



**Illinois Institute of Technology**

**Project Report**

**CSP554: Big Data Technologies**

**REAL-TIME BIGDATA PIPELINE ANALYTICS**

Instructed by: Prof. Joseph Rosen

Submitted by:

Anushka Sonawane (A20490581)  
Sagar Shekhargouda Patil (A20501427)  
Shashank Parameswaran (A20502276)  
Sourabh Patil (A20490933)  
Yuvraj Nikam (A20501952)

# Index

<b>ABSTRACT .....</b>	<b>3</b>
<b>OBJECTIVE.....</b>	<b>3</b>
<b>PROJECT GOALS .....</b>	<b>4</b>
<b>SPECIFIC QUESTIONS.....</b>	<b>4</b>
<b>INTRODUCTION.....</b>	<b>4</b>
DIFFERENT COMPONENTS THAT ARE AVAILABLE IN BIG DATA PIPELINES:.....	4
APACHE SPARK.....	4
BATCH PROCESS .....	4
APACHE HADOOP .....	5
DATA LAKE.....	5
APACHE KAFKA.....	5
AMAZON KINESIS .....	5
JUPYTER.....	5
ML ALGORITHMS .....	6
PANDAS .....	6
<b>WHY APACHE KAFKA?.....</b>	<b>7</b>
<b>WHY JUPYTER NOTEBOOK? .....</b>	<b>7</b>
<b>CLOUD STORAGE SERVICES.....</b>	<b>7</b>
<b>WHY DYNAMODB OVER OTHER DATABASE SERVICES? .....</b>	<b>8</b>
<b>WHY NOSQL OVER SQL? .....</b>	<b>8</b>
<b>WHY TABLEAU?.....</b>	<b>9</b>
<b>ARCHITECTURE DIAGRAMS FOR BIG DATA PIPELINE .....</b>	<b>9</b>
I.    KAFKA.....	9
II.   AMAZON KINESIS .....	10
<b>BIG DATA PIPELINE PROCESSING .....</b>	<b>11</b>
DATA PROPERTIES:.....	11
AWS KINESIS PIPELINE:.....	11
APACHE KAFKA PIPELINE:.....	16
DATA PREPARATION:.....	19
DATA WRANGLING:.....	19
MODEL BUILDING:.....	20
DATA VISUALIZATION:.....	21
<b>COMPARATIVE STUDY BETWEEN KAFKA AND KINESIS PIPELINE .....</b>	<b>25</b>
<b>CONCLUSION.....</b>	<b>25</b>
<b>DATA SOURCES.....</b>	<b>26</b>
<b>SOURCE CODE.....</b>	<b>26</b>
<b>REFERENCES .....</b>	<b>27</b>

## **ABSTRACT**

In this project, we are working on a huge real-time dataset from the City of Chicago Dataset like Traffic Congestion estimates, Red Light Camera Violations, Speed Violations, Traffic Crashes, and Towed Vehicles dataset containing multiple features which are important to draw a conclusion about these violations and their effects or damages may be financial or health-related. We have visualized various parameters of this dataset to get a better understanding of this data and build a Machine Learning model to build the outcome using Logistic Regression and Random Forests. Also, we have built a pipeline by using tools such as Kafka/ AWS Kinesis and AWS DynamoDB by extracting real-time data and loading it into this pipeline.

## **OBJECTIVE**

Build an end-to-end big data pipeline from extracting data in real-time to loading data into a database and performing analytics services such as visualization by exploring the use of tools such as Kafka or AWS Kinesis and AWS DynamoDB or EMR to accept and process data in real-time. The goal of the project is to analyze different criteria and measures associated with the Violations and Crashes dataset. Using different tools to analyze and predict various factors and consequences of violations and crashes. We process the data using a data pipeline using two different approaches and apply various transformations to the data, and then we provide visualizations for various factors associated with the dataset.

## PROJECT GOALS

- Data cleansing by using Lambda functions.
- Upload necessary files to the S3 bucket for data processing.
- Using Lambda functions to upload the data to DynamoDB.
- Injecting the cleansed data into DynamoDB.
- Enabling Kinesis/Kafka Data Streaming for DynamoDB.
- Visualizing the data using Tableau 2023.1
- Using Machine Learning Model Logistic Regression and Random Forests for prediction

## SPECIFIC QUESTIONS

Along with the problem statement, we have addressed the below questions to the best of our ability

Question 1: When are most Car Crashes happening?

Question 2: How much is the financial damage per incident?

Question 3: Are all car crashes fatal?

Question 4: How does road conditions affect the car crash?

## INTRODUCTION

**Different components that are available in big data pipelines:**

### Apache Spark

For large-scale data processing, Apache Spark is a fast and general-purpose cluster computing system. It is well-known for its speed, generality, ease of use, and platform compatibility. It executes programs in memory up to 100 times faster than Hadoop MapReduce and on disk up to ten times faster. An advanced DAG (Direct Acyclic Graph) execution engine in Apache Spark enables in-memory computing and acyclic data flow. It provides over 80 high-level operators, making parallel app development simple. It is a large-scale analytics engine with built-in SQL, streaming, machine learning, and graph processing modules.

### Batch Process

In the batch processing technique, data is processed in batches at fixed intervals of time. The data

is first stored in a batch, and then the batch is processed by a batch processing system.

### **Apache Hadoop**

Apache Hadoop is an open-source software framework that is used to store and process large datasets. It provides a distributed file system and a framework for running MapReduce jobs.

### **Data lake**

A data lake is a centralized repository that allows organizations to store all types of data, including structured, semi-structured, and unstructured data, at scale. Data lakes are used for storing big data and provide a cost-effective way to store large volumes of data.

### **Apache Kafka**

Apache Kafka is an open-source distributed streaming platform that is used for building real-time data pipelines and streaming applications. This technique provides a scalable and fault-tolerant platform for handling large volumes of data in real time.

### **Amazon Kinesis**

Amazon Kinesis cost-effectively processes and analyzes streaming data at any scale as a fully managed service. With Kinesis, we can ingest real-time data, such as video, audio, application logs, website clickstreams, and IoT telemetry data, for machine learning (ML), analytics, and other applications. We can use Amazon Kinesis Data Streams to collect and process large streams of data records in real-time.

### **Jupyter**

Jupyter notebook is accessed through our browser and can be launched locally on our machine as an area server or remotely on a server. The name notebook comes from the fact that it can contain live code, rich text elements such as equations, links, photos, tables, and so on. As a result, you may have a very good notebook to discuss your idea as well as the live code dead one document. For example, we have written major code parts entirely within the Jupyter notebook. The different platform and library tools like Pyspark, Scikit Learn, Numpy, Pandas, Seaborn are used for data analysis for more flexibility.

## **ML Algorithms**

Common machine learning algorithms such as classification, regression, clustering, and collaborative filtering.

## **Pandas**

Pandas library could be a very high-level Python library that has high-performance, easy-to-use data structures. it's many functions for data importing, manipulation, and analysis. specifically, it offers data structures operations for manipulating numerical tables and statistics. Pandas is installed by using the 'pip install pandas' command.

## WHY APACHE KAFKA?

We chose Apache Kafka as a data processing technique for our project considering the following advantages of Kafka:

**Scalability:** AWS Kafka can handle a large volume of data streams and scale up or down as per the demand, making it an ideal choice for organizations that require a highly scalable streaming platform.

**Reliability:** AWS Kafka is highly reliable and can sustain high throughput with low latency. It also provides built-in fault tolerance, ensuring that data is not lost even if a broker fails.

**Real-time data processing:** AWS Kafka provides real-time data processing, allowing you to stream data from multiple sources in real time and process it immediately.

**Flexibility:** AWS Kafka supports a variety of use cases, from real-time data processing and analytics to log aggregation and messaging. This makes it a versatile platform that can be used in a wide range of applications.

**Security:** AWS Kafka provides robust security features, including encryption, access control, and authentication, ensuring that your data is protected against unauthorized access.

**Cost-effective:** With AWS Kafka, you only pay for what you use, making it a cost-effective solution for streaming data processing.

## WHY JUPYTER NOTEBOOK?

Jupyter notebook is accessed through our browser and can be launched locally on our machine as an area server or remotely on a server. The name notebook comes from the fact that it can contain live code, rich text elements such as equations, links, photos, tables, and so on. As a result, you may have a very good notebook to discuss your idea as well as the live code in one document. For example, we have written major code parts entirely within the Jupyter notebook. The different platforms and library tools like Pyspark, Scikit Learn, Numpy, and Pandas are used for data analysis for more flexibility.

## CLOUD STORAGE SERVICES

Google Cloud and AWS S3 buckets are two popular cloud storage services used in today's tech world. Both have their own pros and cons, we are preferring S3 bucket over google cloud services for the following reasons:

1. Wider range of storage classes: S3 provides a wide range of storage classes with different features and pricing options, including Standard, Infrequent Access, Glacier, and others. This allows users to choose the most cost-effective storage class for their specific needs.
2. Better object lifecycle management: S3 has more advanced object lifecycle management features, allowing users to automate the transition of objects between storage classes or to delete them automatically after a specified period of time. Google Cloud Storage has similar features, but they are not as flexible as S3.
3. Better integration with AWS services: S3 integrates seamlessly with other AWS services, such as Lambda, EC2, and CloudFront, allowing for easier development of complex applications. While Google Cloud Storage also integrates with other Google Cloud services, the integration is not as extensive as with AWS.

## **WHY DYNAMODB OVER OTHER DATABASE SERVICES?**

There are multiple database services like Apache Cassandra, MongoDB, Couchbase, and Microsoft Azure Cosmos DB available but as compared to them DynamoDB offers several advantages as

1. DynamoDB is a fully managed service, meaning that AWS handles the underlying infrastructure, scaling, and management of the database.
2. DynamoDB provides predictable and consistent performance, regardless of the size of the database or the number of requests being made. This is achieved through automatic scaling and load balancing.
3. DynamoDB integrates seamlessly with other AWS services, such as Lambda, EC2, and CloudFormation, allowing for easy development of complex applications.
4. DynamoDB provides built-in security features, such as encryption at rest and in transit, fine-grained access control, and integration with AWS Identity and Access Management (IAM).

## **WHY NOSQL OVER SQL?**

Although the data that we are using is structured, we would like to explore and use NoSQL databases and hence we have decided to proceed with AWS DynamoDB as our choice of database for this project.



## WHY TABLEAU?

Tableau is an end-to-end data analytics platform that allows you to prepare, analyze, collaborate, and share your big data insights. It is a powerful tool with several useful features that allows smooth and hassle-free data analysis. Following are its significant features:

- **Intuitive Dashboards:** It allows users to get valuable data insights by creating intuitive and personalized dashboards. You can create a dashboard to track profits and losses, a sales pipeline dashboard, a quarterly expenditure forecast dashboard, and more.
- **Real-time Analytics:** The online product constantly updates data and allows you to access real-time data on a mobile or laptop.
- **Data Integration:** It can connect to various data sources within minutes and extract data. You can integrate it with existing cloud technologies.
- **Data Security:** It allows you to restrict viewing or editing access to ensure data security.

## ARCHITECTURE DIAGRAMS FOR BIG DATA PIPELINE

### I. KAFKA

This is Architecture Diagram for the Kafka Big data pipeline.

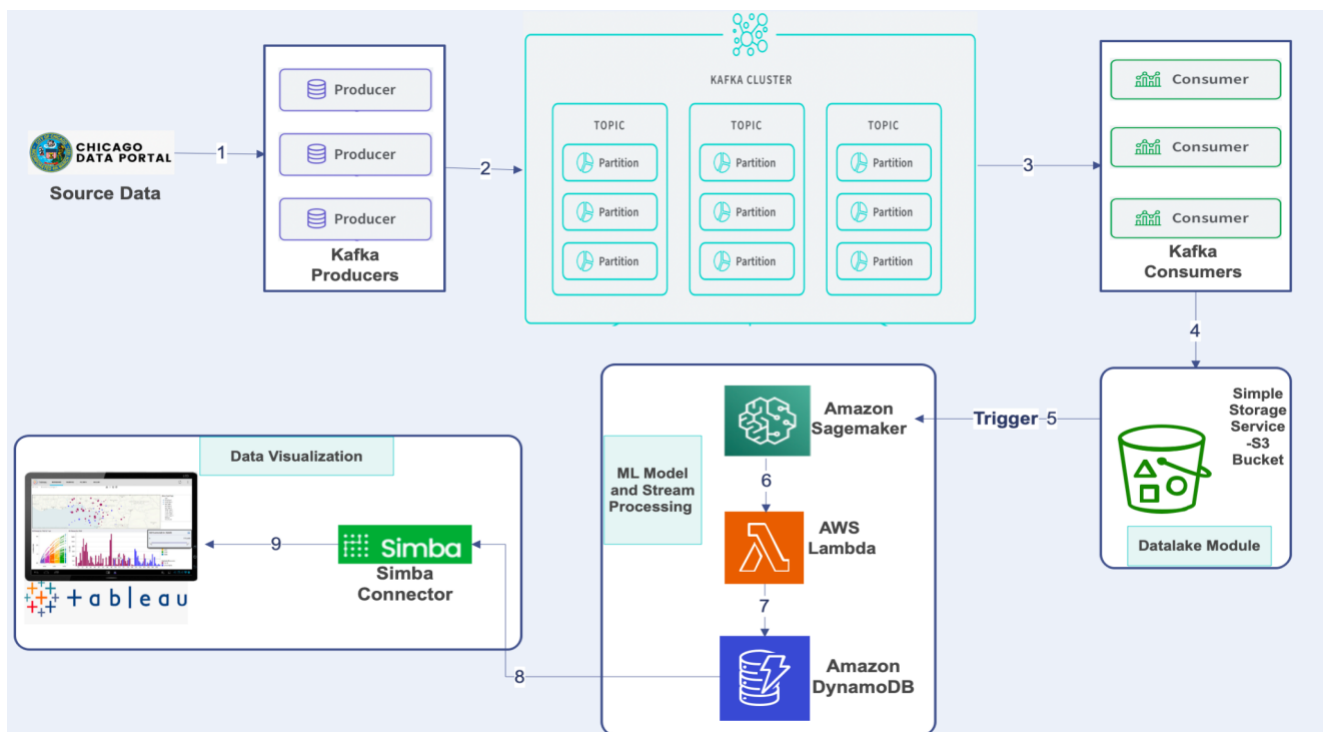


Figure 1.Kafka Architecture

## II. AMAZON KINESIS

This is Architecture Diagram for the Kinesis Big data pipeline.

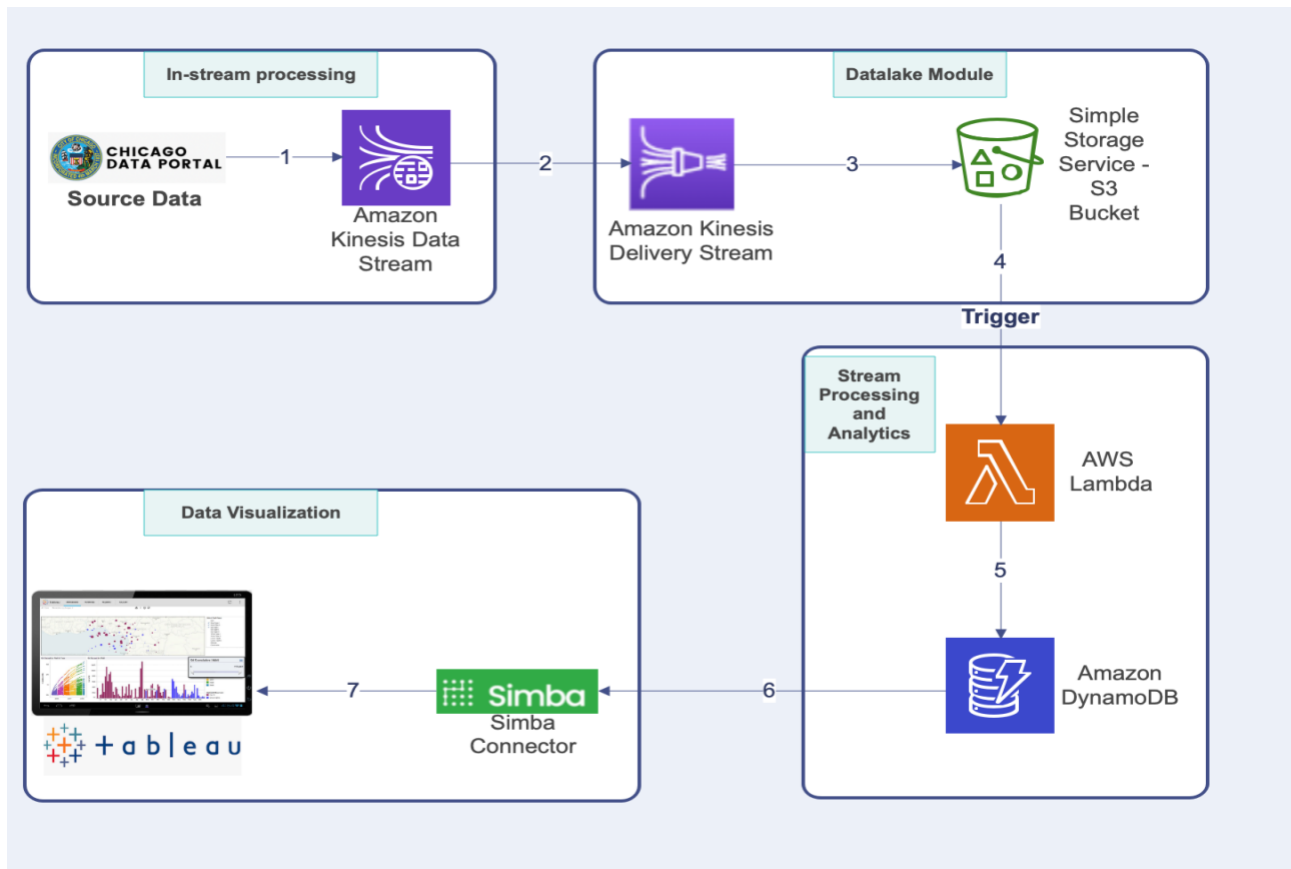


Figure 2.Kinesis Architecture

## **BIG DATA PIPELINE PROCESSING**

### **Data Properties:**

The dataset used in this project is from the City of Chicago dataset. This dataset has real-time data to cover multiple aspects like the taxi dataset, traffic analysis dataset, and more. We have chosen the speed dataset, towing dataset, and red-light violations dataset to get matrices like speed data, various violations data, and vehicles towed data and draw a conclusion and analysis from these data. It is discussed more in the data visualization section.

### **AWS Kinesis Pipeline:**

To start off, we have taken the necessary steps to create a role that is suitable for our needs. This role will enable us to grant specific permissions to certain services that we will be using. We have carefully assessed our requirements and determined that we will need to grant permissions to three services, namely Kinesis, S3, and DynamoDB.

After setting up the role and granting the necessary permissions, we proceeded to set up the Kinesis stream for each data source. We made sure to carefully configure the stream to ensure that it meets our specific requirements. We tested the stream thoroughly to make sure that it works as intended.

To provide a visual representation of the Kinesis stream that we have set up, we have included a screenshot below. This screenshot provides an overview of the configuration settings that we have selected, as well as any other relevant details that may be important to consider. We hope that this will give you a better understanding of the steps that we have taken to set up the Kinesis stream for each data source.

Amazon Kinesis > Data streams

**New on-demand mode for Kinesis data streams**  
 On-demand mode eliminates the requirement to manually provision and scale your data streams. With on-demand mode, your data streams automatically scale their write capacity of up to 200 MiB/second. [Learn more](#)

**Data streams (3) Info**
Process data in real time
Create a Firehose delivery stream
Actions ▾
Create data stream

< 1 > ⚙

<input type="checkbox"/>	Name ▲	Status ▾	Capacity mode ▾	Provisioned shards ▾	Data retention period ▾	Encryption ▾	Consumers with enhanced fan-out ▾
<input type="checkbox"/>	kinesis	Active	On-demand	-	1 day	Disabled	0
<input type="checkbox"/>	kinesis_RedLightViolations	Active	On-demand	-	1 day	Disabled	0
<input type="checkbox"/>	kinesis_SpeedViolations	Active	On-demand	-	1 day	Disabled	0

Figure 3. Kinesis Data Stream Configuration

Once we've set up the Kinesis stream, we can use Kinesis Data Firehose to automatically transfer the data from the Kinesis stream to an S3 bucket, without the need for custom coding or manual intervention. To do this, we create a delivery stream for each Kinesis stream we've set up.

The delivery stream is a destination where Kinesis Data Firehose will put the data that it pulls from the Kinesis stream. In our case, we'll configure the delivery stream to put the data into an S3 bucket, which is a highly scalable and durable object storage service provided by AWS.

After setting up the delivery stream and configuring it to point to the S3 bucket, Kinesis Data Firehose will automatically pull the data from the Kinesis stream and put it into the designated S3 bucket. This process is fully managed by AWS, which means we don't need to worry about infrastructure setup or maintenance, and can focus on analyzing the real-time data that's being streamed.

Amazon Kinesis > Data Firehose

Delivery streams (3)

Find delivery streams

Name	Status	Creation time	Source	Data transformation	Destination Type	Destina...
KDS-S3-blo89	Active	April 19, 2023 at 22:56 ...	kinesis	Not enabled	Amazon S3	my-towin...
KDS-S3-RedLightViolations	Active	April 21, 2023 at 22:07 ...	kinesis_RedLightViolati...	Not enabled	Amazon S3	redlightvi...
KDS-S3-Speed-Violations	Active	April 22, 2023 at 23:29 ...	kinesis_SpeedViolations.	Not enabled	Amazon S3	speed-viol...

Figure 4. Kinesis Firehose Configuration

We have also created separate buckets for each Kinesis data source as shown below:

Amazon S3 > Buckets

Account snapshot

Storage lens provides visibility into storage usage and activity trends. [Learn more](#)

View Storage Lens dashboard

Buckets (4)

Buckets are containers for data stored in S3. [Learn more](#)

Find buckets by name

Name	AWS Region	Access	Creation date
my-towing-data	US East (N. Virginia) us-east-1	Bucket and objects not public	April 19, 2023, 19:51:59 (UTC-05:00)
redlightviolations	US East (N. Virginia) us-east-1	Bucket and objects not public	April 21, 2023, 22:05:21 (UTC-05:00)
speed-violations	US East (N. Virginia) us-east-1	Bucket and objects not public	April 22, 2023, 23:27:52 (UTC-05:00)

Figure 5. Buckets

After setting up the delivery stream to put the data from the Kinesis stream into an S3 bucket, the next step is to configure AWS Lambda to process the data. AWS Lambda is a serverless compute service that allows you to run code without provisioning or managing servers. In this case, we're using Lambda to take the data from the S3 bucket and put it into the appropriate DynamoDB table.

To set this up, we first create a Lambda function for each S3 bucket we're using. The Lambda function is then configured to be triggered whenever new data is added to the corresponding S3 bucket. This way, the Lambda function is automatically invoked every time new data arrives in the S3 bucket.

Once the Lambda function is triggered, it takes the data from the S3 bucket and processes it. The processing logic will depend on the specific requirements of your project. In this case, the Lambda function is putting the data into the appropriate DynamoDB table, which is a fully-managed NoSQL

database provided by AWS.

Below is an example of what the Lambda code might look like. This code is specific to one of the buckets, and similar code would need to be written for the other buckets:

```
lambda_function × (+)

import json
import boto3

def lambda_handler(event, context):
    # Get the S3 object details from the event
    s3 = boto3.client('s3')
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = event['Records'][0]['s3']['object']['key']

    # Get the contents of the S3 object
    obj = s3.get_object(Bucket=bucket, Key=key)
    contents = obj['Body'].read().decode('utf-8')

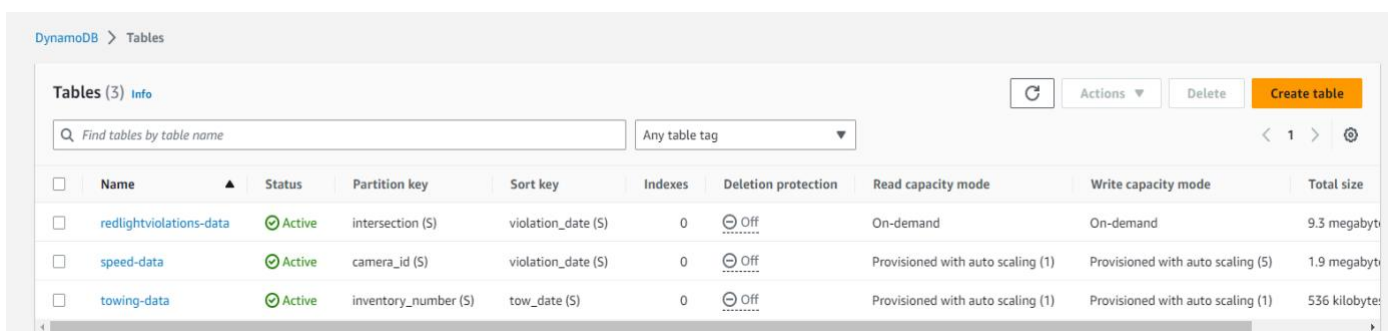
    # Split the contents by '}' to handle multiple JSON objects
    chunks = contents.split('}')
    for chunk in chunks:
        if chunk.strip():
            # Add back the '}' that was removed by the split
            data = json.loads(chunk + '}')

            # Insert the data into the DynamoDB table
            dynamodb = boto3.resource('dynamodb')
            table = dynamodb.Table('towing-data')
            table.put_item(Item=data)

    return {
        'statusCode': 200,
        'body': json.dumps('Documents inserted into DynamoDB')
    }
```

Figure 6.Lambda Function

We have also created separate DynamoDB tables for each data source and below is the Screenshot for the same.



	Name	Status	Partition key	Sort key	Indexes	Deletion protection	Read capacity mode	Write capacity mode	Total size
<input type="checkbox"/>	redlightviolations-data	Active	intersection (S)	violation_date (S)	0	Off	On-demand	On-demand	9.3 megabyte
<input type="checkbox"/>	speed-data	Active	camera_id (S)	violation_date (S)	0	Off	Provisioned with auto scaling (1)	Provisioned with auto scaling (5)	1.9 megabyte
<input type="checkbox"/>	towing-data	Active	inventory_number (S)	tow_date (S)	0	Off	Provisioned with auto scaling (1)	Provisioned with auto scaling (1)	536 kilobyte

Figure 7.DynamoDB Tables

Once we have set up all these, we simply run the Python code for pulling the data from the source and sending it to the Kinesis streams and screenshot of the code for the towing is shown below

```
1 import requests
2 import boto3
3 import json
4
5 # set up the API request to fetch data from Chicago data portal API
6 url = "https://data.cityofchicago.org/resource/ygr5-vcbg.json"
7 params = {
8     "$limit": 1000,
9     "$offset": 0
10 }
11 records = []
12
13 while True:
14     response = requests.get(url, params=params)
15     data = response.json()
16     records += data
17
18     if len(data) < 1000:
19         break
20
21     params["$offset"] += 1000
22
23 # set up the Kinesis client
24 kinesis_client = boto3.client('kinesis', region_name='us-east-1')
25
26 # send data to Kinesis
27 for item in records:
28     # convert the item to JSON format and encode it as bytes
29     item_json = json.dumps(item)
30     item_bytes = item_json.encode('utf-8')
31
32     # send the data to Kinesis
33     response = kinesis_client.put_record(
34         StreamName='kinesis',
35         Data=item_bytes,
36         PartitionKey=item['inventory_number']
37     )
```

Figure 8.Kinesis Stream with Python Code

To summarize the entire process, we begin by running the code that pulls data from the City of Chicago source. This code extracts data in real time and sends it to the appropriate Kinesis stream, which is a managed service provided by AWS that allows for real-time data streaming. The Kinesis stream receives and stores the data in a scalable and durable manner.

Once the data is in the Kinesis stream, the Kinesis Data Firehose delivery stream automatically pulls the data from the Kinesis stream and puts it into the appropriate S3 bucket. This S3 bucket serves as a staging area for the data.

Next, the AWS Lambda function that is associated with the S3 bucket is triggered automatically whenever new data arrives in the S3 bucket. This Lambda function takes the data from the S3 bucket and puts it into the appropriate DynamoDB table. DynamoDB is a fully-managed NoSQL database provided by AWS that allows for fast and flexible querying of structured and unstructured data.

Once the data is in DynamoDB, we use the Simba connector in Tableau to pull the data from DynamoDB into Tableau for visualization. The Simba connector allows for easy integration of DynamoDB with Tableau, and provides a simple and efficient way to visualize the data in a variety of formats.

Overall, this entire process allows for real-time extraction, processing, and visualization of data from the City of Chicago source, providing insights and value to users in an efficient and scalable way.

### **Apache Kafka Pipeline:**

Using a similar process as above, but through Kafka, we have put the data into S3 and DynamoDB. The first step of this project involves fetching data from various APIs. Kafka provides a distributed streaming platform that is scalable and fault-tolerant. To obtain data from API URLs, we have utilized Kafka Producer, which allows us to send messages to specific Kafka topics that we have created for each data source. The producer retrieves data from the API and forwards it to the appropriate Kafka topic. The code below contains the `producer_send(str, str)` function that performs this task. The function is flexible enough to take any topic and URL and fetch the data. A record size of 500 was used because Kafka Consumer seemed to have a limit of ~1MB per message that couldn't be changed (we tried using various parameters such as `fetch_max_bytes`, `max_partition_fetch_bytes` and `max_poll_records`).



```

def producer_send(topic, api_url):
    # Set up the API endpoint and request parameters
    record_size = 500
    print("Starting send for ", topic)

    print(api_url)
    response = requests.get(api_url)
    # Make a GET request to the API and publish the response to Kafka
    #if topic_p == topic:
    if response.status_code == 200:
        data = response.json()
        for item in data:
            # remove the 'location' key if it exists
            if 'location' in item:
                del item['location']
        print("Size of response ", len(data))
        if len(data)>0:
            for i in range(0, len(data), record_size):
                producer.send(topic, data[i:i+record_size])
                producer.flush()
                print('Sending ', i+record_size)
            else:
                print("No data for in specified date range for ", topic)
        else:
            print(f'Error retrieving data from API: {response.status_code}')

producer_dict = get_producer_dict(start_date, end_date)
producer_send(topic_p, producer_dict[topic_p])

```

Figure 9. Kafka Producer code. Please refer to the Kafka\_producer.py for the entire code

The Kafka consumer then retrieves data from the Kafka topics and stores it in Amazon S3. S3 is a highly scalable object storage system designed to manage any volume of data. The below code illustrates this functionality. Just like the producer, the consumer is also versatile in picking up any topic and pushing it to the respective S3 bucket with a use of the dictionary (bucket\_dict) that stores the Topic name and its respective S3 destination as key value pairs. Since we have messages of just ~1MB each, we would have a lot of writes to S3 which makes it costly. To prevent this, we receive messages upto a certain data size (length) using the MAX\_PAYOARD\_SIZE/total\_payloads and then write it to S3.

```

# Consume Kafka messages and write to S3
for message in consumer:
    bucket_name = bucket_dict[topic]
    print(f"Retrieving {message.topic} in {date_str} from Producer")
    try:
        # Decode the message payload
        payload = json.loads(message.value)

        payloads.extend(payload) # append payload to list
        total_size += len(payload) # add payload size to total
        total_payloads_ += 1
        if total_size >= MAX_PAYLOAD_SIZE or total_payloads == total_payloads_:
            # if we have accumulated enough payloads, upload them to S3
            json_array = json.dumps(payloads)
            s3.put_object(
                Bucket=bucket_name,
                Key=f'{message.topic}/{message.partition}/{message.offset}.json',
                Body=json_array,
                ContentType='application/json'
            )
            print("Payload put in S3:", total_size)
            # reset variables for next batch of payloads
            payloads = []
            total_size = 0
            time.sleep(60*20)
    except (ValueError, KeyError) as e:
        # Handle any exceptions and continue consuming
        print(f'Error occurred', e)
        continue

print("Consumer Done!!")

```

Figure 10.Kafka Consumer code. Please refer to the Kafka\_consumer.py for the entire code

The stored data can be used for various data processing tasks. To process the data in S3, we have employed a Lambda function that is triggered whenever an object is created in any S3 bucket. This function moves the corresponding data to the correct DynamoDB table. As you can see in the above consumer code, we had to use a sleep function in the consumer. This is because the lambda function takes some time to write the data into S3. Waiting for 20 mins were optimal as each mini-batch (batch of messages) took around 10 mins to get put into DynamoDB. We configured the lambda function with 512MB memory. For DynamoDB, we used OnDemand read and write capacity. To load the Crashes dataset (~31,000 rows), it cost us 2\$. If we reduced the lambda function memory, we would have to change the sleep time accordingly.

## Data Preparation:

Using Amazon Sagemaker, data available in S3 bucket is first imported into the notebook using a boto3 client. Next, we need to perform feature engineering in order to feed in the best quality of predictor variables into a machine learning model. The Crashes dataset has 48 variables.

Using univariate analysis, i.e., exploring each feature's distribution and summary statistics, we removed features such as work zone type, primary contributory cause, private property, dooring, intersection related etc. These variables had imbalanced distribution with more than 95% of the data points belonging to one category. The model will not be able to learn or assign the right coefficients using these features and hence they were removed.

## Data Wrangling:

```
1 data.loc[:, "traffic_control_device"] = np.where(data.loc[:, "traffic_control_device"] != "NO CONTROLS", 1, 0)
2 data.loc[:, "device_condition"] = np.where(data.loc[:, "device_condition"] == "FUNCTIONING PROPERLY", 1, 0)
3 data.loc[:, "weather_condition"] = np.where(data.loc[:, "weather_condition"].isin(["CLEAR", "CLOUDY/OVERCAST"]), 1, 0)
4 data.loc[:, "lighting_condition"] = np.where(data.loc[:, "lighting_condition"] == "DAYLIGHT", 1, 0)
5 data.loc[:, "roadway_surface_cond"] = np.where(data.loc[:, "roadway_surface_cond"] == "DRY", 1, 0)
6 data.loc[:, "report_type"] = np.where(data.loc[:, "report_type"] == "ON SCENE", 1, 0)
7 data.loc[:, "crash_type"] = np.where(data.loc[:, "crash_type"] == "NO INJURY / DRIVE AWAY", 1, 0)
8 data.loc[:, "num_units"] = data.loc[:, "num_units"].astype(int)
9 data.loc[:, "num_units"] = np.where(data.loc[:, "num_units"] >= 3, 3, data.loc[:, "num_units"])
10 data.loc[:, "most_severe_injury"] = np.where(data.loc[:, "most_severe_injury"] != "NO INDICATION OF INJURY", 1, 0)
11 data.loc[:, "injuries_total"] = np.where(data.loc[:, "injuries_total"] >= 4, 4, data.loc[:, "injuries_total"])
12
13 # to be one hot encoded
14 data.loc[:, "trafficway_type"] = np.where(~data.loc[:, "trafficway_type"].isin(["DIVIDED - W/MEDIAN (NOT RAISED)", "NOT DIVIDED"]), 1, 0)
15 data.loc[:, "crash_hour"] = data.loc[:, "crash_hour"].astype(str)
16 data.loc[:, "crash_day_of_week"] = data.loc[:, "crash_day_of_week"].astype(str)
17
18 # Target variable
19 data.loc[:, "damage"] = np.where(data.loc[:, "damage"] == "OVER $1,500", 1, 0)
```

Figure 11. Data Wrangling

Additionally, some features such as crash ID, street name, crash date, and street number, are excluded as they cannot be directly correlated with damage prediction. Lastly, some features like the first crash type had a combination of different variables in it (object, collision point in vehicle, Turning Y/N) and hence it was removed.

After this we performed feature engineering where we converted most of the categorical variables into one hot encoded variable. We also clubbed categories with few data points without loss of generalization. The target variable was also binary encoded with damage over \$1500 being 1. Crash hour and Crash week were also one hot encoded as they are not linearly related to the target variable. Finally, we excluded features that were correlated with each other. If correlation exists within the data, it would lead to inaccurate coefficient estimates for machine learning techniques such as Logistic

regression. We used the Chi-square test for the categorical variables. We identified a few features that were correlated with one another and removed them.

### Model Building:

The objective of this analysis is to determine whether the damage exceeds \$1500 or not. After feature engineering, we have 38 variables (including one hot encoding of Crash Hour and Crash Week). The dataset was divided into a training set (80%) and a test set (20%), stratified based on the target variable to ensure consistency. We evaluated two classification models: logistic regression and Random Forest. To compare the performance of these models, we will be using the accuracy metric.

Logistic regression is one of the most popular techniques for solving classification problems. It is easy to implement and interpret. It can also interpret model coefficients as indicators of feature importance. We got a training accuracy of 69.46% and a testing accuracy of 69.5% using the logistic regression model.

```
1 # Logistic Regression
2 logreg = LogisticRegression(max_iter=1000)
3 logreg = logreg.fit(x_train, y_train)
4 yhat_logit_test = logreg.predict(x_test)
5 yhat_logit_train = logreg.predict(x_train)
6 print(confusion_matrix(y_test, yhat_logit_test))
7 print("training error: ", accuracy_score(y_train, yhat_logit_train))
8 print("testing error: ", accuracy_score(y_test, yhat_logit_test))
9
```

Figure 12. Logistic Regression

Random Forests is an ensemble learning method that uses multiple decision trees. The class that is most outputted by these individual decision trees is used as the predicted class. It can handle all types of features and requires very little pre-processing. Hyperparameter tuning was performed to get an optimal number of trees, criteria for information gain and depth. The model had a training accuracy of 69.57% and test accuracy of 69.58%. Both models gave very similar results with Random Forests slightly better. Since both training and test accuracy are consistent, there is no overfitting in the model.

```

10 # Random Forests (Bagging model)
11 # Hyperparameter tuning using Grid Search
12 param_grid = {"criterion": ['gini', 'entropy'],
13               "n_estimators": [200, 400, 600],
14               "min_samples_leaf": [40],
15               "max_depth": [3,5,7]
16               #"max_features": [None, 'auto']}
17
18 rf = RandomForestClassifier(n_jobs=8)
19 gs = GridSearchCV(
20     estimator=rf,
21     param_grid=param_grid,
22     cv=3,
23     refit=True,
24     verbose=True,
25     n_jobs=8)
26 gs.fit(x_train, y_train)
27 print(gs.best_params_)
28
29 rf = RandomForestClassifier(max_depth=3,
30                             random_state=4,
31                             min_samples_leaf=40,
32                             n_jobs=4,
33                             n_estimators=200,
34                             criterion='gini'
35                             )
36 rf.fit(x_train.values, y_train.values)
37 yhat_rf = rf.predict(x_test)
38 print("Testing Error: ", accuracy_score(y_test, yhat_rf))
39 yhat_rf_train = rf.predict(x_train)
40 print("Training Error: ", accuracy_score(y_train, yhat_rf_train))

```

Figure 13.Random Forests

A summary of the results is given below:

Model	Training Accuracy	Test Accuracy
Logistic Regression	69.46%	69.50%
Random Forests	69.57%	69.58%

Table 1.Summary of model results

### Data Visualization:

Data visualization is much needed to understand what the data resembles. In this project, we have used Tableau 2023.1 for data visualization. Data from the City of Chicago dataset is sorted and required fields from the huge dataset are used to draw analysis graphs as shown below:

The bar graph below reflects the number of crashes that occurred per day of the week. In this graph, we can see that Saturday is the day when maximum crashes happen while Monday holds the least number of crashes.

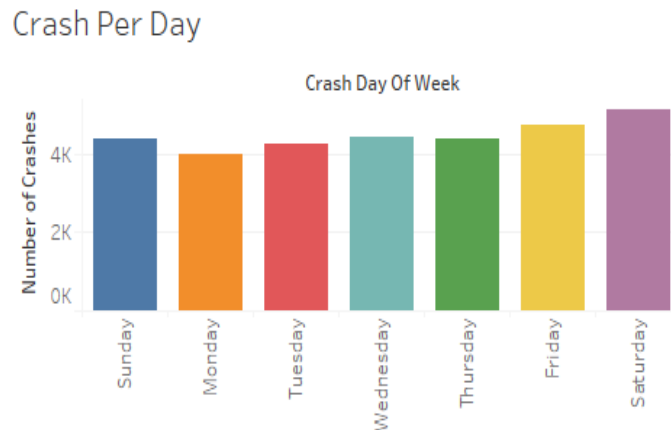


Figure 14. Number of crashes per day

In the below chart, we can see how the car towing data is compared from other states with Illinois. Data set being from Chicago we can see more numbers of towing records from Illinois.

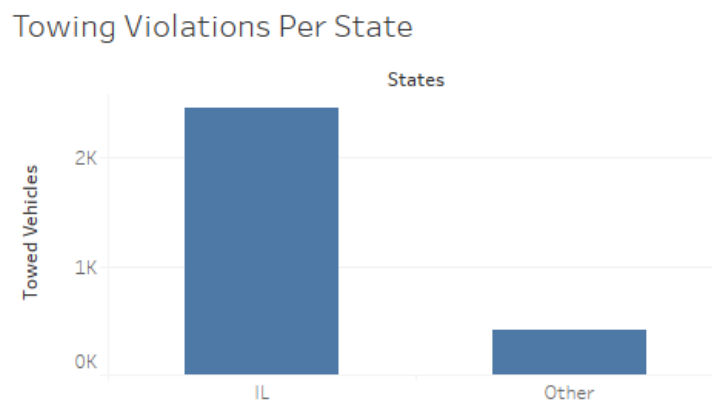


Figure 15. Towing Violations Per State

The below chart represents the data of damage that occurred in the crash. We can see most damages occurred are over \$1500 and very few damages are falling below damage range of \$500.

### Crashes Per Damage Range

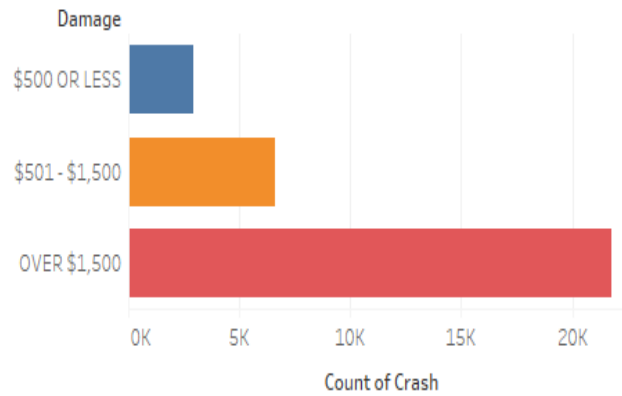


Figure 16. Crashes per damage

In a similar manner in the bar graph below, we have created a chart showing the injuries caused for every crash. Here we can see that most damages have no injuries caused and there are no crashes with fatal incidents.

### Injuries Per Crash

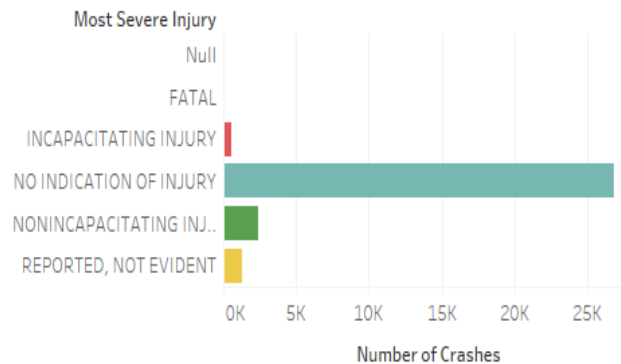


Figure 17. Injuries per crash

In the chart below we are trying to represent the data of crashes that occurred due to road defects. This chart shows that most of the crashes that occurred have no road defects involved.



## Road Defect

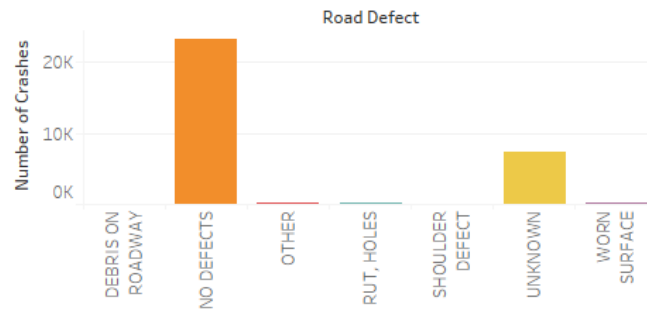


Figure 18.Road Defects

Below graphs represent the speed violations across the Chicago area where we can see the number of speed violations are dense in the downtown area compared to the outskirts. In the second graph, we can see how the speed violation cases are more during the weekdays in comparison to the weekends.

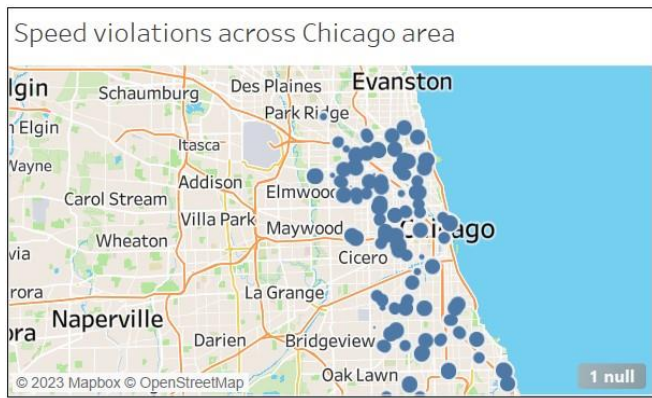


Figure 19.Speed violation in Chicago area

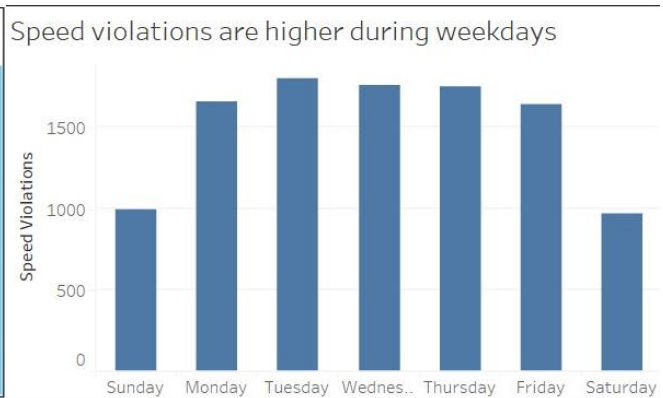


Figure 20.Speed violation per day



Figure 21. Redlight violations per day

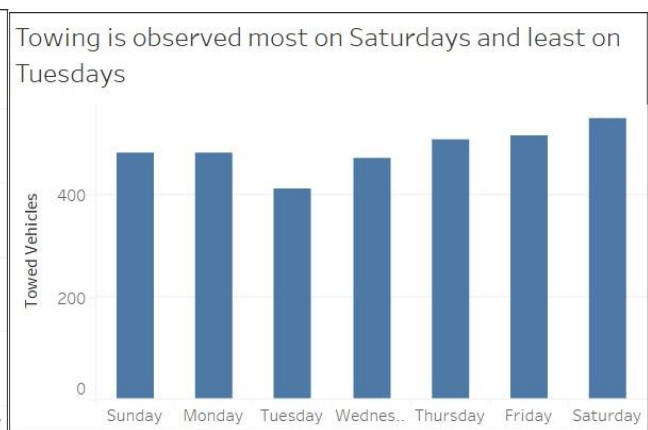


Figure 22.Towed Vehicles per day

In the graphs above we can see how red-light violation cases are more at the weekends. And the other graph represents towing violations for every day of the week. We can see how the violation cases are more on Saturdays and least on Tuesday.



## COMPARATIVE STUDY BETWEEN KAFKA AND KINESIS PIPELINE

Feature	Kafka Pipeline	Kinesis Pipeline
Software cost	None	On Demand or Provision (Very high based on our work)
Infrastructure costs	EC2 or physical infrastructure and can be optimized	Based on AWS rates
Engineering costs	High	Low
Security	Secure, but manual configuration leaves room for human error	Similar security to other AWS products
Ongoing management	Requires engineering effort	Managed by AWS
Data retention	User-defined	Up to 7 days
Performance tuning	Dependent on configuration and manual tuning – especially for features such as high availability	Less room for configuration, but some level of performance guaranteed by AWS

## CONCLUSION

This big data pipeline project has demonstrated the use of various tools and systems for efficiently processing and analyzing large amounts of data. We have successfully built a real time data pipeline using AWS Kinesis/Kafka, S3, Lambda functions, DynamoDB, Tableau and Sagemaker. We used two approaches for the data streaming part AWS Kinesis and Kafka. To process the data, a Lambda function has been employed that is triggered upon any change in S3 and puts the data into the DynamoDB NoSQL database. We were able to successfully visualize our dataset and create Machine Learning models using Sagemaker with the use of models such as Logistic Regression and Random Forests. Also, we were successfully able to answer all the questions we posed at the beginning of this project.

## DATA SOURCES

The data we utilized for this big data pipeline project is available at link [here](#).(City of Chicago datasets)

We used multiple datasets from this link.

## SOURCE CODE

Source code for both the approaches are available at [GitHub link here](#).

Code description is as follows:

File Name	Description
kafka_producer.py	Producer code for Kafka
kafka_consumer.py	Consumer code for Kafka
speedViolations.py, redlightViolations.py, towing.py,	These are the lambda codes using which we pull the data from S3 and put it into DynamoDB tables.
crashes_ml.ipynb	Using Amazon Sagemaker's Jupyter notebooks we built a classification model to predict the damage based on various factors

## REFERENCES

References used for creating AWS kinesis and Apache Kafka pipelines, related to big data technologies:

- [1] Narkhede, N., Shapira, G., & Palino, T. (2017). Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale. “O’Reilly Media, Inc.”
- [2] Jiang, W., & Luo, J. (2022). Big Data for Traffic Estimation and Prediction: A Survey of Data and Tools. Applied System Innovation, 5(1), 23. <https://doi.org/10.3390/asi5010023>
- [3] [What is the AWS Data Pipeline?](#) - AWS Data Pipeline. (n.d.).
- [4] [AWS Kinesis](#)
- [5] [AWS Kinesis Firehose \(Delivery Stream\)](#)
- [6] [AWS S3 triggering Lambda](#)
- [7] [AWS DynamoDB](#)