SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2016; **00**:1–12
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.XXXX

Ahab: A Cloud-based Distributed Big Data Analytics Framework for the Internet of Things

Michael Vögler, Johannes M. Schleicher, Christian Inzinger, and Schahram Dustdar

Distributed Systems Group, TU Wien, Vienna, Austria

SUMMARY

Smart city applications generate large amounts of operational data during their execution, such as information from infrastructure monitoring, performance and health events from used toolsets, and application execution logs. These data streams contain vital information about the execution environment that can be used to fine-tune or optimize different layers of a smart city application infrastructure. Current approaches do not sufficiently address the efficient collection, processing, and storage of this information in the smart city domain. In this paper, we present `Ahab`, a generic, scalable, and fault-tolerant data processing framework based on the cloud that allows operators to perform online and offline analyses on gathered data to better understand and optimize the behavior of the available smart city infrastructure. `Ahab` is designed for easy integration of new data sources, provides an extensible API to perform custom analysis tasks, and a DSL to define adaptation rules based on analysis results. We demonstrate the feasibility of the proposed approach using an example application for autonomous intersection management in smart city environments. Our framework is able to autonomously optimize application deployment topologies by distributing processing load over available infrastructure resources when necessary based on both, online analysis of the current state of the environment, as well as patterns learned from historical data.
Copyright © 2016 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Big Data, Smart City, Internet of Things, Stream Processing, Lambda Architecture

1. INTRODUCTION

Smart city applications are large-scale distributed systems that react to and manipulate their physical environment using an underlying Internet of Things (IoT) infrastructure. The growing number of connected devices in current IoT infrastructures poses challenges not only for smart city applications that need to process and react to data produced by these devices, but especially for operators of the underlying smart city infrastructure who must ensure that deployed applications can optimally fulfill their requirements at all times. The inherently dynamic environment in which smart city applications are executed, creates a number of challenges. Applications must be able to quickly react to changes in business requirements and regulations, efficiently manage unreliable and expensive network links, and aim of maintaining optimal Quality of Service (QoS) in the face of infrastructure outages. While IoT infrastructures provide large amounts of performance and health data about the execution environment of smart city applications, this data is currently not effectively used to improve application execution. Moreover, application management policies for smart city applications that incorporate infrastructure data must be replicated for each new application. We argue that IoT application engineering and management can be significantly simplified by providing a dedicated component for processing, analyzing, and reacting to IoT infrastructure data.

*Correspondence to: voegler@dsg.tuwien.ac.at

In this paper, we introduce `Ahab`, a distributed, cloud-based stream processing framework that offers a unified way for operators to better understand and optimize the managed infrastructure in reaction to data from the underlying infrastructure resources, i.e., connected IoT devices, runtime edge infrastructure, as well as the overall application execution environment. The resulting management policies can be reused for managing various infrastructure components, and the gathered data can be used to guide infrastructure evolution.

The remainder of this paper is structured as follows: In Section 2 we motivate our work using a real-world scenario and outline specific problems as well as requirements. Section 3 introduces `Ahab`, an approach to address the aforementioned problems, along with a detailed evaluation in Section 4. Relevant related research is discussed in Section 5, followed by a conclusion and an outlook on future research in Section 6.

2. MOTIVATION

The success of the smart city paradigm and the advent of the IoT has led to significant convergence of traditional infrastructure and software systems. Todays cities are evolving into complex cyber-physical systems of systems integrating billions of connected and highly distributed devices. These devices are deployed in all vital areas of a city, from its infrastructure to the citizens, forming a complex network of sensors as well as computational power. Two core aspects in the evolution of smart cities are transportation and traffic systems, most notably, the advent of self-driving cars and the evolution of the city's infrastructure towards a cyber-physical adaptive system. Traffic systems already start to react to specific demands, depending on the time of day, weather conditions, and seasonal changes. Traffic lights as well as speed limits adapt to traffic conditions, road blocks, or accidents. However, this is only the first step in a natural evolution towards high density autonomous systems. Future cities will rely on adaptive roads that change according to their environment, as well as high volume autonomous car networks combined with multimodal public transport systems that efficiently transport citizens and goods in and between cities.

In order to enable these systems, it is crucial to provide sustainable computational capabilities. These systems not only need to react in an on-demand manner, they also must be able to adapt their processing capabilities accordingly in a fast and highly efficient manner. Given their complexity, paired with the inherent high availability requirements, it is not sufficient to rely only on the cloud for computational capacity. To ensure they work under all circumstances, it is necessary to utilize the computational network of the IoT infrastructure itself. This has a number of advantages ranging from economical and ecological benefits to the consideration of utilizing localities like saving bandwidth and the ability to sustain operations in disaster situations that could affect network uplinks. For demonstration purposes, we consider the case of a multimodal traffic management system that incorporates autonomous cars as well as means of public transport. The system itself needs to react depending on daily commuter patterns, weather conditions, and accidents. It is responsible for coordinating autonomous cars, traffic lights, and one-way streets, as well as to adapt public traffic interval times accordingly. We specifically take a look at one of the most demanding elements of such a system, the intersection control. In order to enable high volume autonomous car traffic and to utilize the benefits that come with the ability of cars communicating with each other, the system needs to be able to coordinate hundreds of cars per second, per intersection. This enables the high density traffic flow that is one of the major benefits of self driving cars in the smart city domain. To enable this element, it is vital to utilize the available edge infrastructure for processing to handle the highly varying demand that comes along with it.

3. APPROACH

To address the challenges discussed above, we present `Ahab`, a distributed, cloud-based stream processing framework allowing operators to manage and adapt IoT infrastructure components and running applications based on information extracted from data streams published by smart city

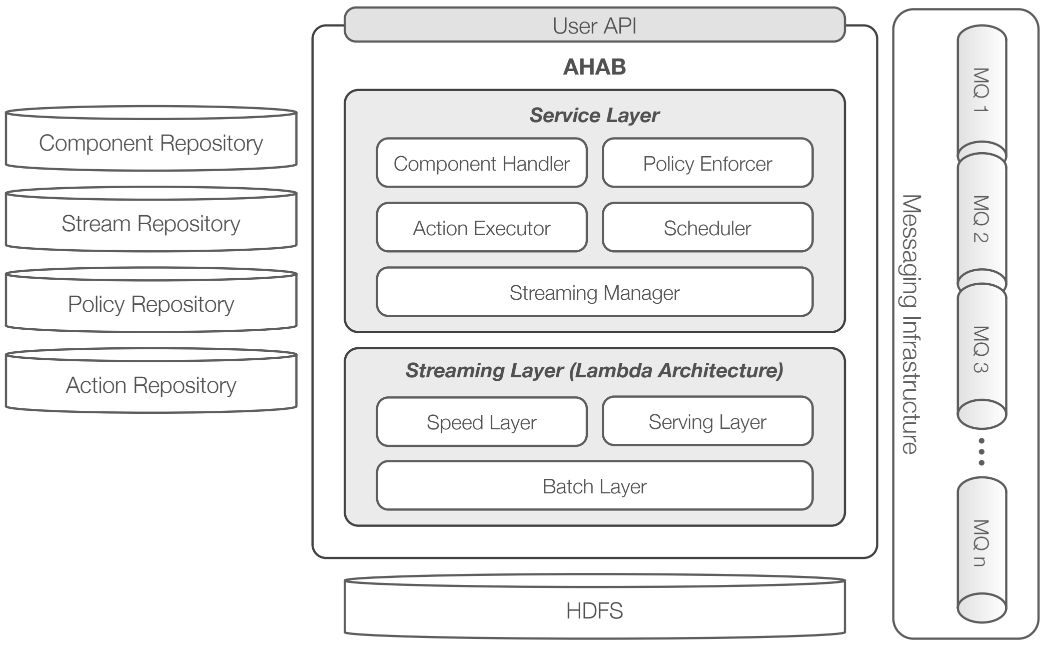

Figure 1. Ahab - Overview

infrastructure resources. The overall architecture of our approach is depicted in Figure 1 and consists of the following components: (i) the User API, (ii) Repositories, (iii) a Messaging Infrastructure, and (iv) `Ahab`, the stream processing framework. In the following, we discuss these components in more detail.

3.1. *User API*

Since our approach should be able to handle different data streams published from various resources, `Ahab` provides a `User API` allowing operators managing smart city infrastructures, to define which resources of the infrastructure provide data and how this data can be processed. The API allows operators to register IoT components and respective management policies, which allows `Ahab` to adapt registered components based on information extracted by processing and analyzing incoming data streams. To provide an extensible and easy way to describe system components in our approach, we integrate and extend MONINA [1], a language for specifying monitoring and adaptation policies. Listing 1 illustrates a simplified example of a typical IoT infrastructure definition.

```
stream Response {
  processing_time_ms : Integer}

action Scale {
  amount : Long}

component Application {
  name : String
  endpoint {
    at "/application"
    stream Response
    action Scale}}

stream AverageResponse {
  processing_time_ms : Integer}

stream ProcessedResponse {
  from Application
  stream Response as r
  create AverageResponse(
    avg(r.processing_time_ms))
  window 10 seconds}

policy ScaleUp {
  from AverageResponse as a
  when a.processing_time_ms > 2000
  execute Application
          .Scale(5)}
```

Listing 1: Sample IoT infrastructure definition

In `Ahab` we introduce the `stream` concept to refer to both, incoming data streams provided by infrastructure components, as well as processing streams created by `Ahab`. To avoid limiting the structure of incoming data streams, our approach allows registering custom processing streams, which take care of transformation and further processing. Next, a `component` defines infrastructure resources that either just produce data streams (e.g., connected edge devices) or need to react on information extracted from streams. To allow components to react on extracted information, components can define `action` attributes. Actions, together with `policy` directives, which define adaptation criteria that need to be met, allow `Ahab` to autonomously manage and adapt system components.

3.2. Repositories

To store registered infrastructure components, used streams, management policies, and actions, `Ahab` uses several repositories.

Component Repository Every component that represents a resource in the IoT infrastructure (e.g., connected IoT devices) and provides data that can be used to analyze the overall behavior of the system, gets first registered via the User API and is stored in the `Component Repository`. Based on this information `Ahab` can correlate incoming data streams with registered components and further process them accordingly. In addition to components that produce data, operators can also register components (e.g., IoT applications) that need to be managed and adapted by `Ahab` in order to react to incoming or processed data streams.

Stream Repository To keep track of the various incoming data and processing streams used by `Ahab`, our approach uses a `Stream Repository`. According to the information stored in the repository, operators can easily add additional streams or management policies that facilitate and further analyze or process data produced from underlying streams.

Policy Repository Since `Ahab` not only processes and analyzes data streams, but also adapts registered components based on policies, operators can register these policies in the `Policy Repository`. This approach allows `Ahab` to keep track of all registered policies and furthermore allows operators to create generic policies that are applicable for more than one component.

Action Repository In order to allow `Ahab` to manage and adapt registered components, operators need to define and register adaptation actions, which are then stored in the `Action Repository`. Once `Ahab` detects that a certain policy is violated or can not be met anymore, it tries to find and execute corresponding adaptation actions.

3.3. Messaging Infrastructure

For handling the multitude of data streams, `Ahab` provides a distributed messaging fabric. This approach minimizes unnecessary network traffic, compared to a centralized message bus, and allows components of the system to freely choose the most suitable (e.g., closest) connection point. For transmitting and consuming data, `Ahab` uses a publish/subscribe mechanism, where producing components publish data in the messaging infrastructure to a specific routing key. This routing key contains the name of the component and the type of data stream it is publishing. For `Ahab` we distinguish different types of data streams: incoming data streams, such as access logs or metrics from infrastructure components, and processing data streams, that process and analyze other data streams to extract more actionable information. Consuming components of `Ahab`, which are mostly processing streams and service layer components, consume data by subscribing to the appropriate routing key. For example, processing streams that are only interested in logs published by a specific component define both the specific component name and the type of stream they are interested in, as routing key. On the other hand, more generic streams that can be used for several components only specify the type of data stream as routing key. This approach provides a flexible mechanism that can be easily extended and allows operators of `Ahab` to choose the suitable data granularity.

3.4. Ahab Architecture

The enabling stream processing approach is a cloud-based framework depicted in the center in Figure 1. `Ahab` is separated in two layers. On the bottom, the `Streaming Layer` handles and processes data streams, and is implemented as a Lambda Architecture[†]. On top, the `Service Layer` is handling the underlying streaming layer and takes care of analyzing, managing, and adapting registered components. In the following, we discuss these two layers in more detail, present the main components of `Ahab`, and describe how they interact with each other.

3.4.1. Streaming Layer To handle massive quantities of real-time data and also provide batch processing of, e.g., historical data, the streaming layer of `Ahab` is implemented as a Lambda Architecture. In general, the Lambda Architecture is a generic, scalable, and robust data processing system, specifically designed to serve massive workloads and a wide range of use cases. To balance latency, fault-tolerance, and throughput, the Lambda Architecture combines both batch processing and stream processing capabilities. Lambda Architectures use batch processing to provide accurate and comprehensive views of batch data, while concurrently running stream processing produces views on real-time data.

The streaming layer of `Ahab` contains the following sub-layers, as proposed by the Lambda Architecture: a `batch layer`, a `speed layer` and a `serving layer`. First, all data that enters `Ahab` via the messaging system is dispatched to both the batch and speed layer for further processing. The batch layer stores the data in HDFS[‡], grouping data originating from the same source. The batch layer then precomputes views by using all available data in HDFS. These views are subsequently used by the serving layer. The serving layer indexes the views produced by the batch layer in order to provide ad-hoc querying. To compensate for the high latency of updates to the serving layer, the speed layer only deals with recent data. The speed layer provides online processing of data streams, which means that results are available almost immediately after data is received by `Ahab`. While these results might not be as accurate or complete as views generated by the batch layer, they can be updated as soon as the results of the batch layer for the same data are available. By combining batch and real-time views in the `serving layer`, `Ahab` can serve massive quantities of incoming data and also cover a broad area of use cases. This approach allows for optimizing components either based on real-time information (e.g., to handle critical situations), or based on batches of historical information, or using a combination of both.

As underlying stream processing engine we employ Apache Spark[§] respectively Apache Spark Streaming running in a Hadoop cluster. Spark provides a unified processing engine and programming model that natively supports processing both stream and batch workloads. Spark discretizes streaming data into micro-batches [2], packages them into small tasks, and assigns them to resources (workers) for processing. In order to deal with large workloads, Spark provides an optimized load balancing and resource usage mechanism, which allows utilizing workers of the cluster more efficiently by dynamically assigning tasks to workers based on locality of data and available resources. In addition, by using small discretized tasks that can run on any resource without affecting correctness, Spark provides fast failure discovery.

3.4.2. Service Layer On top of the streaming layer, the `Service Layer` manages the underlying streaming layer and registered components based on actionable information produced by streams, policies, and adaptation actions. The design of the service layer follows the microservice architecture approach [3], which enables building a scalable, flexible, and evolvable framework. Especially the flexible management and scaling of services is important for `Ahab` in order to enable efficient and fast adaptation of components. In the following we introduce the main components of the service layer.

[†] http://lambda-architecture.net
[‡] https://hadoop.apache.org
[§] http://spark.apache.org

Component Handler To process incoming data streams produced by infrastructure components or to optimize managed infrastructure components with `Ahab`, operators have to register them using the User API, by providing the following information: (i) name of the component, (ii) an endpoint that defines where the component is reachable, (iii) data streams that get published, and (iv) actions that can be executed to adapt the component. This information is then transformed by the `Component Handler` using the MONINA language and stored in the component repository.

Streaming Manager Since `Ahab` is not only handling incoming data streams produced by various infrastructure resources (i.e., components), but also allows operators defining custom streams that process and analyze other data streams managed by `Ahab`, the `Streaming Manager` is responsible for efficiently managing this large number of streams.

When new components are registered using the User API, the manager generates a new key for each provided stream in the component description. This key is then registered in the stream repository and can be used by other streams to consume this stream via the messaging infrastructure. To register custom streams, `Ahab` provides two mechanisms for operators. First, simple streams can be defined using the MONINA language and registered via the User API. This definition is then translated by the streaming manager into an actual streaming application that implements the behavior of the stream. Second, `Ahab` provides a streaming library that allows operators to develop custom streaming applications that implement more complex streams. The developed streaming application is then registered using the User API.

Next, the streaming manager analyzes the streaming application, extracts which streams are produced by the respective application, generates corresponding keys and registers them in the stream repository. Finally, the manager invokes the scheduler to submit the new streaming application to the streaming layer.

Scheduler Since custom streams need to be executed in `Ahab`'s streaming layer, the `Scheduler` is responsible for submitting custom processing streams represented as streaming applications to this layer. When invoked, the scheduler first analyzes which streams are consumed and calculates the needed processing power (e.g., number of cores) for executing this application. Based on the used streams the scheduler decides whether the application should be executed in either the batch or speed layer of the streaming layer. Next, the application is packaged with the required streaming library and additional execution parameters, and finally submitted to the streaming layer where it is executed in the underlying streaming engine.

Policy Enforcer To manage registered components, `Ahab` uses a policy based approach, where operators can define management policies using the MONINA language and register them via the User API. Furthermore, operators can register specific adaptation actions that should be executed when a policy violation is detected. These policies are then stored in the policy repository and forwarded to the `Policy Enforcer`. For each policy registered in the repository, the policy enforcer creates a policy stream that specifically implements the defined policy. This policy stream is then executed using the scheduler and triggers a notification once it detects that the policy can no longer be met.

Action Executor In order to handle notifications triggered by the policy streams, the `Action Executor` is used. The action executor continuously listens for incoming notifications. When it receives a notification, it analyzes which policies are violated and checks the repositories to find suitable adaptation actions for each affected component. Then, the list of found actions is executed to adapt the component and therefore guarantee that the management policy is met again.

4. EVALUATION

To evaluate our approach we implemented a smart city demo application based on the scenario identified in Section 2. Next, we created a test setup in the cloud using CoreOS[¶] to virtualize edge devices of our IoT infrastructure as Docker[‖] containers.

In the remainder of this section we give an overview of the developed smart city demo application, discuss the concrete evaluation setup, present different evaluation scenarios, and analyze the gathered results.

4.1. Smart City Demo Application

In order to evaluate our approach we developed a demo application that implements the concept of Autonomous Intersection Management (AIM) [4], which presents an essential element in enabling autonomous cars in a smart city environment. To fully utilize the autonomous capabilities of self driving cars to allow the high volume traffic in our presented scenario it is essential to enable intelligent resource management. One area where the demand for such an intelligent mechanism is especially demanding, are road intersections. Currently, cities use traffic lights and dozens of signs to assist human drivers to safely pass road intersections. However, with the upcoming advent of autonomous cars the AIM project shows that it is vital to adapt modern-day intersection management in future smart cities allowing autonomous vehicles to interact with intelligent traffic control systems to enable more efficient and effective traffic management. Considering the huge numbers of cars in our scenario, we also need a way that allows smart city operators to scale such intelligent control systems by using any kind of available processing power of a smart city infrastructure (e.g., cloud resources or edge infrastructure).

To demonstrate this aspect, we developed a simple traffic control application that handles incoming requests sent from autonomous cars. These incoming requests get processed by the application to determine if a car's intended path is safe to use or not. To deal with the load generated by the autonomous cars, the application can offload the computation to available resources, which allows scaling across infrastructure boundaries. Furthermore, to analyze the application's performance, the application publishes metrics such as request load and response time.

For the implementation of our demo application we used Spring Boot[**] and separated the application in two components. A possibly replicated processing component that provides the calculation logic and is deployed as a microservice running on resources in the edge. On top, a cloud-based platform component that receives requests by autonomous cars and forwards them to the underlying processing components.

4.2. Setup

For the evaluation of our framework, we created an IoT testbed in our private OpenStack[††] cloud. We reuse a Docker image from our recent work [5] to virtualize and mimic a physical edge device in our cloud. To run several of these virtualized devices, we use a CoreOS cluster of 5 virtual machines, where each VM is based on CoreOS 607.0.0 and uses the m1.medium flavor (3750MB RAM, 2 VCPUs and 40GB disk space). To simulate a medium scale IoT infrastructure we use 100 virtual edge devices evenly distributed among the CoreOS cluster.

As our messaging infrastructure we use a RabbitMQ[‡‡] cluster consisting of 3 VM nodes using Ubuntu 14.04 and the m1.small flavor (1920MB Ram, 1 VCPUs and 40GB disk space). Since the streaming layer of `Ahab` is based on Spark Streaming and Hadoop, we use a Hadoop cluster consisting of one master node (Ubuntu 14.04 VM and m1.medium flavor) and 8 worker nodes

[¶] `https://coreos.com`
[‖] `https://www.docker.com`
[**] `http://projects.spring.io/spring-boot/`
[††] `https://www.openstack.org`
[‡‡] `http://www.rabbitmq.com`

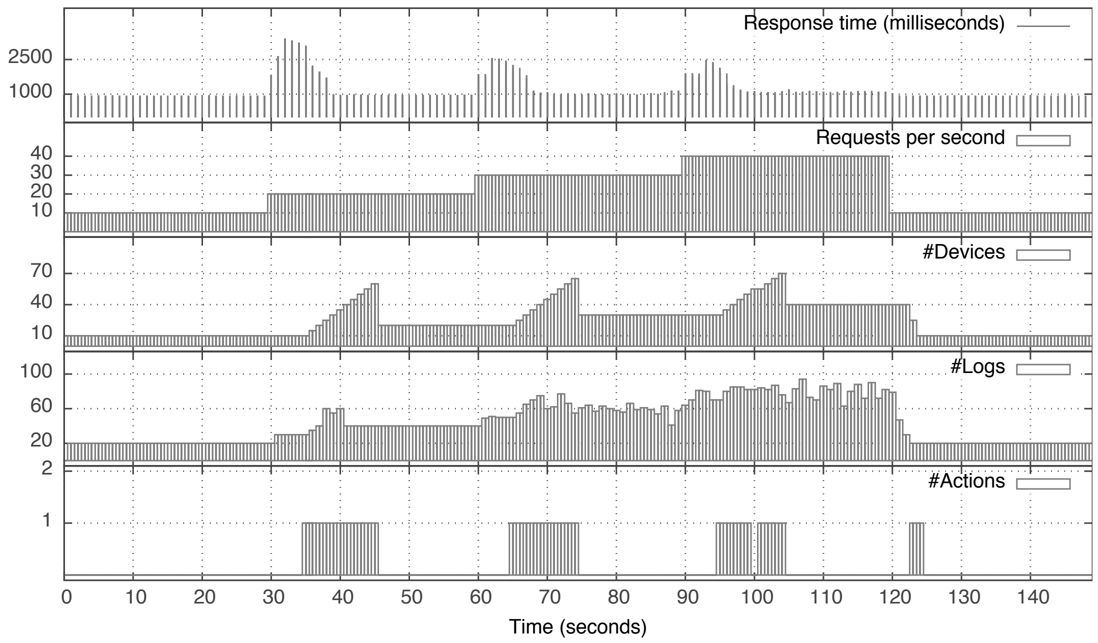

Figure 2. Evaluation Results – Scenario 1

(Ubuntu 14.04 VM and m1.small flavor). The service layer is hosted on an additional VM using Ubuntu 14.04 and the m1.medium flavor.

Finally, the smart city demo application is deployed on a separate VM using Ubuntu 14.04 and the m1.small flavor.

4.3. Scenario 1

In the first experiment we use `Ahab` to elastically scale the smart city demo application across the available IoT infrastructure by analyzing the stream of metrics published by the application. More specifically, in this scenario `Ahab` calculates, for each request sent to the application the response time and the number of concurrent requests per second. Next, `Ahab` analyzes logs published by the IoT infrastructure to detect how many devices are currently used by the application and how many are still available (i.e., idle). In addition to these processing streams, we define a stream that detects changes in the response time of the application by using a sliding window of 4 seconds that gets updated every 2 seconds. In this window the stream calculates the average of the windowed response times and triggers a notification whenever the average changes. Finally, we define a policy that defines the allowed threshold for the average response time increase and an action for scaling up the application by providing additional idle infrastructure devices that can be used. Furthermore, we also define a stream that detects when the current number of requests drops or infrastructure devices are not used anymore. Based on this stream, we define a policy for scaling down the application by releasing infrastructure devices.

Figure 2 illustrates the evaluation results for the first scenario. The x-axis shows the temporal course of the evaluation in seconds. In the 'requests per second' section we see that we started the evaluation by sending 10 concurrent requests per second to the application and increase the load stepwise every 30 seconds to see if `Ahab` can scale up the application. Finally, at 120 seconds we reduce the load to 10 requests per second to see if `Ahab` is also able to scale down the application. In the 'response time' section we see the response time for each incoming request. The 'devices' section illustrates the number of used edge devices by the demo application. The 'logs' section represents the processed incoming data stream of logs and metrics. Finally, the 'actions' section illustrates when `Ahab` detects that a policy was violated and triggers a respective action to

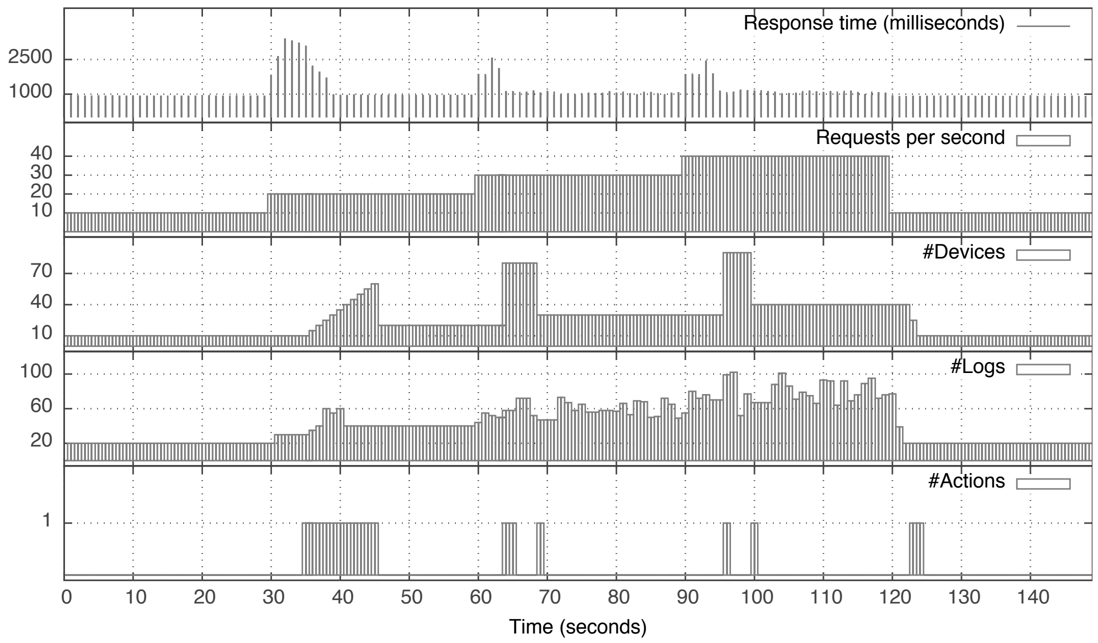

Figure 3. Evaluation Results – Scenario 2

compensate the violation. We executed this experiment 10 times, each of which produced almost identical results.

For the first interval of requests, we see that the response time of the application is almost constant. At 30 seconds, we doubled the requests per second and see that the response time rises, which indicates that the application is overloaded. At 33 seconds, `Ahab` detects that the response time increase is too high, initiates a scale up action, allowing the application to use 5 additional idle edge devices for handling the load. Since at this point a lot of requests got queued up by the application, `Ahab` issues one action per second until 44 seconds, where finally the application can handle the load by using in total 60 devices. After the response time reaches a normal level again and the application is only using a fraction of the devices to handle the requests, `Ahab` issues another action for scaling down the application and releasing devices. We see that the following 2 increases in requests per second at 60 and 90 seconds were also detected and analogously handled by the framework. Furthermore, for increasing requests and number of used infrastructure devices the incoming stream of logs and metrics that needs to be processed and analyzed by `Ahab` also increases. Finally, at 120 seconds, we see that the load drops from 40 to 10 requests per second, which is also detected by `Ahab`, leading to 2 actions for scaling down the application again. In general we see that `Ahab` is able to elastically scale the application by only using information extracted from published real-time data. Furthermore, considering the amount of data that has to be processed, `Ahab` is performing well and provides good results.

4.4. Scenario 2

In the second experiment, we once more use `Ahab` to elastically scale the smart city demo application, but in addition to analyzing and processing real-time data, as done in the first experiment, we now also consider historical data. For that reason we extend the first experiment to not only store incoming data to HDFS, but also save detected changes in the response time and respective actions. We furthermore added a batch processing stream that analyzes how `Ahab` compensated detected changes in the response time of the demo application in the past by using the historical information stored in HDFS. By also considering extracted information from historical data `Ahab` should be able to scale the demo application more efficiently when handling policy violations. We also repeated this experiment 10 times and each test run produced almost identical results.

Figure 3 illustrates the evaluation results for the second scenario. For the first request increase at 30 seconds, we see the same result as in the first scenario. `Ahab` scales the application stepwise, using 5 devices per step, until the application is able to handle the load and the response time reaches a normal level. This is done since at this time there is no historical data present that can be used by `Ahab`. At 44 seconds, `Ahab` detects that not all assigned devices are used by the application and issues another action to release idle devices. At 60 seconds we see the next request increase and once more the response time of the application rises. However, to compensate this increase, `Ahab` now uses the data collected from the previous compensation process and scales up the application by using 60 additional idle devices at once. By doing so, we see that the response time increase is compensated faster and that the overall compensation process takes only 5 seconds, compared to the initial one with more than 10 seconds. For the next request increase from 30 to 40 at 90 seconds, we see that `Ahab` once more uses the historical data to scale up the application. In general we can conclude that by using information from both real-time and historical data, `Ahab` is able to react faster and more efficiently on changes, which leads to improved performance and less policy violations.

5. RELATED WORK

With the advent of the smart city paradigm and its enablers, the Internet of Things (IoT) [6] and the Cloud of Things (CoT) [7], modern cities produce an ever growing amount of data that needs to be handled. To allow cities and governments to facilitate these large data sets, commonly referred to as big data [8, 9], effective management, processing, and analysis is of upmost importance. Recently, big data has received a lot of attention not just from cities and governments, but also from academia and industry. In general, when dealing with big data we can identify the following key challenges [10, 11] relevant for our approach. First, the management of big data, which comprises handling and storing big amounts of data efficiently and effectively. For storing data, Chang et al. propose Bigtable [12], a distributed storage system, which allows for managing structured data. Bigtable is specifically tailored for handling and storing petabytes of data that are distributed across huge clusters of servers. Based on a simple data model that provides clients dynamic control over data layout and format, Bigtable provides a flexible and high-performance solution for managing big data. In addition to Bigtable, there are also various other approaches available for storing big data (e.g., [13] and [14]). Next to storing, also the overall management of data is important. The cloud computing paradigm [15] emerged as a potential candidate that can provide the necessary resources to deal with the immense load needed for handling big data. Sakr et al. [16] and Ranjan et al. [17] present basic goals and challenges for deploying data-intensive applications in the cloud. In addition, Ji et al. [18] provide a comprehensive overview of commonly used approaches for processing big data in the cloud. In addition to general challenges and issues, Xhafa et al. [19] propose an approach for processing big data streams in real-time by facilitating the Yahoo!S4 (the simple scalable streaming system). The approach facilitates S4's actor model in order to allow for distributed computing. To demonstrate the feasibility, the authors evaluated their approach by using real-time data streams from a global flight monitoring system. Based on the evaluation the authors were able to show that their approach provides reasonably fast results. Compared to `Ahab`, the authors solely concentrate on processing real-time data and do not store data for later batch processing to extract additional information. In contrast to classical stream processing systems [20, 21] using a fixed amount of processing nodes, Heinze et al. [22] present an elastically scalable data stream processing system based on FUGU [23]. By using a model that analyzes and estimates latency spikes generated by scaling the system up and down, the authors propose an elastic latency-aware algorithm for the placement of stream processing operators to minimize SLA violations. Based on an evaluation the authors show that their approach can significantly decrease latency violations by postponing scaling decisions and allows the system to adapt its scaling strategy based on user input. Although this approach shares similarities with `Ahab`, our approach adapts managed infrastructure components based on knowledge extracted from processed data streams. Satzger et al. [24] propose ESC, a stream computing platform. In order to adapt to varying computational demands, ESC facilitates the cloud to dynamically attach and release resources. Furthermore, ESC provides a simple programming model based on DAGs that hides underlying aspects such as load distribution from the user. In contrast to `Ahab`, the need for storing data for historical purposes is not addressed in this work.

In addition to processing, analyzing big data is a vital aspect. Hummer et al. [25] present a scalable platform that allows active event-based aggregation of data streams. The approach provides an active query model and a language to correlate data streams. To handle the immense workload, the platform is designed for distributed query execution and can be deployed in the cloud. In order to provide more accurate system analytics, Chen et al. [26] present a query execution model that can be applied on both static relational data and dynamic streaming data. Based on this execution model the authors propose a system that combines stream processing and database capabilities. Although this approach also considers both real-time and historical data, it does not provide a complete and general solution that can be used to adapt infrastructure components. In order to combine processing and storing of big data in the context of IoT, Villari et al. [27] propose AllJoyn Lambda. The authors use AllJoyn, a communication platform for IoT devices, and integrate it with a Lambda Architecture. With this approach the authors provide a scalable solution for processing and storing big data, and furthermore allow real-time analytics. While this approach also uses a lambda architecture for managing both real-time and batch data, the authors solely focus on processing and storing of data, but do not provide a generic user interface that allows operators of smart city infrastructures to facilitate extracted information.

6. CONCLUSION AND FUTURE WORK

Todays smart cities produce an ever growing amount of data. These data streams contain various kinds of information such as monitoring events from underlying infrastructure, metrics published by used toolsets, as well as logs produced by executing smart city applications. This vital information about the smart city environment can be used to adapt and optimize different layers of the infrastructure in order to allow for more efficient execution and management of smart city applications. Therefore, efficient capturing, processing, storage, and analysis of this information is of upmost importance in the smart city domain, but is not sufficiently addressed by current approaches. In this paper, we introduced `Ahab`, a cloud-based, generic, scalable, and fault-tolerant big data analytics framework that allows operators to perform online and offline analyses on the gathered data. Based on this extracted actionable information operators are able to better understand and optimize the behavior of the managed smart city infrastructure. `Ahab` is designed to easily integrate new data streams and provides extensible APIs to perform tailored analysis tasks. To allow for the flexible definition of adaptation rules, the framework furthermore provides a DSL based on the MONINA language. In order to demonstrate the feasibility of the proposed approach we implemented an example application for autonomous intersection management in smart city environments and showed that our framework is able to autonomously optimize application deployment topologies by distributing processing load over available IoT infrastructure resources when necessary. Our approach allows using both, online analysis of the current state of the environment, as well as patterns learned from historical data, in order to manage and adapt application deployments more efficiently.

In our ongoing and future work we plan to build on and extend `Ahab` to address additional challenges in the smart city domain. We will investigate whether unsupervised machine learning techniques are suitable for autonomous improvements of application deployments. Furthermore, we will integrate and align `Ahab` with our work on software-defined IoT systems in the smart city domain [5, 28] to create a comprehensive platform for smart city application engineering. Finally, to also address privacy and security requirements when dealing with big data, we plan to integrate recent approaches (e.g., [29, 30]) in our framework's design.

REFERENCES

1. Inzinger C, Hummer W, Satzger B, Leitner P, Dustdar S. Generic event-based monitoring and adaptation methodology for heterogeneous distributed systems. *Softw: Pract. Exper.* 2014; **44**(7):805–822, doi:10.1002/spe.2254.
2. Zaharia M, Das T, Li H, Hunter T, Shenker S, Stoica I. Discretized Streams: Fault-tolerant Streaming Computation at Scale. *Proc. SOSP*, 1, ACM, 2013; 423–438, doi:10.1145/2517349.2522737.
3. Newman S. *Building Microservices*. O'Reilly Media, Inc., 2015.
4. Hausknecht M, Au TC, Stone P. Autonomous Intersection Management: Multi-intersection optimization. *Proc. IROS*, IEEE, 2011; 4581–4586, doi:10.1109/IROS.2011.6094668.
5. Vögler M, Schleicher JM, Inzinger C, Nastic S, Sehic S, Dustdar S. LEONORE – Large-scale provisioning of resource-constrained IoT deployments. *Proc. SOSE*, 2015; 78–87, doi:10.1109/SOSE.2015.23.
6. Atzori L, Iera A, Morabito G. The Internet of Things: A survey. *Computer Networks* 2010; **54**(15):2787–2805, doi:10.1016/j.comnet.2010.05.010.
7. Distefano S, Merlino G, Puliafito A. Enabling the cloud of things. *Proc. IMIS*, 2012; 858–863, doi:10.1109/IMIS.2012.61.
8. Mayer-Schönberger V, Cukier K. *Big Data: A Revolution that Will Transform how We Live, Work, and Think*. Houghton Mifflin Harcourt, 2013.
9. Bizer C, Boncz P, Brodie ML, Erling O. The meaningful use of big data. *ACM SIGMOD Record* Jan 2012; **40**(4):56–60, doi:10.1145/2094114.2094129.
10. Assunção MD, Calheiros RN, Bianchi S, Netto MA, Buyya R. Big Data Computing and Clouds : Trends and Future Directions. *Journal of Parallel and Distributed Computing* Aug 2014; **79**:1–44, doi:10.1016/j.jpdc.2014.08.003.
11. Chaudhuri S. What next? A Half-Dozen Data Management Research Goals for Big Data and the Cloud. *Proc. PODS*, ACM: New York, New York, USA, 2012; 1–4, doi:10.1145/2213556.2213558.
12. Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* Jun 2008; **26**(2):4:1–4:26, doi:10.1145/1365815.1365816.
13. Lakshman A, Malik P. Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review* Apr 2010; **44**(2):35, doi:10.1145/1773912.1773922.
14. DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W. Dynamo: Amazons Highly Available Key-value Store. *ACM SIGOPS Operating Systems Review* Oct 2007; **41**(6):205, doi:10.1145/1323293.1294281.
15. Agrawal D, Das S, El Abbadi A. Big data and cloud computing. *Proc. EDBT/ICDT*, ACM, 2011; 530–533, doi:10.1145/1951365.1951432.
16. Sakr S, Liu A, Batista DM, Alomari M. A Survey of Large Scale Data Management Approaches in Cloud Environments. *Commun. Surveys Tuts.* 2011; **13**(3):311–336, doi:10.1109/SURV.2011.032211.00087.
17. Ranjan R, Wang L, Zomaya A, Georgakopoulos D, Sun X, Wang G. Recent advances in autonomic provisioning of big data applications on clouds. *IEEE Trans. Cloud Comput.* 2015; **3**(2):101–104, doi:10.1109/TCC.2015.2437231.
18. Ji C, Li Y, Qiu W, Awada U, Li K. Big Data Processing in Cloud Computing Environments. *Proc. ISPAN*, IEEE, 2012; 17–23, doi:10.1109/I-SPAN.2012.9.
19. Xhafa F, Naranjo V, Caballe S. Processing and Analytics of Big Data Streams with Yahoo!S4. *Proc. AINA*, IEEE, 2015; 263–270, doi:10.1109/AINA.2015.194.
20. Abadi DJ, Ahmad Y, Balazinska M, Çetintemel U, Cherniack M, Hwang JH, Lindner W, Maskey A, Rasin A, Ryvkina E, *et al.*. The Design of the Borealis Stream Processing Engine. *Cidr*, vol. 5, 2005; 277–289, doi: http://www.cidrdb.org/cidr2005/papers/P23.pdf.
21. Wu E, Diao Y, Rizvi S. High-performance complex event processing over streams. *Proc. SIGMOD*, vol. 10, ACM, 2006; 407–418, doi:10.1145/1142473.1142520.
22. Heinze T, Jerzak Z, Hackenbroich G, Fetzer C. Latency-aware elastic scaling for distributed data stream processing systems. *Proc. DEBS*, ACM: New York, New York, USA, 2014; 13–22, doi:10.1145/2611286.2611294.
23. Heinze T, Pappalardo V, Jerzak Z, Fetzer C. Auto-scaling techniques for elastic data stream processing. *Proc. ICDEW*, IEEE, 2014; 296–302, doi:10.1109/ICDEW.2014.6818344.
24. Satzger B, Hummer W, Leitner P, Dustdar S. Esc: Towards an elastic stream computing platform for the cloud. *Proc. CLOUD*, IEEE, 2011; 348–355, doi:10.1109/CLOUD.2011.27.
25. Hummer W, Satzger B, Leitner P, Inzinger C, Dustdar S. Distributed continuous queries over Web service event streams. *Proc. NWeSP*, IEEE, 2011; 176–181, doi:10.1109/NWeSP.2011.6088173.
26. Chen Q, Hsu M. Cut-and-Rewind: Extending Query Engine for Continuous Stream Analytics. *Transactions on Large-Scale Data- and Knowledge-Centered Systems XXI*, *LNCS*, vol. 9260. Springer, 2015; 94–114, doi:10.1007/978-3-662-47804-2_5.
27. Villari M, Celesti A, Fazio M, Puliafito A. AllJoyn Lambda: An architecture for the management of smart environments in IoT. *Proc. SMARTCOMP Workshops*, 2014; 9–14, doi:10.1109/SMARTCOMP-W.2014.7046676.
28. Vögler M, Schleicher JM, Inzinger C, Dustdar S. DIANE – Dynamic IoT Application Deployment. *Proc. Mobile Services, Special Track - Services for the Ubiquitous Web*, IEEE, 2015; 298–305, doi:10.1109/MS.2015.49.
29. Perera C, Ranjan R, Wang L, Khan SU, Zomaya AY. Big Data Privacy in the Internet of Things Era. *IT Professional* 2015; **17**(3):32–39, doi:10.1109/MITP.2015.34.
30. Liu C, Ranjan R, Zhang X, Yang C, Chen J. A Big Picture of Integrity Verification of Big Data in Cloud Computing. *Handbook on Data Centers*. Springer New York, 2015; 631–645, doi:10.1007/978-1-4939-2092-1_21.