computers

MDPI

Article

Self-Adaptive Data Processing to Improve SLOs for Dynamic IoT Workloads †

Peeranut Chindanonda , Vladimir Podolskiy * and Michael Gerndt

Chair of Computer Architecture and Parallel Systems, Technical University of Munich, 80333 München, Germany; chindano@in.tum.de (P.C.); gerndt@in.tum.de (M.G.)
* Correspondence: v.podolskiy@tum.de

† This paper is an extended version of our paper published in SeAC 2019.

Received: 31 December 2019; Accepted: 7 February 2020; Published: 14 February 2020

Abstract: Internet of Things (IoT) covers scenarios of cyber–physical interaction of smart devices with humans and the environment and, such as applications in smart city, smart manufacturing, predictive maintenance, and smart home. Traditional scenarios are quite static in the sense that the amount of supported end nodes, as well as the frequency and volume of observations transmitted, does not change much over time. The paper addresses the challenge of adapting the capacity of the data processing part of IoT pipeline in response to dynamic workloads for centralized IoT scenarios where the quality of user experience matters, e.g., interactivity and media streaming as well as the predictive maintenance for multiple moving vehicles, centralized analytics for wearable devices and smartphones. The self-adaptation mechanism for data processing IoT infrastructure deployed in the cloud is horizontal autoscaling. In this paper we propose augmentations to the computation schemes of data processing component's desired replicas count from the previous work; these augmentations aim to repurpose original sets of metrics to tackle the task of SLO violations minimization for dynamic workloads instead of minimizing the cost of deployment in terms of instance seconds. The cornerstone proposed augmentation that underpins all the other ones is the adaptation of the desired replicas computation scheme to each scaling direction (scale-in and scale-out) separately. All the proposed augmentations were implemented in the standalone self-adaptive agent acting alongside Kubernetes' HPA such that limitations of timely acquisition of the monitoring data for scaling are mitigated. Evaluation and comparison with the previous work show improvement in service level achieved, e.g., latency SLO violations were reduced from 2.87% to 1.70% in case of the forecasted message queue length-based replicas count computation used both for scale-in and scale-out, but at the same time higher cost of the scaled data processor deployment is observed.

Keywords: self-adaptive IoT; Internet of Things; message queuing; autoscaling; cloud computing; autonomic computing

1. Introduction

Automation is a long-lasting effort of humanity to liberate one from numerous routine activities tightly interwoven with daily life. Although automation became quite common in the industrial world [1], there is still plenty of areas remaining that could be subject to automation. Automation of personal transportation (self-driving cars) [2], automation of daily care for elderly people and children [3], home automation [4] are among such areas that are nowadays addressed by solutions from the world of Internet of Things.

Internet of Things (IoT) is a paradigm that emerged from the abundance and affordability of hardware that directly interacts with the physical environment; in the literature, each such piece of hardware is often addressed to as an end node. Each piece of hardware is represented either by a small computer, also known as single-board computer (SBC), or by a microcontroller unit (MCU) placed on the same board with various peripherals, also known as a single-board microcontroller (SBM); the direct interaction with the surroundings is achieved by connecting sensors and actuators to SBCs and SBMs. Sensors use physical effects to measure numerical parameters of the environment such as temperature, humidity, pressure, or proximity to the object. Due to the complexity of surroundings, these sensors come in multiple forms, e.g., point sensors, grid sensors, or pairs of sensors such as photoelectric barriers. Actuation enables hardware IoT installation to influence its surroundings, e.g., close the door or signalize the dangerous environment condition via beeping. Although end nodes can be self-contained with sensors and actuators as long as they have sufficient power supply, more sophisticated solutions such as plant automation and predictive maintenance for a fleet of cars require data aggregation to enable high-level analytics and optimization processes. This is achieved by multiple approaches to data transfer that IoT installations support. The data transfer can proceed via multiple wireless protocols such as LoRa, NB-Fi, D7A, Sigfox both on a peer-to-peer basis and using the centralized publisher-subscriber solution with a message broker.

The data aggregation to enable higher-level scenarios for automated or semi-automated optimization and decision making requires centralization of data storage and computation. This centralization can occur in multiple ways. For instance, latency-critical application scenarios demand intermediate compute and storage facilities installed on-site; these are commonly known as edge devices. Edge devices allow us to overcome connectivity issues on the internet but they have limited capacity and thus can be used to only temporarily hide the effects of unavailable services and to lower the data processing latency. A further degree of centralization demands having own data center or deploying the data processing solution on the cloud. Although own data centers (DCs) might be preferred in case of computing and storage capacity being known in advance or in case of strict security requirements, having processing on the cloud enables dynamic scenarios when e.g., the number of active end nodes changes over time or end nodes move freely thus changing their proximity to DCs. Such dynamic scenarios with a centralized processing requirement demand scalability of the data storage and processing pipelines. Oftentimes, this scalability should be provided with a high degree of autonomy such that human intervention is not required; in this case, one would speak of self-adaptive data processing that incorporates facilities for monitoring of the adapted infrastructure and applications, for analyzing the monitoring data and planning the scaling, as well as for scheduling and taking scaling actions at the right time. Conventionally, in cloud computing, self-adaptation is realized through various types of autoscaling [5,6].

The scalability of processing and storage capacity is useful for a limited fraction of IoT scenarios. One of the most prominent examples is a smartwatch equipped with sensors measuring pulse, body temperature, altitude and various other characteristics of their owner [7]. These devices can be turned on and off frequently by their owners, on top of that these devices can change the location. This usage pattern of wearables leads to an opportunity to dynamically adjust compute and storage capacity to the changing number of devices that are online and to schedule computations and data hosting to sites which are in closer proximity to the wearable such that user experience is improved with lower latencies. Another example of an IoT scenario that demands scalability and, in particular, its geographically-aware form is data processing and streaming for vehicles—notably, seamlessly connected mobility and entertainment experience in moving cars can only be achieved by moving the data along the car path, which, in turn, corresponds to terminating and spinning up service instances along the path of the car. Leaving the matter of geographical awareness out of the scope of this paper, we focused on the more general-purpose type of scalability for IoT applications, i.e., on the scalability and self-adaptability of data processing pipelines used in IoT platforms.

Our previous work proposed and investigated various metrics for automating the scaling of message queuing subsystems such as estimated waiting time (EWT) and mixes of metrics with conventional CPU utilization and processing capacity [8]. The results of the work pointed at particular autoscaling metrics and combinations thereof as being the most promising for IoT scenarios; these papers build on the results of metrics evaluations in [8].

This paper extends the previous work in part of adapting the studied metrics and combinations thereof for horizontal autoscaling of the data processing service. In particular, this work elaborates on the idea of using different autoscaling mechanisms for the conduction of scale-ins and scale-outs of the data processor's replicas. We borrow producing rate and message length in IoT platform's queues as well as their forecasted equivalents as metrics of choice from [9] and extend them to better cover volatile workloads as discussed in the above examples. An implementation of the self-adaptive scaling agent in addition to the container orchestration platform Kubernetes is proposed to mitigate dependence on hard-coded parameters that render the highly dynamic adaptation for IoT scenarios impossible. The implementation was evaluated on the typical seasonal workload and had shown that the proposed equations for the desired number of replicas decrease latency SLO violations, e.g., from 2.87% to 1.70% in case of the forecasted message queue length used both for scale-in and scale-out, at the expense of the increased cost which is relevant for real-time IoT applications.

Section 2 of the paper presents the background of the conducted research. Section 3 provides a brief overview of the related works. Section 4 goes into the details of the proposed data processor autoscaling mechanisms and argues about the augmentations made to metrics from [9]. Section 5 explains the metrics and test environment used to evaluate the proposed autoscaling mechanisms. Section 6 summarizes the most important experimental results and discusses them. Section 7 concludes the paper and provides hints for the future work in the self-adaptive systems for IoT.

2. Background

2.1. Kubernetes

Kubernetes is an open-source container orchestrator enabling the deployment of containerized applications to a cluster of servers. At the time of writing, Kubernetes supports vertical autoscaling, cluster autoscaling and horizontal autoscaling. The latter is supported through an integrated Kubernetes' component called Horizontal Pod Autoscaler (HPA). HPA adjusts the number of pods (minimal unit of computation employed by Kubernetes, also a bundle of containers sharing resources) by increasing and decreasing it based on the monitored metric value such as e.g., CPU utilization.

HPA suffers from two major limitations. The first limitation is due to the single way of calculating the desired number of pod replicas as shown in (1).

$$desiredReplicas = \lceil currentReplicas \times \frac{currentMetricValue}{targetMetricValue} \rceil \tag{1}$$

The second limitation is due to hard-coded scaling constraints. At the moment of writing the paper, HPA will scale out the controlled pod replicas at most to be the double of the current number of replicas or four, whichever number is greater (https://github.com/kubernetes/kubernetes/blob/master/pkg/controller/podautoscaler/horizontal.go). Scale-in to the maximum desired number of replicas occurs within a stabilization period, which defaults to five minutes. These limitations of Kubernetes' HPA motivated the design and implementation of the custom self-adaptive agent which will be discussed further in the paper.

2.2. IoT Platform

2.2.1. Overview of the IoT Platform

Figure 1 reveals the reference architecture of the IoT middleware (platform) used in our experiments. The platform was developed by the students of the IoT practical course at the Technical University of Munich in the summer semester 2018 [10]. The following core components of the IoT platform are running in the Kubernetes cluster: IoT back end, IoT gateway, Kafka message queue and the data processor.

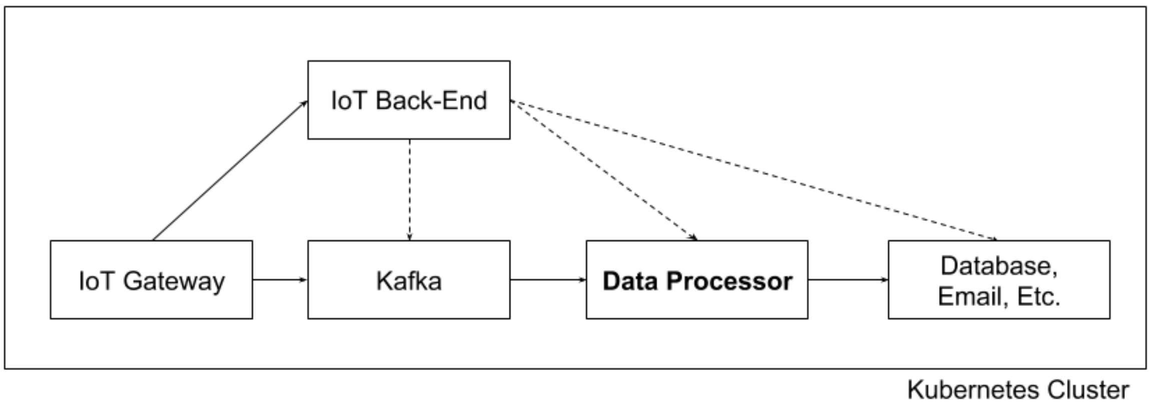


Figure 1. Reference architecture of the Internet of Things (IoT) platform used in experiments [10].

The IoT Platform implements the following general functionality enabling numerous IoT applications:

- accepting incoming traffic from end nodes via one of the following protocols—MQTT, WebSockets, HTTP;
- storing incoming messages into a queue until the data processing software is able to proceed with their processing;
- storing the results of the messages processing to the persistent database;
- access control through support for multiple users, devices and sensors;
- secure communication with end nodes via JWT tokens mechanism;
- data visualization in various formats ready for subsequent analytics.

A typical use-case for this IoT platform involves creating necessary virtual devices and sensors representations in the platform and downloading JWT tokens, adjusting the code deployed to the end node to put messages sent to the platform into the acceptable format and enriching them with downloaded security tokens, and then writing a customized streaming data processing code (in jar format) and uploading it to the Platform. Once the end node establishes the connection, it can securely send observations to the deployed platform—these observations will be processed there and then stored to the persistent database store.

In the following subsections, we dive deeper into major components of the IoT Platform to make the discussion of contributions clearer.

2.2.2. IoT Back-End

IoT back-end is a web application that lets users manage authentication and authorization, edge devices, sensors, storing sensor data, and alerts. The addition of a sensor into IoT back-end triggers the creation of the deployment that implements functions of data processing and of storing the sensor data. The data processing is done on the data consumed from Apache Kafka (https://kafka.apache.org/); data processing code is also responsible for checking the consumed data against the alert conditions and for sending out notifications. The processed data is stored in the persistent scalable Elasticsearch database (https://www.elastic.co/). Appropriate Kafka topics and Elasticsearch indices are also created on the creation of appropriate virtual devices and sensory representations.

2.2.3. IoT Gateway

IoT Gateway is responsible for receiving messages from edge devices via such protocols as MQTT, WebSocket, and HTTP with single Kubernetes deployment per protocol; this essentially means that IoT Gateway component is, in reality, three separate gateways that can be independently scaled and managed [11]. It checks for authentication and authorization of every incoming message, and drops the message if it is not authorized; legit messages are pushed to the Kafka queue.

Each message is in JSON format containing a timestamp in nanoseconds, sensor ID, and sensor reading. A sample message acceptable by the IoT platform is provided below:

```
{
‘‘timestamp’’: 15267967123,
‘‘sensor_id’’: ‘‘123’’,
‘‘value’’: 2420
}
```

2.2.4. Apache Kafka

Apache Kafka is a streaming platform similar to a message queue. Messages can be published to and consumed from some topic which allows us to distinguish between messages by different end nodes. A topic can have multiple data partitions/queues. Messages with a particular topic will be allocated to partitions in round-robin fashion. When multiple consumers are processing messages from a topic at the same time, single partition will be assigned to a single consumer; all the partitions will be assigned equally to available consumers.

2.2.5. Data Processor

Data processor is a component of IoT platform that consumes messages from the Kafka queue, derives an action to take on the message and acts accordingly by storing the payload into the database, sending out a notification, etc. Data processor polls and processes incoming messages as a stream, i.e., one by one; it is used throughout the paper as an example subsystem. The exact action taken on each message does not matter—the user of the IoT platform can customize it by uploading own processing logic.

3. Related Work

Several aspects of self-aware systems in IoT domain were outlined in [12]: self-aware reconfiguration of the high volume storage, self-aware continuous changes in run-time environments, and coordinating self-aware applications by manipulating QoS and power/resources consumption. An agent-based framework for self-adaptive IoT applications FIoT was proposed in [13]. This contribution focuses on the application part of the IoT and incorporates various types of adaptive, detecting and observing agents. Admitting that there are multiple objectives and methods for self-adaptation in IoT, a program to evaluate methods for self-adaptation in IoT named DeltaIoT was introduced [14]. R. Seiger et al. propose to utilize self-adaptive MAPE-K loop to attain the consistency between the physical world's state and its digital representation [15]. The self-adaptability of networks of IoT devices for increased resilience is studied in [16]. E. Casalicchio et al. conducted a study that has demonstrated the opportunity to reduce the response time for containers by using absolute metrics for autoscaling [17]. Gerostathopoulos et al. propose to introduce run-time changes to the architecture-based self-adaptation strategies in software-intensive cyber–physical systems and evaluate several mechanisms to change the self-adaptation strategy such as collaborative sensing, faulty component isolation from adaptation, and enhancing mode switching [18].

Cloud services providers are also known to incorporate support for IoT-related scenarios in their offers. In particular, AWS offers a guideline for autoscaling based on their SQS queue service (https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-using-sqs-queue.html); the guide recommends to employ the queue size divided by the current number of instances as the scaling metric with the acceptable latency (SLO) divided by the average processing time per message being the proposal for computing the target value of the metric.

4. Approach

The previous work [8] suggests using the estimated waiting time as a metric for autoscaling queue-based data processors that could help in saving computing resources. Additional research [9] was made to find suitable autoscaling mechanisms that prevent SLO violations and allow them to save computing resources. This is achieved by employing multiple techniques discussed in Section 4.1.

4.1. Autoscaling Techniques

4.1.1. Efficient Monitoring

Most autoscaling mechanisms fetch the data describing the state of the system from a standalone monitoring solution. Most monitoring solutions have predefined measurement interval; additionally, autoscaling mechanisms have own interval of time between consecutive scaling decisions. Overall, such a chain of entities transmitting the data only once in a while may damage data freshness in situations when the fast reaction is appreciated. The latest data might be obtained by the autoscaling mechanism by fetching the required data directly before use.

4.1.2. Short update Interval

Metric data of many autoscaling mechanisms defaults to update every minute. If there is a burst in incoming traffic, the system would not be able to scale in time and would then violate SLOs. Fetching system information and determining scaling action to take more frequently will make the response to changes in workload faster.

4.1.3. Scaling Direction-Aware Computation of Scaling Factors

Usually, autoscaling mechanisms do not distinguish between increasing or decreasing compute capacity when calculating them. In contrast to this common practice, we propose to employ a separate equation for scaling out and scaling in. The reason behind having separate calculation schemes is due to the necessity of having instances ready just in time for the coming workload when conducting the scale-out; thus, the calculation is better done in advance. On the other hand, the scale-in (or reduction in compute resources) is appreciated only in case the deployed application and the virtual infrastructure would not have to scale-out again within a meaningful amount of time; this amount of time has to outweigh the cost of scaling in and out again [19]. The equation that is used to calculate the desired amount of data processor instances for the IoT platform which capitalizes on the idea of computing the scaled-out and scaled-in amount of instances differently is shown below (see Equation (2)).

$$D = \max(D_{out}, \min(D_{in}, C)), \tag{2}$$

where D is the desired number of instances and C refers to the current number of data processor instances.

4.1.4. Forecasting for Autoscaling in Advance

Scaling out in advance to have enough instances of the service or virtual machine just in time to serve the upcoming workload demands some estimate of the future workload. Forecasting or time-series extrapolation is a technique that allows us to speculate about the upcoming workload and thus, given the capacity of the processing entity, to calculate the desired number of instances in advance. There are multiple methods that can be employed to forecast the amount of workload [20], but further in this paper, we investigate forecasting using double exponential smoothing as it has very small computational overhead in comparison to other candidate approaches.

4.1.5. Scaling Constraints

When scaling in data processor instances, messages in queues will be non-processable for a short amount of time as Kafka topic has to rebalance the work partition assignment. In order to prevent massive performance degradation, we suggest imposing a rule to limit the rate at which data processor can scale in (Kafka team is currently working on a new protocol for rebalancing called incremental rebalance protocol which is expected to minimize the performance impact when scaling—https://cwiki.apache.org/confluence/display/KAFKA/KIP-429%3A+Kafka+Consumer+Incremental+Rebalance+Protocol).

4.2. Data Processor Desired Replicas Calculation Improvements

The research [9] proposes three data processor replicas calculation schemes that we will build upon further; elaborate testing of these schemes uncovered performance issues that were resolved by improving original calculation schemes.

4.2.1. Producing Rate-Based (PR-B) as Scale-Out Equation and Forecast-Based Version of PR-B (I-FPR-B) as Scale-In Equation

This approach was proposed in [9] so that the data processor scales out when the processing capacity reaches a certain P. The research [9] uses message incoming rate instead of message processing rate to compute the processing capacity so that the desired number of instances can be calculated more precisely. This is the equation used in [9] as scale-out equation:

$$D = \lceil \frac{I}{R_{instance} \times P} \rceil, \tag{3}$$

where $R_{instance}$ is the estimated processing rate per instance (a median of the consuming rate from every data processor replica where the consuming rate is an inverse of processing time per message), I is the incoming message rate and P are the target processing capacity as a percentage.

In order to stress-test the abovementioned approach, an additional test was conducted on the deployed IoT platform. First, a set of messages was put in the queue so that many data processor replicas would be spin out. Second, the incoming message rate was drastically decreased, which resulted in the desired number of instances becoming low and thus triggering the scale-in event with a lot of messages still left in the queue unprocessed.

In this paper, we suggest fixing this drawback by calculating the resource capacity from the processing rate. Computing resource capacity with incoming rate could still be useful as it can produce a good estimation of the required number of instances; hence, we suggest the following rewrite of the Equation (3):

$$D = \lceil \frac{\max\{I, R\}}{R_{instance} \times P} \rceil, \tag{4}$$

where $P < 100\%$ and R is the current processing rate of the data processor.

The reason for P to be lower than 100% is that when the message incoming rate is low, processing rate is at its full capability; if, in contrast, P is 100%, the desired number of data processor instances will be the same as the current number of instances and will not trigger a scale-out action.

The original equation used to compute the amount of desired data processor replicas when scaling in based on the forecast is as follows:

$$D = \max_{\forall i \in \{0,1,2,\ldots,n\}} \lceil \frac{I_{t+i}}{R_{instance} \times P} \rceil. \tag{5}$$

The idea is to scale in if there is no need to scale out again within a period of n units of time. As discussed above, the processing rate should be employed to avoid incorrect handling of particular workloads. Hence, a new version of the equation was devised:

$$D = \max_{\forall i \in \{0,1,2,\ldots,n\}} \lceil \frac{\max\{I_{t+i}, R\}}{R_{instance} \times P} \rceil, \tag{6}$$

where $P < 100\%$

4.2.2. Forecasted Producing Rate-Based Scale-Out Equation (O-FPR-B) and Forecasted Producing Rate-Based Scale-In Equation (I-FPR-B)

The difference of (7) from (3) is that (7) uses forecasted message incoming rate. The equation from [9] is as follows:

$$D = \lceil \frac{I_{t+delay_{scale}}}{R_{instance} \times P} \rceil. \tag{7}$$

This equation should also use the processing rate if it is higher than the forecasted messages incoming rate. The rewrite of the Equation (7) is as follows:

$$D = \lceil \frac{\max\{I_{t+delay_{scale}}, R\}}{R_{instance} \times P} \rceil, \tag{8}$$

where $P < 100\%$

The improved scale-in Equation (6) was already discussed in Section 4.2.1.

4.2.3. Forecasted Message Lag-Based Scale-Out Equation (O-FML-B) and Forecasted Message Lag-Based Scale-In (I-FML-B) Equation

The O-FML-B equation from [9] was introduced with a goal to leave the least computing footprint possible; the equation is as follows:

$$D = \lceil \frac{S_{t+delay_{scale}}}{SLO \times R_{instance}} \rceil, \tag{9}$$

where $S_{t+delay_{scale}}$ is a simulation of the queue size over time. On each iteration of the simulation we get the following size of the queue:

$$S_t = \max(0, S_{t-1} + I_t - R_{instance} \times C). \tag{10}$$

For Equation (9), the processing rate is estimated by multiplying the processing rate per instance by the number of instances. However, by using a higher processing rate than the actual processing rate, the desired number of instances will be lower and thus the data processor will not scale-out and SLO could be violated so the new version uses the number of ready instances instead:

$$D = \lceil \frac{SR_{t+delay_{scale}}}{SLO \times R_{instance}} \rceil. \tag{11}$$

Similarly, Equation (10) becomes:

$$SR_t = \max(0, SR_{t-1} + I_t - R_{instance} \times C_{ready}). \tag{12}$$

When scaling in, we want to ensure that system would not have to scale out again within a certain amount of time. This was done in [9] by using the following equations:

$$\exists_{i \in \{1,2,\ldots,n\}} \frac{LNm1_{t+i}}{R_{instance} \times SLO} \geq C - 1 \tag{13}$$

$$\frac{L_t}{R_{instance} \times SLO} \geq C - 1 \tag{14}$$

$$D = \begin{cases} C & \text{if } (13) \wedge (14) \\ C - 1 & \text{otherwise,} \end{cases} \tag{15}$$

where $LNm1$ is similar to (10), but the predicted processing rate is calculated with $R_{instance} \times (C-1)$ instead.

The data processor should scale in only when there is certainty that it should be scaled in. Hence, the scale-in should only be possible when every data processor replica is in the ready state to make sure that the calculation is as accurate as possible. The improved equation for scale-in is proposed as follows:

$$D = \begin{cases} C & \text{if } C \neq C_{ready} \\ C & \text{if } (13) \wedge (14) \\ C - 1 & \text{otherwise.} \end{cases} \tag{16}$$

5. Evaluating Scaling Mechanisms

5.1. Evaluation Method

There are three general requirements that the scaled data processor should fulfill:

- Functional requirement: a certain percentage of messages should have a message waiting time within the SLO;
- Quality requirement: autoscaling mechanism should save compute resources when possible;
- Quality requirement: performance in terms of low average message waiting time should be kept;
- Quality requirement: the data processor should not scale often as the scaling actions could be expensive or could affect the performance of the data processor.

Each autoscaling mechanism is tested against listed requirements. The following metrics are used to evaluate the quality of autoscaling mechanisms:

- number of instance seconds to evaluate how much compute resources does the scaled state consume (lower instance seconds mechanisms preferred);
- average message waiting time to evaluate the performance of the scaled state (lower waiting time preferred);
- number of scaling actions to evaluate the frequency of scaling (lower number of scaling actions preferred).

5.2. Test Architecture

The architecture used for testing different scaling mechanisms shown in Figure 2 is the same architecture that was used in [9]. At first, the test cases have to be specified and submitted in the benchmarking dashboard. The benchmark controller will receive the test specification through WebSocket and will invoke load generation according to the specification.

The data processor always listens for new messages to process them as they appear in Kafka queues. In this case, the data processor will store the message in the ElasticSearch database.

The self-adaptive agent will react to the changes in the environment by executing MAPE-K (monitor-analyze-plan-execute with shared knowledge) loop [21] once in a fixed interval of time. During the monitoring phase, the agent fetches the environment information from Kafka, Kubernetes, and the data processor. Next, in the analysis phase, it calculates the desired number of data processor replicas according to the equations introduced above. The agent will apply scaling constraints in the planning phase; the scaling constraint that is being used is to scale in only one instance at a time if the desired number of instances is less than the current number of instances for 15 seconds. For the execute phase, the agent will create or delete the data processor's Kubernetes pods according to the planning phase demands.

The Prometheus monitoring system is used to fetch metrics from the self-adaptive agent. Those metric data are then displayed as a graph in the benchmarking and visualization dashboard.

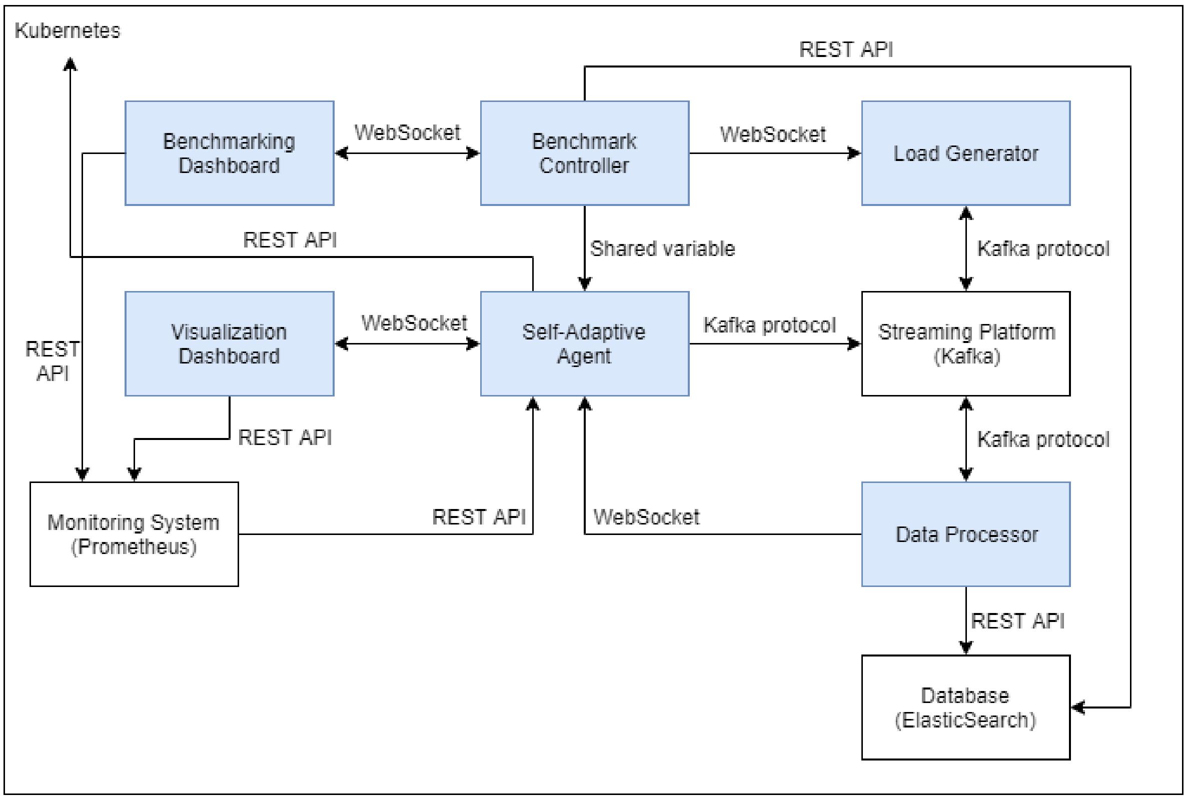


Figure 2. The architecture for evaluating scaling mechanisms [9].

5.3. Test Environment

A machine that has 2x Intel® Xeon® CPU E5-2640 v4 at 2.40 GHz and 128 GB of memory were used for the tests. The single-node Kubernetes is deployed on a KVM virtual machine where all the CPU cores and 24 GB of memory are assigned. All the tests were run on that single-node Kubernetes cluster.

As the functional requirement depends on the user, we settled with 95% of message waiting times have to be within 10 s for this test. The data processors are programmed to process at a maximum rate of 100 messages per second. As the test environment has a limitation that only the maximum of 4 data processors can be running at the same time, we only tested workload message rate up to 350 messages per second. A sample seasonal workload profile as in Figure 3 was tested.

The constant P used in (3)–(8) was set to 90%. The reason for that is that higher value will save more computing resources as less instances will be needed.

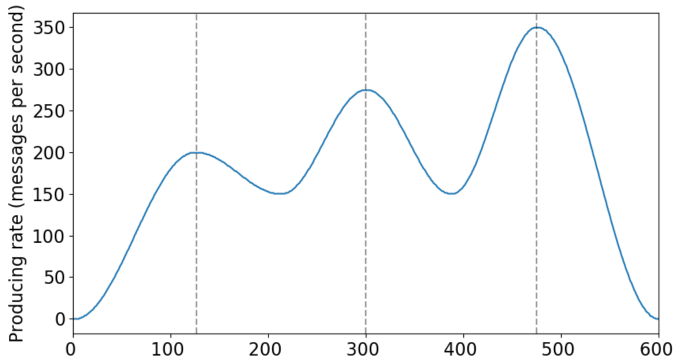


Figure 3. Simulated workload pattern.

6. Results and Discussion

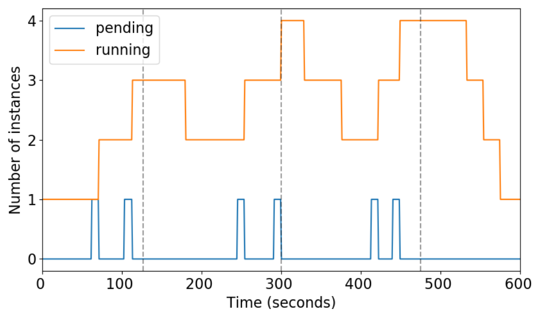
Let us first compare the old and the new equations as in Table 1. The workload used is shown in Figure 3. There is also a number of instances by time graphs in Figure 4. For every new equation, the resource usage as expressed in instance seconds is higher than for the old equation; this is expected as the new equation tries to prevent SLO violations in more scenarios. Using the third equation results in scaling less frequently which may be preferred when scaling actions are expensive in terms of computing resources or costs of SLO violations induced. The average message waiting time of the first and the third equation is lower which means that their performance is better and it would be harder for the SLO to be violated. For the second equation, one does not observe much improvement because this equation usually results in an early scaling. The SLO violation of the third equation is decreased from 2.87% to 1.70% as seen in Table 2. Hence, we can conclude that all the new equations decrease the possibility of SLO violations as intended.

Table 1. Test result comparing different auto scaling equation.

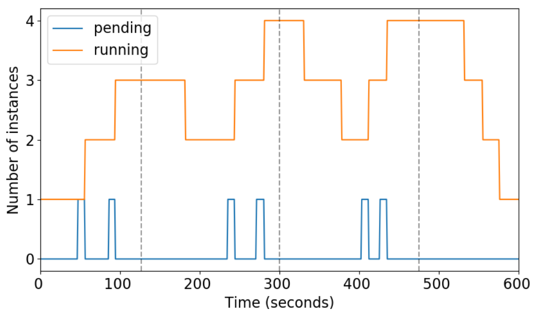
#	Scale-Out Equation	Scale-In Equation	Average Message Waiting Time	Instance Seconds	# Scaling Actions
1	(3)	(5)	64 ms.	1591	12
1 (new)	(4)	(6)	16 ms.	1683	12
2	(7)	(5)	15 ms.	1699	12
2 (new)	(8)	(6)	16 ms.	1732	12
3	(9)	(15)	3184 ms.	1273	8
3 (new)	(11)	(16)	2566 ms.	1304	10

Table 2. Test result showing waiting time distribution of different auto scaling equation.

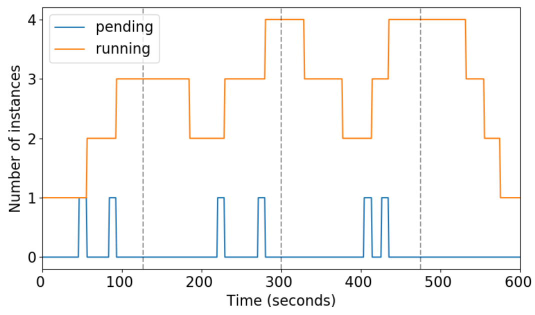
#	Scale-Out Equation	Scale-In Equation	Waiting Time (Seconds)					
			[0,2)	[2,4)	[4,6)	[6,8)	[8,10)	[10,∞)
1	(3)	(5)	99.97%	0.03%	–	–	–	–
1 (new)	(4)	(6)	100.00%	–	–	–	–	–
2	(7)	(5)	100.00%	–	–	–	–	–
2 (new)	(8)	(6)	100.00%	–	–	–	–	–
3	(9)	(15)	42.09%	22.16%	17.63%	10.69%	4.57%	2.87%
3 (new)	(11)	(16)	52.94%	17.45%	16.10%	8.99%	2.81%	1.70%



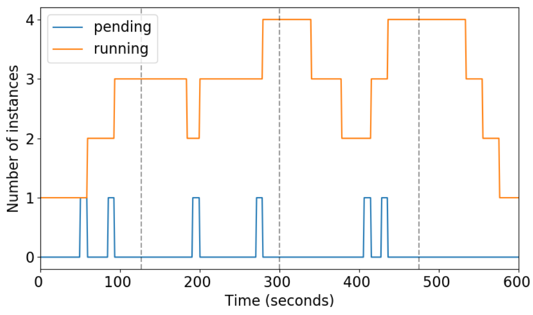
(**a**) (3) with (5)



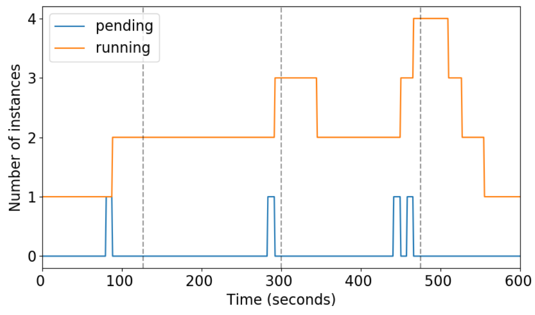
(**b**) (4) with (6)



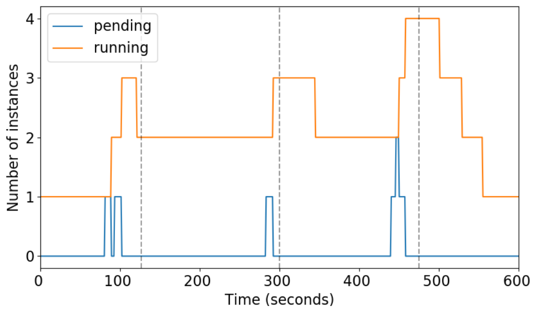
(**c**) (7) with (5)



(**d**) (8) with (6)



(**e**) (9) with (15)



(**f**) (11) with (16)

Figure 4. Number of instances by time of the test for each scaling equation.

For choosing between the three new equations, the logic of choice is still the same as in [9]: (1) the first and second equations should be used when lower message waiting time is preferred; (2) the third equation is to be used when computing resources should be saved; (3) when choosing between the first and the second equation, one should note that the first equation will always result in saving more computing resources than the second one; however, if the workload is extreme and SLOs are violated, then the second equation should be used.

A comparison of queue size-based autoscaling with approaches from AWS's guide (https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-using-sqs-queue.html), previous work [8], and a master's thesis [9] is shown in Table 3. The first difference is the use of a target tracking scaling policy. Target tracking scaling policy is the term used by AWS to describe a policy where we select a scaling metric and its target value (https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scaling-target-tracking.html). AWS's guide suggests to use the target tracking policy of AWS. Kubernetes Horizontal Pod Autoscaler also scales based on selected scaling metrics and its target value, so we say that it uses a target tracking scaling policy.

Table 3. Comparison of different queue-size based auto scaling approaches for queue-based data processor.

	AWS's Guide	Previous Work [8]	Master's Thesis [9]	This Work
Target tracking scaling policy	Yes	Yes	No	No
Use up-to-date metrics	No	No	Yes	Yes
Adaptive processing rate	No	No	Yes	Yes
Metric computed per instance	No	Yes	–	–

When we use metrics that are proportional to the current number of instances, the current number of instances used in the scaling equation like (1) and the current number of instances which are reflected in the metric may not refer to the same number [9]. This is because the metric is gathered or computed in its own interval and the scaling mechanism also recomputes the desired number of instances based on the current number of instances in another interval thus the result could be not accurate in some moment in time [9]. The master's thesis and this work use a custom-made self-adaptive agent for autoscaling which is adjusted to utilize the most up-to-date metrics.

Unlike in AWS's guide and previous work where the processing rate of the data processor is hard-coded in the equation, the master's thesis and this work use adaptive processing rate by receiving the processing rate information from the data processor and use its median value.

The difference between the approach suggested in AWS's guide and previous work is that in [8] we calculate the metric for each data processor whereas Kubernetes HPA will average metric from every data processor to get the final metric value. On the other hand, AWS's guide computes the final value directly from the total queue size. Computing the metric per instance could be useful especially when Kafka is used because each instance would have different partitions assigned and the queue size for each partition could differ; this means that the maximum message waiting time for each instance could be different as well. In this case, a more accurate estimation of maximum message waiting time could be done by finding the maximum out of all the maximum message waiting times per instance. However, Kubernetes currently can only average metric from every instance so average is used in the previous work. Ref. [9] and this work did not use target tracking scaling policy so metric computed per instance is left blank; nevertheless, the desired number of instances is internally computed based on summed values such as total queue size. Computing estimated waiting time for each data processor and then combining them might improve the accuracy of computing the desired number of instances.

7. Conclusions

The paper addressed the challenge of autoscaling the most critical part of the centralized IoT pipeline—data processor—in response to the highly dynamic workloads for IoT scenarios concerned with rapidly moving and disappearing/reappearing end nodes such that SLO violations are minimized. This challenge was addressed by introducing the direction of scaling as a variable determining the desired replicas amount computation scheme and extending the production rate-based and the message queue length-based computation schemes from [9] to tackle the scaling to the requirement of minimizing SLO violations in terms of latency. Evaluation of the implemented prototype of the self-adaptive agent demonstrated an improvement in the quality of service, e.g., the percentage of SLO violations when processing messages by scaled data processor decreased from 2.87% to 1.70% in case of the forecasted message queue length used both for scale-in and scale-out. This quality improvement came at the expense of the cost of the deployment as measured in instance seconds. Hence, the paper suggests to use the old set of metrics from [9] for scaling data processing in cost-critical cases whereas the new proposed set of desired replicas computation schemes is useful for the case of user experience-critical application scenarios.

Future directions in the domain of self-adaptive IoT solutions include ensuring scalability at edge and at the shared end nodes (e.g., multiple Kubernetes pods running on the RaspberryPi ensuring access to observations from different sensors attached), design of metrics and adaptation/scaling mechanisms for the interconnected set of application components forming the IoT pipeline (gateways, queues, data processors, databases, API, web-UI) and development of a set of IoT-specific standardized scalability benchmarks to ensure comparability of the results of studies in this domain.

Author Contributions: Conceptualization, P.C., V.P. and M.G.; data curation, P.C.; formal analysis, P.C.; investigation, P.C.; methodology, P.C.; software, P.C.; supervision, V.P. and M.G.; validation, P.C.; visualization, P.C.; writing—original draft, P.C. and V.P.; writing—review and editing, V.P. and M.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AWS	Amazon Web Services
HPA	Horizontal Pod Autoscaler
JSON	JavaScript object notation
MAPE-K	monitor-analyze-plan-execute with shared knowledge
SLO	service-level objective

References

1. Gilchrist, A. Industrial Internet Use-Cases. In *Industry 4.0: The Industrial Internet of Things*; Apress: Berkeley, CA, USA, 2016; pp. 13–31, doi:10.1007/978-1-4842-2047-4_2. [CrossRef]
2. Wu, W.C.; Liaw, H.T. The Next Generation of Internet of Things: Internet of Vehicles. In *Frontier Computing*; Hung, J.C., Yen, N.Y., Hui, L., Eds.; Springer: Singapore, 2018; pp. 278–282.
3. Balandina, E.; Balandin, S.; Koucheryavy, Y.; Mouromtsev, D. IoT Use Cases in Healthcare and Tourism. In Proceedings of the 2015 IEEE 17th Conference on Business Informatics, Lisbon, Portugal, 13–16 July 2015; Volume 2, pp. 37–44. doi:10.1109/CBI.2015.16. [CrossRef]
4. Mäkinen, S.J. Internet-of-things disrupting business ecosystems: A case in home automation. In Proceedings of the 2014 IEEE International Conference on Industrial Engineering and Engineering Management, Bandar Sunway, Malaysia, 9–12 December 2014; pp. 1467–1470, doi:10.1109/IEEM.2014.7058882. [CrossRef]
5. Podolskiy, V.; Mayo, M.; Koay, A.; Gerndt, M.; Patros, P. Maintaining SLOs of Cloud-Native Applications Via Self-Adaptive Resource Sharing. In Proceedings of the 2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), Umea, Sweden, 16–20 June 2019; pp. 72–81, doi:10.1109/SASO.2019.00018. [CrossRef]
6. Podolskiy, V.; Jindal, A.; Gerndt, M. Multilayered Autoscaling Performance Evaluation: Can Virtual Machines and Containers Co–Scale? *Int. J. Appl. Math. Comput. Sci.* **2019**, *29*, 227–244. [CrossRef]
7. Masoud, M.; Jaradat, Y.; Manasrah, A.; Jannoud, I. Sensors of Smart Devices in the Internet of Everything (IoE) Era: Big Opportunities and Massive Doubts. *J. Sens.* **2019**, *2019*. [CrossRef]
8. Chindanonda, P.; Podolskiy, V.; Gerndt, M. Metrics for Self-Adaptive Queuing in Middleware for Internet of Things. In Proceedings of the 2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W), Umea, Sweden, 16–20 June 2019; pp. 130–133, doi:10.1109/FAS-W.2019.00042. [CrossRef]
9. Chindanonda, P. Self-Adaptive Data Processing for the IoT Platform. Master's Thesis, Technische Universität München, Munich, Germany, 2019.
10. Podolskiy, V.; Ramirez, Y.; Yenel, A.; Mohyuddin, S.; Uyumaz, H.; Uysal, A.N.; Assali, M.; Drugalev, S.; Gerndt, M.; Friessnig, M.; et al. Practical Education in IoT through Collaborative Work on Open-Source Projects with Industry and Entrepreneurial Organizations. In Proceedings of the 2018 IEEE Frontiers in Education Conference (FIE), San Jose, CA, USA, 3–6 October 2018; pp. 1–9, doi:10.1109/FIE.2018.8658377. [CrossRef]
11. Dickel, H.; Podolskiy, V.; Gerndt, M. Evaluation of Autoscaling Metrics for (stateful) IoT Gateways. In Proceedings of the 2019 IEEE 12th Conference on Service-Oriented Computing and Applications (SOCA), Kaohsiung, Taiwan, 18–21 November 2019; pp. 17–24, doi:10.1109/SOCA.2019.00011. [CrossRef]

12. Möstl, M.; Schlatow, J.; Ernst, R.; Hoffmann, H.; Merchant, A.; Shraer, A. Self-aware systems for the Internet-of-Things. In Proceedings of the 2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Pittsburgh, PA, USA, 2–7 October 2016; pp. 1–9.
13. do Nascimento, N.M.; de Lucena, C.J.P. FIoT: An agent-based framework for self-adaptive and self-organizing applications based on the Internet of Things. *Inf. Sci.* **2017**, *378*, 161–176, doi:https://doi.org/10.1016/j.ins.2016.10.031. [CrossRef]
14. Iftikhar, M.U.; Ramachandran, G.S.; Bollansée, P.; Weyns, D.; Hughes, D. DeltaIoT: A Self-Adaptive Internet of Things Exemplar. In Proceedings of the 2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), Buenos Aires, Argentina, 23–29 May 2017; pp. 76–82, doi:10.1109/SEAMS.2017.21. [CrossRef]
15. Seiger, R.; Huber, S.; Heisig, P.; Assmann, U. Enabling Self-adaptive Workflows for Cyber-physical Systems. In *Enterprise, Business-Process and Information Systems Modeling*; Schmidt, R., Guédria, W., Bider, I., Guerreiro, S., Eds.; Springer International Publishing: Cham, Switzerland, 2016; pp. 3–17.
16. Athreya, A.P.; Tague, P. Network self-organization in the Internet of Things. In Proceedings of the 2013 IEEE International Workshop of Internet-of-Things Networking and Control (IoT-NC), New Orleans, LA, USA, 24 June 2013; pp. 25–33, doi:10.1109/IoT-NC.2013.6694050. [CrossRef]
17. Casalicchio, E.; Perciballi, V. Auto-Scaling of Containers: The Impact of Relative and Absolute Metrics. In Proceedings of the 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W), Tucson, AZ, USA, 18–22 September 2017; pp. 207–214, doi:10.1109/FAS-W.2017.149. [CrossRef]
18. Gerostathopoulos, I.; Skoda, D.; Plasil, F.; Bures, T.; Knauss, A. Architectural Homeostasis in Self-Adaptive Software-Intensive Cyber-Physical Systems. In *Software Architecture*; Tekinerdogan, B., Zdun, U., Babar, A., Eds.; Springer International Publishing: Cham, Switzerland, 2016; pp. 113–128.
19. Ilyushkin, A.; Ali-Eldin, A.; Herbst, N.; Bauer, A.; Papadopoulos, A.V.; Epema, D.; Iosup, A. An Experimental Performance Evaluation of Autoscalers for Complex Workflows. *ACM Trans. Model. Perform. Eval. Comput. Syst.* **2018**, *3*. doi:10.1145/3164537. [CrossRef]
20. Podolskiy, V.; Jindal, A.; Gerndt, M.; Oleynik, Y. Forecasting Models for Self-Adaptive Cloud Applications: A Comparative Study. In Proceedings of the 2018 IEEE 12th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), Trento, Italy, 3–7 September 2018; pp. 40–49. doi:10.1109/SASO.2018.00015. [CrossRef]
21. *An Architectural Blueprint for Autonomic Computing*; Autonomic Computing White Paper, Technical Report; IBM: Hawthorne, NY, USA, 2006.