Paper 3725-2019

Embracing the open API ecosystem to give analytics an organizational operational landing spot

Olivier Thierie, Wouter Travers, and Andrew Pease, Deloitte Belgium.

ABSTRACT

Organizations often struggle to make strategic insights actionable. Powerful analytics, available at key operational decision points, can make the difference. An example of such a case was to determine occupancy of key docks at a major international seaport, using object detection techniques on the camera feeds. In order to leverage such strategic insights throughout the organization, a major hurdle is to deploy analytics in the key operational systems.

With Viya, SAS® has embraced the open source community with open source analytics integration and the open API ecosystem. This paper will demonstrate how to leverage CAS for image recognition from within Python and how-to set-up a kappa architecture using Kafka to publish results in real-time to organizational systems like SAP® and Salesforce®. The focus in the paper is on the integration of open source and SAS® analytics in these two systems, rather than on the specific use case.

That said, we begin with a short description of the business use case.

INTRODUCTION

Increasingly global naval ports reach full capacity with limited or prohibitively costly expansion possibilities. Preparing an efficient planning of berth allocation is not an easy task for the port authority. Many factors need to be considered, such as the number of available free anchoring berths and the schedules of the seagoing vessels and barges. Detailed and real-time insight into this information offers opportunities for optimizing quay utilization, reducing transshipment time, maximizing crane usage and efficiently transporting and storing containers. This leveraging of camera feeds to increase dock utilization within a berth served as an ideal use case for demonstrating how SAS® Viya can be leveraged as the main analytics backbone within a Kappa, open API architecture[1].

Apache Kafka is an open source, distributed streaming platform that is designed for a Kappa architecture. Apache Kafka focuses on scalable, real-time, processing of data [1]. Further in the paper, we will discuss our rationale for choosing Kafka, instead of SAS®' own Event Stream Processing engine. Software vendors are increasingly integrating Kafka in their framework architecture by providing open source connectors.

The established operational systems we chose to integrate were SAP® and Salesforce®. In the context of SAP®, we chose to stream data to HANA® making use of their Kafka-Connect-SAP framework. As Salesforce® does not have such a connector yet, we wrote our own solution to stream messages to Salesforce. To make our solution more visible, we also

[1] Kappa architecture: Lambda architecture consolidated into a single computation model that handles batch processing as a lower frequency stream.

provided a Python Flask web application that streams consumed images immediately to the web application.

Our experience indicates that Kafka is a sound choice for integrating different software within the company architecture, while handling high volumes and real time streaming.

METHODOLOGY – ARCHITECTURE

Our solution blends various cloud technologies. The process starts with images from a camera feed. The SAS® Viya server picks them up and passes them to the CAS scoring code. After scoring the image, bounding boxes are retrieved, KPI's get calculated, they are put into an AVRO message and passed to the Kafka producer that produces to our Kafka topic. Three different consumer groups are listening to the topic and grasp asynchronously the messages which were stored on the Kafka topic. Within this set-up, we decided to go for a Confluent Kafka platform to leverage the Avro schema registry capabilities and the usage of open source connect apps. Every message is also stored on our Kafka Cluster for a limited period of time. At the moment of consumption, the full message is pushed to SAP HANA®, KPI Fields are pushed to Salesforce® and so we have a flask app running that is able to stream the pictures in real time. The pictures are shown a single time and when nothing is fed to the flask app, a default picture is shown. In the following section, we review the core components of our solution. Figure 1 visualizes the entire, cloud-based solution. The code of our solution can be found here [2].

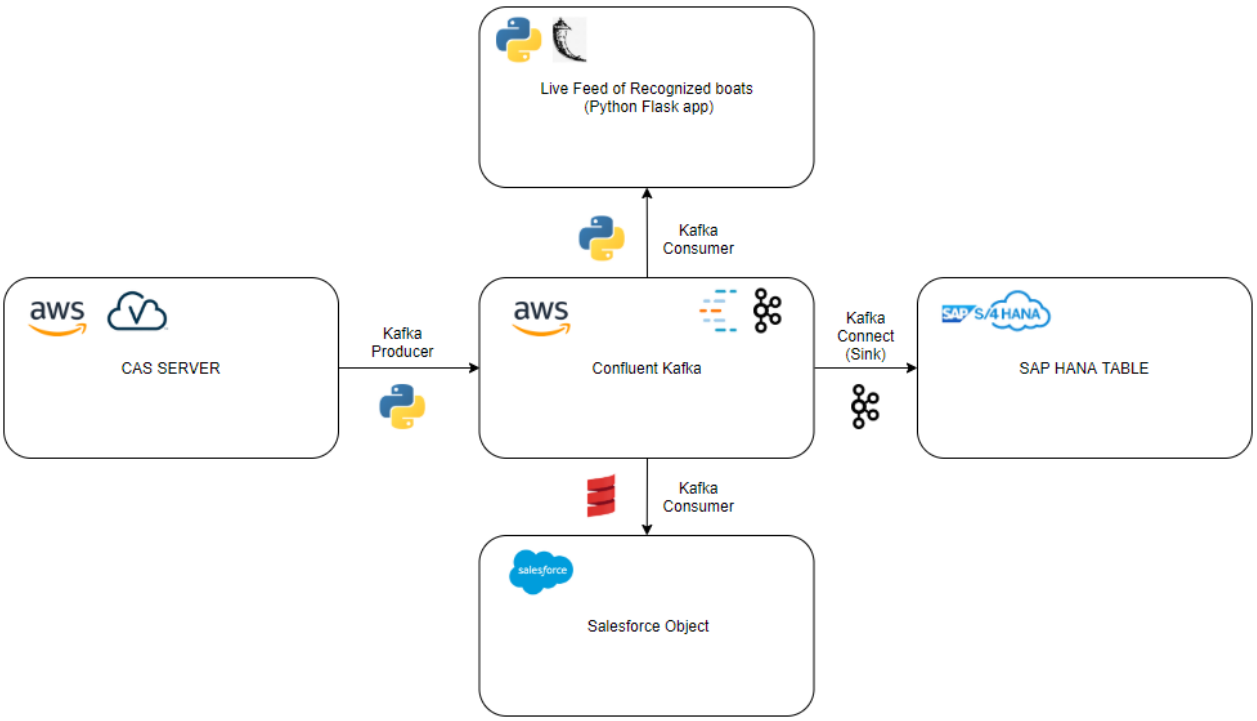

Figure 1: Architectural overview

CONFLUENT KAFKA ON AWS®

The Kafka cluster is the core component of our solution. We make use of the open-source framework of Confluent Kafka, deployed on the managed cloud infrastructure of AWS®.

We prefer the open-source version of Confluent® Kafka (5.0.0) because it contains some tools that make Apache Kafka easier to use. In particular, the Avro Schema Registry and the python client are two vital components for our solution. Furthermore, Confluent® provides a deployment guide on AWS®, which makes configuration and installation of Confluent® Kafka easier, scalable and manageable. In the following paragraphs, we will discuss the setup of our PoC (Proof of Concept) cluster, our current configuration, and finally we explain why we decided to make use of Kafka instead of SAS Event Stream Processing®.

DEPLOYING CONFLUENT KAFKA ON AWS

The Confluent platform is a streaming platform for large-scale distributed environments, built on Apache Kafka. The Platform enables you to connect interfaces and data systems, so you can leverage upon integrated, real time information [3].Confluent provides a step by step guide to deploy a proper virtual private cloud (VPC) on AWS®. The Initial set-up provides a VPC as shown in Figure 2.

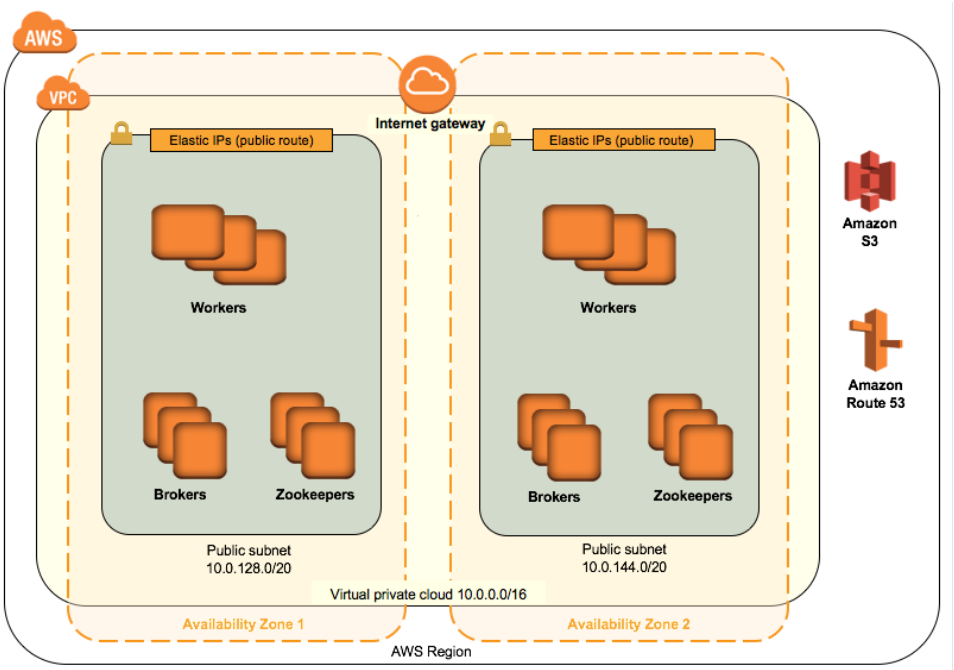

Figure 2: Default Confluent Kafka cluster set up

MINIMIZING COSTS IN A POC SETUP

As we built our Proof-of-Concept, we did not choose the plain vanilla setup[2]. We scaled down the VPC deployment where possible and to settle on a single m4.large machine which functions concurrently as the Worker, Broker and Zookeeper node. This set-up would not be recommended in a production environment and was only done to minimize costs for the PoC. Also, in order to have no firewall issues on the AWS® side, we allowed all traffic by configuring the Network Access Control List (ACL) and the security groups applied on the EC2 instance. This obviously is also not a production best practice.

SERVICES USED IN THE POC CLUSTER

Confluent provides a straightforward way to manage your Kafka services by using the confluent command. In our case, we decided to deploy our services manually in separate screens:

```
[centos@ip-xx-x-xxx-xxx ~]$ screen -list
There are screens on:
        5759.30716.HanaLink     (Detached)
        31870.hannaConnector    (Detached)
        24291.schema-registry   (Detached)
        12586.broker    (Detached)
        12293.zookeeper (Detached)
```

This allows us to have more control about used properties, accessibility and more visible error handling. In the subsequent subsections, we will give you an overview of different services. The HanaLink and HanaConnector will also be discussed further in this paper.

Zookeeper

Apache ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All these kinds of services are used in some form or another by distributed application [4]. In context of Kafka, ZooKeeper is used to store persistent cluster metadata. In a production environment it is recommended to run an ensemble of at least 3 zookeeper servers [5]. We started by invoking one Zookeeper service within a screen. As Zookeeper is only used by Kafka internally, we did not change anything to the property configuration.

Kafka Broker

A Kafka Cluster exists out several Kafka brokers. The brokers contain the topic log partitions. The required set-up with at least three to five different brokers is considerable task [6]. In our PoC setup, we deployed only one Kafka broker in a screen. As we required access to our broker externally, we changed the listeners' properties with the advertised.listeners [1]:

```
advertised.listeners=PLAINTEXT://ec2-XX-XXX-XXX-XXX.us-west-
2.compute.amazonaws.com:9092
```

The used address is the public DNS of our EC2 instance. If we would have used the default option, the external client will not be able to find the brokers. A more production-suitable set-up would consist of multiple machines and brokers.

[2] The setup suggested by the deployment guide of Confluent Platform on AWS

Avro Messages

Data flows as AVRO messages through our cluster. Apache AVRO is a data serialization system that is compact, fast and is a binary data format. A written AVRO message comes with a schema. This ensures a readability of an AVRO message when it is read or stored. AVRO Schemas are defined in JSON [7]. As we are using Confluent Kafka, it is convenient to manage your schemas. The schema registry allows schema's to evolve over time while ensuring compatibility. Our AVRO schema has the following configuration, the configuration itself will be discussed in a subsequent section:

```
deloitte_kafka_schema = avro.loads("""
{"namespace": "be.deloitte.kafka",
  "type": "record",
  "name": "Image",
  "fields": [
      {"name": "imageId" , "type": "string"},
      {"name": "timestamp", "type": { "type": "long", "logicalType":
"timestamp-millis" }},
      {"name": "numOfBoats", "type": "int"},
      {"name": "occupancyRate", "type": "double"},
      {"name": "image", "type" : "string"}
  ]
 }
""")
```

Schema Registry Service

Data is produced by a Kafka producer towards a Kafka topic and a Kafka consumer will read data from a topic. Data written by producers should be readable by consumers. These schemas can evolve over time. The Schema Registry provides centralized schema management ensures the compatibility [8].

The Schema registry is a distributed storage layer for Avro Schemas, by using the schema registry, you have access to the versioned history of your schemas. It provides a plugin to clients that handles schema storage and retrieval for messages that are sent in Avro format. The schema registry is often used in Kafka connect (e.g. kafka-connect-sap). The Schema Registry comes with a REST API that can be used to manage your schemas. The schema registry was also deployed in a screen.

Topics, consumer groups and retention policy

Our AVRO message is produced towards the "deloitteKafka" topic, again as we are in proof of concept environment, we did not allow for replication and partitioning. Also, we adapted the storage time of approximately 3.5 hours. The commands that ensure the stated facts can be found below:

```
bin/kafka-topics --zookeeper localhost:2181 --create --replication-factor 1
--partitions 1 --topic deloitteKafka
bin/kafka-configs --zookeeper localhost:2181  --entity-type topics --
entity-name deloitteKafka --alter --add-config retention.ms=12800000
```

KAFKA VS SAS EVENT STREAM PROCESSING®

The goal of this paper was demonstrating the integration of SAS® Viya within a broader corporate IT-infrastructure. In this context, we decided to integrate exported results coming from a SAS® Viya environment with other big vendor tools such as SAP® and Salesforce®. In this way, the whole landscape can benefit from SAS® Viya's capabilities. Since we already had experience with Kafka, and the focus is on integration, Kafka was chosen as the tool to stream the processed predictions towards other systems. Kafka allows proprietary vendors to write their own Kafka Connectors. Otherwise, because of the scala and python api's, it is also relatively simple to write your own solution when such an out-of-the-box solution is not available.

CONFLUENT® KAFKA LIBRARIES

As mentioned Confluent Kafka provides both Scala and Python APIs. This provides much flexibility towards the developer and enforces creative solutions. In context of this PoC and resulting paper, we wrote a simple framework focused on this particular use case. The Scala framework is able to consume messages and pass them to Salesforce's® REST API. The Python framework is also able to produce, process and consume the messages. Both Frameworks contain an object that handles all configuration related to Kafka. This configuration is then passed to the particular object that produces, consumes and/or processes data.

SAS VIYA ON AWS

DEPLOYING SAS® VIYA ON AWS®

In order to deploy SAS® Viya on AWS®, we followed the quick-start guide that is available on Amazon Web Services [9]. Again, all security settings were minimized to ensure simple accessibility.

VESSEL DETECTION IN SAS® VIYA

In this use case, we are making use of computer vision to determine dock utilization. Dock utilization is a key metric for all international ports. Effective measuring allows detailed and real-time follow-up. Camera feeds can be used to automatically detect vessels and their exact location on the dock, allowing this information to be made available to different parties more quickly.

Object detection

Automatically locating objects within a camera feed of a dock can be done using object detection techniques. In general, there are two different categories of techniques: One stage and two stage object detection. While two stage should have a higher location

accuracy it's also a slower process. As real-time object detection was required only one stage object detection algorithms where considered. These algorithms look at a predefined finite set of image windows and thus do not require to do region proposal first. The selection was made to use the YoloV2 algorithm [10].

The YoloV2 deep learning architecture is predefined in SAS® Viya and can be leveraged using the SAS® DLPy python package. In this way, we leverage on the CAS deep learning actions. We use 8287 labelled images, based on the camera feed.

The code required to train the model in SAS® will be briefly summarized below. Please refer to the SAS® GitHub [11]–[13] for more examples and documentation.

First, we ingest the labeled images. SAS® Viya includes action sets to translate the labelled images between various coordinate types. The images were labelled using the VOC Pascal coordinates

```
#uploading the 8000+ images and corresponding label file to a sas table
#labelling was done using the yolo coordinate type
object_detection_targets = create_object_detection_table(s,
data_path = '/opt/sasinside/DemoData/data',
    local_path = '/opt/sasinside/DemoData/data',
    coord_type = 'yolo',
    output = 'detTbl')
```

Next, we define the anchors. In this case we utilized the get_anchors function, provided by SAS. This is using k-means to find your initial set:

```
#K-means procedure to find initial anchors
anchors=get_anchors(s,data='detTbl',n_anchors=5, coord_type='yolo')
```

The YoloV2 architecture is predefined in SAS®, and the full architecture can be created with the YoloV2 function in a single step:

```
#set up YoloV2 architecture
model = Yolov2(
            conn=s,
            randomMutation = 'none',
            actx = 'leaky',
            coordType='yolo',
            n_classes=11,
            predictionsPerGrid=5,
            width=416,
            height=416,
            randomBoxes = False,
            softMaxForClassProb=True,
            matchAnchorSize=False,
            numToForceCoord=-1,
            rescore=True,
            classScale = 1.0,
            coordScale=1.0,
            predictionNotAObjectScale = 1,
            objectScale=5,
            detectionThreshold=0.2,
            iouThreshold = 0.1,
            act = 'LOGISTIC',
            anchors = anchors
            )
```

Optimization settings and the training are done using the following syntax:

```
optimizer=dict(miniBatchSize=1,logLevel=3,
               maxEpochs=10, regL2=0.0005,
               algorithm=dict(method='momentum', momentum=0.9,
               clipGradMax=100, clipGradMin=-100,
               learningRate=0.0001
               ))
r=model.fit(data='detTbl',
            optimizer=optimizer,
            forceEqualPadding = True,
            # specify data type of input and output layers
            dataspecs=[
            dict(type='IMAGE', layer='Data', data=inputVars),
            dict(type='OBJECTDETECTION', layer='Detect1', data=targets)],
            nthreads=8)
```

By training the YoloV2 algorithm on more than 8000 labeled images captured from historical camera feeds, the algorithm is capable of accurately detecting vessels in more recent camera feeds with high accuracy. The weights and the model are afterwards saved to a sashdat file. These files can be used to load the trained model for scoring in other sessions.

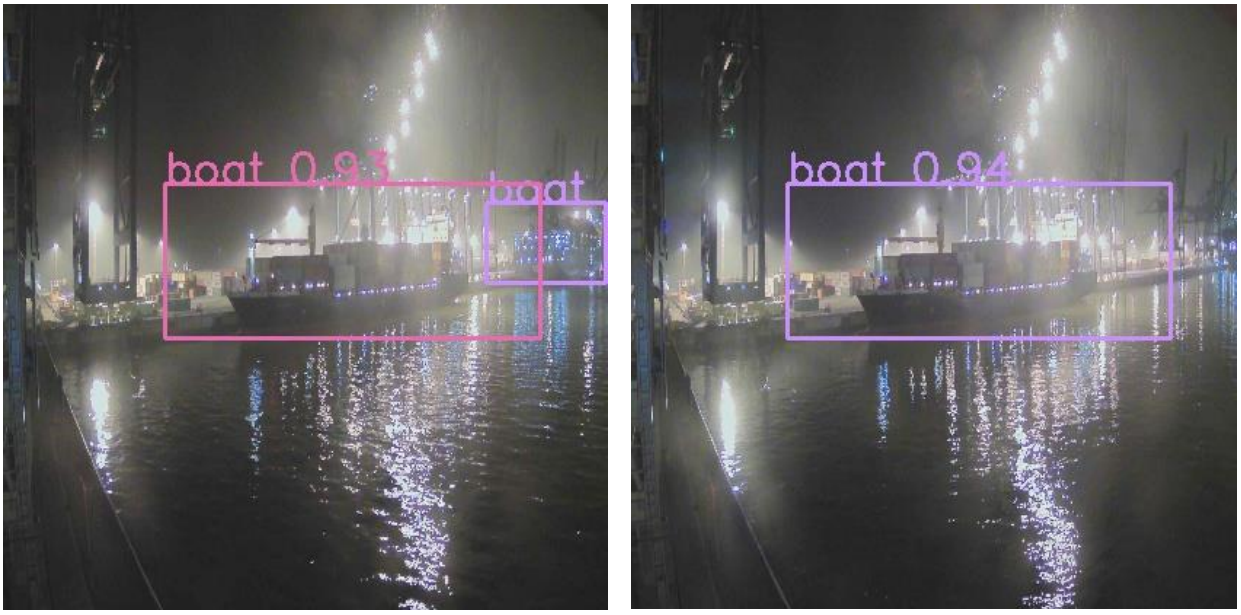

Figure 3: Object detection results on test data

Scoring and producing the results to Kafka

After training the YOLOV2, the model components (Yolov2.sashdat, Yolov2_weights_attr.sashdat and Yolov2_weights.sashdat) were stored on the AWS® Viya® server.

Within python, we defined the following functions to handle the scoring:

```
def viyaStartUp():
    s= CAS('xx.xxx.xxx.xxx',5570,'username','password')
    s.loadactionset('image')
    s.loadactionset('deepLearn')
    model_load = Model(s)
    model_file = '/root/fullModel/Yolov2.sashdat'
    model_load.load(path=model_file)
    test = ImageTable.load_files(conn=s, caslib='dnfs',
path='/data/xxx/validation')
    test.resize(height=416, width=416, inplace=True)
    prd = model_load.predict(data=test)
    return [s,prd]

def getTableName(predict_model_object):
    return
CASTable(predict_model_object.get('OutputCasTables').Name[0],coord_type='yolo
')
def kafkaStartUp():
    return  ProducerApp(kafkaparameters=  KafkaParameters(),
avroSchema=deloitte_kafka_schema, topic="kafka")

def produce_object_detections(conn, table, coord_type, producerApp):
    '''
    Plot images with drawn bounding boxes.
    conn : CAS
        CAS connection object
    table : string or CASTable
        Specifies the object detection castable to be plotted.
    coord_type : string
        Specifies coordinate type of input table
    max_objects : int, optional
        Specifies the maximum number of bounding boxes to be plotted on an
image.
        Default: 10
    num_plot : int, optional
        Specifies the name of the castable.
    n_col : int, optional
        Specifies the number of column to plot.
        Default: 2
    fig_size : int, optional
        Specifies the size of figure.
    '''
    conn.retrieve('loadactionset', _messagelevel = 'error', actionset =
'image')
    input_tbl_opts = input_table_check(table)
    input_table = conn.CASTable(**input_tbl_opts)
    det_label_image_table = random_name('detLabelImageTable')
    num_max_obj = input_table['_nObjects_'].max()

    with sw.option_context(print_messages=False):
```

```
        res = conn.image.extractdetectedobjects(casout = {'name':
det_label_image_table, 'replace': True},
                                                coordtype=coord_type,
                                                maxobjects=num_max_obj,
                                                table=input_table)
        if res.severity > 0:
            for msg in res.messages:
                print(msg)
    outtable = conn.CASTable( det_label_image_table)
    #imageRecordList = list()
    in_df = input_table.fetch()['Fetch']
    out_df = outtable.fetch()['Fetch']
    if len(out_df) == len(in_df):
        print(str(len(out_df)) + " equal table length assumption is met,
producing message buffer")
        for i in range(len(out_df)):
            imageId = str(uuid4())
            t = datetime.now()
            timestamp = round((t-datetime(1970,1,1)).total_seconds())
            nbrOfBoats = int(in_df['_nObjects_'][i])
            imgStr = out_df['_image_'][i]
            nparr = np.frombuffer(imgStr, np.uint8)
            #img_np = cv2.imdecode(nparr, cv2.IMREAD_COLOR)
            base_img = str(base64.b64encode(nparr))
            occupancy_rate = 0
            if nbrOfBoats > 0:
                surface_list = list()
                index = 5
                for ix in range(nbrOfBoats):

surface_list.append(in_df.iloc[i,index+4]*in_df.iloc[i,index+5])
                    index = index + 6
                occupancy_rate = round(sum(surface_list),4)


#imageRecordList.append(ImageRecord(imageId,timestamp,nbrOfBoats,occupancy_ra
te,base_img))
            producerApp.produce(preparedMessageArray=
[ImageRecord(imageId,timestamp,nbrOfBoats,occupancy_rate,base_img)])

    with sw.option_context(print_messages=False):
        conn.table.droptable(det_label_image_table)
 from viya_utils import viyaStartUp, kafkaStartUp, produce_object_detections,
getTableName

def main():
    viya_params = viyaStartUp()
    producer = kafkaStartUp()
    produce_object_detections(viya_params[0],table=
getTableName(viya_params[1]),coord_type='yolo',producerApp=producer)

if __name__ == '__main__':
    main()
```

The main() method will connect to AWS® Viya®, load our model and predict the locations of Vessels in the picture. Then we return the connection and predictions for later use. The getTableName() method returns the reference of the CASTable that contains the predictions. KafkaStartUp() will start a configured producer. Produce_object_detections is based on the dlpy function display_object_detections [13]. We retrieve the detected objects as a CASTable and convert the CASTable with predictions as well as the one with detected objects to a dataframe. Then we iterate over them in order to get the number of boats, the occupancy rate, which is the surface taken by all bounding boxes on the picture and the image as base64 string. These are put into an ImageRecord and produced to our Kafka cluster.

So far, the focus is on streaming an image, together with kpi's and metadata over Kafka. Images get scored using SAS® Viya's CAS server in combination with python's dlpy library. We then retrieve the estimated bounding box coordinates, the number of recognized objects (boats) and we approximate the occupancy rate. These KPIs are then put with the image, a timestamp and a unique identifier (metadata) into an AVRO message. The message is then produced and pushed towards the Kafka cluster.

In a production environment, it would be advised to send the pictures on a separate topic solely as a byte array. We first thought of defining the image as a byte array within the AVRO schema, but ran into conversion issues when creating a JSON file out of our AVRO message. In order to cope with this, we convert the image into her base64-string representation. This definitely has limitations but was sufficient of proving the purpose of this paper.

KAFKA TO SAP HANA®

GENERAL OVERVIEW

In order to link Kafka to SAP®, we make use of the open source Apache Kafka connect framework. This framework ensures connectivity towards external systems such as databases, file systems, key-value stores and search indexes [14]. We use a sink connector to ingest data from a Kafka topic towards an external system. A source connector will stream data from an external system towards a Kafka topic.

SAP® provides an out-of-the-box connector towards HANA®. The source code is available on GitHub [15]. By building the jars and providing the correct configuration (see below), we can successfully ingest data into Hana. With only a trial account available on the SAP® Hana cloud platform [16], we required some extra steps to make the connection work.

PRACTICAL SETUP

We create a Hana table within the trial version of SAP® cloud platform [16] and ensure that the trial instance is up and running. In order to connect and modify the Hana instance, we use the Hana "on-demand" tools, more specifically, the Java EE7 web profile SDK [17]:

```
Sh neo-javaee7-wp-sdk-1.40.15/tools/neo.sh open-db-tunnel
-h hanatrial.ondemand.com
-a <trialaccount>
-u <username>
-i <hanaTable>
-p <password>
```

Which results in opening a db tunnel and a usable JDBC URL:

```
Opening tunnel...

Tunnel opened.

Use these properties to connect to your schema:
  Host name        : localhost
  Database type    : HANAMDC
  JDBC Url         : jdbc:sap://localhost:30015/
  Instance number  : 00


Use any valid database user for the tunnel.
This tunnel will close automatically in 24 hours on 10-mrt-2019 at 13:44 or
when you close the shell.
Press ENTER to close the tunnel now.
```

This command is executed in a screen on our AWS Kafka cluster, and so we ensure the connection for a 24 hour period.

In order to make SAP's® connector work, clone the current version from GitHub and build the project. Subsequently, create the directory confluent-5.0.0/share/java/kafka-connect-sap and move the kafka-connect-hana.jar there. The SAP® HANA JDBC driver (jar) is also required. Download the ngdbc-2.3.55.jar from the internet [18] and copy the jar into the same directory. Move the configuration (sink.properties) to confluent-5.0.0/etc/kafka-connect-sap/.

The used properties can be examined below:

```
name=hana-sink
connector.class=com.sap.kafka.connect.sink.hana.HANASinkConnector
tasks.max=1
topics=deloitteKafka

connection.url=jdbc:sap://localhost:30015/
connection.user=SYSTEM
connection.password=<password>
auto.create=false
schema.registry.url=http://XX. XXX.XXX. XXX:8080
deloitteKafka.table.name="SYSTEM"."KAFKA_HANA_TABLE"
```

The name hana-sink is used as the consumer group. We did not use the option to auto create the table. Normally, this is a sufficient option as the table is then based on the schema that is given by the schema registry:

```
CREATE COLUMN TABLE "SYSTEM"."DUMMY_TABLE" ("imageId" VARCHAR(1000) NOT
NULL, "timestamp" TIMESTAMP NOT NULL, "numOfBoats" INTEGER NOT NULL,
"occupancyRate" DOUBLE NOT NULL, "image" VARCHAR(1000) NOT NULL)
```

Although this looks convenient for auto creation, this can also cause issues. The base64 string image will often create a VARCHAR that exceeds the 1000 characters, In order to solve this, we created the table manually by using the pyhdb API [19] in Python:

```
import pyhdb
connection = pyhdb.connect(
    host="localhost",
    port=30015,
    user="SYSTEM",
    password=<password>
)
cursor = connection.cursor()
cursor.execute('CREATE COLUMN TABLE "SYSTEM"."KAFKA_HANA_TABLE" ("imageId"
VARCHAR(1000) , "timestamp" TIMESTAMP , "numOfBoats" INTEGER,
"occupancyRate" DOUBLE, "image" VARCHAR(5000) )')

cursor.execute("SELECT SCHEMA_NAME, TABLE_NAME FROM TABLES WHERE
SCHEMA_NAME = 'SYSTEM' AND TABLE_NAME = 'KAFKA_HANA_TABLE'")
cursor.fetchall()
[('SYSTEM', 'KAFKA_HANA_TABLE')]
```

In this way, we are able to ingest some of the images, when the base64 string is not exceeding the limit of 5000 characters. This setup ensures that the connector will still ingest the record and pass the image as “None” when the base64 string is exceeding the character length. The connector is launched in a screen (in production, this should be running on the worker node and preferably be distributed):

```
bin/connect-standalone etc/schema-registry/connect-avro-
standalone.properties etc/kafka-connect-sap/kafka-connect-sink.properties
```

In context of this proof of concept, this demonstrates streaming data from Kafka into a SAP® table. In a production environment, we would suggest creating a separate producer and schema for this case. When producing image-records by using the Python pyhdb API, we are able to see ingested records:

```
cursor.execute("SELECT * FROM SYSTEM.KAFKA_HANA_TABLE ")
cursor.fetchall()
[('7fd41f50-dbd8-49d6-be8a-2a9c644ee511',
  datetime.datetime(1970, 1, 1, 0, 0, 23),
  1,
  0.5,
  None),…

cursor.execute("SELECT count(*) FROM SYSTEM.KAFKA_HANA_TABLE ")
cursor.fetchall()
[(18,)]
```

KAFKA TO SALESFORCE®

In order to make use of Salesforce®, we created an account on the EU lightning cloud platform® [20]. The demo environment is fairly complete and we are able to make proper use of the Salesforce components. Within Salesforce®, we created a Salesforce® object (comparable to a table) and a Salesforce® app so we are able to connect to our Salesforce® instance.

As there is currently no connector available to synch data from Kafka to Salesforce®, we wrote our own solution in Scala. Our program can consume messages from Kafka, process/filter the data and pass the fields we are interested in to salesforce by using the Salesforce REST API. We essentially wrote a tiny framework, based on the org.apache.http library [21], that is able to authenticate and handle the different requests in context of our use case. The SalesforceConsumer() will consume the messages and convert them from a JSON to a case class that only contains the relevant fields:

```
def salesForceConsumer(): Unit = {
    val consumer = new KafkaConsumer[String, GenericRecord](props)
    //subscribe to producer's topic

consumer.subscribe(util.Arrays.asList(KafkaParameters.DELOITTE_KAFKA_TOPIC)
)
    while(true) {
      val records: ConsumerRecords[String, GenericRecord] =
consumer.poll(2000)
      println(s"message count: ${records.count()}")
      //print each received record
      import scala.collection.JavaConversions._
      for (record <- records.iterator()) {
        //println(s"Here's your ${record.value()}")
        import DeloitteKafkaProtocol._
        val jsonTransaction =
DeloitteKafka(record.value.get("imageId").toString,
record.value.get("timestamp").asInstanceOf[Long],
record.value.get("numOfBoats").asInstanceOf[Int],
record.value.get("occupancyRate").asInstanceOf[Double]).toJson.toString()
        OathConnectAPI.postRequest("services/data/v44.0/sobjects/",
"Kafka_Salesforce__c", jsonTransaction, oathCreds)

      }
      //commit offsets on last poll
      consumer.commitSync()
      counter = counter + 1
      println(counter)
    }  }
```

For every (successful) POST request, we log the timestamp and unique salesforce ID, so we can find the records back by using the REST API in for example Postman [22]. Later on, we created a view in Salesforce® we can see all the ingested records (figure 4). Currently the timestamp is in Unix epoch. In a production context, it would be advised to convert the timestamp into a real date.

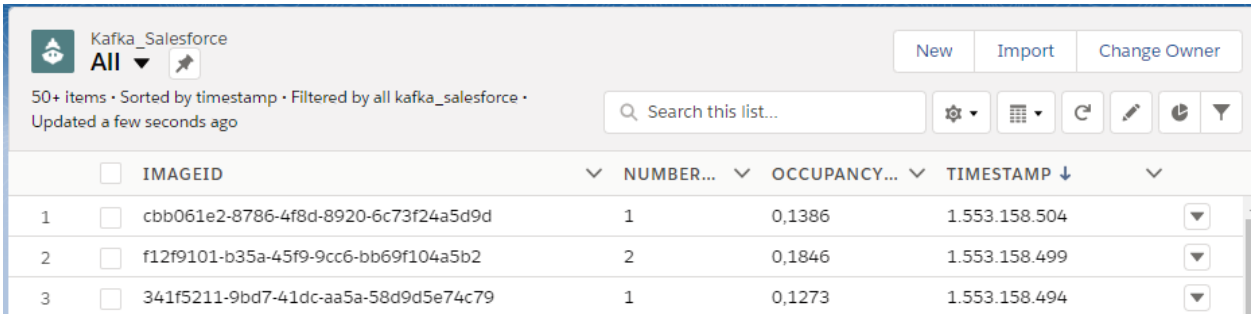

Figure 4. Screenshot of the Object (table) in salesforce

KAFKA TO WEB-APPLICATION (FLASK)

It is relevant to show our boat detection in real time. In this way, real time monitoring can occur upon the current status of the berth. For this, we use a Python flask app that is picking up our messages and serves them real time to a Flask app. The architecture of this solution can be examined in figure 5.

Flask is a micro framework for making web applications in python. We base us on the flask-video app [23] as a start and extended the framework to make a streaming like approach possible. The consumer app consumes the AVRO messages from Kafka, extracts the base64 string out of the message, converts it back to an image and stores the image in the shared directory. At the same moment, an instance of our "Directory Handler" will scan the directory, pick up the images and pop them into a queue. When an image is read from the queue and passed to the Flask app, the image is removed from the directory. In this way, we ensure that a streamed picture is only showed once and that the image is deleted after passing it.

The current setup can still be improved. For example, images are currently picked up randomly, this means that the order of showing is determined by the order of reading their paths. This could be improved by taking the modification date into account or revising the filename to ensure a chronological pick up. Nevertheless, we prove the feasibility of ingesting images and show them after consumption on a web application. The code can be examined here [2].

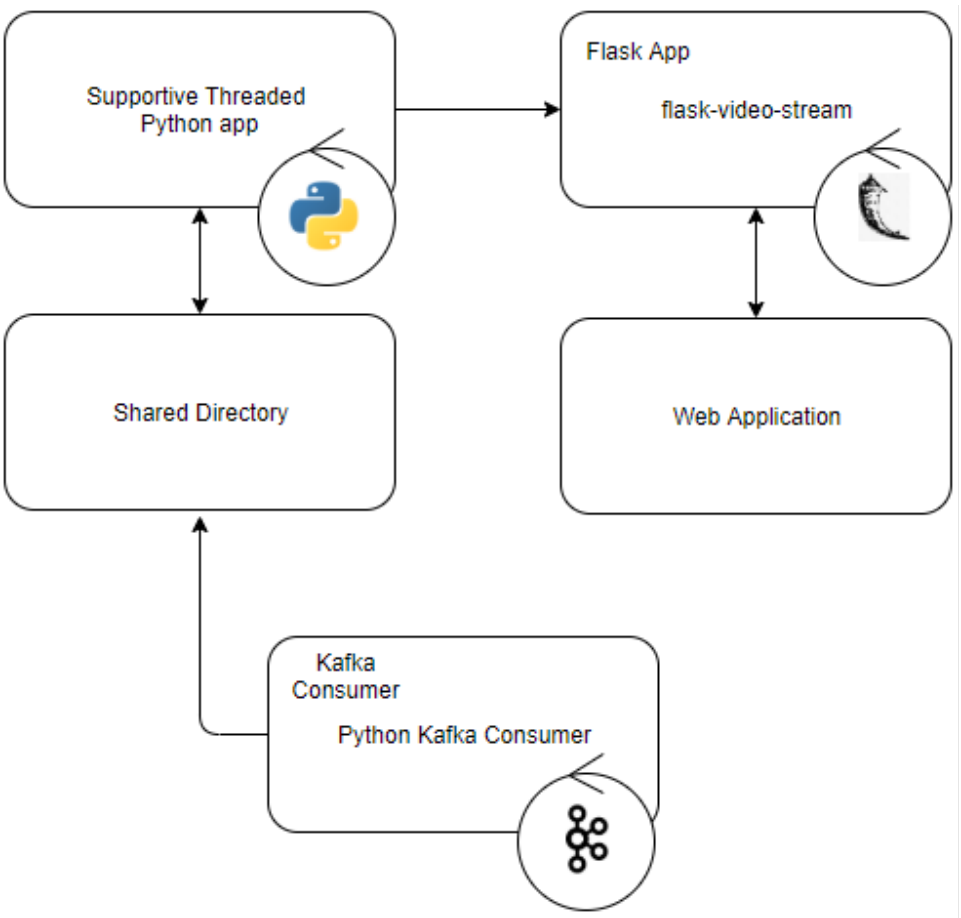

Figure 5: Architecture of the Flask web application

CONCLUSION

This paper proves the feasibility of using Kafka for integrating multiple vendor applications into the corporate IT landscape. It proves that SAS Viya® is an excellent backend for image recognition and other advanced analytics, while other applications can benefit from the gained insights. We also demonstrated the feasibility of configuring connectors or finding workarounds if they are not available. Kafka is a reasonable choice for integrating different applications, while handling high volumes and real time streaming.

REFERENCES

[1] "Apache Kafka," *Apache Kafka*. [Online]. Available: https://kafka.apache.org/documentation/#zk. [Accessed: 03-Mar-2019].

[2] othierie, *Supporting code of Embracing the open API ecosystem to give analytics an organizational operational landing spot SAS Paper 3725-2019, https://github.com/othierie/SGF-Viya-Streaming-Integration*. 2019.

[3] "Confluent Platform on AWS - Quick Start," *Amazon Web Services, Inc.* [Online]. Available: https://aws.amazon.com/quickstart/architecture/confluent-platform/. [Accessed: 03-Mar-2019].

[4] "Apache ZooKeeper." [Online]. Available: https://zookeeper.apache.org/. [Accessed: 03-Mar-2019].

[5] "Running ZooKeeper in Production — Confluent Platform." [Online]. Available: https://docs.confluent.io/current/zookeeper/deployment.html. [Accessed: 03-Mar-2019].

[6] "Kafka Architecture." [Online]. Available: /blog/kafka-architecture/index.html/. [Accessed: 03-Mar-2019].

[7] "Apache Avro™ 1.8.2 Documentation." [Online]. Available: http://avro.apache.org/docs/current/. [Accessed: 08-Mar-2019].

[8] "Schema Registry — Confluent Platform." [Online]. Available: https://docs.confluent.io/current/schema-registry/docs/index.html. [Accessed: 03-Mar-2019].

[9] "SAS Viya on AWS - Quick Start," *Amazon Web Services, Inc.* [Online]. Available: https://aws.amazon.com/quickstart/architecture/sas-viya/. [Accessed: 09-Mar-2019].

[10] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," *ArXiv161208242 Cs*, Dec. 2016.

[11] "SAS Software," *GitHub*. [Online]. Available: https://github.com/sassoftware. [Accessed: 21-Mar-2019].

[12] "SAS Deep Learning Python Interface — DLPy 1.0.1 documentation." [Online]. Available: https://sassoftware.github.io/python-dlpy/index.html. [Accessed: 21-Mar-2019].

[13] *The SAS Deep Learning Python (DLPy) package provides the high-level Python APIs to deep learning methods in SAS Visual Data Mining and Machine Learning, https://github.com/sassoftware/python-dlpy*. SAS Software, 2019.

[14] "Kafka Connect — Confluent Platform." [Online]. Available: https://docs.confluent.io/current/connect/index.html. [Accessed: 09-Mar-2019].

[15] *Kafka Connect SAP is a set of connectors, using the Apache Kafka Connect framework for reliably connecting Kafka with SAP systems: https://github.com/SAP/kafka-connect-sap*. SAP, 2019.

[16] "Home [Europe (Rot) - Trial] > Home - SAP Cloud Platform Cockpit." [Online]. Available: https://account.hanatrial.ondemand.com/cockpit/#/home/trialhome. [Accessed: 21-Mar-2019].

[17] "SAP Development Tools." [Online]. Available: https://tools.hana.ondemand.com/. [Accessed: 09-Mar-2019].

[18] "Download ngdbc JAR 2.3.55 ➔ With all dependencies!" [Online]. Available: https://jar-download.com/artifacts/com.sap.cloud.db.jdbc/ngdbc/2.3.55/source-code. [Accessed: 21-Mar-2019].

[19] *SAP HANA Connector in pure Python, https://github.com/SAP/PyHDB*. SAP, 2019.

[20] "Home | Salesforce." [Online]. Available: https://eu16.lightning.force.com/lightning/setup/SetupOneHome/home. [Accessed: 21-Mar-2019].

[21] "Apache HttpComponents – Apache HttpComponents." [Online]. Available: http://hc.apache.org/. [Accessed: 21-Mar-2019].

[22] "Postman," *Postman*. [Online]. Available: https://www.getpostman.com. [Accessed: 25-Mar-2019].

[23] M. Grinberg, *Supporting code for my article on video streaming with Flask, https://github.com/miguelgrinberg/flask-video-streaming*. 2019.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Olivier Thierie
Deloitte Belgium
othierie@deloitte.com

Wouter Travers
Deloitte Belgium
wtravers@deloitte.com

Andrew Pease
Deloitte Belgium
apease@deloitte.com