IoT Real-Time Water Quality Monitoring

Team Members:

Youssef Shaaban
Ahmed Osama
Alhassan Mohamed
Amr Yasser

Project Mentor:

Abdelrahman Elsaied

Information Technology Institute (ITI)

September 2024

Abstract

The IoT Real-Time Water Quality Monitoring project aims to provide real-time and historical analysis of water quality at Chicago Park District beaches along Lake Michigan. By leveraging IoT sensors deployed in the water, the system collects hourly measurements of critical water quality parameters such as water temperature, turbidity, and wave height. The project implements two data processing pipelines—one for real-time monitoring to aid immediate decision-making, and another for long-term analysis to observe environmental trends over time. The solution is deployed both on cloud infrastructure using Google Cloud Platform (GCP) and on-premises systems, utilizing technologies such as Apache Kafka, Apache Spark, Elasticsearch, Kibana, Google Dataflow, BigQuery, and Power BI. This document details the project's architecture, implementation, challenges faced, and future enhancements.

Contents

Abstract **1**

Contents **2**

List of Figures **4**

List of Tables **5**

Acronyms **6**

1 Project Overview **7**
1.1 Introduction . . . 7
1.2 Pipelines . . . 7
1.2.1 Real-Time Monitoring Pipeline . . . 7
1.2.2 Long-Term Analysis Pipeline . . . 7
1.3 Project Implementation . . . 7
1.3.1 Cloud-Based Implementation (GCP) . . . 8
1.3.2 On-Premise Implementation . . . 8

2 Technologies and Requirements **9**
2.1 Technologies Used . . . 9
2.1.1 Cloud-Based Implementation . . . 9
2.1.2 On-Premise Implementation . . . 10
2.2 System Requirements . . . 10
2.2.1 Hardware Requirements . . . 10
2.2.2 Software Requirements . . . 11

3 Data Sources **12**
3.1 About the Dataset . . . 12
3.2 Dataset Details . . . 12
3.3 API Used for Data Retrieval . . . 12
3.3.1 API Details . . . 13
3.3.2 Example of API Client Implementation . . . 13

4 Architecture **15**
4.1 Overview . . . 15
4.2 Cloud Architecture . . . 15
4.2.1 Components in Cloud Architecture . . . 15
4.2.2 Data Flow . . . 16

4.2.3 Diagram . . . 17
4.3 Local Architecture . . . 17
4.3.1 Components in Local Architecture . . . 17
4.3.2 Data Flow . . . 18
4.3.3 Diagram . . . 19

5 Implementation and Deployment 20
5.1 Cloud-Based Implementation . . . 20
5.1.1 Kafka and Pub/Sub Setup . . . 20
5.1.2 Pub/Sub Configuration . . . 21
5.1.3 Dataflow Configuration . . . 21
5.1.4 Elastic Cloud Setup . . . 22
5.1.5 Kibana Dashboard Setup . . . 23
5.1.6 Cloud Storage to BigQuery Integration . . . 23
5.1.7 Power BI Integration . . . 25
5.2 Local-Based Implementation . . . 25
5.2.1 Tools and Libraries Used . . . 25
5.2.2 Kafka Stream Setup . . . 26
5.2.3 Data Transformation . . . 26
5.2.4 Target Systems . . . 27
5.2.5 Data Streaming Configuration . . . 28
5.2.6 Database Design and Staging . . . 29
5.2.7 SSIS Package for ETL . . . 30

6 Challenges and Future Work 31
6.1 Challenges . . . 31
6.1.1 Cloud Implementation Challenges . . . 31
6.1.2 Local Implementation Challenges . . . 31
6.2 Future Work . . . 32
6.2.1 Cloud Implementation Enhancements . . . 32
6.2.2 Local Implementation Enhancements . . . 32

7 Conclusion 33
7.1 Summary of Work . . . 33
7.2 Key Achievements . . . 33
7.3 Overcoming Challenges . . . 33
7.4 Impact and Future Directions . . . 34
7.5 Final Thoughts . . . 34

8 References 35

List of Figures

4.1 Architecture Diagram of the IoT Real-Time Water Quality Monitoring Project (Cloud) 17
4.2 Architecture Diagram of the IoT Real-Time Water Quality Monitoring Project (Local) 19

5.1 Pub/Sub and Dataflow Integration 21
5.2 Elastic Cloud Deployment Configuration 23
5.3 Kibana Dashboard - IoT Real-Time Water Quality Monitoring 23
5.4 Power BI Dashboard for Water Quality Data 25

List of Tables

3.1 Dataset Columns and Descriptions 13

Acronyms

GCP	Google Cloud Platform
IoT	Internet of Things
SSIS	SQL Server Integration Services
SQL	Structured Query Language
API	Application Programming Interface
ETL	Extract, Transform, Load
CSV	Comma-Separated Values
JSON	JavaScript Object Notation
HTTP	Hypertext Transfer Protocol
CPU	Central Processing Unit
RAM	Random Access Memory
SSD	Solid-State Drive
VM	Virtual Machine

Chapter 1

Project Overview

1.1 Introduction

The goal of this project is to monitor the water quality at Chicago Park District beaches along the Lake Michigan lakefront. The Chicago Park District has deployed sensors in the water at these beaches, which primarily capture hourly measurements during the summer when the sensors are operational. These measurements provide key data on water quality and sensor conditions. Data may be unavailable during other seasons or non-operational periods.

1.2 Pipelines

This project involves two key data processing pipelines, each serving a specific purpose:

1.2.1 Real-Time Monitoring Pipeline

The first pipeline is designed for real-time monitoring. It provides decision-makers with immediate insights through a dashboard that visualizes critical water quality metrics, such as water temperature, turbidity, and wave height, alongside sensor statuses. This real-time data enables swift decision-making, ensuring timely interventions for public safety.

1.2.2 Long-Term Analysis Pipeline

The second pipeline is dedicated to long-term analysis. It helps researchers and analysts study environmental trends and changes in water quality at the beaches over the years. This historical data can be used to observe trends, identify patterns, and support scientific studies. In this project, we played the role of data analysts, visualizing these changes over time to derive meaningful insights.

1.3 Project Implementation

The project was deployed using two different approaches—one on the cloud using Google Cloud Platform (GCP) and the other on an on-premise infrastructure.

1.3.1 Cloud-Based Implementation (GCP)

In the cloud implementation, the pipeline follows these steps:

- Data is ingested from Kafka into Google Cloud Pub/Sub.
- Pub/Sub then streams the data into Google Dataflow, where it is processed and sent to Elasticsearch, with Kibana used for real-time data visualization.
- Additionally, another branch of the data is sent from Pub/Sub to Cloud Storage, processed by Dataflow, and finally loaded into BigQuery. From BigQuery, Power BI is used for long-term trend analysis and visualization.

1.3.2 On-Premise Implementation

For the on-premise implementation, the pipeline follows a similar but locally managed process:

- Data is ingested from Kafka into PySpark, which processes the data and sends it to Elasticsearch for indexing and real-time visualizations.
- In parallel, processed data is sent from PySpark to a relational database. SSIS (SQL Server Integration Services) then loads the data into a data warehouse, where Power BI is used to visualize long-term trends and perform detailed analyses.

Chapter 2

Technologies and Requirements

2.1 Technologies Used

This project employs a variety of tools and technologies for data ingestion, processing, storage, and visualization. Apache Kafka serves as the common component in both the cloud and on-premise implementations. The details of each environment's technologies are listed below:

2.1.1 Cloud-Based Implementation

- **Google Cloud Pub/Sub**: A scalable, asynchronous messaging service used to decouple systems for real-time event-driven architectures. It enables real-time data streaming between the sensors and other cloud services.
- **Google Dataflow**: A fully managed service for stream and batch processing. It is used to transform and enrich data in real time before forwarding it to storage and analysis tools.
- **Elastic Cloud**: A managed service provided via the Google Cloud Marketplace for deploying Elasticsearch in the cloud. Elastic Cloud allows seamless deployment using configuration defaults or custom configurations and integrates with Google Cloud services. This is used for storing and indexing sensor data, making it available for real-time search and analytics through Kibana.
- **Kibana**: A powerful visualization tool that integrates with Elasticsearch to create real-time dashboards for monitoring water quality and other metrics. Kibana provides decision-makers with clear, actionable insights in real time.
- **Google Cloud Storage**: A scalable object storage system used for long-term data storage. Data can be persisted here for batch processing and historical analysis.
- **BigQuery**: A fully managed, serverless data warehouse designed for fast SQL queries on large datasets. Data from Google Cloud Storage is loaded into BigQuery for deeper historical and trend analysis.
- **Power BI**: A business analytics tool that connects to BigQuery to create detailed reports and visualizations, allowing for the analysis of long-term environmental changes at Chicago's beaches.

- **pubsub-group-kafka-connector/1.2.0**: A connector used to seamlessly integrate Kafka with Google Cloud Pub/Sub for efficient message streaming across the cloud environment.

2.1.2 On-Premise Implementation

The on-premise implementation of the project mirrors many components of the cloud architecture but uses different technologies for data processing and storage. The following tools were used for this implementation:

- **Apache Spark 4.2**: A fast, in-memory data processing engine used to process and transform real-time streams from Kafka. Spark is responsible for enriching and preparing data for storage in the local Elasticsearch or SQL Server.
- **Elasticsearch 8.15.1**: A distributed search and analytics engine. It is used in the on-premise implementation to store, index, and analyze real-time water quality data.
- **SQL Server (via pyodbc)**: A relational database management system. SQL Server is used to store enriched data for structured queries and analytics, as well as historical trend analysis.
- **SSIS (SQL Server Integration Services)**: An ETL (Extract, Transform, Load) tool used to load processed data from SQL Server into a data warehouse for long-term analysis.
- **Power BI**: This tool is used for creating visual reports based on data stored in the on-premise SQL Server. It enables local data visualization and analytics for decision-making.
- **Docker**: A containerization platform used to package and run the local services such as Kafka, Spark, and Elasticsearch in isolated environments, ensuring easy deployment and scaling.

2.2 System Requirements

The project demands specific hardware and software resources, particularly for the on-premise implementation. The following are the system requirements:

2.2.1 Hardware Requirements

The following hardware configuration is recommended for running the on-premise implementation efficiently:

- **CPU**: Multi-core processor with at least 4 cores to handle real-time data ingestion and processing.
- **Memory**: Minimum 16 GB of RAM, with 32 GB or more recommended for managing larger data volumes and ensuring smooth operation.

- **Storage**: At least 500 GB of storage capacity to accommodate both real-time data ingestion and long-term data storage needs.
- **Network**: A stable, high-bandwidth internet connection is required to handle real-time data ingestion and to ensure seamless communication between components.

2.2.2 Software Requirements

The software stack used in this project requires the following components:

- **Operating System**: WSL Ubuntu 24.04 or any compatible Linux distribution for the on-premise deployment.
- **Java**: JDK 11 or higher is necessary for running Apache Kafka and PySpark.
- **Python**: Version 3.8 or higher for running data processing scripts and interfacing with PySpark and SQL Server (via pyodbc).
- **Docker**: Version 20.10 or higher is required for containerizing the local services, ensuring scalability and ease of deployment.
- **Apache Kafka**: Version 2.12, used for real-time data streaming and message handling.
- **Apache Spark**: Version 4.2 for data processing and transformation.
- **Elasticsearch**: Version 8.15.1 for storing and indexing real-time data.
- **SQL Server**: Version 2019 or higher, accessed via pyodbc for data storage and querying.
- **Power BI**: A version compatible with data sources like BigQuery and SQL Server for data visualization.
- **SSIS (SQL Server Integration Services)**: For ETL operations between SQL Server and the data warehouse.

Chapter 3

Data Sources

3.1 About the Dataset

The dataset utilized for the IoT Real-Time Water Quality Monitoring project provides comprehensive information about the water quality at beaches managed by the Chicago Park District along Lake Michigan. Below are the relevant details regarding this dataset:

- **Last Updated:** September 29, 2024
- **Metadata Last Updated:** August 12, 2019
- **Date Created:** May 15, 2014
- **Data Provided by:** Chicago Park District
- **Dataset Owner:** Jonathan Levy
- **Views:** 39.4K
- **Downloads:** 25.8K
- **Time Period:** 5/28/2014 - current (in season) plus limited previous measurements
- **Frequency:** Hourly
- **Source Link:** `http://www.chicagoparkdistrict.com`

3.2 Dataset Details

The dataset comprises 45.9K rows and contains 10 columns, each representing specific measurements and attributes related to beach water quality (see Table 3.1).

3.3 API Used for Data Retrieval

To retrieve data for simulating real-time monitoring, the project utilizes the Chicago Park District API powered by Socrata.

Table 3.1: Dataset Columns and Descriptions

Column Name	**Description**	**Data Type**
Beach Name	Name of the beach	Text
Measurement Timestamp	The date and time when the measurements were taken	Floating Timestamp
Water Temperature	Water temperature in Celsius degrees	Number
Turbidity	Water turbidity in Nephelometric Turbidity Units (NTU)	Number
Transducer Depth	Transducer depth in meters	Number
Wave Height	Wave height in meters	Number
Wave Period	Wave period in seconds	Number
Battery Life	Battery voltage indicating remaining battery life	Number
Measurement Timestamp Label	Last Updated value in text format	Text
Measurement ID	Unique record ID (Beach Name + Measurement Timestamp)	Text

3.3.1 API Details

- All communication with the API is conducted through HTTPS, with errors communicated via HTTP response codes.
- Available response types include JSON (including GeoJSON), XML, and CSV, selectable by the "extension" (.json, etc.) on the API endpoint or through content-negotiation with HTTP Accepts headers.
- Each request requires an app token to identify the application, ensuring each application has its unique token.
- Each column in the dataset is represented by a single field in the SODA API, allowing for searching records, limiting results, and customizing data output using filters and SoQL queries.

3.3.2 Example of API Client Implementation

Below is a Python code snippet demonstrating how to use the Socrata API to fetch the first 2000 records from the dataset:

Listing 3.1: API Client Implementation

```
from sodapy import Socrata
import pandas as pd

client = Socrata("data.cityofchicago.org", None)
results = client.get("qmqz-2xku", limit=2000)
results_df = pd.DataFrame.from_records(results)
```

This code utilizes the `sodapy` library to interact with the Socrata API and `pandas` to convert the results into a DataFrame for further analysis.

Chapter 4

Architecture

This chapter outlines the architecture of the IoT Real-Time Water Quality Monitoring project. The architecture consists of several components that work together to monitor and analyze water quality data from the beaches along Chicago's Lake Michigan lakefront. The system leverages Apache Kafka as the central messaging service, with distinct cloud and local implementations for data processing, storage, and visualization.

4.1 Overview

The architecture comprises multiple components that work collaboratively to ensure effective monitoring and analysis of water quality data. The project is deployed in two different environments: a cloud-based implementation leveraging Google Cloud services, and a local implementation hosted on-premise.

4.2 Cloud Architecture

The cloud architecture leverages Google Cloud Platform services and Elastic Cloud for real-time data processing, storage, and visualization. This hybrid approach allows for scalable, efficient handling of water quality monitoring data.

4.2.1 Components in Cloud Architecture

The major components of the cloud architecture include:

- **API:** The entry point for data ingestion, the Chicago Park District API.
- **Kafka:** A distributed streaming platform used for building real-time data pipelines and streaming applications.
- **Google Cloud Pub/Sub:** A fully-managed real-time messaging service that allows for asynchronous communication between applications.
- **Google Cloud Dataflow:** A fully managed stream and batch data processing service used for ETL (Extract, Transform, Load) operations.
- **Google Cloud Storage:** A scalable, durable object storage service that acts as a data lake for raw and processed data.

- **Google BigQuery:** A fully-managed, serverless data warehouse that enables super-fast SQL queries using the processing power of Google's infrastructure.
- **Elastic Cloud:** A hosted Elasticsearch service that stores and indexes data for real-time analysis.
- **Kibana:** A data visualization dashboard for Elasticsearch, used for monitoring and analyzing the data in real-time.
- **Power BI:** A business analytics service by Microsoft for interactive visualizations and business intelligence capabilities.

4.2.2 Data Flow

The data flows through the system as follows:

1. Data is ingested from the API into Kafka, which is running on-premises.
2. Kafka streams the data to Google Cloud Pub/Sub for further processing in the cloud.
3. From Pub/Sub, the data is sent in two parallel paths:
    - Directly to Google Cloud Dataflow for real-time processing.
    - To Google Cloud Storage for storage and potential batch processing.
4. Google Cloud Dataflow processes the streaming data and performs necessary transformations (ETL).
5. After processing, Dataflow sends the data to two destinations:
    - To Elasticsearch (Elastic Cloud) for real-time indexing and querying.
    - To BigQuery for long-term storage and complex analytical queries.
6. Data in Elasticsearch can be visualized and monitored using Kibana.
7. Power BI connects to BigQuery to create reports and dashboards for in-depth analysis.

4.2.3 Diagram

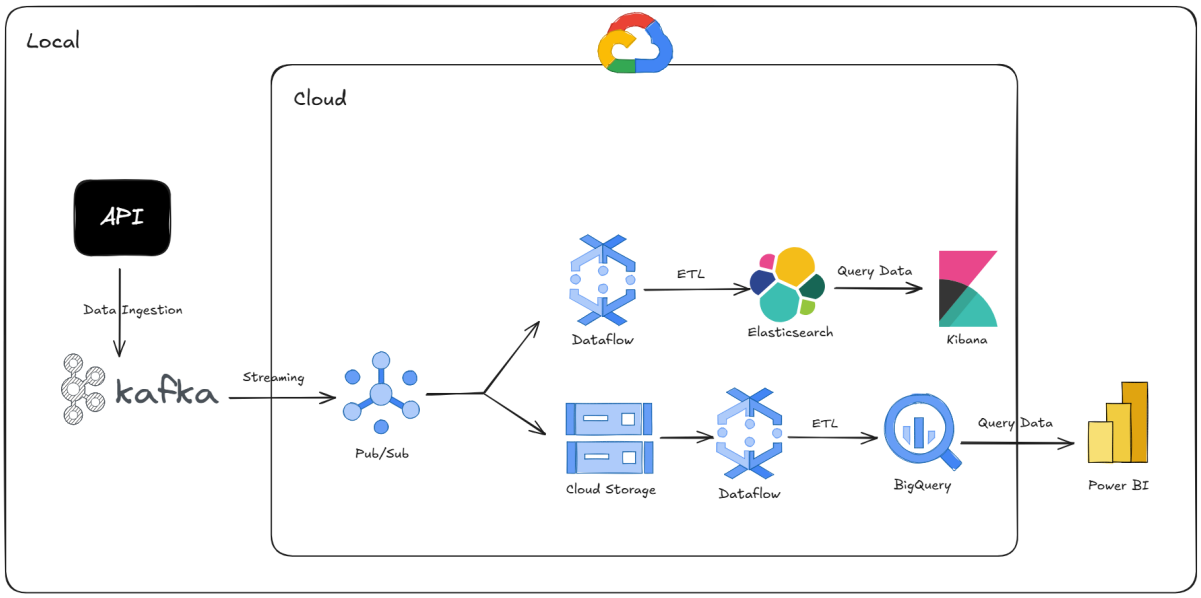

Figure 4.1: Architecture Diagram of the IoT Real-Time Water Quality Monitoring Project (Cloud)

This hybrid cloud architecture combines the strengths of on-premises systems with cloud-based services. It allows for real-time data ingestion and processing through Kafka, Pub/-Sub, and Dataflow, while providing both real-time analytics capabilities via Elasticsearch and Kibana, and long-term data warehousing and analysis through BigQuery and Power BI. The use of Cloud Storage as a parallel path ensures data durability and enables potential batch processing capabilities alongside real-time streaming, offering a flexible and robust solution for water quality monitoring.

4.3 Local Architecture

The local architecture is hosted on-premise, leveraging various technologies for real-time data ingestion, processing, storage, and visualization. This architecture enables both real-time monitoring and long-term analysis of water quality data.

4.3.1 Components in Local Architecture

The major components of the local architecture include:

- **API:** The entry point for data ingestion, the Chicago Park District API.
- **Kafka:** A distributed streaming platform used for building real-time data pipelines and streaming applications.
- **Apache Spark:** A unified analytics engine for large-scale data processing, used here for ETL (Extract, Transform, Load) operations.

- **Elasticsearch:** A distributed search and analytics engine used for real-time data indexing and querying.
- **Kibana:** A data visualization dashboard for Elasticsearch, used for monitoring and analyzing the data in real-time.
- **Staging Database:** An intermediate storage for processed data before it's moved to the data warehouse.
- **SSIS (SQL Server Integration Services):** A platform for building enterprise-level data integration and data transformations solutions.
- **Data Warehouse:** A system used for reporting and data analysis of integrated data from one or more disparate sources.
- **Power BI:** A business analytics service by Microsoft for interactive visualizations and business intelligence capabilities.

4.3.2 Data Flow

The data flows through the system as follows:

1. Data is ingested from the API into Kafka.
2. Kafka streams the data to Apache Spark for real-time processing.
3. Spark performs ETL operations and sends the processed data in two directions:
    - To Elasticsearch for real-time indexing and querying.
    - To a Staging database for temporary storage.
4. Data in Elasticsearch can be queried and visualized using Kibana.
5. SSIS extracts data from the Staging database, potentially transforms it further, and loads it into the Data Warehouse.
6. Power BI connects to the Data Warehouse to create reports and dashboards for in-depth analysis.

4.3.3 Diagram

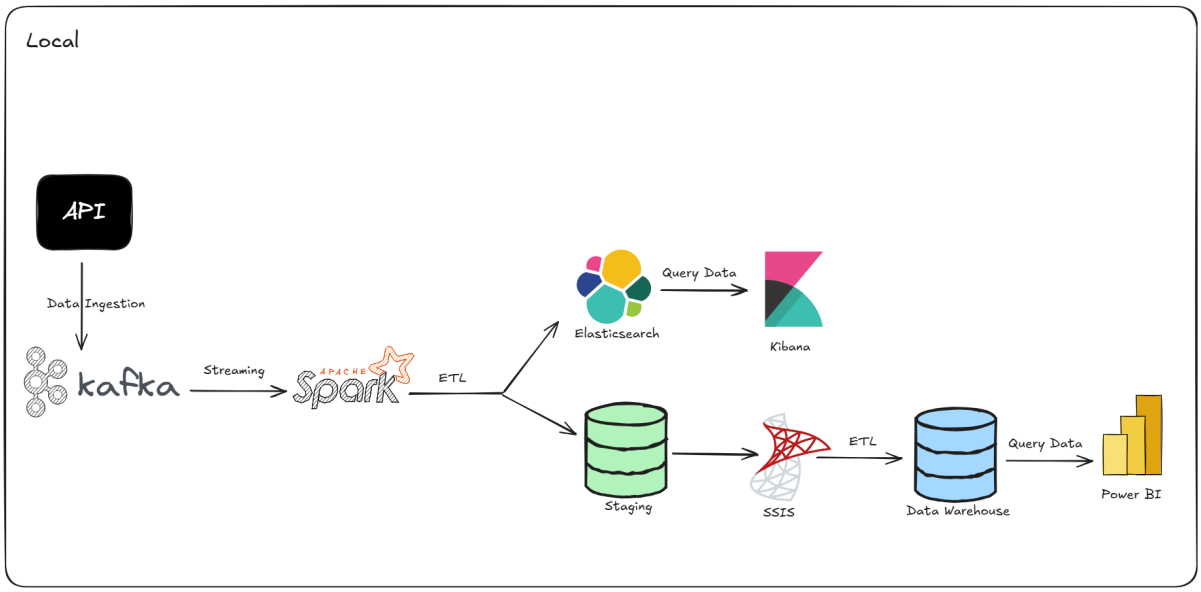

Figure 4.2: Architecture Diagram of the IoT Real-Time Water Quality Monitoring Project (Local)

This architecture allows for both real-time monitoring through the Elasticsearch-Kibana pipeline and long-term trend analysis through the Data Warehouse-Power BI pipeline. The use of Kafka and Spark ensures scalability and real-time processing capabilities, while the combination of staging and data warehouse allows for data integration and complex analytical queries.

Chapter 5

Implementation and Deployment

This chapter focuses on the practical steps and configurations required to deploy both the cloud and local implementations of the IoT Real-Time Water Quality Monitoring project. For a detailed explanation of the system's architecture and components, please refer to Chapter 4.

5.1 Cloud-Based Implementation

The cloud-based implementation uses Kafka to stream data from the Socrata API and integrates with Google Cloud Pub/Sub, Dataflow, Elastic Cloud (Elasticsearch), BigQuery, and Power BI. Below are the steps involved in setting up and running the cloud-based pipeline.

5.1.1 Kafka and Pub/Sub Setup

Kafka Installation

Download and install 'kafka_2.12-3.8.0' on WSL. After unzipping the directory, create the necessary subdirectories 'plugins' and 'connect'.

Listing 5.1: Kafka Installation Commands

```
tar -xzf kafka_2.12-3.8.0.tgz
cd kafka_2.12-3.8.0
mkdir plugins
mkdir connect
```

Pub/Sub Connector Installation

Place 'pubsub-group-kafka-connector-1.2.0' inside the 'plugins' folder and create 'cps-sink-connector.properties' in the 'connect' directory.

Listing 5.2: Configuration for Pub/Sub Connector

```
name=CPSSinkConnector
connector.class=com.google.pubsub.kafka.sink.
   CloudPubSubSinkConnector
tasks.max=10
key.converter=org.apache.kafka.connect.storage.StringConverter
```

```
value.converter=org.apache.kafka.connect.converters.
   ByteArrayConverter
topics=water-quality
cps.project=iti-gp-434315
cps.topic=gcp-water-quality
gcp.credentials.file.path=connect/iti-gp-434315-750e14809a55.json
```

5.1.2 Pub/Sub Configuration

The Pub/Sub topic 'gcp-water-quality' is created to receive data from Kafka. Two subscriptions are created:

- **gcp-water-quality**: Sends data to Google Dataflow and Elasticsearch.
- **waterquality_Cloud_storage**: Sends data to Google Cloud Storage.

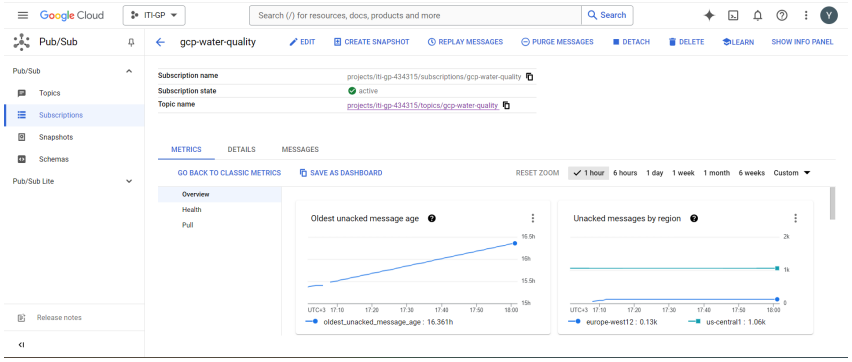

Figure 5.1: Pub/Sub and Dataflow Integration

5.1.3 Dataflow Configuration

The Dataflow job is configured using the 'PubSub_to_Elasticsearch_Xlang' template. Below is the configuration used for launching the job:

Listing 5.3: Dataflow Job Configuration

```
{
    "launch_parameter": {
        "jobName": "from-pubsub-to-elk",
        "containerSpecGcsPath": "gs://dataflow-templates-us-
           central1/latest/flex/PubSub_to_Elasticsearch_Xlang",
        "parameters": {
            "inputSubscription": "projects/iti-gp-434315/
               subscriptions/gcp-water-quality",
            "connectionUrl": "elk-water-quality:<encrypted-
               connection-url>",
            "apiKey": "<encrypted-api-key>",
            "batchSize": "1000",
            "errorOutputTopic": "projects/iti-gp-434315/topics/
               dead-letter-queue"
```

```
        },
        "environment": {
            "numWorkers": 2,
            "tempLocation": "gs://dataflow-staging-us-central1
               -124626422513/tmp",
            "enableStreamingEngine": true
        }
    }
}
```

Transformation Script for Dataflow

Before sending data to Elasticsearch, the following Python transformation script is used to clean and standardize the data:

Listing 5.4: Dataflow Transformation Script

```
import json
from datetime import datetime

def process(value):
    data = json.loads(value)
    transformed_data = {}
    transformed_data['beach_name'] = data.get('Beach␣Name', '').
       strip()
    timestamp_str = data.get('Measurement␣Timestamp', '')
    try:
        transformed_data['measurement_timestamp'] = datetime.
           strptime(timestamp_str, "%Y-%m-%dT%H:%M:%S.%f").
           isoformat()
    except ValueError:
        transformed_data['measurement_timestamp'] = None
    transformed_data['water_temperature_celsius'] = float(data.
       get('Water␣Temperature', None))
    transformed_data['battery_status'] = 'low' if
       transformed_data.get('battery_life_voltage', 0) < 12 else
       'normal'
    return json.dumps(transformed_data)
```

5.1.4 Elastic Cloud Setup

Elastic Cloud is provisioned through Google Cloud Marketplace. The deployment is created with the following parameters:

- **Version**: 8.15.1
- **Region**: GCP Iowa (us-central1)
- **Autoscaling**: Disabled
- **Cluster Topology**: Contains hot, warm, cold, and frozen nodes for optimized storage.

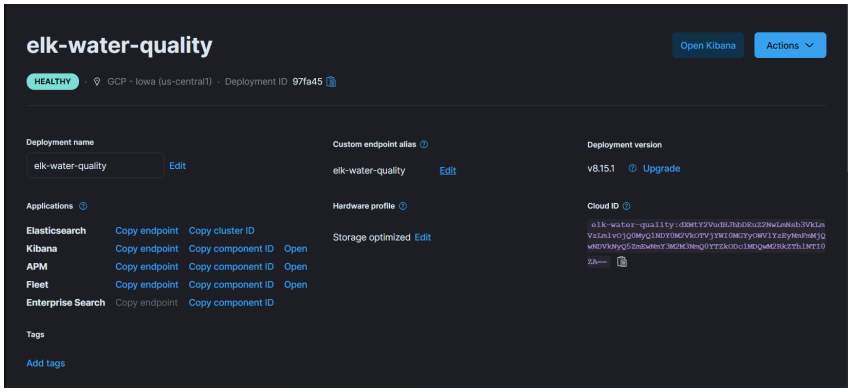

Figure 5.2: Elastic Cloud Deployment Configuration

5.1.5 Kibana Dashboard Setup

A custom Kibana dashboard, "IoT Real-Time Water Quality Monitoring", was created to visualize the data ingested into Elasticsearch from the Pub/Sub pipeline. The dashboard includes time-series graphs, heatmaps, and geospatial visualizations.

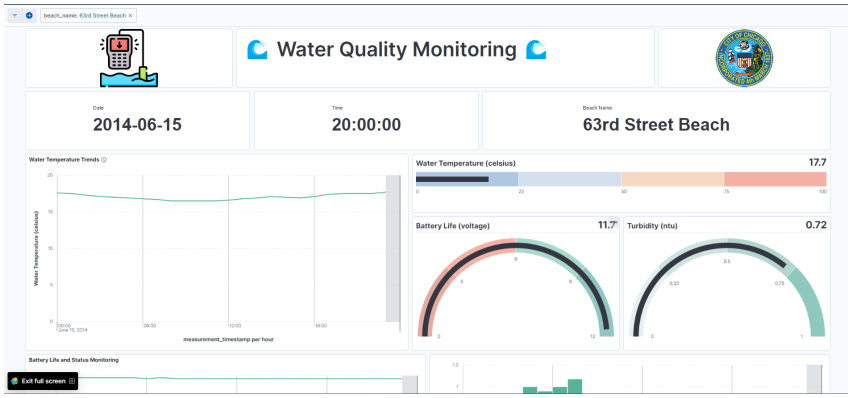

Figure 5.3: Kibana Dashboard - IoT Real-Time Water Quality Monitoring

5.1.6 Cloud Storage to BigQuery Integration

Cloud Storage Setup

A Google Cloud Storage bucket called 'waterdatastore' was created to store the data processed by the Pub/Sub pipeline. The details of the bucket are:

- Bucket name: waterdatastore
- Location: US (multi-region)
- Storage class: Standard
- Soft delete policy: 7 days

Dataflow Job Configuration for BigQuery

A second Dataflow job was configured to stream data from Cloud Storage to BigQuery, using the following configuration:

Listing 5.5: Dataflow Job Configuration (Cloud Storage to BigQuery)

```
{
    "launch_parameter": {
        "jobName": "cloudstoragetobigquary",
        "containerSpecGcsPath": "gs://dataflow-templates-us-
           central1/latest/flex/Stream_GCS_Text_to_BigQuery_Xlang
           ",
        "parameters": {
            "inputFilePattern": "gs://waterdatastore/*",
            "outputTable": "iti-gp-434315:water_quality_data.
               newtable",
            "bigQueryLoadingTemporaryDirectory": "gs://dataflow-
               staging-us-central1-124626422513",
            "pythonExternalTextTransformGcsPath": "gs://
               bigquarytransformations/transformation.py",
            "stagingLocation": "gs://dataflow-staging-us-central1
               -124626422513/staging"
        },
        "environment": {
            "numWorkers": 2,
            "tempLocation": "gs://dataflow-staging-us-central1
               -124626422513/tmp",
            "enableStreamingEngine": true
        }
    }
}
```

BigQuery Schema and Setup

A BigQuery dataset ‘water_quality_data‘ was created, and the following schema was defined for the water quality data:

- beach_name: STRING, NULLABLE
- measurement_timestamp: DATETIME, NULLABLE
- water_temperature_celsius: FLOAT, NULLABLE
- turbidity_ntu: FLOAT, NULLABLE
- transducer_depth_meters: FLOAT, NULLABLE
- wave_height_meters: FLOAT, NULLABLE
- wave_period_seconds: INTEGER, NULLABLE
- battery_life_voltage: FLOAT, NULLABLE

- measurement_id: STRING, NULLABLE
- battery_status: STRING, NULLABLE

5.1.7 Power BI Integration

The final step involves connecting Power BI to BigQuery for visualizing the water quality data. Power BI uses the BigQuery connector to pull data from the 'water_quality_data' dataset and generate interactive reports and dashboards.

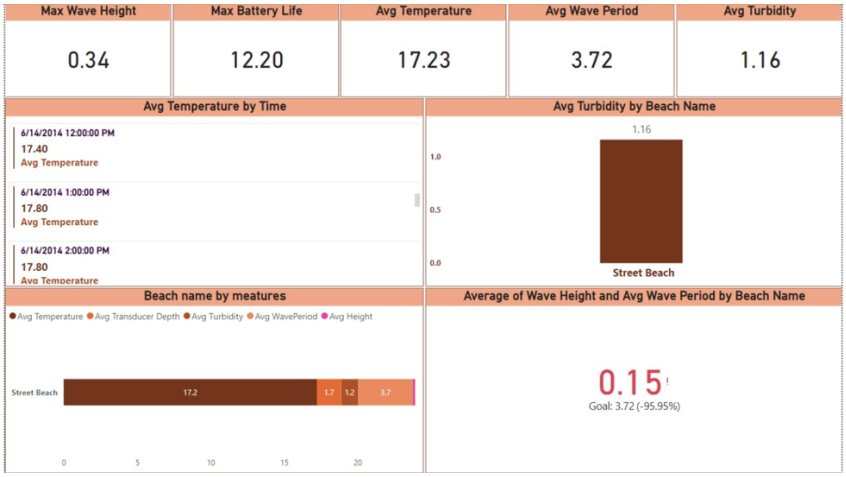

Figure 5.4: Power BI Dashboard for Water Quality Data

5.2 Local-Based Implementation

The local implementation focuses on processing water quality data using Apache Kafka and Apache Spark for real-time streaming and transformations, then storing it in SQL Server for structured analysis or Elasticsearch for real-time analytics.

5.2.1 Tools and Libraries Used

- **Apache Kafka 2.12**: Used as the source of real-time streaming data.
- **Apache Spark 4.2**: Utilized for real-time data streaming processing and transformations.
- **SQL Server (via pyodbc)**: The target database for structured storage of water quality data.
- **Elasticsearch 8.15.1**: A search and analytics engine used for indexing and searching the data.
- **Pandas**: Handles Spark DataFrames in Python for insertion into SQL Server.

5.2.2 Kafka Stream Setup

Water quality data is ingested from a Kafka topic named ‘water-quality‘. The data structure consists of several fields, including:

- Beach Name
- Measurement Timestamp
- Water Temperature (Celsius)
- Turbidity (NTU)
- Transducer Depth (meters)
- Wave Height (meters)
- Wave Period (seconds)
- Battery Life (voltage)
- Measurement ID

Initializing the Spark Stream

The Spark session is initialized to handle data streaming and processing. The connection to the Kafka topic is set up using the following configuration:

Listing 5.6: Kafka Stream Setup

```
df_kafka = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "water-quality") \
    .option("startingOffsets", "latest") \
    .load()
```

The raw Kafka message is extracted and cast to a ‘STRING‘, and then parsed into a structured format using the schema defined in Spark SQL.

5.2.3 Data Transformation

Various transformations are applied to clean and prepare the data for further storage or indexing. This includes:

- **String Trimming**: The ‘Beach Name‘ and ‘Measurement ID‘ are trimmed.
- **Type Casting**: String values are cast to their appropriate data types (e.g., float for water temperature, integer for wave period).
- **Derived Fields**: A ‘battery_status‘ field is calculated based on the battery voltage.

Listing 5.7: Transformation with Derived Fields

```
df_transformed = df_parsed.withColumn(
    "battery_status",
    when(col("battery_life_voltage") < 12, "low").otherwise("
       normal")
)
```

5.2.4 Target Systems

The transformed data is stored either in SQL Server or Elasticsearch for further analysis and real-time analytics.

SQL Server (water_consumer_wh.py)

In this script, the transformed data is stored in SQL Server for structured data analysis. The data from the Spark DataFrame is first converted to a Pandas DataFrame and then inserted into the 'water_quality_data' table in SQL Server.

Listing 5.8: Writing to SQL Server

```
def write_to_sqlserver(df):
    conn = pyodbc.connect(
        "DRIVER={ODBC␣Driver␣17␣for␣SQL␣Server};"
        "SERVER=192.168.1.23;"
        "DATABASE=water_staging;"
        "UID=sa;"
        "PWD=root"
    )
    pandas_df = df.toPandas()
    cursor = conn.cursor()
    for index, row in pandas_df.iterrows():
        cursor.execute("""
            INSERT INTO water_quality_data (beach_name,
               measurement_timestamp, water_temperature_celsius,
            turbidity_ntu, transducer_depth_meters,
               wave_height_meters, wave_period_seconds,
            battery_life_voltage, measurement_id, battery_status)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
        """, row['beach_name'], row['measurement_timestamp'], row
           ['water_temperature_celsius'],
            row['turbidity_ntu'], row['transducer_depth_meters'
               ], row['wave_height_meters'],
            row['wave_period_seconds'], row['
               battery_life_voltage'], row['measurement_id'],
            row['battery_status'])
    conn.commit()
    cursor.close()
    conn.close()
```

Elasticsearch (water_consumer_elk.py)

For Elasticsearch, the data is indexed for real-time analytics and searchability. The Elasticsearch Python client is used to create and index a document for each record in the 'water-quality' index.

Listing 5.9: Indexing to Elasticsearch

```
class ElasticsearchWriter:
    def __init__(self, es_host, index_name):
        self.es_host = es_host
        self.index_name = index_name
        self.es = None

    def open(self, partition_id, epoch_id):
        self.es = Elasticsearch(self.es_host)
        return True

    def process(self, row):
        doc = row.asDict()
        try:
            self.es.index(
                index=self.index_name,
                id=doc["measurement_id"],
                document=doc,
                pipeline="add_timestamp"
            )
        except Exception as e:
            print(f"Error indexing document: {e}")

    def close(self, error):
        if self.es:
            self.es.close()
```

5.2.5 Data Streaming Configuration

Both consumers (for SQL Server and Elasticsearch) write data in append mode, and the stream runs continuously, awaiting incoming data from Kafka.

Listing 5.10: Streaming Data to Target Systems

```
df_final.writeStream \
    .foreachBatch(lambda df, batchId: write_to_sqlserver(df)) \
    .outputMode("append") \
    .start() \
    .awaitTermination()
```

In the case of Elasticsearch, the following code is used for stream configuration:

Listing 5.11: Streaming Data to Elasticsearch

```
df_final.writeStream \
    .foreach(ElasticsearchWriter("localhost:9200", "water-quality
       ")) \
```

```
    .outputMode("append") \
    .start() \
    .awaitTermination()
```

5.2.6 Database Design and Staging

The data is staged in SQL Server with a full schema, including fields for beach information, water measurements, and battery status. A separate date dimension table (‘date_dim‘) is created for date-specific data, which helps with analysis and joins with the fact table.

Date Dimension Table

The ‘date_dim‘ table stores date components (year, month, day, etc.) and is linked to the fact table using a surrogate key (‘timestamp_id‘).

Listing 5.12: Date Dimension Schema

```
CREATE TABLE date_dim (
    timestamp_id INT PRIMARY KEY,
    measurement_timestamp DATE,
    year INT,
    month INT,
    day_of_week INT,
    day INT
);
```

Fact Table for Water Quality Measurements

The ‘fact_water_quality_measurements‘ table stores the detailed water quality data with foreign key relationships to the beach and date dimension tables.

Listing 5.13: Fact Table Schema

```
CREATE TABLE fact_water_quality_measurements (
    measurement_id INT PRIMARY KEY,
    beach_id INT,
    timestamp_id INT,
    water_temperature_celsius FLOAT,
    turbidity_ntu FLOAT,
    transducer_depth_meters FLOAT,
    wave_height_meters FLOAT,
    wave_period_seconds INT,
    battery_life_voltage FLOAT,
    battery_status VARCHAR(10),
    FOREIGN KEY (beach_id) REFERENCES beach_dim(beach_id),
    FOREIGN KEY (timestamp_id) REFERENCES date_dim(timestamp_id)
);
```

5.2.7 SSIS Package for ETL

SQL Server Integration Services (SSIS) is used for extracting data from the staging table and loading it into the fact table. Derived columns are used to transform the data, and lookups are implemented to match the beach and timestamp dimensions.

—

This concludes the local-based implementation for real-time water quality monitoring using Kafka, Spark, SQL Server, and Elasticsearch.

—

Chapter 6

Challenges and Future Work

6.1 Challenges

6.1.1 Cloud Implementation Challenges

- **Cost Management**: Managing costs when utilizing cloud services like Google Cloud Pub/Sub and Dataflow is critical, especially with high data volumes. Monitoring usage and optimizing resource allocation are necessary to prevent unforeseen expenses.
- **Integration Complexity**: Integrating multiple cloud services (e.g., Pub/Sub, Dataflow, Elasticsearch) demands significant configuration and troubleshooting effort.
- **Latency Issues**: While scalable, cloud services may suffer from network latency, which can impede real-time data processing, particularly when handling multiple data sources.
- **Setup Complexity**: The initial configuration of GCP services posed significant hurdles, requiring extensive research and testing to ensure a functional environment.
- **Log Management**: Tracking and managing logs across cloud services to ensure system health and diagnose issues remains a challenging but necessary task.

6.1.2 Local Implementation Challenges

- **Resource Limitations**: Running Kafka and related components locally can strain system resources, particularly when handling large data volumes or many concurrent connections, hindering scalability.
- **Data Consistency**: Ensuring data consistency across Kafka and its sinks can be problematic, particularly during network disruptions or system failures.
- **Complex Configuration**: Managing configurations for different environments (e.g., development, testing, production) introduces risks and inefficiencies.
- **Monitoring and Debugging Limitations**: The lack of robust monitoring tools in local setups makes identifying and resolving issues more difficult.

- **Data Transformation Challenges**: Dealing with invalid or missing data during transformations required careful exception handling using PySpark to maintain data quality.
- **Compatibility Issues**: Integration challenges with Spark and the Elasticsearch-Hadoop connector necessitated switching to the `elasticsearch-py` library for smoother operations.

6.2 Future Work

6.2.1 Cloud Implementation Enhancements

- **Cost Optimization**: Explore strategies for optimizing cloud expenses, such as reserved instances, preemptible VMs, and regular billing analysis to pinpoint potential savings.
- **Adopting Serverless Architecture**: Investigate shifting certain components to a serverless architecture (e.g., Cloud Functions for data stream processing) to streamline management and reduce costs.
- **Advanced Security**: Implement stronger security protocols, including encryption for both data at rest and in transit, and conduct regular audits to ensure compliance with best practices.
- **Scalability Improvements**: Perform load testing to gauge system limits and implement auto-scaling strategies for components like Dataflow and Pub/Sub to accommodate variable workloads.

6.2.2 Local Implementation Enhancements

- **Performance Tuning**: Optimize Kafka's performance through techniques such as adjusting topic partitions or fine-tuning consumer settings for better resource utilization.
- **Improved Monitoring**: Introduce tools like Prometheus and Grafana for enhanced metric visualization and more effective troubleshooting in local environments.
- **Automated Configuration Management**: Leverage tools like Ansible or Terraform to standardize and automate configuration management across environments, reducing manual intervention and errors.

Chapter 7

Conclusion

7.1 Summary of Work

This project developed an IoT-based Real-Time Water Quality Monitoring system for Chicago Park District beaches along Lake Michigan. By implementing both cloud-based and on-premise architectures, the system successfully ingests, processes, stores, and visualizes water quality data in real time, while also supporting long-term trend analysis.

The real-time pipeline delivers immediate insights into critical water quality metrics, enabling quick decision-making to ensure public safety. The long-term analysis pipeline provides valuable data for researchers and policymakers to analyze environmental changes over time.

7.2 Key Achievements

- **Dual Deployment**: The system was deployed in both cloud and on-premise environments, showcasing its adaptability across different infrastructure setups.
- **Real-Time Data Processing**: Utilizing technologies like Apache Kafka, Google Cloud Pub/Sub, and Apache Spark, the system achieves real-time data ingestion and processing.
- **Interactive Data Visualization**: Powerful visualization tools, such as Kibana and Power BI, were used to create dashboards that facilitate both real-time monitoring and historical analysis.
- **Scalable Architecture**: Designed an architecture that scales with data growth, leveraging cloud elasticity and local resources for cost-effectiveness.

7.3 Overcoming Challenges

Challenges such as integration complexity, resource constraints, and data consistency issues were encountered and addressed, strengthening both problem-solving skills and the system's overall resilience.

7.4 Impact and Future Directions

The system's real-time monitoring capabilities have a direct impact on public safety by providing authorities with actionable data. The long-term data collection supports environmental research and policy development. Future enhancements will focus on optimizing costs, improving security, and exploring machine learning applications for predictive insights. Expanding the system to monitor additional environmental parameters or extending its deployment to other locations will further increase its utility.

7.5 Final Thoughts

This project highlights the potential of IoT and big data technologies in tackling environmental challenges. By combining advanced data processing frameworks with robust visualization tools, the system delivers actionable insights that benefit communities and drive informed decision-making.

Chapter 8

References

1. City of Chicago. Beach Water Quality Automated Sensors. https://data.cityofchicago.org/Parks-Recreation/Beach-Water-Quality-Automated-Sensors/qmqz-2xku/about_data
2. Socrata. API Documentation. https://dev.socrata.com/foundry/data.cityofchicago.org/qmqz-2xku
3. Google APIs. Pub/Sub Kafka Connector Documentation. https://github.com/googleapis/java-pubsub-group-kafka-connector/?tab=readme-ov-file
4. Apache Kafka. Kafka Connect Documentation. https://kafka.apache.org/documentation.html#connectconfigs
5. Google Cloud. Dataflow Template: Pub/Sub to Elasticsearch. https://cloud.google.com/dataflow/docs/guides/templates/provided/pubsub-to-elasticsearch
6. Elastic. Ingest Data Directly from Google Pub/Sub into Elastic Using Google Dataflow. https://www.elastic.co/blog/ingest-data-directly-from-google-pub-sub-into-elastic-using-google-dataflow
7. Apache Spark. https://spark.apache.org/