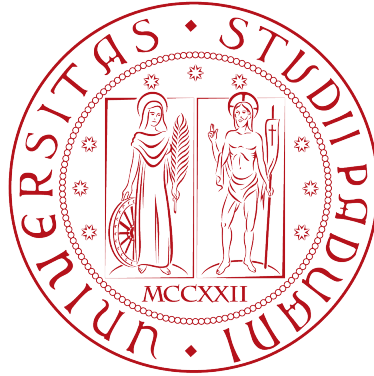


University of Padua

DEPARTMENT OF INFORMATION ENGINEERING

MASTER DEGREE IN ICT FOR INTERNET AND MULTIMEDIA



Scalable Cloud Agnostic Architecture for IoT
Data Analysis and Machine Learning

Master Thesis

Supervisor

Prof. Lorenzo Vangelista

Candidate

Alessandro Discalzi

ID 2088235

ACADEMIC YEAR 2023-2024

Summary

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 how an architecture like this can be designed and implemented, how it can be tested and how it can be used in a real-world scenario.

“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	3
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	10
2.1.3 Microsoft Azure	18
2.2 Present Solutions	23
2.2.1 Alleantia IoT Edge Hub	23
2.2.2 Eclipse Kura	24
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	26
2.3.2 Tiny Engine	26
2.3.3 Federated Learning and Transfer Learning	27
3 Proposed Solution	28
3.1 Architecture	28
3.1.1 Data Collection Layer	29
3.1.2 Data Storage Layer	29
3.1.3 Data Analysis Layer	30
3.1.4 Cloud Server (IaaS) vs. SaaS Solutions	30
3.1.5 Why the proposed architecture is cloud-agnostic	30
3.2 Test Implementation	31
3.2.1 Data Collection layer	31
3.2.2 Data Storage Layer	32

3.2.3	Data Analysis Component	32
4	PoC implementation for a real-world scenario	34
4.1	Client's requirements	34
4.1.1	User Types and Usage Patterns	34
4.2	System Architecture	36
4.2.1	Architecture 1: Preemptive Analytics Processing	36
4.2.2	Architecture 2: On-Demand Analytics Processing	38
4.2.3	Comparative Summary	39
4.3	Implementation	40
4.3.1	Rationale for Choosing Architecture 1	40
4.3.2	Implementation Steps	41
4.3.3	Technical Detail and Tools	43
4.4	Testing	44
4.4.1	Unit Testing	44
4.4.2	Integration Testing	45
4.4.3	Performance Testing	45
5	Conclusion	47
5.1	Objectives achieved	47
5.2	Future developments	47
5.2.1	Base architecture	47
5.2.2	Real World Implementation	48
6	Appendix A	50
6.1	Java vs Python for API Development	50
6.1.1	Java and Spring Boot	50
6.1.2	Python and FastAPI	51
6.1.3	Why Java is a Better Option	51
7	Appendix B	53
7.1	Linux Services and Cron Jobs	53
7.1.1	Linux Services	53
7.1.2	Cron Jobs	55
7.1.3	Conclusion	55
8	Bibliography	56

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
3.2	Example of an analytics CSV file stored in object storage	33
4.1	Architecture 1: Preemptive Analytics Processing	37
4.2	Architecture 2: On-Demand Analytics Processing	38

List of Tables

4.1	Data Volume Estimates	35
4.2	Architecture Comparison	40
4.3	Performance Metrics for Calculating, Retrieving, and Listing Analytics	45

Chapter 1

Introduction

1.1 The Company

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

221e operates under a dual-layer business model and 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.
2. Finished products, both general-purpose multi-sensor hardware platforms and a software for controlling the systems.

Targeting the Wearable Devices market and, more in general, the Internet of Things (IoT) industry, 221e capitalizes on the limitless applications within these fields.



Figure 1.1: 221e's logo

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

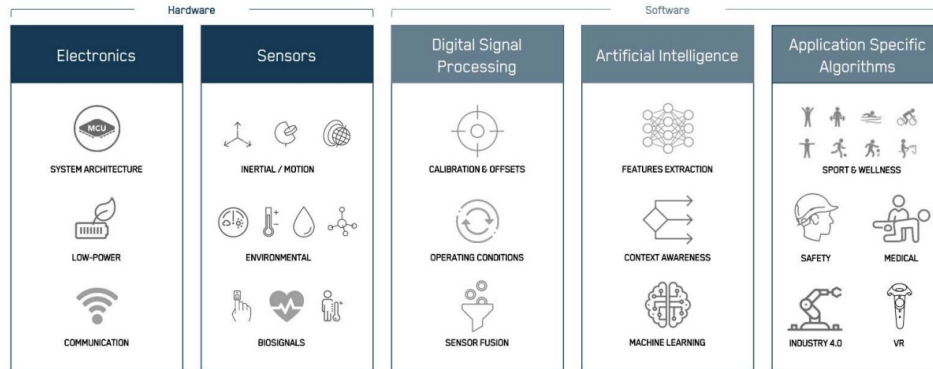


Figure 1.2: 221e's technology echosystem

1.2 Objectives and requirements

The idea of the project is to create a cloud-agnostic² architecture for a network of heterogeneous **IoT devices**³. The architecture needs to provide a way to ingest, to store and to analyze data ingested from multiple sensors. The architecture must also be designed to be able to scale horizontally⁴ and vertically⁵.

1.2.1 Cloud Infrastructure

The architecture needs to ingest, store and process large amounts of IoT data leveraging the power of cloud services. The final product to be developed is an architecture that can be deployed on any cloud provider. The main advantage of the architecture being cloud-agnostic is that since it can be deployed on any cloud provider, virtually without any modification each customer is able 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**⁶, **Amazon Web Services**⁷ and **Microsoft Azure**⁸. AWS⁹ and Microsoft Azure were chosen because of already developed experience of the company, and me. Aruba Cloud instead has been used because of a partnership between the company and the provider that started during the development of the project. Testing the architecture on at least two of the three providers can assess the cloud-agnostic nature of the architecture.

²Cloud Agnostic: Able to be deployed on every cloud provider effortlessly

³IoT Devices: Devices connected to the internet and equipped with a set of sensors.

⁴Horizontal scaling: capability of adding more machines or nodes to a system to handle an increase in workload

⁵Verical scaling: capability of increasing the capacity of a single machine or server by adding more resources such as CPU, RAM, or storage.

⁶Aruba Cloud. URL: <https://www.arubacloud.com/>.

⁷Amazon Web Services. URL: <https://aws.amazon.com/>.

⁸Microsoft Azure. URL: <https://azure.microsoft.com/>.

⁹AWS: Amazon Web Services

1.2.2 Data collection

The system must be able to ingest data from online devices and offline data sources such as an on-premise NAS¹⁰. Online devices are connected to the internet and can send data to the cloud via MQTT protocol. MQTT protocol is a lightweight messaging protocol designed for devices with low bandwidth and high latency. MQTT uses a publish-subscribe model: devices send messages to an MQTT broker, a software that receives and republish messages sent by IoT devices. This protocol is popular in IoT development because it's simple, easy to implement, and reliable. Different quality of service (QoS) levels can be used to ensure an effective message delivery. On-premise data sources consists of files stored on a local machine, the system must provide a way to upload these files to the cloud.

1.2.3 Data processing

The system must be able to preprocess the data before its storage. The preprocessing of the data includes data validation, cleaning and transformation which can be done both in the 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 provides a way to encrypt data at rest and manage the encryption keys.

Furthermore, each cloud provider uses a shared responsibility model for security. A shared responsibility model is an agreement that defines the responsibilities of the cloud provider and the customer for security. The responsibilities of the customer and the cloud provider vary depending on the service used. In general the cloud provider is responsible for the security of the cloud and the customer is responsible for security in the cloud. The shared responsibility model of each cloud provider is shown below, and a comparison with an on-premise solution is provided. It's important to keep in mind that the shared responsibility model of each cloud provider is subject to change and even if in general the responsibilities are the same, the details may vary.

¹⁰NAS: Network Access Storage

Aruba Shared Responsibility Model **Aruba Shared Responsibility Model**¹¹ is a model that defines the responsibilities of Aruba and the customer for security, in the image below the shared responsibility model is shown and compared with with an on-premise solution. The main focus of Aruba's shared responsibility model is to differentiate responsibilities of the customer and the provider in case of *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS) services and *Software as a Service* (SaaS) are used.

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



 Customer
 Aruba Cloud

Figure 1.3: Aruba Shared Responsibility Model

¹¹ *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**¹² is a model that defines the responsibilities of AWS and the customer for security, in the image below the shared responsibility model, with a clear division of responsibilities between the provider and the customer, is shown.

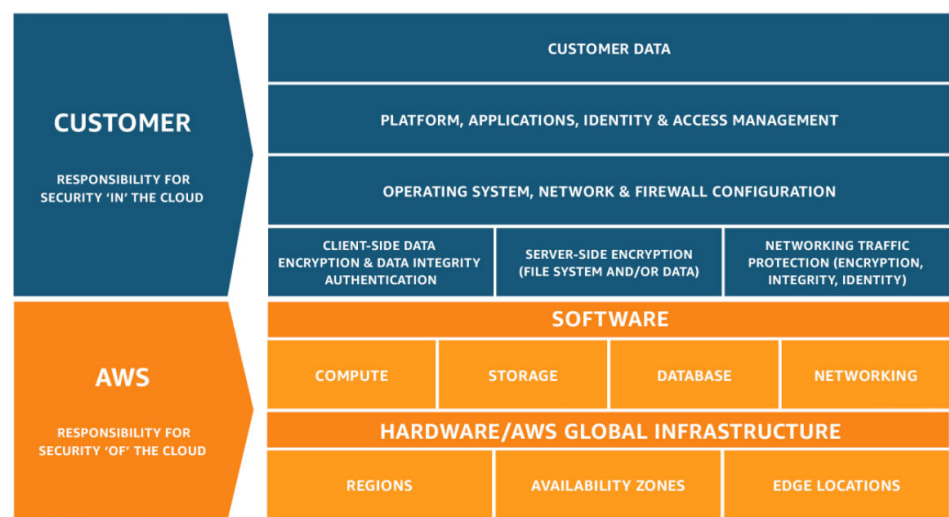


Figure 1.4: AWS Shared Responsibility Model

¹²AWS Shared Responsibility Model. URL: <https://aws.amazon.com/compliance/shared-responsibility-model/>.

Azure Shared Responsibility Model **Azure Shared Responsibility Model**¹³ also defines the responsibilities of Azure and the customer for security, in the image below the shared responsibility model is shown with the main focus of dividing the responsibilities of the customer and the provider in case of the usage of *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS) services and *Software as a Service* (SaaS) services.

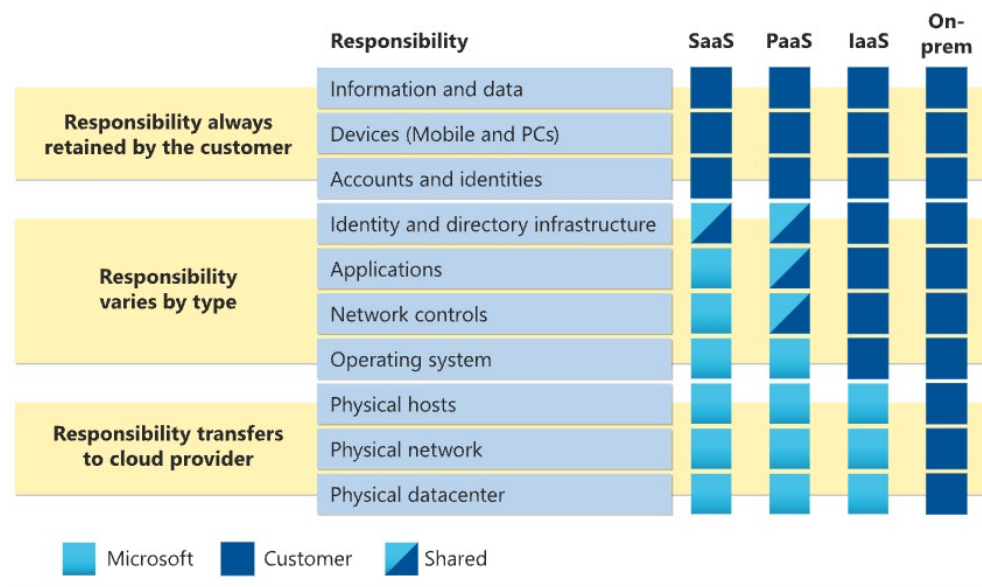


Figure 1.5: Azure Shared Responsibility Model

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.

¹³Azure 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 users reviews that can be found in [TrustRadius](#)⁷ website and my experience as well.

These are among the most used cloud services in the world, offering a wide range of features that can be used to develop the project. It's important to mention that when talking about cloud, the term *Service* can span from IaaS (Infrastructure as a Service) to SaaS (Software as a Service) and PaaS (Platform as a Service). More in general this term refers to something the user can access over the internet and that is hosted on the cloud provider's servers with a paid subscription. Another important concept to mention is that when referring to a *Fully Managed* service, it means that the cloud provider takes care of the infrastructure, the maintenance, the updates, and the security of the service. This allows the user to focus on the development of the application without worrying about the underlying infrastructure.

2.1.1 Aruba Cloud

Aruba Cloud is a cloud provider that offers a wide range of services like virtual machines, object storage, databases, and managed [Kubernetes](#)⁸ (an open-source platform for automating the deployment, scaling, and management of containerized applications

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/>.

⁸[Kubernetes](#). URL: <https://kubernetes.io/>.

across a cluster of machines).

Bare Metal

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

Pros:

- High performance
- 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 virtualizers⁹ such as OpenStack, VMware and Hyper-V.

Pros:

- High performance
- Reliable
- Cost-effective
- Supports multiple operating systems
- Scalable
- SLA¹⁰ up to 99.95%

Cons:

- Resources are guaranteed only in the professional plans, in the other plans resources are shared

⁹A virtualizer, or hypervisor, is software that allows multiple virtual machines (VMs) to run on a single physical machine

¹⁰SLA (Service Level Agreement): defines the level of service expected from the provider. It outlines specific metrics such as uptime, response times, and support availability, along with the responsibilities, performance standards, and consequences if the agreed-upon service levels are not met.

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 Storage¹¹ 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 defined as TB/month, this can be a problem for high traffic scenarios

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
- Autoscaling enabled
- Secure

¹¹Object storage: A data storage architecture that manages and stores data as discrete units called "objects." Each object contains the data itself, metadata, and a unique identifier.

- Cost-effective with respect to virtual machines
- Highly redundant
- Low latency
- Rapid deployment with respect to virtual machines
- Kubernetes native API (A set of API that lets you query and manipulate the state of Kubernetes objects)

Cons:

- Complex to use in case the user is not familiar with Kubernetes

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 with respect to on-premises solutions
- Cost-effective with respect to on-premises solutions
- Versatile: can run batch jobs (Jobs that perform analytics on the data) and more complex machine learning jobs (Jobs that train machine learning models)

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 edge¹².

Pros:

- Fully managed
- Flexible
- Native support for Retrieval Augmented Generation (RAG) models¹³

Cons:

- Costly
- Hard to future proof

¹²In this context edge devices are proper IoT devices connected to the internet

¹³Retrieval Augmented Generation (RAG). URL: <https://aws.amazon.com/it/what-is/retrieval-augmented-generation/>.

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 being consistent and performant.

Pros:

- Fully managed
- Fast p
- Predictable performance
- Scalable
- Highly available¹⁴
- NoSQL

Cons:

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

Elastic MapReduce (EMR)

AWS EMR is a big data platform that simplifies the deployment and management of big data frameworks, like [Apache Hadoop](https://hadoop.apache.org/)¹⁵ and [Apache Spark](https://spark.apache.org/)¹⁶ on AWS.

Pros:

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

Cons:

- Complex to use in case user is not familiar with big data frameworks
- Costly

¹⁴The availability of a service is how much time it is operational and accessible.

¹⁵*Apache Hadoop*. URL: <https://hadoop.apache.org/>, a library for distributed data processing.

¹⁶*Apache Spark*. URL: <https://spark.apache.org/>, an engine for distributed data processing.

Glue

AWS Glue is a fully managed ETL¹⁷ service that enables efficient data integration on a large scale.

Pros:

- Fully managed
- Pay per use
- Scalable
- Provides a centralised metadata repository¹⁸ which allows for automated data discovery and cataloguing across different data sources
- Supports different data sources and formats
- Allow for ETL job scheduling
- Data encryption

Cons:

- Costly for high workloads
- Performance issues with large datasets
- Complex to use in case the user is not familiar with ETL

Greengrass

AWS Greengrass is both a client software and a cloud service. The client software is a runtime deployable on edge devices while the cloud service allows for device management and runtime deployment. It also enables for data encryption both at rest and in transit and it can also extend device functionality with AWS Lambda functions, a more exhaustive description about AWS Lambda can be found in 2.1.2.

Pros:

- Edge computing
- Encryption at rest and in transit
- Extend device functionality with AWS Lambda functions
- Allows for machine learning at the edge

Cons:

- Restrained to AWS services, an architecture based on Greengrass is not platform agnostic by construction
- Resource intensive for small devices
- Need a connection for the initial setup

¹⁷ETL (Extracting, Transforming, and Loading): Is a job that extract data from a datasource, preprocess the data and loads it into another system

¹⁸AWS Glue data catalog. URL: <https://docs.aws.amazon.com/glue/latest/dg/catalog-and-crawler.html>.

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. The main difference with Greengrass is that IoT Core does not require a client runtime to be installed on the edge device.

Pros:

- Composed of multiple services so only the necessary ones can be used
- Encryption at rest and in transit
- Supports MQTT, HTTP, and WebSockets for commu
- Allows for device management
- Allows for machine learning at the edge
- It can be installed on devices

Cons:

- If installed on devices the architecture is not platform agnostic
- Can't be installed on all devices

Kendra

AWS Kendra is an enterprise search service that allows developers to add search capabilities across various content repositories leveraging built-in connectors¹⁹.

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 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/)²⁰

Pros:

¹⁹A connector in this scenario is an API like service that allows AWS kendra to ingest data from a variety of data sources. A database of prebuilt connectors is offered by AWS

²⁰*Apache Parquet*. URL: <https://parquet.apache.org/>.

- Fully managed
- Can read data from IoT core and Kinesis Data Streams
- Scalable
- Can load data into different AWS services
- Supports batching based on time, where the batch job is started every given amount of time
- Supports batching based on size, where the batch job is started when the size of the data reaches a certain threshold

Cons:

- Limited transformation capabilities
- Does not support batching based on other rules than time and size

Kinesis Data Streams

AWS Kinesis Data Stream is a service that simplifies 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
- Data are kept available for 24 hours by default

Cons:

- Limits on the data retention period

Lake Formation

AWS Lake Formation is a service that simplifies the creation, security and management of data lakes²¹.

Pros:

- Fully managed
- Scalable
- Simplifies lake creation process when compared to manual creation
- Simplifies data ingestion process when compared to manual ingestion
- Simplifies permission management when compared to a data lake implemented manually

²¹A data lake is a repository which allows to store both structured and unstructured data in a centralized way

- Provides data auditing
- Supports machine learning
- Supports data cataloguing

Cons:

- Costly with respect to manual implementation

Lambda

AWS Lambda is a serverless²² compute service that automatically manages the compute resources, scaling based on the workload. 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
- Supports multiple programming languages
- Easy to deploy and maintain
- Low time to market
- Supports custom made libraries as well as AWS libraries

Cons:

- Limited execution time (15 mins)
- Limited memory (10GB)
- Limited environment variables (4KB)
- Maximum 1000 concurrent executions
- Cold start
- Not cost-effective for high workloads, there is an exponential increase in cost when more RAM or CPU is needed

²²Serverless: a service that is offered without the provisioning of a server

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](https://flink.apache.org/)²³, an open-source framework that process streaming data. This service provides developers with the tools to build sophisticated, high-throughput, low-latency data processing applications. MSF automates infrastructure management, enabling teams to focus on application logic rather than operational problems. Its built-in scalability allows the service to handle varying workloads efficiently, making it suitable both for batch and stream processing.

Pros:

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

Cons:

- Complex to use in case the user is not familiar with Apache Flink

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](https://kafka.apache.org/)²⁴ 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 capabilities without the burden of managing Kafka infrastructure.

Pros:

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

Cons:

- Local testing challenges: hard to replicate the same environment in production and locally
- Not suitable for high-traffic scenarios
- Complex to use in case the user is not familiar with Apache Kafka

²³*Apache Flink*. URL: <https://flink.apache.org/>.

²⁴*Apache Kafka*. URL: <https://kafka.apache.org/>.

Sage Maker

AWS SageMaker is a cloud-based machine learning platform that enables developers to build, train, and deploy machine learning models at any scale. SageMaker provides a suite of tools that simplify each step of the machine learning workflow, from data labelling and preparation to model tuning and deployment. The platform supports various machine learning frameworks and has a set of built-in algorithms that can leverage its managed infrastructure to 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.

Pros:

- Scalable
- Highly available
- Secure
- Durable: redundance is built in
- Cost-effective with respect to on-premises solutions or file system storage
- 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 an object storage service. 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 that could be suitable for different workloads

Cons:

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

Cosmos DB

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

Pros:

- Fully managed
- Scalable
- Support multiple models of data (key-value, document, column-family, graph)
- Data are available globally
- Offers multiple consistency levels: Levels of consistency define how up-to-date and synchronized data is across a distributed system at the cost of performance
- Easy to set up

Cons:

- Expensive
- Queries can be slow if not run on the indexes

DataBricks

Azure Databricks is an 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 for machine learning and data processing (TensorFlow²⁵, PyTorch²⁶, Scikit-learn²⁷, etc.)
- Open data lakehouse²⁸ platform

Cons:

- Cost can rise exponentially with big-data processing
- Complex to set up and use in case the user is not familiar with Apache Spark

Data Explorer

Azure Data Explorer is a service providing real-time and high-volume data analytics. 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, the user needs to know the data structure at prior to ingest data

²⁵ *TensorFlow*. URL: <https://www.tensorflow.org/>.

²⁶ *PyTorch*. URL: <https://pytorch.org/>.

²⁷ *Scikit-learn*. URL: <https://scikit-learn.org/stable/>.

²⁸ An open data lakehouse is a platform or architecture that conjugates the pros of a data lake and a data warehouse

Data Factory

Azure Data Factory is data integration service. It provides tools to orchestrate data workflows while monitoring executions.

Pros:

- Fully managed
- Scalable
- Can perform data Analytics using Synapse, an Azure service that allows for data warehousing and big data processing

Cons:

- Complex configuration

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 Apache Hadoop²⁹
- Supports Python for data analytics

Cons:

- Data governance challenges, setting up the right user permissions can be challenging

Event Grid

Azure Event Grid is an event routing³⁰ service that simplifies the development of event-driven applications.

Pros:

- Fully managed
- Scalable
- Supports MQTT5
- Supports event sources from other Azure services as well as custom sources

²⁹ *Apache Hadoop*.

³⁰ A service that manages pub-sub messages consumption

- Supports multiple event types
- Supports multiple programming languages
- Supports multiple event patterns

Cons:

- Complexity
- Limitations in event storage and retention (7 days)
- Considering the cost it's not convenient for an architecture needing basic event routing

Event Hubs

Azure Event Hubs is a ingestion service. It can be used to stream millions of events per second with low latency, from multiple sources and to any destination.

Pros:

- Fully managed
- Scalable
- Secure
- Low latency
- Supports Apache Kafka
- Schema registry: centralize repository for schema management³¹
- 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 computing service enable code to be runned in response to events without the need to provision or manage the infrastructure.

Pros:

- Fully managed
- Pay per use
- Scalable

³¹An event broker usually ingest event data from different sources. For each source a schema is needed to deserialize the data.

- Supports multiple programming languages
- Easy to deploy and maintain
- Low time to market: the user can focus on the code and not on the infrastructure
- Supports custom libraries: the user can upload custom libraries to be used in the function

Cons:

- Limited execution time (10 minutes)
- Cold start problem: the first time a function is called it takes longer to start up
- Not cost-effective for high workloads: there is an exponential increase in cost when more RAM or CPU is needed

IoT Hub

Azure IoT Hub is a cloud service that serves as the bridge between IoT devices and 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. It also offers a client runtime that can be installed on edge devices.

Pros:

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

Cons:

- The client runtime is not platform agnostic: once it's installed only Azure services can be used
- Not well documented
- Costly
- Does not fully support MQTT5: only a subset of the protocol's features is supported

Machine Learning

Azure Machine Learning is a 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 capabilities³²
- Pay as you go

Cons:

- Cost rises when training big models

2.2 Present Solutions

In this section are presented the solutions that are currently available on the market and that could be integrated into the architecture engineering process.

2.2.1 Alleantia IoT Edge Hub

IoT Edge Hub is Alleantia³³'s plug-and-play solution for the industrial IoT.

Pros:

- Plug and play
- Device management: allows to manage and update devices remotely
- Alarms and events
- Log management
- Report generation
- Integration with Microsoft Azure

Cons:

- Platform dependent
- Does not support Amazon Web Services

³²MLOps is a practice that introduce DevOps techniques in the machine learning development workflow

³³Alleantia. URL: <https://www.alleantia.com/>.

2.2.2 Eclipse Kura

Eclipse Kura³⁴ is an open-source IoT Edge Framework that serves as a platform for building IoT gateways. It's based on Java/OSGi³⁵ and it provides API access to the hardware interfaces of IoT Gateways³⁶.

Pros:

- Open source: allows for customization
- Platform agnostic
- Allows for flexible and modular development
- Provides access to hardware interfaces via APIs
- Introduces AI capabilities at the edge

Cons:

- Computational complexity is high for less powerful devices
- Not well documented

2.2.3 Eurotech Everyware Cloud

Eurotech³⁷ Everyware Cloud is a IoT Integration Platform with a microservices architecture that allows to connect, configure and manage IoT gateways and devices.

Pros:

- Cloud-based
- Allows to connect, configure and manage IoT gateways and devices
- Supports multiple protocols
- Supports multiple cloud providers (AWS and Azure)

Cons:

- Last update in 2019: the platform could be outdated considering the pace of Cloud services development

2.2.4 STMicroelectronics X-Cube Cloud

STMicroelectronics³⁸ X-Cube Cloud is a software package that enables the connection of STM32 microcontrollers to the cloud. STM32 are a family of 32-bit microcontrollers developed by STMicroelectronics and are among the most used microcontrollers in the IoT field.

Pros:

- Supports multiple cloud providers

³⁴Eclipse Kura. URL: <https://eclipse.dev/kura/>.

³⁵A framework that enhances Java's modular programming capabilities

³⁶An IoT gateway is a physical device that enable sensors communication towards the internet

³⁷Eurotech. URL: <https://www.eurotech.com/>.

³⁸STMicroelectronics. URL: https://www.st.com/content/st_com/en.html.

- Offers a secure connection to the cloud
- Supports multiple protocols

Cons:

- Specific for STM32 microcontrollers
- A specific version of the software is needed for each provider if the user want to use all the features
- A generic version of the software is available but it has limited features and works only with a subset of microcontrollers

2.2.5 MQTTX

MQTTX³⁹ is a cross-platform MQTT 5.0 client tool that can be used to publish and subscribe to MQTT messages.

Pros:

- Connection management
- Log capabilities
- Data pipelines
- Device simulation capabilities: Can create a virtual device to test the connection

Cons:

- Data pipelines feature is not well documented, the pipeline creation process is not clear and there are low debug capabilities

2.2.6 EMQX

EMQX⁴⁰ is an open-source MQTT broker designed to be highly scalable.

Pros:

- Supports MQTT 5.0
- Supports web sockets
- Supports multiple cloud services

Cons:

- Even though it's open-source it's not free software, the user needs to pay for the enterprise version to access all the features

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.

³⁹*MQTTX*. URL: <https://mqttx.app/>.

⁴⁰*EMQX*. URL: <https://www.emqx.io/>.

2.3.1 Tensorflow Lite

Tensorflow Lite⁴¹ is the mobile and edge version of **Tensorflow**⁴² that allows to run machine learning models on edge devices.

Pros:

- Lightweight
- Supports multiple operating systems both mobile and edge
- Easy to use
- Well documented

Cons:

- On-device training is limited to Unix-based systems, only inference is supported on edge devices

2.3.2 Tiny Engine

Tiny Engine^{43,44} is a specialized machine learning framework designed to build, train, and deploy models on edge devices. Tiny Engine can utilize 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:

- Supports multiple platforms
- Allows to convert Tensorflow Lite models to C++
- Easy deployment of pre-trained models

Cons:

- Need to pre-train the model before deploying it
- MCUnet⁴⁵ is the only well documented model
- Custom models are hard to deploy and there is a lack of documentation

⁴¹ *TensorFlow Lite*. URL: <https://www.tensorflow.org/lite>.

⁴² *TensorFlow*.

⁴³ *TinyEngine*. URL: <https://github.com/mit-han-lab/tinyengine>.

⁴⁴ Ji Lin et al. "On-Device Training Under 256KB Memory". In: *arXiv:2206.15472 [cs]* (2022). URL: <https://arxiv.org/abs/2206.15472>.

⁴⁵ 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>, An algorithm for deep learning on microcontrollers.

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 computational resources are limited. As described in “EdgeFed: Optimized Federated Learning Based on Edge Computing”⁴⁶, 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”⁴⁷, 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
- Predictions made on the edge reduce latency

Cons:

- Complicated to implement
- Limited capabilities compared to centralized training

⁴⁶Yunfan 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).

⁴⁷M. 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

Proposed Solution

In this chapter, the proposed solution is presented. The chapter is divided into two sections: architecture, and implementation. The architecture section describes how a cloud-agnostic architecture capable of collecting, processing, and storing data from IoT devices and archived data can be designed. The implementation section present a test implementation of the proposed solution that demonstrate the feasibility of the architecture and its cloud-agnostic nature.

3.1 Architecture

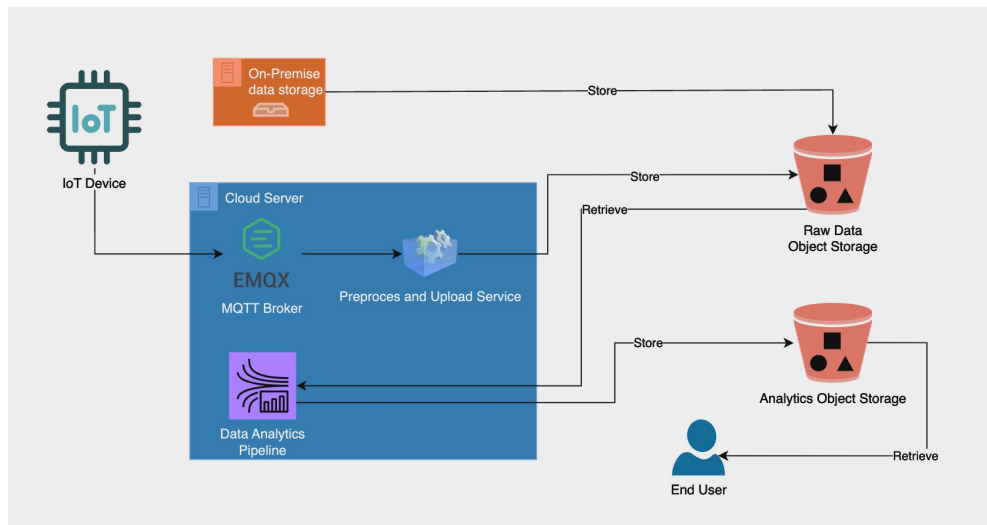


Figure 3.1: Architecture of the proposed solution

The proposed solution's architecture consists of a composition of loosely coupled microservices¹² that work together to collect, process, and store data. The architecture

¹Loosely coupling is an approach in programming that aims to limit as much as possible the dependencies between different components

²Microservices is an architectural style that consists in dividing a large application in many smaller

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. A schema representing the architecture is shown in figure 3.1.

3.1.1 Data Collection Layer

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

Data Collection from IoT Devices

In our setup, numerous IoT devices can send data to the cloud via MQTT to a specific MQTT broker. 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 topic³ for each device, ensuring that messages are delivered to the correct destination. Each device is configured to publish data on a specific topic, which is managed by a cloud-based broker. The data containing a timestamp and key-value pairs representing sensor readings is sent in JSON⁴ format. Once the data reaches the broker, it is preprocessed before storage. This preprocessing step involves parsing the JSON data, extracting the timestamp and sensor values, and converting the data to CSV format⁵. CSV was chosen for its simplicity, broad compatibility with data processing tools, and because it's less verbose than JSON. Finally, the preprocessed data is stored in object storage, where it can be accessed by the data processing pipeline.

Data Collection from Archived Data

To collect data from archived sources, files from on-premises systems are uploaded to the cloud using a straightforward script that leverages cloud's API. Since these files are already in CSV format no preprocessing is needed, so they are directly uploaded to object storage, making them readily accessible for the data processing pipeline. These files hold historical data that has been gathered over time across different scenarios.

3.1.2 Data Storage Layer

The data storage layer is designed to store raw and analyzed CSV data. It ensures that raw data are available for the processing pipeline and that analyzed data are stored for future use. Raw data and analysis outputs are saved in different buckets⁶ to ensure a less complicated data management pipeline. This layer can leverage on any Object Storage service offered by any provider since all of them are cost effective, scalable and redundant.

parts that are able to work independently from one another

³A topic, when talking about MQTT protocol, is defined by a string. Each device when publishing messages will use a specific topic. All the subscribers of that specific topic are able to see the messages.

⁴JSON is a popular data exchange format based on JavaScript programming language. It's widely used in API development and IoT fields. It usually consists of key-value pairs. Each value can be an atomic value, an object or even an array of objects.

⁵CSV (Comma Separated Values) is a lightweight format for exchanging data. It usually consists of an header and a number of rows where each column is separated by the colon character and each row by the newline character.

⁶In Object Storage, a bucket is a container able to store object within a specific namespace.

3.1.3 Data Analysis Layer

The data analysis layer is dedicated to examining the collected data and generating meaningful insights. This layer comprises multiple services that retrieve preprocessed data from object storage to carry out various analytical tasks. The data is first downloaded from the object storage to a local storage leveraging the object storage APIs. This needs to be done since Object Storage services do not allow to access the content of the files directly.

Once the data is retrieved one or more analytics services run to analyze the data. These services can leverage simple statistical assessments as well as more advanced machine learning and deep learning techniques. The analysis process extracts valuable insights that can be showed to the users of a certain application to enhance their experience. The analyzed data are then saved in the object storage again, but in a different bucket making them accessible for future use by other services or users. There is the possibility to start the analysis automatically when new data is uploaded to the object storage, or to trigger it manually for example implementing a set of APIs⁷ that can be called by the user. This is an implementation choice that depends on the specific requirements of the project and that cannot be decided at a general architectural level.

3.1.4 Cloud Server (IaaS) vs. SaaS Solutions

The usage of a cloud server to host both the MQTT broker and the data analytics pipeline offers many advantages with respect to a SaaS solution. A SaaS architectures can be convenient with respect to a IaaS one when talking about management and responsibilities. On the other hand, an IaaS based architecture allows for more flexibility and control of the underlying infrastructure. This can be especially useful in case that custom configurations and dependencies are needed for the data analytics pipeline. A cloud server allows for better cost management with the only downside being the responsibility of managing the operating system and applying security patches. In contrast, SaaS solutions, are generally easier to set up and maintain but can also have higher costs over time due to higher subscription fees. The cloud server approach is also inherently cloud-agnostic since every provider offers an IaaS service. This aspect ensures that users can select the cloud provider that best fits with their requirements without needing to change the architectural design of the solutions. With this solution, the MQTT broker and the data analytics pipeline are hosted on the same server but they function independently so the principles of microservices architectures are respected. In summary, this architecture is robust, scalable, and flexible. Its cloud-agnostic nature allows for an easy implementation with any provider in the market. Furthermore, the cost-effective design can be particularly important for long-term projects. This approach not only meets the immediate operational needs but also positions organizations to adapt and evolve as their requirements change over time.

3.1.5 Why the proposed architecture is cloud-agnostic

The proposed architecture is designed to work with any cloud provider. Considering the cloud services described in 2.1, the architecture can be implemented using any of providers since each of them offers some Object Storage and some computing services. The data storage layer in the design uses an Object Storage service to store raw and

⁷An API (Application Programming Interface) is a piece of software that aims to facilitate the information exchange between a server and some other software.

preprocessed data. This layer can be implemented using any cloud provider's Object Storage service, as all providers offer similar services. The data analysis layer uses a cloud server to run the data processing pipeline. This layer can be implemented using any cloud provider's IaaS service and with every operative system, as all providers offer similar services. The data collection layer instead uses a MQTT broker to collect data from IoT devices. This layer can be implemented using any cloud provider's MQTT broker service, as well as on a computing service where a MQTT broker can be installed. Considered the fact that the same base architecture can be implemented using any cloud provider, the proposed solution can be considered cloud-agnostic.

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 layer, the data storage layer and the data analysis layer. 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. In this implementation to test the cloud agnostic nature of the architecture, each layer has been implemented using both Azure and Aruba. The implementation is based on the architecture described in 3.1 and aims to demonstrate the feasibility of the proposed solution.

3.2.1 Data Collection layer

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 that we have chosen for the test implementation **EMQX**, whose pros and cons have been discussed in 2.2.6. Once the broker was been installed and configured in the cloud server, it was tested by connecting some simulated IoT devices which was then 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** client, analyzed in 2.2.5. This MQTT client allows users to simulate IoT devices and publish messages to an MQTT broker. The simulated devices were configured to publish JSON messages to the broker using a specific topic, with each of the messages 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 thanks to **paho-mqtt**⁸, an MQTT client library for Python, subscribes to the MQTT broker,

⁸ *Paho MQTT*. URL: <https://pypi.org/project/paho-mqtt/>.

receives messages, and preprocesses the data. 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 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 several sample files to object storage and verifying that the files were successfully uploaded to the right directory and accessible to the data processing pipeline.

3.2.2 Data Storage Layer

The data storage layer is responsible for storing raw and analyzed data. This layer ensures that raw data is available for the processing pipeline and that analyzed data is stored for future use. The data storage layer consists simply of an object storage service. Each cloud provider analyzed in 2.1 offers an object storage service with a set of APIs that allow users to store and retrieve data, since the API can vary from provider to provider, all the scripts interacting with the storage itself have been developed to be able to work with both Azure and Aruba. The data storage layer was tested by uploading raw and analyzed data to object storage and verifying that the data was successfully stored and accessible to the data processing pipeline.

3.2.3 Data Analysis Component

The data analysis component is responsible for processing and analyzing the collected data. This component can consist of 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 sequentially to optimize performance and resource usage. The scripts can be implemented in any programming language that supports the required data processing and analysis tasks. The choice of language depends on the specific requirements of the project and the availability of libraries and tools for data processing and analysis. 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

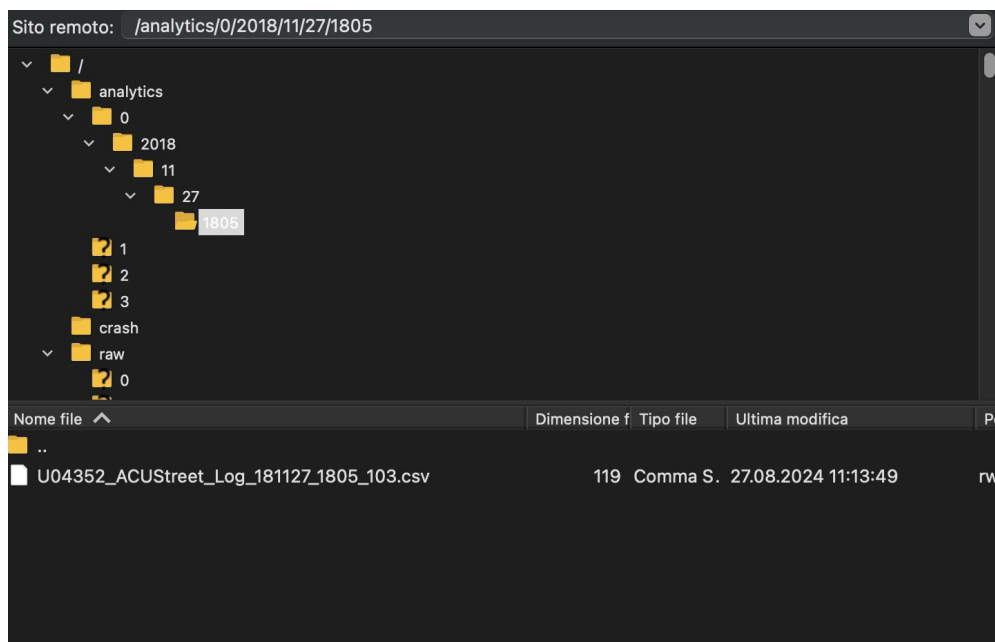


Figure 3.2: Example of an analytics CSV file stored in object storage

API, depending on the requirements of the project. The results can be visualized using various data visualization tools such as [Matplotlib](https://matplotlib.org/)⁹, [Plotly](https://plotly.com/)¹⁰, or [Grafana](https://grafana.com/)¹¹, providing insights into the underlying patterns and trends in the collected data.

⁹ *Matplotlib*. URL: <https://matplotlib.org/>.

¹⁰ *Plotly*. URL: <https://plotly.com/>.

¹¹ *Grafana*. URL: <https://grafana.com/>.

Chapter 4

PoC implementation for a real-world scenario

In this chapter, we will discuss the implementation of the proposed solution based on a real-world . 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 IMU data from a rider's suit. An Inertial Measurement Unit (IMU) is a set of sensor able to measure inertial data. Usually an IMU consists of a 3-axis acellerometer, gyroscope and magnetometer. The goal is to provide feedback on the rider's performance based on analysis performed on the IMUs data. 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.

4.1.1 User Types and Usage Patterns

In this section the different types of users and their usage patterns are described. The users are divided into four categories: Private Users with Sporadic Use, Private Users with Intensive Use, Private Users with Multiple Products, and Professional Users. Each category has specific usage patterns and data upload requirements. The estimated number of users and data volume per year are also provided in table 4.1. The implementation of the system will be based on these usage patterns to ensure that the system can handle the expected data volume and user interactions.

- **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 architectures are based on the same core components described in 3.1 maintaining the same core components thus ensuring that the system is scalable, fault-tolerant, and platform-agnostic. The choice between the two architectures will be based on the client's specific needs and priorities. Both options propose the usage of a set of APIs to facilitate the interaction between the various components.

4.2.1 Architecture 1: Preemptive Analytics Processing

Overview

The architecture shown in figure 4.1 is engineered to process 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 cloud services.

Detailed Workflow

Data Ingestion: Sensor data collected by the biker suit is uploaded to raw data object storage via a mobile application or via a custom firmware installed on the suit's charging station, ensuring a secure transfer and storage of data for subsequent processing.

Trigger Analytics Pipeline: An upload event triggers a function or script on the Cloud Server to initiate the analytics pipeline. Alternatively, the pipeline can be triggered by an API call upon data upload.

Data Processing: The 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.

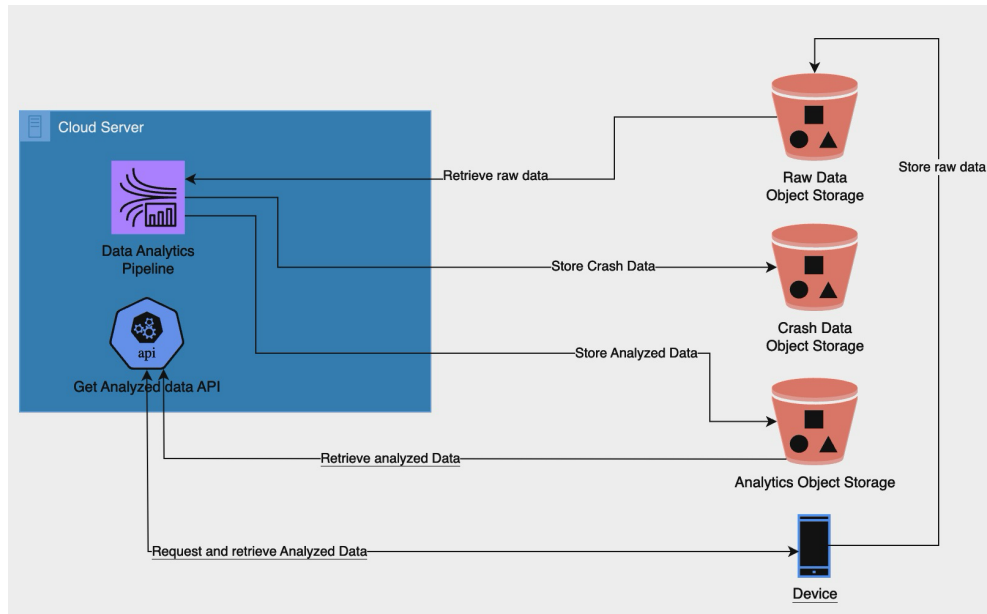


Figure 4.1: Architecture 1: Preemptive Analytics Processing

- **Crash Detection:** Utilizing specific algorithms to detect crash events.

Storage: Processed analytics data is stored in a separate storage bucket within the 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 must be able to access analytics data upon request. This is done by providing an API endpoint capable of retrieving the requested data 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

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

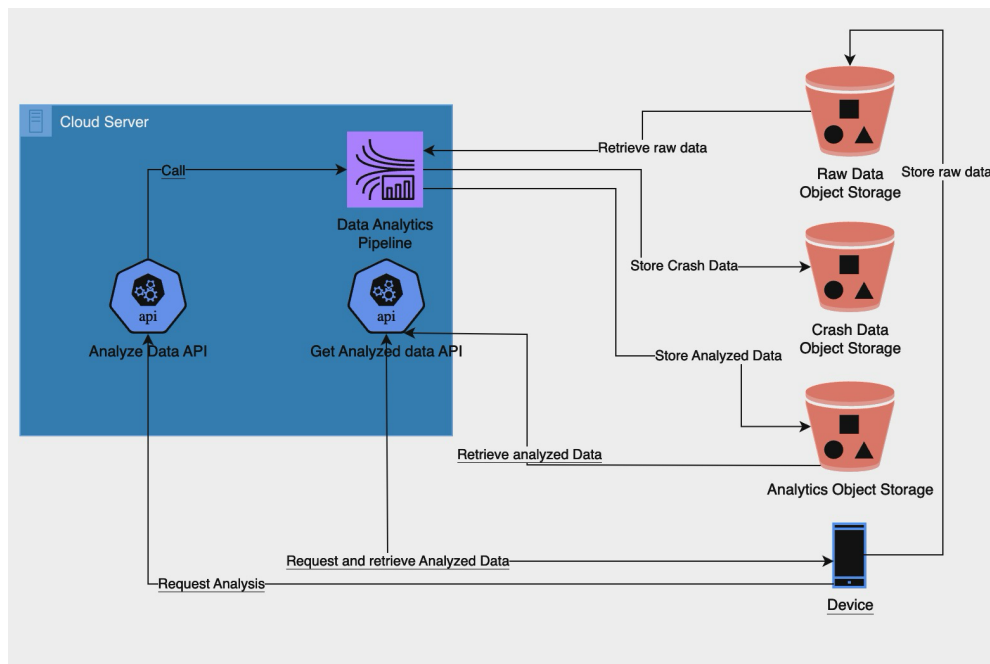


Figure 4.2: Architecture 2: On-Demand Analytics Processing

Overview

The architecture schematized in figure 4.2 process sensor data on demand. Users can request ride analytics processing for a specific period of time, triggering the processing pipeline. This approach is resource-efficient, processing data only upon user request.

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 Cloud Server, initiating the analytics pipeline for processing the requested data.

Data Processing: Data processing involves several sub-stages on the 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 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

- **Object Storage:** For raw, analytics, and crash data storage.
- **Job triggering API:** To handle on-demand processing.
- **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 expensive	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 the 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 the immediate availability of analytics insights, aligning well with the client's focus on enhancing rider safety and performance. This section outlines the rationale behind choosing Architecture 1 and then gives a walk through the various 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 description of the following steps 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 the 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 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 turns, 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 the 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

Aloud server instance 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.

Object Storage

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 of 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 of 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 reimplementation 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 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.

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 used 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. A study of the pros and cons of each language and framework will help in determining which one is better suited for API development. Besides the personal experience, the information reported here is based on general industry trends and best practices.

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 variety 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. A non-experienced developer may find it challenging to set up.

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 variety 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 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, especially in large-scale and performance-intensive 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. “Mcnnet: 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).
- Apache Flink*. URL: <https://flink.apache.org/> (cit. on p. 16).
- Apache Flink*. URL: <https://flink.apache.org/>.
- Apache Hadoop*. URL: <https://hadoop.apache.org/> (cit. on pp. 11, 20).
- Apache Kafka*. URL: <https://kafka.apache.org/> (cit. on p. 16).
- Apache Parquet*. URL: <https://parquet.apache.org/> (cit. on p. 13).
- Apache Spark*. URL: <https://spark.apache.org/> (cit. on p. 11).
- Aruba Cloud*. URL: <https://www.arubacloud.com/> (cit. on pp. 2, 7, 47).

- 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 Glue data catalog*. URL: <https://docs.aws.amazon.com/glue/latest/dg/catalog-and-crawler.html> (cit. on p. 12).
- 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. 55).
- Eclipse Kura*. URL: <https://eclipse.dev/kura/> (cit. on p. 24).
- EMQX*. URL: <https://www.emqx.io/> (cit. on p. 25).
- Eurotech*. URL: <https://www.eurotech.com/> (cit. on p. 24).
- FastAPI*. URL: <https://fastapi.tiangolo.com/>.
- Grafana*. URL: <https://grafana.com/> (cit. on p. 33).
- Kubernetes*. URL: <https://kubernetes.io/> (cit. on p. 7).
- Matplotlib*. URL: <https://matplotlib.org/> (cit. on p. 33).
- Microsoft Azure*. URL: <https://azure.microsoft.com/> (cit. on pp. 2, 7, 47).
- MQTTX*. URL: <https://mqttx.app/> (cit. on p. 25).
- Paho MQTT*. URL: <https://pypi.org/project/paho-mqtt/> (cit. on p. 31).
- Plotly*. URL: <https://plotly.com/> (cit. on p. 33).
- PyTorch*. URL: <https://pytorch.org/> (cit. on p. 19).
- Retrieval Augmented Generation (RAG)*. URL: <https://aws.amazon.com/it/what-is/retrieval-augmented-generation/> (cit. on p. 10).
- Scikit-learn*. URL: <https://scikit-learn.org/stable/> (cit. on p. 19).
- Spring Boot*. URL: <https://spring.io/projects/spring-boot>.
- STMicroelectronics*. URL: https://www.st.com/content/st_com/en.html (cit. on p. 24).
- TensorFlow*. URL: <https://www.tensorflow.org/> (cit. on pp. 19, 26).
- TensorFlow Lite*. URL: <https://www.tensorflow.org/lite> (cit. on p. 26).
- TinyEngine*. URL: <https://github.com/mit-han-lab/tinyengine> (cit. on p. 26).
- Trust Radius*. URL: <https://trustradius.com/> (cit. on p. 7).