Original Scientific Paper
Received: 2022-03-07
Accepted: 2022-04-10

MICROSERVICE ARCHITECTURE FOR SCALABLE IoT PLATFORMS

Daniela Kunej, RIT Croatia, dnk4284@g.rit.edu

Abstract

Internet of Things (IoT) is the next step of the technological evolution in a world of constant technology enhancements. The concept of IoT describes the shift of communication over the Internet, from human to machine, to machine to machine(M2M). These machines together form a large network of sensors sending vast amounts of data. Just like any new technology, the evolution of the Internet of Things requires adaptations and adjustments as time shows the issues the technology encounters. In this paper, different architectures of IoT platforms are presented and analyzed, and finally, a custom solution for an architecture of an IoT platform is suggested using microservices to solve scalability issues that often arise in fast-growing environments like that of the IoT.

Introduction

The last few decades have caused a constant enhancement of technology, bringing in revolutionary ideas and life-altering applications of the technology available. The Internet of Things(IoT) is the next step of the technological evolution. IoT is the concept of connecting a vast variety of things to the Internet, ranging from smartphones and televisions to simple house appliances like air conditioners and coffee machines. For example, the global company Samsung presents smart devices including smartphones, smart TVs, smart watches, air conditioners, and many more creating their own concept of “Smart Things” as described on their website: https://www.samsung.com/us/smartthings/.

The concept of IoT describes the shift of communication over the Internet, from human to machine, to machine to machine(M2M). These machines together form a large network of sensors sending vast amounts of data. Latest-generation sensors are producing an almost continual stream of high-dimensional data creating new challenges (Buyya et al., 2016).

Just like any new technology, the evolution of the Internet of Things requires adaptations and adjustments as time shows the issues the technology encounters. There is no unified architecture that is agreed upon to be the best for such platforms and varies greatly depending on the requirements and possibilities of those building it. However, one main takeaway everyone can agree upon is that it is extremely challenging to develop a system architecture that is able to follow the growth of the IoT domain, and what becomes more and more evident, is that the amount of data keeps increasing exponentially, making scalability one of the most important features of the platform being built.

All IoT architectures encounter different challenges, including:

- Security challenges
- Connectivity challenges
- Compatibility challenges
- Scalability challenges

As IoT becomes a ubiquitous part of our everyday surroundings, handling multiple aspects of our lives and holding an increasing amount of our data in constant observation, the importance of the architecture becomes more evident, and sparks numerous proposals of solutions to these challenges.

To develop an architecture that presents a good solution for this type of platform, different architectures of IoT platforms are presented and analyzed, and finally, a proposal of a solution for the architecture of an IoT platform is suggested using microservices in an attempt to solve issues which often arise in fast-growing environments like that of the IoT.

This work consists of:

- A brief description of the Internet of Things and explanation of its importance
- Analysis of existing IoT platforms and their advantages and drawbacks
- A custom proposal of an IoT platform with detailed explanations and elaborations of design and technology choices

Brief Description of the Internet of Things

Definition

“The Internet of things, or IoT, is a system of interrelated computing devices, mechanical and digital machines, objects, animals or people that are provided with unique identifiers (UIDs) and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction” (Gillis, 2021).

Or even more simply put by the IT department of Shrimati Kashibai Navale College of Engineering in their paper published in IARJSET(International Advanced Research Journal in Science, Engineering and Technology) “Introduction to IoT”:

“The phrase Internet of Things (IoT) refers to connecting various physical devices and objects throughout the world via the Internet” (Gokhale, Bhat, 2018).

From these definitions, we can see that the key concept of IoT is the connectivity of any sort of device to the Internet as well as their ability to constantly send data and/or measurements over a network. The part that the Internet plays in our lives has therefore shifted towards integrating machine-to-machine communication to provide a virtual environment that seamlessly incorporates into our lives.

The term “Internet of Things” was coined in 1999. by Kevin Ashton who used it in a supply-chain management presentation to highlight the importance of the “things”, we interact with every day as technology advances. Needless to say, since then, this statement has become immensely more accurate, with its accuracy still increasing every day.

Historically, Radio-Frequency Identification or RFID was the dominant technology in IoT, more recently replaced with Wireless networks (WSN) and Bluetooth-enabled devices which are previously researched and analyzed extensively. However, the unique requirements of IoT, such as scalability, heterogeneity support, integration, and data stream processing are given less attention and are still very open for research and advancement suggestions.

How it works

An IoT environment is built up by multiple smart devices, with access to the internet, in embedded systems that are used to collect, send, and perform actions on the data that they collect from their surroundings. These devices collect the data that they are configured to collect, and by connecting to an IoT gateway send the data to an endpoint at which the data is analyzed, aggregated, or manipulated for further software solutions depending on the requirements of the clients.

Importance

The obvious importance of the Internet of Things is that of our everyday lives – it helps people live and work smarter, giving them complete control over multiple aspects of their lives previously unimaginable. They analyze and manipulate their home's security, temperature, air humidity, and many other factors simply using their smartphones whenever they need, and wherever they are, as long as they are connected to the Internet, which, we can all agree, has already become a must-have at all times. Recently, these types of sensors were extensively researched and described by John R. Delaney and Alex Colon in their article "The Best Smart Home Security Systems for 2022", where they state that users can: "remotely control your door locks, lights, thermostats, vacuums, lawn mowers, and even pet feeders, using your smartphone and an app." (Delaney, Colon, 2022).

However, today, business is what makes the world turn, and IoT has also become essential to business. Therefore, many companies, and especially corporations, include IoT in their way of working, as the profitability of IoT for their business becomes undeniable. IoT can be used to monitor their overall business processes, improve the customer experience, save time and money, enhance employee productivity, integrate, and adapt business models, make better business decisions, and generate more revenue. As more and more businesses realize the potential of IoT in their work, IoT will continue to gain momentum. The importance of IoT in business is more extensively described in the article "How Business Processes Are Evolving With The IoT" where it is stated that: "All business processes need to be future-proof and adaptable when dealing with technological advances. Many businesses are already using the Internet of Things (IoT). As the technology becomes more and more integrated into our lives, business processes have to continue to adapt as well as the way these are managed" (Meghamala, 2019).

Challenges

Security challenges

Due to extreme resource restrictions, security is often back-seated for functionality to save battery life and keep processing power necessities to a minimum. Although there is a lack of privacy standards and end-to-end security solutions – "Key Management System (KMS) with a zero-trust network feature and blockchain is rapidly addressing the privacy and trust threats with reinforced security features" (Imran et al., 2020).

Connectivity challenges

Connectivity is an extremely important aspect of IoT as the devices provide data that is valuable if received correctly and continuously- especially in sensors that are meant to monitor process data and supply information. Connectivity needs to be available to all devices at a low cost, and most importantly, needs to be extremely reliable. Otherwise, the data provided to the gateway can be incomplete, invalid, and if not handled properly, can defeat the purpose of IoT completely. Due to the complexity of wireless connectivity, the information from sensors cannot be transferred seamlessly from the cloud, devices, infrastructure and applications, making this one of the biggest challenges of IoT (Ankit, 2021).

Compatibility challenges

Considering that no international standard of compatibility has yet been defined for the field of IoT, devices from different manufacturers are often incompatible and unable to communicate with each other. "IoT is growing in many different directions, with many different technologies competing to become the standard. This will cause difficulties and require the deployment of extra hardware and software when connecting devices" (Banafa, 2017).

Scalability challenges

"The Internet of Things must be scalable in order to accumulate the billions of connected devices that will exist in the next five years" (Lester, 2022). Even with today's technological enhancements, without proper architectural planning of the IoT platform, it will be unable to handle the vast amounts of devices and continuous data stream. Considering the importance of the stability and robustness of the systems, the architecture needs to be scalable to flexibly adapt to any increase in connecting devices and the amount of information it is fed with.

Existing IoT platforms

Google Cloud IoT

One of the world's leading Internet of Things platforms at the moment. The platform connects to hardware from Intel and Microchip automatically and is compatible with several operating systems.

Google Cloud IoT's main features include: AI and machine learning capabilities, data analysis in real-time, data visualization, location tracking.

Core use cases: predictive maintenance, real-time asset tracking, logistics, and supply chains, smart cities, and buildings. Google Cloud IoT is a system with numerous services integrated as one solution.

These services include:

- Cloud IoT Core which is used to collect and manage device data. MQTT and HTTP protocol bridges are utilized for connectivity and communication with the Google Cloud Platform
- Cloud Pub/Sub handles data ingestion and message routing for additional data processing
- Google BigQuery allows for secure real-time data analytics
- AI Platform makes use of machine learning capabilities
- Google Data Studio visualizes data through the creation of reports and dashboards
- The Google Maps Platform aids in visualizing the location of linked items (Janson, 2021).

Cisco IoT Cloud Connect

Cisco IoT Cloud Connect is designed with mobile operators in mind and is the best IoT cloud platform for industrial and individual use cases. Cisco also gives reliable IoT hardware, including switches, access points, routers, gateways, and other devices.
Core features of Cisco IoT Cloud Connect: Powerful industrial solutions, high-level security, edge computing, centralized connectivity, and data management
Core use cases: Connected cars, fleet management, home security and automation, payment solutions, industrial networking, healthcare (Janson, 2021).

Cisco IoT Cloud Connect

Cisco IoT Cloud Connect is designed with mobile operators in mind and is the best IoT cloud platform for industrial and individual use cases. Cisco also gives reliable IoT hardware, including switches, access points, routers, gateways, and other devices.
Core features of Cisco IoT Cloud Connect: Powerful industrial solutions, high-level security, edge computing, centralized connectivity, and data management
Core use cases: Connected cars, fleet management, home security and automation, payment solutions, industrial networking, healthcare (Janson, 2021).

Salesforce IoT Cloud

Salesforce's focus is on customer relationship management and has used IoT solutions to enhance this market segment. The Salesforce IoT Cloud platform provides a personalized experience and companies can understand customer data more accurately, improve the UX/CX, and increase revenue.
Salesforce IoT Cloud core functions:
Complete customer, product, and CRM integration
Rules, condition, and event management through simple UI

Proactively resolve customers' problems and needs.
Core use cases: Government administration, chemicals, machinery, financial services, marketing, and advertising (Janson, 2021).

Custom architecture proposal for scalable IoT platform

For reasons explained in the previous chapters, proper planning, and development of the architecture of the IoT platform is extremely important. Only if this is done properly, can an IoT platform serve its purpose to its full extent and potential.
This chapter therefore focuses on presenting a custom solution of a microservice architecture for scalable IoT platforms.

Architecture: Monolith vs Microservice

Monolith Architecture

The Monolith architecture (shown in Figure 1) for software applications is the more traditional way of designing software systems where the application is built as a single indivisible unit.
The standard components of such an architecture system are the ***user interface*** which is served to the end-user presenting ways of communicating with the system, the ***business layer*** which contains the business logic of the application, and the ***data interface*** which communicates with the database at the lowest level.

All these components are situated within one code base, and when a change is needed, it must be implemented on the entire stack. Because of this, applications built in monolithic architecture lack modularity.

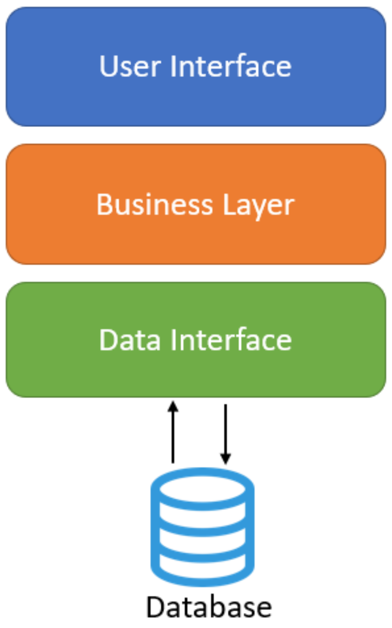


Figure 1: Monolithic Architecture (Gnatyk, 2018)

Pros of Monolithic Architecture

- **Simple to handle** logging, handling, caching, and performance monitoring because the functionalities concern only one application
- **Simple to debug and test**: The end-to-end case is covered by one application
- **Simple to develop and deploy**

Cons of Monolithic Architecture

- **Hard to understand**: When the application begins to scale up in size, the code becomes hard to understand and manage
- **Hard to make changes**: Code changes affect the entire stack, making changes dangerous and hard to orchestrate due to tight coupling
- **Hard to scale up**: Specific components cannot be scaled up separately
- **Hard to introduce new technologies**: If a new technology needs to be introduced, the entire application must be rewritten, instead of just specific code of the component

Microservice Architecture

The Microservice architecture (shown in Figure 2) is a more recent approach to designing software systems. In contrast to a monolithic architecture, microservice architecture divides the single unified unit that is a monolith, into independent units that are each concerned with a single application process. Each service can be managed, changed, deployed, and scaled independently from the rest of the processes of the application (Gnatyk, 2018).

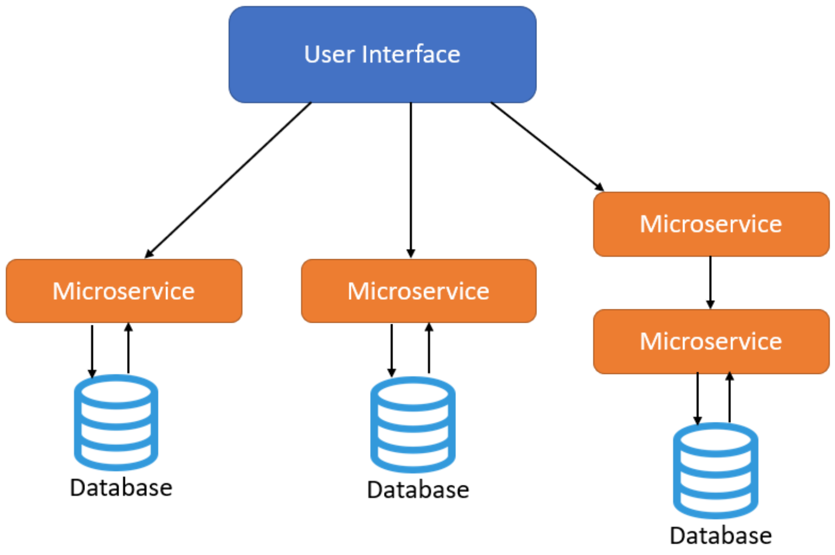


Figure 2: Microservice Architecture(Gnatyk, 2018)

Pros of Microservice Architecture

- **Easy to scale up**: This is considered the biggest benefit of the microservice architecture. Each component can be scaled and is virtually infinite as to how large it can be scaled up to, so fast-growing applications can handle any increase in users or transactions, while a monolithic architecture would be constrained by such growth.

- **Components are independent**: New features are easily added. Each component can be changed and deployed separately, and mistakes only impact one specific process so it is lower risk.
- **Easy to understand**: Smaller components allow easier overview and management - rather than having to analyze the entire application, only the part of interest can be analyzed.
- **Flexibility**: Each component can be developed in any technology, so developers are free to implement new technologies keeping the application more competitive.

Cons of Microservice Architecture

- **Complexity**: Each component and its connections must be handled separately including the deployment process.
- **Cross-cutting concerns**: Any process that affects others parts of the system is harder to maintain; like logging, metrics, auditing, externalized configurations, etc.
- **Testing**: End-to-end tests are much harder to develop in a microservice architecture than in a monolithic architecture.

Custom Microservice Architecture Solution Overview

As technology develops, the monolithic architecture has become less and less popular for large-scale applications. Microservices have taken the lead as applications become fast-paced and fast-growing making agility, scalability, and flexibility more important than ever. This chapter presents a custom solution of microservice architecture for an IoT platform, combining two very important sections of the new era of technology, microservices and IoT. To demonstrate the flexibility and simplicity of integration with the microservice architecture, multiple programming languages are used: Typescript - Node.js and Java - Spring Framework. The proposed architecture is as shown in Figure 3:

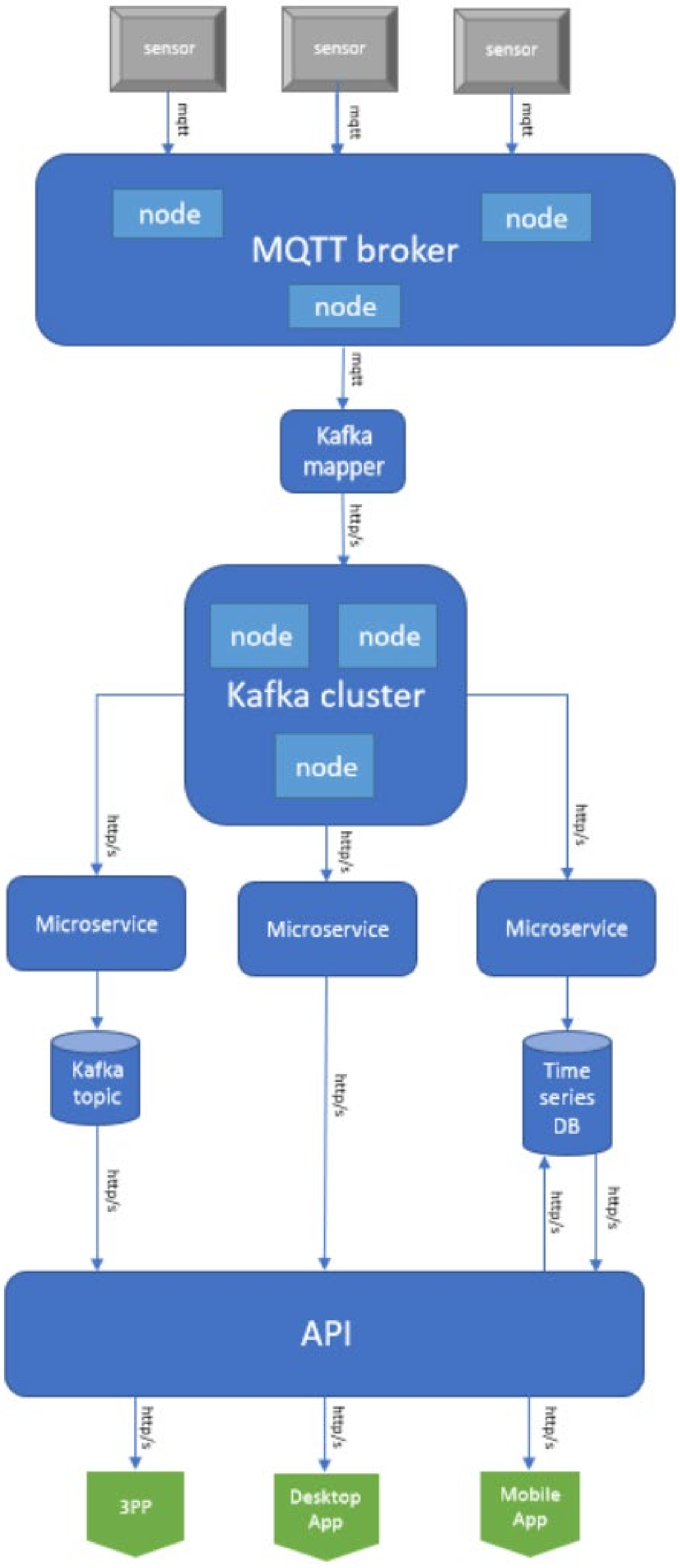


Figure 3: Custom IoT Platform Architecture

Sensors

The first components that can be seen in the system are the sensors that send continuous streams of data to the MQTT broker. They are the initial data sources that send their new data values to the MQTT Broker in specified periods using the MQTT protocol further explained in the following chapter.

MQTT Protocol

As per the official mqtt.org website, the MQTT (Message Queuing Telemetry Transport) protocol is defined as:

"A standard messaging protocol for the Internet of Things (IoT). It is designed as an extremely lightweight publish/subscribe messaging transport that is ideal for connecting remote devices with a small code footprint and minimal network bandwidth" (n.d., 2022).

MQTT is the most widely used protocol within IoT projects. It is designed as a simple protocol that relies on the Publish/Subscribe features to share data between the clients and servers as shown in Figure 4 (Kalyan, n.d.).

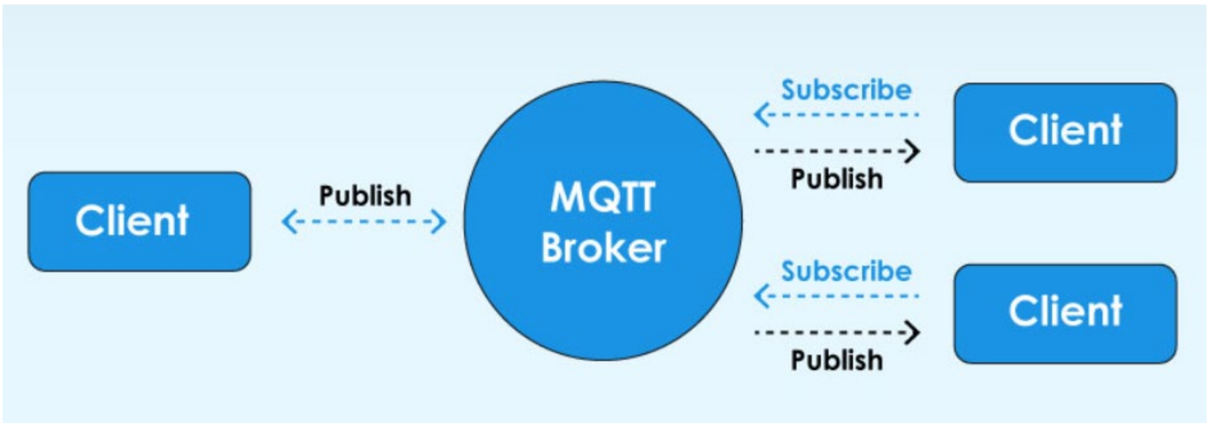


Figure 4: MQTT protocol (Kalyan, n.d.)

There are many reasons why the MQTT protocol is the most competitive solution for IoT and M2M:

- **Lightweight Code Integration**: A few lines of code are enough to integrate with the protocol in almost any programming language.
- **Data Packets Compression**: Extremely resource-efficient in devices with low battery or CPU power. In terms of data amount sent is even more significant for MQTT case, which is larger than for HTTP. But we can compress the size for MQTT with optional additional layer of compression (Ghosh, 2019).
- **Speed**: Real-time, no delays. MQTT Protocol is used to forward the message towards the MQTT broker because according to measurements in 3G networks, throughput of MQTT is 93 times faster than HTTP's (Serozhenko, 2017).
- **Stability**: If a client is unexpectedly disconnected, subscribers can be informed with instructions to fix the issue. The MQTT IoT protocol can transfer data even with unstable connections. It provides three options for Quality of Service (QoS) which is responsible for the message delivery (Petrova, Solovev, 2020).

- **Retained Messages**: Each topic can have a retained message that will automatically be sent to each new client subscribing to the topic. Each client that subscribes to a topic pattern that matches the topic of the retained message receives the retained message immediately after they subscribe. The retained message eliminates the wait for the publishing clients to send the next update (HiveMQ team, 2018).

MQTT protocol is described through its 5 main components:

Broker – server which manages messages between clients
Topic – the destination which the data reached on the broker
Message – the message dispatched to the topic
Publish – the process of sending the message to the broker
Subscribe – the process of subscribing to a client's data

"The MQTT protocol is the standard for all major cloud platforms, including Microsoft Azure, IBM Cloud, and Amazon Web Services. Facebook even uses MQTT for its Facebook Messenger and Instagram apps. For industrial applications, MQTT is especially well-suited for remote monitoring, and its lightweight properties make MQTT one of the most widely used protocols for IoT and IIoT applications" (Collins, 2020).

For these reasons, in the proposed architecture, MQTT is used as the main communication protocol for communicating with external devices.

IoT-Simulator

The IoT-Simulator is a custom-made solution created in order to produce a continuous stream of data. The simulator acts as a group of real life sensors would, sending updated values towards the IoT platform periodically. This way, a real life IoT platform can be simulated as a continual data stream is produced.
This component is developed as a Node application, using Typescript.
To demonstrate the platform solution, five different sensors have been implemented:

1. Temperature sensor – numeric value
2. Barometer – numeric value
3. Humidity sensor – numeric value
4. Motion sensor – boolean value
5. Accelerometer – numeric value

Four instances of each sensor are created, creating a total of 20 clients connecting to the MQTT broker and publishing data to a topic defined for each sensor **"*{clientId}/data*"** in specified intervals.

Implementation example:

```
export class TemperatureSensor implements Sensor {


  //declare variables to hold client id value and mqttConnector object
  private readonly clientId: string
  private readonly mqttConnector: MqttConnector

  //constructor used to instantiate above mentioned variables and connect
   clients to MQTT broker
  constructor() {
    this.clientId = `temperature-${v4()}`
    this.mqttConnector = new MqttConnector(this.clientId)

    this.mqttConnector.client().on('connect', () => {
      return this.publish()
    })
  }

  //publish method to send random numeric value to MQTT broker each 500
   milliseconds.
  public publish<NumericSensorData>(): void {
    setInterval(() => {
      this.mqttConnector.publish(
        SensorData.create({
          id: this.clientId,
          type: SensorType.NUMERIC,
          timestamp: new Date().getTime(),
          value: Math.floor(Math.random() * 100),
        })
      )
    }, 500)
  }
}
```

Protocol Buffers are used to define the format of the message because Protobuf performs better than JSON. Also, MQTT Protocol is used to forward the message towards the MQTT broker because according to measurements in 3G networks, throughput of MQTT is 93 times faster than HTTP's (Serozhenko, 2017).

Protocol Buffers – Protobuf

Protocol buffers were developed by Google to provide a language-neutral, platform-neutral mechanism for serializing data. Similar mechanisms that are more widely known, are XML and JSON. However, Protobuf is smaller, faster, and simpler in comparison. By defining the structure of the data once in a proto schema, special generated source code can be easily integrated for reading and writing the structured data from any data stream and in any programming language.

Protocol Buffers are used to define the format of the message because Protobuf performs up to 6 times faster than JSON (Krebs, 2017).

In this solution, the sensor data message requires fields to represent the client id, the value the sensor holds, what type the sensor is, and the timestamp of when the measurement is last taken.

This format is defined in the SensorData.proto schema:

```
message SensorData {
  required string id = 1;
  required double value = 2;
  required SensorType type = 3;
  required int64 timestamp = 4;

  enum SensorType {
    BOOLEAN = 0;
    NUMERIC = 1;
  }
}
```

This schema defines our message of sensor data to be transmitted. Each field in the message definition has a unique number assigned to it to identify the fields in the message binary format. Each field is marked with the keyword “required” as none of the data in our message can be omitted. The SensorType enum within our schema defines the type of sensor that is sending data and can be Boolean or Numeric.

MQTT Broker

An MQTT broker is the central hub of the functionality of MQTT. It is the intermediary that allows communication between MQTT clients. As previously explained, the MQTT broker receives messages that are published by the clients, creates selections of the messages depending on the topic to which they are published, and then relays them to those subscribed to the mentioned topic. Because of this communication model, MQTT is a highly efficient and scalable protocol. Considering the issue of scalability often encountered in IoT systems, it is an ideal solution as additional nodes of the broker can be configured at any point in time, including after initial deployment, without downtime. This clearly depicts the advantage of both the MQTT protocol, as well as the microservice architecture described in Figure 2.

Generally, two types of MQTT brokers are used:

1. Managed Brokers – this type does not require custom setup to enable MQTT communication and is provided to the user as a service. The most known managed broker service is AWS IoT Core.
2. Self-Hosted Brokers – this type requires custom installation of the broker, which provides the clients with more flexibility, but also requires in-depth knowledge of the system for management and scaling. Open-source implementations of self-hosted brokers include Mosquitto and HiveMQ.

HiveMQ

Considering the goal of this work is to present a custom solution for scalable architecture, HiveMQ is chosen to be implemented within the platform as a self-hosted broker. This broker makes data transfer simple while also ensuring its efficiency, speed, and reliability.

HiveMQ had one of its most successful years yet in 2021 as the automotive industry embraced MQTT and HiveMQ as the standard for connected cars. In 2022 it is expected that more than 50% of cars produced globally will be connected with MQTT and HiveMQ (Götz, 2021).

HiveMQ key features:
Efficient: Transfers data to and from connected clients in an efficient, fast, and reliable manner.
Optimized: Designed to optimize the use of cloud resources by using MQTT to reduce the bandwidth needed for data transfers.
Secure: Connects any device or system safely and reliably using the MQTT protocol.
Fast: Sends and receives data from clients quickly using the Publish/Subscribe push technology.
Scalable: Can scale up to 10 million connected devices with no data loss.
Open: Open API and pre-built extensions allow easy integration to other systems such as Kafka, SQL, and NoSQL databases.

For the implementation within the provided solution architecture, a single HiveMQ node is deployed from a Docker image.
Snippet from docker-compose.yml:

```
hive:
  image: "hivemq/hivemq4"
  ports:
    - 1883:1883
```

Once the data is published to the broker from the connected clients via MQTT, the broker is then ready to serve the data to connected subscribers. According to the proposed architecture, at this point, the data is sent to an Apache Kafka cluster.

Apache Kafka

Apache Kafka is an open-source distributed event streaming platform. It is used for high-performance data streaming, analytics, and data integration.

Apache Kafka has gained great popularity in recent years and is used by many large and well-known global companies including Goldman Sachs, Intuit, and Cisco.

The Kafka cluster is the component of Apache Kafka that stores data streams. Data streams are sequences of messages that are produced by other applications to be stored within the cluster. Once the data is stored, it can be sequentially consumed by other applications. The number of other applications that can consume the data is virtually limitless, as separate consumer groups can be created and the data received on specific topics can be dispatched to all mentioned groups (Sax, 2018).

In the proposed architecture, all data is stored on one topic, to further demonstrate the possibility of multiple microservices consuming messages from the same topic. Once this is demonstrated, it becomes clear that the number of microservices connecting to the Kafka cluster is also limitless, as by adding Kafka nodes, the cluster can handle any number of additional clients connecting.

To achieve the data transfer between the MQTT broker and the Kafka cluster, an additional component is created to map the messages served by the MQTT broker. Similar to HiveMQ, the Kafka server is deployed using a docker image.
Snippet from docker-compose.yml:

```
kafka:
  image: confluentinc/cp-kafka:latest
  container_name: kafka
  ports:
    - "9092:9092"
  depends_on:
    - zookeeper
  environment:
    KAFKA_BROKER_ID: 1
    KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
    KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
              PLAINTEXT:PLAINTEXT,PLAINTEXT_INTERNAL:PLAINTEXT
    KAFKA_ADVERTISED_LISTENERS:
              PLAINTEXT://localhost:9092,PLAINTEXT_INTERNAL://broker:29092
    KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
    KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
```

HiveMQ-Kafka Mapper

This component is a custom developed application implemented as a Spring application and main functionality consists of an MQTT Consumer and a Kafka Publisher.
The MQTT Consumer contains the code for creating an MQTTClient which connects to HiveMQ and subscribes to a specific topic(*{clientId}/data*).

MQTTConsumer.java core functionality code snippet:

```
//Code to create Mqtt Client to subscribe to topics

MqttConnectionOptions connOpts = new MqttConnectionOptions();

connOpts.setCleanStart(false);
client = new MqttAsyncClient("tcp://localhost:1883", "mqtt-client");
IMqttToken token = client.connect(connOpts);
token.waitForCompletion();


/*

* Code omitted for brevity

*/


MqttSubscription[] subscriptions = {new MqttSubscription("+/data")};
client.subscribe(subscriptions);
```

Upon message arrival to the subscriber, the message is further published using the Kafka Publisher.

```
//Method to publish payload to Kafka Producer
public void messageArrived(String s, MqttMessage mqttMessage) throws Exception {
   publisher.publish(KafkaPublisher.TOPIC,
         mqttMessage.getPayload()
   );
}
```

KafkaPublisher.java core functionality code snippet:

```
TOPIC = "sensor-data";

producer = new KafkaProducer<>(props);


//Method to publish payload to Kafka topic
public void publish(String clientId, byte[] payload) {
   producer.send(new ProducerRecord<>(TOPIC, clientId, payload));
}
```

At this point, all data collected from the IoT-simulator (20 clients publishing data periodically), is forwarded from the MQTT broker to the Kafka cluster to one shared topic ***“sensor-dat**a*” ready to be consumed by all microservices that are connected as subscribers.

Microservices

A microservice can simply be described as an encapsulated component consisting of a single application process. Within the proposed architecture, three different examples of microservices are described and one is implemented into the IoT platform.

Data Trafficer

The first microservice provided in this custom architecture solution is the Data Trafficer. The Data Trafficer is a microservice responsible for consuming data from the Kafka topic and storing the data into a time-series database of our choice. For this project, InfluxDB is chosen.

Time Series Database – InfluxDB

InfluxDB is an open-source time-series database created for storing and retrieving time series data often used for IoT sensor data and real-time analytics.
In addition to InfluxDB, Chronograf is used as the user interface and administrative component of the InfluxDB platform.
Within the Data Trafficer, an InfluxDB client needs to be created to connect and communicate with the database.

InfluxDBClient.java core functionality code snippet:

```
//Constructor to create influx database client

public InfluxDbClient() {
   InfluxDBClientOptions options = new InfluxDBClientOptions.Builder()
         .url("http://localhost:8086")
         .authenticate("influxdb", "influxdb".toCharArray())
         .build();
   influxDB = InfluxDBClientFactory.create(options);
}


//Method to write the measurement data to the database

public void writePoint(SensorData sensorData) throws InterruptedException {
   WriteApiBlocking writeApi = influxDB.getWriteApiBlocking();

   Point point = Point.measurement("sensor")
         .addTag("sensor_id", sensorData.getId())
         .addField("sensor_id", sensorData.getId())
         .addField("type", sensorData.getType().toString())
         .addField("value", sensorData.getValue())
         .addField("timestamp", sensorData.getTimestamp())
         .time(Instant.now(), WritePrecision.MS);
```

```
  writeApi.writePoint(point);
}
```

Data Trafficer Kafka Consumer

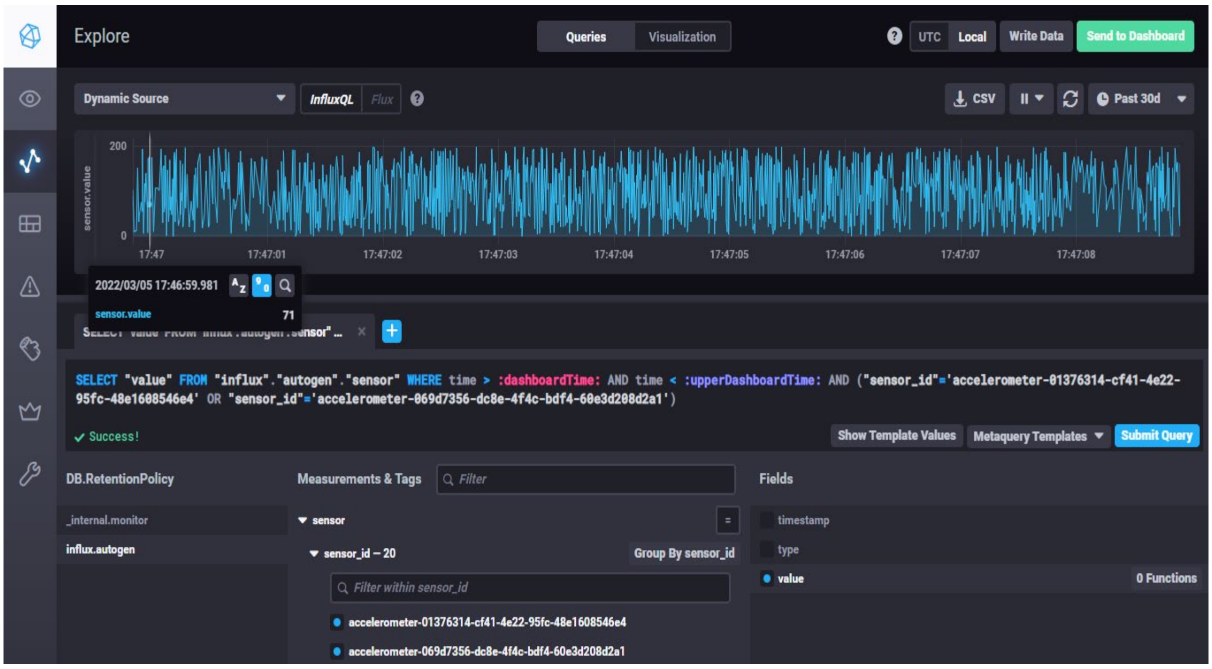
Finally, we create a Kafka Consumer which continuously attempts retrieving new data from the queue, and then forwards the record consumed to the database.

DTKafkaConsumer.java core functionality code snippet:

```
//Create Kafka Consumer object
final KafkaConsumer consumer = new KafkaConsumer(props);

//Constantly try to poll messages from the Kafka topic
while (true) {
  final ConsumerRecords<String, byte[]> consumerRecords =
            consumer.poll(Duration.ofMillis(1000));

  //Write measurement data to database for each record consumed
  consumerRecords.forEach(record -> {
      SensorData data = SensorData.parseFrom(record.value());
      influxDbClient.writePoint(data);
  });
  consumer.commitAsync();
}
```

Chronograf

At this point, we query and visualize the data from the Influx database using the Chronograf user interface. Figure 5 shows the Chronograf UI visualization of the data values received for two specific accelerometers.



```
SELECT "value" FROM "influx"."autogen"."sensor" WHERE time > :dashboardTime: AND time < :upperDashboardTime: AND ("sensor_id"='accelerometer-01376314-cf41-4e22-95fc-48e1608546e4' OR "sensor_id"='accelerometer-069d7356-dc8e-4f4c-bdf4-60e3d208d2a1')
✓ Success!
```

Figure 5: Chronograf User Interface

Microservices: Possible Future Applications

Considering the microservice architecture, other applications can easily be developed and used within the same IoT platform. Application implementations that are out of scope for this work, include but are not limited to the following examples:

Live Data Service

Another example of a microservice that can be a useful business case in an IoT platform could be a Live Data Service.

Just as the Data Trafficer implemented a consumer of the data on the Kafka topic to store the data to the time series database, a Live Data Service microservice could also consume the data, but rather than store it in a time-series database, directly push the data to subscribed users providing the users with a live stream of the data on the sensors.

This could be used to implement a monitoring system, or any other use case in which the user may want to immediately receive any change of value on a sensor.

Alert Engine

In addition to the previously mentioned microservices, an Alert Engine is another extremely useful use case for a microservice within an IoT platform. This microservice can contain an application that consumes the data and implement certain conditions under which an alert should be sent to users. For example, we can define a rule within an Alert Engine that we want to be informed if the temperature sensor on a car reads a value higher than 100 degrees Celsius, as well as an action we want to do upon receiving this sort of measurement. For demonstration purposes, the image of the proposed architecture sends the alert to another Kafka topic (for example “*sensor-alerts*”) which any user subscribed can receive data from. In this case, maybe not only the driver of the car should be informed of this, but also the producer of the car for further internal analysis.

API Gateway– GraphQL API

The API Gateway is here defined as the final component of the IoT platform.

An API gateway is a management tool for APIs that is placed between an end-user and a collection of backend services, in this case, microservices. It is designed to accept all requests as API calls and aggregate the microservices which are needed to return the appropriate result to the request.

In the custom solution provided, the GraphQL API is used as the API gateway and simply accepts a remote request and returns a response. A Node.js application is created to serve this purpose. To provide the functionalities to the user, queries are implemented in our application to define how each request is handled by GraphQL. A Resolver class consisting of two types of actions is created for this. First, an “***analytics***” function, which returns a response to the user containing the last N values for a specific client /sensor.

```
//Code to define a query returning N number of measurements for some client

@Query(() => GraphQLJSON)
public analytics(
  @Arg("clientId") id: string,
  @Arg("limit", { defaultValue: 1 }) limit: number
): Promise<JSON> {
  return this.analyticsService.analytics(id, limit)
}
```

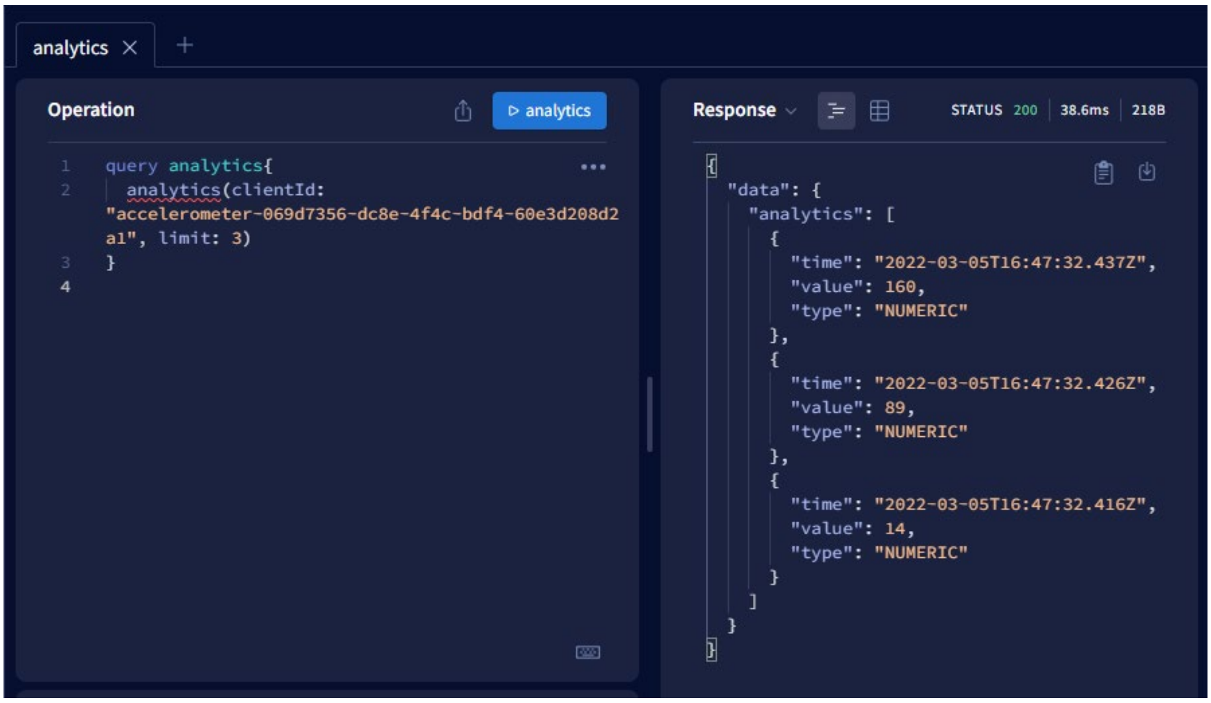
And second, a “***statistics***” function, which allows the user to obtain statistical values of the data of a specific client/sensor. These statistics functions include: MEAN, COUNT, MIN, MAX, and SUM which are all defined in the StatisticEnum and are dynamically forwarded to the query to return the chosen statistic from the Influx database.

```
//Code to define a query returning the statistical value of measurements on a client

@Query(() => GraphQLJSON)
public statistics(
  @Arg("clientId") id: string,
  @Arg("statistic", () => StatisticEnum ) statistic: StatisticEnum
): Promise<JSON> {
  return this.analyticsService.statistics(id, statistic)
}
```

As seen in the above code, an analyticsService object is used to serve the methods to the resolver. In the AnalyticsService class, these methods are defined as functions that execute specific queries towards the database to retrieve the data prompted by the user.

```
//Code to define the queries executed towards the database on user request
const analyticsResult = await this.influxClient.query(`SELECT "value","type" FROM "sensor" WHERE
"sensor_id"='${id}' ORDER BY time DESC LIMIT ${limit}`)

const statResult = await this.influxClient.query(`SELECT ${statistic}("value") FROM "sensor" WHERE
"sensor_id"='${id}' ORDER BY time DESC`)
```

GraphQL can then be used using its user interface Apollo Server which is also implemented for additional functionality and simplicity of presentation. Results of queries are presented to the user in JSON format with specific fields of interest for each query as defined in the application. The result of an executed query that retrieves the last 3 measurements for a specific accelerometer is shown in Figure 6.

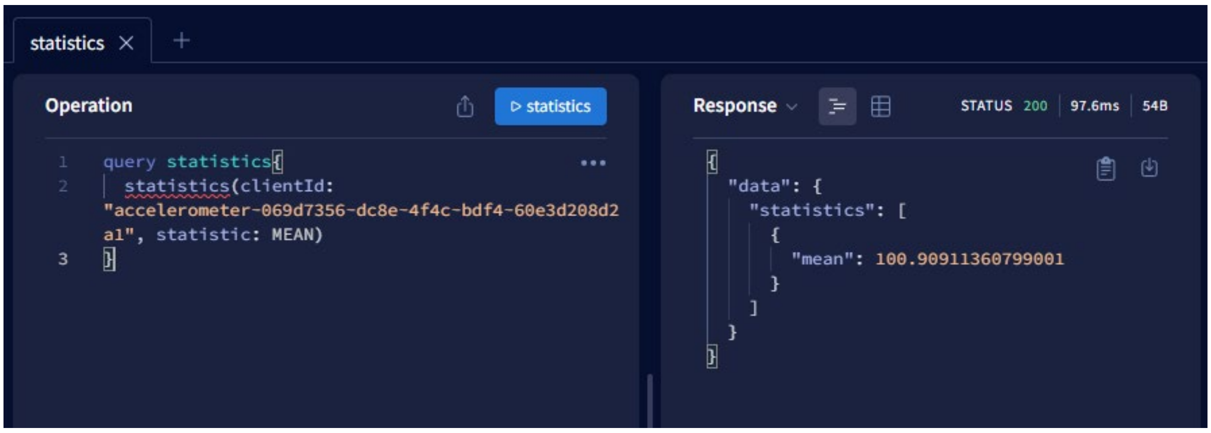


```
query analytics{
  analytics(clientId:
"accelerometer-069d7356-dc8e-4f4c-bdf4-60e3d208d2
a1", limit: 3)
}
```

```
{
  "data": {
    "analytics": [
      {
        "time": "2022-03-05T16:47:32.437Z",
        "value": 160,
        "type": "NUMERIC"
      },
      {
        "time": "2022-03-05T16:47:32.426Z",
        "value": 89,
        "type": "NUMERIC"
      },
      {
        "time": "2022-03-05T16:47:32.416Z",
        "value": 14,
        "type": "NUMERIC"
      }
    ]
  }
}
```

Figure 6: GraphQL retrieve last 3 measurements on accelerometer

The result of an executed query that retrieves the mean value of the received input data for a specific accelerometer is shown in Figure 7. As previously mentioned, the available statistics functions include: MEAN, COUNT, MIN, MAX, and SUM which are retrieved the same way as in Figure 7 by replacing the “statistic” field in the query.



```
query statistics{
  statistics(clientId:
"accelerometer-069d7356-dc8e-4f4c-bdf4-60e3d208d2
a1", statistic: MEAN)
}
```

```
{
  "data": {
    "statistics": [
      {
        "mean": 100.90911360799001
      }
    ]
  }
}
```

Figure 7: GraphQL retrieve the mean value for accelerometer

Custom microservice architecture for scalable IoT platform overview

Once the architecture proposed in Figure 3 is implemented, the final overview of the platform with the technologies used is described in Figure 8.

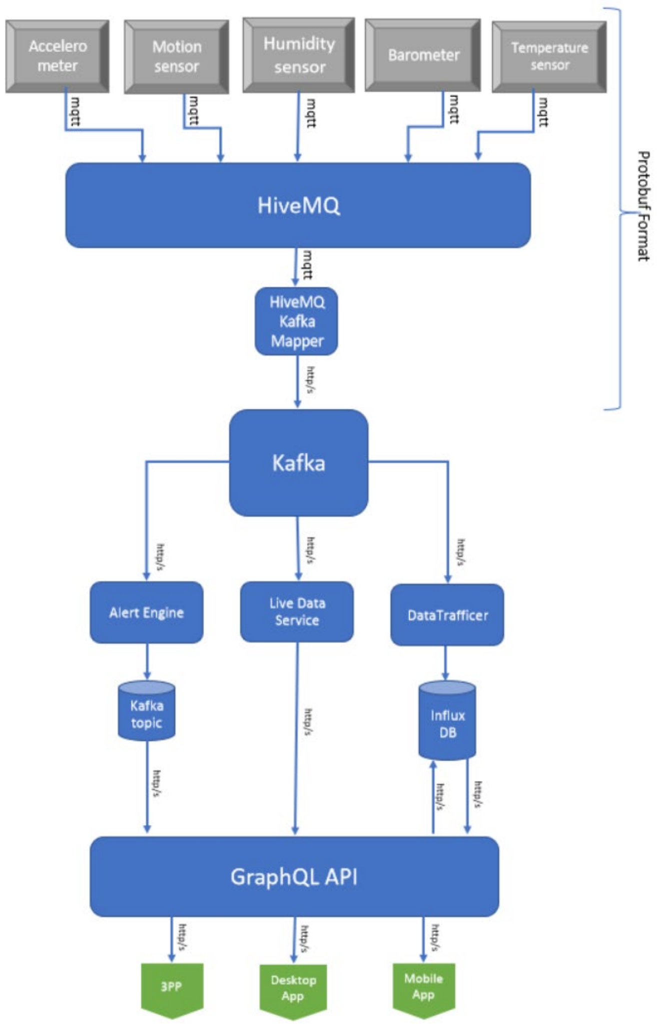


Figure 8: Implemented IoT platform solution

Conclusion

To conclude, IoT is a continuously, extremely quickly growing domain of today's technology and adaptations and enhancements need to constantly be researched and attempted to build the best functioning system possible for IoT platforms. These platforms are undeniably becoming a big part of our lives handling immense amounts of our data daily.

Even though there is no unified architecture that is agreed upon to be the best for such platforms and varies greatly depending on the requirements and possibilities of those building it, many agree that the previously widely used monolithic architecture no longer serves the requirements of the IoT environment. Because of its lack of modularity, scalability, and flexibility, it is being replaced with the microservice architecture, which is more flexible, easier to maintain, and most importantly limitlessly scalable.

This work presents a custom solution and technology stack organized as a microservice architecture to implement a scalable IoT platform. To demonstrate this, the following is implemented to achieve a competitive and optimized IoT platform:

- **IoT-Simulator** of a continuous stream of sensor data
  The IoT Simulator provides an inbound rate of messages of >500 messages per second, with the possibility of being modified to produce any amount of data making it an ideal component for penetration testing of any IoT platform.
- **Protocol Buffers** format and **MQTT** protocol
  MQTT and Protocol Buffers are used to increase communication speed between components in the IoT platform. Protobuf performs up to 6 times faster than JSON. (Krebs, 2017). According to measurements in 3G networks, throughput of MQTT is 93 times faster than HTTP's (Serozhenko, 2017).
- **HiveMQ** as the MQTT broker
  Considering HiveMQ recent success and expectations of 50% of globally produced cars being connected using MQTT and HiveMQ by 2022, it is the most competitive and relevant technology to use in such a platform architecture research paper.
- **Apache Kafka** cluster for data streaming and sequencing
  Apache Kafka employs sequential disk I/O for enhanced performance for implementing queues compared to other message brokers. For this reason, Kafka requires less hardware, and is ideal for this paper, however depending on the environment in production, others may perform better (Levy, 2018).
- A custom **microservice** for handling business case logic
  Data Trafficer - microservice responsible for consuming data from Kafka and storing the data into a time-series database. Time-series databases are optimized for time-stamped or time series data and are the fastest growing database category.
- **GraphQL** as an API gateway presented to end-users using Apollo
  GraphQL enhances user experience by making querying for wanted results easier to formulate and read than those of the more standard Rest API.

References

Ankit. (2021, July 15). 5 Challenges of IOT Connectivity & Tips to Overcome Them. https://huddle.eurostarsoftwaretesting.com/challenges-of-iot/

Banafa, A. (2017, March 14). Three Major Challenges Facing IoT. IEEE. https://iot.ieee.org/newsletter/march-2017/three-major-challenges-facing-iot.html

Buyya, R., & Dastjerdi, A. V. (Eds.). (2016). Internet of Things : Principles and Paradigms. Elsevier Science & Technology.

Collins, D. (2020, August 15). What is MQTT and when is it used in motion applications? Motion Control Tips. https://www.motioncontroltips.com/what-is-mqtt-and-when-is-it-used-in-motion-applications/

Delaney, J. R., & Colon, A. (2022, March 10). The Best Smart Home Security Systems for 2022. PCMag. https://www.pcmag.com/picks/the-best-smart-home-security-systems

Dix, P. (2021, July). Why Time Series Matters for Metrics, Real-Time Analytics and Sensor Data. https://www.influxdata.com/time-series-database/

Gaur, C. (2020, April). Google Protocol Buffer - Serializing Structured Data. XENONSTACK.

Gillis, A. S. (2022, March). What is the internet of things (IoT)? Retrieved March 6, 2022, from https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT

Gnatyk, R. (2018). Microservices vs Monolith: Which architecture is the best choice for your business? [Abstract]. N-ix, (October).

Götz, C. (2021, January 7). Trends to watch in 2021 for MQTT and HiveMQ. https://www.hivemq.com/blog/trends-to-watch-in-2021-for-hivemq-and-mqtt/

Imran, M. A., Zoha, A., Zhang, L., & Abbasi, Q. H. (2020). Grand challenges in iot and sensor networks. Frontiers in Communications and Networks, https://doi.org/10.3389/frcmn.2020.619452

Janson, C. (2021, July 22). Top 5 IoT Development Platforms in 2021. https://www.iotforall.com/top-5-iot-development-platforms-in-2021

Kalyan, A. (n.d.). A Brief Understanding of the Role of MQTT Protocol in IoT. Beyond Root. Retrieved February 19, 2022, from https://beyondroot.com/blog/a-brief-understanding-of-the-role-of-mqtt-protocol-in-iot/

Krebs, B. (2017, January). Beating JSON performance with Protobuf. https://auth0.com/blog/beating-json-performance-with-protobuf/

Lester, R. (2022, March). Scalability – What it means and why it's so critical to IoT. IT Pro Portal. https://www.itproportal.com/features/scalability-what-it-means-and-why-its-so-critical-in-the-iot/

Levy, E. (2019, May 7). Kafka vs. RabbitMQ: Architecture, Performance & Use Cases. https://www.upsolver.com/blog/kafka-versus-rabbitmq-architecture-performance-use-case

Meghamala, P. (2019, July 16). How Business Processes Are Evolving With The IoT. https://iot.electronicsforu.com/content/tech-trends/business-processes-evolving-iot/

MQTT: The Standard for IoT Messaging. (2022). Retrieved March 6, 2022, from https://mqtt.org/

Sax M.J. (2018) Apache Kafka. In: Sakr S., Zomaya A. (eds) Encyclopedia of Big Data Technologies. Springer, Cham. https://doi.org/10.1007/978-3-319-63962-8_196-1

Serozhenko, M. (2017, March). MQTT vs. HTTP: which one is the best for IoT? https://medium.com/mqtt-buddy/mqtt-vs-http-which-one-is-the-best-for-iotc868169b3105

The Complete MQTT Broker Selection Guide. (2020). Catchpoint.