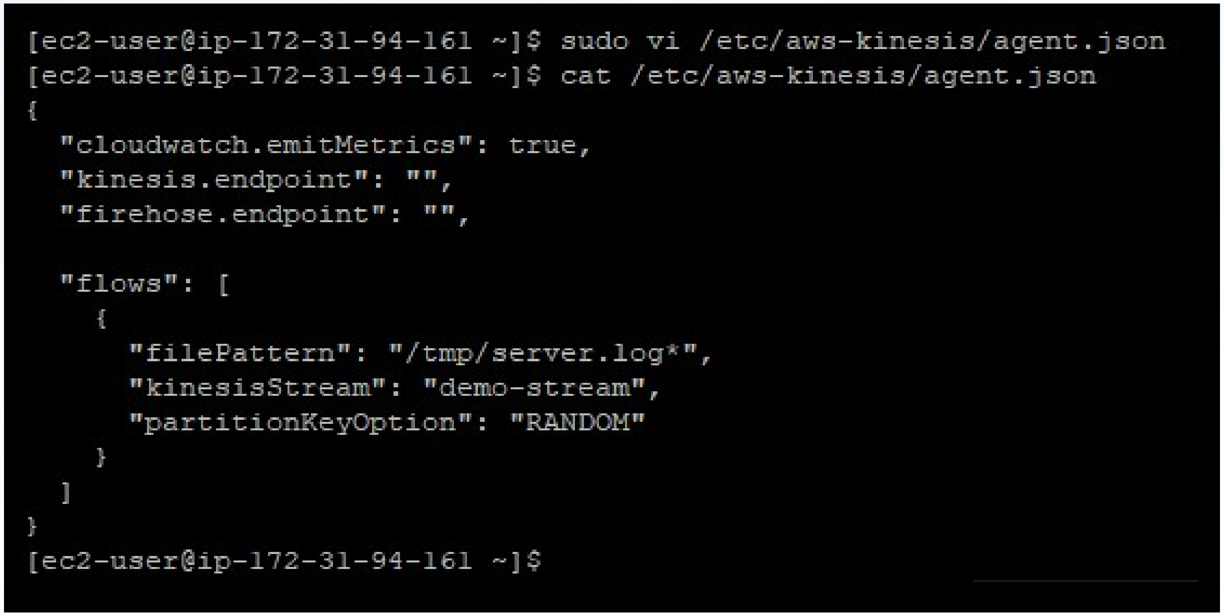
```
while(true) do
     sleep 10;
    echo `date +"%H:%M:%S"` "log message" >> /tmp/server.log
done;
```

- `sudo chmod +x generatelogs.sh`
- `./generatelogs.sh &` (note: & runs the script in background)

Demo: Kinesis Agent

Step 4: Configure Kinesis Agent

- `sudo vi /etc/aws-kinesis/agent.json`
- In this file, change the "filePattern" to "/tmp/server.log" and "kinesisStream" to your stream name (ex: “demo-stream")



```
[ec2-user@ip-172-31-94-161 ~]$ sudo vi /etc/aws-kinesis/agent.json
[ec2-user@ip-172-31-94-161 ~]$ cat /etc/aws-kinesis/agent.json
{
  "cloudwatch.emitMetrics": true,
  "kinesis.endpoint": "",
  "firehose.endpoint": "",

  "flows": [
    {
      "filePattern": "/tmp/server.log*",
      "kinesisStream": "demo-stream",
      "partitionKeyOption": "RANDOM"
    }
  ]
}
[ec2-user@ip-172-31-94-161 ~]$
```

Demo: Kinesis Agent

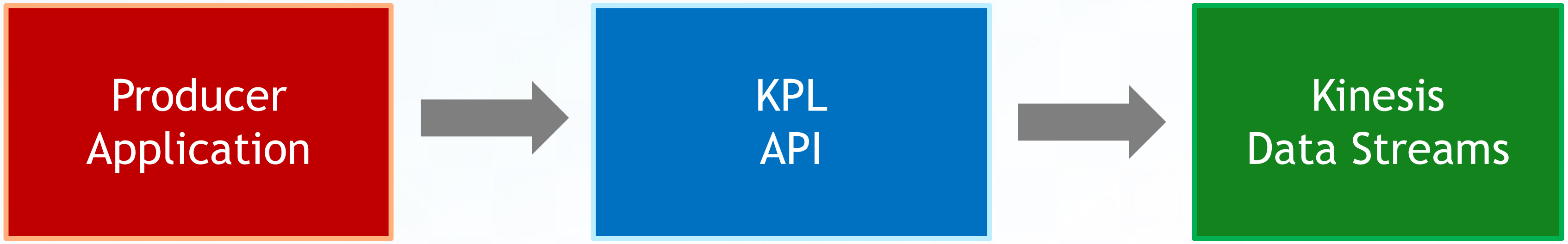
Step 5: Restart the Kinesis Agent

-
```
sudo service aws-kinesis-agent restart
```

Step 6: Check the logs (and observe that the contents of the file are sent to stream)

Kinesis Producer Library (KPL)

- An Amazon Kinesis Data Streams producer is an application that puts user data records into a Kinesis data stream.
- KPL simplifies producer application development, allowing developers to achieve high write throughput to a Kinesis data stream.
- It acts as an intermediary between your producer application code and the Kinesis Data Streams API actions.



Producer Application

KPL API

Kinesis Data Streams

Kinesis Producer Library (KPL) - Features

- Writes to one or more Kinesis data streams with an automatic and configurable retry mechanism
- Collects records and uses PutRecords to write multiple records to multiple shards per request
- Aggregates user records to increase payload size and improve throughput
- Integrates seamlessly with the Kinesis Client Library (KCL) to de-aggregate batched records on the consumer
- Submits Amazon CloudWatch metrics on your behalf to provide visibility into producer performance

NOTE: KPI is different than Kinesis Data Streams API

KPL Key Concepts: Record

- A **KPL user record** is a blob of data that has particular meaning to the user.
  - Examples: A JSON blob representing a UI event on a website, or a log entry from a web server.

- A **Kinesis Data Streams record** is an instance of the **Record** data structure defined by the Kinesis Data Streams service API.

  - It contains :
    - a partition key
    - sequence number
    - a blob of data.

KPL Key Concepts: Batching

- **Batching** refers to performing a single action on multiple items instead of repeatedly performing the action on each individual item.
  - With non-batching, you would place each record in a separate Kinesis Data Streams record and make one HTTP request to send it to Kinesis Data Streams.
  - With batching, each HTTP request can carry multiple records instead of just one.
- The KPL supports **two types** of batching:
  - **Aggregation** – Storing multiple records within a single Kinesis Data Streams record.
  - **Collection** – Using the API operation PutRecords to send multiple Kinesis Data Streams records to one or more shards in your Kinesis data stream.

KPL Key Concepts: Aggregation

- **Aggregation** refers to the storage of multiple records in a Kinesis Data Streams record.

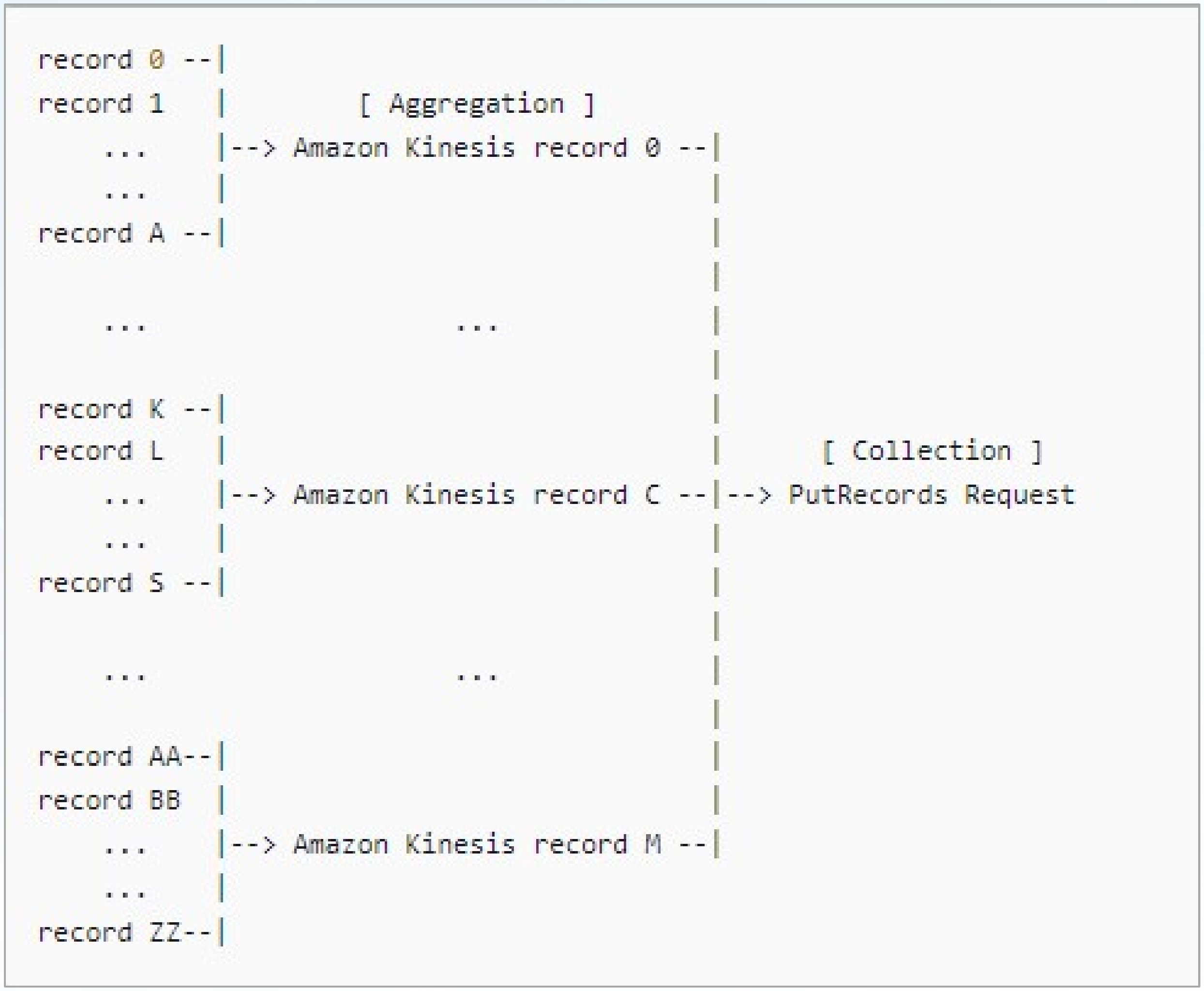
- Aggregation allows customers to increase the number of records sent per API call, which effectively increases producer throughput.

- Kinesis Data Streams shards support up to 1,000 Kinesis Data Streams records per second, or 1 MB throughput. Using aggregation, we can combine multiple records into a single Kinesis Data Streams record, thus improving the throughput.

KPL Key Concepts: Collection

- Collection refers to batching multiple Kinesis Data Streams records and sending them in a single HTTP request with a call to the API operation **PutRecords**, instead of sending each Kinesis Data Streams record in its own HTTP request.
- This increases throughput compared to using no collection, because it reduces the overhead of making many separate HTTP requests. In fact, PutRecords itself was specifically designed for this purpose.
- Collection differs from aggregation in that it is working with groups of Kinesis Data Streams records. The Kinesis Data Streams records being collected can still contain multiple records from the user.

KPL Key Concepts: Collection



```
record 0 --|
record 1   |        [ Aggregation ]
    ...    |--> Amazon Kinesis record 0 --|
    ...    |                              |
record A --|                              |
                                          |
    ...                   ...             |
                                          |
record K --|                              |
record L   |                              |      [ Collection ]
    ...    |--> Amazon Kinesis record C --|--> PutRecords Request
    ...    |                              |
record S --|                              |
                                          |
    ...                   ...             |
                                          |
record AA--|                              |
record BB  |                              |
    ...    |--> Amazon Kinesis record M --|
    ...    |
record ZZ--|
```

KPL Code Review: Simple Code (Async)

```
// KinesisProducer gets credentials automatically.
// It also gets region automatically from EC2 metadata service.

KinesisProducer kinesis = new KinesisProducer();

// Put some records
for (int i = 0; i < 100; ++i) {
    ByteBuffer data = ByteBuffer.wrap("myData".getBytes("UTF-8"));

    // doesn't block
    kinesis.addUserRecord("myStream", "myPartitionKey", data);
}

// Do other stuff ...
```

KPL Code: Barebones Producer Code

```
// KinesisProducer gets credentials automatically.
// It also gets region automatically from EC2 metadata service.

KinesisProducer kinesis = new KinesisProducer();

// Put some records
for (int i = 0; i < 100; ++i) {
    ByteBuffer data = ByteBuffer.wrap("myData".getBytes("UTF-8"));

    // doesn't block
    kinesis.addUserRecord("myStream", "myPartitionKey", data);
}

// Do other stuff ...
```

KPL Code: Responding to Results Synchronously

```
KinesisProducer kinesis = new KinesisProducer();

// Put some records and save the Futures
List<Future<UserRecordResult>> putFutures = new LinkedList<>();

for (int i = 0; i < 100; i++) {
    ByteBuffer data = ByteBuffer.wrap("myData".getBytes("UTF-8"));
    // doesn't block
    putFutures.add(kinesis.addUserRecord("myStream", "myPartitionKey", data));
}

// Wait for puts to finish and check the results
for (Future<UserRecordResult> f : putFutures) {
    UserRecordResult result = f.get(); // this does block
    if (result.isSuccessful()) {
        System.out.println("Put record into shard " + result.getShardId());
    } else {
        for (Attempt attempt : result.getAttempts()) {
            // Analyze and respond to the failure
        }
    }
}
```

KPL Code: Responding to Results Asynchronously

```
KinesisProducer kinesis = new KinesisProducer();

FutureCallback<UserRecordResult> myCallback = new FutureCallback<>() {
    @Override public void onFailure(Throwable t) {
        /* Analyze and respond to the failure  */
    };
    @Override public void onSuccess(UserRecordResult result) {
        /* Respond to the success */
    };
};

for (int i = 0; i < 100; ++i) {
    ByteBuffer data = ByteBuffer.wrap("myData".getBytes("UTF-8"));

    ListenableFuture<UserRecordResult> f
         = kinesis.addUserRecord("myStream", "myPartitionKey", data);

    // If the Future is complete by the time we call addCallback, the callback will be
invoked immediately.
    Futures.addCallback(f, myCallback);
}
```

Demo: Running KPL Sample Code

https://github.com/awslabs/amazon-kinesis-producer/blob/master/java/amazon-kinesis-producer-sample/src/com/amazonaws/services/kinesis/producer/sample/SampleProducer.java

Step 1: Launch an EC2 instance (with Linux AMI)

- Attach a Role that has "AWSKinesisFullAccess“
- Connect to the EC2 via SSH.

Step 2: Install Java

- `sudo yum install java-1.8.0-openjdk-devel`
- `sudo /usr/sbin/alternatives --config java`
- `sudo /usr/sbin/alternatives --config javac`
- `java –version`

Step 3: Install Maven

- `sudo wget http://repos.fedorapeople.org/repos/dchen/apache-maven/epel-apache-maven.repo -O /etc/yum.repos.d/epel-apache-maven.repo`
- `sudo sed -i s/\$releasever/6/g /etc/yum.repos.d/epel-apache-maven.repo`
- `sudo yum install -y apache-maven`

Demo: Running KPL Sample Code

Step 4: Install Git

- `sudo yum install git`

Step 5: Clone the sample from git to local folder called *kpl-local*

- `git clone https://github.com/awslabs/amazon-kinesis-producer.git kpl-local`

Step 6: Edit ***stream name*** and ***region*** in **SampleProducerConfig** file

- `cd /home/ec2-user/kpl-local/java/amazon-kinesis-producer-sample/src/com/amazonaws/services/kinesis/producer/sample`
- `vi SampleProducerConfig`

Step 7: Compile & Run the Sample

- `cd /home/ec2-user/kpl-local/java/amazon-kinesis-producer-sample`
- `mvn clean package`
- `mvn exec:java -Dexec.mainClass="com.amazonaws.services.kinesis.producer.sample.SampleProducer"`

KCL Concepts

- **KCL consumer application** – an application that is custom-built using KCL and designed to read and process records from data streams.

- **Consumer application instance** - KCL consumer applications are typically distributed, with one or more application instances running simultaneously in order to coordinate on failures and dynamically load balance data record processing.

- **Worker** – a high level class that a KCL consumer application instance uses to start processing data.

- **Lease** – data that defines the binding between a worker and a shard. Distributed KCL consumer applications use leases to partition data record processing across a fleet of workers. At any given time, each shard of data records is bound to a particular worker by a lease identified by the **leaseKey** variable.

KCL Concepts

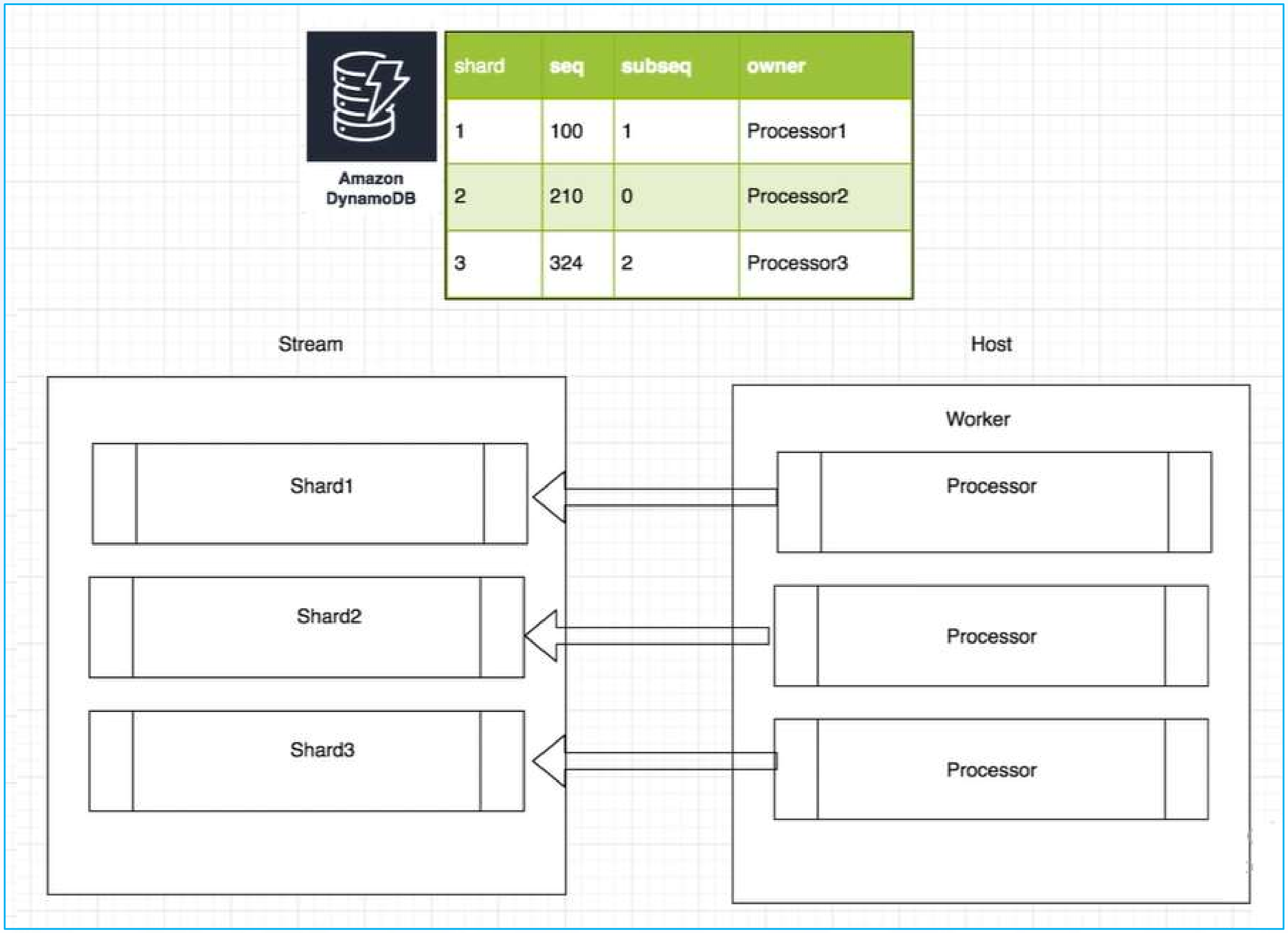
- **Lease table** - a unique Amazon DynamoDB table that is used to keep track of the shards in a KDS data stream that are being leased and processed by the workers of the KCL consumer application.
  - The lease table must remain in sync (within a worker and across all workers) with the latest shard information from the data stream while the KCL consumer application is running.
- **Record processor** – the logic that defines how your KCL consumer application processes the data that it gets from the data streams.
  - At runtime, a KCL consumer application instance instantiates a worker, and this worker instantiates one record processor for every shard to which it holds a lease.

How KCL works ?

The KCL acts as an intermediary between your record processing logic and Kinesis Data Streams. The KCL performs the following tasks:

- Connects to the data stream
- Enumerates the shards within the data stream
- Uses leases to coordinate shard associations with its workers
- Instantiates a record processor for every shard it manages
- Pulls data records from the data stream
- Pushes the records to the corresponding record processor
- Checkpoints processed records
- Balances shard-worker associations (leases) when the worker instance count changes or when the data stream is resharded (shards are split or merged)

KCL Architecture



shard	seq	subseq	owner
1	100	1	Processor1
2	210	0	Processor2
3	324	2	Processor3

KCL Architecture

- KCL processes multiple shards of the Kinesis data stream using many instances in your cluster

- KCL spins up as many processors as there are shards to process. These processor may be spun up on a single node or many nodes.

- A unique Amazon DynamoDB table, called **lease table**, is used to keep track of the shards in a KDS data stream that are being leased and processed by the workers of the KCL consumer application.

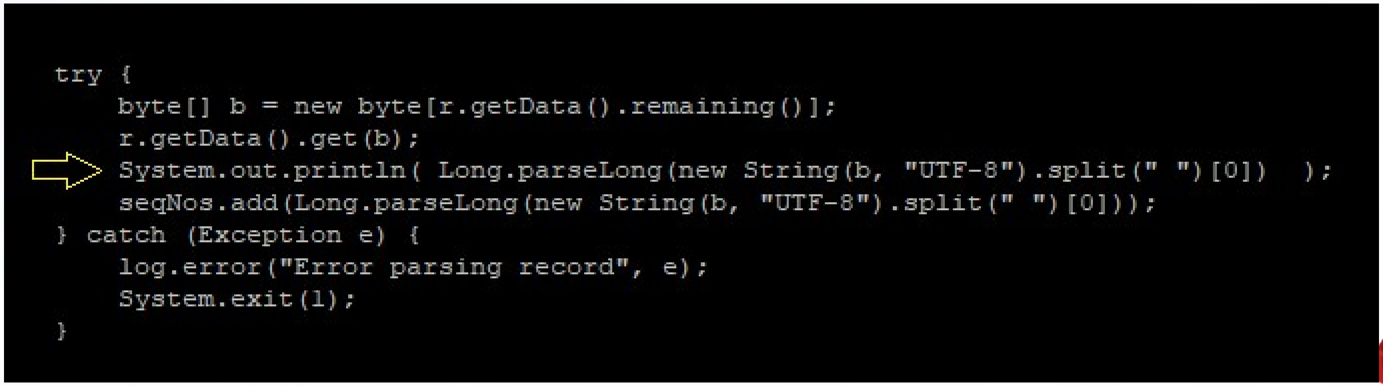
Demo: Running KCL Sample Code

- **We will be running this code on the same EC2 instance that we used for running KPL code.**

Step 1: Make sure the Role attached to the EC2 instance has the following policies attached.

- CloudWatchFullAccess
- AmazonDynamoDBFullAccess
- AmazonKinesisFullAccess

Step 2: Add the following line of code in the ***processRecords*** method of ***SampleConusmer.java*** file to print the data of the record on the console.



```
try {
    byte[] b = new byte[r.getData().remaining()];
    r.getData().get(b);
    System.out.println( Long.parseLong(new String(b, "UTF-8").split(" ")[0])  );
    seqNos.add(Long.parseLong(new String(b, "UTF-8").split(" ")[0]));
} catch (Exception e) {
    log.error("Error parsing record", e);
    System.exit(1);
}
```

Demo: Running KCL Sample Code

Step 3: Recompile the and run the code.

- `cd /home/ec2-user/kpl-local/java/amazon-kinesis-producer-sample`
- `mvn clean package`
- `mvn exec:java -Dexec.mainClass="com.amazonaws.services.kinesis.producer.sample.SampleConsumer"`

Kinesis Data Firehose

- Amazon Kinesis Data Firehose is a fully managed service for delivering real-time streaming data to destinations such as Amazon S3, Amazon Redshift, Amazon OpenSearch Service, Splunk, and any custom HTTP endpoint or HTTP endpoints owned by supported third-party service providers.

- With Kinesis Data Firehose, you don't need to write applications or manage resources. You configure your data producers to send data to Kinesis Data Firehose, and it automatically delivers the data to the destination that you specified.

- You can also configure Kinesis Data Firehose to transform your data before delivering it.

Kinesis Data Firehose: Key Concepts

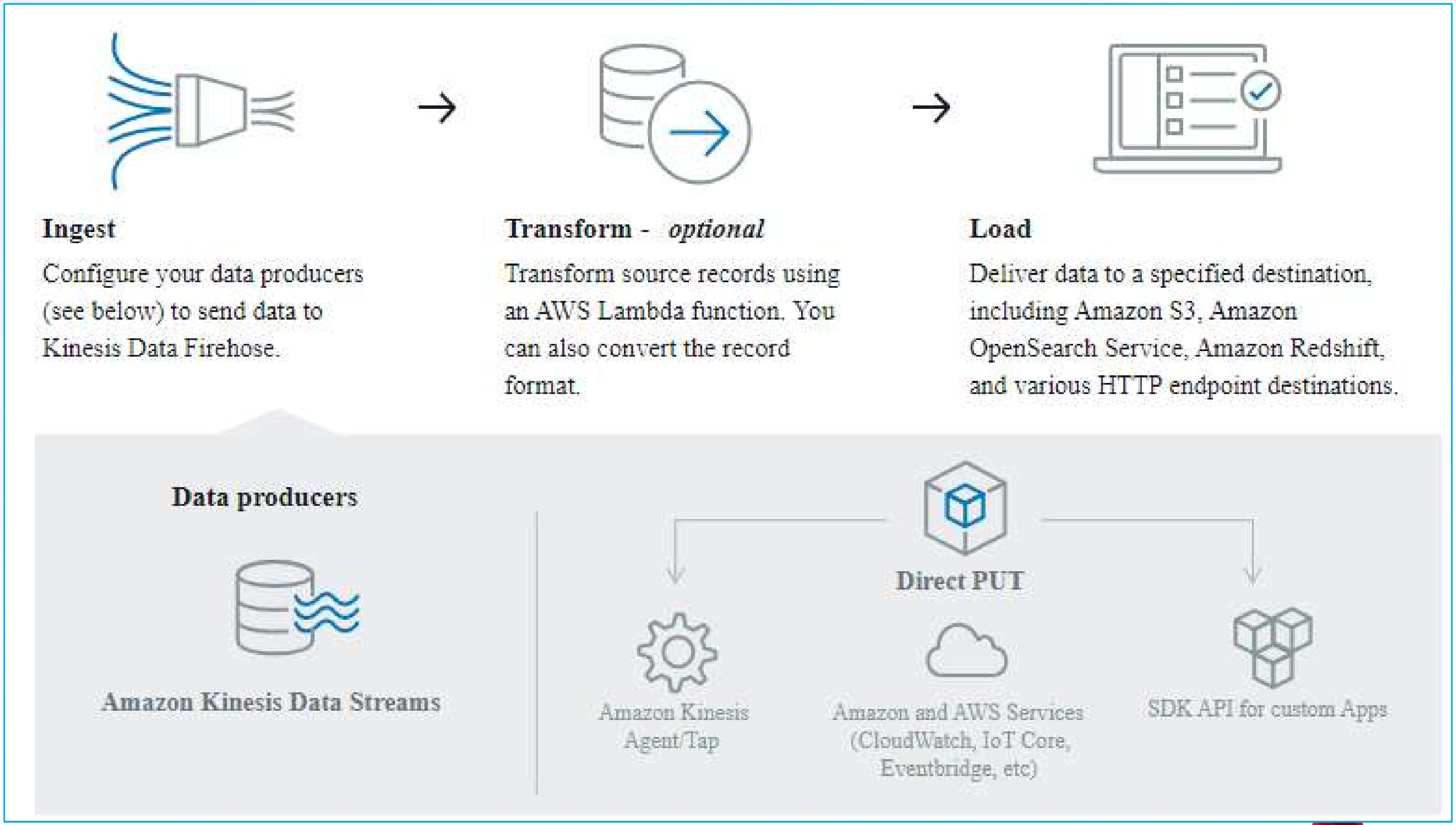
- **Kinesis Data Firehose delivery stream -** The underlying entity of Kinesis Data Firehose. You use Kinesis Data Firehose by creating a Kinesis Data Firehose delivery stream and then sending data to it.

- **Record -** The data of interest that your data producer sends to a Kinesis Data Firehose delivery stream. A record can be as large as 1,000 KB.

- **Data Producer** - Producers send records to Kinesis Data Firehose delivery streams.
  - For example, a web server that sends log data to a delivery stream is a data producer.
  - You can also configure your Kinesis Data Firehose delivery stream to automatically read data from an existing Kinesis data stream, and load it into destinations.

- **Buffer size and Buffer interval -** Kinesis Data Firehose buffers incoming streaming data to a certain size or for a certain period of time before delivering it to destinations.
  - Buffer Size is in MBs and Buffer Interval is in seconds.

Kinesis Data Firehose - How it Works?



Ingest

Configure your data producers (see below) to send data to Kinesis Data Firehose.

Transform - ***optional***

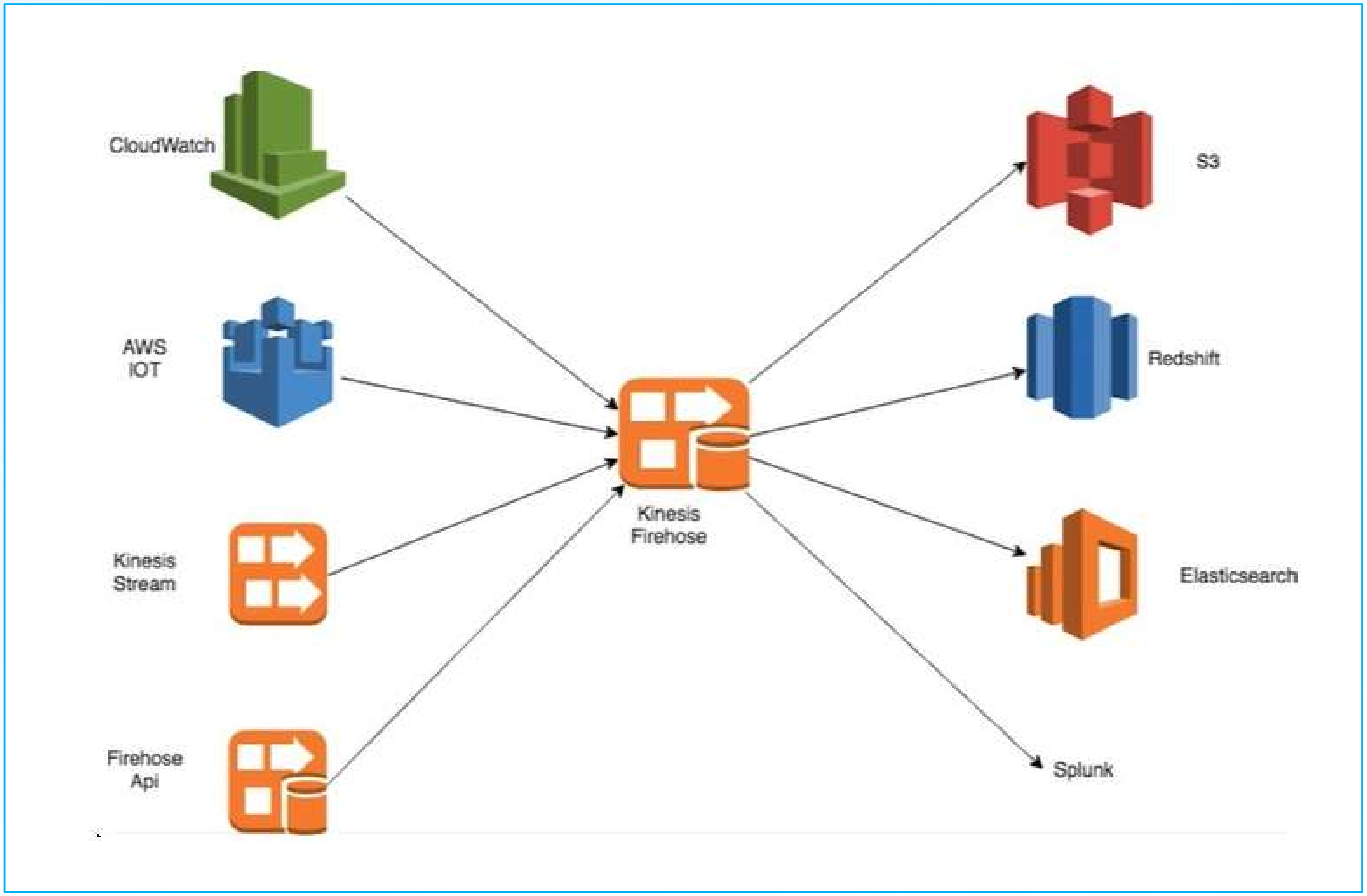
Transform source records using an AWS Lambda function. You can also convert the record format.

Load

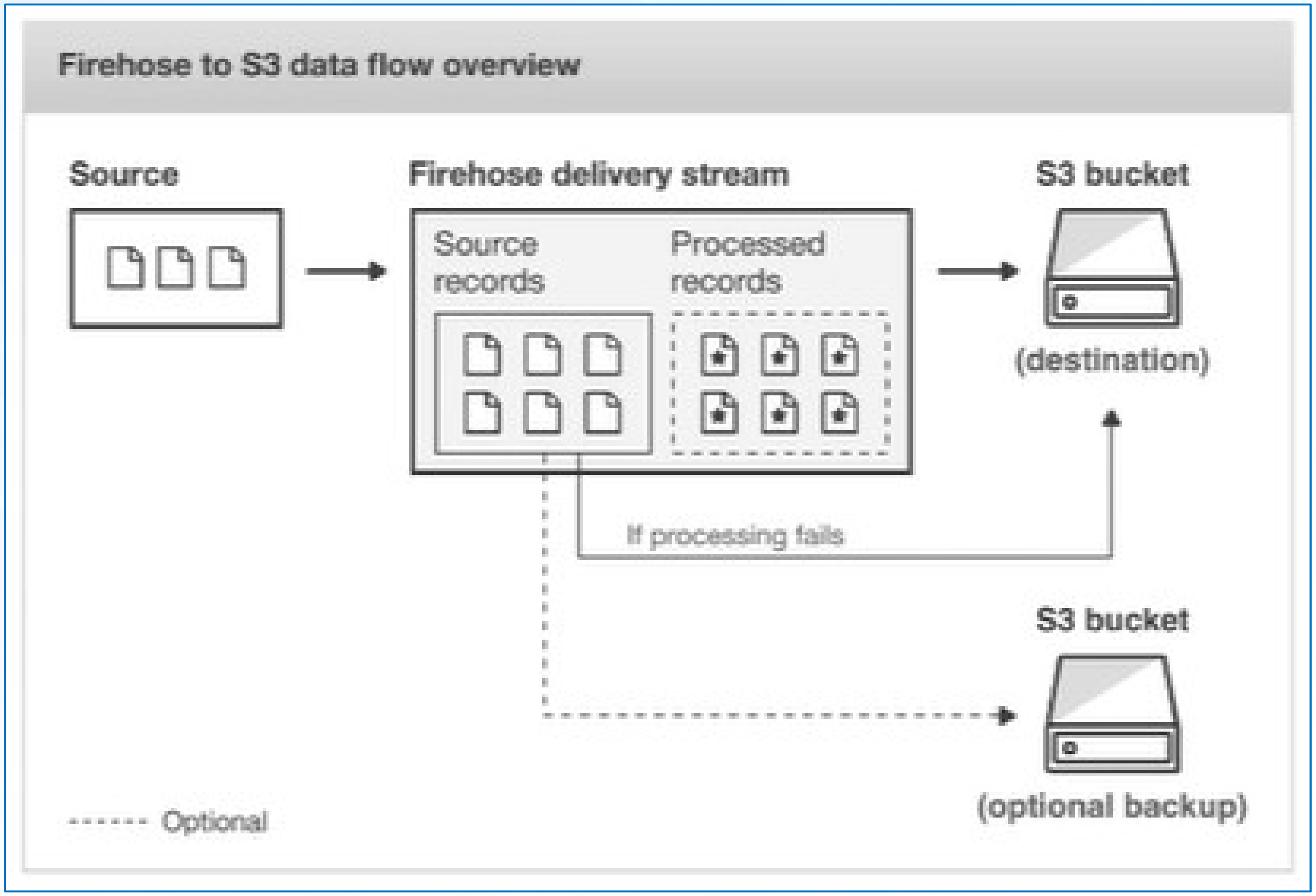
Deliver data to a specified destination, including Amazon S3, Amazon OpenSearch Service, Amazon Redshift, and various HTTP endpoint destinations.

Data producers
Amazon Kinesis Data Streams
Direct PUT
Amazon Kinesis Agent/Tap
Amazon and AWS Services (CloudWatch, IoT Core, Eventbridge, etc)
SDK API for custom Apps

Kinesis Data Firehose - How it Works?



Demo – Writing records from CLI to S3



Demo – Writing records from CLI to S3

Step 1 : Create a “Kinesis Data Firehose” delivery stream from Kinesis service.

Step 2 : Set the following configuration options

- Source: Direct PUT
- Destination: Amazon S3
- Delivery Stream Name: demo-firehose
- Transform and convert records
  - Data transformation: Enabled
  - AWS Lambda function
    - Create Function
    - Template: General Kinesis Data Firehose Processing
    - Function name : *demo-firehose*
    - Execution role: Create a new role with basic Lambda permissions
    - Code*: As given in the script file (see the slide)*
      - Set the timeout for the function to above 1 min
        - **Configurations > General configuration > Edit**
  - Destination settings:
      - S3 Bucket: Browse to an S3 bucket where you want to write the data

Demo - Writing records from CLI to S3

Step3 : Add several records to the delivery stream from AWS CLI terminal.

- Use the following command from the AWS CLI to add records

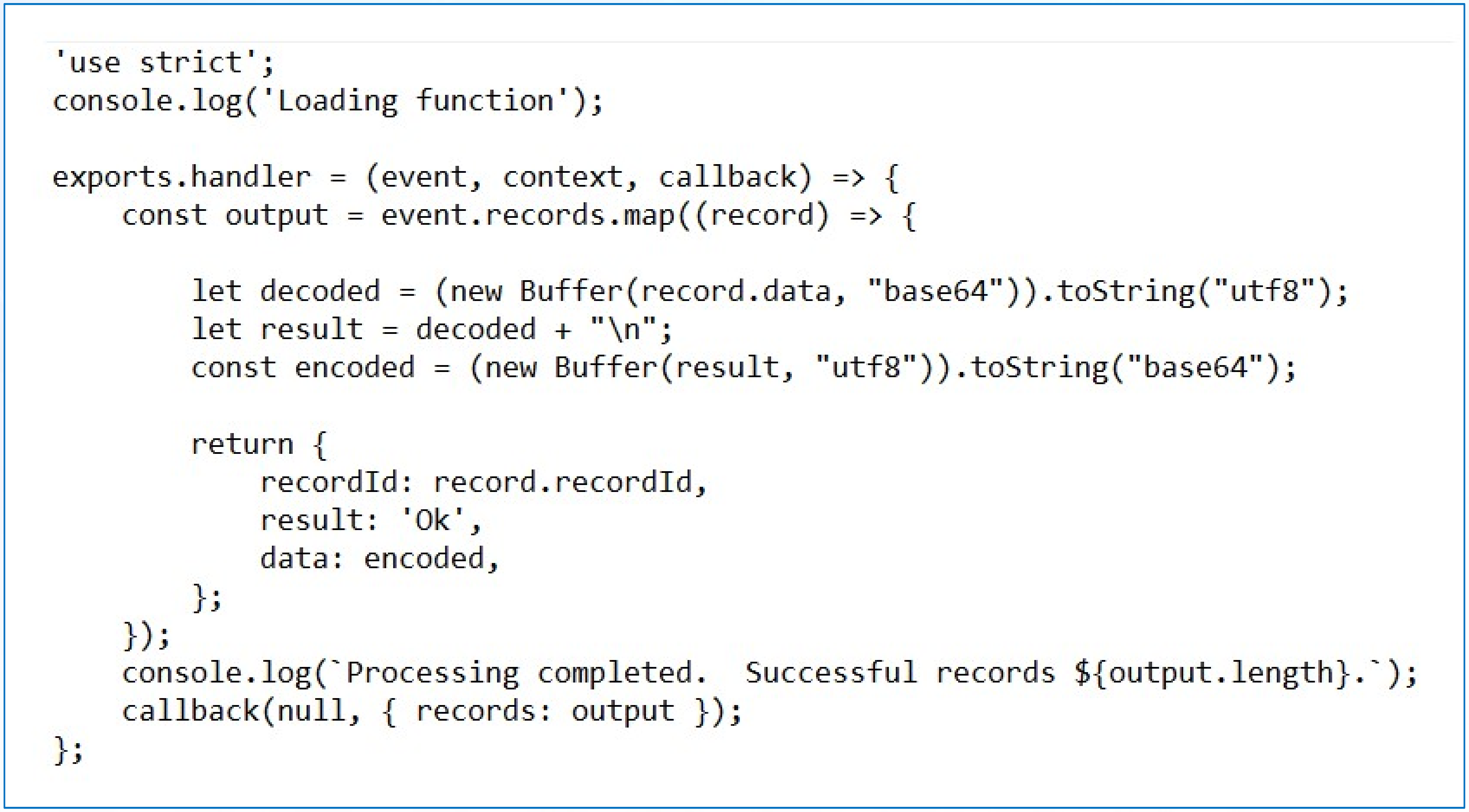
```
aws firehose put-record --delivery-stream-name demo-firehose --cli-binary-format raw-in-base64-out --record Data=100
```

NOTE: You can also test, by sending 'Test Data' to the stream.

- Open the stream
- Test with demo data
  - Click on 'Start sending demo data' button
  - After 5 or 6 seconds click on 'Stop Sending Demo Data' button

Step 4 : Check the data being added to the S3 bucket after being transformed as per the lambda code.

Demo: Lambda Function Code



```
'use strict';
console.log('Loading function');

exports.handler = (event, context, callback) => {
    const output = event.records.map((record) => {

        let decoded = (new Buffer(record.data, "base64")).toString("utf8");
        let result = decoded + "\n";
        const encoded = (new Buffer(result, "utf8")).toString("base64");

        return {
            recordId: record.recordId,
            result: 'Ok',
            data: encoded,
        };
    });
    console.log(`Processing completed.  Successful records ${output.length}.`);
    callback(null, { records: output });
};
```

THANK YOU