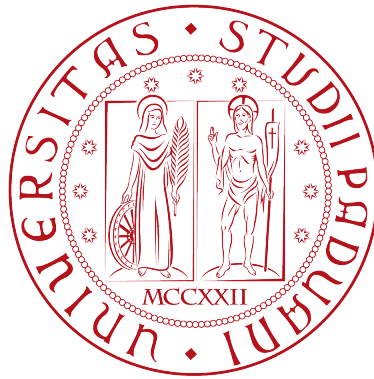


University of Padua

DEPARTMENT OF INFORMATION ENGINEERING

MSc in ICT for Internet and Multimedia



Scalable Cloud Agnostic Architecture for IoT
Data Analysis and Machine Learning

Master Degree

Relator

Prof. Lorenzo Vangelista

Master Candidate

Alessandro Discalzi

ID 2088235

ACADEMIC YEAR 2023-2024

Summary

This document describes the work done during the 750-hours final project at **221e S.r.l.** The project's goal is to architect and develop a cloud-based architecture capable of ingesting and processing data from heterogeneous IoT sensors and on-premises storage. The system must be designed to be scalable and fault-tolerant, and it must be platform-agnostic.

This document is going to describe the company, the idea behind the project, the work done and an assessment of what I developed and learned during my internship.

*“If the past is just dust
Then the future could be our dream”*

— Lorna Shore

Acknowledgements

Prof. Lorenzo Vangelista, my thesis supervisor, deserves my deepest gratitude for his exceptional support and guidance throughout the completion of this research.

My family, for their encouragement and understanding throughout this academic endeavour, has my heartfelt thanks.

I am truly grateful to Luca Perosa, Bledar Gogaj, Marco Lionello, and all my peers at SCAI ITEC, for their unwavering support when I made the decision to pursue a Master’s degree.

I extend my sincere appreciation to PhD. Roberto Bortoletto, my company tutor, and all my colleagues in 221e for their invaluable support and guidance throughout my final project.

Last but not least, I want to express my gratitude to all my friends for having my back and being there through high and lows. Your friendship means a lot to me, and I appreciate the support and the good times we’ve shared.

Padova, October 2024

Alessandro Discalzi

Contents

Acknowledgements	iii
1 Introduction	1
1.1 The Company	1
1.2 Objectives and requirements	2
1.2.1 Cloud Infrastructure	2
1.2.2 Data collection	2
1.2.3 Data processing	3
1.2.4 Security	3
1.2.5 Scalability	6
2 Technologies	7
2.1 Cloud	7
2.1.1 Aruba Cloud	7
2.1.2 Amazon Web Services	9
2.1.3 Microsoft Azure	18
2.2 Present Solutions	23
2.2.1 Alleantia IoT Edge Hub	23
2.2.2 Eclipse Kura	23
2.2.3 Eurotech Everyware Cloud	24
2.2.4 STMicroelectronics X-Cube Cloud	24
2.2.5 MQTTX	25
2.2.6 EMQX	25
2.3 Machine Learning at edge	25
2.3.1 Tensorflow Lite	25
2.3.2 Tiny Engine	26
2.3.3 Federated Learning and Transfer Learning	26
3 Methodology	28
3.1 Architecture	28
3.1.1 Data Collection Layer	29
3.1.2 Data Collection from IoT Devices	29
3.1.3 Data Collection from Archived Data	29
3.1.4 Data Storage Layer	29
3.1.5 Data Analysis Layer	29
3.1.6 Cloud Server vs. SaaS Solutions	30
3.2 Test Implementation	30
3.2.1 Data Collection Component	30

3.2.2	Data Analysis Component	31
4	Real world implementation	33
4.1	Client's requirements	33
4.2	System Architecture	35
4.2.1	Architecture 1: Preemptive Analytics Processing	35
4.2.2	Architecture 2: On-Demand Analytics Processing	36
4.2.3	Comparative Summary	38
4.3	Implementation	39
4.3.1	Rationale for Choosing Architecture 1	39
4.3.2	Implementation Steps	39
4.3.3	Technical Detail and Tools	42
4.4	Testing	43
4.4.1	Unit Testing	43
4.4.2	Integration Testing	43
4.4.3	Performance Testing	43
5	Conclusion	46
5.1	Objectives achieved	46
5.2	Future developments	46
5.2.1	Base architecture	46
5.2.2	Real World Implementation	47
5.3	Final considerations	48
6	Appendix A	49
6.1	Java vs Python for API Development	49
6.1.1	Java and Spring Boot	49
6.1.2	Python and FastAPI	50
6.1.3	Why Java is a Better Option	50
7	Appendix B	52
7.1	Linux Services and Cron Jobs	52
7.1.1	Linux Services	52
7.1.2	Cron Jobs	54
7.1.3	Conclusion	54
8	Bibliography	55

List of Figures

1.1	221e's logo	1
1.2	221e's technology ecosystem	2
1.3	Aruba Shared Responsibility Model	4
1.4	AWS Shared Responsibility Model	5
1.5	Azure Shared Responsibility Model	6
3.1	Architecture of the proposed solution	28
4.1	Architecture 1: Preemptive Analytics Processing	35
4.2	Architecture 2: On-Demand Analytics Processing	37

List of Tables

4.1	Data Volume Estimates	34
4.2	Architecture Comparison	38
4.3	Performance Metrics for Calculating, Retrieving, and Listing Analytics	44

Chapter 1

Introduction

1.1 The Company

221e S.r.l.¹, an innovative startup established in 2012 in Italy, has business units in Padova, Treviso, and Bergamo. The company leverages advancements in IT, microelectronics, sensors, and control algorithms to develop miniaturized wireless embedded systems.

Operating under a dual-layer business model, 221e offers:

1. OEM (Original Equipment Manufacturer) services to third-party clients needing a technology partner for product development. This involves R&D contracts followed by commercial agreements for the supply of engineered systems or technology licensing.
2. Finished products, particularly general-purpose multi-sensor hardware platforms, to direct customers or distributors.

Targeting the Wearable Devices market and the broader Internet of Things (IoT) industry, 221e capitalizes on the limitless applications within these fields. The name 221e, which represents the infinity Unicode character (∞), reflects this boundless potential.

Despite being a small business, 221e is rapidly growing, driven by innovation and entrepreneurship, riding the wave of IoT and wearable device advancements.



Figure 1.1: 221e's logo

¹221e S.r.l. URL: <https://www.221e.com/>.

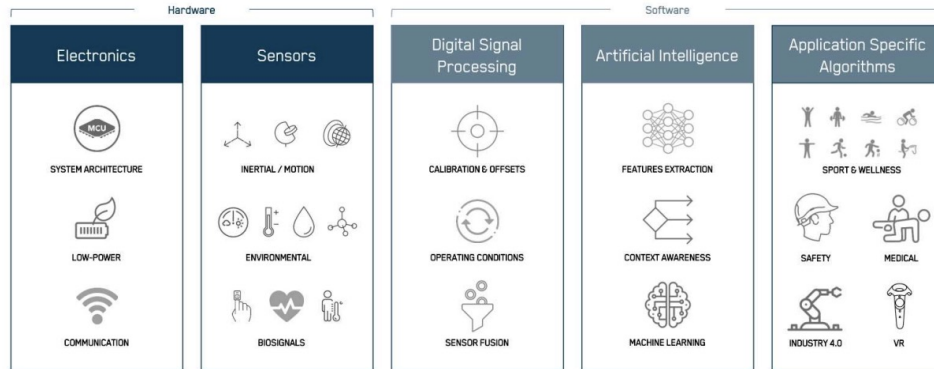


Figure 1.2: 221e's technology echosystem

1.2 Objectives and requirements

The idea behind the project is to create a cloud-agnostic architecture for the 221e's IoT devices. The architecture should be able to ingest data from multiple sources, store it and analyze it. The architecture should be able to scale horizontally and vertically, and should be able to be deployed on multiple cloud providers.

1.2.1 Cloud Infrastructure

The system must be able to ingest, store and process large amount of IoT data leveraging the power of any cloud provider present in the market. The final product to be developed is a cloud agnostic architecture that can be deployed on any cloud provider. The main advantage of a cloud agnostic architecture is that it can be deployed on any cloud provider, virtually without any modification. This allows customers to choose the cloud provider that best fits his needs. The providers taken into account to develop the architecture during this project are the following: [Aruba Cloud](https://www.arubacloud.com/)², [Amazon Web Services](https://aws.amazon.com/)³ and [Microsoft Azure](https://azure.microsoft.com/)⁴. AWS and Azure were chosen because the already developed experience by the company and me while Aruba Cloud was chosen because of a partnership between the company and the cloud provider that started during the development of the project.

1.2.2 Data collection

The system must be able to ingest data from online devices and on-premise data sources. Online devices are devices that are connected to the internet and can send data to the cloud via MQTT protocol. On-premise data source are offline files that are stored on a local machine and must be uploaded to the cloud. The system must provide a way to upload these files to the cloud.

²Aruba Cloud. URL: <https://www.arubacloud.com/>.

³Amazon Web Services. URL: <https://aws.amazon.com/>.

⁴Microsoft Azure. URL: <https://azure.microsoft.com/>.

1.2.3 Data processing

The system must be able to preprocess the data before storage. The preprocessing of the data includes data validation, data cleaning and data transformation, this can be done in cloud or on-premise. The system must be also able to process data after storage. The processing of the data includes data analysis and machine learning operations. The goal of the data processing is to extract useful information from the data and to provide insights to the customer.

1.2.4 Security

Security is a major concern for the system. In certain scenarios, the data collected could be sensitive and thus must be protected both in transit and at rest.

Security in transit

Data in transit are transferred using MQTT protocol which is not encrypted by default. The system must provide a way to encrypt and secure the data in transit. MQTT brokers however supports authentication and authorization through certificates as well as TLS/SSL encryption. Using a broker that supports these features is a must.

































Security at rest

Data at rest is stored in cloud storage. The cloud storage must provide a way to encrypt the data at rest. The system must also provide a way to manage the encryption keys. Each cloud service provider taken into account provide a way to encrypt data at rest and manage the encryption keys.

Furthermore, each cloud provider uses a shared responsibility model for security.

Aruba Shared Responsibility Model *Aruba Shared Responsibility Model*⁵ is a model that defines the responsibilities of Aruba and the customer for security. Aruba is responsible for the security of the cloud, while the customer is responsible for security in the cloud.

The responsibilities of the customer and Aruba varies depending on the service, as shown in the following table.

	On-premises	IaaS	PaaS	SaaS
Data				
Application				
Operating System				
Virtualization				
Servers				
Storage				
Network				
Physical				

 Customer


 Aruba Cloud

Figure 1.3: Aruba Shared Responsibility Model

⁵*Aruba Shared Responsibility Model*. URL: <https://kb.arubacloud.com/en/computing/use-and-technology/shared-responsibility-model.aspx>.

AWS Shared Responsibility Model **AWS Shared Responsibility Model**⁶ is a model that defines the responsibilities of AWS and the customer for security. AWS is responsible for the security of the cloud, while the customer is responsible for security in the cloud.

It's important to keep in mind that services in services catalogued as *Infrastructure as a Service* (IaaS) the customer is responsible for the security of the operating system and the applications, while in fully managed services the customer is responsible only for the security of the data.

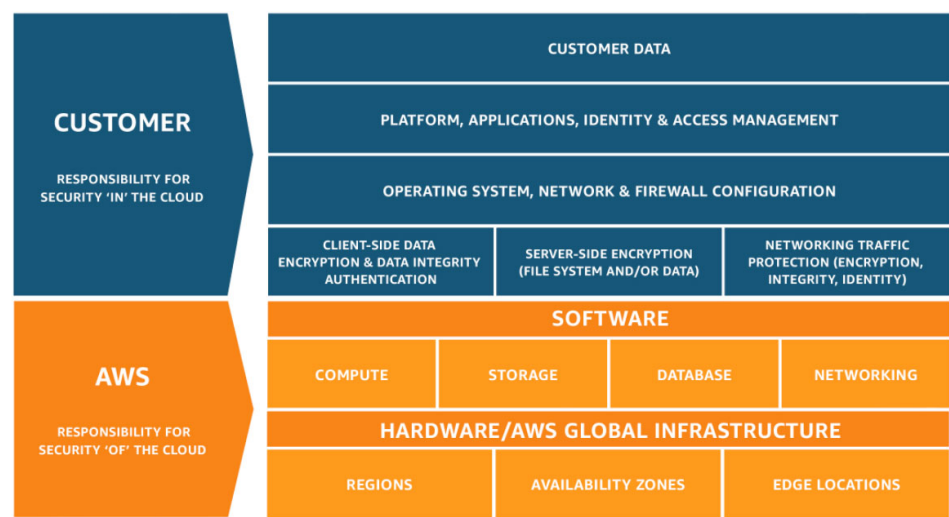


Figure 1.4: AWS Shared Responsibility Model

⁶AWS Shared Responsibility Model. URL: <https://aws.amazon.com/compliance/shared-responsibility-model/>.

Azure Shared Responsibility Model [Azure Shared Responsibility Model⁷](https://learn.microsoft.com/en-us/azure/security/fundamentals/shared-responsibility) is a model that defines the responsibilities of Microsoft and the customer for security. The responsibility varies depending on whether the service is *Software as a Service* (SaaS), *Platform as a Service* (PaaS), *Infrastructure as a Service* (IaaS) or on premise. Regardless of the deployment type, the customer always retains data, endpoints account and access management responsibilities.

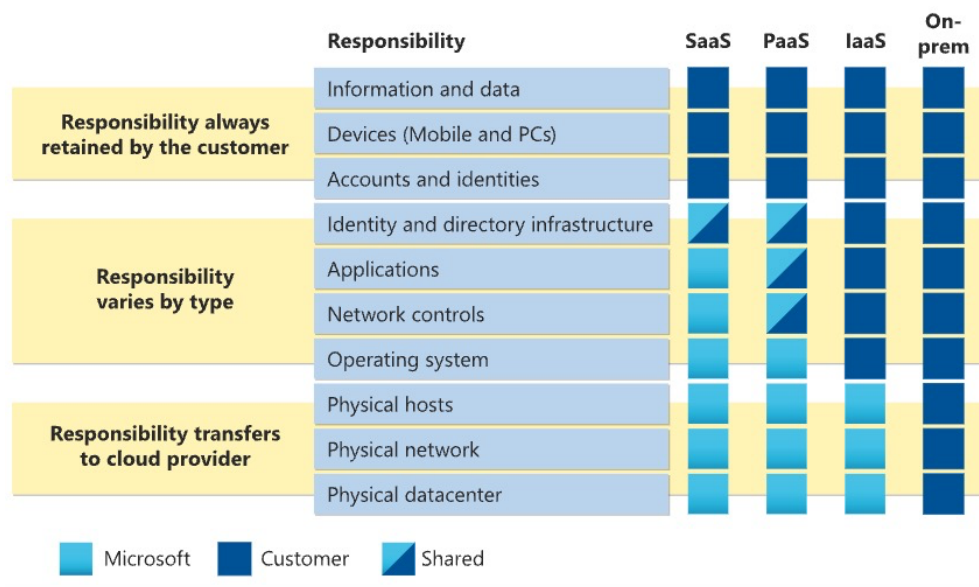


Figure 1.5: Azure Shared Responsibility Model

1.2.5 Scalability

The system must be able to scale horizontally by adding more instances of the same service and must be able to scale vertically by increasing the resources of the service. The system must be able to automatically scale based on the load of the system. Stress tests must be performed to verify the scalability of the system.

⁷ *Azure Shared Responsibility Model*. URL: <https://learn.microsoft.com/en-us/azure/security/fundamentals/shared-responsibility>.

Chapter 2

Technologies

In this chapter it's reported the study made on the various technologies taken into account to develop the project.

2.1 Cloud

This section describes the services offered by [Aruba Cloud](#)¹, [Amazon Web Services](#)² and [Microsoft Azure](#)³, exploring their features, pros and cons. Services features are better described in the respective official [Aruba Cloud](#)⁴, [AWS](#)⁵ and [Azure](#)⁶ documentation, while pros and cons are based on the company experience, users reviews that can be found in [TrustRadius](#)⁷ website and my experience as well.

These services are among the most used cloud services in the world, offering a wide range of services that can be used to develop the project. It's also important to mention that these providers were chosen right away because of the already developed experience with them, both in the company and by me.

2.1.1 Aruba Cloud

Aruba Cloud is a cloud service provider that offers a wide range of services like virtual machines, object storage, databases, and managed Kubernetes.

Bare Metal

Aruba Cloud Bare Metal is a service that provides dedicated servers with high performance and reliability.

Pros:

- High performance

¹*Aruba Cloud.*

²*Amazon Web Services.*

³*Microsoft Azure.*

⁴*Aruba Documentation.* URL: <https://kb.arubacloud.com/en/home.aspx>.

⁵*AWS Documentation.* URL: <https://docs.aws.amazon.com/>.

⁶*Azure Documentation.* URL: <https://docs.microsoft.com/en-us/azure/>.

⁷*Trust Radius.* URL: <https://trustradius.com/>.

- Reliable
- Cost effective
- Supports multiple operating systems

Cons:

- Need to manage the whole infrastructure
- Not scalable
- Not upgradable

Virtual Private server (VPS)

Aruba Cloud VPS is a service that provides virtual servers with high performance and reliability. It uses virtualizers such as OpenStacks, VMware and Hyper-V.

Pros:

- High performance
- Reliable
- Cost effective
- Supports multiple operating systems
- Scalable
- SLA up to 99.95%

Cons:

- Guaranteed resources only in the PRO plans

Virtual Private cloud (VPC)

Aruba Cloud VPC is a service that provides a virtual network isolated from the public network. It allows for the creation of multiple subnets and the configuration of security groups. It can be used to create a whole private cloud infrastructure.

Pros:

- Isolated network
- Multiple subnets
- Security groups
- Scalable
- SLA up to 99.98%

Cons:

- Complexity
- Costly

Object Storage

Aruba Cloud Object Storage is a service that provides scalable and secure object storage. It can be used to store and retrieve large amounts of unstructured data.

Pros:

- Scalable
- Secure
- Cost effective
- Highly available
- Reservable plans or pay per use

Cons:

- Limit for traffic to the public network

Managed Kubernetes

Aruba Cloud Managed Kubernetes is a service that provides a fully managed Kubernetes cluster. It can be used to deploy, scale, and manage containerized applications.

Pros:

- Fully managed
- Scalable
- Secure
- Cost effective
- Maximum redundancy
- Minimum latency
- Rapid deployment
- Kubernetes native API

Cons:

- Complexity

2.1.2 Amazon Web Services

Batch

AWS Batch is a fully managed service that enables developers to easily and efficiently run thousands of batch and machine learning computing jobs on AWS.

Pros:

- Fully managed
- Scalable
- Cost effective

- Supports different batch processing scenarios
- Supports machine learning
- Easy to use
- Versatile

Cons:

- Not well documented

Bedrock

AWS Bedrock is a fully managed service that simplifies the deployment and management of machine learning models.

Using AWS Bedrock users can choose from a variety of pre-trained models and deploy them on the edge.

Pros:

- Fully managed
- Flexible
- Native support for Retrieval Augmented Generation (RAG) models

Cons:

- Costly
- New dependencies may introduce problems
- The chosen model may not be future proof

DynamoDB

AWS DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. Tables can store and retrieve virtually any amount of data, serving any level of request traffic. It automatically spreads the data and traffic for the table over a sufficient number of servers to handle the request capacity specified by the customer and the amount of data stored, while maintaining consistent and fast performance.

Pros:

- Fully managed
- Fast and predictable performance
- Scalable
- Highly available
- NoSQL

Cons:

- Hard to make changes against bulks of records
- Need to know at prior which queries will be made

Elastic map reduce (EMR)

AWS EMR is a big data platform that simplifies the deployment and management of big data frameworks, like Apache Hadoop and Apache Spark, on AWS.

Pros:

- Fully managed
- Scalable
- Petabyte scale data processing
- Easy resources provisioning
- Reconfigurable

Cons:

- Complexity
- Costly

Glue

AWS Glue is a fully managed ETL service that enables efficient data integration on a large scale.

Pros:

- Fully managed
- Pay per use
- Scalable
- Provides a centralized metadata repository
- Supports different data sources and formats
- Can automatically discover and catalog data from various sources
- Allow for job scheduling
- Data encryption

Cons:

- Costly for high workloads
- Performance issues with large datasets
- Complexity

Greengrass

AWS Greengrass is an open source edge runtime and cloud service used to build, deploy, and manage device software. It enables the devices to process the data locally, while still using the cloud for management, analytics, and durable storage.

It also enables encryption at rest and in transit and it can also extend device functionality with AWS Lambda functions.

Pros:

- Edge computing
- Encryption at rest and in transit
- Extend device functionality with AWS Lambda functions
- ML models deployment

Cons:

- Restrained to AWS services
- Not platform agnostic
- Resource intensive for small devices
- Need a connection for the initial setup

IoT Core

AWS IoT Core is a fully managed cloud service that lets connected devices easily and securely interact with cloud applications and other devices. It is composed of multiple services like Device Management, Device Defender, Device Advisor, and IoT Analytics and only some of them can be used during the development.

Pros:

- Composed of multiple services so only the necessary ones can be used
- Encryption at rest and in transit
- Supports MQTT, HTTP, and WebSockets
- Allows for device management
- Allows for machine learning at edge
- Can trigger events thanks to custom rules

Cons:

- Not platform agnostic if installed on devices
- Lacks of integration for some devices

Kendra

AWS Kendra is a fully managed enterprise search service that allows developers to add search capabilities across various content repositories leveraging on built in connectors.

Pros:

- Fully managed
- Scalable
- Supports multiple data sources
- Easy to use and set up
- Accurate search results

Cons:

- Costly

Kinesis Data Firehose

AWS Kinesis Data Firehose is a fully managed service that simplifies the process of capturing, transforming and loading streaming data. It acts as an ETL service that can capture, transform, and load streaming data into a variety of AWS services. Additionally it can transform raw data in column oriented data formats like [Apache Parquet](https://parquet.apache.org/)⁸

Pros:

- Fully managed
- Can read data from IoT core and Kinesis Data Streams
- Scalable
- Can transform data
- Can load data into different AWS services
- Supports batching based on time or size

Cons:

- Not always cost effective
- Limited transformation capabilities
- Does not support batching based on more complex rules

⁸Apache Parquet. URL: <https://parquet.apache.org/>.

Kinesis Data Streams

AWS Kinesis Data Stream is a fully managed service that simplify the capture, processing and loading of streaming data in real time at any scale thus enabling real-time data analytics with ease.

Pros:

- Fully managed
- Scalable
- Real-time and fast data processing
- Keeps data for 24 hours by default

Cons:

- Not always cost effective
- Limited data retention
- Limited data transformation
- Not useful for certain batch processing scenarios

Lake Formation

AWS Lake Formation is a fully managed service that simplifies the creation, security and management of data lakes. It allows for cleaning and transforming the data using machine learning.

Pros:

- Fully managed
- Scalable
- Secure
- Simplifies lake creation
- Simplifies ingestion management
- Simplifies permission management
- Provides data auditing
- Supports machine learning
- Supports data cataloging

Cons:

- Complexity
- Costly
- Not native support for all data sources

Lambda

AWS Lambda is a serverless compute service that automatically manages the compute fleet, scaling precisely with the size of the workload. The key advantage of AWS Lambda is that it allows developers to run code without provisioning or managing servers, creating a highly scalable, flexible, and cost-effective environment for executing code. AWS Lambda supports a variety of programming languages and integrates seamlessly with other AWS services, making it a versatile tool for deploying microservices, building data processing workflows, and developing real-time applications. The service is designed to handle various use cases, from running simple, single-function applications to complex, multi-step workflows.

Pros:

- Fully managed
- Serverless
- Pay per use
- Scalable
- Easy to integrate with other AWS services
- Supports multiple programming languages
- Easy to deploy and maintain
- Can run parallel executions
- Low time to market
- Supports custom libraries

Cons:

- Limited execution time (15 mins)
- Limited memory
- Limited environment variables
- Maximum 1000 concurrent executions
- Cold start
- Not cost effective for high workloads

Managed Service for Apache Flink (MSF)

AWS Managed Service for Apache Flink (MSF) is a fully managed service that simplifies the creation and execution of real-time applications using **Apache Flink**⁹, an open-source stream processing framework. This service provides developers with the tools to build sophisticated, high-throughput, low-latency data processing applications. MSF automates the underlying infrastructure management, enabling teams to focus on application logic rather than operational concerns. With built-in scalability, the service

⁹Apache Flink. URL: <https://flink.apache.org/>.

can handle varying workloads efficiently, making it suitable for both batch and stream processing. MSF also offers tight integration with other AWS services, providing a cohesive ecosystem for end-to-end data processing workflows.

Pros:

- Fully managed
- Scalable
- Supports batch and stream processing
- Real-time processing
- Large-scale data processing

Cons:

- Complexity

Managed Streaming for Kafka (MSK)

AWS Managed Streaming for Apache Kafka (MSK) is a fully managed service that simplifies the setup, scaling, and management of **Apache Kafka**¹⁰ clusters. Apache Kafka is a distributed streaming platform widely used for building real-time data pipelines and streaming applications. MSK allows developers to leverage Kafka's powerful capabilities without the operational burden of managing Kafka infrastructure. The service ensures high availability, durability, and security, integrating seamlessly with other AWS services. This enables users to build robust, scalable streaming solutions while focusing on application development rather than infrastructure management.

Pros:

- Fully managed
- Scalable
- Cost effective
- Secure
- High availability
- Easy to integrate with other AWS services

Cons:

- Local testing challenges: hard to replicate the same environment in production and locally
- Not suitable for high traffic scenarios
- Complexity

¹⁰ *Apache Kafka*. URL: <https://kafka.apache.org/>.

Sage Maker

AWS SageMaker is a comprehensive cloud-based machine learning platform that enables developers and data scientists to build, train, and deploy machine learning models at scale. SageMaker provides a suite of tools that simplify each step of the machine learning workflow, from data labeling and preparation to model tuning and deployment. The platform supports a variety of machine learning frameworks and integrates with other AWS services, offering flexibility and interoperability. SageMaker's built-in algorithms and managed infrastructure allow users to focus on developing innovative models without worrying about the complexities of underlying hardware and software management.

Pros:

- Fully managed
- Scalable
- Supports multiple machine learning frameworks
- Supports multiple programming languages
- Allow for easy model deployment

Cons:

- Cannot schedule training jobs
- Costly for high workloads

Simple Storage Service (S3)

AWS S3 is an object storage service offering scalability, data availability, security, and performance. With S3, any amount of data can be stored and retrieved from anywhere on the web. Data is stored as objects in buckets, with each object representing a file and its metadata.

Pros:

- Scalable
- Highly available
- Secure
- Durable
- Cost effective
- No bucket size limit
- No limit to the number of objects that can be stored in a bucket
- Has different storage classes to fit frequent access, infrequent access, and long-term storage

Cons:

- Not suitable for small files

- Object size limit (5TB)
- Maximum 100 buckets per account
- Max 5GB per file upload via PUT operation

2.1.3 Microsoft Azure

Blob Storage

Azure Blob Storage is a fully managed object storage service that is highly scalable and available. It can store large amounts of unstructured data, making it suitable for a wide range of workloads.

Pros:

- Fully managed
- Scalable
- Highly available
- Secure
- Cost effective
- No limit to the number of objects that can be stored in a container
- Has different storage tiers to fit frequent access, infrequent access, and long-term storage
- Different storage options (Blob, archive, queue, file and disk)

Cons:

- Not suitable for small files
- Object size limit (4TB)
- Maximum 2PB per account in US and Europe
- Maximum 500TB per account in other regions

Cosmos DB

Azure Cosmos DB is a fully managed NO-SQL database service supporting multiple data models. It supports multiple NoSQL databases like PostgreSQL, MongoDB, and Cassandra.

Pros:

- Fully managed
- Scalable
- Multi model
- Global distribution
- Cons:istency levels based on the application needs

- Easy to use

Cons:

- Expensive
- Slow for complex queries

DataBricks

Azure Databricks is a fully managed Apache Spark-based analytics platform supporting a variety of libraries and languages.

Pros:

- Fully managed
- Scalable
- Supports multiple programming languages (Python, R, Scala, SQL, Java)
- Supports multiple libraries (Tensorflow, PyTorch, Scikit-learn, etc.)
- Open data lakehouse

Cons:

- Costly
- Complexity
- Hard to configure

Data Explorer

Azure data explorer is a fully managed, real-time and high volume data analytics service. It offers speed and low latency, being able to get quick insights from raw data.

Pros:

- Fully managed
- Scalable
- Real-time data processing
- Low latency
- Supports multiple data sources
- Supports structured, semi-structured and unstructured data
- Fast data ingestion
- Can use batch processing

Cons:

- Complexity
- Costly
- Limited capabilities for data transformation
- Hard configuration

Data Factory

Azure data factory is a fully managed cloud-based data integration service. It provides tools to orchestrate data workflows while monitoring executions.

Pros:

- Fully managed
- Scalable
- Can perform data Analytics using Synapse

Cons:

- Complexity
- Limited transformation capabilities

Data Lake Storage

Azure Data Lake Storage is a secure and scalable data lake platform. It provides a single place to store structured and unstructured data, making it easy to perform big data analytics.

Pros:

- Fully managed
- Scalable
- Secure
- Cost effective
- Compatible with Hadoop
- Supports Python for data analytics

Cons:

- Data governance challenges

Event Grid

Azure Event Grid is a fully managed event routing service that simplifies the development of event-driven applications.

Pros:

- Fully managed
- Scalable
- Supports MQTT5
- Supports multiple event sources
- Supports multiple event handlers
- Supports multiple event types

- Supports multiple programming languages
- Supports multiple event patterns

Cons:

- Complexity
- Limitations in event storage and retention
- Costly

Event Hubs

Azure Event Hubs is a fully managed, real-time data ingestion service that is simple, secure, and scalable. It can be used to stream millions of events per second with low latency, from any source to any destination. It offers also native support for Apache Kafka, allowing user to run existing Kafka applications.

Pros:

- Fully managed
- Scalable
- Secure
- Low latency
- Supports Apache Kafka
- Schema registry: centralize repository for schema management
- Real time data processing

Cons:

- Costly
- Complexity
- Limitation in event storage
- Consumers need to manage their state of processing

Functions

Azure functions is a serverless compute service that enables developers to run code in response to events without the need to manage the infrastructure.

Pros:

- Fully managed
- Pay per use
- Scalable
- Supports multiple programming languages
- Easy to deploy and maintain

- Can run parallel executions
- Low time to market
- Supports custom libraries

Cons:

- Limited execution time (10 mins)
- Cold start
- Not cost effective for high workloads

IoT Hub

Azure IoT Hub is a cloud service that serves as the bridge between IoT devices and solutions in the cloud, facilitating reliable and secure communication. It can handle and manage a large number of devices making it suitable both for small-scale and enterprise-level solutions.

Pros:

- Secure
- Supports MQTT, AMQP, and HTTP
- Allows for device management
- Allows for machine learning at edge
- Can trigger events thanks to custom rules
- Can extend device functionality with Azure Functions

Cons:

- Not platform agnostic if installed on devices
- Lacks of integration for some devices
- Not well documented
- Costly
- Does not fully support MQTT5

Machine Learning

Azure Machine Learning is a fully managed service that allows developers to build, train, and deploy machine learning models.

Pros:

- Fully managed
- Scalable
- Supports multiple machine learning frameworks
- Supports multiple programming languages

- Allow for easy model deployment
- Cost effective
- Has MLOps capabilities
- Pay as you go

Cons:

- Cost raises when training big models
- Hard to optimize

2.2 Present Solutions

In this section are presented the solutions that are currently available on the market.

2.2.1 Alleantia IoT Edge Hub

IoT Edge Hub is [Alleantia](https://www.alleantia.com/)¹¹'s plug and play solution for the industrial IoT. It offers a wide range of features like device management, alarms and events, log management and report generation. It also supports the integration with [Microsoft Azure](#)¹².

Pros:

- Plug and play
- Device management
- Alarms and events
- Log management
- Report generation
- Integration with Microsoft Azure

Cons:

- Not platform agnostic
- Does not support [Amazon Web Services](#)¹³

2.2.2 Eclipse Kura

[Eclipse Kura](#)¹⁴ is an open source IoT Edge Framework that serves as a platform for building IoT gateways. It's based on Java/OSGi and it provides API access to the hardware interfaces of IoT Gateways.

Pros:

- Open source

¹¹*Alleantia*. URL: <https://www.alleantia.com/>.

¹²*Microsoft Azure*.

¹³*Amazon Web Services*.

¹⁴*Eclipse Kura*. URL: <https://eclipse.dev/kura/>.

- Platform agnostic
- Allows for flexible and modular development
- API access to hardware interfaces
- Introduces AI capabilities at the edge

Cons:

- Computational complexity for small devices
- Not well documented
- No native support for cloud services

2.2.3 Eurotech Everyware Cloud

Eurotech¹⁵ Everyware Cloud is a cloud-based IoT Integration Platform with a microservices architecture that allows to connect, configure and manage IoT gateways and devices.

Pros:

- Cloud-based
- Microservices architecture
- Allows to connect, configure and manage IoT gateways and devices
- Supports multiple protocols
- Supports multiple cloud services

Cons:

- Last update in 2019

2.2.4 STMicroelectronics X-Cube Cloud

STMicroelectronics¹⁶ X-Cube Cloud is a software package that enables the connection of STM32 microcontrollers to the cloud.

Pros:

- Supports multiple cloud services
- Offers generic and secure connection to the cloud
- Supports multiple protocols

Cons:

- Specific for STM32 microcontrollers
- Specific packages for each cloud service if you want to use the full potential
- The generic solution only runs on a subset of STM32 microcontrollers

¹⁵*Eurotech*. URL: <https://www.eurotech.com/>.

¹⁶*STMicroelectronics*. URL: https://www.st.com/content/st_com/en.html.

2.2.5 MQTTX

MQTTX¹⁷ is a cross-platform MQTT 5.0 client tool that can be used to publish and subscribe to MQTT messages.

Pros:

- Cross-platform
- Supports MQTT 5.0
- Connection management
- Log capabilities
- Data pipelines
- Device simulation

Cons:

- MQTT client only

2.2.6 EMQX

EMQX¹⁸ is an open source MQTT broker designed to be highly scalable.

Pros:

- Open source
- Scalable
- Supports MQTT 5.0
- Supports web sockets
- Supports multiple cloud services

Cons:

- The free version has limitations

2.3 Machine Learning at edge

This section describes the technologies that can be used to build and deploy machine learning models at the edge and the Federated Learning approach.

2.3.1 Tensorflow Lite

Tensorflow Lite¹⁹ is the mobile and edge version of **Tensorflow**²⁰ that allows to run machine learning models on edge devices.

Pros:

¹⁷MQTTX. URL: <https://mqttx.app/>.

¹⁸EMQX. URL: <https://www.emqx.io/>.

¹⁹TensorFlow Lite. URL: <https://www.tensorflow.org/lite>.

²⁰TensorFlow. URL: <https://www.tensorflow.org/>.

- Lightweight
- Supports multiple platforms
- Easy to use
- Well documented

Cons:

- Need to pretrain the model before deploying it
- On-device training is limited to Unix-based systems

2.3.2 Tiny Engine

Tiny Engine²¹²² is a specialized machine learning framework designed to build, train, and deploy models on edge devices. It enables developers to convert TensorFlow Lite models into C++ code, facilitating efficient deployment on resource-constrained devices. Tiny Engine supports multiple platforms and utilizes pre-trained models, making it a versatile tool for various edge computing applications. The framework is lightweight, ensuring minimal resource consumption and fast inference times, which are critical for real-time, on-device machine learning tasks. Tiny Engine is particularly suited for applications in areas such as IoT, where low power consumption and quick response times are essential.

Pros:

- Lightweight
- Supports multiple platforms
- Converts Tensorflow Lite models to C++
- Uses pre trained models

Cons:

- Need to pretrain the model before deploying it
- Better support for MCUnet²³
- Not well documented for deployment of custom models

2.3.3 Federated Learning and Transfer Learning

Federated Learning is a machine learning approach that trains an algorithm across multiple decentralized edge devices or servers holding local data samples, without exchanging them. This approach is advantageous because it allows for privacy preservation and data security, minimizing the risk of sensitive information being exposed. Federated Learning also makes it feasible to train models on devices with low computational power, which is particularly useful in edge computing environments where

²¹ *TinyEngine*. URL: <https://github.com/mit-han-lab/tinyengine>.

²² Ji Lin et al. "On-Device Training Under 256KB Memory". In: *arXiv:2206.15472 [cs]* (2022). URL: <https://arxiv.org/abs/2206.15472>.

²³ Ji Lin et al. "Mcunet: Tiny deep learning on iot devices". In: *Advances in Neural Information Processing Systems* 33 (2020). URL: <https://arxiv.org/abs/2007.10319>.

computational resources are limited. As described in “EdgeFed: Optimized Federated Learning Based on Edge Computing”²⁴, this method enables the development of sophisticated machine learning models by utilizing the collective power of multiple devices, enhancing both the efficiency and effectiveness of the training process.

Another important approach is Transfer Learning, a technique that transfers the knowledge from a model trained on a specific task to a new, related task. This approach significantly improves the performance of the model on the new task and reduces the time and resources needed for training. Transfer Learning is especially valuable when there is limited data available for the new task, as it leverages the pre-existing knowledge embedded in the model. As detailed in “Federated learning for IoT devices: Enhancing TinyML with on-board training”²⁵, this method can enhance the capabilities of IoT devices by enabling them to perform complex tasks with improved accuracy and efficiency, without the need for extensive retraining.

Pros:

- Privacy preservation
- Data security
- No need for data centralization
- Low computational power needed
- Low latency

Cons:

- Complex to implement
- Limited capabilities

²⁴Yunfan Ye et al. “EdgeFed: Optimized Federated Learning Based on Edge Computing”. In: *IEEE Access* 8 (2020), pp. 209191–209198. DOI: [10.1109/ACCESS.2020.3038287](https://doi.org/10.1109/ACCESS.2020.3038287).

²⁵M. Ficco et al. “Federated learning for IoT devices: Enhancing TinyML with on-board training”. In: *Information Fusion* 104 (2024), p. 102189. ISSN: 1566-2535. DOI: <https://doi.org/10.1016/j.inffus.2023.102189>. URL: <https://www.sciencedirect.com/science/article/pii/S1566253523005055>.

Chapter 3

Methodology

In this chapter, we will present the methodology used to develop the proposed solution. The chapter is divided into two sections: architecture, and implementation. The architecture section will present the proposed solution's architecture for the chosen cloud providers, while the implementation section will detail the steps taken to implement the solution.

3.1 Architecture

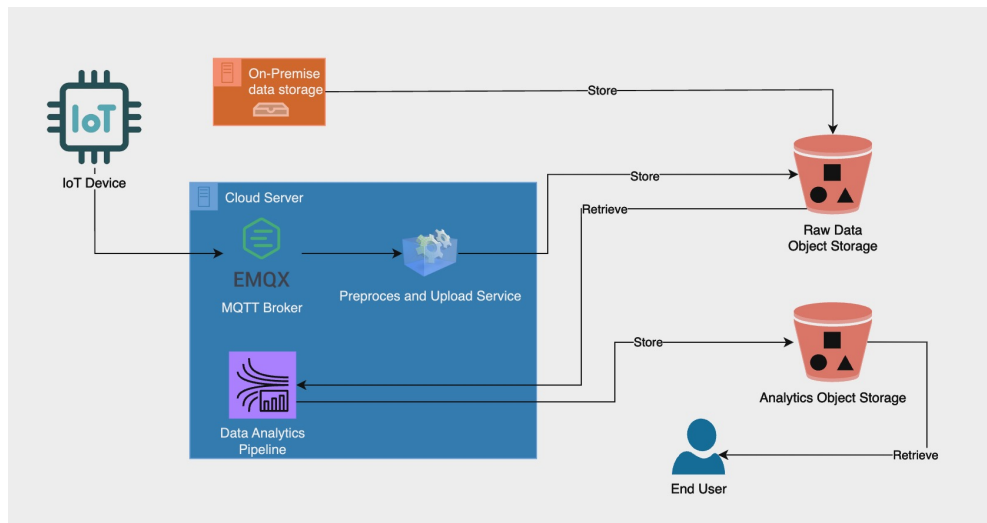


Figure 3.1: Architecture of the proposed solution

The proposed solution's architecture consists of a composition of loosely coupled microservices that work together to collect, process, and store data. The architecture is divided into three layers: the data collection layer, the data storage layer and the data analysis layer. Each layer plays a specific role in the overall system, ensuring scalability, flexibility, and resilience.

3.1.1 Data Collection Layer

The data collection layer is responsible for gathering data from IoT devices and archived data.

3.1.2 Data Collection from IoT Devices

The data collection from IoT devices utilizes the MQTT protocol, a lightweight messaging protocol designed for devices with low bandwidth and high latency. MQTT operates on a publish-subscribe model, where devices publish messages to a broker, which then forwards the messages to subscribers. This protocol is ideal for IoT devices due to its simplicity, ease of implementation, and reliability, with support for different quality of service (QoS) levels to ensure message delivery.

In our scenario, a large number of IoT devices publish data to the cloud. Each device publishes data to a specific topic managed by a cloud-based broker. The data is sent in JSON format, with each message containing a timestamp and a set of key-value pairs representing the data. Once the data is published to the broker, it undergoes preprocessing before being stored in object storage. This preprocessing involves parsing the JSON data, extracting the timestamp and key-value pairs, and converting the data to CSV format. The CSV format was chosen for its simplicity and compatibility with common data processing tools and libraries as well as its reduced verbosity compared to JSON. The preprocessed data is then stored in object storage, making it accessible to the data processing pipeline.

3.1.3 Data Collection from Archived Data

The data collection from archived data is done by uploading files from on-premises to the cloud via a simple script that leverages cloud API. The files, which are already in CSV format, are uploaded to the object storage, where they can be accessed by the data processing pipeline. The files contain historical data that have been collected over time in a number of different scenarios.

3.1.4 Data Storage Layer

The data storage layer is responsible for storing the raw CSV data and making it accessible to the data processing pipeline. This layer utilizes cloud-based object storage for scalability and durability. The data, always stored in CSV format, is uploaded to the object storage by the data collection layer and accessed by the data analysis layer. Once the analysis is done, the results are stored in a separate object storage bucket, making them accessible to other services or users. The object storage provides a reliable and cost-effective solution for storing large amounts of data, with built-in redundancy and scalability features.

3.1.5 Data Analysis Layer

The data analysis layer is responsible for analyzing the collected data and generating insights. This layer consists of several services that retrieve the preprocessed data from the object storage and perform various analytical tasks. The data, initially preprocessed and stored in CSV format, is accessed directly from the object storage by the microservices.

Once the data is retrieved, the microservices perform advanced analytics and machine learning tasks using custom algorithms tailored to specific requirements. These analytical processes transform the raw data into valuable insights, enabling more informed decision-making and deeper understanding of the underlying patterns and trends in the collected data. Right after the analysis is done, the results are stored in a separate object storage bucket, making them accessible to other services or users.

3.1.6 Cloud Server vs. SaaS Solutions

The choice of using a cloud server to host both the MQTT broker and the data analytics pipeline provides several advantages over SaaS solutions. While SaaS architectures have been considered, a cloud server offers more flexibility and control over the infrastructure, which is crucial for custom configurations and dependencies required by the data analytics pipeline. It is also more cost-effective for long-term projects, as SaaS solutions typically incur significant monthly fees. A cloud server allows better control over costs, with the trade-off being the need to manage infrastructure and security patches.

Furthermore, this solution is seamlessly implementable with any cloud provider since it doesn't rely on specific services unique to a particular provider. This cloud-agnostic approach allows users to choose the cloud provider that best fits their needs without altering the solution's architecture. Additionally, even though the MQTT broker and the data analytics pipeline are hosted on the same server, they remain independent, respecting the principles of a microservices architecture.

In summary, this architecture provides a robust, scalable, and flexible solution for data collection, processing, and storage, with the added benefit of being cloud-agnostic and cost-effective for long-term deployments.

3.2 Test Implementation

In this section the base implementation of the proposed solution is presented. The implementation is divided into two main components: the data collection component and the data analysis component. The data collection component is responsible for collecting data from IoT devices and archived data, while the data analysis component is responsible for processing and analyzing the collected data.

3.2.1 Data Collection Component

The data collection component is responsible for collecting data from IoT devices and archived data. This component consists of three main parts: the MQTT broker, the preprocessing pipeline and the data upload script.

MQTT Broker

MQTT broker is a lightweight messaging broker that implements the MQTT protocol. The broker is responsible for receiving messages from IoT devices and forwarding them to subscribers. The broker is hosted on a cloud server and is accessible to all IoT devices connected to the network. The broker is configured to use a specific topic for each device, ensuring that messages are delivered to the correct destination. The chosen MQTT broker for the test implementation is [EMQX](#)¹, whose pros and cons

¹[EMQX](#).

have been discussed in 2.2.6. Once the broker have been installed and configured in the cloud server, it was tested by connecting a number of simulated IoT device and sending messages to the broker. The messages were successfully received by the broker and forwarded to the subscriber, demonstrating the broker's functionality. The IoT devices used in the test were simulated using the **MQTTX**² client, analyzed in 2.2.5. This MQTT client allows users to simulate IoT devices and publish messages to an MQTT broker. The devices were configured to publish messages to the broker using a specific topic, with each message containing a timestamp and a set of key-value pairs representing the data.

Preprocessing pipeline

The preprocessing pipeline is responsible for parsing the JSON data received from the MQTT broker, extracting the timestamp and key-value pairs, and converting the data to CSV format. The pipeline is implemented as a Python script that subscribes to the MQTT broker, receives messages, and preprocesses the data. The script uses the **Paho MQTT**³ client library to connect to the broker and receive messages. Once the data is received, it is parsed, and the timestamp and key-value pairs are extracted. The data is then converted to CSV format and stored in object storage. The preprocessing pipeline ensures that the data is in a suitable format for further analysis and processing. The script was tested by connecting it to the MQTT broker and receiving messages from the simulated IoT devices. The messages were successfully parsed, and the data was converted to CSV format, demonstrating the pipeline's functionality.

Data Upload Script

The data upload script is responsible for uploading archived data from on-premises to the cloud. The script is implemented as a Python script that uses the cloud provider's API to upload files to object storage. The files contain historical data that have been collected over time in a number of different scenarios. The script reads the files from a local directory, uploads them to object storage, and makes them accessible to the data processing pipeline. In this scenario the files are already stored in CSV format, making them suitable for direct upload to object storage. The script was tested by uploading a number of sample files to object storage and verifying that the files were successfully uploaded in the right directory and accessible to the data processing pipeline.

3.2.2 Data Analysis Component

The data analysis component is responsible for processing and analyzing the collected data. This component can consist on one or more data analytics scripts based on the requirements of the project. Each of the scripts needs to retrieve the correct data from the object storage, perform the required analysis, and store the results in a separate object storage bucket. In the case more than one file needs to be retrieved from storage, the scripts need to manage the data retrieval and processing in parallel to optimize performance and resource usage. The scripts can be implemented in any programming language that supports the required data processing and analysis tasks, such as Python or R. The choice of language depends on the specific requirements of the project and the availability of libraries and tools for data processing and analysis.

²**MQTTX**.

³**Paho MQTT**. URL: <https://pypi.org/project/paho-mqtt/>.

This component was tested by implementing a simple data analysis script that retrieves the preprocessed data from object storage, calculates simple statistics, and stores the results in a separate object storage bucket. The script was implemented in Python using various Python libraries such as Pandas and NumPy for data manipulation and analysis. The script was tested by retrieving the preprocessed data from object storage, calculating the statistics, and storing the results in a separate object storage bucket. The results were successfully stored, demonstrating the functionality of the data analysis component. Once the analytics are uploaded they can be accessed via a web interface or a REST API, depending on the requirements of the project. The results can be visualized using various data visualization tools such as Matplotlib, Plotly, or Tableau, providing insights into the underlying patterns and trends in the collected data.

Chapter 4

Real world implementation

In this chapter, we will discuss the implementation of the proposed solution based on a real-world scenario presented to the company by a client during the internship period. The client is a company that specializes in high-performance protective gear and technical apparel for motorsports and action sports enthusiasts. The goal of the project was to develop a system that could analyze IMUs data from a rider's suit and provide feedback on the rider's performance. The client wanted to use this system to improve the rider's performance both for road users and motorsport pilots.

4.1 Client's requirements

The problem involves managing and processing data collected from various Motorcycle suits used by private and professional users. These products, used in different scenarios, generate data that needs to be efficiently collected, processed, stored, and analyzed to provide insights and ensure user safety and satisfaction. The solution must handle sporadic and intensive data uploads, support multiple types of users and devices, and ensure data integrity and accessibility.

User Types and Usage Patterns

- **Private Users with Sporadic Use:** Upload data approximately once a week.
- **Private Users with Intensive Use:** Upload data approximately three times a week.
- **Private Users with Multiple Products:** Use multiple products based on the scenario (e.g., road, track, off-road), with each of them offering different levels of protection. The products are classified into two categories:
 - **Lightweight Protection:** Daily usage, data uploads three times a week.
 - **Extended Protection:** Weekly usage (e.g. track or off-road during the weekend), data uploads once a week.
- **Professional Users:** Managed by authorized technicians, data uploads five times a week.

Data Uploads and Storage

- **File Size:**
 - Private users: 50 MB per upload.
 - Professional users: 250 MB per upload.
- **Data Volume:**
 - Weekly: 3.5 TB.
 - Yearly: 43 TB.

	Private Users		Professional Users	
	Sporadic Use	Intensive Use	Lightweight	Extended
Number of Users	30,000	10,000	1,000	30
Number of Uploads per Week	1	3	1	5
Number of Weeks per Year	16 (4 months)	8 (2 months)	8 (2 months)	36 (9 months)
Average File Size	50 MB		250 MB	
Overall Data Volume per Week	≈1.5 TB		≈250 GB	≈40 GB
Overall Data Volume per Year	≈24 TB	≈12 TB	≈2 TB	≈1.5 TB

Table 4.1: Data Volume Estimates

Data Handling and User Interaction

- **Data Transfer:**
 - Ensure data transfer completion even with battery constraints or disconnections.
 - Implement rules to ensure data transfer occurs only when the device is connected to a charger.
 - Manage data consistency and state at the firmware level.
- **Notifications:**
 - Notify private users of new data insights via push notifications or a dedicated portal.
 - Notify the R&D team for professional users.
- **Data Retention:**
 - Metrics stored permanently.
 - Raw data stored temporarily (2-4 weeks) unless a crash event is detected or for professional users.
 - Users have a time window to download and save their raw data.

4.2 System Architecture

Considering the requirements and usage patterns, the system architecture must be designed to handle the sporadic and intensive data uploads from different types of users. The architecture must be scalable, fault-tolerant, and platform-agnostic to support future growth and changes in the system. Two different architectures were proposed to meet the client's requirements: an on-demand analysis architecture and a preemptive analysis architecture. Both architecture are based on the same core components described in 3.1, but they differ in the way data is processed and analyzed. It's also worth mentioning that even if only [Aruba Cloud](#)¹ was considered for this project, the system is designed to be platform-agnostic and can be deployed on any cloud provider.

4.2.1 Architecture 1: Preemptive Analytics Processing

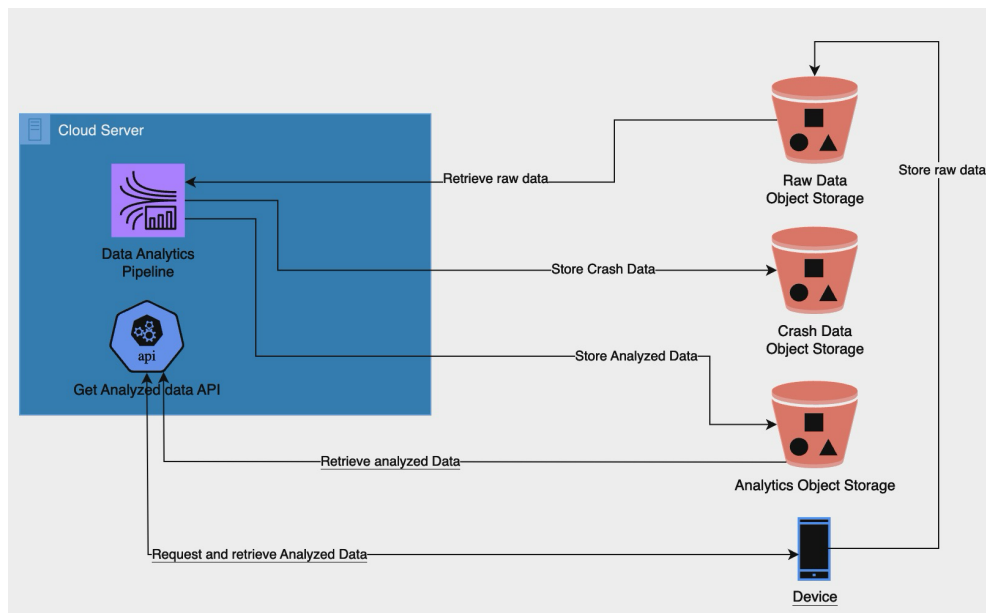


Figure 4.1: Architecture 1: Preemptive Analytics Processing

Overview

This architecture processes sensor data from biker suits as soon as it is received. The processed analytics are stored and readily available for users. This approach emphasizes real-time data processing to provide immediate insights and archive crash detection using Aruba's cloud services.

Detailed Workflow

Data Ingestion: Sensor data collected by the biker suit is uploaded to raw data object storage via app or dedicated hardware, ensuring a secure transfer and storage

¹ [Aruba Cloud](#).

of data for subsequent processing.

Trigger Analytics Pipeline: A push event triggers a function or script on the Aruba Cloud Server to initiate the analytics pipeline. Alternatively, the pipeline can be triggered by an API (HTTP or WebSocket) call upon data upload.

Data Processing: The Aruba Cloud Server runs the data analytics pipeline, which involves several key stages:

- **Data Preprocessing:** Cleaning and transforming raw data.
- **Analytics:** Applying algorithms to analyze the data.
- **Crash Detection:** Utilizing specific algorithms to detect crash events.

Storage: Processed analytics data is stored in a separate storage bucket within Aruba object storage. If a crash is detected, the raw data is duplicated in a crash-specific bucket. The original raw data is programmatically deleted to manage storage capacity efficiently.

User Access: Users are notified when analytics are available. The API Gateway provides an endpoint for users to request analytics data. The API retrieves the requested analytics data from the analytics bucket.

Data Listing: An API endpoint lists available analytics data based on metadata, facilitating easier management and retrieval for users.

Components

- **Aruba Object Storage:** For raw, analytics, and crash data storage.
- **API or Event-Driven Services on Aruba Cloud Server:** To trigger the analytics pipeline.
- **Aruba Cloud Server Instances:** For running the analytics pipeline.
- **APIs:** For data retrieval and listing.

Pros and Cons

Pros:

- Immediate availability of analytics data.
- Timely crash detection and data saving.
- Suitable for applications requiring real-time monitoring.

Cons:

- High resource usage due to continuous processing.
- Potentially high operational costs.
- Scalability challenges with high data volume.

4.2.2 Architecture 2: On-Demand Analytics Processing

Overview

This architecture processes sensor data on demand. Users request analytics for specific dates, triggering the processing pipeline. This approach is resource-efficient, processing data only when necessary using Aruba's cloud services.

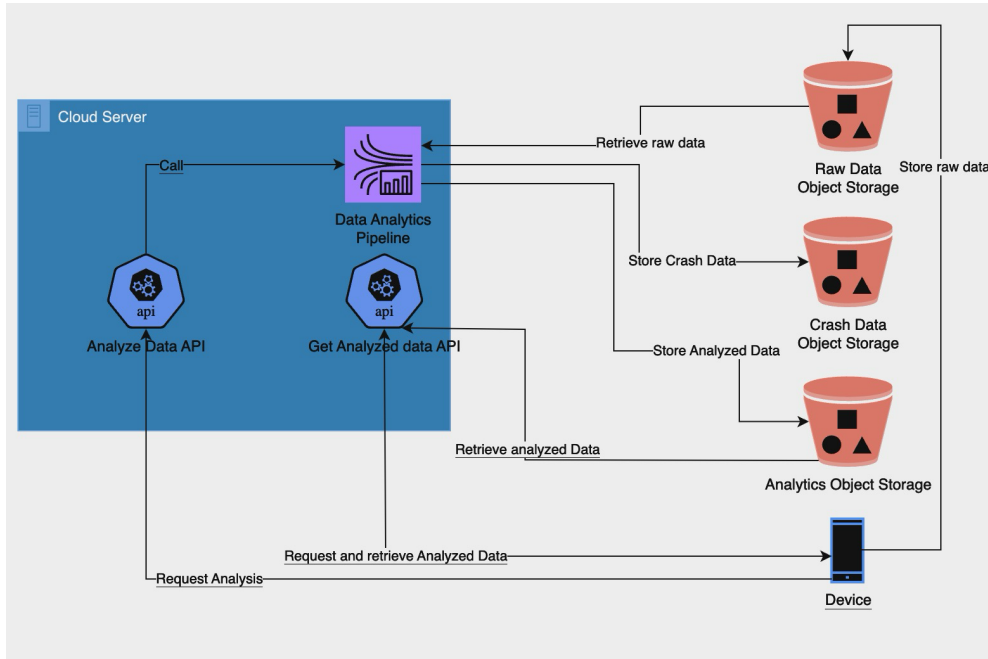


Figure 4.2: Architecture 2: On-Demand Analytics Processing

Detailed Workflow

Data Ingestion: The initial stage involves collecting sensor data from the biker suit. This data is uploaded to raw data object storage via mobile application or dedicated hardware, ensuring a secure transfer from the suit to a storage location where it can be accessed and processed.

Data Listing: After ingestion, the Listing API is utilized. This API endpoint lists all available raw data based on its metadata, facilitating easier management and retrieval for subsequent processing steps.

Analytics Request: The user requests data analytics through an API endpoint or a WebSocket at this stage. Upon receiving the request, a background job is triggered on the Aruba Cloud Server, initiating the analytics pipeline for processing the requested data.

Data Processing: Data processing involves several sub-stages on the Aruba Cloud Server:

- **Data Preprocessing:** Raw data is cleaned and transformed to prepare it for analysis.
- **Analytics:** Algorithms are applied to analyze the data and extract meaningful insights.
- **Crash Detection:** Specialized algorithms detect crash events within the data.

Storage: After processing, the resulting analytics data is stored in a designated analytics bucket within the Aruba object storage. If a crash is detected, the raw data is duplicated in a crash-specific bucket. To efficiently manage storage capacity, the original raw data is programmatically deleted after processing.

User Access: In the final stage, the processed data is made available to the user. Users are notified via WebSocket if they made the analytics request through this method. The API Gateway provides an endpoint for users to request the analytics data, which is retrieved from the analytics bucket and delivered to the user.

Components

- **Aruba Object Storage:** For raw, analytics, and crash data storage.
- **Job triggering API:** To handle on-demand processing.
- **Aruba Cloud Server Instances:** For running the analytics jobs.
- **APIs:** For data retrieval and listing.

Pros and Cons

Pros:

- Resource-efficient with on-demand processing.
- Easier to scale due to reduced continuous load.
- Cost-effective due to lower operational costs.

Cons:

- Delays in providing analytics data due to on-demand processing.
- Requires robust job scheduling and management.

4.2.3 Comparative Summary

Aspect	Architecture 1 (Preemptive)	Architecture 2 (On-Demand)
Data Processing	Continuous	On-demand
User Experience	Immediate insights	High delays
Resource Usage	High	Optimized
Scalability	Challenging	Easier
Cost Efficiency	Potentially costly	More cost-effective
Crash Detection	Immediate	On-demand

Table 4.2: Architecture Comparison

Both Architecture 1 (Preemptive Analytics Processing) and Architecture 2 (On-Demand Analytics Processing) have distinct advantages and considerations. Architecture 1 was chosen primarily for its emphasis on real-time data processing and immediate availability of analytics insights. This architecture ensures that sensor data from biker suits is processed as soon as it is received, enabling timely crash detection and data storage. The continuous processing nature of Architecture 1 supports applications requiring real-time monitoring, despite potential challenges in resource usage and scalability. Another important factor in selecting Architecture 1 is the client's focus on user experience and the need for immediate insights to enhance rider safety and performance.

4.3 Implementation

In this section, we delve into the implementation of the chosen system architecture, detailing the various steps undertaken to bring the proposed solution to life. Architecture 1, the Preemptive Analytics Processing architecture, was selected due to its emphasis on real-time data processing and immediate availability of analytics insights, aligning well with the client's focus on enhancing rider safety and performance. This section will outline the rationale behind choosing Architecture 1 and then walk through the implementation steps, ensuring each aspect of the system is comprehensively addressed.

4.3.1 Rationale for Choosing Architecture 1

Architecture 1 was chosen over Architecture 2 due to several key factors:

- **Real-time Data Processing:** Architecture 1 supports the immediate processing of sensor data as soon as it is received, providing timely insights and crash detection, which are critical for enhancing rider safety and performance.
- **Immediate Availability of Analytics:** Users can access processed analytics data almost instantly, ensuring that the insights are available when needed without delays.
- **User Experience:** The client prioritized user experience, particularly for professional users and high-intensity private users who require quick feedback on their performance.
- **Crash Detection:** The ability to detect crashes immediately and store relevant data for further analysis was a significant requirement that Architecture 1 could efficiently handle.

Given these considerations, Architecture 1 was the optimal choice for the client's needs, providing the necessary real-time capabilities and user experience enhancements.

4.3.2 Implementation Steps

The implementation of Architecture 1 involved several crucial steps, each designed to ensure the system met the client's requirements for data handling, processing, and user interaction. It's worth mentioning that the first implementation has been treated as a PoC (Proof of Concept), in the following steps description will be mentioned which steps have been implemented in the PoC and which ones are yet to be developed.

Data Ingestion

The initial step in the implementation process was to establish a robust data ingestion mechanism. Sensor data collected by the biker suits needed to be securely uploaded to raw data object storage.

Steps:

1. **Mobile Application Integration:** Develop or integrate a mobile application to facilitate data upload from the biker suit to the cloud. Ensure the app supports secure data transfer protocols.

2. **Dedicated Hardware:** If applicable, implement dedicated hardware to manage data upload directly from the suit to the cloud, bypassing the need for a mobile app.
3. **Secure Transfer:** Utilize secure transfer methods (e.g., HTTPS, SSL/TLS) to ensure data integrity and confidentiality during upload.
4. **Object Storage Setup:** Configure Aruba Object Storage to receive and store raw sensor data securely.

For the PoC, the Secure Transfer and Object Storage Setup steps have been implemented. To manage the file upload from the biker suit, a simple script was developed to simulate the data transfer process. The mobile application integration and dedicated hardware components will be considered in the next phase of development.

Trigger Analytics Pipeline

Upon successful data upload, an event or API call triggers the analytics processing pipeline on the Aruba Cloud Server.

Steps:

1. **Event-Driven Services:** Implement event-driven services using cloud functions or serverless architecture to trigger the analytics pipeline upon data upload.
2. **API Integration:** Develop APIs to allow manual or automated triggering of the analytics pipeline via HTTP or WebSocket calls.

For the PoC, the trigger mechanism was simulated using an API call at the end of the data upload script. Whether to keep this approach or to implement an event-driven service will be decided based on the system's performance and client feedback as well as the cloud provider's capabilities.

Data Processing

The core of the implementation involves processing the uploaded sensor data through various stages, including preprocessing, analytics, and crash detection.

Steps:

1. **Data Preprocessing:** Develop scripts or functions to clean and transform raw data, preparing it for analysis.
2. **Analytics Algorithms:** Implement algorithms to analyze the preprocessed data, extracting insights related to rider performance.
3. **Crash Detection Algorithms:** Develop specialized algorithms to detect crash events within the data, ensuring timely identification and response.

The PoC implementation of the analytics pipeline was developed as a background subroutine triggered by the API call at the end of the data upload script. In this phase a simple set of metrics was calculated, e.g. Average speed, maximum acceleration, number of left and right turn, maximum deceleration, etc. The next steps will involve refining the preprocessing, analytics, and crash detection algorithms to provide more detailed and actionable insights.

Storage of Processed Data

Processed analytics data needs to be stored in a dedicated storage bucket, with special handling for crash data.

Steps:

1. **Analytics Data Storage:** Configure a separate storage bucket within Aruba Object Storage for processed analytics data.
2. **Crash Data Handling:** Implement mechanisms to duplicate raw data in a crash-specific bucket if a crash is detected. Ensure original raw data is programmatically deleted post-processing to manage storage capacity efficiently.

For the PoC implementation, a separate storage bucket was set up to store the processed analytics data. The crash data handling mechanism will be developed in the next phase to ensure that crash events are detected and stored appropriately. In both cases the analytics and crash data are directly uploaded by the analytics pipeline described in the previous step.

User Access and Notification

Users must be notified when analytics are available and should have seamless access to retrieve their data.

Steps:

1. **User Notification:** Implement push notifications or WebSocket notifications to inform users when their analytics data is ready.
2. **API Gateway:** Set up an API Gateway to provide endpoints for users to request and retrieve analytics data.
3. **Metadata Listing:** Develop API endpoints to list available analytics data based on metadata, facilitating easier data management and retrieval.

For the PoC the API Gateway has been set up to provide an endpoint for users to request their analytics data. The metadata listing functionality has been implemented as an API endpoint to list available analytics data based on metadata. The user notification system will be developed in the next phase to ensure users are informed when their analytics data is ready for access.

Data Retention and Management

Ensure proper data retention policies are in place to manage storage efficiently while retaining essential metrics and crash data.

Steps:

1. **Permanent Metrics Storage:** Store key metrics permanently, ensuring long-term availability for analysis and user access.
2. **Temporary Raw Data Storage:** Implement policies to store raw data temporarily (2-4 weeks), unless a crash event is detected.
3. **User Download Window:** Provide users with a time window to download and save their raw data before it is deleted.

For the PoC, the permanent metrics storage has been implemented, ensuring that key metrics are stored securely for long-term access. The raw data cleaning has been developed as a batch process to manage storage capacity efficiently. The user download window functionality will be developed in the next phase to allow users to access and save their raw data before it is deleted.

4.3.3 Technical Detail and Tools

The implementation of the proposed solution involved the use of various tools and technologies to ensure the system's robustness, scalability, and efficiency. The following technical details provide an overview of the tools used in the implementation process:

Cloud server

Aruba cloud server instances were used to host the API Gateway, analytics pipeline, and other backend services. The cloud server provided a scalable and reliable environment for running the analytics pipeline and managing user requests.

Aruba Object Storage

Aruba Object Storage was used to store raw sensor data, processed analytics data, and crash data. It provides a secure and scalable storage solution for managing large volumes of data efficiently. The storage is divided into three main buckets:

- **Raw Data:** For storing the raw sensor data uploaded from the biker suits.
- **Analytics Data:** For storing the processed analytics data generated by the analytics pipeline.
- **Crash Data:** For storing the raw data in case a crash event is detected.

API Gateway

The API Gateway was used to provide endpoints for users to request and retrieve their analytics data. It acts as a central entry point for API requests, enabling seamless communication between users and the system. The API Gateway provide a set of different APIs:

- **Calculate Analytics:** An endpoint to trigger the analytics pipeline for a specific data set.
- **Retrieve Analytics:** An endpoint to retrieve the requested analytics data.
- **List Data:** An endpoint to list available data based on metadata, usable for RAW, Analytics and Crash data.

The API Gateway was first implemented using Python and FAST API, because of the simplicity and the speed of development. Since performance issues were pretty evident even with a small number of concurrent users, the API Gateway was then reimplemented using Java and Spring Boot and deployed on Wildfly (Executed as a Linux service), which provided better performance and scalability. A detailed comparison between Java and Python for API development can be found in [6.1](#), while a description on how to start an application as a Linux service can be found in [7.1](#). The implemented API Gateway already support TLS/SSL encryption and a basic

authentication system, but it will be further improved in the next phase to support more advanced security features.

Batch Processing

Batch processing was mainly used to clean the raw data and manage storage capacity efficiently. The batch process runs periodically to clean up old raw data and ensure that storage space is optimized. Batch processing was implemented using Python scripts and scheduled to run at specific intervals, leveraging Linux cron jobs for automation. A more detailed description about cron jobs can be found in [7.1](#).

4.4 Testing

After the implementation of the system, a series of tests were conducted to ensure that the system met the client's requirements and performed as expected. The testing phase involved various types of tests, including unit tests, integration tests, and performance tests, to validate the system's functionality, reliability, and scalability.

4.4.1 Unit Testing

Unit tests were conducted to verify the functionality of individual components of the system, such as upload scripts, API endpoints and data cleaning processes. Even if this is not a best-practice, unit tests were performed manually to ensure that each component worked as intended and produced the expected output. The unit tests were conducted in a controlled environment to isolate each component and identify any potential issues or bugs. In a future phase, a more comprehensive unit testing framework will be implemented to automate the testing process and ensure that all components are thoroughly tested.

4.4.2 Integration Testing

Integration tests were performed to validate the interactions between different components of the system, such as the API Gateway, analytics pipeline, and object storage. The integration tests focused on verifying that the components worked together seamlessly and communicated effectively to process and store data accurately. The integration tests were conducted in a test environment that replicated the production setup to ensure that the system would perform as expected in a real-world scenario. These tests were performed manually, but in the next phase, an automated integration testing framework will be implemented to streamline the testing process and ensure that all components are integrated correctly.

4.4.3 Performance Testing

Performance tests were conducted to evaluate the system's scalability, reliability, and responsiveness under various load conditions. The tests measured the system's throughput, latency, and resource utilization to identify bottlenecks or performance issues. An automated stress test script simulated a high number of concurrent users accessing the system and uploading data. Initial performance issues were identified with the API Gateway developed in Python, leading to a reimplementaion in Java and Spring Boot. The subsequent performance tests validated the improvements. It's

worth mentioning that these tests have been performed on a single server with a subset of users, in the next phase the system will be deployed on multiple servers to evaluate its performance in a distributed environment.

The results of the stress tests performed on the Python implementation of the API Gateway are not presented here, as they are not relevant to the final implementation. The results of the stress tests performed on the Java API Gateway are summarized in the following table:

Number of Users	Calculate Analytics	Retrieve Analytics	List Data
1	30 sec	221 ms	234 ms
10	85 sec (1 min 25 sec)	252 ms	228 ms
50	467 sec (7 min 47 sec)	327 ms	227 ms
250	2284 sec (38 min 4 sec)	268 ms	301 ms

Table 4.3: Performance Metrics for Calculating, Retrieving, and Listing Analytics

Analysis of Results

The performance metrics indicate how the system handles different numbers of concurrent users for three operations: Calculate Analytics, Retrieve Analytics, and List Data. Below is an analysis of the results:

- **Calculate Analytics:**
 - With 1 user, the operation takes 30 seconds.
 - As the number of users increases to 10, the time rises significantly to 85 seconds.
 - For 50 users, the time increases drastically to 467 seconds.
 - With 250 users, the operation takes 2284 seconds.
 - The significant increase in time with more users suggests that the system experiences a performance bottleneck when calculating analytics under high load, the performance bottleneck is mainly due to the high number of data being transferred from the object storage to the analytics pipeline in the server.
 - Even if the performance is not optimal, the 40 minutes required to calculate the analytics for 250 users is still acceptable for the client's requirements as the operation is not time-sensitive and would be performed while the users are charging their devices.
- **Retrieve Analytics:**
 - The response time for retrieving analytics is relatively stable, ranging from 221 ms to 327 ms as the number of users increases from 1 to 250.
 - The relatively minor variations in response time indicate that the system handles retrieving analytics efficiently, even under higher loads.
- **List Data:**
 - The response time for listing data remains relatively consistent, ranging from 227 ms to 301 ms as the number of users increases from 1 to 250.

- This consistency indicates that the system efficiently handles the listing of data without significant performance degradation under load.

Conclusion

The performance tests highlight that while the system handles retrieving and listing data efficiently under increasing loads, calculating analytics exhibits significant performance degradation as the number of users increases. Since the bottleneck is mainly due to the high number of data being transferred from the object storage to the analytics pipeline in the server, the next phase will focus on optimizing the data transfer process to improve the system's performance. The quickest solution would be to use more, less powerful servers to distribute the load. The system could also be optimized by saving raw data directly in the server, but this would require a significant amount of storage space and would not be scalable in the long run as well as the higher cost of the storage. The best solution would be to deploy the application in a containerized environment, using Kubernetes to manage the containers and distribute the load among different servers. This would allow the system to scale horizontally, adding more servers as needed, and would also provide a more efficient way to manage the data transfer process.

Chapter 5

Conclusion

In this chapter, the conclusions of the work are presented. The objectives achieved are discussed, as well as the future developments of the project. The author also reflects on what was learned during the development of the project.

5.1 Objectives achieved

The main goal of the project, as described in 1.2 was to develop a scalable cloud-agnostic architecture able to ingest data from multiple sources, process it and store it in a data lake. The architecture was successfully engineered, developed and tested, as described in 3. The sample architecture has been tested both with [Aruba Cloud](#)¹ and [Microsoft Azure](#)² to assess its cloud-agnostic nature. In chapter 4 a real world implementation of the base architecture was presented, showing the flexibility and scalability of the solution. The security constraints were also taken into account and obtained thanks to the shared responsibility model of the cloud providers and the use of encryption at rest and in transit.

5.2 Future developments

The future development of the project can be focused both on the base architecture described in chapter 3 and on the real world implementation described in chapter 4.

5.2.1 Base architecture

For the base architecture, the future developments can be focused on the following points:

- **Edge Computing:** the architecture can be extended to support edge computing. A useful feature would be to be able to train machine learning models on the cloud and deploy them on the edge devices.

¹*Aruba Cloud.*

²*Microsoft Azure.*

- **Test on other cloud providers:** the architecture can be tested on other cloud providers to assess its cloud-agnostic nature.

5.2.2 Real World Implementation

The real world implementation can benefit a lot more from future developments, since it's a real world use case of the base architecture. The future developments can be focused on the following points:

Data Ingestion

- **Mobile Application Integration:**
 - Develop a mobile application for data data management and analytics visualization.
 - Ensure secure data transfer protocols are implemented.
- **Dedicated Hardware:**
 - Implement dedicated hardware for direct data upload from suits to the cloud.

Trigger Analytics Pipeline

- **Event-Driven Services:**
 - Implement event-driven services (e.g., cloud functions or serverless architecture) to trigger the analytics pipeline upon data upload.

Data Processing

- **Advanced Data Preprocessing:**
 - Enhance the preprocessing scripts to handle more complex data cleaning and transformation tasks.
- **Enhanced Analytics Algorithms:**
 - Develop more sophisticated algorithms for analyzing preprocessed data and extracting detailed insights related to rider performance.
- **Crash Detection Algorithms:**
 - Develop specialized algorithms to detect crash events with higher accuracy.

User Access and Notification

- **User Notification System:**
 - Implement push notifications or WebSocket notifications to inform users when their analytics data is ready.

Data Retention and Management

- **User Download Window:**
 - Provide users with a time window to download and save their raw data before it is deleted.

Technical Detail and Tools

- **Security Enhancements:**
 - Enhance security features for API Gateway (e.g., advanced authentication, encryption).

Testing

- **Automated Unit Testing Framework:**
 - Implement an automated unit testing framework to ensure all components are thoroughly tested.
- **Comprehensive Integration Testing:**
 - Perform extensive integration testing to validate interactions between all system components.

Additional Enhancements

- **Performance Optimization:**
 - Optimize the performance of the analytics pipeline.
 - Conduct load testing to ensure the system can handle high data volumes and concurrent users.
- **Scalability Improvements:**
 - Enhance the system architecture to support future growth and increased data volume.
 - Deploy the system on a containerized platform for improved scalability and resource utilization.

5.3 Final considerations

The development of this project has been a great learning experience. I gained valuable insights into cloud computing, big data processing, and data analytics. The project has provided a hands-on opportunity to work with cutting-edge technologies and tools, and to apply theoretical knowledge to real-world scenarios. I have also learned about the challenges and complexities involved in designing and implementing a scalable and secure data processing system. The project has reinforced the importance of proper planning, design, and testing in software development, and has highlighted the need for continuous learning and adaptation in the fast-paced field of technology. Overall, the project has been a rewarding experience that has enriched my skills and knowledge in cloud computing and architecture designing. I am looking forward to applying these learnings in future projects and endeavors.

Chapter 6

Appendix A

6.1 Java vs Python for API Development

In the realm of API development, Java and Python are two of the most prominent programming languages, each offering distinct features, benefits, and trade-offs. This section will explore these languages, focusing on their usage in API development with Spring Boot for Java and FastAPI for Python.

6.1.1 Java and Spring Boot

Java is a statically-typed, object-oriented programming language that has been a staple in enterprise-level applications for decades. Its robustness, extensive libraries, and strong community support make it a reliable choice for large-scale systems.

Spring Boot is a framework designed to simplify the development of Java applications. It is part of the larger Spring Framework ecosystem and is widely used for building production-ready standalone applications with minimal configuration.

Pros of Java and Spring Boot:

- **Performance:** Java's statically-typed nature and JVM optimization result in high performance and efficient memory management, which is crucial for large-scale applications.
- **Scalability:** Java applications, particularly those built with Spring Boot, are known for their scalability. Spring Boot's support for microservices architecture allows for easy scaling and maintenance.
- **Security:** Java offers robust security features, and Spring Boot provides built-in security mechanisms, making it easier to develop secure APIs.
- **Mature Ecosystem:** Java has a mature ecosystem with a plethora of libraries, tools, and frameworks. Spring Boot, in particular, integrates seamlessly with other Spring projects and third-party tools.
- **Community Support:** Java's long-standing presence in the industry means it has extensive community support and documentation, which can be invaluable for troubleshooting and development.

Cons of Java and Spring Boot:

- **Complexity:** Java's syntax and the Spring Boot framework can be complex and verbose, leading to a steeper learning curve for beginners.
- **Configuration:** Although Spring Boot reduces the configuration overhead compared to traditional Spring applications, it can still be more cumbersome compared to the lightweight configurations in some other languages.

6.1.2 Python and FastAPI

Python is a dynamically-typed, interpreted language known for its simplicity and readability. Its versatility and ease of use have made it popular across various domains, including web development, data science, and automation.

FastAPI is a modern, high-performance web framework for building APIs with Python 3.7+ based on standard Python type hints. It is designed to be easy to use and offers automatic interactive API documentation.

Pros of Python and FastAPI:

- **Ease of Use:** Python's simple and readable syntax makes it accessible to beginners and allows for rapid development.
- **Fast Development:** FastAPI leverages Python's dynamic capabilities and type hints to provide features like automatic data validation and interactive API documentation, accelerating development.
- **Flexibility:** Python is highly flexible, and FastAPI's design allows developers to easily integrate with other libraries and tools.
- **Asynchronous Support:** FastAPI natively supports asynchronous programming, making it well-suited for applications requiring high concurrency.
- **Automatic Documentation:** FastAPI automatically generates interactive API documentation using Swagger UI and ReDoc, which is highly beneficial for development and testing.

Cons of Python and FastAPI:

- **Performance:** Python, being an interpreted language, is generally slower than compiled languages like Java. This can be a drawback for CPU-intensive tasks.
- **Scalability:** While Python applications can be scaled, it often requires more effort and optimization compared to Java applications. FastAPI, however, improves this aspect with its asynchronous capabilities.
- **Type Safety:** Python's dynamic typing can lead to runtime errors that are not caught at compile time, which can affect the reliability of the code.

6.1.3 Why Java is a Better Option

When comparing Java with Spring Boot and Python with FastAPI for API development, Java tends to be a better option for several reasons:

- **Performance and Efficiency:** Java's performance, aided by the JVM, is superior to Python. For API development, where high throughput and low latency are critical, Java's efficiency makes a significant difference.

- **Enterprise-Grade Scalability:** Java's robust ecosystem and the comprehensive features of Spring Boot make it well-suited for large-scale, enterprise-level applications. The scalability and maintainability of Java applications are generally higher, making them ideal for businesses expecting substantial growth.
- **Security:** Java's strong type system and Spring Boot's extensive security features provide a solid foundation for developing secure APIs. This is particularly important for applications handling sensitive data.
- **Community and Ecosystem:** The extensive community support and the mature ecosystem of libraries and frameworks in Java are crucial advantages. Developers have access to a wealth of resources, making it easier to find solutions to problems and ensuring long-term project viability.
- **Stability and Reliability:** Java's long history in enterprise environments has proven its stability and reliability. Businesses often prefer Java for mission-critical applications due to its consistent performance and predictable behavior.

While Python with FastAPI offers ease of use and rapid development, especially for smaller projects or prototyping, Java with Spring Boot stands out as a more robust, scalable, and secure choice for developing APIs, particularly in large-scale, performance-intensive, and enterprise-level scenarios.

Chapter 7

Appendix B

7.1 Linux Services and Cron Jobs

In this section, we will discuss the use of Linux services and cron jobs for automating tasks and managing processes on Linux-based systems. Understanding these tools is essential for maintaining the stability and efficiency of a server environment.

7.1.1 Linux Services

Linux services are background processes that start when the system boots and continue running without user intervention. They are managed by the init system, such as System V init or systemd. In modern Linux distributions, systemd is the most commonly used init system.

Creating a Linux Service with systemd

To create and manage a service using systemd, you need to create a unit file with a `.service` extension. This file contains configuration details about the service. Here is an example of how to create a simple service:

1. Create a unit file in the `/etc/systemd/system/` directory:

```
sudo nano /etc/systemd/system/myservice.service
```

2. Add the following content to the unit file:

```
[Unit]
Description=My Custom Service
After=network.target

[Service]
ExecStart=/usr/bin/python3 /path/to/your/script.py
Restart=always
User=nobody
Group=nogroup

[Install]
WantedBy=multi-user.target
```

Explanation of the sections:

- **[Unit]**: Contains general information about the service.
- **[Service]**: Defines how the service should be executed and managed.
- **[Install]**: Specifies the runlevels or targets at which the service should be enabled.

3. Reload the systemd manager configuration to recognize the new service:

```
sudo systemctl daemon-reload
```

4. Start the service:

```
sudo systemctl start myservice
```

5. Enable the service to start on boot:

```
sudo systemctl enable myservice
```

6. Check the status of the service:

```
sudo systemctl status myservice
```

7.1.2 Cron Jobs

Cron jobs are scheduled tasks that run at specified intervals. They are managed by the `cron` daemon, which reads configuration files known as `crontabs`. Each user, including the root user, can have their own crontab file. A useful tool for developing cron expressions is [Crontab Guru](https://crontab.guru/)¹.

Creating a Cron Job

To create a cron job, you need to edit the crontab file for the appropriate user:

1. Edit the crontab file:

```
crontab -e
```

2. Add a line with the schedule and command you want to run:

```
# Example of a cron job that runs a script every day at 2 AM
0 2 * * * /usr/bin/python3 /path/to/your/script.py
```

The schedule syntax is as follows:

```
* * * * * command_to_run
- - - - -
| | | | |
| | | | +---- Day of the week (0 - 7) (Sunday=0 or 7)
| | | +----- Month (1 - 12)
| | +----- Day of the month (1 - 31)
| +----- Hour (0 - 23)
+----- Minute (0 - 59)
```

3. Save and close the crontab file. The `cron` daemon will automatically recognize the changes and schedule the task.

Managing Cron Jobs

Here are some common commands for managing cron jobs:

- **List cron jobs:** `crontab -l`
- **Edit cron jobs:** `crontab -e`
- **Remove all cron jobs:** `crontab -r`

7.1.3 Conclusion

Using Linux services and cron jobs effectively allows for efficient automation and management of tasks and processes on a Linux server. Services managed by `systemd` provide a robust way to ensure essential background processes are always running, while cron jobs offer flexible scheduling for periodic tasks. Mastery of these tools is crucial for system administrators and developers working in Linux environments.

¹*Crontab guru.* URL: <https://crontab.guru/>.

Chapter 8

Bibliography

Article references

- Ficco, M. et al. “Federated learning for IoT devices: Enhancing TinyML with on-board training”. In: *Information Fusion* 104 (2024), p. 102189. ISSN: 1566-2535. DOI: <https://doi.org/10.1016/j.inffus.2023.102189>. URL: <https://www.sciencedirect.com/science/article/pii/S1566253523005055> (cit. on p. 27).
- Lin, Ji et al. “Mcunet: Tiny deep learning on iot devices”. In: *Advances in Neural Information Processing Systems* 33 (2020). URL: <https://arxiv.org/abs/2007.10319> (cit. on p. 26).
- Lin, Ji et al. “On-Device Training Under 256KB Memory”. In: *arXiv:2206.15472 [cs]* (2022). URL: <https://arxiv.org/abs/2206.15472> (cit. on p. 26).
- Ye, Yunfan et al. “EdgeFed: Optimized Federated Learning Based on Edge Computing”. In: *IEEE Access* 8 (2020), pp. 209191–209198. DOI: [10.1109/ACCESS.2020.3038287](https://doi.org/10.1109/ACCESS.2020.3038287) (cit. on p. 27).

Website references

- 221e S.r.l.* URL: <https://www.221e.com/> (cit. on p. 1).
- Alleantia.* URL: <https://www.alleantia.com/> (cit. on p. 23).
- Amazon Web Services.* URL: <https://aws.amazon.com/> (cit. on pp. 2, 7, 23).
- Apache Flink.* URL: <https://flink.apache.org/> (cit. on p. 15).
- Apache Kafka.* URL: <https://kafka.apache.org/> (cit. on p. 16).
- Apache Parquet.* URL: <https://parquet.apache.org/> (cit. on p. 13).
- Aruba Cloud.* URL: <https://www.arubacloud.com/> (cit. on pp. 2, 7, 35, 46).
- Aruba Documentation.* URL: <https://kb.arubacloud.com/en/home.aspx> (cit. on p. 7).
- Aruba Shared Responsibility Model.* URL: <https://kb.arubacloud.com/en/computing/use-and-technology/shared-responsibility-model.aspx> (cit. on p. 4).

- AWS Documentation*. URL: <https://docs.aws.amazon.com/> (cit. on p. 7).
- AWS Shared Responsibility Model*. URL: <https://aws.amazon.com/compliance/shared-responsibility-model/> (cit. on p. 5).
- Azure Documentation*. URL: <https://docs.microsoft.com/en-us/azure/> (cit. on p. 7).
- Azure Shared Responsibility Model*. URL: <https://learn.microsoft.com/en-us/azure/security/fundamentals/shared-responsibility> (cit. on p. 6).
- Crontab guru*. URL: <https://crontab.guru/> (cit. on p. 54).
- Eclipse Kura*. URL: <https://eclipse.dev/kura/> (cit. on p. 23).
- EMQX*. URL: <https://www.emqx.io/> (cit. on pp. 25, 30).
- Eurotech*. URL: <https://www.eurotech.com/> (cit. on p. 24).
- Microsoft Azure*. URL: <https://azure.microsoft.com/> (cit. on pp. 2, 7, 23, 46).
- MQTTX*. URL: <https://mqttx.app/> (cit. on pp. 25, 31).
- Paho MQTT*. URL: <https://pypi.org/project/paho-mqtt/> (cit. on p. 31).
- STMicroelectronics*. URL: https://www.st.com/content/st_com/en.html (cit. on p. 24).
- TensorFlow*. URL: <https://www.tensorflow.org/> (cit. on p. 25).
- TensorFlow Lite*. URL: <https://www.tensorflow.org/lite> (cit. on p. 25).
- TinyEngine*. URL: <https://github.com/mit-han-lab/tinyengine> (cit. on p. 26).
- Trust Radius*. URL: <https://trustradius.com/> (cit. on p. 7).

“Purtroppo gli scienziati non usano il POMPARGHEDRIO”

— *Diego Licciardello*

Dedicated to my family, my friends, and my mentors.