A cloud-based Analytics-Platform for user-centric Internet of Things domains – Prototype and Performance Evaluation

Theo Zschörnig
Institute for Applied Informatics (InfAI), Leipzig University
zschoernig@infai.org

Jonah Windolph
Institute for Applied Informatics (InfAI), Leipzig University
windolph@infai.org

Robert Wehlitz
Institute for Applied Informatics (InfAI), Leipzig University
wehlitz@infai.org

Bogdan Franczyk
Leipzig University, Wrocław University of Economics
franczyk@wifa.uni-leipzig.de

Abstract

Data analytics have the potential to increase the value of data emitted from smart devices in user-centric Internet of Things environments, such as smart home, drastically. In order to allow businesses and end-consumers alike to tap into this potential, appropriate analytics architectures must be present. Current solutions in this field do not tackle all of the diverse challenges and requirements, which were identified in previous research. Specifically, personalized, extensible analytics solutions, which still offer the means to address big data problems are scarce. In this paper, we therefore present an architectural solution, which was specifically designed to address the named challenges. Furthermore, we offer insights into the prototypical implementation of the proposed concept as well as an evaluation of its performance against traditional big data architectures.

1. Introduction

The growing importance and adaption of the Internet of Things (IoT) in different domains is tightly coupled to the emergence of improved and new technologies. In this regard, it is estimated that the size of the market for enabler solutions in the European Union will grow to 15 billion Euros in 2025 [1]. This includes technological approaches to provide analytical capabilities to businesses, industry and end-consumers. Looking at the diverging characteristics of different IoT domains, analytics architectures need to be designed to handle a multitude of analytical problems and scenarios, which inherently differ from one another in terms of data volume, velocity, variety etc. Besides the need to employ big data technologies, the specifics of user-centric domains, such as smart home, which are characterized by their fast changing and highly individual application scenarios, present additional challenges and requirements for analytics architectures. For instance, during our research in this field, we found that there are no appropriate solutions to provide the needed flexibility in data processing orchestration and analytics scenario adaptation while still being able to handle big data problems under real-time requirements. Therefore, in this paper we present an architectural solution, which aims to overcome these shortcomings. Additionally, we evaluate our solution in terms of performance compared to a state of the art big data analytics system.

In the following, we describe the motivation behind conducting our research (Sect. 2). We further present challenges for and requirements of analytics architectures in user-centric IoT domains as well as already existing solutions and their shortcomings using the example of smart home (Sect. 3). Continuing, we introduce our architectural concept (Sect. 4) as well as its prototypical implementation (Sect. 5). In Section 6, we describe the results of two performance tests, which we conducted to evaluate our approach. Finally, the paper concludes with a summary of our findings and an outlook into additional research (Sect. 7).

2. Motivation

The provision of suitable analytics architectures for intelligent data analysis in the field of user-centric IoT domains, such as smart home, is associated with a multitude of challenges and requirements. Looking at IoT analytics architecture research in general, there are several studies and architectural proposals naming these. In this regard, we conducted an extensive literature review, following [2], to comprise an overview of them.

URI: https://hdl.handle.net/10125/64550
978-0-9981331-3-3

The key requirements for IoT analytics architectures are the ability to *handle big data problems* in terms of different 'v's such as velocity, variety and volume [3, 4, 5, 6, 7] in *real-time* [3, 4, 6, 7, 8, 13]. While the precise definition of real-time computing is rather subjective and varies depending on the use case [16], we found that in terms of IoT analytics it is closely linked to the value of the information to be derived from data processing. Therefore, analytics architectures need to enable users to process and analyze data in a timeframe, which is fitting for their respective application scenario. Looking at smart home environments, different use cases such as home security as well as disaster detection and prevention require at least low latency [17]. In this regard, *time criticalness* of analytics scenarios and the need for *low latency of data processing* in IoT environments was also frequently named in previous research [3, 4, 6, 7]. Furthermore, the *integration of the data from a multitude of sources* must be possible [3, 7, 18]. This includes the *integration of historical and real-time data* [6, 18]. Data transmission and processing must be *secure* [3, 4, 6] and the *privacy* of users concerning their data has to be considered at all times [3, 4, 8]. Moreover, the data processing capabilities must be *scalable* [9, 10, 11, 12] and *handle input from a multitude of sources* [3, 6, 7]. This *input data may arrive asynchronously*, e.g. because of connectivity issues [3, 4, 6]. Besides, analytics architectures also have to be *energy efficient* [9, 11] and address *high network usage* [9, 13] created by the increased number of data sources at the edge of the network. In addition, *all ingested data as well as the analytics results need to be stored* [15, 18] and *made available for other applications* [6, 18]. Furthermore, analytics architectures for IoT use cases need to offer the tools to *visualize data* [7]. Ultimately, the ability to *flexibly extend and modify the data processing capabilities* of an IoT analytics architecture is important [4, 6, 9]. In this regard, architectures have to enable *personalized analytics based on different user needs* [14, 15].

Various analytics architectures in different IoT domains operate in similar framework conditions. However, there are also important differences, which make existing solutions insufficiently suitable. One major difference is that analytics architectures for user-centric IoT domains need to handle two different types of analytics problems in terms of data set size and velocity. On one side, regular big data problems characterized by huge data volumes and high data velocity have to be addressed. For example, the training of machine learning algorithms for energy consumption prediction. On the other side, analytics scenarios also evolve around small data sets of only a few sources, e.g. temperature tracking of a single room in a smart home. Additionally, available data sources at different smart homes as well as expected insights into the data differ from user to user and may change over time. Therefore, analytics architectures need to enable its users to flexibly change analytics scenarios while still offering advanced data processing capabilities.

Current solutions in this field use lambda architectures and specialized big data technologies, such as Apache Spark or Apache Flink, for data processing. While these approaches excel in terms of velocity and volume of data processing [29], they lack the described flexibility because of their steep learning curves during the design and implementation of data analytics pipelines. In contrast, current systems, which are flexible to some extend, do not offer the processing capabilities to tackle big data problems, e.g. because they are not scalable. In order to bridge the gap between the needed flexibility in modeling and orchestrating analytics pipelines and the requirements of big data processing in real-time in user-centric IoT domains, we propose a new architectural approach in this paper. The main goal of the approach is to combine the ability to flexible design, apply and change analytics scenarios with the processing capabilities to tackle big data problems.

3. Related Research

While reviewing previous research, we found nine architectural approaches for analytics solutions in smart home environments. We evaluated all of these proposals against the requirements and challenges described in section 2. The results of this evaluation can be found in table 1.

Most of the reviewed approaches use IoT middleware solutions, such as NodeRED, for data ingestion and designing their data processing and storage capabilities around big data technologies and the lambda architecture concept (see [4, 19]). There are also approaches using only local data processing (see [20]) providing high-energy efficiency and low latency, but requiring extensive configuration, therefore limiting their scalability.

It is noteworthy, that none of the evaluated architectural proposals sufficiently met all requirements and furthermore could not address all the found challenges. Moreover, especially the requirement for flexible data processing extension as well as personalization of data analytics were only partially tackled by two solutions. In this regard, *Fortino et al.* propose an architecture for activity

recognition in smart home environments [21]. While they describe their solution as a platform- and software-as-a-service solution, the amount of configuration needed and individualization remains nebulous. *Hasan et al.* propose a cloud-based architecture, which exposes its analytics capabilities as services [4]. Although, these services may be reconfigured, it is along the boundaries of their functionalities, therefore only offering limited extension potential.

Regarding the need for personalized, flexible analytics, this highlights the urgency for an architectural approach, which offers scalability and the tools to handle big data as well as the named real-time requirements while still being flexible in terms of analytics capabilities extension and personalization of analytics scenarios. In the following, we present our architectural approach, which aims to address these issues.

4. Architectural Concept

In order to solve the challenges mentioned before we present the architectural concept as seen in figure 1. The central concept behind it is the kappa architecture. Derived from the more commonly utilized lambda architecture, the main goal behind it is to treat all data as streams therefore omitting the need for a dedicated batch layer for data processing [27].

In the proposed approach, so-called *analytics operators* do all data processing of data streams. An *analytics operator* describes a single data processing task. For example, the application of an arithmetic or statistical method to the input data stream. After successful processing, the results are written to an output data stream on the *streaming platform*. *Analytics operators* have inputs, outputs and configuration values. Inputs can be both, primitive and complex data types. The number of inputs is variable and depends on the data processing performed. For example, the addition of two values from one or two input data streams requires the definition of two inputs in the corresponding *analytics operator*. In addition, the outputs of an *analytics operator* are derived from the implemented method of data processing and may be primitive as well as complex data types. An *analytics operator* has at least one input and one output. In contrast, the definition of configuration values is optional. These can also have different data types and enable the context-dependent use of *analytics operators*. For example, in an *analytics operator* that enables the conversion of temperature values, a configuration value can be used to determine the temperature scale into which the input value is to be converted. At runtime, *analytics operators* are usually encapsulated programmatically or using virtualization technologies.

At design-time, various *analytics operators* are composed into *analytics flows*, which additionally describe the data flow in between *analytics operators*. Hence, *analytics flows* are designed by users to engage different analytics scenarios and provide a structured description of all the tasks and the data flow.

Since all data in the proposed architecture is handled as a stream, a *streaming platform*, including a *log data store* and a framework to enable data processing on the data, is a main component of the architecture. We suppose that data from IoT devices is ingested using IoT-middleware solutions and then pushed into the *log data store*. From there, *analytics operators* may access the streaming data, process and write it back to the *log data store*. The main advantage over lambda architectural approaches using the proposed concept is that changing requirements in analytics scenarios need to be only reflected at one data processing pipeline (job version n). In this regard, it is possible to either create a new *analytics pipeline* with changed parameters and configurations of the involved *analytics operators* (job version n + 1) or to use a different *analytics flow* altogether (job version m).

Access to all data and analytics results is possible via a *serving database*, which ingests data streams when requested by applications or users, allowing further aggregation of the data as well as the usage of appropriate database technologies for different types of data. In addition, applications may directly access the *log data store* to pull streaming data.

Table 1: Overview of existing smart home architectural proposals with regard to challenges to be solved and non-functional requirements.

Source	Intgr. data from different sources	Intgr. historic and real-time data	Flex. ext. of data processing	Share capabilities and data	(near) Real time analytics	Data visualization	Data storage	Big Data	Asynchronous data input	Distributed data input	Security	Privacy	Personalization of analytics	Scalable data processing	High network usage	Energy effiency	Time criticalness
[22]					✓			✓						✓			✓
[23]	✓						✓	✓		✓	✓	✓		✓			
[24]					✓											✓	✓
[21]	✓	✓	(✓)	✓	✓			✓		✓			(✓)	✓	✓		✓
[20]	✓				✓		✓	✓	✓	✓				✓	✓		✓
[25]	✓	✓			✓		✓	✓		✓				✓	✓		✓
[4]	✓	✓	(✓)	✓	✓	✓	✓	✓	✓	✓		✓					✓
[19]	✓	✓		✓	✓	✓	✓	✓		✓				✓			✓
[26]	✓	✓		✓	✓		✓	✓		✓				✓			✓

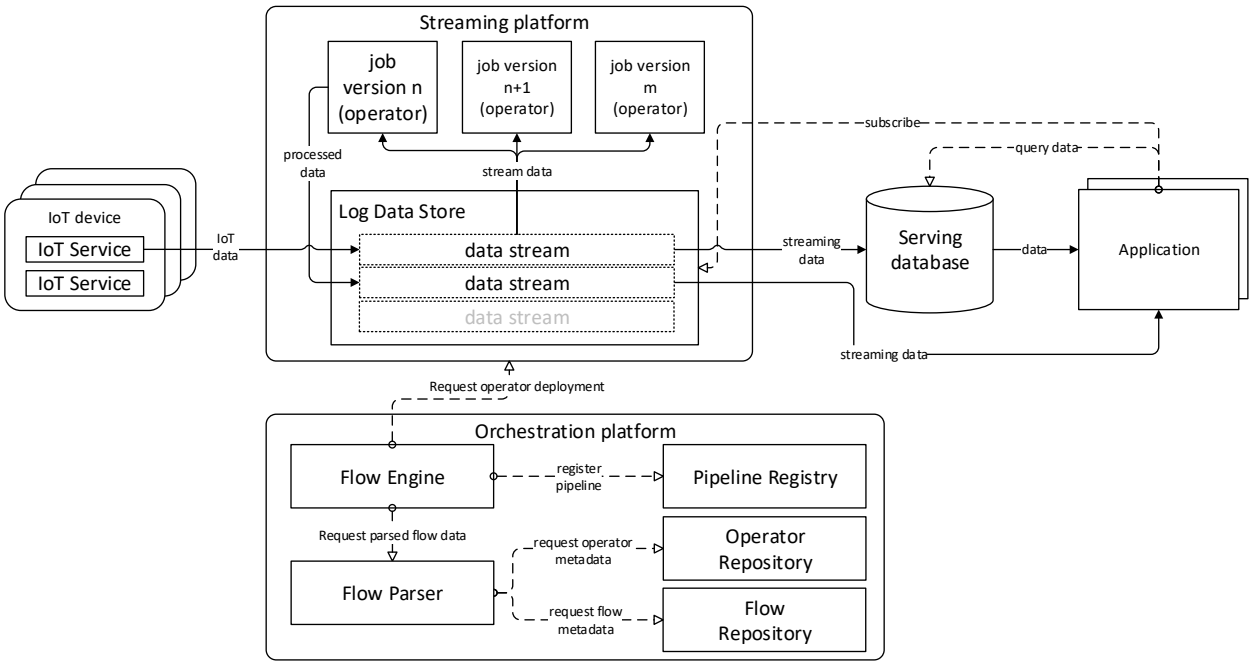

Figure 1: Overview of the proposed architectural approach.

In order to enable flexible *analytics operator* deployment and management, we further introduced an *orchestration platform*. The main purpose of the orchestration platform is to hide the complexity of the underlying big data technologies. This shifts the focus of the entire analytics platform to a modular approach concerning *analytics flow* design, promoting reusability of *analytics operators* across *analytics flows* and deployment environments. Additionally, we found that while it is possible to implement analytics scenarios manually in user-centric IoT domains, this is rather cumbersome and, especially regarding smart home platform providers, economically unwise.

By decoupling the orchestration of *analytics operators* from the actual *streaming platform*, the proposed architecture promotes its own reusability because it is independent from any specific *streaming platform*. The *orchestration platform* contains several components, which we describe in the following.

4.1. Flow engine

The *flow engine* controls the execution and orchestration of *analytics flows*. It uses an interface to start *analytics flows*, which are instantiated as *analytics pipelines*, and to stop the *analytics pipelines* that it started on the *streaming platform*. As soon as the *flow engine* receives a message to start an *analytics flow*, it accesses the interface of the *flow parser* and requests an execution list of *analytics operators* of the corresponding *analytics flow*. This list contains the *analytics operators* to be started as well as the mapping of the input and output data streams in-between them. Since *analytics flows* are not assigned to predefined data sources, the user does the assignment to source data streams, e.g. from IoT devices, dynamically.

Individual *analytics operators* are started by calling an interface of the underlying *streaming platform*. *Analytics flows* can be infinitely instantiated with different input data streams. The resulting *analytics pipelines* are registered in the *pipeline registry* and are removed by the *flow engine* after termination.

4.2. Flow parser

The *flow parser* provides the execution list of *analytics operators* to the *flow engine*. The main task of the component is to transform the saved representation *of analytics flows* from the *flow repository* into an execution list of *analytics operators*. This is done by applying predefined conversion rules. Subsequently, further information regarding *analytics operators*, e.g. metadata, is loaded from the *operator repository*. The decoupling of this component from the *flow engine* makes it possible to adjust the parsing logic of *analytics flow* from the *flow repository*. As a result, there is no need to commit to a specific *analytics flow* metamodel.

4.3. Flow repository

The *flow repository* serves as a storage location for *analytics flows*, thus enabling their reusability. *Analytics flows* are usually stored using graph-based metamodels. The creation and update of *analytics flows*, but also the access to them and their metadata is possible via an interface that is available for all components of the overall architecture. In this respect, it is possible to create and change *analytics flows* without using dedicated graph designer components.

4.4. Operator repository

The *operator repository* stores the metadata of all *analytics operators* that implement methods for processing data streams, therefore enabling their reusability across different *analytics flows*. The *operator repository* enables the creation, retrieval and deletion of metadata for the *analytics operators* via an interface. This metadata is used by other services to control and manage data processing logic.

4.5. Pipeline registry

The *pipeline registry* stores information regarding all active *analytics pipelines*. Via an interface, it is possible to register new and modify existing *analytics pipelines* as well as to delete them. In the presented architecture, *analytics pipelines* are registered and edited by the *flow engine*.

5. Prototype

During our research, we implemented the proposed architectural approach to provide a proof of concept and to allow for performance testing. Our solution specifically aims to provide more flexibility in terms of the adaption of changed requirements of analytics scenarios as well as reusability of *analytics operators*. In this regard, we found that current state of the art software engineering practices, namely the microservice paradigm together with container virtualization offer sufficient properties in order to tackle these goals.

Another priority of our research was to provide a reproducible architecture, which is why we choose scalable open source components, whenever possible. Subsequently, we choose Apache Kafka as the central *streaming platform*. Rather than being only a distributed publish-subscribe message queue, using its peripheral libraries, namely Kafka Streams, it offers data stream processing capabilities comparable to other state of the art solutions such as Apache Spark or Flink.

We deployed our prototype using Kubernetes as container orchestration and management platform, which is the de-facto industry standard in this regard. Kubernetes in conjunction with container-based Kafka Streams *analytics operators* offers start up times of only seconds [28], hence supporting the flexible and low-latency (re)deployment of *analytics pipelines*.

The prototypical implementations of all its components, introduced in section 4, also rely on the microservice paradigm and are described in the following.

5.1. Flow engine

The *flow engine* controls the instantiation of *analytics flows* and manages *analytics pipelines*. Implemented in the programming language Golang, it provides a set of management operations for *analytics pipelines* via a REST interface. The endpoints allow the instantiation of *analytics flows* and stopping of *analytics pipelines*. Because the *analytics flows* contain no information about the data sources that provide input for the first *analytics operators* of an analytics flow, this information needs to be communicated to the *flow engine* when starting an *analytics pipeline*. This is done by a POST request when calling the *flow engine*. Accordingly, the request must contain a JSON object with the required information about the data sources.

In order to start an *analytics operator*, the *flow engine* uses the API of the underlying container orchestration solution Kubernetes and starts new Docker containers, which are an instance of the corresponding Docker image of an *analytics operator*. These containers are configured via environment variables and allow for flexible, multiple instantiations of an *analytics operator*. The resulting *analytics pipeline* metadata is then stored in the *pipeline registry*. The termination of *analytics pipelines* is also controlled via the *flow engine*. In this case, it deletes all the Docker containers belonging to an *analytics pipeline* via the container orchestration solution API and deregisters it in the *pipeline registry*. The *flow engine* uses drivers to access APIs of different container orchestration solutions. These can be exchanged as needed. A combination of different solutions is also possible.

5.2. Analytics operator library

In order to interface seamlessly with the *flow engine*, we developed an *analytics operator library*

around Kafka Streams. Using this library, it is possible for data scientists to easily implement *analytics operators*. The library acts as a wrapper in this regard, hiding the complexity of Kafka Streams while still allowing for merging, filtering, etc. of data streams. Additionally, the library parses the configuration supplied by the *flow engine* to an *analytics operator* instance and accesses Kafka topics of data streams as defined by the user.

5.3. Flow parser

We implemented the *flow parser* using the programming language Golang. It offers functionalities that enable the data structure of the *analytics flows* to be converted into execution lists of *analytics operators*. It provides these methods via a REST interface.

The *flow parser* is able to retrieve all required data for a transformation from the *flow repository* based on the unique identifier of the *analytics flow*. It creates the execution list of *analytics operators* from the flow model data of the *analytics flow*.

Furthermore, the *flow parser* creates an array, which contains information about all input data sources of an *analytics operator*.

5.4. Flow repository

The *flow repository* stores metadata about *analytics flows*. In the implemented prototype, the *flow repository* is written using the Python scripting language. It provides a REST interface, which provides CRUD endpoints. The persistence of the data is guaranteed by a MongoDB, which saves all data records in JSON format. This includes the information necessary to generate the actual flow chart, containing nodes, edges and additional information.

5.5. Operator repository

The *operator repository* stores metadata about existing *analytics operators*. In the developed prototype, the repository is implemented using the Python scripting language. It provides a REST interface exposing CRUD endpoints. An *analytics operator* record is stored as a JSON document in a MongoDB. *Analytics operators* are instantiated in the developed prototype as Docker containers. An *analytics operator* data set consists of the reference to its corresponding Docker image, two lists (inputs and outputs) in which the inputs and outputs of an *analytics operator* are defined as well as additional metadata. As of now, about 20 analytics operators, offering different data manipulation and analytics capabilities, are available.

5.6. Pipeline registry

The *pipeline registry* is implemented using the programming language Golang. Different REST endpoints make it possible to register new *analytics pipelines*, retrieve information about them and delete them, if needed. In the implemented prototype, *analytics pipelines* are typically registered and managed by the *flow engine*. A MongoDB is used as the persistence layer. The retrieval of metadata of an *analytics pipeline* provides accurate information about the contained *analytics operators* and the data flows in-between them.

5.7. Frontend application

In order to ease usability of the orchestration platform, we implemented a frontend application written in Angular 6. Using this application, users can access all REST APIs of the involved services using input masks. This includes a graphical flow chart modeler, which was implemented using JointJS. Additionally, the creation of custom graphs and visualizations is possible.

6. Experimental Evaluation

In this section, we present a quantitative evaluation of the performance of the proposed architecture using detailed simulations based on real-world datasets. More specific, the feasibility of the proposed architecture to handle big data problems in real-time is evaluated. In this regard, we designed two experiments to compare the performance of *analytics operators* as an implementation of the kappa architecture against the de-facto standard implementation of a lambda architecture Apache Spark [29] in terms of operator and CPU core parallelism.

6.1. System Setup and Deployment

The proposed architecture, as well as the Spark cluster, were deployed at a private cloud service using Rancher version 2.2.3 as a frontend and Kubernetes version 1.13.5 as the engine for container orchestration. All Kubernetes cluster nodes were virtual machines running on hypervisors using the kernel-based virtual machine (KVM) module of Suse Linux Enterprise Server 12 SP4. The KVM

hypervisors provided an Intel XEON E5 CPU core, 512 GB RAM and SSD as well as Infiniband storage solutions. The actual Kubernetes cluster comprised 16 virtualized nodes, having 8 CPU kernels, 64 GB RAM and 256 GB SSD storage, each.

Apache Kafka was used as the central *log data store* and ran on version 2.0.1 in the cluster, being deployed as a replica set on Kubernetes. Apache Spark was deployed in the cluster as well using version 2.4.1. The Spark cluster used the Kubernetes scheduler for executor deployment and the Structured Stream API for stream processing. All components of the proposed architecture, as described in section 4 and 5, were deployed in the Kubernetes cluster as well.

6.2. Metrics

We measured *message throughput* as well as *adjusted message throughput*. In our evaluation, we defined *message throughput* as incoming messages per second. The basis of this calculation is the overall number of measurements in our test data set, divided by the execution time of an experiment in a given configuration.

During testing, we observed that scaling out Apache Spark executors and *analytics operators* from our architecture requires the partitioning of the Kafka topic, which holds the input data. This was necessary, so that measurements, which are logically linked, are placed on the same partition and therefore consumed in the right order and by the same *analytics operator*. Because of this, the resulting partitions were uneven in terms of data set size leading to distorted metrics measurements. The reason for this was that some of the scaled-out *analytics operators* stopped processing data before the entire data set had ended. Moreover, we witnessed an extended startup phase of the Spark cluster as compared to our proposed solution. Therefore, we introduced an additional metric, which we called *adjusted message throughput*. This metric ignores all data processing from the startup phase and during the period, when at least one partition has run out of data. Since it is plausible that analytics pipelines are running continuously in real-word scenarios, the omission of these two phases gives a more realistic insight into the message throughput.

6.3. Methodology

All experiments ran in changed configurations consecutively on the Kubernetes cluster to avoid side effects. In order to capture the metrics described before, we accessed the monitoring data of Apache Spark executors and the *analytics operators* of our solution. Runtime metrics of Spark executors were accessed using the Spark-native history server. In contrast, the runtime metrics of the *analytics operators* of the proposed architecture were exposed at the JMX port of the underlying Java Virtual Machine and written to an instance of InfluxDB using jmxtrans.

Additionally, CPU and RAM metrics were captured using the Rancher-native cluster monitoring, which allows for monitoring individual containers.

6.4. Experiments

We conducted two experiments to evaluate our proposed architecture in a real-world scenario. In this regard, we used real-world data, which was compiled in past research projects. The used data set contains 36,147,070 measurements of energy consumption data of about 1,000 smart meters over a timeframe of about 5 months. The entire data set was pushed into a Kafka topic to mitigate effects from slow data emission at the source of the data and directly access it from Kafka. The topic was partitioned with respect to the different experiments to allow *analytics operator*/executor parallelism.

6.4.1 Experiment 1: Outlier detection

Using averaging and standard deviation, the system had to detect outliers in the consumption data. The input was the entire data set and all data was grouped by meter identification.

In order to enable the experiment in the proposed architecture, we implemented an *analytics operator*, which was able to group the input data by meter identification, calculate the rolling average and standard deviation of the data and then tag outliers in the data. In Apache Spark, we implemented an appropriate processing logic.

This experiment allowed to utilize the entire test data set, thus creating meaningful runtimes of the experiment in both systems. Yet, it still has real world-relevance, as evidenced by similar experiments in [4] and [19].

6.4.2 Experiment 2: Timeslot

The system did the same tasks as in experiment 1. In addition, all outliers were grouped by the time period in which they occurred (grouping by hour).

This experiment was designed in order to simulate an *analytics pipeline* in our proposed architecture, which requires two *analytics operators*. The results of this experiment were supposed to

provide insights into how the performance of the proposed solution is impacted by multiple writes and reads to and from the log data store by *analytics operators*. In this regard, we used the *analytics operator* described in experiment 1 and added an additional *analytics operator*, which did the grouping by timeframe. It is noteworthy, that the second *analytics operator* only received the measurements, which were outliers. The Spark processing logic was extended according to changed requirements as well.

6.5. Experimental Results

Regarding the validity of the results, we conducted both experiments using different configurations in terms of *analytics operator*/executor scale and CPU core assignment to single *analytics operators*/executors. In addition, we conducted a pre-test to determine the optimal values for the maximum batch size (1,000,000 measurements) and shuffle partitions (32) in the deployed Apache Spark cluster. During our experiments, we changed the configurations of used *analytics operators*/executors and utilized CPU cores. For example, the configuration "1-1" stands for the usage of one *analytics operator*/executor with one CPU core assigned for the entire run of the experiment. In contrast, the configuration "8-4" means that eight *analytics operators*/executors were used and each of them got four CPU cores assigned.

Looking at the results of both experiments, as seen in figures 2 and 3, we observed, that the Apache Spark cluster (*spark*) achieved a higher *message throughput* than our proposed solution (*proposal*) in

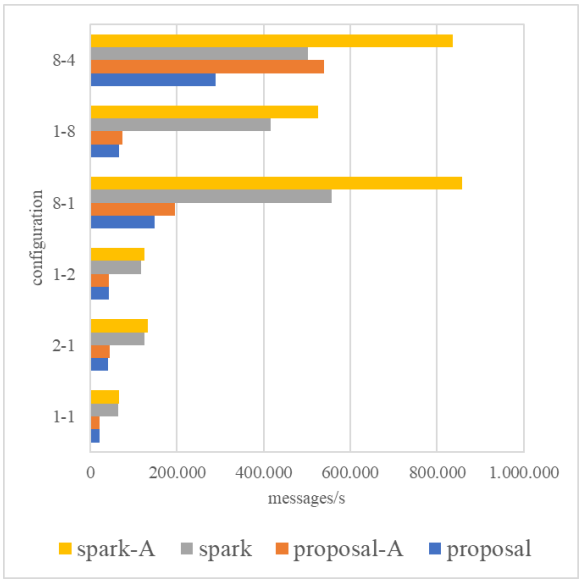

Figure 2: Results of experiment "Outlier detection".

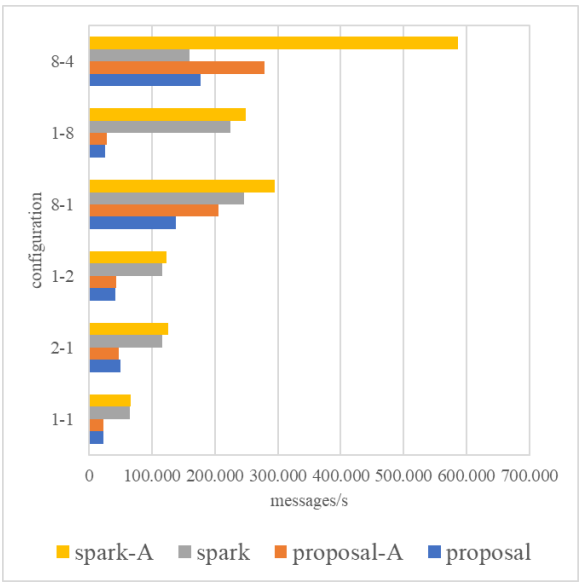

Figure 3: Results of experiment "Time slot".

every configuration but one (8-4 in experiment 2). When comparing *adjusted message throughput* (*spark-A* and *proposal-A*), the spark cluster achieved higher rates in every configuration of both experiments. Notably, the difference in *message throughput* between the proposed solution and the Spark cluster decreased in experiment 1 from a factor of 6.32 when comparing 8 parallel *analytics operators*/executors with only one CPU core assigned to 1.74 when 8 *analytics operators*/executors with 4 CPU cores each were used. Using this configuration, the proposed architecture was able to process around 540.000 messages per second compared to around 840.000 of the Spark cluster.

The results of experiment 2 suggest that the difference in *message throughput* of the proposed architecture compared to the Spark cluster is even lower than in experiment 1 (with the exception of configuration 1-8). Especially the results of configuration 8-4 are interesting, since our proposed solution achieved a higher *message throughput* of about 178.000 messages per second than the Spark cluster with about 160.000 messages per second. This indicates, that the lightweight approach of our proposed solution could indeed add flexibility to *analytics pipeline* deployment, when used in environments in which startup times of *analytics operators*/executors play a key role.

6.5. Discussion

The findings of the conducted experiments indicate that the performance of our approach, in

terms of *message throughput*, is not as good as specialized big data technologies used in lambda architectures, namely Apache Spark. Still, both systems provide good scalability with respect to *analytics operator*/executor parallelism. In contrast, the proposed solution seems to be better when scaling vertically, e.g. when offering more CPU cores to individual *analytics operators*. Other research in this field presents similar results and suggests that the difference could be even lower using other big data systems such as Apache Storm [29]. While this seems promising in reducing the discrepancy in *message throughput* between both systems, the added flexibility of our approach regarding *analytics pipeline* adaptation stems from the use of lightweight libraries, which is also reflected in the lower startup time of *analytics operators*. Moreover, the reusability of *analytics operators* and *flows* adds to the flexibility of the overall architecture.

In addition, with increasing complexity of the *analytics pipelines*, the difference in *message throughput* between both systems decreases. This indicates that the usage of specialized single task *analytics operators* is advantageous as compared to designing heavyweight all-purpose *analytics operators*. Further investigation is needed as to why *message throughput* of the proposed solution dropped at all in experiment 2. Since the first *analytics operator* did the same task as in experiment 1 and the second *analytics operator* had to process only thousands of messages, the difference in *message throughput* should have been marginal.

Finally, the experiments showed that the implemented prototype is able to handle the considered real-world data set and application.

7. Conclusions & Outlook

In this paper, we have presented an approach to address important requirements and challenges of analytics architectures in user-centric IoT domains, such as smart home. In this regard, we reviewed past research and compared existing architectural approaches against the identified challenges and requirements. Since none of the investigated solutions could sufficiently address key requirements, namely the ability to provide tools to handle real-time big data problems, while still being able to cater to small, flexible analytics scenarios, we presented our own architectural approach. This approach evolves around the kappa architecture concept and uses microservices to provide an orchestration engine for *analytics operator* deployment. Therefore, it tries to address the aforementioned problem and was prototypically implemented and evaluated in regards of its performance. The results of this evaluation suggest, that the proposed architecture is able to fill the gap between big data processing and flexibility in terms of small data analytics scenarios. Besides, this paper analyzes the performance of two state of the art data processing architectures, providing insights to practitioners and researchers alike.

Additional research in this field needs to assess the proposed architecture qualitatively in terms of its functional properties. In this regard, the proposed architecture has already been extended in [30] to address the found requirements and challenges of user-centric IoT domain analytics architectures, which were not investigated in this paper, e.g. privacy or high network usage.

Furthermore, in terms of usability, a comparison with similar solutions, which offer interactive data analytics capabilities, e.g. Apache Zeppelin, is needed.

8. Acknowledgements

The work presented in this paper is partly funded by the European Regional Development Fund (ERDF) and the Free State of Saxony (Sächsische Aufbaubank - SAB).

9. References

[1] https://www.statista.com/statistics/686198/iot-solutions-market-in-the-european-union-eu/, accessed 5-13-2019.

[2] Vom Brocke, J., A. Simons, K. Riemer, B. Niehaves, R. Plattfaut, and A. Cleven, "Standing on the Shoulders of Giants: Challenges and Recommendations of Literature Search in Information Systems Research", Communications of the Association for Information Systems, 37(1), 2015.

[3] Stolpe, M., "The Internet of Things: Opportunities and Challenges for Distributed Data Analysis", ACM SIGKDD Explorations Newsletter, 18(1), 2016, pp. 15–34.

[4] Hasan, T., P. Kikiras, A. Leonardi, H. Ziekow, and J. Daubert, "Cloud-based IoT Analytics for the Smart Grid: Experiences from a 3-year Pilot", in Proceedings of the 10th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities (TRIDENTCOM). 2015.

[5] Sun, Y., H. Schengong, A.J. Jara, and R. Bie, "Internet of Things and Big Data Analytics for Smart and Connected Communities", IEEE Access, 4, 2016, pp. 766–773.

[6] Cheng, B., S. Longo, F. Cirillo, M. Bauer, and E. Kovacs, "Building a Big Data Platform for Smart Cities: Experience and Lessons from Santander", in 2015 IEEE International Congress on Big Data (BigData Congress), New York, New York, USA. 2015.

[7] Rozik, A.S., A.S. Tolba, and M.A. El-Dosuky, "Design and Implementation of the Sense Egypt Platform for Real-Time Analysis of IoT Data Streams", Advances in Internet of Things, 06(04), 2016, pp. 65–91.

[8] Rehman, M.H.u., E. Ahmed, I. Yaqoob, I.A.T. Hashem, M. Imran, and S. Ahmad, "Big Data Analytics in Industrial IoT Using a Concentric Computing Model", IEEE Communications Magazine, 56(2), 2018, pp. 37–43.

[9] Schooler, E.M., D. Zage, J. Sedayao, H. Moustafa, A. Brown, and M. Ambrosin, "An Architectural Vision for a Data-Centric IoT: Rethinking Things, Trust and Clouds", in 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), Atlanta, GA, USA, 05.06.2017 - 08.06.2017. 2017. IEEE.

[10] Siow, E., T. Tiropanis, and W. Hall, "Analytics for the Internet of Things", ACM Computing Surveys, 51(4), 2018, pp. 1–36.

[11] Wich, M. and T. Kramer, "Enrichment of Smart Home Services by Integrating Social Network Services and Big Data Analytics", in Proceedings of the 49th Annual Hawaii International Conference on System Sciences (HICSS), Koloa, HI, USA. 2016.

[12] Yasumoto, K., H. Yamaguchi, and H. Shigeno, "Survey of Real-time Processing Technologies of IoT Data Streams", Journal of Information Processing, 24(2), 2016, pp. 195–202.

[13] Sharma, S.K. and X. Wang, "Live Data Analytics With Collaborative Edge and Cloud Processing in Wireless IoT Networks", IEEE Access, 5, 2017, pp. 4621–4635.

[14] Auger, A., E. Exposito, and E. Lochin, "Sensor observation streams within cloud-based IoT platforms: Challenges and directions", in 2017 20th Conference on Innovations in Clouds, Internet and Networks, Paris. 2017.

[15] Biswas, A.R. and R. Giaffreda, "IoT and cloud convergence: Opportunities and challenges", in 2014 IEEE World Forum on Internet of Things (WF-IoT), Seoul, Korea (South), 06.03.2014 - 08.03.2014. 2014. IEEE.

[16] Stankovic, J.A., "Misconceptions about real-time computing: a serious problem for next-generation systems", Computer, 21(10), 1988, pp. 10–19.

[17] Brush, A.J., M. Hazas, and J. Albrecht, "Smart Homes: Undeniable Reality or Always Just around the Corner?", IEEE Pervasive Computing, 17(1), 2018, pp. 82–86.

[18] Marjani, M., F. Nasaruddin, and A. Gani, "Big IoT Data Analytics: Architecture, Opportunities, and Open Research Challenges", IEEE Access, 5, 2017, pp. 5247–5261.

[19] Pham, L.M., "A Big Data Analytics Framework for IoT Applications in the Cloud", VNU Journal of Science: Computer Science and Communication Engineering, 31(2), 2016.

[20] Lin, Y.-H., "Novel smart home system architecture facilitated with distributed and embedded flexible edge analytics in demand-side management", International Transactions on Electrical Energy Systems, 17(7), 2019, e12014.

[21] Fortino, G., A. Giordano, A. Guerrieri, G. Spezzano, and A. Vinci, "A Data Analytics Schema for Activity Recognition in Smart Home Environments", in Ubiquitous Computing and Ambient Intelligence. Sensing, Processing, and Using Environmental Information. 2015.

[22] Bhole, M., K. Phull, A. Jose, and V. Lakkundi, "Delivering analytics services for smart homes", in 2015 IEEE Conference on Wireless Sensors (ICWiSe), Melaka, Malaysia. 2015.

[23] Chakravorty, A., T. Wlodarczyk, and C. Rong, "Privacy Preserving Data Analytics for Smart Homes", in 2013 IEEE Security and Privacy Workshops, San Francisco, CA. 2013.

[24] Constant, N., D. Borthakur, M. Abtahi, H. Dubey, and K. Mankodiya, Fog-Assisted wIoT: A Smart Fog Gateway for End-to-End Analytics in Wearable Internet of Things, 25.01.2017.

[25] Yassine, A., S. Singh, M.S. Hossain, and G. Muhammad, "IoT big data analytics for smart homes with fog and cloud computing", Future Generation Computer Systems, 91, 2019, pp. 563–573.

[26] Ta-Shma, P., A. Akbar, G. Gerson-Golan, G. Hadash, F. Carrez, and K. Moessner, "An Ingestion and Analytics Architecture for IoT Applied to Smart City Use Cases", IEEE Internet of Things Journal, 5(2), 2018, pp. 765–774.

[27] https://www.oreilly.com/ideas/questioning-the-lambda-architecture, accessed 1-5-2017.

[28] Medel, V., R. Tolosana-Calasanz, J.Á. Bañares, U. Arronategui, and O.F. Rana, "Characterising resource management performance in Kubernetes", Computers & Electrical Engineering, 68, 2018, pp. 286–297.

[29] Persico, V., A. Pescapé, A. Picariello, and G. Sperlí, "Benchmarking big data architectures for social networks data processing using public cloud platforms", Future Generation Computer Systems, 89, 2018, pp. 98–109.

[30] Zschörnig, T., R. Wehlitz, and B. Franczyk, "A Fog-enabled Smart Home Analytics Platform", in Proceedings of the 21st International Conference on Enterprise Information Systems, 21st International Conference on Enterprise Information Systems, Heraklion, Crete, Greece, 5/3/2019 - 5/5/2019.