Project Title	AN OPEN, TRUSTED FOG COMPUTING PLATFORM FACILITATING THE DEPLOYMENT, ORCHESTRATION AND MANAGEMENT OF SCALABLE, HETEROGENEOUS AND SECURE IOT SERVICES AND CROSS-CLOUD APPS
Project Acronym	RAINBOW
Grant Agreement No	871403
Instrument	Research and Innovation action
Call / Topic	H2020-ICT-2019-2020 / Cloud Computing
Start Date of Project	01/01/2020
Duration of Project	36 months

D4.2 – Data Management Services

Work Package	WP4 – RAINBOW Data Management Services
Lead Author (Org)	Demetris Trihinas (UCY)
Contributing Author(s) (Org)	M. Symeonides, G. Pallis, M. D. Dikaiakos (UCY); A-V. Michailidou, T. Toliopoulos, G. Vlahavas, A. Gounaris (AUTH); S. Kousiouris, S. Venios (SUITE5)
Reviewers	John Kaldis (USYS), John Avramides (INTRA)
Due Date	31.03.2022
Actual Submission	31.03.2022
Version	1.0

Dissemination Level

X	PU: Public (*on-line platform)
	PP: Restricted to other programme participants (including the Commission)
	RE: Restricted to a group specified by the consortium (including the Commission)
	CO: Confidential, only for members of the consortium (including the Commission)

The work described in this document has been conducted within the project RAINBOW. This project has received funding from the European Union's Horizon 2020 (H2020) research and innovation programme under the Grant Agreement no 871403. This document does not represent the opinion of the European Union, and the European Union is not responsible for any use that might be made of such content.

Versioning and contribution history

Version	Date	Author	Notes
0.0	04.02.2022	Demetris Trihinas (UCY)	Deliverable structure and content placeholders
0.1	18.02.2022	Demetris Trihinas (UCY)	Introduction, data processing SOTA
0.2	25.02.2022	Demetris Trihinas (UCY), Moysis Symeonides (UCY)	Fog analytics SOTA, Chapter 4 requirements
0.3	04.03.2022	Demetris Trihinas (UCY), Moysis Symeonides (UCY)	Chapter 4 overview and scheduling algorithms
0.4	11.03.2022	Moysis Symeonides (UCY)	Chapter 5 requirements
0.5	14.03.2022	Thodoris Toliopoulos (AUTH)	Chapter 3 content
0.6	16.03.2022	Demetris Trihinas (UCY), Moysis Symeonides (UCY), George Pallis (UCY), Marios Dikaiakos (UCY), Sotiris Koussiouris (SUITE5), Stefanos Venios (SUITE5)	Chapter 4 and 5 finalized, conclusion.
0.7	18.03.2022	Thodoris Toliopoulos (AUTH), Anna Valentini Michailidou (AUTH), George Vlahavas (AUTH), Anastasios Gounaris (AUTH)	Chapter 3 finalized.
0.8	20.03.2022	Demetris Trihinas (UCY), Moysis Symeonides (UCY)	Document ready for internal review
0.9	29.03.2022	Giannis Kaldis (USYS), John Avramides (INTRA)	Review – minor corrections
1.0	31.03.2022	Demetris Trihinas (UCY)	Document finalized and ready for submission

Table of Contents

List of tables

List of figures

Executive Summary

The purpose of Deliverable D4.2 is to provide a thorough report on the final release of the RAINBOW Data Management Services which are designed and developed within the scope of Work Package 4 (WP4). The services designed within WP4, enhance the RAINBOW ecosystem with intelligent data management services, such as data storage and sharing mechanisms (T4.1), which can be deployed alongside the fog continuum so that analytic insights are extracted from fog services via geo-distributed data processing (T4.2) with the use of high-level analytic query abstractions (T4.3).

This deliverable provides a comprehensive overview of the purpose and functionality of each RAINBOW service involved, presenting architectural decisions and functional design that has been developed/improved since the early release of the services and documented in D4.1. Notable new features include (i) the Storage Fabric designed and implemented on top of the overlay mesh network interconnecting through the fog node local storage agents to provide decentralized coordination for the querying of monitoring data (T4.1); (ii) the design and implementation of fog-aware analytics job scheduling algorithms that enable users to denote various optimization strategies and exploit trade-offs between optimization criteria (T4.2); and (iii) the implementation of the streaming analytics query model packaged as an analytics job compiler that can be used on different distributed data processing backends (T4.3).

The deliverable then proceeds with a report on the validation and fulfilment of the requirements set in D1.1 and extended in depth in D4.1 to expand the requirements mapped to WP4. Finally, this deliverable concludes by providing in the form of an Appendix the documentation made available at the deliverable release for the three services developed within WP4.

Table of Abbreviations

5G	Fifth Generation
ACID	Atomicity, Consistency, Isolation, and Durability
API	Application Programming Interface
AST	Abstract Syntax Tree
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DBMS	Database Management System
DSL	Domain-Specific Language
Dx.x	Deliverable x.x
EBNF	Extended Backus–Naur Form
GB	Gigabyte
GHz	Gigahertz
HTTP	HyperText Transfer Protocol
I/O	Input/Output
ILP	Integer Linear Programming
IoT	Internet of Things
KPI	Key Performance Indicator
MG	Megabyte
MIMO	Multiple-Input Multiple-Output
ML	Machine Learning
MR	Map Reduce
MS	Millisecond
NP	Non-deterministic Polynomial-time
QoS	Quality of Service
RAM	Random Access Memory
REST	Representational State Transfer
SLO	Service Level Objective
SOTA	State of the Art
SQL	Structured Query Language
SSL	Secure Sockets Layer
TLS	Transport Layer Security
Tx.x	Task for Work Package x.x
UC	Use Case
vCPU	Virtual Central Processing Unit
VPN	virtual private network
VXLAN	Virtual Extensible Local Area Network
W	Watt
WP	Work Package

1 Introduction

Deliverable 4.2, henceforth simply referred to as D4.2, provides a comprehensive overview and documentation report for the final version of the RAINBOW Data Management Services that are developed within the scope of Work Package 4 (WP4). These services are integral for the broad vision of RAINBOW as they contribute to various key features that the RAINBOW ecosystem has to offer to IoT application developers and operators during the deployment and execution of their applications.

Specifically, the **Distributed Data Storage and Sharing service**, contributes to providing persistent storage of monitoring data on the fog nodes themselves. This is a vital feature as data collected in the fog continuum does not move for storage and processing which in the end, may compromise data privacy and will have an imminent effect in the performance of analytic computations that must be completed in time for mission-critical services (i.e., vehicle traffic control). To achieve this, the Distributed Data Storage and Sharing service features a high-performance indexing scheme that enabled the querying of any fog node of the topology for data (data location transparent to user). Towards this, we have given the service the name "Storage Fabric". Key features that contribute to its ability to retrieve data with low-latency (stable algorithmic complexity), replicate data across multiple nodes, and partition data schemas for high-availability, will be introduced in Chapter 3.

The next key service of the RAINBOW Data Management layer is the **RAINBOW Analytics Stack** which is responsible for providing low-latency data processing across geo-distributed realms. This service is developed on top of the popular and open-source Apache Storm framework streaming big data analytics.

There are three key novelties developed along with the RAINBOW Analytics Stack. The first entails the design of novel analytic job Schedulers that extend the capabilities of Storm to optimize streaming jobs by acknowledging the limitations that fog environments feature. These are the dynamicity of the environment itself (fog nodes added/removed), resource heterogeneity, network uncertainty, fog nodes that end up with low data quality and fog nodes that are battery powered. Towards this, several Schedulers have been designed as part of the **Distributed Data Processing service** with the intend for users to simply specify through the RAINBOW Dashboard what should be optimized without having to code the "how", leaving this burden to the RAINBOW-enabled Schedulers.

The second novelty entails the design of a high-level and declarative query model, packaged under the **Fog Analytics service** that supports the abstraction of analytics from

real-time monitoring data to ease the description and programmability of continuous analytic jobs. This query model is completely decoupled from the underlying distributed processing engine to promote the reuse of analytics jobs. In turn, the compilation of analytic jobs provides some initial optimizations that attempt to reduce the unnecessary computation (and distribution over the network) of intermediate query results that are a significant overhead in geo-distributed environments.

The third novelty involves the testing the analytics stack of data-intensive IoT services. This is achieved through the design of an emulation framework that enables the rapid provisioning of emulated testbeds to evaluate "what-if" scenarios in consolidated environments so that KPIs can be assessed under extreme conditions, including network uncertainty, entity mobility, load fluctuations, node failures and connectivity degradation.

1.1 Document Purpose and Scope

The purpose of this deliverable is to provide a comprehensive overview and documentation report of the second -and final- release of the RAINBOW Data Management Services which contribute to providing interoperable analytic capabilities to fog-enabled deployments via intelligent data storage and processing mechanisms capable of operating in the fog continuum and on top of trusted overlay mesh networks. In respect to this, D4.2 aims to derive a clear overview of the final implementation and feature validation of the three components comprising the RAINBOW Data Management Services and are developed under the umbrella of WP4, namely: (i) the Distributed Data Storage and Sharing Service; (ii) the Distributed Data Processing Service; and (iii) the Fog Analytics Service. To this end, D4.2 documents for each component of the RAINBOW Data Management layer their new functionalities and improvements since the early release of these services along with their documentation. In addition, D4.2 provides a requirements fulfilment report of the functional requirements of each component.

Finally, we note that D4.2 is partially based on a number of scientific papers, which introduce core concepts of the components that are part of the RAINBOW Data Management Services and WP4. These papers, are highlighted below:

Table 1: Scientific Papers Published within WP4 Scope

WP4 Scientific Papers	RAINBOW Partners
Data Placement in Dynamic Fog Ecosystems. T. Toliopoulos, A-V Michailidou, A. Gounaris, **IEEE International Workshop on Self-Managing Database Systems (SMDB),** May 2022	**AUTH**
[Under submission] Explainable Distance-based Outlier Detection in Data Streams, T. Toliopoulos, A. Gounaris, **VLDB 2022**	**AUTH**
BenchPilot: Repeatable & Reproducible Benchmarking for Edge Micro-DCs. J. Georgiou, M. Symeonides, M. Kasioulis, D. Trihinas, G. Pallis and M. D. Dikaiakos. **2022 IEEE Symposium on Computers and Communications (ISCC)**, June 2022.	**UCY**
Demo: Emulating 5G-Ready Mobile IoT Services. M. Symeonides, D. Trihinas, G. Pallis and M. D. Dikaiakos. **In 2022 ACM/IEEE International Conference on Internet-of-Things Design and Implementation (IoTDI '22),** May 2022	**UCY**
5G-Slicer: An emulator for mobile IoT applications deployed over 5G network slices. M. Symeonides, D. Trihinas, G. Pallis and M. D. Dikaiakos, Constantinos Psomas and Ioannis Krikidis. **In 2022 ACM/IEEE Conference on Internet-of-Things Design and Implementation (IoTDI '22),** May 2022	**UCY**
PROUD: PaRallel OUtlier Detection for Streams. T. Toliopoulos, C. Bellas, A. Gounaris, and A. Papadopoulos. **In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)**. Association for Computing Machinery, New York, NY, USA, 2717–2720.	**AUTH**
Fogify: A Fog Computing Emulation Framework. M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis and M. D. Dikaiakos, **2020 IEEE/ACM Symposium on Edge Computing (SEC)**, San Jose, CA, USA, 2020, pp. 42-54.	**UCY**
[Best Demo Award] Emulating Geo-Distributed Fog Services. M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis and M. D. Dikaiakos, **2020 IEEE/ACM Symposium on Edge Computing (SEC)**, San Jose, CA, USA, 2020, pp. 187-189.	**UCY**
A Self-stabilizing Control Plane for Fog Ecosystems. Z. Georgiou, C. Georgiou, G. Pallis, E. M. Schiller, and D. Trihinas, **In 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC),** pp. 13-22, December 2020.	**UCY**

1.1 Document Relationship with other Deliverables and Work Packages

This deliverable is built on the foundation of D4.1 that provided an initial report on the design and implementation of the RAINBOW Data Management Services, while also highlighting the requirements that must be satisfied to overcome the challenges introduced when deploying data storage and analytics services in the fog continuum. The requirements list is a by-product of the initial requirement mapping introduced in D1.1. To this end, D4.2 provides a report on the requirement validation and accomplishment, while also serving as detailed documentation on the design, implementation, and release of the final features of the RAINBOW Data Management Services based on the updated RAINBOW architecture as introduced in D5.3.

1.2 Document Structure

The rest of this deliverable is structured as follows: Chapter 2 provides an updated state-of-the-art analysis in respect to the key technology axes relevant to the contributions of WP4. Chapter 3 presents the final version of the RAINBOW Distributed Data Storage and Sharing service. In turn, Chapter 4 presents the final version of the RAINBOW Distributed Data Processing service, while Chapter 5 presents the RAINBOW Fog Analytics Service, which capitalizes on the Distributed Data Processing Service to provide a high-level query language for fast submission of optimized streaming analytic jobs for fog deployments. Finally, Chapter 6 concludes this deliverable and outlines future directions.

1.3 Interim Review Comments

D4.2 is structured in a way that it takes into account all reviewer comments in relevance to the overall project and specifically to WP4. Towards this, D4.2 features:

- An updated state-of-the-art in relevance to the novel aspects proposed in WP4. In addition to this, a clear description is provided in Chapter 2 (SOTA) of what is the "RAINBOW Approach" in terms of advancing the SOTA.
- A more technical approach has been followed in the way each WP4 service is presented in terms of its architecture and place in the RAINBOW ecosystem. This has been done without repeating content from the DoA and with the least amount of content overlap with D4.1.
- All RAINBOW requirements from D1.1 that have been mapped to WP4 services (D4.1) have been assessed and are all successfully achieved.
- The Fogify emulator for data-intensive IoT applications that has been presented during the interim review but was not described in D4.1, is documented in this deliverable.

2 State of the Art and Key Technology Axes Challenges

In this Section, we will update the State-of-the-Art presented in D4.1. Particularly, we present SOTA work in the respective areas with focus on the new functionality and algorithms that are introduced in D4.2 highlighting the key differences of the contributions made by the services developed within the scope of RAINBOW WP4.

2.1 Geo-Distributed Data Storage and Sharing

Edge computing frameworks, like RAINBOW, are highly distributed as hundreds or even thousands of devices that are placed at multiple locations. These devices need to access persistently stored data, modify and save them to a database. Moreover, real-time analysis on-the-fly is crucial when dealing with Edge Computing applications and streaming data, for example the analysis of sensor data. When choosing a Database Management System for such scenario, new challenges arise compared to a centralized solution.

Scalability, is crucial in Edge Computing as we deal with tens or hundreds of devices, as mentioned previously. High scalability also increases the elasticity of the DBMS and the ability to handle workload changes. Scalability in such scenarios is horizontal by means of adding more devices when demand for resources arises [1] [2]. Couchbase [1] provides multi-dimensional scaling that scales queries, indexes and data, supporting more than one hardware profile, resulting in isolation of services.

The reliability and fault-tolerance of a distributed DBMS is often achieved through data replication, i.e., copies of data that are stored in multiple devices [1] [3]. The master-slave model is used in multiple DBMSs, whereas multiple replication types like transactional, snapshot or merge and schemas like full or partial exist. Multi-master replication is preferable when dealing with multiple devices. Amazon Aurora [2], ArangoDB [3], CouchDB [4], PostgreSQL [5] and Redis [6] provide this feature. The Multi-master paradigm also increases the availability and response time of the DBMS. Also, distributed DBMSs must handle more aspects regarding concurrency and recovery, when compared to a centralized DBMS. More specifically, data consistency is trickier due to the multiple

[1] https://www.couchbase.com
[2] https://aws.amazon.com/rds/aurora
[3] https://www.arangodb.com
[4] https://couchdb.apache.org
[5] https://www.postgresql.org
[6] https://redis.io

copies of data and distributed commits. Riak [7] tackles this by allowing conflicting copies of data to exist at the same time while guaranteeing eventual consistency.

Moreover, DBMSs should also take into account the failure of links and devices. Apache Cassandra [8] features no single point of failure. A *Write Ahead Log* [9] [10] is also used to keep logs of transactions in case of device failure.

In order to optimize the use of multiple edge devices and reduce bottlenecks, i.e., overloading of certain devices, load balancing algorithms must be a part of the DBMS. This can be achieved by dynamically altering the placement of data to devices in order to off-load them. Slicer [4] is a service that partitions data using keys while monitoring the load of each key and making rebalancing moves. Accordion [2] keeps a load balanced state through scaling (adding or removing devices) and predicts bottlenecks based on transaction affinity. Other works [5], [6] achieve fine-grained partitioning of data by detecting and carefully placing the "hot" tuples.

The nature of data RAINBOW deals with is related to streaming, while multiple data are produced per second. Their fast analysis and real-time response are crucial in Edge Computing scenarios. Saving and accessing data through the disk would cause a large non-acceptable overhead, thus the use of in-memory databases for analysis and distribution is the way forward. VoltDB [11] is a main memory DBMS based on H-Store[12], providing elastic scalability, rapid failover and consistent low latency, while performing single thread distributed transactions. Redis [6] is an open-source, fast main-memory data structure store. Redis can eliminate delays in data retrieval achieving very fast response times with read and write operations taking less than a millisecond, while it also provides high availability, scalability, fast fault recovery, built-in replication and on-disk persistence. Hazelcast IMDG [13] is an open-source in-memory data grid. The main advantage of using data grids is speed, especially when dealing with vast streaming data. Data are evenly distributed to the cluster nodes providing horizontal scaling. Apache Ignite [10] is an open-source, distributed store designed to work with big data and clusters of nodes. The form in which data is being stored is in key-value pairs which can be replicated or partitioned across the nodes of the cluster, achieving scalability and fault-tolerance. Ignite also supports co-located processing enabling the analysis of data on nodes and achieving lower data transferring across the network making it suitable for

[7] https://riak.com
[8] https://cassandra.apache.org
[9] https://hbase.apache.org
[10] https://ignite.apache.org
[11] https://www.voltdb.com
[12] https://hstore.cs.brown.edu/
[13] https://hazelcast.com/products/imdg

data-intensive or compute-intensive analytics like RAINBOW use cases. However, as explained in the next section, the built-in replication and partitioning mechanisms need to be by-passed in the context of WP4, since Ignite, like all the other afore-mentioned systems, has been designed for data-center rather than fog environments.

Finally, a distributed DBMS needs to take additional security measures due to the extensive number of users and devices that access the data. The security should be considered both in the communication, that is at the exchanging of data as well as in data by means of authentication and encryption. The first part can be achieved through SSL and TLS protocols and the use of VPN while the second through digital certificates. What DBMSs can provide to security is the encryption of data upon storing them using keys [14] as well as authentication and access control mechanisms [15] [16]. In RAINBOW data storage solutions, we capitalize on the work in WP2 so that all physical geo-distributed instances of a single logical database operate in a trusted manner.

The RAINBOW Approach. As already mentioned, RAINBOW leverages an established data center-oriented solution, such as Ignite, and develops lightweight components to by-pass the built-in replication and partitioning components, so that the final data storage solution is suitable for heterogeneous distributed fog settings. Any access to the data and nodes is fully protected due to the provisions in WP2.

2.2 Geo-Distributed Data Placement

An important aspect when trying to optimize analytics in a geo-distributed environment, like the RAINBOW project's use cases, is the placement of data. A query's latency can be highly affected by the location of its input data and their replication mainly due to low bandwidths or even privacy and security reasons. Thus, it is crucial to carefully consider where to place input data and to decide whether to move them or not, through replication, in order to overcome any overheads that may occur during the execution of the query.

To deal with these challenges RAINBOW incorporates a query-driven data placement technique to minimize data transferring. More specifically, the location of input data and their replicas as well as the placement of the analytical tasks, which essentially are the bolts of a Storm topology, are carefully considered when deciding the placement of the source vertices; the spouts in a Storm terminology. Moreover, data freshness reduction

[14] https://aws.amazon.com/dynamodb
[15] https://www.mongodb.com
[16] https://fauna.com

resulting from replicated data and data quality degradation in exchange for lower latency, is also considered.

Many works have dealt with the data-placement problem but none of them achieves the novelty of RAINBOW mentioned above. One of the most prominent solutions is Iridium [7], which optimizes the query response time by moving input data away from the bottleneck nodes. The size of the replicated data is calculated by finding the amount of data that will incur the minimum data transfer overhead. The problem is cast as an ILP optimization but, no freshness and quality criteria are considered. In addition, the RAINBOW framework deals with streaming data, while the solution in [7] targets batch queries.

Yugong [8] is a system that manages data and job placement in geo-distributed data centers and aims to minimize bandwidth usage. Data placement is altered through migration and replication of data during runtime. More specifically, when there is a change in the workloads or the bandwidth usage exceeds a threshold, a data migration plan is decided using an optimization solver. Moreover, the replication of data is updated at specific intervals using a greedy algorithm. As in Iridium, no freshness and quality criteria are considered. Samya [9] is another advanced geo-distributed data management system, but its focus is on supporting transactions rather than performing data placement.

Additionally, AdaptDB [10] partitions datasets across a cluster and refines these partitions as distributed join operations are executed. Partitioning trees are utilized, and the goal is to migrate data blocks between these trees to ensure load balancing. Also, Sword [11] introduces a workload-aware data placement and replication solution that minimizes the query makespan. In both of these works, no geo-distribution and resource heterogeneity aspects are considered.

Finally, in all of the above cases, the stability of the nodes, which is important in a fog ecosystem like RAINBOW, is not considered.

The RAINBOW Approach. RAINBOW addresses the problem of data placement in distributed databases in fog environments through replicating data closer to computation, while considering the impact on data freshness and quality that any data placement decision may have. This approach complements the common rationale to move computations closer to data and helps in meeting real-time latency constraints. RAINBOW proposes two novel techniques to decide when and where to replicate local data from a storage instance to a remote one to cope with the fog ecosystem's challenges. The first technique takes into account the stability of the fog cluster's nodes and tries to

replicate data from unstable instances to stable ones, in order to eliminate data loss due to node failures/disconnections. The second technique is query-driven; based on the periodic queries and the placement of the processing vertices, a multi-objective optimization problem is solved, to place data on nodes and reduce latency taking into account freshness and data quality degradation issues.

2.3 Scheduling Streaming Analytic Jobs in the Fog Continuum

Scheduling analytic jobs is an optimization problem with the indent to derive a clear mapping between the tasks of the analytics job and the workers of the underlying analytics engine so that resources and QoS requirements are meet before and during execution [2]. Recently, a paradigm shift is being observed, where data harvested from IoT devices is not "shipped" to central cloud datacenters for processing due to the requirements for milli-second responses that mission critical applications require, including the use-cases of the RAINBOW project (human-robot collision avoidance, car navigation assistance and autonomous drone swarm coordination) [3]. Hence, it makes no sense to place IoT services at the network extremes, while leaving the data processing to the cloud datacenter.

As such, RAINBOW embraces -in place- data processing where analytic jobs are scheduled and executed in the fog continuum shared among the collaborating fog nodes allocated to the IoT service to reduce any potential overheads of disseminating data back-and-forth to the cloud. In this sense, fog data processing presents similarities with geo-distributed data processing but moves away from batch processing over reliable networks where network delays are negligible [4]–[6]. Specifically, fog data processing deals almost exclusively with streaming data, while the fog nodes present a high degree of resource and network heterogeneity, can be mobile and extremely ephemeral, with all these contradicting the operating requirements of distributed data processing frameworks that are optimized for homogeneous machine clusters found in the cloud [7].

As the SOTA landscape in geo-distributed processing is large, diverse and with various aspects already covered in D4.1, this section puts particular focus on schedulers designed for optimizing distributed streaming processing over Apache Storm and how the task scheduling grasps on various requirements for edge and fog computing. At this point we note that the default Storm Scheduler adopts a pseudo-random round-robin task placement strategy to the worker nodes without even exploiting data locality [8]. This default scheduling algorithm is simplistic and not optimal in terms of throughput. Specifically, not acknowledging resource heterogeneity of the worker nodes results in some of the nodes being extremely strained and over-utilized while other nodes with

resource availability not being efficiently utilized. Hence, the scheduling ends up being problematic and far from optimal.

To tackle resource heterogeneity, one of the prominent requirements for edge computing, the R-Storm Scheduler [9] employs a resource-aware optimization for Storm jobs by solving a quadratic multi-dimensional Knapsack Problem in an attempt to optimize job throughput through task placement when acknowledging the heterogeneity of worker nodes in terms of compute and memory resources. On the other hand, the T3-Scheduler for Storm [10] puts focus on placing the job's tasks that communicate with each other on nodes that are closer in terms of network distance. In turn, the T-Storm Scheduler [11] supports the application of query operators over streaming settings by considering the inter-node and inter-process traffic to assign workload to the nodes, rather than the default approach adopted by the Storm engine. Also, T-Storm does not require the use of all worker nodes on the cluster and some may end up not being used at all. Similarly, the TS-Storm Scheduler [12] attempts to solve the inter-node imbalance problem by adopting a constraint-based optimization algorithm to dynamically eliminate the performance bottleneck of the topology.

While all aforementioned techniques constitute interesting approaches, they present key limitations. In particular, both resource and network capacity are assumed to be fixed and thus, agnostic to the actual workload. This introduces an obvious downside, as the workload in an edge/fog realm does not remain unchanged during the whole lifecycle of the streaming IoT application and therefore, performing the scheduling only once during the application deployment is unrealistic. A scheduler that performs runtime optimization is D-Storm [13], which employs a greedy algorithm in the form of a variant bin packing process that is periodically executed to acknowledge the dynamicity of the changes in the underlying environment and minimize inter-node communication latency. In turn, another key limitation for all techniques is that after ensuring resource availability they only approach the problem in terms of a single objective, which is either latency or throughput. None of the aforementioned techniques embraces multi-objective optimization.

In an edge/fog realm, other than latency and throughput, which are undoubtedly key performance indicators, other objectives can affect the efficacy of a deployment. Specifically, data quality can play an important role, where malfunctioning nodes, missing data, corrupted data and network uncertainty can lead to less useful results and in the end, affect performance as well [14]. In turn, energy consumption can critically affect the liveness of the underlying processing infrastructure [16]. Specifically, if nodes are battery-powered, then scheduling tasks on these nodes may be beneficial for performance at the current point in time but may not be available when actually needed.

Both data quality and energy-consumption are objectives where trade-offs with performance guarantees can be explored.

The RAINBOW Approach. The RAINBOW Analytics stack offers its users with a high level of flexibility in the analytics job scheduling process. Specifically, upon defining analytic queries and packaging them in continuous jobs, users are free to express the optimization strategy and in turn, the underlying execution engine is not bound to a specific algorithm implementation. Towards this, users specify what should be optimized, leaving the how to the novel RAINBOW scheduling algorithms for streaming analytic jobs. To date, users may leave the RAINBOW-tailored baseline Storm scheduler that adopts a fair task allocation strategy, opt to optimize the deployment in terms of performance, or explore trade-offs between performance and data quality, or performance and energy consumption, to ensure that computations return not just timely results but also reliable results while the scheduling is also energy-aware. All scheduling strategies have been designed and implemented for Apache Storm and can be used with any vanilla Storm deployment. However, these have also been enhanced and elevated for RAINBOW-aware Storm deployments where configurations are automatically received and processed through RAINBOW Service Graphs and data can be streamed and extracted through a RAINBOW Spout that efficiently accesses data through the RAINBOW Storage Fabric.

2.4 Analytics Query Expressiveness and Interoperability

The embedding of a distributed data processing engine to the deployment of an IoT service implies advanced knowledge of a particular programming model for the underlying processing engine. This has the unfortunate requirement of writing multiple lines of code just to submit a single query that results in a steep learning curve and limits the ability of IoT platform operators to quickly submit exploratory and ad-hoc queries not envisioned beforehand in the system design phase [16].

For example, it is the job of IoT platform operators to discover interesting insights from data assets, but not working out how to synchronize distinct dataflows to execute iterative ML tasks. The importance of this is highlighted in a recent Seattle report, where the ACM Fellows identify the design of declarative programming models decoupling the definition of data-oriented tasks from the engineering of the underlying infrastructure as a prominent inhibitor for advancing Data Science [17]. In turn, while the landscape of analytics frameworks is still fairly open and non-dominant, the lack of interoperability is a considerable hindrance for users. Specifically, switching from one platform to another requires the re-coding of complex analytic jobs [18]. This means that any coded analytics jobs would have to be scratched and re-introduced if the organization is to migrate from one framework to another resulting in what is now dubbed as the analytics stack lock-in

[19]. Therefore, the design of query abstractions that are decoupled from underlying processing engines and which can express explicitly the modeling, compilation and optimization needs of processing analytic insights on the edge, is of significant interest.

To alleviate the implication of advanced knowledge of a programming model, operator abstractions have already been introduced for the distributed data processing frameworks, such as for Apache Storm and Spark. For instance, Trident [20] is a framework, that introduces an abstraction layer on top of Storm. This provides users with high-level operators (i.e., aggregations, filters, joins) applicable to ingested data streams with the intent to minimize the programming effort in designing DAG topologies. Similarly, the Spark ecosystem includes two packages with high-level query abstractions that sit on top of the Spark engine, namely SparkSQL [21] and Structured Streaming [22]. The former provides a set of SQL-like operators on top of the Spark programming model, while the latter enriches SparkSQL with streaming capabilities. Even if these approaches are in the right direction, they are bounded to the underlying engine and focus only on the analytics queries without providing edge- and fog-oriented operators and optimizations.

Domain-Specific Languages (DSLs) offer pre-defined abstractions to represent concepts from an application domain and DSL compilers are rather optimized for this specific domain. Towards this, Summingbird [23] is a framework providing a domain-specific language and implementation in Scala attacking the analytics lock-in problem. Users write Summingbird queries for MR jobs that can then be transparently compiled to run on either or both, Hadoop and Storm. In turn, Beam (former Google Dataflow) [24] is an open-source framework that simplifies the mechanics behind creating interoperable data processing pipelines. Once such pipeline is defined, the user can then select to execute it on one of the supported backends (i.e., Hadoop, Spark) with these deployed on a local environment (i.e., laptop) or in the cloud.

However, the unique characteristics of IoT processing and Edge Computing demand new operators to express different constraints and optimizations such as sampling, upper error-bounds, bounded resources, placement awareness, among others [16]. Recently, a handful of frameworks have been proposed to derive analytic insights for edge computing and network telemetry. For example, Edgent (formerly known as Quarks) [25] is a framework providing micro-kernel run-time with small footprint that are particularly tailored to running on IoT gateways, network routers, and edge devices. Edgent provides users with the ability to denote when certain query operators should be applied on the edge and when data should be moved to the cloud backend for more complex processing. Similar frameworks have also been proposed from the research community as well, including CalculIoT [26] and λ-flow [3]. Tailored to edge network telemetry analytics,

Sonata [27] is a framework that offers scalable streaming processing. The framework provides a declarative pipeline-interface that allows network operators to express their analytic queries. Under the hood, Sonata uses the programmable data-plane of network switches for query preprocessing and Spark for query execution. However, Sonata is developed solely for packet-level network telemetry analytics without any acknowledgment for other types of data. In contrast to the aforementioned, StreamSight [28] (initially developed by UCY) is a framework for edge-enabled IoT services which provides rich and declarative query abstractions for expressing complex analytics over data streams and compiling these queries into streaming processing jobs for distributed processing engines. StreamSight offers several query operators to derive high-level analytic insights, along with execution optimizations and constraints tailored for edge computing to achieve latency, robustness, and approximations in query execution. Among the optimizations made available include data approximation techniques (i.e., sampling, filtering), the reuse of intermediate query results through query compiler optimizations and the sharing of query results among multiple jobs. All these reduce both the computational and network pressure in edge deployments.

The RAINBOW Approach. The RAINBOW Analytics stack adopts and extends StreamSight to provide users with a declarative language to define interoperable streaming analytic jobs that can also be optimized for fog deployments. The StreamSight query model considers that input is a monitoring metric stream and output is again the monitoring stream but altered after the application of various operators, including filters, transformations, aggregations, and groupings. This query model is extended to support multivariate metric streams so that metrics logically grouped together (i.e., collected through the same monitoring probe) can be accessed from a unified stream for queries making use of more than one metric. The StreamSight query model is realized and implemented for Spark-Streaming with each implementation denoted as an “analytics query compiler”. For the purposes of the RAINBOW project, two additional compilers have been created and are actively supported. Specifically, a compiler for the vanilla version of Apache Storm has been created, while another compiler has been designed for RAINBOW-enabled service graphs and analytic job optimizations that only RAINBOW, to date, supports. Through a StreamSight declarative query, any annotated analytic job optimizations are used to provide the relevant input required for the analytics job scheduling and the provisioning of the desired RAINBOW-enabled Scheduler.

3 Distributed Data Storage and Sharing Service

In this Section, we present a comprehensive documentation report referring to the final release of the Distributed Data Storage and Sharing Service.

3.1 Overview

3.1.1 The RAINBOW Data Storage and Sharing component

The Distributed Data Storage and Sharing service is responsible for meeting the RAINBOW ecosystem's needs regarding resilient data storage in a dynamic fog environment. To this end, the service provides a distributed solution comprising instances communicating with each other in a peer-to-peer fashion departing from the leader-worker paradigm; still, the system behaves -from the perspective of its users- as a single coherent cluster and a single logical database. The various micro-services that are implemented, make the data exchange with external components and users easy to execute with full ACID compliance during database operations.

The purpose of the service is not to create a new distributed database management system, but to use the most suitable one and implement new components and micro-services that satisfy the requirements of the fog ecosystem. As such, the Data Storage and Sharing service builds upon the open-source distributed DBMS Apache Ignite by implementing novel data placement methods as well as creating a logical *Storage Fabric*, providing a transparent and lightweight service for resilient storage.

Figure 1 presents a high-level approach of the Data Storage service. The service comprises a distributed Ignite cluster with each instance (*Storage Agent*) deployed in each fog node in the context of the overlay mesh network of the RAINBOW cluster. The purpose of each *Storage Agent* is to persistently store the metadata extracted from the *Monitoring Agent* deployed on the same fog node, through ACID transactions, and provide the means for the rest of the RAINBOW services to extract them.

Storage Agents communicate internally, through Ignite's protocols and implemented micro-services, creating the *Storage Fabric*, a logical entity that supports one-off queries and data extraction through a decentralized API. Each *Agent* provides the way to extract either locally stored data only or data from every *Storage Agent* transparently to the requesting services.

Furthermore, the Data Storage and Sharing service provides an automated way to replicate and partition data from one *Agent* to another by implementing two techniques,

with each one being driven by different problems in fog environments. The first technique allows the service to cope with the dynamic nature of the fog ecosystem where node failures are a usual phenomenon. A single *Storage Agent* (acting as the *cluster head*) continuously monitors the health of the cluster by storing the restarts and the failures of each instance. This allows the service to replicate data from unstable nodes to more stable ones in order to avoid data loss/unavailability issues due to fog node failures.

The second technique is query-driven and cooperates with RAINBOW's Distributed Data Processing service in order to cut down the network congestion through minimizing the latency when data movement from a *Storage Agent* to an *Analytic Worker* is required. By monitoring the latency between the fog nodes and the location of the *Analytic Workers*, the technique solves an optimization problem, considering both the freshness and the quality of the data. Moving monitoring metrics to other nodes may improve the latency at the expense of staleness/freshness, since replicated queried metrics are available after some time period from their generation. Instead of replicating data, the performance in terms of latency may be improved through performing data sampling; this again comes with a tradeoff, since it results in lower data quality, if less data are transmitted when down-sampling is applied.

Finally, the Data Storage and Sharing service is also used by the rest of the RAINBOW's services to temporarily or persistently store timestamped data, i.e. the results of the queries running on the Distributed Data Processing service.

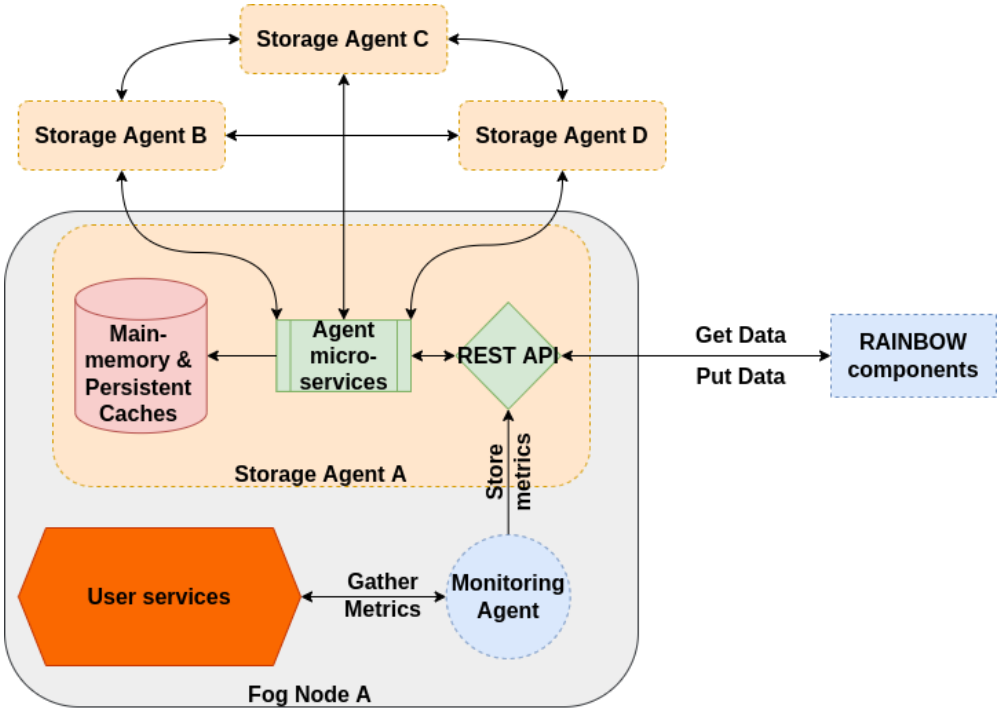


Figure 1 High-level overview of the Distributed Data Storage and Sharing service's architecture and communication with RAINBOW components

3.1.2 Distributed main-memory database management system - Apache Ignite

The basis of the Distributed Data Storage and Sharing service is the database management system itself. Apache Ignite was chosen as the distributed database that can be used as a main-memory database and supports different persistent storage options. It also provides different ways to store and access data.

The distributed protocol of Ignite adheres to the serverless paradigm instead of the leader/worker one that most DBMSs use. This allows for a more dynamic flow of storage instances where a new instance can enter the cluster and an old one leave it seamlessly, without the permission of another one. Each new node that tries to connect to the Ignite cluster needs to know at least one address from an instance that is already part of it. After the insertion, the new node can communicate and has access to the rest of the cluster nodes. This tackles network congestion problems that arise from the leader/worker paradigm, where all nodes, trying to connect to the cluster, need to communicate with the leader instance exclusively. The number of nodes in a cluster can go up to thousands preserving linear performance.

Ignite provides three different types of instances, namely *Server*, *Client* and *Thin Client*. Server instances are mainly responsible for storing data. Client instances can access stored data and either perform computations or query the data. Thin clients are smaller

versions of the Client type that attach to a specific node for minor computations and querying.

The data structure that is used to store data is called cache. It is a key-value structure where both the key and the value can be any type of object, i.e., Java class. It supports three modes of caches, each one serving different needs. The first one is the LOCAL mode in which the data are stored locally and can only be accessed by the specific Server instance that stored them or by a Thin Client attached to that Server. Each time a LOCAL cache is initialized, it is created in every node of the cluster, regardless of whether it will be used. The second mode is the PARTITIONED cache which partitions the data to the different server nodes in a balanced manner. A backup policy is available for the PARTITIONED mode in order to tackle loss of partitions problems. The final mode is the REPLICATED one which replicates every data point to every other node in the cluster. Furthermore, the data from each cache can be accessed either through the key or an SQL-like interface.

Finally, each instance can run custom-made micro-services depending on the needs of the user. The deployment can be achieved either by using custom instance filters or by providing deployed instances for each service. An instance can access remote instances on different (fog) nodes.

Apache Ignite is the basis for the Distributed Data Storage and Sharing service due to the above features. First, it allows the user to create local caches without the use of the default partitioning/replication schemes. Secondly, it guarantees consistency and fully supports ACID transactions while providing main-memory I/O, which is an important aspect for RAINBOW. Furthermore, the serverless paradigm enables quick and lightweight movement of storage instances in and out of the cluster. Finally, the custom-made micro-services that Ignite offers to the user can support the decentralized API and the novel data placement methods, replacing the default partitioning/replication schemes, that are integral to the service.

3.1.3 Leveraging Apache Ignite for RAINBOW

The RAINBOW Data Storage and Sharing service implements Ignite's *Server* instance type with two flavours and deploys them on the different nodes based on the desired functionality. The first one is the common Server instance that is used for data extraction and ingestion and is deployed on every fog node with the RAINBOW mesh stack. The second flavour is an extension of the first one, meaning that it can also store and extract data with the addition of monitoring the cluster's health along with taking data

movement decisions based on the data placement techniques. It is deployed on the cluster head, usually a resource-rich fog node or a cloud node.

The cluster head instance is the first one deployed and initializes the cache creation and the micro-services that will be running on the cluster. The rest of the instances start running afterwards and are immediately connected to the cluster through the address of the head.

Each *Storage Agent* is responsible for storing the local data that are produced through the fog node's *Monitoring Agent*. The instance contains two local caches in cooperation with a replicated cache for the storage. All of them are key-value caches with SQL-like properties, such as indices on different fields. The first one is a purely in-memory cache and stores only the latest values from the monitoring metrics. This cache is used for quick I/O of the metrics from the different RAINBOW services that require them, i.e., the Distributed Data Processing service. The second cache is used to persistently store the historical values of the monitoring metrics up to a user-specified time range. The time range is used to limit the volume of data and the memory resources needed for the storage. The third cache is available to persistently store metadata on each monitoring metric that is ingested through the *Monitoring Agents*. This cache is replicated to every *Storage Agent* with each one of them having direct access to the metadata of all the stored metrics. Figure 2 presents the schemas for each cache with the keys being above the line and the values below.

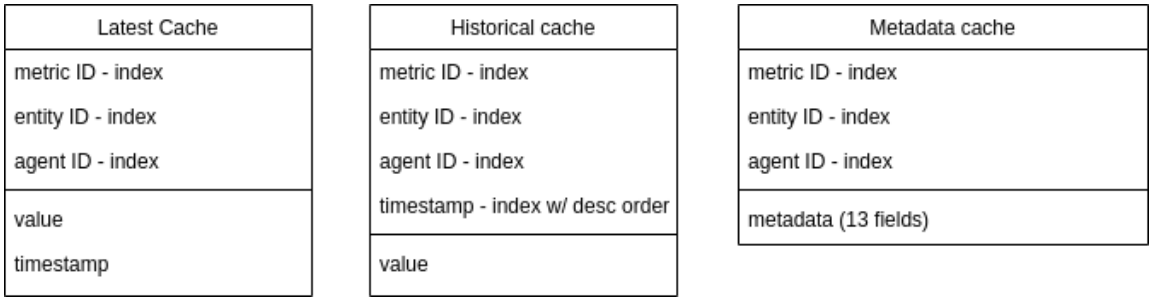


Figure 2 Monitoring schemas with indices

In addition to the aforementioned caches that are exclusively tailored to monitoring metrics, there are four others, with each one having a different functionality. In order to store analytic results and temporary data from the rest of the RAINBOW services, two caches for timestamped data are used. Both are similar with each other, with the only difference being the persistency. One of them stores data purely in-memory while the other one uses the disk for persistent storage with eviction options to restrict disk space usage.

The final two caches are used by the cluster head to store the restarts and failures of each *Storage Agent* and the data points that have been replicated alongside their destinations. Both caches persistently store data to the disk and are used by the data placement micro-service to make decisions on possible future data placement sources and destinations.

Each *Storage Agent* runs three micro-services that help with data ingestion and extraction through a REST API with private endpoints available only to the rest of the RAINBOW services. The cluster head contains an additional micro-service that monitors the cluster's health and decides the data placement options based on the two novel techniques described in the next section.

3.1.4 Data placement algorithms

In order to cope with the dynamic nature of the fog ecosystem as well as minimize data movement through the local network, two novel data placement techniques have been introduced to the Distributed Data Storage and Sharing service. The first one is based on the stability of the fog nodes in the RAINBOW cluster and replicates data points from unstable nodes to more stable ones. The second one, cooperates with the Distributed Data Processing service to minimize the data movement from one fog node to another when the queries require it.

The result of both techniques is to replicate the monitoring metrics stored on one *Storage Agent* to at least one other. After the decision for the placement has been made, any new ingested values for monitoring metrics, that belong to the set of data points that are replicated, are also ingested by the replica destination node. The techniques are used in the service instead of the replication and partitioning methods that the DBMS itself uses, which is based on balancing the data on the set of cluster nodes.

3.1.4.1 Stability-based placement

The rationale of this technique is to replicate the monitoring metrics of the *Storage Agents* on fog nodes that are unstable to the more stable ones of the cluster. This tackles the main problem of the fog environments, which is the increased node failure rate compared to cloud environments. The replication prevents data loss, when a failure occurs since the data can be retrieved from the remote nodes where the replicas are stored. This process is part of the Data Storage and Sharing service and uses the distributed DMBS's internal communication protocol, which is faster and more secure than using external APIs and services. The cluster head instance monitor the health of the cluster in the background, storing both restarts and failures for each fog node, while it periodically runs the

placement algorithm that takes the decision whether a node's data need to be replicated or not.

Ignite creates many types of events, from cache-related, such as writing/reading data, to node-type ones; the latter refer to node restarts, failures and insertions in the cluster. A single listener daemon is created in the cluster head that monitors the cluster's health and each time a node-type event is reported, it is immediately being registered and stored in the respective cache, called *status*, available to every *Storage Agent* in the cluster. This cache stores, for each node *X* in the cluster, the value that contains the timestamp (*X.start*) that the instance was first introduced in the cluster along with two lists, *X.fails* and *X.restarts*, containing each timestamp that the node failed or restarted respectively.

The algorithm runs in specific time intervals and finds the most unstable nodes in order to replicate their data to the most stable ones. The first step of the process is to find the unstable nodes of the storage cluster. It computes a cost function *U(x)* for each stored node *X* in the *status* cache. The cost function is based on the number of node's failures along with the time period between them and is computed using the following equation:

$$U(x) = \sum_{i=1}^{X.fails.size} \left(a/(X.fails[i].timestamp - X.fails[i-1].timestamp)\right) + a/(X.fails[0].timestamp - X.start)$$

The *a* variable is a dynamic variable that is based on the age of the failure and the number of failures for specific time periods, having an increased value for the most recent ones. At the beginning of the process, we split the failures into three distinct time periods. The first one (*old)* contains the failures that are older than a week, the second one (*mid*) refers to the failures older than 2 days but newer than a week and the final one (*new*) contains the failures in the last 2 days. Considering the three time periods, variable *a* is computed based on the following equation:

$$a = \begin{cases} 0.5 * fails.filter(old)..timestamp \in old \\ fails.filter(mid)..timestamp \in mid \\ 2 * fails.filter(new)..timestamp \in new \end{cases}$$

The process assigns an increased score to the nodes with recent failures compared to the ones with many old ones. This helps to identify the nodes that might be more prone to failures, since they may have problems with resource congestion and power failures that could need actions to be resolved. On the other hand, the nodes with old failures are probably less prone to new ones, since these problems are most probably fixed since

then. Moreover, nodes that have recently been chosen as unstable for replication are removed from the list as well as nodes that have reached a maximum capacity for outgoing replicas. Since the continuous increase of the cost for nodes can cause problems for newly created ones that experience frequent failure, i.e., the older node will have a bigger score than the newer one, the maximum capacity stops the older nodes from always be on the top of the list. At the end, the function returns the *N* nodes with the highest scores based on the cost function.

The second part of the algorithm uses a similar process to rank the nodes based on their stability by using a different cost function. The following equation presents the cost function *S(x)* that computes the stability score for each node *X*:

$$S(x) = -(X.restarts.size + 3 * X.fails.size) * (currentTime - X.start)$$

This cost function uses a combination of the failures and the restart times (which contain scheduled restarts) and returns its product with the time that it is alive. A node with no failures and restarts gets a score of 0, which is the highest expected score. Moreover, to further build upon the stability of a node, the process ignores every instance that has failed more than once or that have been unstable recently. The function, similarly, to the previous one, returns the *N* nodes with the highest stability score.

The final part of algorithm decides, for each unstable node, the destination of its data from the stable pool. The process is straightforward and for each unstable node, it chooses the first stable one and removes it from the pool. This helps to reduce the workload from a single stable node if it was chosen as the destination for more than one unstable nodes. An exception in this function occurs when a node from the top 50% of the stable pool has a subset of the unstable node's data. In this case, the rest of the data from the unstable node are also replicated to complete a full replica.

3.1.4.2 Query-based placement

RAINBOW ecosystem consists of multiple fog computing nodes, denoted as CN. Each pair of nodes $(u, v), u, v \in CN$, has a specific communication latency, $lat(u, v)$. These nodes also act as sources, by hosting the *Monitoring* and *Storage Agents* to produce and store local monitoring metrics. In this network, users submit queries for analysis to the Distributed Data Processing service that are translated into Apache Storm topologies which consist of vertices i. The vertices comprise spouts S, which acquire data, and bolts, which execute the analysis part. The aim of the proposed policy is to decide from which node u, a spout i will receive data; this is essentially a decision on the placement of spouts. The placement of the bolts is decided using RAINBOW's Distributed Data Processing

query scheduling algorithms. We use the binary variable $x_{i,u} = 1$ to denote that spout $i \in S$ will receive data from node $u \in CN$.

Each spout can receive data either from a source node or from a node that holds or can hold replicated data. If a spout is assigned to the node that is also its input source, then the transfer time cost is zero. However, in some cases, this node may not be available to execute a spout or it may have been defined as unstable. In the latter case, the source node's data needs to be replicated to other nearby nodes. This data replication incurs an extra delay, due to the time it takes to transfer data from the source to the replica location. The time for a spout i to receive data from a replica location or a source node is denoted as $D(i,u), u \in CN_i$, where CN_i is the subset of fog nodes from where a spout i can receive data. Essentially, $D(i,u)$ quantifies the freshness degradation (i.e., staleness) of input data.

Without loss of generality (since we focus only on data placement in this algorithm), we assume that the actual analysis of data is either encapsulated in a single bolt or that all the bolts are assigned to a single fog node, u_b. This node is known beforehand and the time to transfer data from a spout to this node is considered in the cost model and is equal to $lat(u,u_b)$, where u is the node that sends data to the spout.
The overall latency of the query, denoted as L, is calculated using the following formula:

$$L = max\{\sum_{u \in CN_i} \left(lat(u,u_b) * x_{i,u}\right)\}, \forall i \in S$$

Moreover, the solution can opt to down-sample initial data to reduce the query latency. More specifically, a sampling technique can be performed on the input data, such as to minimize their transfer time through the network. The quality degradation, q_i of a spout i is a continuous variable in the range between 0.1 and 1 (the lower the q_i; the higher the quality reduction) that defines the sampling selectivity and affects the query's latency as follows:

$$L = max\{\sum_{u \in CN_i} \left(lat(u,u_b) * x_{i,u}\right) * q_i\}, \forall i \in S$$

In general, the optimization problem considers the overall latency, the data freshness degradation, and the data quality while keeping the quality above a given threshold Q. The objective is described by the following formula:

$$F = \frac{L * \left(\beta + max\{\sum_{u \in CN_i}\left(D(i,u) * x_{i,u}\right)\}\right)}{minq_i\}, \forall i \in S}$$

where β is a constant variable. The denominator q_i aims to penalize excessive quality reduction of a single spout. The formulation of the optimization problem is provided by:

$$minF(x,q)$$

$$s.t \sum_{u \in CN_i} x_{i,u} = 1, \forall i \in S$$

$$x_{i,u} \in \{0,1\}, \forall i \in S, u \in CN_i$$

$$avg_{i \in S}(q_i) \geq Q$$

$$q_i \in [0.1,1], \forall i \in S$$

The decision variables are both the $x_{i,u}$ and the q_i ones. This results in a NP-hard problem, which is broken down into two ILP subproblems. First, the optimal node from which each spout will receive data is decided, using a solver such as Gurobi [17]. Next, quality reduction is examined and using the same solver, the sampling ratio for each spout is decided.

3.2 Requirements Fulfillment

ID	FR.DSS.1		
Title	**Storage for real-time and historical monitoring data**		
Requirement Description	The Distributed Data Storage and Sharing service must provide the means to store both real-time and historical monitoring data using efficient data structures for fast read/write query operations.		
Validation	Completed	Status	Fulfilled
The Distributed Data Storage and Sharing service initializes and uses three distinct data structures (Ignite caches) to provide access to both latest and historical values for the monitoring metrics using its ingestion and extraction services. The values and the metrics' metadata (i.e., description) are stored in separate caches to cut down the storage size and decrease the process time when only the values need to be updated. Furthermore, the service uses both in-memory and a combination of in-memory and disk options for storage to reduce the I/O times. A pure in-memory cache is used to			

[17] https://www.gurobi.com

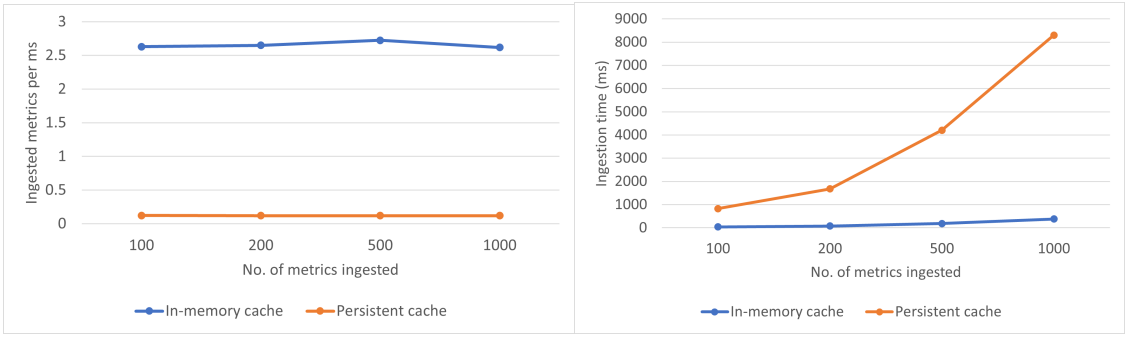
store the latest values for all monitoring metrics while a cache that fits a subset of the data in-memory and stores them persistently in the disk is used for the historical values.

During data ingestion, the incoming metrics are written firstly to the in-memory cache and afterwards to the combination cache. The runtime of the in-memory ingestion process is greatly decreased since it stores the data exclusively to the main-memory. Also, the second write process runtime is hidden in the background since the values are already available in-memory for a data extraction query.

Data extraction is completed using the respective cache when only the latest or a subset (or all) of the historical values are queried. Similarly, to the ingestion process, the output runtime of the in-memory cache is negligible since only the main-memory is involved. Even when the historical values of a metric are due for extraction, the historical cache starts by checking the data pages that are stored in-memory and afterwards tries to find the rest of the data from the disk pages. This cuts down the runtime, especially when all of the requested data are stored in the main-memory pages. Furthermore, when a query is periodically executed, the data pages holding the resulting data are transferred to the main-memory, replacing the pages that have not been active recently.

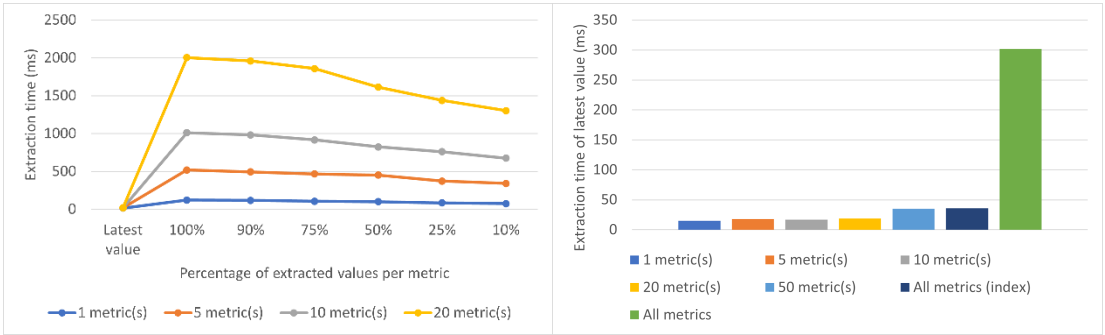
In order to test the effectiveness of the data ingestion and extraction, we have completed a set of experiments in a physical machine with 4GB RAM, a 2-core CPU and a 7200RPM HDD.

The left plot from the figure below, presents the average number of metrics that can be ingested per millisecond from the in-memory and the persistent cache, while the right plot presents the total ingestion time for both cases. The x axis represents the total number of different metrics that where ingested as a batch. As expected, the in-memory speed is approximately 4.5% of the disk speed being able to write approximately 2.6 values per ms. Furthermore, the speed of both caches is linear in comparison to the number of values ingested.



The second set of experiments present the extraction performance of the component. The left plot indicates the runtime it takes to get the API response during an extraction request for different metric sizes. The x axis presents the percentage of the requested values from the total (1000) stored values of each metric. The "Latest value" column indicates that only the in-memory cache for the latest value of each metric was used. Building upon the previous experiment, the in-memory cache speed is significantly increased in comparison to the persistent cache, while the performance is linear compared to the requesting percentage.

Finally, the right plot presents the in-memory's performance when requesting the latest value for different metric sizes. A point of importance in this plot is the two rightmost columns where the values for all metrics are requested. On the first case, the metrics are explicitly requested, while on the second case the storage agent needs to first find the set of local metrics and afterwards extract their values, thus the difference in the runtimes.



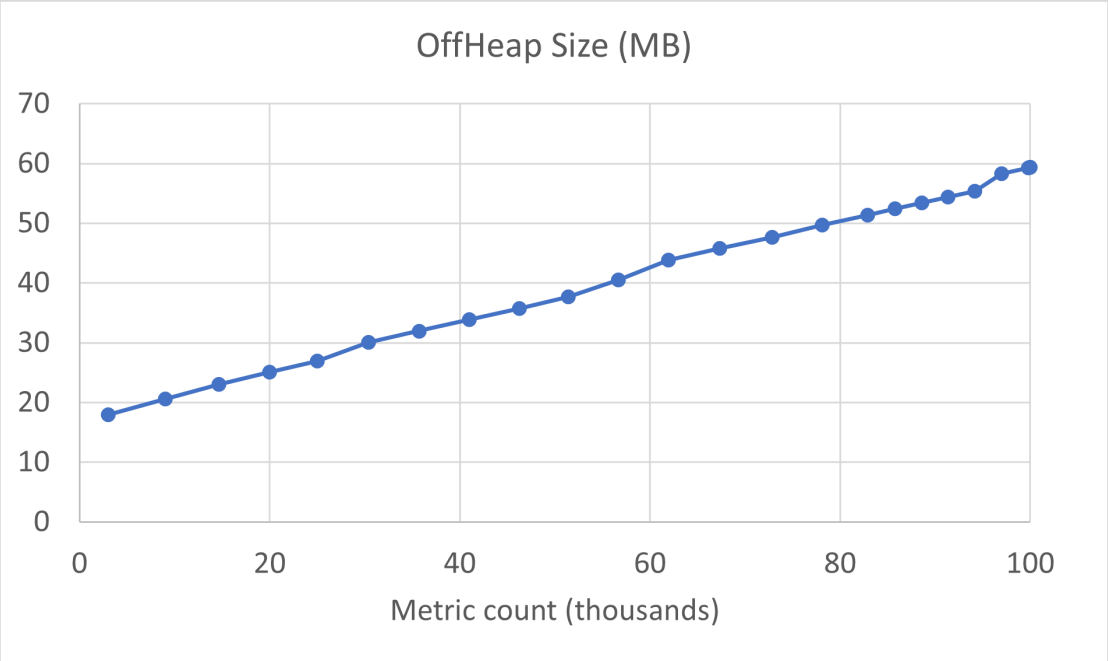
ID	FR.DSS.2
Title	**Historical data eviction based on user-desired eviction policies**

Requirement Description	The Data Storage and Sharing service must provide the means to control the time range of the historical data (data eviction period). This in turn limits the volume of the persistent data and the memory resources needed by the service so that even fog nodes with limited storage capabilities can be supported.		
Validation	Completed	Status	Fulfilled

The Distributed Data Storage and Sharing service is built on top of Apache Ignite which uses data regions to store ingested data. The service creates two distinct regions, with the first one using only the main-memory for storage, while the second one persistently stores data on disk concurrently with main-memory usage for faster I/O. The data regions are fully configurable from the data page size to the total storage size. In our solution the default storage size is 256MB for each data region. The size can be configured during the deployment of the service depending on the node resources.

Furthermore, a data eviction policy is always running in the background flushing older data pages from both disk and the main-memory. The default eviction period is 168 hours (1 week) where data points that have been ingested before the specified period are deleted from storage. The eviction period can be configured during the deployment of the service depending on the user preferences and the fog node's resources.

Combining both of the aforementioned options, the service limits the total size of the main-memory used for storage as well as the disk space usage where pages are flushed when the eviction policy deems it necessary. Both limiters are especially useful in a fog environment where resources are sparse and the service should not intrude with the RAINBOW users' applications.
The following plot presents the linear increase in OffHeap size (main-memory storage) when storing thousands of metrics along with their values. In this case, when up to 100 thousand different metrics with a single value are stored, the needs for memory go up to 60MB. This means that on the default parameterization (256MB of main-memory space), the component can store up to 427 thousand metrics. This can also be translated to either 427 thousand different single-value metrics or a combination of a smaller number of metrics with more values, e.g., 1000 metrics with 427 values for each.



ID	FR.DSS.3		
Title	**Efficient partitioning and replication algorithms**		
Requirement Description	The Data Storage and Sharing service must provide the means to efficiently utilize the underlying fog resources by partitioning and/or replicating stored data when needed.		
Validation	Completed	Status	Fulfilled

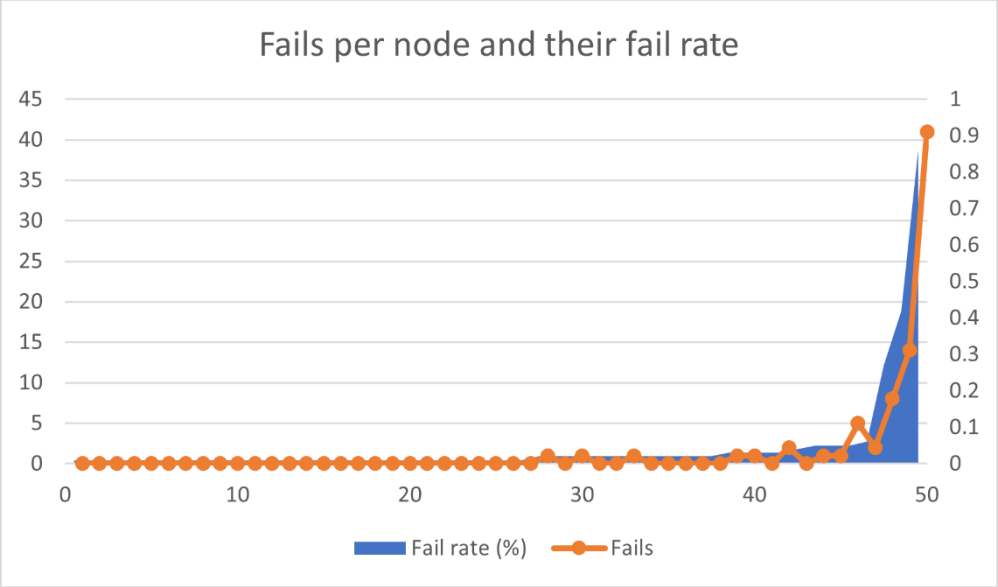
The Distributed Data Storage and Sharing service is built on top of Apache Ignite, an inherent distributed DBMS. Ignite's mechanisms try to balance the data volume on the set of homogeneous and well-connected cluster nodes. In our solution, we implemented two novel techniques that take into account the fog instability as well as the network limits during data movement.

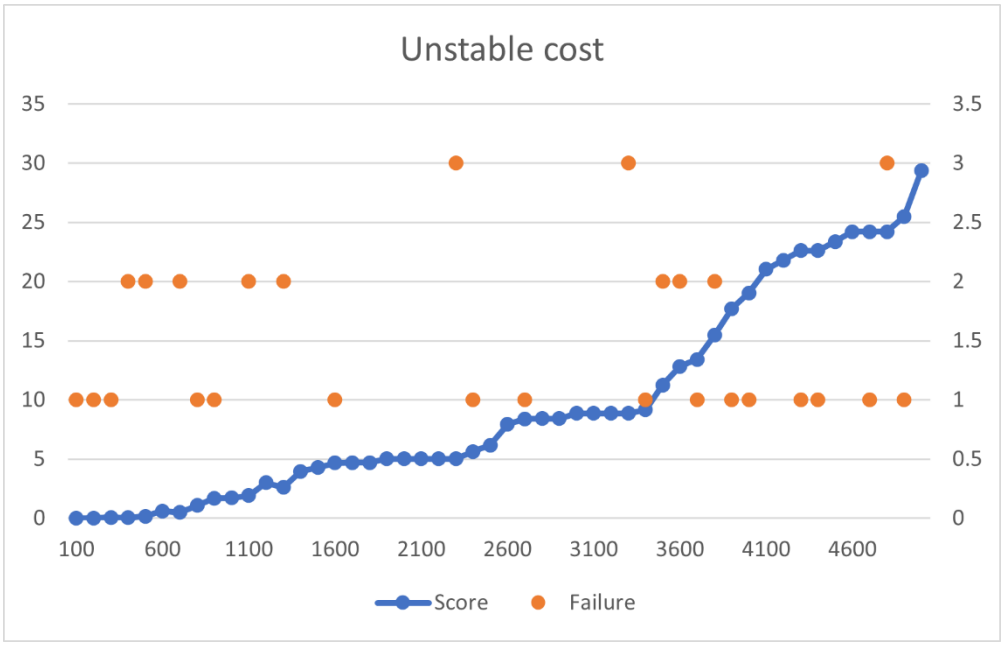
The first technique tackles the data availability problem due to unstable fog nodes by replicating their data to remote instances. The first set of experiments presents the instability of the nodes (first figure) and an unstable node's cost function progress during its lifetime (second figure).

The experiment is a simulation of 5000 iterations with 50 fog nodes. The fail rate follows a zipfian distribution with most of the nodes having approximately 0.01% of

failure and only 3 nodes having 0.27% up to 0.86% rate. As expected, in the first figure the nodes with higher failure rate, have an increased number of failures and only 5 nodes from the rest experience a single failure during the total set of iterations.

The second figure presents the progress of the cost function measuring the instability of the node with the highest failure rate (0.86%) during its lifetime. Based on the equation and the progress plot, the cost is increased each time a failure occurs on the node, while being close in time to the previous failure. The score is further increased when the node has been alive for a longer time period. Since this can cause problems for newly created nodes that experience frequent failure, i.e., the older node will have a bigger score than the newer one, a constant number for the maximum capacity of outgoing replicas for an unstable node is used.

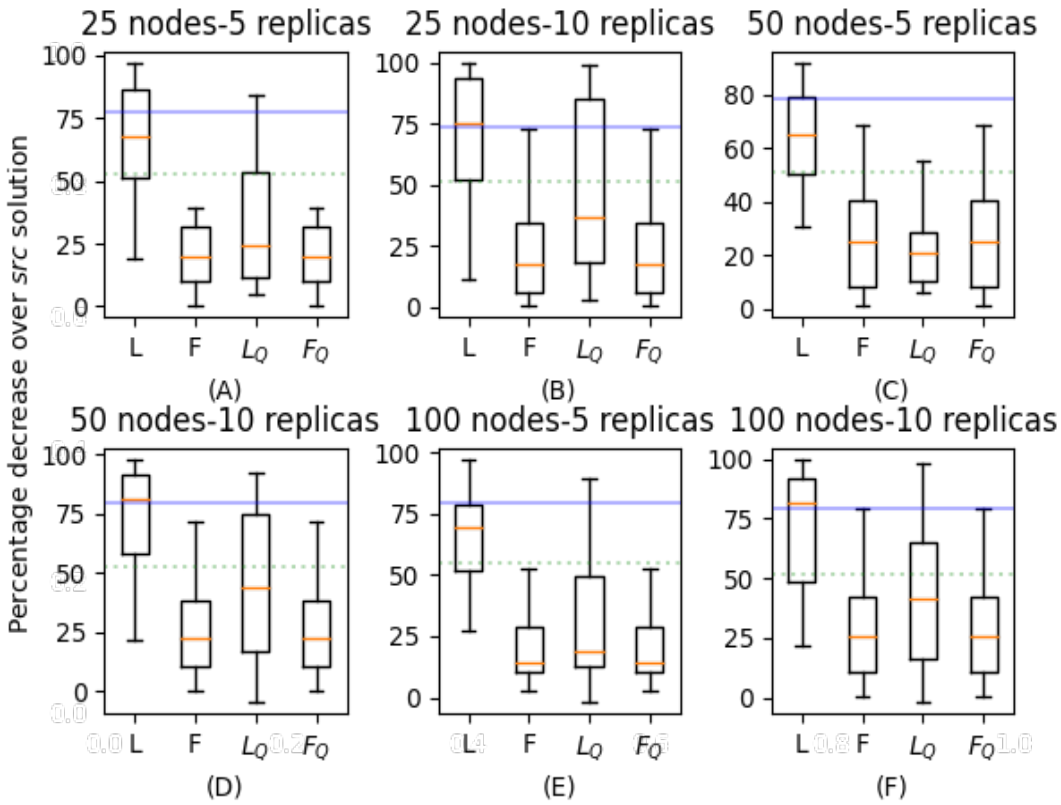




The second technique (section Query-based placement) considers the network links between the fog nodes as well as the data freshness and quality in order to decide the replication destination for a source node.

The second set of experiments aim to evaluate the query-based data placement technique. The comparison is against a baseline that assigns spouts only on the source nodes, using no replication; this baseline is denoted as *src*. Essentially, in the baseline, $D(i,u)$ is always zero.

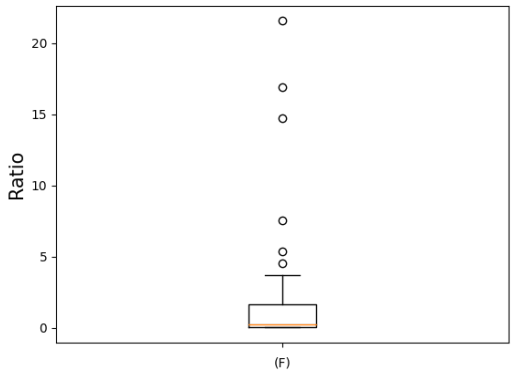
In the experiment, the number of spouts is set to 10 and we experiment with different number of fog nodes, specifically 25, 50 and 100. Each node can host up to 5 or 10 replicas. The communication cost between nodes is uniformly distributed in the range [0.1, 10] and we simulate bottlenecks or malfunctioning nodes through increasing their communication latencies by a factor produced by ipfian distribution. Finally, the β value is set to 1 and Q to 0.8.



The previous figure presents the experiment results for different numbers of nodes and maximum number of replicas per node. Each experiment was conducted 500 times and the runs, where the use of replicas is profitable, are examined. We show the percentage of reduction on L and F; 10% reduction is that query-based data placement strategy decreases the corresponding value of the *src* by 10%.

We examine the cases both when quality degradation is not allowed (denoted as simple L and Q in the figures) and when it is allowed (denoted asL_Q, F_Q). The dotted green and solid blue lines represent the average percentage of maximum quality degradation of RAINBOW's solution and *src*, respectively. From the plots, we can observe that when quality degradation is allowed, the improvements over *src* are lower, as expected. But still, they are significant and they can reach an order of magnitude for the latency (decreases of 90%). Regarding the times that replication was chosen, these were 4.4%, 12.2%, 4.6%, 9.2%, 4.6%, 13.6% for the A, B, C, D, E, F cases in the figure, respectively. Two additional observations are as follows: (i) the more the replicas allowed per node, the more the cases, where query-driven replica generation is decided; and (ii) the more the maximum number of allowed replicas and the more the nodes, the higher the decrease in latency and the F objective function.

Regarding data freshness, in following figure, we present the ratio of data freshness degradation delay and latency for the (F) case. In most of the cases, the ratio is below 1 which means the freshness degradation delay D, does not exceed the Latency L while the mean ratio is 1.63 and the median value is 0.25.



Finally, we examine a scenario with 50 nodes and 10 replicas, where no bottleneck nodes exist. That essentially means that we do not weight the communication latency of certain nodes using a zipfian distribution. We compare with case (D) from the first figure. The results presented in Table 2 show that the technique is capable of yielding lower but tangible improvements even in the most challenging (but not realistic) case, where the communication delays follow a uniform distribution.

Table 2 Average L and F percentage of reduction over src

	w/o bottlenecks	w/ bottlenecks (D)
F	15.06	25.0
L	45.77	73.42
% of optimized runs	5.0	9.2

ID	FR.DSS.4		
Title	**Secure data access**		
Requirement Description	The Data Storage and Sharing service must provide the means to access the storage. The access can either be used for writing or querying data. Additionally, the data should be available to the requesting service or component from any instance of the distributed data storage.		
Validation	Completed	Status	Fulfilled

RAINBOW's Secure Overlay Mesh, as its name says, implements an overlay networking protocol that dynamically auto-configures nodes, regarding the acquired IP addresses, when they join the fog cluster. The IPv6 of each fog node is the truncated SHA512 hash of the public key which is used for spoofing prevention and attestation when the node connects to the RAINBOW network.

The Distributed Data Storage and Sharing service is deployed along with the RAINBOW stack on each fog node. During the deployment, the IPv6 of the fog node, which is generated when the node enters RAINBOW's Secure Overlay Mesh, is used as the identifier of the *Storage Agent*. Each time the agent ingests new monitoring metrics from the *Monitoring Agent*, it appends its identifier to the key of each metric before storing them.

The service provides a REST API for data extraction and ingestion. The ingestion of data points is only allowed to the *Monitoring Agent* that is deployed in the same fog node along with the *Storage Agent* and to the rest of the storage instances in the cluster (when data are replicated).

Furthermore, the data extraction is allowed only to local services that are deployed in the same fog node as the *Storage Agent.* Services on remote nodes, e.g., *Analytic Workers*, can only access the stored data using the IPv6 of the *Storage Agents*. If the provided IPv6 addresses are not correct the storage instances will not return any results. The storage instances of the Ignite cluster can also access the data from remote nodes using the necessary IPv6 addresses which can be accomplished through the internal cluster networking protocol.

Finally, to further increase the security during data exchange and sharing, the Secure Overlay Mesh does not allow external connections to the services of the cluster nodes. This means that Ignite instances from outside physical machines cannot connect to the Distributed Data Storage and Sharing service even if it has one of the addresses of the cluster nodes.

ID	FR.DSS.5
Title	**In-memory cache for routing tables**
Requirement Description	The Data Storage and Sharing service must provide the means to temporarily cache the routing tables for the secure CJDNS overlay network protocol.

Validation	Completed	Status	Fulfilled
RAINBOW's Secure Overlay Mesh implements an overlay networking protocol that is built on top of encrypted P2P routing mechanisms. Initially, this mechanism relied on CJDNS; yet the final implementation relied on a CJDNS[18] fork called Yggdrasil[19]. In practice, Yggdrasil is a GOLANG port of CJDNS (developed in C++) that incorporates several routing optimizations. According to the encrypted P2P protocol, each node holds a local cache of the routing table that stores destinations that are either one hop destinations or multiple-hop vectors that are discovered using data source routing. Each path (single-node or multi-node) has a limited time validity since nodes can alter their positions dynamically. The set of 'volatile' vectors constitutes a data structure that acts as a cache for the Distributed Data Storage and Sharing service. This allows the provision of cluster-related metadata that can be consumed directly by the Distributed Data Processing service or/and any other RAINBOW component that needs it to derive statistics and analytic query results. The cache of the routing tables can be queried individually or as part of an analytics query. Thus, the data placement and all micro-services that are part of the Data Storage service also affect the local copy of the routing table's data. The performance evaluation is covered by the experiments in FR.DSS.1 and FR.DSS.3, as part of the service's implemented techniques.			

ID	FR.DSS.6		
Title	**Custom schema support for app-specific data storage**		
Requirement Description	The Data Storage and Sharing service must provide the option to temporarily store application-specific data from the application instances that run on the control plane.		
Validation	Completed	Status	Fulfilled
The Distributed Data Storage and Sharing service is not limited to storing monitoring metrics from RAINBOW's *Monitoring Agents*. Other RAINBOW components can also use it to store data in a timeseries format. Upon startup, it initializes and creates 2			

[18] https://github.com/cjdelisle/cjdns
[19] https://github.com/yggdrasil-network

additional caches, one purely in-memory and one for persistent storage, where timestamped data points can be stored.

The Distributed Data Processing service from the RAINBOW stack along with other interested services can use the REST API to store and extract data from the 2 caches. The services can choose whether to use the temporary (in-memory) cache or the persistence-enabled one, depending on their needs.

ID	FR.DSS.7		
Title	**Deployment in geo-distributed realms**		
Requirement Description	The Data Storage and Sharing service must be able to function properly in a geo-distributed environment with heterogeneous networking and physical machines.		
Validation	Completed	Status	Fulfilled

The Distributed Data Storage and Sharing service is built on top of Apache Ignite, which is inheritably suitable for deployment in heterogeneous ecosystems. It is packaged in a docker container and deployed, along with the rest of the RAINBOW stack, in every fog node of the cluster. Each service instance is transparent and does not intrude in the operation of the rest of the services deployed in the fog nodes, limiting its resource needs. It operates under the serverless paradigm which decreases the network needs for internal communication since each instance can directly "talk" with each other without the permission of a leader instance.

Furthermore, since Ignite's mechanisms for data movement (replication/partitioning) are not tailored to dynamic fog environments, our service implements two novel techniques that take into account the nature of fog ecosystems as well as the network limitations. The first technique monitors the fog nodes and frequency of their failures/restarts in order to replicate data from unstable nodes to more stable ones. This helps prevent data loss on failures since the failed node's data can be extracted from the replica nodes.

The second technique cooperates with the Distributed Data Processing service in order to minimize the data movement when the processing service queries data from the storage instances. It decides on data placement destinations based on the location of the processing workers as well as the network latency and data quality/freshness measures.

Combining both placement techniques along with the distributed nature of the Ignite DBMS and the implementation choices described in the previous sections, makes the Distributed Data Storage and Sharing service tailored for dynamic heterogeneous environments comprising of resource-limited fog nodes.

3.3 Documentation and Code Repository

The documentation of the Distributed Data Storage and Sharing service can be found in the respective section of the RAINBOW documentation site:

https://rainbow-h2020.eu/docs/getting-started/rainbow-distributed-data-storage/

The source code of the Distributed Data Storage and Sharing service is open-source and can be found, along with the documentation, in the RAINBOW source code repository:

https://gitlab.com/rainbow-project1/rainbow-storage

3.4 Novel aspects

The Distributed Data Storage and Sharing service presents a novel solution for geo-distributed fog environments that allows for consistent and fail-safe data storage of continuously produced monitoring metrics. It leverages a data-centered based NoSQL database management system and renders it suitable for the fog ecosystem by using local ACID transactions for data ingestion and extraction using a combination of main-memory and persistent data structure. It provides the *Storage Fabric* which makes the data location transparent to the requesting service and allows for secure extraction from any *Storage Agent* in the cluster.

Furthermore, the service by-passes the build-in traditional replication and partitioning mechanisms, which try to place data to different instances in a balanced manner, unsuitable to fog ecosystems. It implements two novel data placement techniques to tackle the most common problems encountered in such ecosystems, i.e., node stability and data movement over the mesh network.

4 Distributed Data Processing Service

In this Section, we present a comprehensive documentation report referring to the final release of the Distributed Data Processing Service as part of the RAINBOW Analytics Stack.

4.1 Overview

4.1.1 The RAINBOW Analytics Stack

The RAINBOW Analytics Stack is comprised of various components and is responsible for the RAINBOW ecosystem's needs for data stream processing so that real-time analytic insights can be extracted from the vast amounts of monitoring data collected from both the underlying fog resources and performance indicators from deployed IoT applications. To this end, the Analytics Stack provides a completely distributed solution, with the data processing performed -in place- right where the data is generated so that analytic insights are extracted with low-latency, and with the collected data never leaving the overlay mesh network interconnecting the collaborating fog nodes.

For RAINBOW, Distributed Stream Processing builds upon the Apache Storm[20] ecosystem with our aim being to not implement yet another distributed data processing engine but rather to design novel scheduling algorithms that are decoupled from the underlying engine and acknowledge the unique settings found in the majority of geo-distributed environments that IoT applications are deployed in. In turn, to ease the rapid design of streaming analytic jobs, RAINBOW also attacks the analytic job programmability challenges. This is achieved by introducing StreamSight, a framework that provides a query model enabling users to utilize a high-level descriptive language to "stich" monitoring streams together and through the application of various aggregations and transformations to generate streams that emit analytic insights.

Figure 3 High-level overview of the components of RAINBOW's Analytics Stackdepicts a high-level overview of the components comprising the RAINBOW Analytics Stack and Figure 4 presents how these components interact with each other and with other RAINBOW services. A typical analytics job starts from the *Analytics Editor* (1). Through the *Analytics Editor* one can design queries by composing *insights* from various monitoring metric streams and through the application of various operators transform the raw streams into an insight stream.

[20] https://storm.apache.org/

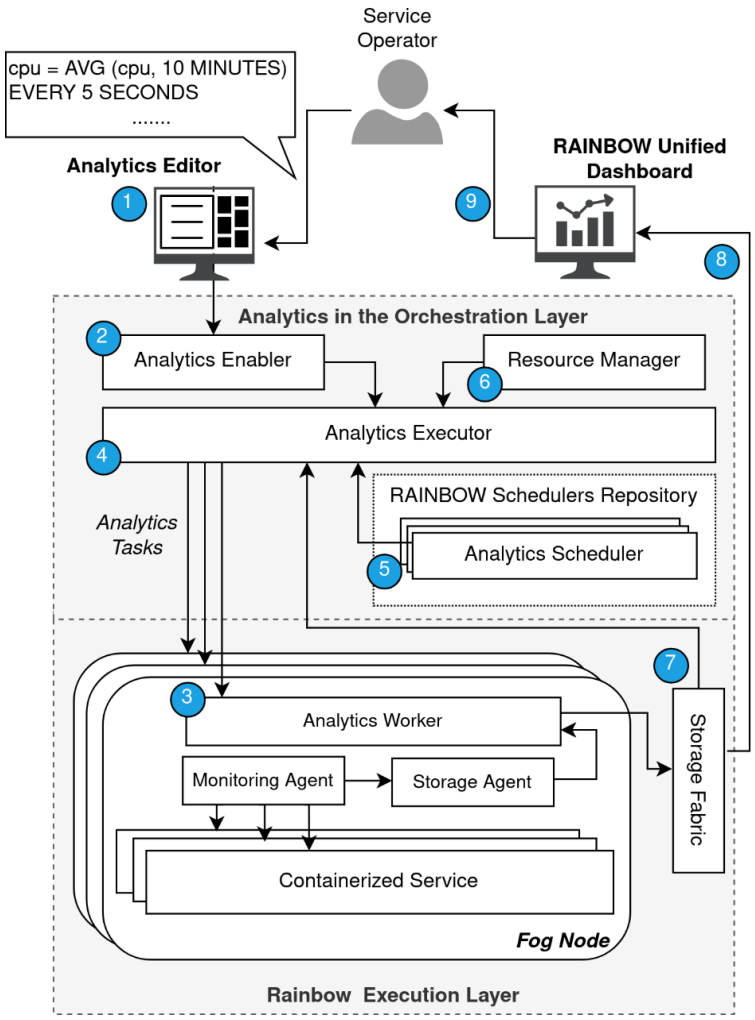


Figure 3 High-level overview of the components of RAINBOW's Analytics Stack

When all desired queries have been designed, the user submits the queries through *Analytics Editor* to the *Analytics Enabler* (2). *Analytics Enabler* will compile the queries to the respective analytics job, ready for deployment. Thus, prior to deployment the queries pass through the StreamSight compiler which will attempt to optimize the query logical plan so that the job is better facilitated for the fog continuum. For example, less operator shuffling will be attempted, and intermediate data generated will be reused instead of re-computed.

After the job optimization, the job is compiled into its final form and is shipped to the *Analytics Executor* (3). The role of this component is to coordinate the job deployment and facilitate the provisioning of the execution environment on the fog nodes that have been denoted as nodes that will host *Analytic Workers* (4). With a provisioned execution environment in hand, the *Analytics Executor* invokes the respective *Analytics Scheduler* (5) from the RAINBOW Schedulers' Repository, which will perform an analytics task

placement algorithm to provide near-to-optimal efficiency for analytic queries based on the user-desired optimization polices. Furthermore, *Analytics Executor* interacts with the *Resource Manager* (6) of the RAINBOW Orchestrator to receive information about the underlying infrastructure resources, like available CPUs, memory, network bandwidth, while data placement metadata will be requested from the *Storage Fabric* (7). It should be mentioned that, in a real deployment, each *Storage Agent* is capable of providing data locality information.

So, in Figure 3, the *Storage Fabric* represents a logical sub-component that abstracts and unifies the functionality offered by inter-connected local *Storage Agents* by providing a decentralized API for access to monitoring data. Hence, monitoring data are immediately made available through the RAINBOW secure overlay mesh network without data needing to be moved to a central (cloud) location that will provide data access but with both a performance penalty and costs incurred for data movement. With information about resource availability and storage metadata, and the RAINBOW-enabler Analytic Scheduler, the *Analytic Executor* executes the job at runtime and supervises its execution, updates the job scheduling, by following the invoked scheduler, and stores the generated results back to *Storage Fabric*. Finally, *RAINBOW Unified Dashboard* retrieves the results from *Storage Fabric* (8) and displays them to the end user (9).

The following sequence diagram of Figure 4 depicts more formally the interactions among the RAINBOW's components and the components of the Analytics' stack. The service operator edits a set of Analytic Queries, written with StreamSight query language, and submits them through the Analytic Editor to the Analytics Enabler. We note that StreamSight and its query model will be discussed in Chapter 5, with this Chapter focusing on Distributed Stream Processing. The Analytics Enabler, as we described before, optimizes the submitted queries and generates an executable streaming artifact. The artifact is submitted to the Analytic Executor, and the executor invokes the respective RAINBOW-enabled Scheduler, requests resources from the Resource Manager, retrieves data placement metadata from the Storage Fabric, and calculates the placement of the required tasks. Then, it assigns the tasks to the Analytics Workers, and the workers execute, store the results, and periodically disseminate heartbeat responses to the Analytics Executor. When the executor has assigned all the tasks, informs the Analytics Enabler, and, consequently, the enabler informs the user through the dashboard. Finally, the user requests the computed results through the dashboard, and the dashboard retrieves them via API call on Storage Fabric.

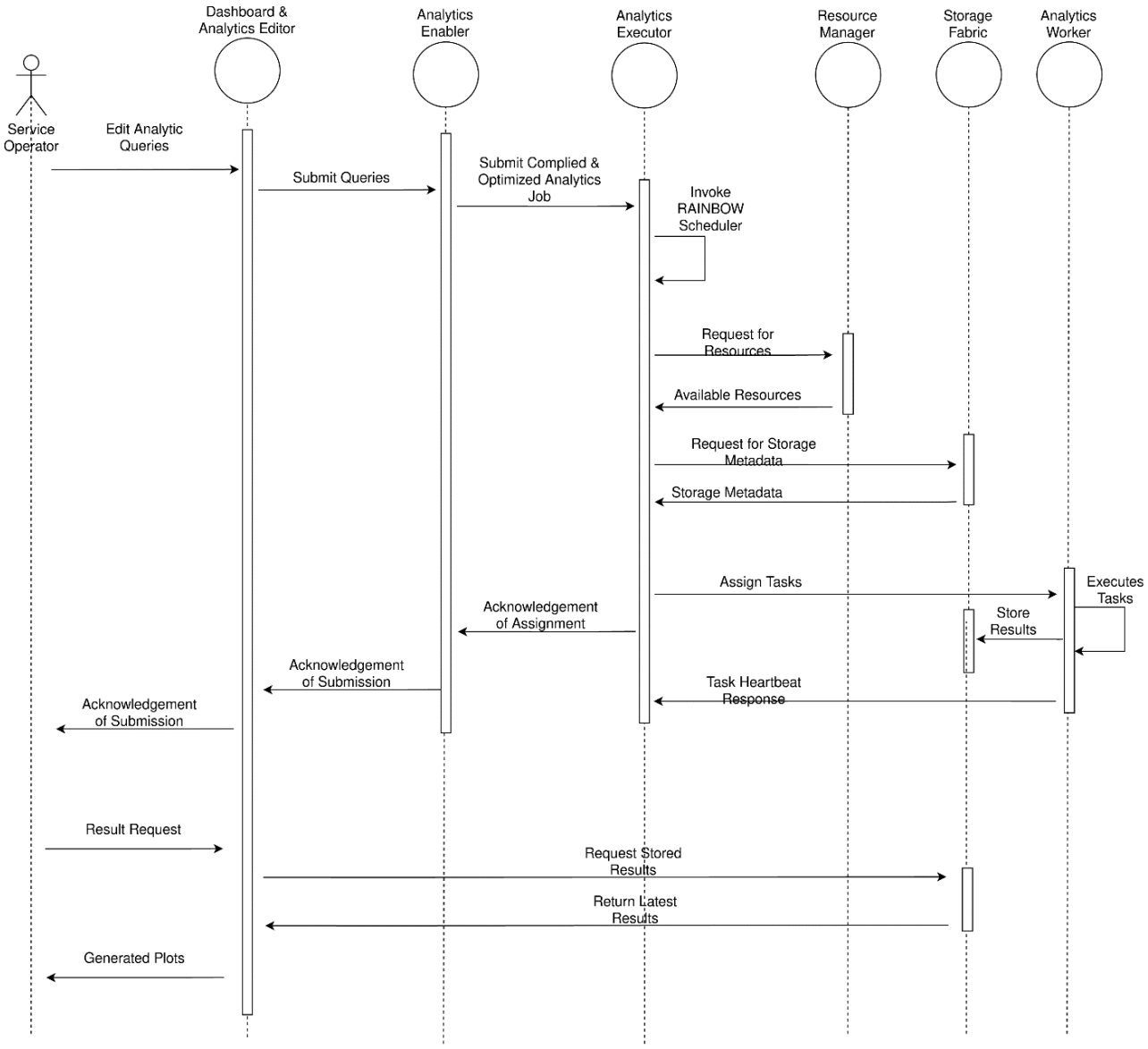


Figure 4 Sequence Diagram of RAINBOW Analytics Stack

4.1.2 Distributed Stream Processing – Enhancing the Apache Storm Ecosystem

Prior to introducing how components of the RAINBOW Analytics Stack fit and interact with the Apache Storm ecosystem, a brief overview of the terminology and how stream processing is achieved by Storm is provided. A Storm cluster is comprised (architecture-wise) of two basic components: a *Leader* node, denoted with the name *Nimbus*, and *Worker* nodes, which are denoted as *Supervisors*. *Nimbus*, quite similar to the *JobTracker* in a MapReduce cluster (e.g., Hadoop), is the entity responsible for the analytics job coordination that includes the scheduling of analytic tasks to *Supervisors* and the overall overview of the cluster lifecycle management (e.g., handling failures). In turn, *Supervisors* are the nodes that accept analytic tasks from *Nimbus* and coordinate their execution on

the local environment they have access to. For RAINBOW this environment is the fog node where the *Supervisor* is deployed on. Hence, the *Supervisors* are the actual implementation of the *Analytics Workers* that the RAINBOW Mesh Stack features. In turn, Nimbus is one of the main software components comprising the *Analytics Executor*. It is worth mentioning that a third component is also required for the successful deployment of a Storm cluster, although not considered part of Storm per se. This third component is *ZooKeeper*[21], which handles the cluster communication overlay between *Nimbus* and the *Supervisor* nodes along with some additional functionality including worker health monitoring.

A high-level overview of how components of the RAINBOW Analytics Stack fit within the Apache Storm ecosystem is showcased in Figure 5. Specifically, the Analytics Executor is responsible for coordinating the execution of streaming analytic jobs on the application's fog nodes that feature a deployed Analytics Worker.

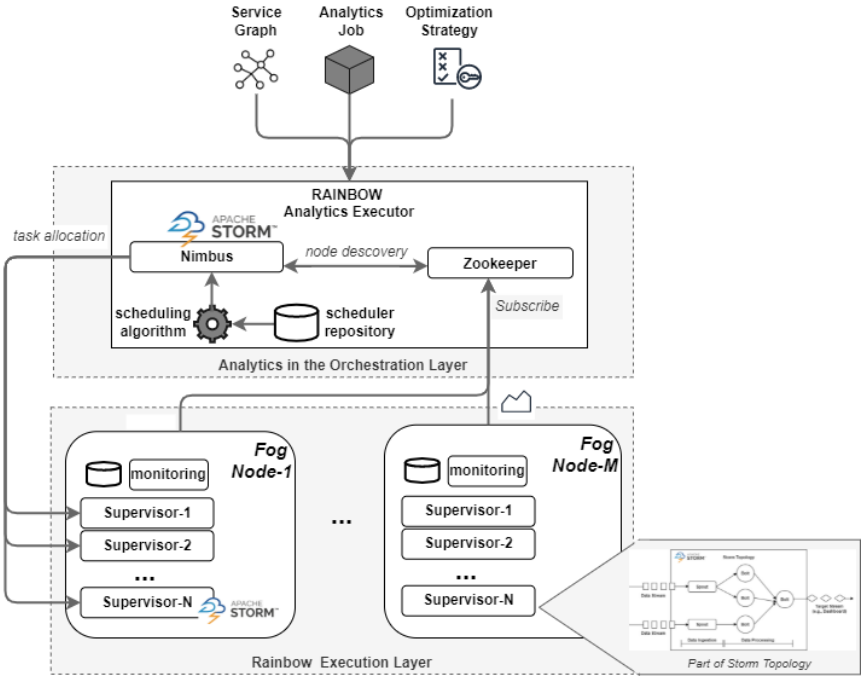


Figure 5: RAINBOW Analytics Stack Components within Storm Ecosystem

As input, the Analytics Executor accepts: (i) the artifacts of the analytics job, this is jar file of the Storm job, and although in Chapter 5 the jar file can include a set of optimization performed by the RAINBOW StreamSight compiler, this can be a vanilla Storm job; (ii) the optimization strategy that the user desires for the deployed job. This may be left to the default fog-agnostic Storm scheduling or may be a request to optimize for performance

[21] https://zookeeper.apache.org/

(reduce streaming latency) or may be a request to explore a trade-off either between performance and data quality or performance and energy consumption. More on the scheduling optimization is showcased in the following section; and (iii) the application Service Graph (SG). The SG is utilized by the scheduling algorithms to have an understanding of the underlying execution environment so that the placement is both resource and topology-aware. For example, when optimizing for performance, one must know the resource availability of the underlying heterogeneous fog nodes and when exploring a trade-off between performance and energy consumption, one must additionally know the power states available by the underlying fog node (e.g., for a Raspberry Pi these are $P_{idle} = 4W$ and $P_{active} = 8W$).

4.1.3 Streaming Job Scheduling

Stream-processing systems are unique and present different challenges than those related to tuning databases and batch-processing systems. First, stream-processing applications are long-running (potentially infinite) programs. Tuning such applications requires determining an experiment's duration such that it is long enough to provide accurate performance measurements of a configuration yet short enough to quickly converge. Second, there are generally two metrics of interest that can be optimized for performance; throughput and latency. Extending the metrics of interest increases the problem dimensionality and therefore, tuning becomes a multi-objective optimization challenge.

In Storm, an analytics query is described as a *Topology*, namely the input data structure received by the Storm cluster for continuous execution and analytics insight extraction. Note that an analytics job may contain multiple queries and therefore, multiple Topologies. In its most simplistic form, a *Topology* is a Directed Acyclic Graph (DAG) comprised of multiple nodes that can take one of two forms. Specifically, nodes can be *Spouts* or *Bolts*, as shown in Figure 6.

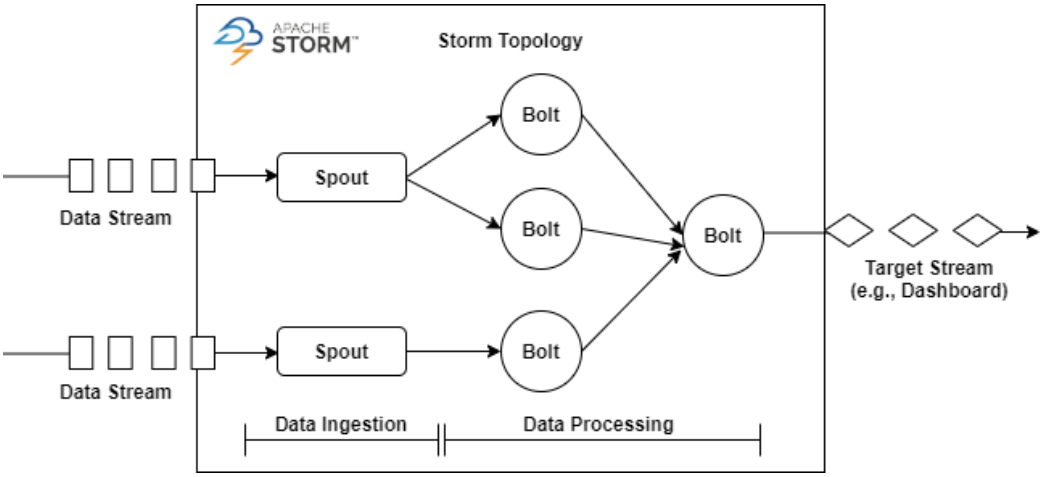


Figure 6: Decomposed Storm Topology

A *Spout* node is linked to a data source and is in charge of handling data ingestion by receiving data as a stream of tuples and delegating these tuples to respective *Bolts*, based on the configured *Topology*. Examples of data sources are various DBMSs, distributed file-systems and even high-performance queueing services. To improve the data ingestion process and support dynamic changes of the underlying environment (i.e., add/remove fog nodes), we have designed and developed a *Spout* node capable of ingesting streaming data from the *Storage Fabric* interconnecting the *Storage* instances deployed over an application's fog nodes.

Through the Spout we have created, access to the Storage Fabric is provided so that monitoring data relevant to the deployed analytic queries can be accessed. Specifically, via the Spout, monitoring data can be extracted in two modes: either through a per *metric* request, where a specific metric is requested via its metric id or through a per *entity* request, where all the metrics relevant to a monitored entity are requested. In the latter case, an entity can be a fog node or a containerized execution environment. With this approach a node global address scheme for data ingestion is not required. Searching for where data is stored is not a job performed by the distributed stream engine but rather requests are delegated to the Storage Fabric which, in turn, performs this operation efficiently as it maintains a global indexing scheme for monitoring data.

Bolts on the other hand, are the nodes performing the actual data processing and can be implemented to perform data aggregations, groupings, filtering and even data transformations. It is worth noting that a Bolt may consume data from multiple streams and, in turn, generate multiple streams that are further connected to other downstream Bolts.

One of the most important factors that lead to the selection of Apache Storm as the underlying stream processing engine for RAINBOW is the ease it provides when there is

a need to customize the assignment of analytic tasks to worker nodes. More specifically, one can actuate a custom scheduler by implementing the `IScheduler` interface, and without the need to resort to source code refactoring. With this functionality the scheduling decisions of the optimization algorithms are enforced into the use-cases. In Particular, when instantiating Nimbus, users as allowed to give as input a Scheduler implementation that adheres to the `IScheduler` interface so that DAG operators are placed on worker nodes based on an algorithm that deviates from the default Storm Scheduler. These operators are the *Spout* and *Bolt* nodes of a Storm *Topology*.

Figure 7 depicts the `IScheduler` interface (v2.3.x). The `IScheduler` interface contains two important methods for implementation. Specifically, the `prepare(Map config, StormMetricsRegistry metrics)` method is called upon (at least) once and provides initial cluster configuration information that may be useful for the custom scheduler implementation. Such information includes the worker nodes, utilization data, etc. In turn, the `schedule(Topologies topologies, Cluster cluster)` method is the method that actually performs the scheduling processing and therefore, allocating DAG operators (segments of the Storm Topology) to Supervisors. The input to the scheduling process is the topologies/queries that must be decomposed into segments and assigned to supervisors, along with the cluster object capturing in a list all the available Supervisors that can process tasks on the cluster.

Modifier and Type	Method and Description
default void	cleanup() called once when the system is shutting down, should be idempotent.
Map	config() This function returns the scheduler's configuration.
void	prepare(Map<String,Object> conf, StormMetricsRegistry metricsRegistry)
void	schedule(Topologies topologies, Cluster cluster) Set assignments for the topologies which needs scheduling.

Figure 7: Storm IScheduler Interface (v2.3.x)

Note that in a static configuration of the underlying infrastructure or in a homogeneous worker node cluster, the scheduling process only needs to run once as no changes are foreseen to the deployment. However, this is far from the case in a highly dynamic fog continuum with heterogeneous fog nodes that are dynamically (de-) provisioned and hence, the `prepare(…)` method is actually called upon each time the scheduling is updated. Hence, to support the periodic update of the storm scheduling process, all custom schedulers designed by RAINBOW include in the `prepare(…)` method the following data that is extracted from the Storage Fabric and the Orchestrator's Resource Manager: (i) the IoT application's service graph; (ii) the fog nodes in the current `cluster` along with their current resource capabilities (i.e., cpu-speed, power levels); and (iii) up-to-date monitoring data for the fog nodes along with network statistics that include link quality and latency.

The RAINBOW Analytics Stack features 4 implementations of Storm Schedulers. The first Scheduler is dubbed as the "**`BaselineScheduler`**", as it essentially performs fog-agnostic optimization by adopting the default Storm Scheduler. As explained in Section 2.2, the default Storm Scheduler adopts a "fairness" policy where tasks are placed on worker nodes in a (pseudo-) round-robin fashion. This Scheduler is extended to accept cluster information in the form of a RAINBOW Service Graph and monitoring data accessed through the RAINBOW Storage Fabric. This Scheduler is available from the first RAINBOW Platform release and in second year of the project (Y2) was updated to incorporate the new updates made to the Service Graph model (D3.2).

The second RAINBOW Scheduler is designed to acknowledge both resource and network heterogeneity. For (reference) simplicity this scheduler is dubbed as the "**`PerfScheduler`**", as it inherently optimizes the performance of the stream processing by considering as the most important QoS metric the average latency of the Storm Topology. As explained in D4.1, the average latency is defined as the latency measured from the slowest path in the DAG (from data source to sink node) with regards to a single input data batch and consists of the average communication latency between the operators in the path. The batch size is configurable and depends on the scheduling periodicity. In turn, the aforementioned slowest path is denoted as the *critical path* and for fog deployments extending to geo-distributed realms (just like all RAINBOW Demonstrators) the communication overhead is the dominating factor contributing to the total stream processing performance. This is justified, as the execution latency of each operator (bolt) is considered negligible when the operators receive sufficient resources for execution.

The **`PerfScheduler`** capitalizes and extends research from AUTH/UCY [17], where an algorithmic process is designed to model and solve the Storm operator placement problem as a constraint satisfaction problem. In brief, the constraint satisfaction process attempts to process an operator mapping to Analytic Worker nodes, in which the latency of the critical path is minimized while ensuring, at the same time, that the compute and memory capabilities of an operator can be fulfilled by the candidate Worker nodes in terms of CPU cores, speed and available RAM. However, leaving the modeled placement problem as-is is not a realistic option as the defined problem is NP-hard. To tackle the high complexity and make the scheduling ideal for stream processing, we adopt two low computational heuristics that when combined together form the final hybrid efficient scheduling solution. First, the scheduling solves a relaxed version of the Non-Linear Programming Formula for each operator considering the placement of its parent nodes and then further optimize it heuristically. Nonetheless, the previous solution may employ numerous Workers unnecessarily and easily fall into local optima. Thus, we additionally

use a spring relaxation algorithm that produces a solution with low or no intra-operator parallelism that in the end, outputs a close to the optimal placement with linear complexity. Finally, we note that this Scheduler is available from the first RAINBOW Platform release and in Y2 was updated to incorporate the new updates made to the Service Graph model (D3.2), while a number of minor bug fixes were performed after being pinpointed through the both the RAINBOW testing process and the Demonstrators.

4.1.4 New RAINBOW-Enabled Schedulers for Apache Storm

This Section introduces the two new custom Schedulers designed and implemented for Storm engines deployed in fog realms. Examples of how the previous and new RAINBOW Schedulers work are shown in Section 4.3 (Requirements Fulfillment).

4.1.4.1 PerfDQScheduler: exploring trade-off between performance and data quality

The third RAINBOW Scheduler is designed with the intent to optimize the placement of Storm operators to Analytic Worker nodes by considering data quality as a first-class citizen with a bi-objective optimization process considering both performance and data quality. This Scheduler extends the `PerfScheduler` and for (reference) simplicity is dubbed as the "**`PerfDQScheduler`**". Data quality is an important aspect for IoT applications. Low quality can lead to less useful results as analytic insights are produced with high uncertainty that may impact mission-critical tasks (i.e., pedestrian detection for self-piloting cars/drones). The quality of data can be categorized into multiple dimensions. Some examples of these are *completeness*, *timeliness* and *accuracy*. Some of the most common factors that lead to decreased data quality include the heterogeneity of data sources, missing and dirty data due to network malfunctions or security constraints. Data quality at a first glance, may not seem as a significant overhead, but the reality is far from it when faced with a resource-constraint environment and scenarios with high data inter-arrival rates.

Towards this, the design of the `PerfDQScheduler` is motivated by the fact that the more the quality checks, the less an Analytics Worker on a fog node can be assigned tasks of upstream operators, thus inducing higher communication cost (i.e., more workers required), which is a direct contradiction to the performance optimization. Hence, the `PerfDQScheduler` algorithmic design attempts to optimize the level at which data quality checks are performed considering the computational and memory overhead they impose. As such, the scheduling process limits the fraction of the data for which data quality checks are performed in a systematic manner and regardless of what exactly these checks are, with the aim being to extend quality checks as much as possible but without contributing to the increase of the communication latency.

Now, since the communication latency is the dominating inhibitor in data stream processing, the data quality checks take a share of the worker node computational (and memory) resources and this share is "just enough" to not require additional workers or an increase in data shuffling. To achieve this, the algorithmic process extends the `PerfScheduler` with the data quality problem dimension. This includes adding the data quality check(s) resource requirements to the constraint satisfaction process, which are only realized after an initial learning phase of which these requirements are profiled through compute and memory data provided by the RAINBOW Monitoring. As a final note, although another constraint is added to the constraint satisfaction process, thanks to the novel hybrid process adopted in the algorithmic process (linear programing with heuristics, spring relaxation), the computation complexity does not actually increase.

4.1.4.2 PerfEnergyScheduler: exploring trade-off between performance and energy-consumption

On a similar note, the fourth RAINBOW Scheduler, dubbed "**`PerfEnergyScheduler`**", is designed with the intent to optimize the placement of Storm operators to Analytic Worker nodes via a bi-objective optimization process considering both performance and the topology energy-consumption. Energy-awareness, and subsequently green computing, is an important aspect for fog computing, especially when fog nodes are battery-powered. The intent of this scheduler is that in a heterogeneous fog environment, fog nodes may present not only different resource and network capacity but also different power levels. The latter is particularly important as many nodes may be "selectable" for the placement of a Storm operator in terms of their computational/memory capabilities. However, if they feature different power levels ($E = P \cdot t$) then the energy consumption contributed to performing the analytic task will incur a higher energy footprint if the more power-hungry nodes are selected. To give an example, a Raspberry Pi 4 model B presents a P_{idle} of $4W$ and P_{active} of $8W$, while the power of an Nvidia Jetson Nano may range from 32-56W. If an operator can satisfactorily run on the Raspberry Pi, then there will be significant energy savings. Moreover, the problem can become even worse if the topology is battery-powered, as the selection of the power-hungry nodes, instead of the equivalent energy-saving nodes, can severely impact an analytic job in the near future as the power-hungry nodes may be deemed unavailable, when needed, very early on due to battery exhaustion.

Towards this, the `PerfEnergyScheduler` algorithmic design capitalizes on the `PerfScheduler`. Specifically, the resource list of a fog node is extended to include, other than computational and memory profile, a power profiling that features the node's power levels (given as a distinct list or range) and if the node is battery-powered. Just like with

the `PerfDQScheduler`, if power related information is not annotated on a fog node in the service graph, then the realization of the energy-aware scheduling will only kick-in after an initial learning phase of which these requirements are profiled.

In terms of profiling, a linear regression energy model is constructed to assess the computational, I/O and network energy consumption contribution. As such, during the operator to fog node mapping, the compute and memory constraints are first satisfied and then the nodes that satisfy the requirements are sorted based on their power profile and the least power-hungry nodes are selected. At the end, by selecting the energy-saving nodes some share of communication latency is traded for achieving overall energy savings as the number of nodes required in the stream processing may be more.

As a final note, this scheduler features and additional weighting parameter, denoted as $w \in [0,1]$, that enables the user to configure how energy saving the algorithm placement should be so that latency is not hampered beyond expectations. A $w = 0$ denotes that the algorithm is not actually energy-aware and resorts to a pure resource-aware implementation (PerfScheduler), a $w = 1$ denotes that the algorithm is fully energy-aware, while a value in between applies the weighting process in the operator-to-node mapping process.

4.2 Additional Functionality and Improvements

This subsection introduces additional functionality that has been added to the RAINBOW Analytics stack and is relevant to distributed data processing. We note that the two Schedulers for Storm that have been introduced in the previous section are also new functionality, however, to provide a more structured reading experience for Storm and the scheduling process, these have been included in the aforementioned section.

4.2.1 The Fogify Emulator

A plethora of challenges inhibits the ease of experimentation and testing of edge and fog deployments that deluges our ability to understand the performance inefficiencies of these deployments, while also evaluating the efficacy and efficiency of scheduling algorithms designed to optimize analytic job placement. In order to perform large-scale experimentation, one requires a significant investment for physical fog devices, but even with this, configuring and setting up experiments with heterogeneous resources, networking and performing this at a geo-scale is a slow process. The difficulty in experimentation increases even more if entities can be mobile and if certain tests must be made under extreme conditions, including device failures, link drops and workload variations. To overcome these challenges and facilitate the rapid design of experiment

testbeds for large-scale data-intensive IoT applications, we have created an emulator as part of the RAINBOW Analytics Stack.

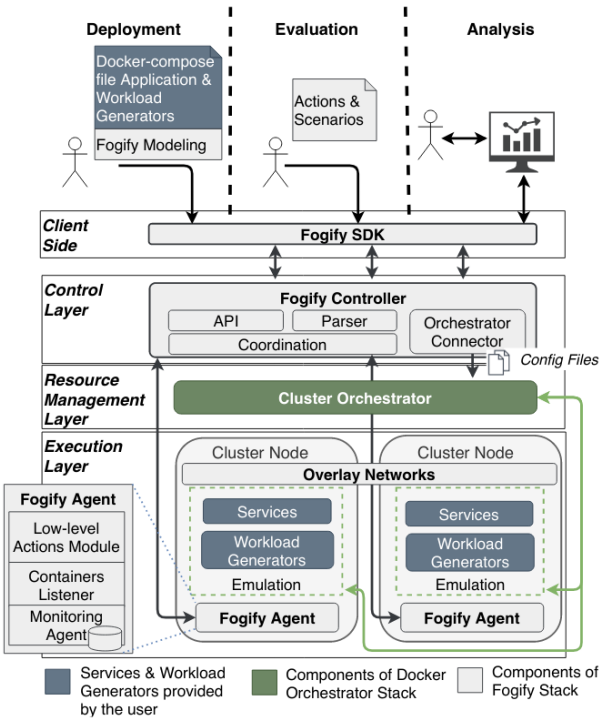


Figure 8: High-Level Overview and Logical Interplay of the Fogify Emulator for Data-Intensive IoT Applications

The Fogify emulator[22] enables RAINBOW users to (i) design large-scale fog testbeds using modeling abstractions to configure fog nodes and networks, (ii) deploy the emulated infrastructure on a user's laptop or a computer cluster for large-scale experiments, and (iii) rapidly define reproducible experiments and "what-if" scenarios. To use Fogify, no changes are required to the business logic of a containerized IoT application, with users only extending their Docker Compose description with Fogify primitives to describe the desired emulated testbed (where the app "sits" on). Specifically, users can annotate application services with infrastructure requirements (i.e., cores, clock speed, memory, disk) and configure network connectivity between infrastructure offerings (i.e., uplink, downlink, packet drops, bandwidth). Upon executing an experiment, the Fogify Controller will provision the emulated testbed and during runtime, collect a plethora of monitoring data that can be used to profile the deployment and assess desired KPIs. During runtime, Fogify can execute ad-hoc "one-off" actions or an entire script of actions with these including horizontal/vertical scaling actions, workload fluctuations, link drops and/or quality variations and node failures. With such actions, one can realistically assess the efficacy and efficiency of a deployment.

[22] https://ucy-linc-lab.github.io/fogify/

It is important to note that with Fogify the application services are actually run (wall clock time) and produce data, with only the fog infrastructure emulated by shaping the isolated containerized runtimes accordingly. To achieve this, two key novelties are embedded in the Fogify Controller [39].

Although, containerized runtimes offer resource isolation through linux namespaces and resource constraining through cgroups, mapping emulation requirements for compute resources is not straightforward like memory (e.g., 2GB emulated node mapped to 2GB container "sliced" from larger host). Hence, to enforce compute limits (aka "cpu-capping"), and get a CPU @ 1.7GHz on a 16 core @2.3GHz host, Fogify has contributed to the extension of linux container CPU cgroup with the ability to set a proportion of the host CPU allocated to emulated node. This is achieved with the introduction of a new metric denoted as the Cumulative Clock Rate (CCR) and equal to the number of cores multiplied by the CPU clock speed. With this, the CPU rate can be configured to $nod\ e_{CCR}/hos\ t_{CCR}$. The second key novelty of Fogify is the provisioning of the network fabric for the emulated testbed. In brief, Fogify builds upon the RAINBOW networking logic where an overlay mesh network is deployed per topology (VXLAN) and a side-car proxy sits on every node. The proxy separates in-out traffic and builds a tree-like structure by utilizing Classful Queuing Disciplines (qdisc) to support network rule chaining and traffic filtering which act as QoS queues.

4.2.2 The 5G-Slicer Network Plugin

While Fogify can create realistic emulation experiment testbeds, it makes one key assumption; it considers that the emulated infrastructure is fixed in position. To overcome this limitation, the RAINBOW Analytics Stack employs a Fogify plugin, denoted as 5G-Slicer [40]. This plugin enables fog infrastructure with computing and analytic capabilities to present mobility. Therefore, with 5G-Slicer, the relevant modeling abstractions are provided for mobile entities, trajectory updating and mobile connectivity. In turn, when mobility is involved in computations, the network connectivity varies and therefore, 5G-Slicer enhances the Fogify network overlay so as to update signal strength based on various distance-based models and 5G connectivity protocols including massive MIMO (Multiple-Input Multiple-Output) and antenna beamforming.

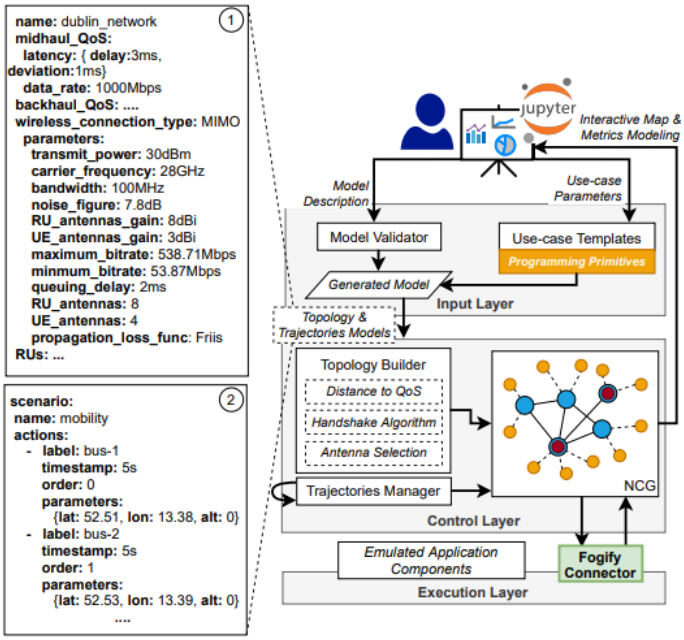


Figure 9: 5G-Slicer Network and Mobility Modeling Plug-in for the Fogify Emulator

4.3 Requirements Fulfillment

This Section provides a report on the fulfillment of the requirements list of the Distributed Data Processing service as documented in D4.1.

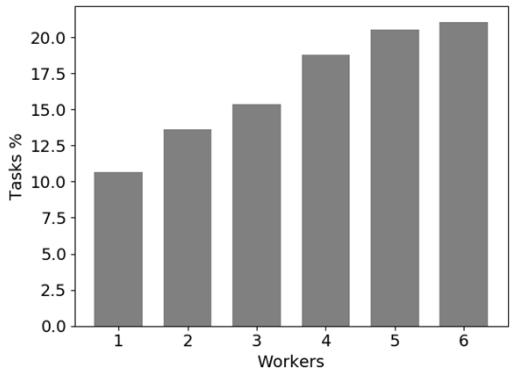
To avoid repetition, we note that all experiments conducted in this Section have been conducted via the Fogify emulator (except FR.DPS.3) and the experiment descriptions are openly available and can be reproduced. Requirement FR.DPS.3 is showcased through a real deployment so that the energy-aware scheduling is fed with real data for the worker nodes' power consumption.

ID	FR.DPS.1		
Title	**Execute analytic tasks over heterogeneous fog nodes**		
Requirement Description	The RAINBOW Distributed Data Processing Service must provide the means for its *Analytic Workers* to execute analytic tasks in place and over heterogeneous fog nodes. This means that fog nodes may be configured with a wide range of resource capabilities, while the configuration of the Analytics Workers must be done with full transparency and no additional effort from RAINBOW users (zero-conf).		
Validation	Completed	Status	Fulfilled

The Distributed Data Processing Service's *Analytics Workers* adopt the vanilla implementation of Apache Storm Supervisors (v2.3.x – to date latest stable version). The Workers (Storm Supervisors) have been packaged and configured as deployable Docker containerized services so that the deployment of an Analytics Worker can be performed by the RAINBOW Orchestrator on requested Fog Nodes that meet Storm Supervisor minimum system requirements (2 vCPU, 2GB RAM). Storm Supervisors are inherently deployable on heterogeneous host environments, however the scheduling of analytics jobs, by the default Storm Scheduler, does not acknowledge the underlying heterogeneity as scheduling simply embraces a "fairness" task allocation policy. Towards this, the RAINBOW Monitoring reports the underlying environment capacity in terms of CPU clock speed, CPU core count and memory allocated to Storm. These metrics are periodically collected as the environment may be dynamically altered due to vertical scaling actions (see FR.DPS.4). With these metrics, the RAINBOW Analytics Job Schedulers take into consideration the heterogeneity of the entire fog topology allocated to the reference service and schedules tasks according the capabilities of the underlying fog nodes to improve the overall performance of the analytics jobs.

An example of a heterogeneous fog environment is shown below, where 6 nodes are provisioned and a RAINBOW Analytics Worker is deployed on each of them. These workers are configured with different computational processing capabilities while memory is set to 4GB on all nodes and network connectivity is stable with a fixed (artificial) latency of 20ms set on downlink and uplink connections. In the depicted plot, we see that in order to meet the performance requirements for the given analytics job (expressed in terms of latency), the RAINBOW-enabled PerfScheduler that optimizes jobs for performance, allocates more tasks to the powerful nodes so that the job can achieve the desired performance.

Worker	Processing Power
Worker-1	1500MHz
Worker-2	2500MHz
Worker-3	3500MHz
Worker-4	4500MHz
Worker-5	5500MHz
Worker-6	6500MHz



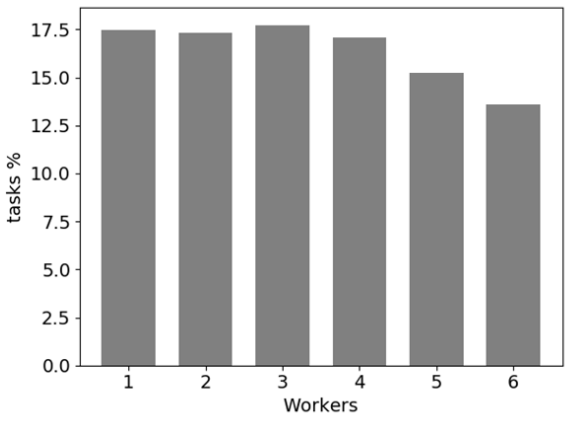
ID	FR.DPS.2
Title	**Deploy analytic jobs in geo-distributed fog topologies**

Requirement Description	The Distributed Data Processing Service must be able to deploy and execute analytic jobs over geo-distributed environments and even function under heterogeneous networking settings between the *Analytics Workers* and the *Analytics Executor*.		
Validation	Completed	Status	Fulfilled

During deployment, the Analytics Workers (Storm Supervisors) network configuration is automatically appended with the necessary configurations so that network traffic is routed through the RAINBOW overlay mesh network that is established to support secure, encrypted and reactive routing among the fog nodes dedicated to the deployed application. This configuration is completely transparent to the user and no manual effort is required whatsoever. In regards to network heterogeneity, something considered as the norm in edge and fog environments, the RAINBOW-enabled Storm Schedulers take into consideration this case as well. Specifically, this Scheduler takes advantage of the detailed monitoring statistics made available by the RAINBOW Monitoring, including connection latency, bandwidth and error rate, for task allocation.

An example of a network heterogeneous fog environment is shown below, where 6 nodes are provisioned and a RAINBOW Analytics Worker is deployed on each of them. These workers are configured with the same computational processing capabilities but present different network connectivity in terms of the latency that is artificially injected to their downlink and uplink connections. In the depicted plot, we see that to meet the performance requirements for the given analytics job, the RAINBOW-enabled PerfScheduler that optimizes jobs for performance, allocates less tasks to the more "distant" nodes so that the job can achieve the desired performance.

Worker	Latency
Worker-1	0ms
Worker-2	7ms
Worker-3	14ms
Worker-4	21ms
Worker-5	28ms
Worker-6	35ms



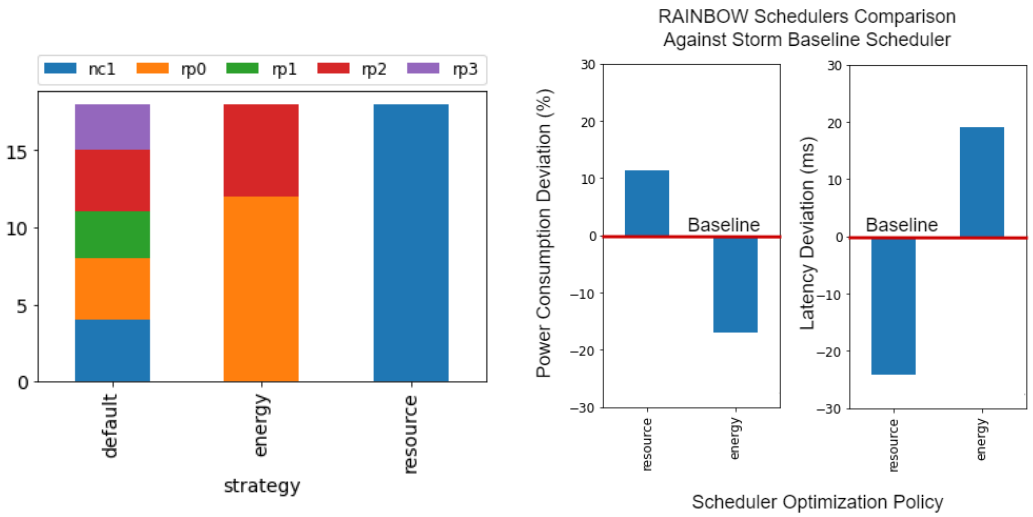
ID	FR.DPS.3		
Title	**Analytics task placement based on different job optimization strategies for geo-distributed fog realms**		
Requirement Description	The RAINBOW Distributed Data Processing Service should provide users with the flexibility of selecting the policy/policies under which the task placement and execution of their analytics jobs will be optimized by the *Analytics Scheduler*. Specifically, RAINBOW must support a wide range of schedulers, each of which are embedded with an algorithmic process capable of optimizing the task placement of IoT applications deployed even in geo-distributed fog environments.		
Validation	Completed	Status	Fulfilled

The RAINBOW Analytics stack is not bounded to a specific scheduler for streaming analytic jobs. Rather, RAINBOW extends the IScheduler interface of Storm by enriching it with the ability to parse RAINBOW Service Graphs and the periodic extraction of requested monitoring data. With this, RAINBOW platform developers (and even users) may design their own optimization strategies and implement schedulers that can be used by the distributed stream processing of RAINBOW. To date, RAINBOW provides users with 4 Storm Schedulers (introduced in detail in Sections 4.1.3 and 4.1.4). The BaselineScheduler maps tasks to fog nodes by adopting a fairness strategy using a round-robin allocation mode; The PerfScheduler acknowledges both the heterogeneity of the underlying fog nodes resources and network link performance to optimize the scheduling by minimizing stream processing latency; The PerfDQScheduler extends the performance scheduler to also consider data quality as an addition problem dimension; and the PerfEnergyScheduler explores tradeoffs between power levels and performance to incur energy savings when running streaming analytic jobs.

The following depicts an example of a fog deployment running an analytics job composed of 5 StreamSight queries executed using 3 different schedulers to see how the deployment is optimized depending on the embraced scheduler. The schedulers under-examination are the BaselineScheduler, PerfScheduler (denoted in plots as resource) and the PerfEnergyScheduler (denoted in plots as energy). The deployment for all three experiment runs is comprised of 5 fog nodes including 1 Dell PowerEdge R610 server featuring 12cores@2.4GHz with 12GB RAM and power ranging from 70-200W (denoted as *nc*) and 4 Raspberry Pi's v4 model B equipped with a quad core

ARM Cortex-A72@1.5GHz and 4GB RAM and power ranging from 4-8W (denoted as *rp0-3*). Moreover, rp1 and rp3 are battery-powered.

The following 2 figures highlight the results of the experimentation. Specifically, the left plot depicts the percentage of operators mapped to each fog node, while the right 2 plots depict the overall power consumption and latency incurred of the two RAINBOW Schedulers when compared to the baseline. In the left plot we immediately observe that in the case of the baseline scheduler, tasks are allocated fairly to all fog nodes irrespective of their resource capabilities. In turn, when embracing the PerfScheduler, all tasks are allocated to the powerful *nc1* as resources permit this and communication latency is minimized. This has a direct positive effect on the net latency that drops (in comparison to the baseline) by 24ms. However, to sufficiently achieve the required computations, *nc1* operates at high power levels that result a 12% increment in the net energy consumption. On the other hand, when embracing the PerfEnergyScheduler, the tasks are shared among the two Raspberry Pi's that are "plugged-in" and this has a direct effect in the net energy savings that drop by 18% when compared to the baseline, albeit the net latency (as expected) increases by 18ms.



ID	FR.DPS.4
Title	**Operation under dynamic topology adaptation**
Requirement Description	The Distributed Data Processing Service must be able to acknowledge dynamic alterations of the underlying infrastructure. These alterations may take various forms and include the change of

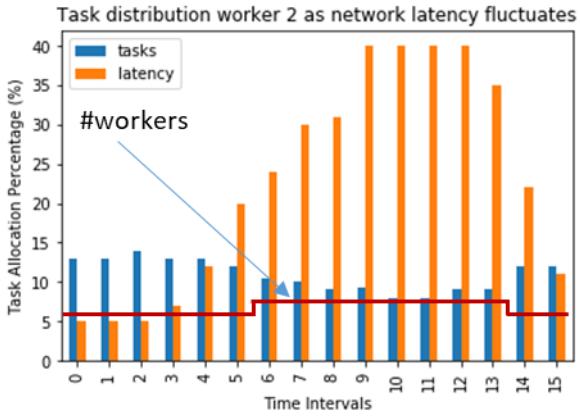
	provisioned resources, including the alteration of the current fog node(s) resources and/or the addition/removal of fog nodes.		
Validation	Completed	Status	Fulfilled

As previously mentioned, RAINBOW Analytic Workers are containerized and when new fog nodes are added (i.e., via horizontal scaling), the workers are both deployed and configured (i.e., Analytics Executor IP) automatically by the RAINBOW Orchestrator. In turn, the `prepare(…)` method of the RAINBOW-enabled Storm Schedulers is always called upon when the scheduling is periodically executed and through this method, data regarding the current Storm topology are updated on the Service Graph and passed to the scheduling while monitoring metrics are also received so that capacity and resource availability (e.g., vertical scaling, network fluctuations) are considered during the updated task allocation.

An example of a heterogeneous fog environment is shown below, where 6 nodes are provisioned and a RAINBOW Analytics Worker is deployed on each of them. These workers are configured with different computational processing capabilities while memory is set to 4GB on all nodes and network connectivity is stable with a fixed (artificial) latency of 20ms set on downlink and uplink connections. In the depicted plot, we see that in order to meet the performance requirements for the given analytics job (expressed in terms of latency), the RAINBOW-enabled resource-aware Storm Scheduler that optimizes jobs for performance, allocates more tasks to the powerful nodes so that the job can achieve the desired performance.

As an example, let us consider a resource heterogeneous deployment similar to the deployment presented in FR.DPS.1. In this, 6 nodes are initially provisioned and a RAINBOW Analytics Worker is deployed on each of them. These workers are initially configured with the same network connectivity. In this experiment a RAINBOW Scheduler SLO is described using the StreamSight Query Model through the RAINBOW Dashboard and the SLO is passed to the RAINBOW Orchestrator for runtime assessment. If a violation occurs due to the average network latency rising over 20ms then a new fog node is added to the stream processing and similarly, when the network latency drops below 20ms a node is removed. For the Storm scheduling we adopt the RAINBOW-enabled BaselineScheduler. In the plot that follows, we observe the task allocation for Worker-2 where initially this worker receives approx. 13% and when latency increases beyond the threshold, a new worker is added and we observe that Worker-2 task allocation drops to 8% by “sharing” (with a short delay) part of its load with the new worker. Similarly, when the connectivity problems seize to exist, a

fog node is decommissioned and the percentage of tasks that Worker-2 receives gradually increases to 12%.



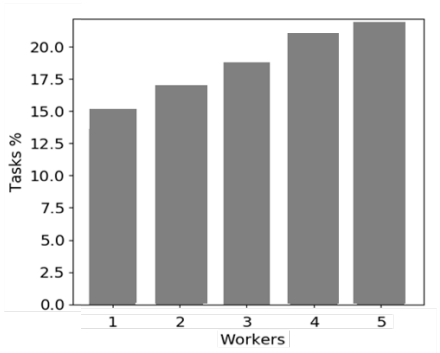
ID	FR.DPS.5		
Title	**Operation under unexpected events and extreme network uncertainties**		
Requirement Description	The Distributed Data Processing Service must be able to both acknowledge that the current deployment is undergoing unexpected events, at the same time, continue seamlessly and uninterrupted the execution of analytic jobs. These unexpected "events" may take various forms and include sudden increases in link/network latencies, the appearance of link failures, temporal link disconnections, node processing saturation and complete device fail-stops.		
Validation	Completed	Status	Fulfilled

RAINBOW-enabled Storm Schedulers feature the ability to internally speculate the time duration of running streaming jobs that compute continuous analytic insights based on monitoring data collected by the RAINBOW-Monitoring through its Storm probing interface. Analytics job speculation is an optional feature that users can enable through the RAINBOW Dashboard. When speculation is enabled, any performance degradation (node task latency is X percent above the median) is reported and the RAINBOW-enabled Scheduler decides if the alteration of the underlying infrastructure is transient or permanent. In the first case, the system temporarily redirects the load from problematic workers to other, under-utilized,

Analytics Workers so that it can self-stabilize under transient faults. If the problem persists after a considerable time interval (e.g., a node fail-stop), then the algorithmic process will not assign tasks to these faulty nodes. As transient uncertainties are significantly harder to cope with, the following examples will illustrate such cases.

In the first example, we consider the resource heterogeneous fog environment described in FR.DPS.1, where we observe (left plot) that the RAINBOW-enabled PerfScheduler, allocates larger percentage of tasks to the more powerful nodes. Now, consider that the first two worker nodes are introduced to an artificial CPU workload that undercuts the CPU power of these nodes that is made available to Storm by 75%. We observe (right plot) that the RAINBOW-enabled Storm Scheduler that optimizes for performance, acknowledges the slow performing workers, labels them as stragglers, and allocates larger portions of tasks to the remaining nodes so that performance targets can be achieved.

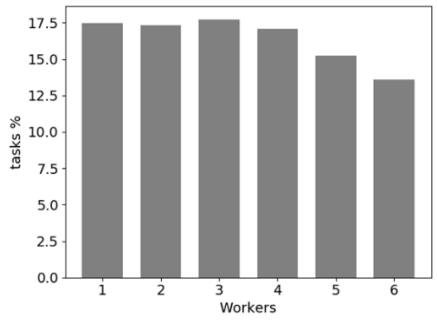
Worker	Processing Power
Worker-1	1500MHz
Worker-2	2500MHz
Worker-3	3500MHz
Worker-4	4500MHz
Worker-5	5500MHz

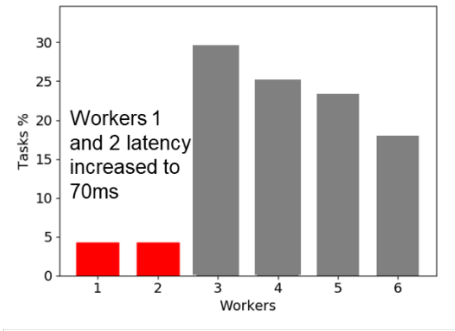


In the second example, we consider the network heterogeneous fog environment described in FR.DPS.2, where we observe (left plot) that the RAINBOW-enabled PerfDQScheduler that optimizes for performance and data quality, also allocates larger percentage of tasks to nodes that feature "better" connectivity in terms of latency.

Now, consider that the first two worker nodes are introduced to an increment of their latency to 70ms that impairs the network connectivity and will cause delays in data movement for analytic computations. We observe (right plot) that the RAINBOW-enabled Storm Scheduler, acknowledges the network instability, labels these two nodes as stragglers, and temporarily "penalizes" them by allocating larger portions of tasks to the remaining nodes so that performance targets can be achieved.

Worker	Latency
Worker-1	0ms
Worker-2	7ms
Worker-3	14ms
Worker-4	21ms
Worker-5	28ms
Worker-6	35ms





4.4 Documentation and Code Repository

The documentation of the RAINBOW Analytics Stack can be found in the respective section of the RAINBOW documentation site:

https://rainbow-h2020.eu/docs/getting-started/rainbow-analytics/

The documentation includes a getting started guide, examples and a complete documentation of the service API calls and service interfaces.

The source code of the RAINBOW Analytics Stack is open-source and can be found in the RAINBOW source code repository:

https://gitlab.com/rainbow-project1/rainbow-analytics

5 Fog Analytics Service – StreamSight

In this Section, we present a comprehensive documentation report referring to the final release of the Fog Analytics Service denoted as StreamSight.

5.1 Overview

One of the key components of the RAINBOW Analytics Stack is StreamSight. In particular, StreamSight provides a high-level declarative language that is based on a query model intended to ease the definition of streaming analytic queries, while also providing automated query optimizations specifically tailored for fog deployments. StreamSight is part of the RAINBOW Dashboard and User Services Layer, while Figure 10 depicts a high-level overview of the basic components that StreamSight is comprised of.

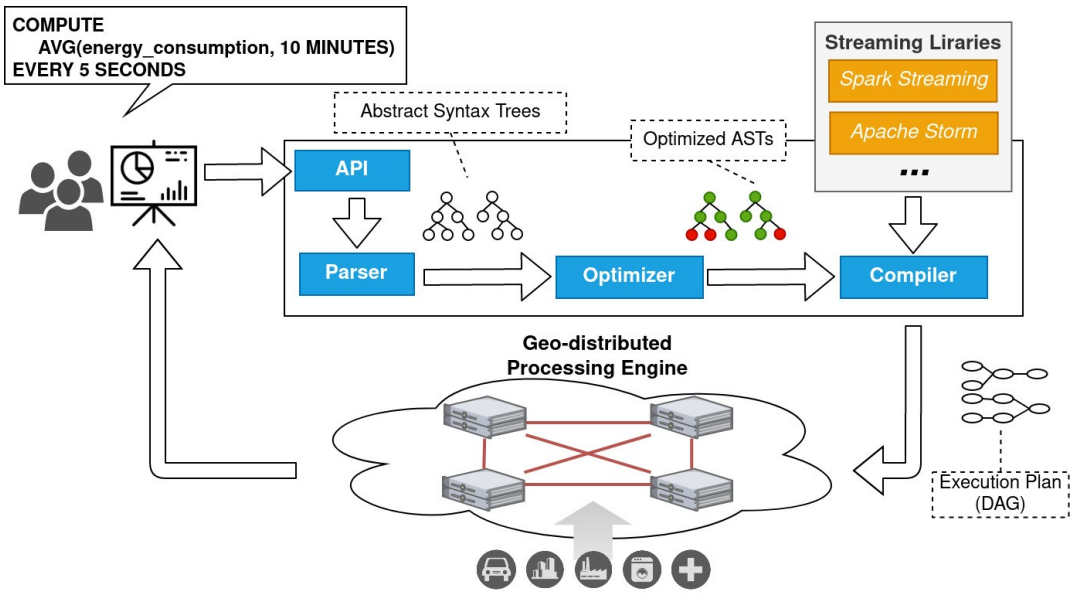


Figure 10: High-Level Overview of StreamSight

In brief, users can take advantage of StreamSight through the Analytics Editor Perspective of the RAINBOW Dashboard (see Section 5.2) where a declarative SQL-like query language can be used to compose streaming analytic queries that extract insights relevant to IoT applications from the monitoring streams exposed by RAINBOW Monitoring through the Storage Fabric. A notable feature of the StreamSight query model is that it is completely decouple from the underlying distributed processing engine, making the queries (and entire analytic jobs) reusable on different engines, while also alleviating the steep learning curve and significant cost for development and debugging when in need to adopt different distributed processing engine programming models.

Once a query is given by the user, the StreamSight *Parser* (in the background) will parse the query and translate it into an Abstract Syntax Tree (AST) representation. An AST expresses the language's grammar rules, and the final level of the tree are the tokens and symbols of the query model. If no valid AST can be constructed from a query, the process stops and returns the suitable error message. Through this process, the Parser guarantees the correctness of the submitted queries. With the AST in hand, the StreamSight *Optimizer* will attempt to optimize the AST so a "better" query logical plan can be derived. "Better" here means that the query will be optimized for fog deployments by extracting correlations between queries of the same job, pruning unnecessary computations and reducing data shuffling to reduce, in turn, communication latency.

In turn, the AST can also be annotated with the scheduling policies introduced in Chapter 4. Up to this point, the queries of a streaming analytic job are interoperable and completely decoupled from the underlying distributed engine. Hence, the next StreamSight component is the *Compiler* which takes as input the optimized plans and generates the final executable artifact. To achieve this, the Complier recursively traverses the optimized plans and "translates" each operator to the underlying distributed engine code. The generated code should be specific for each underlying engine, so during the translation process, Optimizer invokes the selected streaming library for the respected underlying distributed data processing engine. To date, StreamSight supports three streaming library compilers, namely for Spark-Streaming, Storm and RAINBOW-enabled Storm deployments. What parts of the StreamSight query model are supported by these three libraries is highlighted in Section 5.2.4.

5.2 New Functionality and Improvements

5.2.1 StreamSight Query Model

The StreamSight query model offers users the ability to create *insights*, denoted as high-level queries composed from raw monitoring metric streams. In a nutshell, an insight is a new data stream that comes from one (or more) processed stream(s). Query model operators introduce aggregations, compositions, and transformations on top of multiple monitoring metrics exposed by the input stream.

insight_name = **COMPUTE** *composite_expression*
[**WHEN** *filter_expression*]
[**EVERY** *time_interval*]
[**WITH** *optimizations*]

Figure 11: StreamSight Abstract Syntax

Figure 11 depicts the basic structure of an insight. The simplest insight structure includes only the *insight_name* followed by a **COMPUTE** statement. The **COMPUTE** statement requires a composite expression (e.g., an aggregation function on a stream). Furthermore, the model offers three optional primitives, namely, (i) **WHEN** primitive that filters a stream by applying specific predicates; (ii) **EVERY** primitive that alternates a purely streaming execution to a (micro-)batch query evaluation; and (iii) the **WITH** statement in which users define optimizations provided by the RAINBOW platform, such as sampling, prioritizing of the results, constraints enforcement, etc.

We note that during the first reporting period (coincides with the first RAINBOW release) the focus of this task was to extend the StreamSight query model to fit the needs of RAINBOW and fog deployments in general. The complete query model has been thoroughly described in D4.1. Nonetheless, the appendix contains a compact version of the query model in a formative EBNF depiction. The focus of the second reporting period was to implement the operators' part of the query model and create the relevant StreamSight compilers. In the next sections, we present examples of queries compiled using the StreamSight query model and a description of the journey of a query (and analytics job) from its inception by the user, to the point of execution by the RAINBOW-enabled distributed processing engine. We note that query examples presented are actual (or inspired from) queries created for the tests performed by the demonstrators for the purposes of D6.8.

5.2.2 Representative Queries

5.2.2.1 Typical SLOs

One of the most common approaches of orchestrators to provide stable SLOs is to perform scaling actions based on summarized analytics on target metrics. During the use case evaluation, demonstrators defined various SLOs based on target utilization metrics. For instance, Figure 12 depicts how RAINBOW's query language can describe the latter, with the target metric being the CPU utilization. Before continuing with the actual analytic query, we have to define the declaration of the stream source. Specifically, the lines 1-4 declare a metric stream that came from the RAINBOW's Storage Fabric, as well as a set of parameters that should be valid for Storage Fabric to return the data points. Rather than fetching all data points to the processing engine and filtering them there, we decided on the declaration of streams that filter out unnecessary data earlier (on the Storage Fabric) to minimize the data transfer between the storage and the processing layers.

After the definition of the stream, users can utilize it to perform the actual analysis. Lines Z-Y introduce the average (AVG) aggregation for one minute (60 SECONDS) of metric CPU percentage (cpu_ptc) from the metric-stream cpu_stream. The output of this computation

will generate every 10 seconds but will be disseminated to the Storage Fabric only if the result is higher or equal to 80% (WHEN >=60.0). We should note here that query language has a plethora of different statistics, including maximum, minimum, average, percentiles, count, and many more. Third RAINBOW use-case (UC3) utilized the previous query to increase its performance.

```
cpu_stream = STREAM FROM storageFabric (metricID="cpu_ptc",
                                        entityType="POD",
                                        podNamespace="{namespace}",
                                        podName="{podName}") ;

avg_cpu = COMPUTE (
            AVG("cpu_ptc" FROM (cpu_stream), 60 SECONDS)
            WHEN >= 60.0
          ) EVERY 10 SECONDS;
```

Figure 12 CPU-based SLO analytic query

5.2.2.2 Abnormal Values of a Metric Stream

Moreover, in a deployment where there are numerous physical components, like swarm of drones or robotic grippers, operators may need to evaluate unexpected values from environmental or infrastructure metrics, such as abnormal temperature. For instance, when the temperature of a drone's fan exceeds three standard deviations of the cumulative average of its historical temperature measurements, it would be considered abnormal, and the system need to land the drone and notify the technical department to evaluate the functionality of this drone. The insight "abnormal_temperature" (Figure 13) highlights how the previously mentioned example can be expressed via our query language. Since the language give us the opportunity to use multiple aggregation functions on the same stream, we are able to apply the average (AVG), the cumulative average (RUNNING_MEAN) and the cumulative standard deviation (RUNNING_STD) in a single line along with arithmetic operators.

```
temperature = STREAM FROM storageFabric (metricID="temperature");
abnormal_temperature = COMPUTE  AVG(temperature, 10 MINUTES) WHEN >
                                        RUNNING_MEAN(temperature) +
                                        3*RUNNING_STD(temperature)
```

Figure 13 Complex Analytic Query for Abnormal Values Detection

5.2.2.3 *Performance Evaluation of Custom Metrics*

In a Human-Robot Collaboration within the Industrial Ecosystems use case (UC1), operators need to take actions based on the processing rate of the incoming data. The software design of this use case utilizes message queues for injecting robots' and humans' data to separate processing services. To guarantee the performance of the deployed services, operators need to scale their services if any of the services (human or robot data processing service) are not capable of performing in time processing. For the latter reason, the user exposes the message queue statistics to the RAINBOW monitoring, and the RAINBOW Analytics Service can retrieve and generate performance insights based on them.

Figure 14 highlights two RAINBOW streaming analytic queries: (i) robot processing delay ratio and (ii) human processing delay ratio. Initially, lines 1-2 retrieve the respective streams from the RAINBOW Storage Fabric. Then, the system computes a normalized (100.0*) ratio between the maximum robot's generation data rate (robot_publish_rate) and processed data rate (robot_deliver_rate) for the last 10 seconds. The latter insight will be executed every 5 seconds. Similarly, the operator needs to compute an analogous ratio for humans (walkers). Since humans' data processing is more crucial for accident avoidance, the aggregation window and the interval are 5 and 1 seconds, respectively. In this example, UC3 queries highlight the flexibility of our query model to express more complex queries with mathematical operators, like +, -, *, /, etc.

```
walker_stream = STREAM FROM storageFabric (metricID="walker_%");
robot_stream = STREAM FROM storageFabric (metricID="robot_%");
robot_delay_ratio =  COMPUTE (
                        100.0 *
                        MAX("robot_publish_rate" FROM (robot_stream), 10 SECONDS) /
                        MAX("robot_deliver_rate" FROM (robot_stream), 10 SECONDS)
                     ) EVERY 5 SECONDS;
walker_delay_ratio = COMPUTE (
                        100.0 *
                        MAX("walker_publish_rate" FROM (walker_stream), 5 SECONDS) /
                        MAX("wolker_deliver_rate" FROM (walker_stream), 5 SECONDS)
                     ) EVERY 1 SECONDS;
```

Figure 14 Multiple Analytic Queries for Performance Evaluation

5.2.2.4 *Machinery Maintenance*

Following the performance of the previous example, the functionality of the existing machinery is also a crucial indicator of this UC. The outliers' detection in monitoring stream from robot joints may prevent mis-functionality on the whole production pipeline. Even if there are many implementations of outliers' detection in streaming

analytics, we selected to provide a set of distance-based outlier detection algorithms, as we introduced in [40]. A distance-based outlier detector considers as outliers all data points that have less than k-neighbours in a distance, denoted as R. Furthermore, the solution is extended with the explanatory algorithms introduced in [42]. Traditional distance-based outlier detection outputs only the data points that are deemed outliers under the specified user parameters (k,R). The explanatory techniques complement the detector's output by providing information on the outlier's vicinity, e.g., whether it is isolated or belongs to a cluster of varying size and density. This helps identify whether it is an extreme case or a problematic behaviour that might affect nearby (future) data points (i.e., if the outlier belongs to a small cluster) in order to prevent them.

In our implementation, we consider only single dimension outlier detection. That dimension is the current stream value as described in the query definition from the user. Because the outlier detector in a streaming setting needs a window and a sliding interval, we created a new window operator named OUTLIER_DETECTOR that comes with an extra WITH statement in which the user selects a specific algorithm and its parameters. Next, we demonstrate an example of PMCOD algorithm [41] that is able to detect outliers on the count joint movements and has as parameters a window of ten minutes, five seconds sliding interval, takes into consideration fifty neighbors in range equals to 0.5.

```
moves = STREAM FROM storageFabric (metricID="robot_joints_moves");
outliers = COMPUTE OUTLIER_DETECTION(moves, 10 MINUTES) EVERY 5 SECONDS
           WITH ALGORITHM pmcod AND K=50 AND R=0.5
```

Figure 15 Machinery Maintenance via Outlier Detection Query

5.2.2.5 Notifications based on events

Observing urban mobility (UC2) is a demanding task that needs on-time responses and notifications. Figure 16 introduces a representative query that an operator would like to evaluate on a deployment. Specifically, the operator exports the "dangerous" events from the application as custom metrics. When the service identifies a "dangerous" situation exposes a custom monitoring metric to the RAINBOW's monitoring stack. The value of the metric is decimal and defines its level of emerging. Through line 1, the StreamSight retrieves the metric stream from Storage Fabric. Then, lines 2-4 perform the execution of the actual query. The query adds all dangerous events that appeared in a node and returns the node ids with a score of 5 or more. When the analytic query outputs a result, the RAINBOW platform updates its dashboard, and, consequently, notifies the user.

```
event_stream = STREAM FROM storageFabric (metricID="dengerous_event");
notif = COMPUTE (
            SUM("dangerous_event" FROM (event_stream), 10 SECONDS) BY node_id WHEN >= 5.0
        ) EVERY 1 SECONDS;
```

Figure 16 Query for Event-based Notifications

5.2.2.6 Energy and Performance Optimizations

In an urban mobility scenario, there is a need for energy efficient processing and low-latency results. Furthermore, there is a possibility for battery powered devices while also the physical devices that are deployed in the urban environment needs power to be functional. To handle such regular situations in Fog environments, query language provides a wide range of optimizations providing them by "WITH" primitive. For instance, Figure 17 introduce some optimizations related to performance and energy consumption. Specifically, the user selects the scheduler to be the energy aware algorithm for the "running_cpu_util" insight. The scheduler will be invoked from the RAINBOW's schedulers repository at the query submission. The system will place the query's tasks and operators based on the scheduler's suggestions.

Moreover, in the second insight, the user selects to apply the summary statistic (average) on a sample of incoming data, sacrificing a portion of accuracy for on time results. However, the SAMPLE primitive does not guarantee nor predict the error of the generated results. For the latter, the query language provides the MAX_ERROR and CONFIDENCE primitives that tuning the sampling size to fulfil the desired maximum error and confidence.

The last insight of Figure 17 introduces a such query that computes the average GPU memory load with the maximum error not exceeding 5% and its confidence to be 95%.

```
cpu_util = STREAM FROM storageFabric (metricID="cpu_ptc");
running_cpu_util = COMPUTE RUNNING_AVG(cpu_util)
                   WITH SCHEDULER=RAINBOW_EnergyAware

network_io = STREAM FROM storageFabric (nodeID="{node-id}");
sum_network_io = COMPUTE AVG(network_io, 30 MINUTES) EVERY 15 SECONDS
                 WITH SAMPLE 0.2

gpu_memory = STREAM FROM storageFabric (metricID="gpu_memory");
gpu_memory_load = COMPUTE AVG(gpu_memory, 10 MINUTES)
                  WITH MAX_ERROR 0.05 AND CONFIDENCE 0.95
```

Figure 17 Multiple Queries with Energy- and Performance-aware Execution

5.2.3 The Journey of a query

Users can create streaming analytic jobs in two ways. One way is by designing their queries externally (away from RAINBOW Dashboard) and then submitting through the Analytics Enabler API. The API is on-line and available in Analytics Enabler Repository (Section 5.4). The second way is actually the more intuitive way, where users embrace the Analytics Perspective of the RAINBOW Dashboard to both create new analytic jobs and subsequently queries. Figure 18 introduces the interface to create a new analytic job, while Figure 19 presents the query interface. From this figure, one can immediately observe that queries can be constructed without even knowing the simple StreamSight language primitives as the interface provides users with a graphical guide of dropdown menus to guide the user in the query definition.

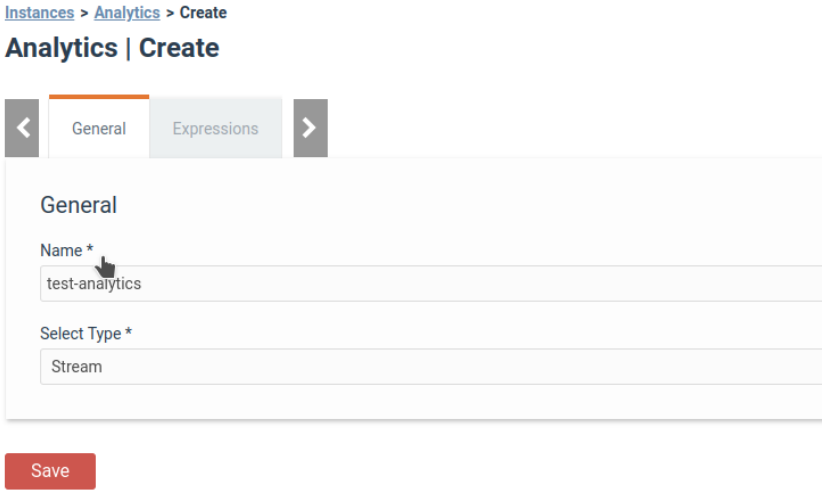


Figure 18: RAINBOW Dashboard - Create New Analytics Job

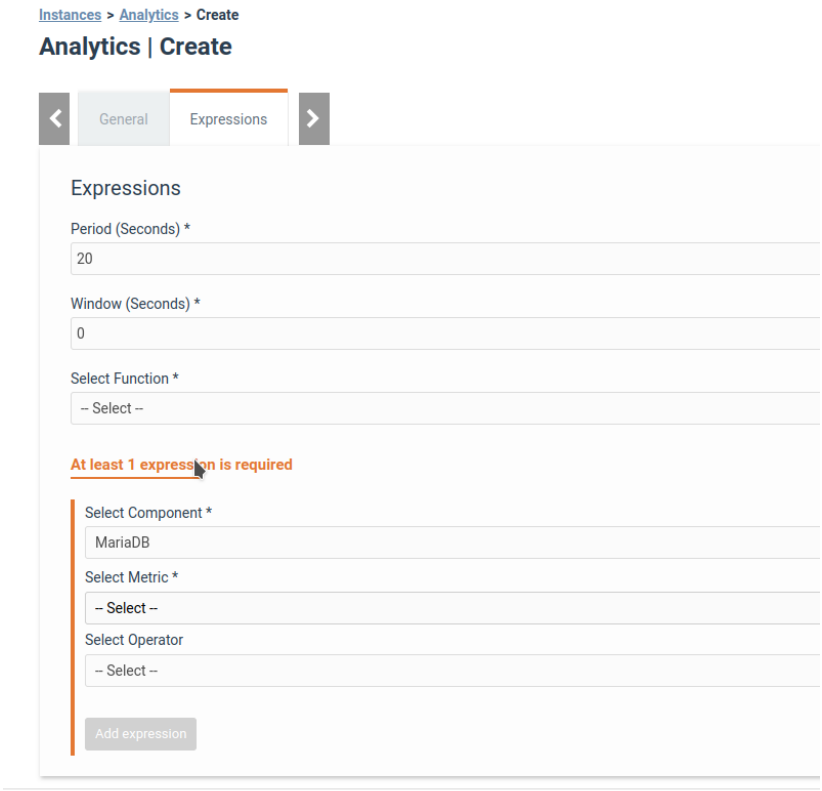


Figure 19: RAINBOW Dashboard - Create New Analytics Query (Assistant Interface)

When an analytics job is complete, users submit the job through the Dashboard by clicking on the submit job button. At this point, the Dashboard communicates with the Analytics Enabler via a restful HTTP request and submits the query (or queries). In the background, Analytics Enabler utilizes the StreamSight compiler. Specifically, the queries of the analytic job are translated into their AST representation and are also passed through the optimization process to derive efficient query logical plans for their execution in a fog environment.

Suppose a user submits the following query through the RAINBOW Dashboard:

```
insight1 = COMPUTE (
              AVG("SYSTEM_CPU_VISIBLETOTAL" FROM (stream2), 10 SECONDS)
           ) EVERY 5 SECONDS;
```

Figure 20 StreamSight Query Example

The query calculates the average available system's CPU of all cluster as it is reported the last 10 seconds, and the query will be evaluated every 5 seconds. When the user submits the previous query, the StreamSight Compiler generates an abstract syntax tree (AST). The AST representation generated by StreamSight is the following:

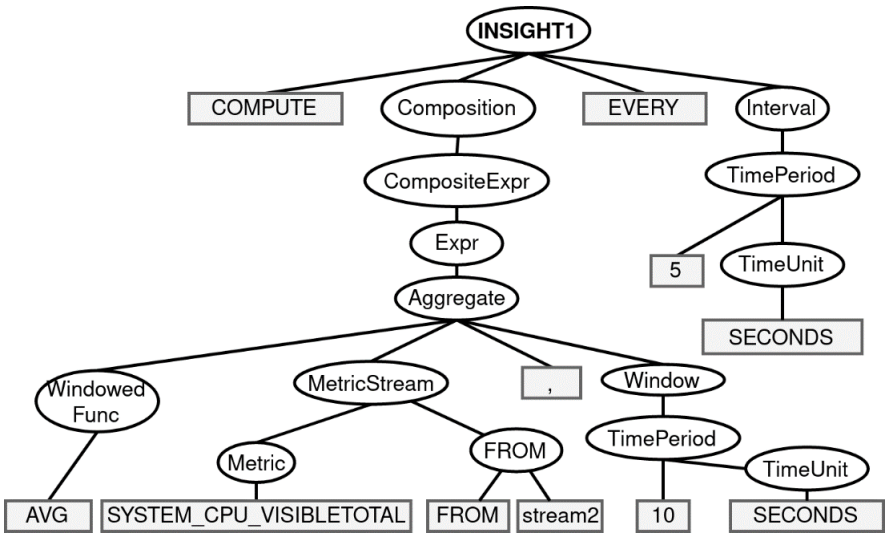


Figure 21 AST Representation of StreamSight Query

At this point, the system gathers all ASTs of the queries and optimizes them by merging (sub-)queries that perform the same operators on the same data streams. Since in this example we submitted only one query, there is no optimization opportunity. We should note that an example of the optimization process is presented in Section 5.3 (FR.AS.4).

Next, the generated AST of the query is ready to be compiled into an executable that will be deployed on the underlying distributed data processing engine. Hence, the StreamSight Compiler for RAINBOW-enabled Storm deployments is invoked and the AST is translated into Storm operators supporting RAINBOW optimizations. Specifically, the StreamSight translation process traverses the tree structure of the AST and automatically generates the respective Storm code. Followingly, we provide a prototype Apache Storm's code snippet that highlights the previously described query (aka the streaming average)

```
public class SumBolt extends BaseWindowedBolt {
    OutputCollector collector;
    String field;
    private Map<String, Object> conf;
    //Current sum
    private double sum = 0;

    public SumBolt(String field){
        this.field = field;
    }

    @Override
    public void prepare(Map<String, Object> topoConf, TopologyContext
collector) {
        this.collector = collector;
        this.conf = topoConf;
    }
@Override
public void execute(TupleWindow inputWindow)
    /*
     * The inputWindow gives a view of
     * (a) all the events in the window
     * (b) events that expired since last activation of the window
     * (c) events that newly arrived since last activation of the window
     */
    List<Tuple> tuplesInWindow = inputWindow.get();
    List<Tuple> newTuples = inputWindow.getNew();
    List<Tuple> expiredTuples = inputWindow.getExpired();

    List<Tuple> tuples = inputWindow.get();
    if((boolean)this.conf.getOrDefault("ustreamsight.logs",false)){
        System.out.println("All:"+tuplesInWindow.size());
        System.out.println("New:"+newTuples.size());
        System.out.println("Exp:"+expiredTuples.size());
    }
    // Optimized SUM
    for (Tuple tuple : newTuples) {
        if(tuple.contains(this.field)) {
            sum += (double) tuple.getValueByField(this.field);
        }
    }
    for (Tuple tuple : expiredTuples) {
        if(tuple.contains(this.field)) {
            sum -= (double) tuple.getValueByField(this.field);
        }
    }
    collector.emit(tuples,new Values(sum));

}

@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields(this.field));
}
```

Figure 22 Translated Code (Apache Storm) of the StreamSight Query

Then, submits the code to the Storm cluster, and Storm (based on updated schedulers) is responsible for placing the operators and tasks on the underlying cluster. Figure 22 depicts the previous query AST compiled into a Storm job. The picture is taken from the user interface of the Apache Storm and highlights the generated operators and the logical plan of the submitted query. Storm has already placed the operators/tasks on the Analytics Workers.

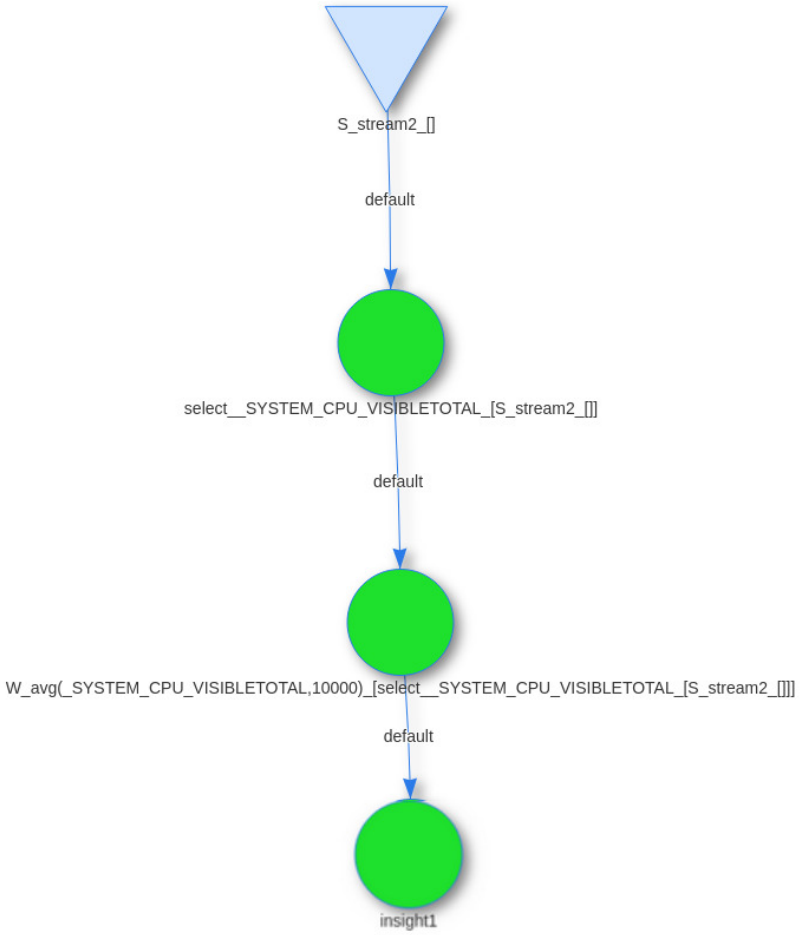


Figure 23 Apache Storm Execution Plan (DAG) of the StreamSight Query

Note: an example of a query AST compiled into both a Spark and Storm job, without any chances required to the AST, is presented in Section 5.3 (FR.AS.3)

5.2.4 StreamSight Compilers' Coverage of RAINBOW Query Model

The following table depicts the coverage of the RAINBOW query model by the different compiler implementations. From a first glance, all compilers support the basic query operators required to output streaming analytics. However, only the RAINBOW-enabled Storm compiler presents the functionality to provide the fog realm optimizations proposed by RAINBOW and introduced in D4.1.

Functionality	Spark Streaming	Storm	RAINBOW Storm
Descriptive statistics	X	X	X

Data filtering	X	X	X
Data transformations	X	X	X
Data grouping	X	X	X
Windowing	X	X	X
Multiple data streaming and joins		X	X
Sampling	X		X
Query prioritization	X		X
Outlier Detection			X
Operator placement directives			X
Job scheduling policy hints			X

Table 3: StreamSight Compilers' Coverage of RAINBOW Query Model

5.3 Requirements Fulfillment

This Section provides a report on the fulfillment of the requirements list of the Fog Analytics Service as documented in D4.1.

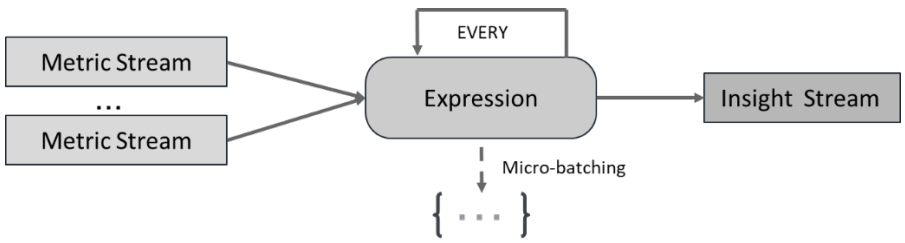
ID	FR.AS.1		
Title	**High-Level declarative query model for fog analytics**		
Requirement Description	The Fog Analytics Service must provide RAINBOW users with the ability to design analytics jobs composed of queries extracting analytic insights from monitoring data harvested by the deployment of their IoT applications over a fog environment. To achieve this a descriptive query model must be provided with the syntax, for query composition, understandable even from non-expert users and should not imply knowledge of a particular programming model or assume a specific distributed processing engine.		
Validation	Completed	Status	Fulfilled
The high-level declarative query model for streaming analytics in fog/edge environments has been described in D4.1, while an updated version of the query			

model in a normative EBNF form is available in the appendix of this deliverable. In brief, RAINBOW eases the programmability and compilation of streaming analytic jobs by providing high-level operators that can be composed together to apply aggregations, transformations, filters and groupings on data streamed from the fog nodes belonging to the deployment of an IoT application. These operators can also be "windowed", with windows either being event or time-based and time-based windows can be tumbling or sliding windows.

In brief, we described many examples in D4.1 Section 5.2.2 related with the expressivity of the query model. Moreover, the previous Section 5.2.2 of the current deliverable presents examples of many declarative analytic queries created and used by the RAINBOW demonstrators when tests were being performed for D6.8.

ID	FR.AS.2		
Title	**Streaming analytics (continuous queries)**		
Requirement Description	The Fog Analytics Service must support streaming analytic queries that will be evaluated in real-time. This requirement comes from both users, which would like to observe their applications and be aware of inefficiencies, and the RAINBOW scheduler that needs to analyze the monitored data as soon as possible.		
Validation	Completed	Status	Fulfilled

The RAINBOW query model supports streaming analytics and the application of continuous analytics jobs. For the query model, the input of a query are 1 or more monitoring metric streams where are the application of a chain of query operators, the output is a stream as well. This output is denoted as an *insight stream* and nothing segments an insight stream from an input stream and this has been designed intentionally so that an insight stream may be further given to other queries so more insights are generated.

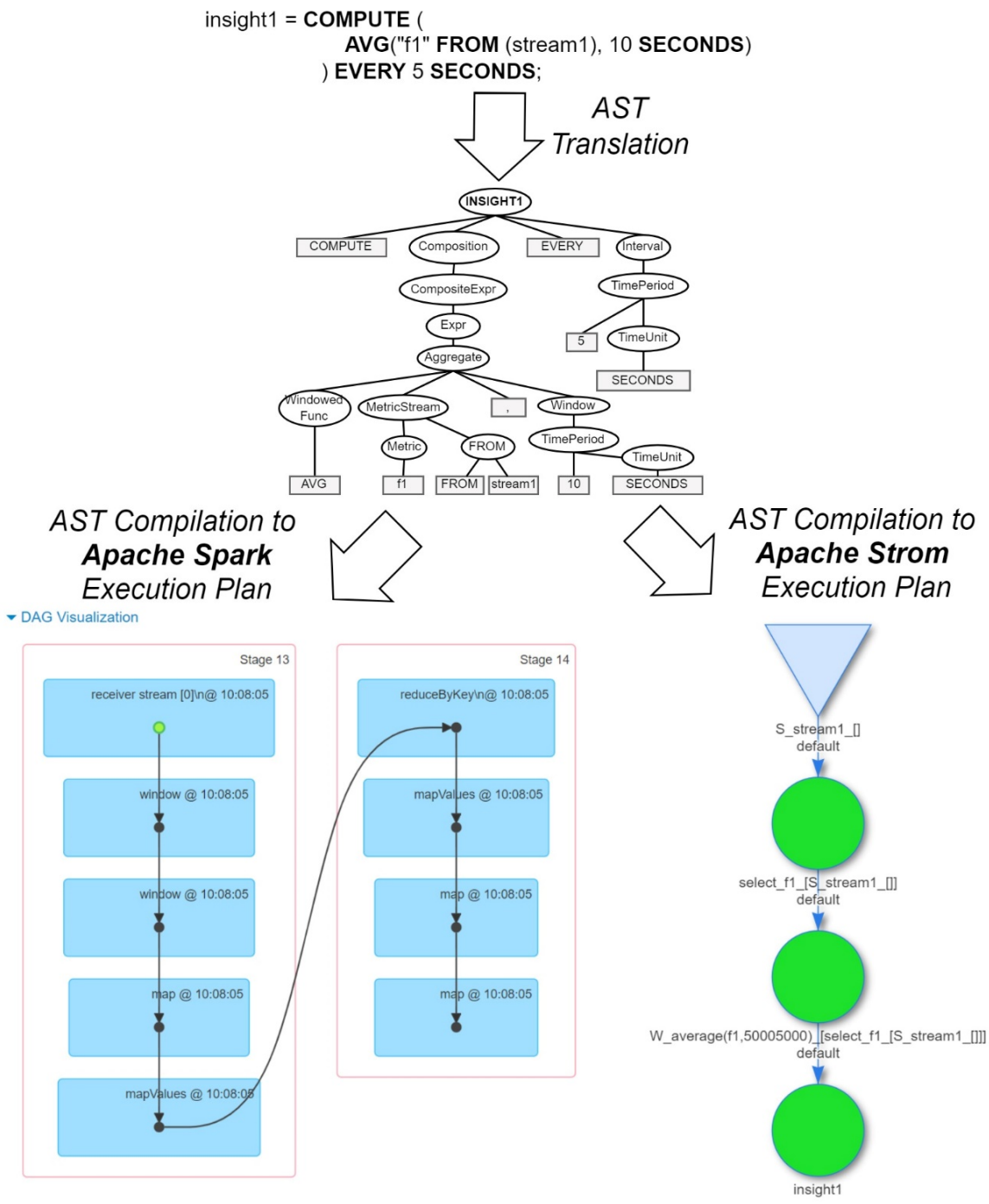


Moreover, the RAINBOW query model supports two streaming execution modes, a pure streaming mode where on the arrival of a new tuple the query operators are applied and a micro-batch mode where the incoming datapoints are windowed (as elaborated in FR.AS.1) and insights are computed per window. Examples of both modes have been introduced in FR.AS.1.

ID	FR.AS.3		
Title	**Query model decoupled from the underlying processing engine**		
Requirement Description	Fog Computing is a constantly evolving environment, so RAINBOW query model should not be coupled with a specific underlying engine. Specifically, the model should be easily translatable into different distributed engines with minimum effort.		
Validation	Completed	Status	Fulfilled

The StreamSight query model is designed intentionally to be decoupled from any programming primitives of both the underlying execution environment and the adopted distributed data processing engine. Specifically, analytic queries are translated automatically in the background to an abstract syntax tree (AST) and with this representation, StreamSight employs optimizations to derive a more efficient query logic plan (see FR.AS.4) and afterwards compile the continuous job artifacts that will be used for the execution. To compile the job, StreamSight currently features three compiler implementations for: (i) Spark Streaming; (ii) Storm (vanilla version); and (iii) RAINBOW-enabled Storm that features the use of the Analytics Enabler and the StreamSight job optimizations for the Schedulers referenced in Chapter 4. Nothing excludes the adoption of more streaming engines other than the effort required to implement the new compiler. Most importantly, queries are designed once and can be used not only multiple times but also on multiple engines, thus breaking the so called "analytics governance lock-in".

The following example illustrates an example of a StreamSight query, its mapping to an AST representation and the same query translated into Apache Spark and Storm.



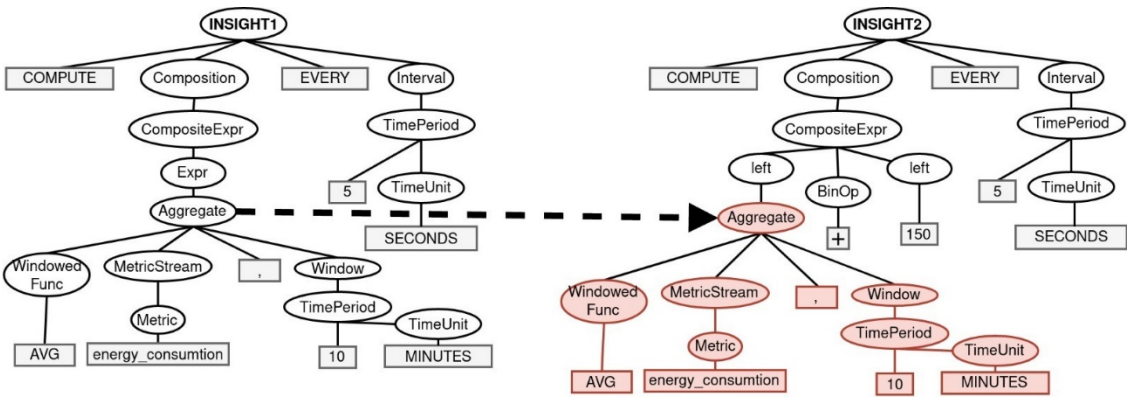
ID	FR.AS.4
Title	**Optimization of generated analytics job execution plan**

Requirement Description	The compilation of a set of queries should generate a highly optimized executable that minimizes the data transfer and computations in the execution time.		
Validation	Completed	Status	Fulfilled

Once a StreamSight query is automatically translated in the background into an AST representation (FR.AS.3), then the AST formalism is given to the StreamSight Query Optimizer in an attempt to derive a query logical plan that is more efficient. "Efficient" for a fog/edge environment is regarded a query that reduces both the computational effort applied and the amount of data needed to be disseminated when a data shuffling operator (i.e., reduce, group-by) is applied. Towards this, the Optimizer recognizes when query operators are being re-applied and intermediate results are being re-computed again and again, and when this is detected results are "pushed" through the logical plan to avoid the unnecessary processing effort and reducing the overall data that must be disseminated. In turn, for streaming engines such as Spark and Storm, queries of the same job that are always executed as independent processes ignoring the fact that parts of these queries could be exactly the same computation. StreamSight analyses the list of ASTs given for a job and detects such similarities, altering the ASTs to accept and share input from other queries.

The following example illustrates two exemplary queries part of the same continuous analytics job where the results of the first query are given as direct input to the hierarchical structure of the second query to avoid the re-computation of results that are already available. This possible thanks to the fact that for StreamSight output streams (insights) have nothing different from input streams, thus enabling insights to be attached as input to other queries (as previously mentioned in FR.AS.2).

energy_10min = **COMPUTE AVG**(energy_consumption, 10 **MINUTES**) **EVERY** 5 **SECONDS**
energy_plus_100_10min = **COMPUTE AVG**(energy_consumption, 10 **MINUTES**) **+** 100 **EVERY** 5 **SECONDS**



5.4 Documentation and Code Repository

As StreamSight is part of the RAINBOW Analytics Stack, the documentation can be found in the respective section of the RAINBOW documentation site:

https://rainbow-h2020.eu/docs/getting-started/rainbow-analytics/

The documentation includes a getting started guide, examples and a complete documentation of the service API calls and service interfaces.

The source code of the RAINBOW Analytics Stack is open-source and can be found in the RAINBOW source code repository:

https://gitlab.com/rainbow-project1/rainbow-analytics

6 Conclusion

In the present Deliverable, we highlighted a detailed analysis of all essential information towards the implementation details of the RAINBOW Data Management Services, as they are planned and created during the implementation of Work Package 4. Data Management Services are composed of three components, specifically: (i) Distributed Data Storage and Sharing Service, (ii) Distributed Data Processing Service, and (iii) Fog Analytics Service.

Starting from the state-of-the-art, we examined a wide range of related efforts. We highlighted their advantages and limitations. Thus, holding a clear outline of the state-of-the-art and RAINBOW's requirements, as described in D4.1, we underlined how we advance the state-of-the-art and fulfill the functional and non-functional requirements. Furthermore, we delivered implementation details of each component, offered examples of their use, and justified the fulfillment of the D4.1 requirements.

Specifically, Distributed Data Storage and Sharing Service is realized as an extension of the Apache Ignite in-memory data storage, and we extended the Apache Ignite with novel techniques for data sharing, balancing, and replication on Fog Computing infrastructures. We described in detail algorithmic and implementation aspects. Furthermore, Distributed Data Storage and Sharing Service realizes a unified storage fabric abstracting the underlying data retrieval processes and facilities in the interoperability among the RAINBOW services.

The next component of the Analytics Services is the Distributed Data Processing service that facilities in fog-enabled data processing. As we previously described, we utilized Apache Storm as the default processing engine of RAINBOW, extending it with a handful of novel Fog-aware scheduling algorithms, tailored to specific aspects, e.g., energy consumption, performance, etc. Furthermore, we evaluated the fulfillment of the Distributed Data Processing service requirements by providing a comparison between the baseline (Apache Storm vanilla implementation) and RAINBOW-enabled scheduling algorithms. To show the performance and utilization (e.g., energy) gains of our algorithms, we performed repeatable experiments on both real and emulated Fog environments.

Finally, the Fog Analytics Service is the last component of the RAINBOW analytics stack. It offers a high-level declarative query model abstracting the analytics description from the real-time monitoring data. To highlight the usability of the RAINBOW query model, we introduced a wide range of queries that RAINBOW use cases have applied to analyze their applications and as SLOs indicators. Even though the model is decoupled from the

underlying engine, it facilitates the declaration of fog-aware optimizations that RAINBOW schedulers can apply. Finally, we explained how the proposed query model and its compilation process satisfied the RAINBOW's requirements.

7 References

[1] R. Taft, I. Sharif, A. Matei, *et al.*, "CockroachDB: The Resilient Geo-Distributed SQL Database," 2020. doi: 10.1145/3318464.3386134.

[2] M. Serafini, E. Mansour, A. Aboulnaga, *et al.*, "Accordion: Elastic scalability for database systems supporting distributed transactions," 2014. doi: 10.14778/2732977.2732979.

[3] A. Lakshman and P. Malik, "Cassandra - A decentralized structured storage system," 2010. doi: 10.1145/1773912.1773922.

[4] A. Adya, D. Myers, J. Howell, *et al.*, "Slicer: Auto-sharding for datacenter applications," 2016.

[5] M. Serafini, R. Taft, A. J. Elmore, *et al.*, "Clay: Fine-grained adaptive partitioning for general database schemas," 2016. doi: 10.14778/3025111.3025125.

[6] R. Taft, E. Mansour, M. Serafini, *et al.*, "E-Store: Fine-grained elastic partitioning for distributed transaction processing systems," 2014. doi: 10.14778/2735508.2735514.

[7] Q. Pu, G. Ananthanarayanan, P. Bodik, *et al.*, "Low Latency Geo-distributed Data Analytics," *Comput. Commun. Rev.*, 2015, doi: 10.1145/2829988.2787505.

[8] Y. Huang, Y. Shi, Z. Zhong, *et al.*, "Yugong: Geo distributed data and job placement at scale," 2018. doi: 10.14778/3352063.3352132.

[9] S. Maiyya, I. Ahmad, D. Agrawal, and A. El Abbadi, "Samya: A geo-distributed data system for high contention aggregate data," 2021. doi: 10.1109/ICDE51399.2021.00128.

[10] Y. Lu, A. Shanbhag, A. Jindal, and S. Madden, "AdaptDB: Adaptive partitioning for distributed joins," 2016. doi: 10.14778/3055540.3055551.

[11] K. A. Kumar, A. Quamar, A. Deshpande, and S. Khuller, "SWORD: workload-aware data placement and replica selection for cloud data management systems," *VLDB J.*, 2014, doi: 10.1007/s00778-014-0362-1.

[12] N. Mehran, D. Kimovski, and R. Prodan, "A Two-Sided Matching Model for Data Stream Processing in the Cloud – Fog Continuum," *2021 IEEE/ACM 21st Int. Symp. Clust. Cloud Internet Comput.*, pp. 514–524, 2021.

[13] R. Gracia-Tinedo, M. Sanchez-Artigas, P. Garcia-Lopez, Y. Moatti, and F. Gluszak, "Lamda-Flow: Automatic Pushdown of Dataflow Operators Close to the Data," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 112–121. doi: 10.1109/CCGRID.2019.00022.

[14] Z. Hu, B. Li, and J. Luo, "Flutter: Scheduling tasks closer to data across geo-distributed datacenters," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, 2016, pp. 1–9. doi: 10.1109/INFOCOM.2016.7524469.

[15] A. Vulimiri, C. Curino, P. B. Godfrey, *et al.*, "WANalytics: Geo-Distributed Analytics for a Data Intensive World," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 1087–1092. doi: 10.1145/2723372.2735365.

[16] K. Hsieh, A. Harlap, N. Vijaykumar, *et al.*, "Gaia: Geo-Distributed Machine Learning Approaching {LAN} Speeds," in *14th {USENIX} Symposium on Networked Systems*

Design and Implementation ({NSDI} 17), 2017, pp. 629–647.

[17] A.-V. Michailidou, A. Gounaris, M. Symeonides, and D. Trihinas, "EQUALITY: Quality-aware intensive analytics on the edge," *Inf. Syst.*, vol. 105, p. 101953, 2022, doi: https://doi.org/10.1016/j.is.2021.101953.

[18] A. Toshniwal, S. Taneja, A. Shukla, *et al.*, "Storm@twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014, pp. 147–156. doi: 10.1145/2588555.2595641.

[19] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-Storm," *Proc. 16th Annu. Middlew. Conf.*, Nov. 2015, doi: 10.1145/2814576.2814808.

[20] L. Eskandari, J. Mair, Z. Huang, and D. Eyers, "T3-Scheduler: A topology and Traffic aware two-level Scheduler for stream processing systems in a heterogeneous cluster," *Futur. Gener. Comput. Syst.*, vol. 89, pp. 617–632, 2018, doi: https://doi.org/10.1016/j.future.2018.07.011.

[21] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in *2014 IEEE 34th International Conference on Distributed Computing Systems*, 2014, pp. 535–544.

[22] C. Li, J. Zhang, and Y. Luo, "Real-time scheduling based on optimized topology and communication traffic in distributed real-time computation platform of storm," *J. Netw. Comput. Appl.*, vol. 87, pp. 100–115, 2017, doi: https://doi.org/10.1016/j.jnca.2017.03.007.

[23] X. Liu and R. Buyya, "D-Storm: Dynamic Resource-Efficient Scheduling of Stream Processing Applications," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, 2017, pp. 485–492. doi: 10.1109/ICPADS.2017.00070.

[24] A. Karkouch, H. Mousannif, H. Al Moatassime, and T. Noel, "Data quality in internet of things: A state-of-the-art survey," *J. Netw. Comput. Appl.*, vol. 73, pp. 57–81, 2016, doi: https://doi.org/10.1016/j.jnca.2016.08.002.

[25] C. Xu, K. Wang, P. Li, *et al.*, "Renewable Energy-Aware Big Data Analytics in Geo-Distributed Data Centers with Reinforcement Learning," *IEEE Trans. Netw. Sci. Eng.*, vol. 7, no. 1, pp. 205–215, 2020, doi: 10.1109/TNSE.2018.2813333.

[26] M. Symeonides, D. Trihinas, Z. Georgiou, G. Pallis, and M. Dikaiakos, "Query-driven descriptive analytics for IoT and edge computing," 2019. doi: 10.1109/IC2E.2019.00-12.

[27] D. Abadi, A. Ailamaki, D. Andersen, *et al.*, "The Seattle Report on Database Research," *SIGMOD Rec.*, vol. 48, no. 4, pp. 44–53, Feb. 2020, doi: 10.1145/3385658.3385668.

[28] N. Jain, S. Mishra, A. Srinivasan, *et al.*, "Towards a streaming SQL standard," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1379–1390, 2008.

[29] D. Trihinas, "Interoperable Data Extraction and Analytics Queries over Blockchains," in *Transactions on Large-Scale Data- and Knowledge-Centered Systems XLV: Special Issue on Data Management and Knowledge Extraction in Digital Ecosystems*, A. Hameurlain, A. M. Tjoa, R. Chbeir, et al., Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2020, pp. 1–26.

[30] "Trident."

[31] M. Armbrust, R. S. Xin, C. Lian, *et al.*, "Spark SQL: Relational Data Processing in Spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on*

Management of Data, 2015, pp. 1383–1394. doi: 10.1145/2723372.2742797.
[32] M. Armbrust, T. Das, J. Torres, *et al.*, "Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 601–613. doi: 10.1145/3183713.3190664.
[33] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin, "Summingbird: A Framework for Integrating Batch and Online MapReduce Computations," *Proc. VLDB Endow.*, vol. 7, no. 13, pp. 1441–1451, Aug. 2014, doi: 10.14778/2733004.2733016.
[34] T. Akidau, R. Bradshaw, C. Chambers, *et al.*, "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing," *Proc. VLDB Endow.*, vol. 8, pp. 1792–1803, 2015.
[35] Apache, "Edgent." 2019.
[36] J. Ding and D. Fan, "Edge Computing for Terminal Query Based on IoT," in *2019 IEEE International Conference on Smart Internet of Things (SmartIoT)*, 2019, pp. 70–76. doi: 10.1109/SmartIoT.2019.00020.
[37] A. Gupta, R. Harrison, M. Canini, *et al.*, "Sonata: Query-Driven Streaming Network Telemetry," Aug. 2018.
[38] Z. Georgiou, M. Symeonides, D. Trihinas, G. Pallis, and M. Dikaiakos, "StreamSight: A Query-Driven Framework for Streaming Analytics in Edge Computing," 2018.
[39] M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis, and M. D. Dikaiakos, "Fogify: A Fog Computing Emulation Framework," 2020. doi: 10.1109/SEC50012.2020.00011.
[40] M. Symeonides, D. Trihinas, G. Pallis, M. D. Dikaiakos, C. Psomas and I. Krikidis, "5g-slicer: An emulator for mobile iot applications deployed over 5g network slices," 2022.
[41] T. Toliopoulos, C. Bellas, A. Gounaris, and A. Papadopoulos, "PROUD: PaRallel OUtlier Detection for Streams," 2020. doi: 10.1145/3318464.3384688.
[42] T. Toliopoulos and A. Gounaris "Explainable Distance-based Outlier Detection in Data Streams," 2022.

Appendix

In the appendix, we provide the extended Backus–Naur form (EBNF) of analytics query language (aka RAINBOW-enabled StreamSight model). Specifically, Figure 24 and Figure 25 present EBNF grammar file and lexer file, respectively.

```
parser grammar uStreamSightParser;
@header {
    package eu.rainbowh2020;
}
options{
    tokenVocab=uStreamSightLexer;
}

statements              :  statement (SEMI_COLON statement)* SEMI_COLON? EOF
                        ;

statement               : streamDefinition #streamDefinitionExpr
                        | insightDefinition #insightDefinitionExpr
                        ;

streamDefinition         : streamId COLON STREAM FROM streamProvider
                        ;

streamId                : ID
                        ;

streamProvider          : ID LPAR keyValueParams? RPAR
                        ;

keyValueParams          : keyValuePair ( COMMA keyValuePair)*
                        ;

keyValuePair            : ID EQUAL value
                        ;

value                    : STRING
                         | INTEGER
                         | FLOAT
                         ;

insightDefinition       : insightId EQUAL COMPUTE composition ;//(FROM membership)? ;

insightId               : ID
                        ;

composition             : intervalComposition #interExpr
                        | filteredComposition #filterExpr
                        | groupingComposition #groupExpr
                        | expression #basecaseExpr
                        ;

intervalComposition     : expression EVERY interval;

filteredComposition     : expression WHEN OPERATOR expression;

groupingComposition     : aggregate BY grouping (EVERY interval)?
                        ;

expression              : expression MULOP expression # mulopExpr
                        | expression ADDOP expression # addopExpr
                        | operation #opExpr
                        | FLOAT #DOUBLE
                        ;

operation               : aggregate    #aggregateExpr
```

```
                    | metricStream #metricStreamExpr
                    | cumulative #cumulativeExpr
                    ;

metric              : STRING
;

membership          : LPAR member (COMMA member)* RPAR;

member              : ID;

aggregate           : windowedFunction LPAR metricStream COMMA window RPAR #windowedAggrExpr
                    ;

cumulative          : function LPAR metricStream RPAR
                    ;

metricStream        : metric (FROM membership)?
;

function            :   ID;

windowedFunction    :   ID;

window              : timeperiod
                    ;
interval            : timeperiod
                    ;
grouping            : STRING
                    ;
timeperiod          : INTEGER TIME_PERIOD;

```

Figure 24 Antlr EBNF grammar file

```
lexer grammar uStreamSightLexer;
@header {
    package eu.rainbowh2020;
}

COMPUTE: 'COMPUTE' | 'compute';
STREAM: 'STREAM' | 'stream';
FROM: 'FROM' | 'from';
BY: 'BY' | 'by';
EVERY: 'EVERY' | 'every';
WHEN: 'WHEN' | 'when';

TIME_PERIOD             : 'MILLIS' |'ms'| 'SECONDS' | 's' | 'MINUTES' | 'm' | 'HOURS' | 'h';

ID : [a-zA-Z_]+[a-zA-Z_0-9]*;
STREAM_PROVIDER:    ID;

SEMI_COLON: ';' ;
COLON: ':' ;
EQUAL: '=';
LPAR : '(';
RPAR : ')';
LBR: '[';
RBR: ']';

COMMA: ',';

ADDOP                   : ('+' | '-');

MULOP                   : ('*' | '/' | '%');

OPERATOR                : ('>' | '<' | '==' | '!=' | '>=' | '<=');

STRING: '"' .*? '"' ;
INTEGER                 :  '-'? [0-9]+ ;
FLOAT                   : ('+' | '-') ?  [0-9] + ('.' [0-9] +)? ;

// Whitespace
WS  :  [ \t\r\n\u000C]+ -> skip ;

BlockComment
    : '/*' .*? '*/'
      { // System.out.println("BC> " + getText());
        skip();}
    ;


LineComment
    :   '#' ~('\n'|'\r')*
        { //System.out.println("LC> " + getText());
        skip();}
    ;
```

Figure 25 Antlr EBNF lexer file