Serverless Streaming Architectures and Best Practices

June 2018

© 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Notices

This document is provided for informational purposes only. It represents AWS's current product offerings and practices as of the date of issue of this document, which are subject to change without notice. Customers are responsible for making their own independent assessment of the information in this document and any use of AWS's products or services, each of which is provided "as is" without warranty of any kind, whether express or implied. This document does not create any warranties, representations, contractual commitments, conditions or assurances from AWS, its affiliates, suppliers or licensors. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Contents

Executive Summary

Serverless computing allows you to build and run applications and services without thinking about servers. This means you can focus on writing business logic instead of managing or provisioning infrastructure. AWS Lambda, our serverless compute offering, allows you to write code in discrete units called functions, which are triggered to run by events. Lambda will automatically run and scale your code in response to these events, such as modifications to Amazon S3 buckets, table updates in Amazon DynamoDB, or HTTP requests from custom applications. AWS Lambda is also pay-per-use, which means you pay only for when your code is running. Using a serverless approach allows you to build applications faster, at a lower cost, and with less on-going management. AWS Lambda and serverless architectures are well-suited for stream processing workloads which are often event-driven and have spiky or variable compute requirements. Stream processing architectures are increasingly deployed to process high volume events and generate insights in near-real time.

In this whitepaper we will explore three stream processing patterns using a serverless approach. For each pattern, we'll describe how it applies to a real-world use-case, the best practices and considerations for implementation, and cost estimates. Each pattern also includes a template which enables you to easily and quickly deploy these patterns in your AWS accounts.

Introduction

What is serverless computing and why use it?

Serverless computing allows you to build and run applications and services without thinking about servers. Serverless applications don't require you to provision, scale, and manage any servers. You can build them for nearly any type of application or backend service, and everything required to run and scale your application with high availability is handled for you.

Building serverless applications means that your developers can focus on their core product instead of worrying about managing and operating servers or runtimes, either in the cloud or on-premises. This reduced overhead lets developers reclaim time and energy that can be spent on developing great products which scale and that are reliable.

Serverless applications have three main benefits:

- No server management
- Flexible scaling
- Automated high availability

In this paper we will focus on serverless stream processing applications built with our serverless compute service AWS Lambda. AWS Lambda lets you run code without provisioning or managing servers. You pay only for the compute time you consume - there is no charge when your code is not running.

With Lambda, you can run code for virtually any type of application or backend service - all with zero administration. Just upload your code and Lambda takes care of everything required to run and scale your code with high availability. You can set up your code to automatically trigger from other AWS services or call it directly from any web or mobile app.

What is streaming data?

Streaming Data is data that is generated continuously by thousands of data sources, which typically send in the data records simultaneously, and in small sizes (order of kilobytes). Streaming data includes a wide variety of data such as log files generated by mobile or web applications, e-commerce purchases, in-game player activity, information from social networks, financial trading floors, or geospatial services, and telemetry from connected devices or instrumentation in data centers.

Streaming data can be processed in real-time or near real-time, providing actionable insights that respond to changing conditions and customer behavior quicker than ever before. This is in contrast to the traditional database model, where data is stored then processed or analyzed at a later time, sometimes leading to insights derived from data that is out of date.

Who Should Read this Document

This document is targeted at Architects and Engineers seeking for a deeper understanding of serverless patterns for stream processing and best practices and considerations. We assume a working knowledge of stream processing. For an introduction to Stream Processing, please see to the Whitepaper: Streaming Data Solutions on AWS with Amazon Kinesis.

Stream Processing Application Scenarios

Streaming data processing is beneficial in most scenarios where new, dynamic data is generated on a continual basis. It applies to most big data use-cases and can be found across diverse industry verticals, as shown in Table 1. In this Whitepaper, we'll focus on the Internet of Things (IoT) industry vertical to provide examples of how to apply stream processing architectures to real-world challenges.

Scenarios/ Verticals	Accelerated Ingest-Transform-Load	Continuous Metrics Generation	Responsive Data Analysis
IoT	Sensor, device telemetry data ingestion	Operational metrics and dashboards	Device operational intelligence and alerts
Digital Ad Tech Marketing	Publisher, bidder data aggregation	Advertising metrics like coverage, yield, and conversion	User engagement with ads, optimized bid/buy engines
Gaming	Online data aggregation, e.g., top 10 players	Massively multiplayer online game (MMOG) live dashboard	Leader board generation, player-skill match
Consumer Online	Clickstream analytics	Metrics like impressions and page views	Recommendation engines, proactive care

Table 1. Streaming Data Scenarios Across Verticals.

There are several characteristics of a stream processing or real-time analytics workload:

- It must be reliable enough to handle critical updates such as replicating the changelog of a database to a replica store like a search index, delivering this data in order and without loss.
- It must support throughput high enough to handle large volume log or event data streams.
- It must be able to buffer or persist data for long periods of time to support integration with batch systems that may only perform their loads and processing periodically.
- It must provide data with latency low enough for real-time applications.

- It must be possible to operate it as a central system that can scale to carry the full load of the organization and operate with hundreds of applications built by disparate teams all plugged into the same central nervous system.
- It has to support close integration with stream processing systems.

Serverless Stream Processing

Traditionally, stream processing architectures have used frameworks like Apache Kafka to ingest and store the data, and a technology like Apache Spark or Storm to process the data in near-real time. These software components are deployed to clusters of servers along with supporting infrastructure to manage the clusters such as Apache ZooKeeper. Today, companies taking advantage of the public cloud no longer need to purchase and maintain their own hardware. However, any server-based architecture still requires them to architect for scalability and reliability and to own the challenges of patching and deploying to those server fleets as their applications evolve. Moreover, they must scale their server fleets to account for peak load and then attempt to scale them down when and where possible to lower costs—all while protecting the experience of end users and the integrity of internal systems.

Serverless compute offerings like AWS Lambda are designed to address these challenges by offering companies a different way of approaching application design – an approach with inherently lower costs and faster time to market that eliminates the complexity of dealing with servers at all levels of the technology stack. Eliminating infrastructure and moving to a per-pay-request model offers dual economic advantages:

- Problems like cold servers and underutilized storage simply cease to exist, along with their cost consequences—it's simply impossible for a serverless compute system like AWS Lambda to be cold because charges only accrue when useful work is being performed, with millisecond-level billing granularity.
- The elimination of fleet management, including the security patching, deployments, and monitoring of servers disappears, along with the challenge of maintaining the associated tools, processes, and on-call rotations required to support 24x7 server fleet uptime. Without the burden of server management, companies can direct their scarce IT resources to what matters—their business.

With greatly reduced infrastructure costs, more agile and focused teams, and faster time to market, companies that have already adopted serverless approaches are gaining a key advantage over their competitors.

Three Patterns We'll Cover

In this whitepaper, we will consider three serverless stream processing patterns:

- **Sensor Data Collection with Simple Transformation** – in this pattern, IoT sensor devices are transmitting measurements into a ingest service. As data is ingested, simple transformations can be performed to make the data suitable for downstream processing. Example use-cases: medical sensor devices generate patient data streams that must be de-identified to mask Protected Health Information (PHI) and Personally Identifiable Information (PII) to meet HIPAA compliance.
- **Stream Ingest Transform Load (ITL)** – this pattern extends the prior pattern to add field level enrichment from relatively small and static data sets. Example use-case(s): add data fields to medical device sensor data such as location information or device details looked up from a database. This is also a common pattern used for log data enrichment and transformation.
- **Real-time Analytics** – this pattern builds upon the prior patterns and adds the computation of windowed aggregations and anomaly detection. Example use-case(s): tracking user activity, performing log analytics, fraud detection, recommendation engines, and maintenance alerts in near-real-time.

In the sections that follow, we will provide an example use-case of each pattern. We will discuss the implementation choices and provide an estimate of the costs. Each sample pattern described in the paper is also available in Github (please see Appendix B) so you can quickly and easily deploy them into your AWS account.

Cost Considerations of Server-Based vs Serverless Architectures

When comparing the cost of a serverless solution against server-based approaches, you must consider several indirect cost elements that are in addition to the server infrastructure costs.

These indirect costs include additional patching, monitoring and other responsibilities of maintaining server-based applications that can require additional resources to manage.

A number of these cost considerations are listed in Table 2.

Cost Consideration	Server-based architectures	Serverless architectures
Patching	All servers in the environment must be regularly patched; this includes the Operating System (OS) as well as the suite of applications needed for the workload to function.	As there are no servers to manage in a serverless approach, these patching tasks are largely absent. You are only responsible for updating your function code when using AWS Lambda.
Security stack	Servers will often include a security stack including products for malware protection, log monitoring, host based firewalls and IDS that must be configured and managed.	Equivalent firewall and IDS controls are largely taken care of by the AWS service and service specific security logs such as CloudTrail are provided for auditing purposes without requiring setup and configuration of agents and log collection mechanisms.
Monitoring	Server-based monitoring may surface lower-level metrics that must be monitored, correlated and translated to higher service-level metrics. For example, in a stream ingestion pipeline, individual server metrics like CPU utilization, network utilization, disk IO, disk space utilization must all be monitored and correlated to understand the performance of the pipeline.	In the serverless approach, each AWS service provides CloudWatch metrics that can be directly used to understand the performance of the pipeline. For example: Kinesis Firehose publishes CloudWatch metrics for IncomingBytes, IncomingRecords and S3 DataFreshness that lets an operator understand more directly the performance of the streaming application.
Supporting infrastructure	Often, server based clusters need supporting infrastructure such as cluster management software, centralized log collection that must also be managed.	AWS manages the clusters providing AWS services and removes this burden from the customer. Further, services like AWS Lambda deliver log records to CloudWatch Logs allowing centralized log collection, processing and analysis.
Software licenses	Customers must consider the cost of licenses and commercial support for software such as the Operating Systems, streaming platforms, application servers, and packages for security, management and monitoring.	The AWS service prices include software licenses and no additional packages are needed for security, management and monitoring of these services.

Table 2. Cost considerations when comparing serverless and server-based architectures.

Example Use-case

For this whitepaper, we will focus on a use-case of medical sensor devices that are wired to a patient receiving treatment at a hospital. First, sensor data must be ingested securely at scale. Next, the patient's protected health information (PHI) is de-identified in order to be processed in an anonymized way. As part of the processing, the data may need to be enriched with additional fields or the data may be transformed. Finally, the sensor data is analyzed in real-time to derive insights such as detecting anomalies or developing trend patterns. In the sections that follow, we'll detail this use-case with example realizations of the three patterns.

Sensor Data Collection

Wearable devices for health monitoring is a fast growing IoT use-case that allow real-time monitoring of a patient's health. In order to do this, first, the sensor data must be ingested securely and at scale. It must then be de-identified to remove the patient's personal health information (PHI) so that the anonymized data can be processed in other systems downstream.

An example solution that meets these requirements is shown in Figure 1.

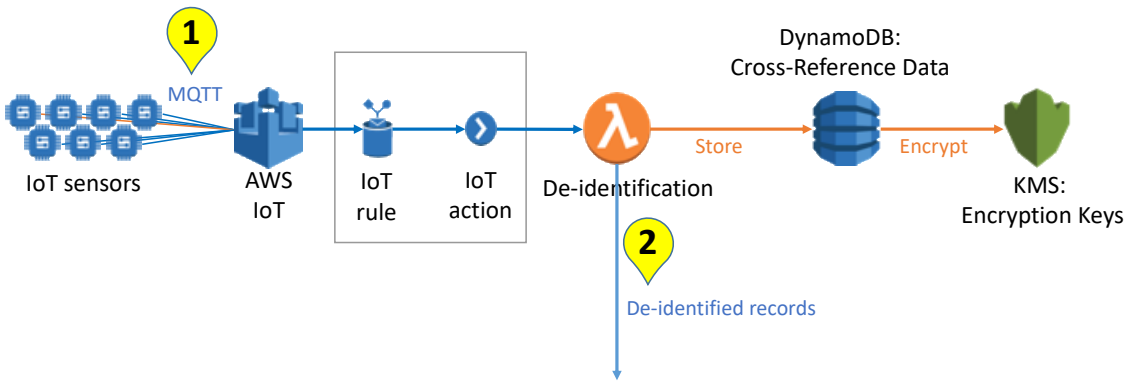


Figure 1. Overview of Medical Device Use-Case – Sensor or Device Data Collection

In Point 1 of Figure 1, one or more medical devices ("IoT sensors") are wired to a patient in a hospital. The devices transmit sensor data to the hospital IoT gateway which are then forwarded securely using the MQTT protocol to the AWS IoT gateway service for processing. A sample record at this point is:

```
{
  "timestamp": "2018-01-27T05:11:50",
  "device_id": "device8401",
  "patient_id": "patient2605",
  "name": "Eugenia Gottlieb",
  "dob": "08/27/1977",
  "temperature": 100.3,
  "pulse": 108.6,
  "oxygen_percent": 48.4,
  "systolic": 110.2,
  "diastolic": 75.6
}
```

Next, the data must be de-identified in order to be processed in an anonymized way. AWS IoT is configured with an IoT Rule that selects measurements for a specific set of patients and an IoT Action that delivers these selected measurements to a Lambda de-identification function. The Lambda performs three tasks. First the function removes PHI and PII attributes (Patient Name and Patient DOB) from the records. Second for the purpose of future cross-reference the function encrypts and stores the Patient Name and Patient DOB attributes in a DynamoDB table along with the Patient ID. And finally the function sends the de-identified records to a Kinesis Data Firehose delivery stream (Point 2 in Figure 1). A sample record at this point is shown below – note that the date of birth (“dob”) and “name” fields are removed:

```
{
  "timestamp": "2018-01-27T05:11:50",
  "device_id": "device8401",
  "patient_id": "patient2605",
  "temperature": 100.3,
  "pulse": 108.6,
  "oxygen_percent": 48.4,
  "systolic": 110.2,
  "diastolic": 75.6,
}
```

Best Practices

Consider the following best practices when deploying this pattern:

- Separate the Lambda Handler entry-point from the core logic. This allows you to make a more unit-testable function.
- Take advantage of container re-use to improve the performance of your Lambda function. Make sure any externalized configuration or dependencies that your code retrieves are stored and referenced locally after initial execution. Limit the re-initialization of variables/objects on every invocation. Instead use static initialization/constructor, global/static variables, and singletons.
- When delivering data to S3, tune the Kinesis Data Firehose buffer size and buffering interval to achieve the desired object size. With small objects, the cost of PUT and GET actions on the object will be higher.
- Use a compression format to further reduce storage and data transfer costs. Kinesis Data Firehose supports GZIP, Snappy and Zip data compression.

Cost Estimates

The monthly cost of the AWS services from the ingestion of the sensor data into AWS IoT Gateway, de-identification in a Lambda function and storing cross-reference data into DynamoDB Table can be $117.19 for the small scenario, $1,132.01 for the medium scenario and $4,977.99 for the large scenario.

Please refer to Appendix A.1 – Sensor Data Collection for a detailed breakdown of the costs per service.

Streaming Ingest Transform Load (ITL)

After sensor data has been ingested, it may need to be enriched or modified with simple transformations such as field level substitutions and data enrichment from relatively small and static data sets.

In the example use-case, it may be important to associate sensor measurements with information on the device model and manufacturer. A solution to meet this need is shown in Figure 3. De-identified records from the prior pattern are ingested into a Kinesis Data Firehose Delivery Stream (Point 2 in Figure 2).

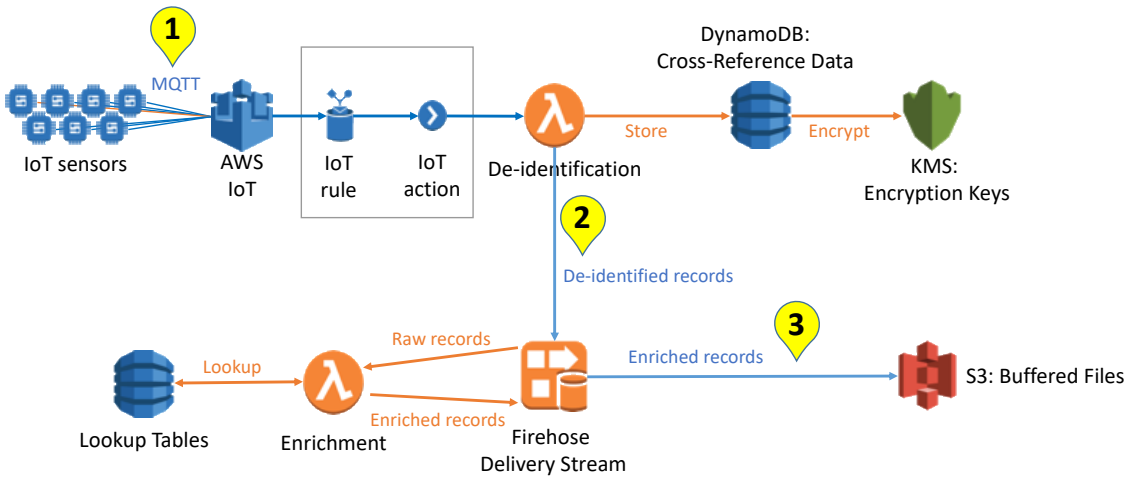


Figure 2. Overview of Medical Device Use-Case – Stream Ingest Transform Load (ITL)

The solution introduces a Lambda function that is invoked by Kinesis Data Firehose as records are received by the delivery stream. The Lambda function looks up information about each device from a DynamoDB table and adds these as fields to the measurement records. Firehose then buffers and sends the modified records to the configured destinations (Point 3 in Figure 2). A copy of the source records is saved in S3 as a backup and for future analysis. A sample record at this point is shown below with the enriched fields highlighted:

```
{
  "timestamp": "2018-01-27T05:11:50",
  "device_id": "device8401",
  "patient_id": "patient2605",
  "temperature": 100.3,
  "pulse": 108.6,
  "oxygen_percent": 48.4,
  "systolic": 110.2,
  "diastolic": 75.6,
  "manufacturer": "Manufacturer 09",
  "model": "Model 02"
}
```

Using AWS Lambda functions for transformations in this pattern removes the conventional hassle of setting up and maintaining infrastructure. Lambda runs more copies of the function in parallel in response to concurrent transformation invocations, and scales precisely with the size of the workload down to the individual request. As a result, the problem of idle infrastructure and wasted infrastructure cost is eliminated.

Once data is ingested into Firehose, a Lambda function is invoked that performs simple transformations:

- Replace the numeric timestamp information with a human readable string that allows us to query the data based on day, month or year. E.g. the timestamp “1508039751778” is converted to the timestamp string “2017-10-15T03:55:51.778000”.
- Enrich the data record by querying a table (stored in DynamoDB) using the Device ID to get the corresponding Device Manufacturer and Device Model. The function caches the device details in memory to avoid having to query DynamoDB frequently and reduce the number of Read Capacity Units (RCU). This design takes advantage of container reuse in AWS Lambda to opportunistically cache data when a container is reused.

Best Practices

Consider the following best practices when deploying this pattern:

- When delivering data to S3, tune the Kinesis Data Firehose buffer size and buffer interval to achieve your desired object size. With small objects, the cost of object actions – PUTs and GETs – will be higher.

- Use a compression format to reduce your storage and data transfer costs. Kinesis Data Firehose supports GZIP, Snappy, or Zip data compression.
- When delivering data to Redshift consider the best practices for loading data into Redshift.
- When transforming data in the Firehose delivery stream using an AWS Lambda function, consider enabling Source Record Backup for the delivery stream. This feature backs up all untransformed records to S3 while delivering transformed records to the destinations. Though this increases your storage size on S3, this backup data can come in handy if you have an error in your transformation lambda function.
- Firehose will buffer records up to the buffer size or 3MB, whichever is smaller, and invoke the transformation Lambda function with each buffered batch. Thus, the buffer size determines number of Lambda function invocations and the amount of work sent in each invocation. A small buffer size means a large number of Lambda function invocations and a larger invocation cost. A large buffer size means fewer invocations but more work per invocation and depending on the complexity of the transformation the function may exceed the 5-minute maximum invocation duration.
- The lookup during the transformation happens at the rate of ingest record rates. Consider using Amazon DynamoDB Accelerator (DAX) to cache results to reduce the latency for lookups and increase lookup throughput.

Cost Estimates

The monthly cost of the AWS services from the ingestion of the streaming data into Kinesis Data Firehose, transformations in a Lambda function and delivery of both the source records and transformed records into S3 can be as little as $18.11 for the Small scenario, $138.16 for the Medium scenario and $672.06 for the Large scenario.

Please refer to Appendix A.2 – Streaming Ingest Transform Load (ITL) for a detailed breakdown of the costs per service.

Real-Time Analytics

Once streaming data is ingested and enriched, it can now be analyzed to derive insights in real-time.

In the example use-case, the de-identified and enriched records needs be analyzed in real-time to detect anomalies with any of the devices in the hospital and notify the appropriate device manufacturers. By assessing the condition of the devices, the manufacturer can start to spot patterns that indicate when a failure is likely to arise. In addition, by monitoring information in near real-time, the hospital provider can quickly react to concerns before anything goes wrong. Should an anomaly is detected, the devices are immediately pulled out and sent for inspection. The benefits of this approach include a reduction in device downtime, increased device monitoring, lower labor costs, and more efficient maintenance scheduling. This also allows the device manufacturers to start offering hospitals more performance-based maintenance contracts.

A solution to meet this requirement is shown in Figure 3.

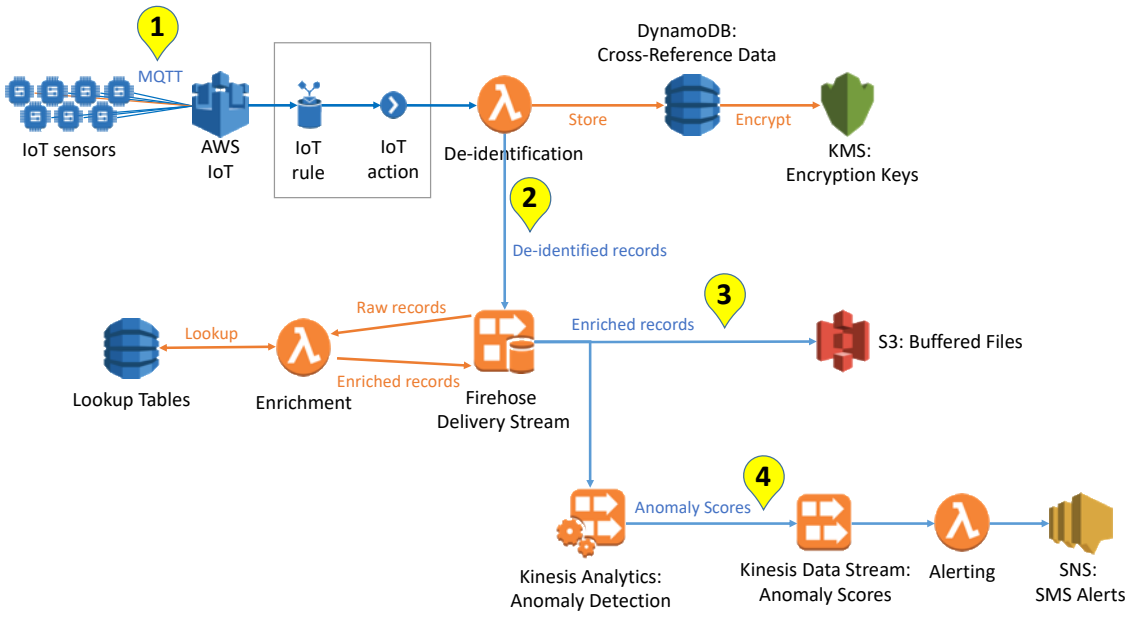


Figure 3. Overview of Medical Device Use-Case – Real-Time Analytics

Copies of the enriched records from the prior pattern (Point 4 in Figure 3) are delivered to a Kinesis Data Analytics application that detects anomalies in the measurements across all devices for a manufacturer. The anomaly scores (Point 5 in Figure 3) are sent to a Kinesis Data Stream and processed by a Lambda function. A sample record with the added anomaly score is shown below:

```
{
  "timestamp": "2018-01-27T05:11:50",
  "device_id": "device8401",
  "patient_id": "patient2605",
  "temperature": 100.3,
  "pulse": 108.6,
  "oxygen_percent": 48.4,
  "systolic": 110.2,
  "diastolic": 75.6,
  "manufacturer": "Manufacturer 09",
  "model": "Model 02",
  "anomaly_score": 0.9845
}
```

Based on a range or threshold of anomalies detected, the Lambda function sends a notification to the manufacturer with the model number and device id and a set of measurements that caused the anomaly.

The Kinesis Analytics application code consists of an anomaly detection pre-built function, **RANDOM_CUT_FOREST.** This function is the crux of the anomaly detection. The function takes the numeric data in the message, in our case "temperature", "pulse", "oxygen_percent", "systolic" and "diastolic" to determine the anomaly score.

To learn more on the function RANDOM_CUT_FOREST you read the amazon kinesis analytics document - https://docs.aws.amazon.com/kinesisanalytics/latest/sqlref/sqlrf-random-cut-forest.html

The following is an example of anomaly detection. The diagram shows three clusters and a few anomalies randomly interjected. The red squares show the records that received the highest anomaly score according to the RANDOM_CUT_FOREST function. The blue diamond represent the remaining records. Note how the highest scoring records tend to be outside the clusters.

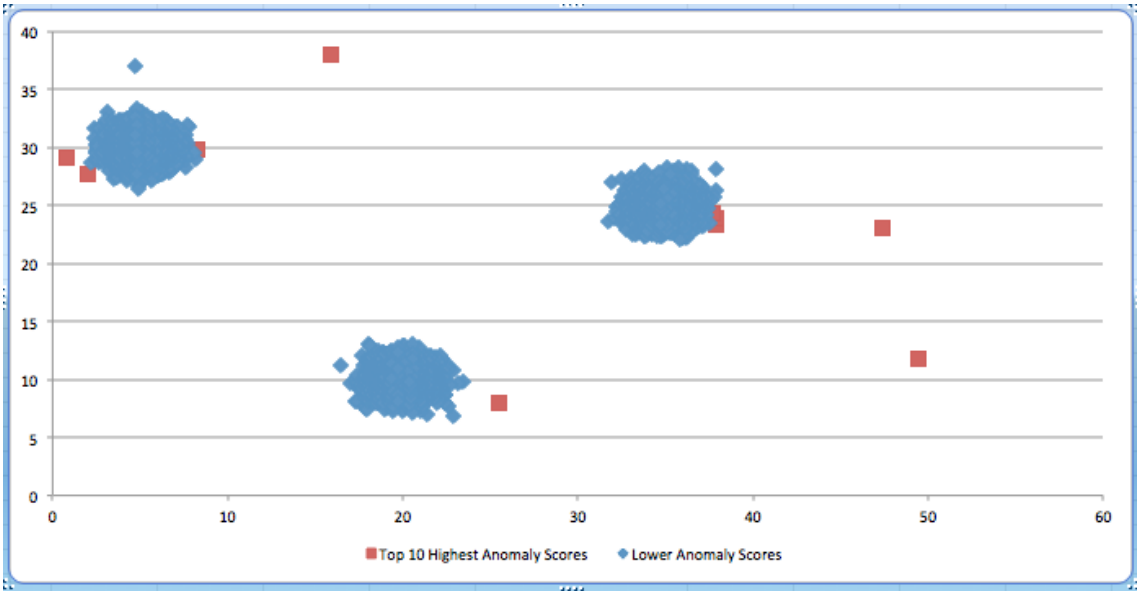


Figure 4. Example of anomaly detection.

Below is the Kinesis Analytics Application code. The first block of the code is to store the output of the anomaly score generated by the RANDOMM_CUT_FOREST function. The block of code uses the incoming sensor data stream ("STRAM_PUMP") to call the pre-built anomaly detection function RANDOM_CUT_FOREST

```
-- Creates a temporary stream and defines a schema
CREATE OR REPLACE STREAM "TEMP_STREAM" (
  "device_id"  VARCHAR(16),
  "manufacturer"        VARCHAR(16),
  "model"        VARCHAR(16),
  "temperature" integer,
  "pulse" integer,
  "oxygen_percent" integer,
  "systolic" integer,
  "diastolic" integer,
  "ANOMALY_SCORE"  DOUBLE);

-- Compute an anomaly score for each record in the source stream
-- using Random Cut Forest
CREATE OR REPLACE PUMP "STREAM_PUMP" AS INSERT INTO
"TEMP_STREAM"
SELECT STREAM
"device_id", "manufacturer", "model", "temperature", "pulse",
"oxygen_percent", "systolic", "diastolic", "ANOMALY_SCORE" FROM
```

```
TABLE(RANDOM_CUT_FOREST(
    CURSOR(SELECT STREAM "device_id", "manufacturer", "model",
"temperature", "pulse", "oxygen_percent", "systolic",
"diastolic"
      FROM "SOURCE_SQL_STREAM_001")
  )
);
```

The post-processing Lambda function in this use case performs the following simple tasks on the analytics data records with the anomaly scores:

- The Lambda function uses two environment variables called **ANOMALY_THRESHOLD_SCORE** and **SNS_TOPIC_ARN**. The environment variable **ANOMALY_THRESHOLD_SCORE** you need to set after running initial testing using controlled data to determine the appropriate value to set. The **SNS_TOPIC_ARN** is the SNS Topic to which the lambda function will deliver the anomaly records.
- The Lambda function iterates through the a batch of analytics data records looking at the anomaly score and find records that has an anomaly score that exceeds the threshold.
- The Lambda function then publishes the threshold records to the SNS Topic defined in the environment variable. In your deployment script referred in the section Appendix B.3 under Package and Deploy, you will set the variable **NotificationEmailAddress** for your e-mail that will be used to subscribe to the SNS Topic.

The sensor data is also stored into S3 making the data available for all sorts of future analysis by different data scientists working on different domains. The stream sensor data is passed to a Kinesis Firehose Delivery stream where it is buffered and zipped before doing a PUT operation into S3.

Best Practices

Consider the following best practices when deploying this pattern:

- Setup Amazon CloudWatch Alarms. Using the CloudWatch metrics that Amazon Kinesis Data Analytics provides: Input bytes and input records (number of bytes and records entering the application), Output bytes, output record and MillisBehindLatest (tracks how far behind the application is in reading from the streaming source)

- Defining Input Schema. Adequately test the inferred schema. The discovery process uses only a sample of records on the streaming source to infer a schema. If your streaming source has many record types, there is a possibility that the discovery API missed sampling one or more record types, which can result in a schema that does not accurately reflect data on the streaming source.
- Connecting To Outputs. We recommend that every application have at least two outputs. Use the first destination to insert the results of your SQL queries. Use the second destination to insert the entire error stream and send it to an S3 bucket through an Amazon Kinesis Firehose delivery stream.
- Authoring Application Code:
    - During development, keep window size small in your SQL statements so that you can see the results faster. When you deploy the application to your production environment, you can set the window size as appropriate.
    - Instead of a single complex SQL statement, you might consider breaking it into multiple statements, in each step saving results in intermediate in-application streams. This might help you debug faster.
    - When using tumbling windows, we recommend that you use two windows, one for processing time and one for your logical time (ingest time or event time). For more information, see Timestamps and the ROWTIME Column.

Cost Estimates

The monthly cost of the AWS services for doing the anomaly detection in Kinesis Analytics, reporting of the anomaly score using the lambda function to an SNS Topic and storing the anomaly score data to an S3 bucket for future analysis can be **$705.81** for the Small scenario, **$817.09** for the Medium scenario and **$1312.05** for the Large scenario.

Please refer to Appendix A.3 – Real Time Analytics for a detailed breakdown of the costs per service.

Customer Case Studies

Customers of different sizes and across different business segments are using a serverless approach for data processing and analytics. Below are some of their stories. To see more serverless case studies and customer talks, go to our AWS Lambda Resources page.

THOMSON REUTERS

Thomson Reuters is a leading source of information—including one of the world's most trusted news organizations—for the world's businesses and professionals. In 2016, Thomson Reuters decided to build a solution that would enable it to capture, analyze, and visualize analytics data generated by its offerings, providing insights to help product teams continuously improve the user experience. This solution, called Product Insights, ingests and delivers data to a streaming data pipeline using AWS Lambda and Amazon Kinesis Streams and Amazon Kinesis Data Firehose. The data is then piped into permanent storage or into an Elasticsearch cluster for real-time data analysis. Thomson Reuters can now process up to 25 billion events per month.

Read the case study »

iRobot is a leading global consumer robot company, designs and builds robots that empower people to do more both inside and outside the home. iRobot created the home-cleaning robot category with the introduction of its Roomba Vacuuming Robot in 2002. Today, iRobot reports that connected Roomba vacuums operate in more than 60 countries, with total sales of connected robots projected to reach more than 2 million by the end of 2017. To handle such scale at a global level, iRobot implemented a completely serverless architecture for its mission-critical platform. At the heart of this solution is AWS Lambda, AWS IoT Platform and Amazon Kinesis. With serverless, iRobot is able to keep the cost of the cloud platform low, and manage the solution with fewer than 10 people.

Read the case study »

Nextdoor

Nextdoor is a free private social-network for neighborhoods. The Systems team at Nextdoor is responsible for managing the data ingestion pipeline, which services 2.5 billion syslog and tracking events per day. As the data volumes grew, keeping the data ingestion pipeline stable became a full time endeavor that distracted the team from core responsibilities, like developing the product.

Rather than continue running a large infrastructure to power data pipelines, Nextdoor decided to implement a serverless ETL built on AWS Lambda. See Nextdoor's 2017 AWS re:Invent talk to learn more about Nextdoor's serverless solution and how you can leverage Nextdoor-scale serverless ETL through their open-source project Bender.

Hear the Nextdoor talk »

Conclusion

Serverless computing eliminates the undifferentiated heavy-lifting associated with building and managing server infrastructure at all levels of the technology stack, and introduces a pay-per-request billing model where there are no more costs from idle compute capacity. With data stream processing you can evolve your applications from traditional batch processing to real-time analytics which allows you to extract deeper insights on how your business performs. In this whitepaper, we reviewed how by combining these two powerful concepts developers can work with a clean application model that helps them deliver complex data processing applications faster and organizations to only pay for useful work. To learn more about serverless computing, visit our page Serverless Computing and Applications. You can also see more resources, customer talks, and tutorials on our Serverless Data Processing page.

Further Resources

For more serverless data processing resources, including tutorials, documentation, customer case studies, talks, and more, visit our Serverless Data Processing Page. For more resources on serverless and AWS Lambda, please see the AWS Lambda Resources page.

Read related whitepapers about serverless computing and data processing:

- Streaming Data Solutions on AWS with Amazon Kinesis
- Serverless: Changing the Face of Business Economics
- Optimizing Enterprise Economics with Serverless Architectures

aws

Contributors

The following individuals and organizations contributed to this document:

- Akhtar Hossain, Sr. Solutions Architect, Global Life science, Amazon Web Services
- Maitreya Ranganath, Solutions Architect, Amazon Web Services
- Linda Lian, Product Marketing Manager, Amazon Web Services
- David Nasi, Product Manager, Amazon Web Services

Document Revisions

Date	Description
Month YYYY	Brief description of revisions.
Month YYYY	First publication

Appendix A – Detailed Cost Estimates

In this Appendix we provide the detailed costs estimates that were summarized in the main text.

Common Cost Assumptions

We estimate the monthly cost of the resources required to implement each pattern for three traffic scenarios:

- Small – peak rate of 50 records / second, average 1 KB per record
- Medium – peak rate of 1000 records / second, average 1 KB per record
- Large – peak rate of 5000 records / second, average 1 KB per record

We assume that there are 4 peak hours in a day where records are ingested at the peak rate for the scenario. In the rest of the 20 hours, the rate falls to 20% of the peak data rate. This is a simple variable rate model used to estimate the volume of data ingested monthly.

Appendix A.1 – Sensor Data Collection

The detailed monthly cost of the Sensor Data Collection pattern is estimated in Table 3 below. The services are configured as follows:

- AWS IoT Gateway Service Connectivity per day is assumed at 25%/day for the small use case, 50%/day for the medium use case and 70%/day for the large use case.

- Kinesis Firehose buffer size is 100MB
- Kinesis Firehose buffer interval is 5 minutes (300 seconds)

	Small	**Medium**	**Large**
Peak Rate (Messages/Sec)	100	1000	5000
Record Size (KB)	1	1	1
Daily Records (Numbers)	2880000	28800000	144000000
Monthly Records (Numbers)	86400000	864000000	4320000000
Monthly Volume (KB)	86400000	864000000	4320000000
Monthly Volume (GB)	82.39746094	823.9746094	4119.873047
Monthly Volume (TB)	0.08046627	0.804662704	4.023313522

(Table continued on next page)

AWS IoT Costs			
No of Devices	1	1	1
Connectivity - Percentage Time / day	25	50	75
Messaging (Number of Msgs/day)	2880000	28800000	144000000
Rules Engine (Number of Rules)	1	1	1
Device Shadow	0	0	0
Device Registry	0	0	0
Total Cost - Based on AWS IoT Core Calculator	$112.00	$1,123.00	$4,952.00
Amazon Kinesis Firehose Delivery Stream			
Record Size Rounded Up to 5 KB	5	5	5
Monthly Volume for Firehose(KB)	432000000	4320000000	21600000000
Monthly Volume for Firehose(GB)	411.9873047	4119.873047	20599.36523
Firehose Monthly Cost	11.94763184	119.4763184	597.3815918
Amazon Dynamo DB			
RCU	1	1	1
WCU	10	10	10
Size (MB)	1	1	1
RCU Cost	0.0936	0.0936	0.0936
WCU Cost	4.68	4.68	4.68
Size Cost	0	0	0
DynamoDB Monthly Cost	4.7736	4.7736	4.7736

AWS Key Management Service (KMS) Cost			
Monthly Record Number	86400000	864000000	4320000000
Number of Encryption Request - 20,000 free	86380000	863980000	4319980000
Encryption Cost	259.14	2591.94	12959.94
KMS Monthly Cost	259.14	2591.94	12959.94
AWS Lambda			
Invocations	59,715	597,149	2,985,745
Duration (ms)	16,496,242	164,692,419	824,812,095
Memory(MB)	1536	1536	1536
Memory-Duration (GB/Sec)	24,744.35	247,443.63	1,237,218.14
Lambda Monthly Cost	0.42	4.24	21.22
Estimated Total Monthly Cost	**$388.28**	**$3,843.43**	**$18,535.32**

Table 3. Sensor data collection - details of estimated costs.

Appendix A.2 – Streaming Ingest Transform Load (ITL)

The detailed monthly cost of the Streaming Ingest Transform Load (ITL) pattern is estimated in Table 4 below. The services are configured as follows:

- Kinesis Firehose buffer size is 100MB
- Kinesis Firehose buffer interval is 5 minutes (300 seconds)
- Buffered records are stored in S3 compressed using GZIP, assuming 1/4 compression ratio.

	Small	Medium	Large
Peak Rate (records/second)	100	1000	5000
Record Size (KB)	1	1	1
Amazon Kinesis Firehose			
Monthly Volume (GB) (Note 1)	411.987	4,119.87	20,599.37
Kinesis Monthly Cost	**$11.95**	**$119.48**	**$597.38**
Amazon S3			
Source Record Backup Storage (GB)	21.02	210.22	1,051.08
Transformed Records Storage (GB)	21.02	210.22	1,051.08
PUT API Calls (Note 2)	17280	17280	84375
S3 Monthly Cost	**$2.47**	**$23.87**	**$119.35**
AWS Lambda			
Invocations	59,715	597,149	2,985,745
Duration (ms)	16,496,242	164,962,419	824,812,095
Function Memory (MB)	1536	1536	1536
Memory-Duration (GB-seconds)	24,744.36	247,443.63	1,237,218.14
Lambda Monthly Cost (Note 3)	**$0.42**	**$4.24**	**$21.22**
Amazon DynamoDB			
Read Capacity Units (Note 4)	50	50	50
DynamoDB Monthly Cost	**$4.68**	**$4.68**	**$4.68**
Total Monthly Cost	**$18.11**	**$138.16**	**$672.06**

Table 4. Streaming Ingest Transform Load (ITL) - details of estimated costs.

Notes:

1. Kinesis Firehose rounds up the record size to the nearest 5KB. In the three scenarios above, each 1KB record is rounded up to 5KB when calculating the monthly volume.

2. S3 PUT API calls were estimated assuming one PUT call per S3 object created by the Firehose delivery stream. At low record rates, the number of S3 objects is determined by the Firehose buffer duration (5 minutes). At high record rates, the number of S3 objects is determined by the Firehose buffer size (100MB).
3. The AWS Lambda free tier includes 1M free requests per month and 400,000 GB-seconds of compute time per month. The monthly cost estimated above is before the free tier is applied.
4. The DynamoDB Read Capacity Units (RCU) estimated above were the result of caching lookups in memory and taking advantage of container reuse. This meant that the number of RCU required on the Table is reduced.

Appendix A.3 – Real-Time Analytics

The detailed monthly cost of the Real-Time Analytics pattern is estimated in the Table 5 below.

	Small	**Medium**	**Large**
Peak Rate (Messages/Sec)	100	1000	5000
Record Size (KB)	1	1	1
Daily Records (Numbers)	2880000	28800000	144000000
Monthly Records (Numbers)	86400000	864000000	4320000000
Monthly Volume (KB)	86400000	864000000	4320000000
Monthly Volume (GB)	82.39746094	823.9746094	4119.873047
Monthly Volume (TB)	0.08046627	0.804662704	4.023313522
Amazon Kinesis Analytics			
Peak Hours in a day (hrs)	4	4	4
Average Hours in a day (hrs)	20	20	20
Kinesis Processing Unit (KPU)/hr - Peak	2	2	2
Kinesis Processing Unit (KPU)/hr - Avg	1	1	1
Kinesis Analytics Monthly Cost	$692.40	$692.40	$692.40
Amazon Kinesis Firehose Delivery Stream			
Record Size Rounded Up to 5 KB	5	5	5
Monthly Volume for Firehose (KB)	432000000	4320000000	21600000000
Monthly Volume for Firehose(GB)	411.9873047	4119.873047	20599.36523
Kinesis Firehose Monthly Cost	11.94763184	119.4763184	597.3815918

(Table continued on next page)

	Small	Medium	Large
Amazon S3			
S3 PUTs per Month based on Size only	843.75	843.75	843.75
S3 PUTs per Month based on Time only	8640	8640	8640
Expected S3 PUTs (max of size & time)	8640	8640	8640
Total Puts (source backup + Analytics data)	17280	17280	17280
Analytics Data Compressed (GB)	21.02150444	21.02	21.02
Source Data Compressed (GB)	21.02	21.02	21.02
Source Record Backup	0.48346	0.48346	0.48346
PUTs	0.0864	0.0864	0.0864
Analytics Data Records	0.483494602	0.48346	0.48346
S3 Monthly Cost	1.053354602	1.05332	1.05332
AWS Lambda			
Invocations	59,715	597,149	2,985,745
Duration (ms)	16,496,242	164,692,419	824,812,095
Memory(MB)	1536	1536	1536
Memory-Duration (GB/Sec)	24,744.35	247,443.63	1,237,218.14
Lambda Monthly Cost	0.42	4.24	21.22
Estimated Total Monthly Cost	$705.82	$817.17	$1,312.05

Table 5. Real-time analytics - details of estimated costs.

Appendix B – Deploying and Testing Patterns

Common Tasks

Implementation details of the three patterns are described in the following sections. Each pattern can be deployed, ran, and tested independently of the other patterns. To deploy each pattern, we provide links to the AWS Serverless Application Model (AWS SAM) template that can be deployed to any AWS Region. AWS SAM extends AWS CloudFormation to provide a simplified syntax for defining the Amazon API Gateway APIs, AWS Lambda functions, and Amazon DynamoDB tables needed by your serverless application.

The solutions for three patterns can be downloaded from the public GitHub repo below:

https://github.com/aws-samples/aws-serverless-stream-ingest-transform-load
https://github.com/aws-samples/aws-serverless-real-time-analytics
https://github.com/awslabs/aws-serverless-sensor-data-collection

Create or Identify an S3 Bucket for Artifacts

To use the AWS Serverless Application Model (SAM), you need an S3 bucket where your code and template artifacts are uploaded. If you already have a suitable bucket in your AWS Account, you can simply note the S3 bucket name and skip this step. If you instead choose to create a new bucket then you can follow the steps below:

1. Log into the S3 console.
2. Choose **Create Bucket** and type a bucket name. Ensure that the name is globally unique – we suggest a name like <random-string>-stream-artifacts. Choose the AWS Region where you want to deploy the pattern.
3. Choose **Next** on the following pages to accept the defaults. On the last page, choose **Create Bucket** to create the bucket. Note the name of the bucket as you'll need it to deploy the three patterns below.

Create an Amazon Cognito User for Kinesis Data Generator

To simulate hospital devices to test the Streaming Ingest Transform Load (ITL) and Real-Time Analytics patterns, you will use the Amazon Kinesis Data Generator (KDG) tool. You can learn more about the KDG Tool in this blog post.

You can access the Amazon Kinesis Data Generator here. Click on the Help menu and follow the instructions to create a Cognito username and password that will use to log into the KDG.

Appendix B.1 – Sensor Data Collection

In this section we will describes how you can deploy the use-case into your AWS Account and then run and test.

Review SAM Template

Review the Serverless Application Model (SAM) template in the file 'SAM-For-SesorDataCollection.yaml' by opening the file in an editor of your choice. You can use Notepad++ which renders the JSON file nicely.

This template creates the following resources in your AWS Account:

- An S3 Bucket that is used to store the De-Identified records.
- A Firehose Delivery Stream and associated IAM Role used to buffer and collect the De-Identified records compressed in a zip file and stored in the S3 Bucket
- An AWS Lambda Function that performs the De-Identification of the incoming messages by removing the PHI/PII Data. The function also stores the PHI / PII Data into DynamoDB along with PatientID for cross-reference. The PHI / PII data are encrypted using AWS KMS Keys.
- An AWS Lambda Function that does hospital Device Simulation for the use case. The Lambda function uses generates sensor simulation data and publishes to IoT MQTT Topic
- A DynamoDB table that stores encrypted cross-reference data Patient ID, Timestamp, Patient Name and Patient Date of Birth.

Package and Deploy

Follow the following steps to package and deploy the Sensor Data Collection scenario:

1. Clone and download the files from the GitHub folder here to a folder on your local machine. On your local machine make sure you have the following files:

   1.1 DeIdentification.zip
   1.2 PublishIoTData.zip
   1.3 SAM-For-SesorDataCollection.yaml
   1.4 deployer-sensordatacollection.sh

2. Create an **S3 Deployment Bucket** in the AWS Region where you intend to deploy the solution. Note down the S3 bucket name. You will need the S3 bucket name later.

3. From your local machine upload the following lambda code zip files into the **S3 Deployment Bucket** you just created in Step 2:

   3.1 DeIdentification.zip
   3.2 PublishIoTData.zip

4. In the AWS Management console launch an ec2 Linux instance that will be used to run the CloudFormation template. Launch an ec2 instance of type - Amazon Linux AMI 2018.03.0 (HVM), SSD Volume Type (t2.macro) ec2 in the AWS Region where you want to deploy the solution. Make sure you enable SSH access to the instance. For details on how to launch an ec2 instance and enable SSH access see - https://aws.amazon.com/ec2/getting-started/

5. On your local machine open the deployer-sensordatacollection.sh file in a text editor and update the three variables indicated as **PLACE_HOLDER** – **S3ProcessedDataOutputBucket** (the S3 bucket Name where the Processed Output Data will be stored), **LamdaCodeUriBucket** (the S3 Bucket Name you created in Step 2 and uploaded the lambda code files) and the environment variable **REGION** to the AWS Region where you intend to deploy the solution. Save the deployer-sensordatacollection.sh file.

6. Once the ec2 instance you just launched is the instance state running, using SSH log into the ec2 Linux box. Create a folder called samdeploy under /home/ec2-user/. Upload the following files into the folder /home/ec2-user/samdeploy

   5.1 SAM-For-SesorDataCollection.yaml
   5.2 deployer-sensordatacollection.sh

7. On the ec2 instance change directory to /home/ec2-user/samdeploy. Next you will run two ClouFormation CLI commands called package and deploy. Both the steps are in a single script file **deployer-sensordatacollection.sh**. Review the script file. You can now execute the package and deploy the SAM template by running the following command at the command prompt:

   $ sh ./deployer-sensordatacollection.sh

View Stack Details

You can view the progress of the stack creation by logging into the CloudFormation console. Ensure you choose the AWS Region where you deployed the stack. Locate the Stack named **SensorDataCollectionStack** from the list of stacks, choose the Events tab and refresh the page to see the progress of resource creation.

The stack creation takes approximately 3-5 minutes. The stack's state will change to **CREATE_COMPLETE** once all resources are successfully created.

Test the Pipeline

The **SensorDataCollectionStack** includes an **IoT Device Simulator Lambda Function** called PublishToIoT. The lambda function is triggered by AWS CloudWatch event rule. The event rule invokes the lambda function on a schedule of every 5 minutes. The Lambda function generates simulated sensor device messages matching the pattern discussed earlier and publishes it to the MQTT topic.

The function takes a JSON string as input called the **SimulatorConfig** to set the number of messages to generate per invocation. In our example we have set 10 messages per invocation of the lambda function. The input parameter to the lambda function is set to the JSON string {"NumberOfMsgs": "10"}.

The solution will start immediately after the stack has deployed successfully. You observe the followings:

1. The CloudWatch Event / Rule triggers every 5 min to invoke the Device Simulator lambda function. The lambda function is configured to generate by default 10 sensor data messages per invocation and publish these to the IoT Topic – "LifeSupportDevice/Sensor".
2. The processed data (without the PHI / PII) will appear in the **S3 Processed Data Bucket.**
3. In the DynamoDB console you will see the cross reference data composed of the PatientID, PatientName and PatientDOB in the Table – PatientReferenceTable.

To stop the Testing of the pattern simply go to the CloudWatch console and disable the Events/Rule called - SensorDataCollectionStack-IoTDeviceSimmulatorFunct-XXXXXXX

NOTE:
At the time of writing this whitepaper the team at AWS Solution Group has created a robust IoT Device Simulator. To help customers more easily test device integration and IoT backend services. This solution provides a web-based graphical user interface (GUI) console that enables customers to create and simulate hundreds of virtual connected devices, without having to configure and manage physical devices, or develop time-consuming scripts. More details can be found at https://aws.amazon.com/answers/iot/iot-device-simulator/

*However our simple pattern you will use the **IoT Device Simulator Lambda Function** that is invoke by the CloudWatch Event/Rule. By default the Rule is scheduled to trigger every 5 minute*

Cleaning up Resources

Once you have tested this pattern, you can delete and clean up the resources created so that you are not charged for these resources.

1. On the CloudWatch Console in the AWS Region where you deployed the pattern, under Events / Rules disable the Rule – SensorDataCollectionStack-IoTDeviceSimmulatorFunct-XXXXXXX

1. On the S3 console, choose the output **S3 Processed Data Bucket** and choose **Empty Bucket**.
2. On the CloudFormation console, choose the **SensorDataCollectionStack** stack and choose **Delete Stack**.
3. Finally on the EC2 console terminate the ec2 Linux instance you created to run the CloudFormation template to deploy the solution

Appendix B.2 – Streaming Ingest Transform Load (ITL)

In this section, we'll describe how you can deploy the pattern in your AWS Account and test the transformation function and monitor the performance of the pipeline.

Review SAM Template

Review the Serverless Application Model (SAM) template in the file 'streaming_ingest_transform_load.template'.

This template creates the following resources in your AWS Account:

- An S3 Bucket that is used to store the transformed records and the source records from Kinesis Firehose.
- A Firehose Delivery Stream and associated IAM Role used to ingest records.
- An AWS Lambda Function that performs the transformation and enrichment described above.
- A DynamoDB table that stores device details that are looked up by the transformation function.

- An AWS Lambda Function that inserts sample device detail records into the DynamoDB table. This function is invoked once as a custom CloudFormation resource to populate the table when the stack is created.
- A CloudWatch Dashboard that makes it easy to monitor the processing pipeline.

Package and Deploy

In this step, you'll use the CloudFormation package command to upload local artifacts to the artifacts S3 bucket you chose or created in the previous step. This command also returns a copy of the SAM template after replacing references to local artifacts with the S3 location where the package command uploaded your artifacts.

After this, you will use the CloudFormation deploy command to create the stack and associated resources.

Both steps above are included in a single script deployer.sh in the github repository. Before executing this script, you need to set the artifact S3 bucket name and region in the script. Edit the script in any text editor and replace PLACE_HOLDER with the name of the S3 bucket and region from the previous section. Save the file.

You can package and deploy the SAM template by running the following command:

```
$ sh ./deployer.sh
```

View Stack Details

You can view the progress of the stack creation by logging into the CloudFormation console. Ensure you choose the AWS Region where you deployed the stack. Locate the Stack named **StreamingITL** from the list of stacks, choose the Events tab and refresh the page to see the progress of resource creation.

The stack creation takes approximately 3-5 minutes. The stack's state will change to **CREATE_COMPLETE** once all resources are successfully created.

Test the Pipeline

Follow the steps below to test the pipeline:

1. Log into the CloudFormation console and locate the stack for the Kinesis Data Generator Cognito User you created in Create an Amazon Cognito User for Kinesis Data Generator above.
2. Choose the **Outputs** tab and click on value for the key **KinesisDataGeneratorUrl**.
3. Log in with the **username** and **password** you used when creating the Cognito User CloudFormation stack earlier.
4. From the Kinesis Data Generator, choose the Region where you created the serverless application resources, choose the **IngestStream** delivery stream from the drop down.
5. Set the Records per second as 100 to test the first traffic scenario.
6. Set the Record template as the following to generate test data:

```
{
"timestamp" : {{date.now("x")}},
"device_id" : "device{{helpers.replaceSymbolWithNumber("####")}}",
"patient_id" : "patient{{helpers.replaceSymbolWithNumber("####")}}",
"temperature" : {{random.number({"min":96,"max":104})}},
"pulse" : {{random.number({"min":60,"max":120})}},
"oxygen_percent" : {{random.number(100)}},
"systolic" : {{random.number({"min":40,"max":120})}},
"diastolic" : {{random.number({"min":40,"max":120})}},
"text" : "{{lorem.sentence(140)}}"
}
```

We are using a **text** field in the template to ensure that our test records are approximately 1KB in size as required by the scenarios.

7. Choose **Send Data** to send generated data at the chosen rate to the Kinesis Firehose Stream.

Monitor the Pipeline

Follow the steps below to monitor the performance of the pipeline and verify the resulting objects in S3:

1. Switch to the CloudWatch Console and choose Dashboards from the menu on the left.
2. Choose the Dashboard named **StreamingITL**.
3. View the metrics for Lambda, Kinesis Firehose and DynamoDB on the dashboard. Choose the duration to zoom into a period of interest.

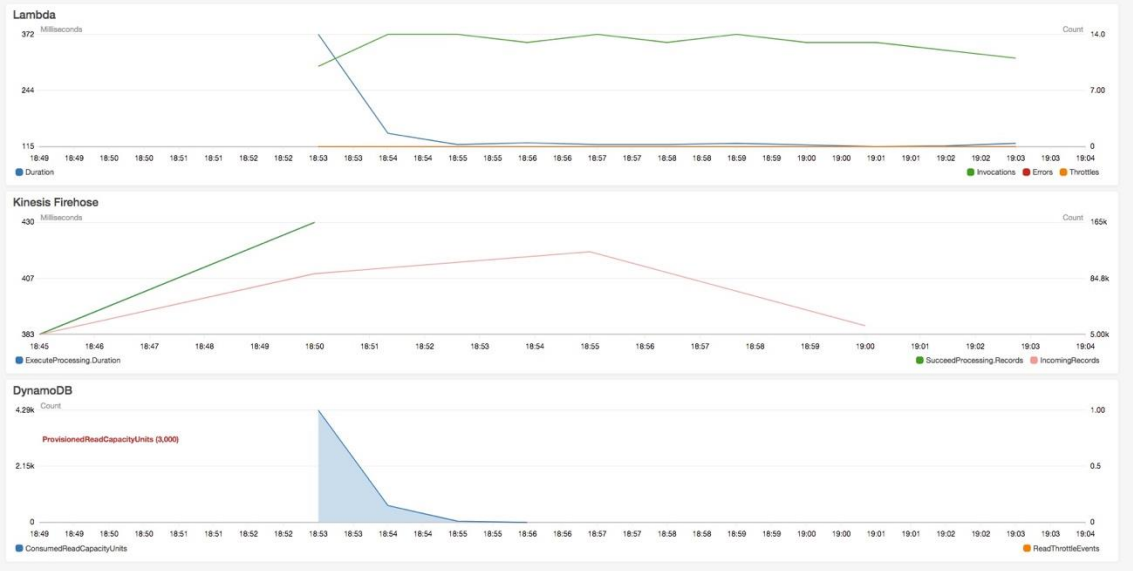


Figure 7. CloudWatch Dashboard for Streaming ITL.

4. After around 5-8 minutes, you will see transformed records arrive in the output S3 bucket under the prefix **transformed/**.
5. Download a sample object from S3 to verify its contents. Note that objects are stored GZIP compressed to reduce space and data transfers.
6. Verify that the transformed records contain a human readable time-stamp string, device model and manufacturer. These are enriched fields looked up from the DynamoDB table.

7. Verify that a copy of the untransformed source records is also delivered to the same bucket under the prefix **source_records/**.

Once you have verified the pipeline is working correctly for the first traffic scenario, you can now increase the rate of messages to 1000 requests / second and then to 5000 requests / second.

Cleaning up Resources

Once you have tested this pattern, you can delete and clean up the resources created so that you are not charged for these resources.

4. Stop sending data from the Kinesis Data Generator.
5. On the S3 console, choose the output S3 bucket and choose **Empty Bucket**.
6. On the CloudFormation console, choose the **StreamingITL** stack and choose **Delete Stack**.

Appendix B.3 – Real-Time Analytics

In this section describes how you can deploy the use-case into your AWS Account and then run and test.

Review SAM Template

Review the Serverless Application Model (SAM) template in the file 'SAM-For-RealTimeAnalytics.yaml' by opening the file in a text editor of your choice. You can use Notepad++ which renders the JSON file nicely.

This template creates the following resources in your AWS Account:

- An S3 Bucket (**S3ProcessedDataOutputBucket**) that is used to store the Real-Time Analytics records containing the anomaly score
- A Firehose Delivery Stream and the associated IAM Role used as an input stream to the Kinesis Analytic service.
- A Kinesis Analytics Application named **DeviceDataAnalytics** with one input stream (Firehose Delivery Stream), Application Code (SQL Statements), a Destination

Connection (Kinesis Analytics Application Output) as a Lambda Function and a second Destination Connection (Kinesis Analytics Output) as Kinesis Firehose Delivery Stream

- A SNS Topic named **publishtomanufacturer** and an e-mail subscription the SNS Topic. You configure the e-mail in the deployment script **deployer-realtimeanalytics.sh.** The variable to set your e-mail is named **NotificationEmailAddress** in the deployment script.
- An AWS Lambda Function that interrogates the data record set received from the Analytics Stream picking up and publishing the record to a SNS Topic where the anomaly score is higher than a threshold defined (in this case in the Lambda function environment variable).
- A second AWS Lambda Function named KinesisAnalyticsHelper that is used to start the Kinesis Analytics Application **DeviceDataAnalytics** immediately after the Kinesis Analytics Application is created
- A Kinesis Firehose Delivery Stream that aggregates that records from the Analytics Destination Stream, buffers the record and zips and put the zipped file into the S3 bucket (**S3ProcessedDataOutputBucket).**

Package and Deploy

Follow the following steps to package and deploy the Real-Time Analytics scenario:

1. Clone and download the files from the GitHub folder here to a folder on your local machine. On your local machine make sure you have the following files:

   1.1 KinesisAnalyticsOuputToSNS.zip
   1.2 SAM-For-RealTimeAnalytics.yaml
   1.3 deployer-realtimeanalytics.sh

2. Create an **S3 Deployment Bucket** in the AWS Region where you intend to deploy the solution. Note down the S3 bucket name. You will need the S3 bucket name later.
3. From your local machine upload the following lambda code zip files into the **S3 Deployment Bucket** you just created in Step 2:

   3.1 KinesisAnalyticsOuputToSNS.zip

4. In the AWS Management console launch an ec2 Linux instance that will be used to run the CloudFormation template. Launch an ec2 instance of type - Amazon Linux AMI 2018.03.0 (HVM), SSD Volume Type (t2.macro) ec2 in the AWS Region where you want to deploy the solution. Make sure you enable SSH access to the instance. For

details on how to launch an ec2 instance and enable SSH access see - https://aws.amazon.com/ec2/getting-started/

5. On your local machine open the deployer-realtimeanalytics.sh file in a text editor and update the five variables indicated as **PLACE_HOLDER** -**S3ProcessedDataOutputBucket** (the S3 bucket Name where the Processed Output Data will be stored), **NotificationEmailAddress** (the e-mail address you specify to receive notification that the anomaly score has exceeded a threshold value), **AnomalyThresholdScore** (the threshold value that the Lambda function will use to identify the records to send for notification**), LamdaCodeUriBucket** (the S3 Bucket Name you created in Step 2 and uploaded the lambda code files) and the variable **REGION** to the AWS Region where you intend to deploy the solution. Save the deployer-sensordatacollection.sh file.

6. Once the ec2 instance you just launched is in the instance state running, using SSH, log into the ec2 Linux box. Create a folder called samdeploy under /home/ec2-user/. Upload the following files into the folder /home/ec2-user/samdeploy

   6.1 SAM-For-RealTimeAnalytics.yaml
   6.2 deployer-realtimeanalytics.sh

7. On the ec2 instance change directory to /home/ec2-user/samdeploy. Next you will run two ClouFormation CLI commands called package and deploy. Both the steps are in a single script file **deployer-realtimeanalytics.sh**. Review the script file. You can now execute the package and deploy the SAM template by running the following command at the command prompt:

   ```
   $ sudo yum install dos2unix
   $ dos2unix deployer-realtimeanalytics.sh
   $ sh  ./ deployer-realtimeanalytics.sh
   ```

8. As part of the deployment of the pattern, an e-mail (the e-mail specified in step 5) subscription is setup to the SNS Topic. Check your e-mail in-box for an e-mail requesting **subscription confirmation**. Open the e-mail and confirm the subscription verification. Subsequently you will be receiving e-mail notifications for the device data records that has exceeded the specified threshold.

View Stack Details

You can view the progress of the stack creation by logging into the CloudFormation console. Ensure you choose the AWS Region where you deployed the stack. Locate the Stack named **DeviceDataRealTimeAnalyticsStack** from the list of stacks, choose the Events tab and refresh the page to see the progress of resource creation.

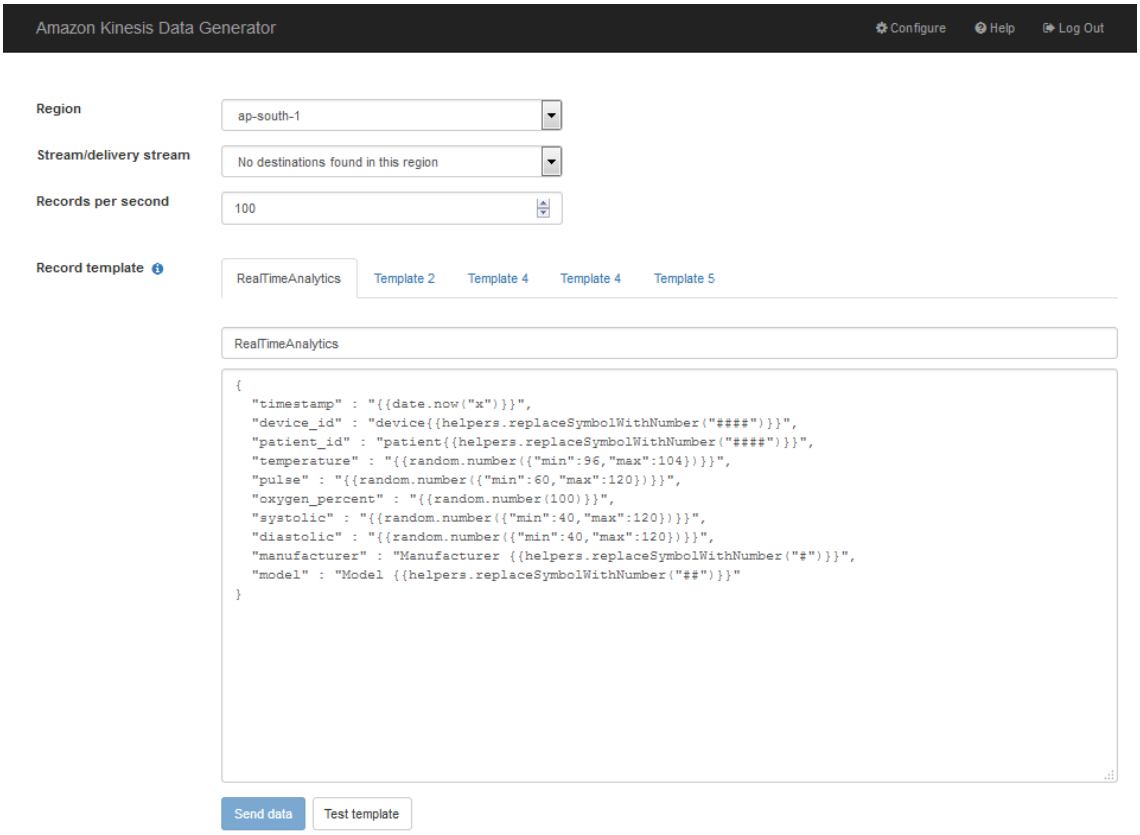
The stack creation takes approximately 3-5 minutes. The stack's state will change to **CREATE_COMPLETE** once all resources are successfully created.

Test the Pipeline

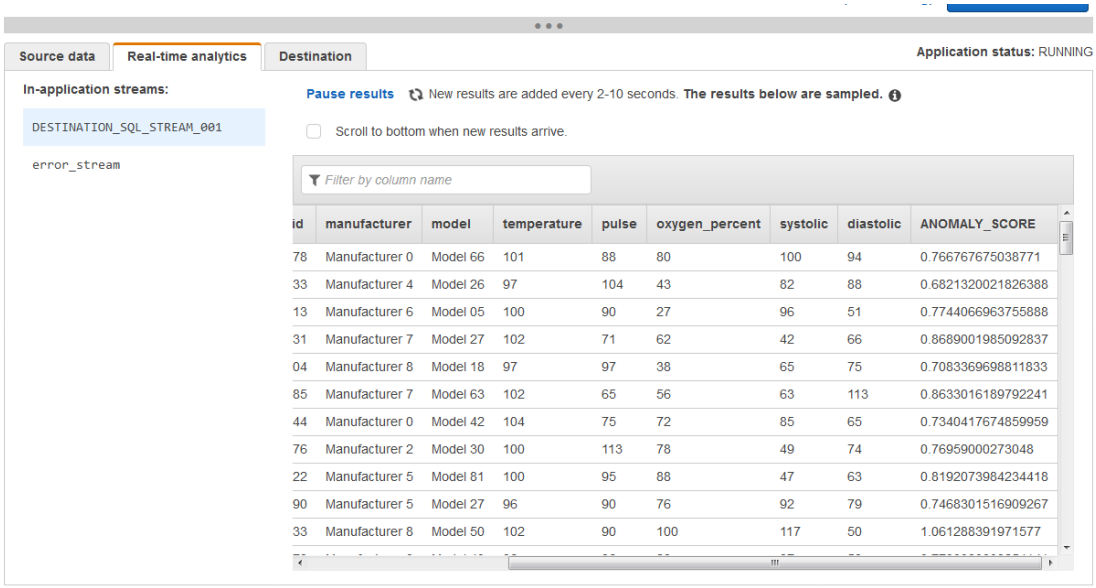
To test this pattern you will use the Kinesis Data Generator (KDG) to Tool to generate and publish test data. Refer to section **Create an Amazon Cognito User for Kinesis Data Generator** section of the whitepaper. Using the username and password that you generated during the configuration log into the KDG tool. Provide the following information:

1. Region : Select the Region where you have installed the **DeviceDataRealTimeAnalyticsStack**
2. Stream / Delivery Stream: Select the delivery stream called **DeviceDataInputDeliveryStream**
3. Records per Second: Enter the record generation / submission rate for simulating the hospital device data
4. Record template: KDG uses a record template to generate random data for each of the record fields. We will be using the following JSON template to generate the records that will be submitted to the Kinesis Delivery Stream - **DeviceDataInputDeliveryStream**

```
{
  "timestamp" : "{{date.now("x")}}",
  "device_id" :
"device{{helpers.replaceSymbolWithNumber("####")}}",
  "patient_id" :
"patient{{helpers.replaceSymbolWithNumber("####")}}",
  "temperature" : "{{random.number({"min":96,"max":104})}}",
  "pulse" : "{{random.number({"min":60,"max":120})}}",
  "oxygen_percent" : "{{random.number(100)}}",
  "systolic" : "{{random.number({"min":40,"max":120})}}",
  "diastolic" : "{{random.number({"min":40,"max":120})}}",
  "manufacturer" : "Manufacturer
{{helpers.replaceSymbolWithNumber("#")}}",
  "model" : "Model {{helpers.replaceSymbolWithNumber("##")}}"
}
```



To run the RealTime Analytics application click on the **Send Data** button located towards the bottom of the KDG Tool. As the KDG begins to pump device data records to the Kinesis Delivery Stream, the records are streamed into the Kinesis Analytics Application. The Application code analyzes the streaming data and applies the algorithm to generate the anomaly score for each of the rows. You can view the data stream in the Kinesis Analytics console. The diagram below the sampling of the data stream.



The Kinesis Analytics Application is configured with two Destination Connections. The first destination connector (or output) is a Lambda function. The lambda function iterates through a batch of records delivered by the Application DESTINATION_SQL_STREAM_001 and interrogates the anomaly score field for the record. If the anomaly score exceeds the threshold defined in the lambda function environment variable **ANOMALY_THRESHOLD_SCORE**, the lambda function publishes the record to a Simple Notification Service (SNS) Topic named - **publishtomanufacturer**

The second Destination Connection is configured to a Kinesis Firehose Delivery Stream – **DeviceDataOutputDeliveryStream.** The delivery stream buffers the records, and zips the buffered records to a zip file before putting into the S3 bucket - **S3ProcessedDataOutputBucket**

Observe the followings:

1. In your e-mail (that you specified in the deployment script) inbox the first e-mail you will receive device data records for which the anomaly score has exceeded the specified threshold

2. In the AWS Kinesis Data Analytics console select the Application named **DeviceDataAnalytics** click the Application detail button towards the bottom, this will take you to the **DeviceDataAnalytics** application detail page. Towards the middle of the page under Real-Time Analytics click the button “Go to the SQL Results”. On the real-Time Analytics page observe the Source Data, Rael-Time Analytics Data and the Destination Data using the tabs.

3. Records with the anomaly score are stored in the **S3 Processed Data Bucket.** Review the records that includes the anomaly score.

To stop the Testing of the pattern simply go to the browser where you are running the KDG Tool and click the “Stop Sending Data to Kinesis” button.

Cleaning up Resources

Once you have tested this pattern, you can delete and clean up the resources created so that you are not charged for these resources.

1. Go back to the browser where you launched the KDG Tool and click the stop button. The tool will stop sending any addition data to the input kinesis stream.

7. On the S3 console, choose the output **S3 Processed Data Bucket** and choose **Empty Bucket**.
8. On the Kinesis console stop the Kinesis Data Application - **DeviceDataAnalytics**
9. On the CloudFormation console, choose the **SensorDataCollectionStack** stack and click **Delete Stack**.
10. Finally on the EC2 console terminate the ec2 Linux instance you created to run the CloudFormation template to deploy the solution