Upilio: Leveraging the Serverless Paradigm for Building a Versatile IoT Application

Markus Mock(✉) and Stefan Arlt

University of Applied Sciences Landshut, Landshut, Germany
{Markus.Mock,Stefan-Alexander.Arlt}@haw-landshut.de

Abstract. Serverless computing has emerged over the last couple of years as a flexible paradigm for deploying cloud-based applications and allowing developers to focus on their applications and reduce application maintenance costs over the lifetime of an application. However, there has not been an examination of whether a complex application can be built and operated with high performance and low operating cost relying entirely on the serverless paradigm. This paper presents the design, implementation, performance, and cost evaluation of what we believe to be a representative kind of IoT application, a cloud-based energy data management system named Upilio. Upilio is a versatile data collection and analysis platform for IoT sensor data. Upilio's functionality is implemented entirely using AWS Lambda serverless functions and managed services to store data, and even the graphical user interface does not need a dedicated web server. Our empirical evaluation shows that the system, including its serverless online analytics (OLAP) functionality, is cost-effective, requiring only a fraction of the server cost necessary for operating such a system using on-premise hardware. Thus, Upilio demonstrates that complex IoT system scenarios can be implemented successfully with good performance and cost characteristics leveraging the serverless paradigm.

Keywords: Serverless Computing · FaaS · IoT

1 Introduction

Over the past fifteen years, cloud computing has fundamentally changed the computing landscape. Many applications that were traditionally run on-premises have moved to the cloud and are now often delivered as Software-as-a-Service (SaaS). Due to the unprecedented scale and elasticity of cloud computing resources and the ensuing agility, many new and entirely cloud-based companies have been created. While cloud computing has liberated those companies from procuring, upgrading, and maintaining their hardware, they typically still need to configure their servers and operating systems hosted by their cloud provider of choice to be able to run their applications.

Published by Springer Nature Switzerland AG 2022
F. Montesi et al. (Eds.): ESOCC 2022, LNCS 13226, pp. 121–136, 2022.
https://doi.org/10.1007/978-3-031-04718-3_8

Recently, serverless computing [16] has emerged as a new paradigm for deploying cloud-based applications with the promise of unburdening application developers even from configuring and scaling their cloud-based servers and instead enabling them to concentrate entirely on their application. Additionally, as computing time is only incurred when a function is performed, serverless computing also comes closer to deliver the original promise of cloud computing of paying only for actually used resources coupled with virtually infinite scalability. Moreover, due to the event-driven programming model, serverless computing also seems ideally matched to the implementation of IoT applications, which need to perform computations whenever new sensor data is produced.

However, while this technology has enormous potential, there are also challenges. For instance, there has been concern about potentially high operational costs due to the new billing model (e.g., [18]) and problems created due to storage disaggregation [30]. This paper examines whether a practical IoT application can be implemented with both good performance and cost-efficacy. We investigate the efficacy of using AWS Lambda for building a practical application that solves a vital real-world problem, namely energy (data) management. More importantly for our purposes, in addition to its practical relevance, we also believe that a cloud-based energy data processing system that collects, stores, and enables manual and automatic analytical operations on energy-consumption and production data is a good benchmark for evaluating the serverless paradigm for its viability for building complex practical applications because it combines continuous data collection from an IoT sensor system with analytical functionality and a graphical user interface to interact with the system.

To investigate how well the serverless paradigm can address these varied demands, we designed and implemented Upilio.[1] Upilio implements the data collection and monitoring part of an energy management system (EMS) for buildings by combining various simple data ingest, storage, and processing functions as AWS Lambda code to collect and store EMS data in real-time. It continuously applies machine-learning algorithms to produce and update predictions of energy consumption and production.

We evaluate the performance and cost-effectiveness of Upilio based on a workload that is typical for our university environment. We extract the relevant cost components and extrapolate operating costs to both higher and lower demands and show that Upilio operates cheaply even for large sites with many buildings and sensors with high data velocity. Combined with its extensibility and easy deployment, we conclude that Upilio and similar IoT applications can be implemented using the serverless paradigm with excellent cost and scaling characteristics.

This paper makes the following contributions:

– It provides a demonstration of how a complex real-world application can be implemented entirely using serverless technology.

[1] Upilio is Latin for "shepherd"; the Upilio system takes care of the users' sensor data in the cloud.

- It shows that with serverless technology combined with the infrastructure-as-code approach, a sophisticated system can be deployed and operated without first building a costly on-premise infrastructure and how updates to the edge sensor configuration can be performed from the cloud.
- It shows how a versatile web-based graphical user interface that is available 24/7 can be realized without dedicated web server hardware, i.e., serverlessly.
- It presents a blueprint for building scalable and versatile serverless IoT applications.
- Finally, the paper demonstrates that the system, including serverless OLAP functionality is cost-effective, requiring only a fraction of the server cost for operating the system on-premise hardware.

The remainder of the paper is structured as follows: after reviewing background and related work, Sect. 3 provides an overview of the Upilio system and its operating environment and provides details on how the analytics operations for OLAP and the graphical user frontend are implemented serverlessly. Section 4 provides an experimental evaluation demonstrating the benefits and cost-effectiveness of our approach and Sect. 5 provides conclusions.

2 Background and Related Work

Serverless computing has been receiving much attention recently as a potential fulfillment of cloud computing's original promise of liberating users from the burden of procuring and managing hardware and software (operating) systems and letting them instead focus entirely on their application-level code. Besides some open-source approaches, which still require someone to host and run the platform, many commercial offerings have emerged over the past years; for a comparison consult for instance [24]. Castro et al. [16] offer the following definition: "Serverless computing is a platform that hides server usage from developers and runs code on-demand automatically scaled and billed only for the time the code is running." They distinguish this general definition from a specific embodiment, namely Function-as-a-Service (FaaS), which however is often used interchangeably in the literature but, which narrowly defined, is a subset of serverless computing where the unit of computation are functions, which are executed typically in response to some event. AWS Lambda [11] is a commercial offering of a FaaS platform on top of which we have built Upilio. Using AWS Lambda, developers write code without considering on what hardware it will be executed.

Applications are broken into separate functions, which can be implemented in a variety of programming languages, e.g., Javascript, Java, Go, or Python; for Upilio, we have used the Python language. Users are billed in 100 ms-increments for actual compute resource usage and a Lambda function can execute for at most 900 s. The maximum memory available to the function has to be specified when a Lambda function is deployed; the selected memory determines the billing rate. The more memory is configured, the higher the CPU performance that the function is executed on; an independent selection of the two is impossible.

These Lambda functions are executed in lightweight containers [1] providing portability and security, as well as virtually unlimited scalability since the functions can be executed on as many machines as are available, and the user is willing to pay for. Other commercial cloud providers besides Amazon offer similar FaaS offerings, e.g., Google Cloud Functions [19] or Microsoft Azure Functions [14]. A primary advantage of the FaaS model is its simple event-driven programming model.

One recent development in IoT data processing is applying the FaaS model to the IoT domain to take advantage of the convenient programming model and the excellent scalability properties. Open-source offerings such as Apache Whisk [4] and commercial platforms such as AWS Greengrass [10] provide an execution environment for specific edge devices, thereby making it possible to execute functions that were initially written for the FaaS cloud platform at the edge. AWS Greengrass is a framework with a collection of software libraries that enables the execution of Lambda functions on IoT devices that are Greengrass-enabled. In Upilio, we run the Greengrass core system on Raspberry PI single-board computers that serve as data collectors at the edge (cf. Sect. 3.1). Greengrass provides the software engineering benefit of code reuse, as many Lambda functions written for the cloud can be executed unchanged on the edge devices. Lambda functions running on IoT devices can process data, execute cloud messaging operations and even perform learning inference operations and AWS Greengrass provides some prebuilt components to facilitate the development of such edge functionalities.

One additional advantage of the AWS Greengrass platform is that Lambda functions and configuration files can be deployed and updated from the Greengrass cloud service. This capability frees developers of edge applications from writing their own software management and update platform and from the time-consuming process of manually updating the software and configuration settings at multiple edge devices whenever a change is required, thereby increasing the agility of edge applications and reducing operational costs.

Aslanpour et al. [5] have looked more generally at the opportunities and challenges of applying the serverless paradigm to edge computing and, therefore, to IoT scenarios. Like others, they also point out the excellent match between the event-driven nature of IoT applications and the serverless programming model. However, on the downside, they also point out that high latency due to cold-start issues can be problematic for some applications.

Baresi et al. [15] propose a serverless architecture for a specific edge computing use case, namely mobile augmented reality. Using IBM's OpenWhisk serverless framework [23] in locally located servers, they compare the latency and general performance of their augmented reality application when executing the functions that provide a reality augmentation serverlessly in the cloud or at the edge. As expected, the latencies at the edge are lower than in cloud-based FaaS systems. However, their approach required them to set up their own edge serverless environment. In our approach, we are leveraging AWS Lambda's Greengrass integration, which allows the execution of serverless functionality at

the edge, on Greengrass-enabled devices, Raspberry PI single-board computers in our setup.

Wang et al. [31] present LaSS, an architecture for running latency-sensitive serverless applications at the edge. They use a queuing-theoretic framework to allocate resources to containers executing serverless functions to ensure that latency goals are met. They used OpenWhisk to implement a prototype of their system and evaluated it using a benchmark consisting of a handful of different latency-critical applications. Their work should be valuable to cloud providers for extending existing public clouds to support latency-critical serverless functions, precisely when those can be executed at the edge, such as AWS Greengrass or Azure IoT. There were no aggressive latency demands in our examine use case requiring edge execution. However, some IoT application scenarios do require ensuring maximum latencies. Currently, in AWS Greengrass edge execution, this can only be achieved by proper provisioning at the edge and potential overprovisioning as you would when operating a serverful application.

One step towards solving this overprovisioning problem for a public cloud is Pelle et al.'s work [26]. They propose a middleware layer for AWS Greengrass, which receives application-specific performance metrics and uses this information to change the edge configuration, e.g., by changing the placement of Lambda function execution on edge devices. They evaluate their system using simulations. However, how well the (positive) simulation results translate to an implemented system needs to be evaluated.

An actual prototypical serverless platform specifically for edge computation is presented and evaluated by Pfandzelter and Bermbach [27]. Their design is specifically geared towards resource efficiency and meant to run on single-board computers. They present a prototype implementation using Docker containers to place function handlers and a management service running on each edge host directly. To reduce resource requirements and latency, clients perform requests using the CoAP protocol, which is used in many IoT systems, rather than HTTP resulting in lower latency as CoAP is based on UDP transport rather than TCP. Their experimental comparison found that their platform introduced only minimal overhead compared to native Node.js execution.

Upilio, like many IoT scenarios, also requires performing analytics operations on the acquired sensor data. There has been a fair amount of work looking at serverless analytics. For instance, Nastic et al. [25] present a combined cloud and edge real-time data analytics platform that can perform analytics both at the edge and in the cloud. Simpler and latency-critical functions are executed at the edge, while more complex analytics can be executed in the cloud. Their model proposes processing the edge real-time data serverlessly. To facilitate that, they propose an extension of what they label the traditional streams model by adding serverless data analytic functions into the data stream. In contrast to some of their application domains, e.g., vital signs monitoring in a medical context, the latency imposed by transmission to the cloud is generally immaterial for a cloud-based energy management system such as Upilio. However, their proposal is similar to Upilio's approach since we also execute simple processing

functions at the edge using the AWS Greengrass core as described in Sect. 3.1. Simple anomaly detection methods can be executed at the edge so that energy consumption anomalies can also be detected directly at the edge without data transmission to the cloud.

3 Upilio: Design and Implementation

In order to make our results as generalizable as possible, Upilio's architecture follows these design goals. First, the system had to be able to accommodate heterogeneous sensor equipment, as is typically found when buildings are instrumented with energy sensors and are not built with instrumentation from scratch. However, even in the latter case, being able to switch suppliers to avoid provider lock-in is desirable. Moreover, since our architecture is built for heterogeneity, it also applies to other IoT sensor scenarios. Second, the system has to be easy to deploy without expensive capital investments and minimal operational demands. While automated control was not a requirement for our pilot system, the design still has to be extensible to allow for automated control, which motivated our choice of leveraging the AWS IoT Greengrass platform for our data collection operations as it enables us to execute some analyses, e.g., anomaly detection, at the edge as well; a requirement that many general IoT applications share as well. In addition, it is desirable that the fleet of IoT devices can be (re-) configured and updated from the cloud to minimize personnel costs.

Upilio is used to collect, store and process energy- and resource-related data from our university's three campuses, the main campus located in Landshut and two satellite campuses located in Lower Bavaria. To limit the number of data connections from each site and provide an opportunity for trading off data freshness and communication bandwidth, again a requirement shared by many IoT scenarios, measurement devices do not communicate directly with the Upilio cloud backend. Instead, each site uses one (and, for large sites, potentially multiple instances to avoid bottlenecks) data collector for which we use Raspberry Pi [28] single-board computers. They also serve as Greengrass core devices as explained in Sect. 3.1. Currently, at the main campus, Upilio is continuously ingesting data from approximately one hundred sensors. They measure electricity consumption in various buildings and laboratories, measuring both power consumption and aggregate energy use, taking measurements every minute. Regional heat and water consumption, again at building and sub-building levels, are measured in 15-min intervals. Besides, the university has solar panels and the electricity production from this installation with a peak power of 100 kW is measured every minute, as is weather data from an on-campus weather station, which measures global irradiance, temperature, and relative humidity at several ground levels.

The remainder of this section describes how we leveraged the serverless paradigm in Upilio's design and implementation in more detail and presents the reference architecture that should be usable as a blueprint for similar IoT sensor scenarios. Section 3.1 describes how data is represented and collected at the edge. Section 3.2 describes the data ingress APIs and Sect. 3.3 how the web-based

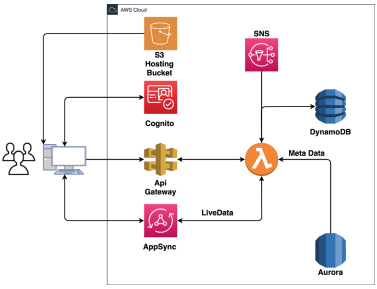

Fig. 1. Upilio frontend architecture. The serverless web frontend is implemented with static Javascript code in S3 buckets calling Lambda functions on REST APIs.

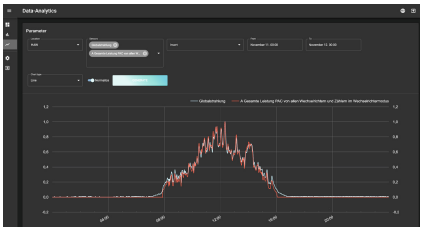

Fig. 2. A view of Upilio's analytics dashboard showing a combination of the photovoltaic energy produced at the main campus on the selected 24 h period overlayed with the global radiation, both normalized to display them at the same scale.

GUI was implemented serverlessly. Finally, Sect. 3.4 describes how analytics was implemented.

3.1 Data Representation and Real-Time Collection

To enable the use of sensors from different manufacturers that represent data in different formats and to be able to incrementally add new sensors to a running system without interrupting its operations, an essential requirement for our Upilio design was for it to be extensible. Adding a new sensor type should not require any code changes to the existing storage and processing backend. Moreover, it is essential to be able to change the frequency of measurements without manually updating configuration files at each sensor site because typically, sensors are located in utility tunnels or access-restricted locations so that updating them in situ is cumbersome. Furthermore, there are bound to be many sensors, and updating them all one by one would be laborious.

Data Representation. Interface definition languages (IDLs) are a well-known mechanism for representing data types in an extensible and portable way. We chose to use the Thrift IDL language [2], which is a language-neutral, platform-neutral, and extensible mechanism for serializing structured data in a compact and efficient form. Thrift comes with tools to automatically create serialization and deserialization code stubs from the data type description. Data types can be updated by adding additional (optional) struct fields without breaking the existing processing code.

Every sensor device is assigned a unique device id and its measurement value is represented as a double value. Also, two (Unix) timestamps are recorded:

the sensor timestamp, a timestamp set by the sensor, which records when the measurement was taken, and the reading timestamp, which records when the collector node (=RPI) reads the sensor value from the sensor. We use the NTP protocol to keep the clocks of our collector nodes synchronized to UTC so that we can correlate measurements taken at different sensors based on their timestamp within the NTP-synchronization accuracy, which is in the millisecond range more than sufficient for the frequency at which energy consumption needs to be measured. From this primary data type, more complex data types, for instance, to represent the collection of measurements of a weather station are composed. In addition, all of our data type definitions carry a version number making rolling updates to new definitions possible. This data representation should cover most sensor IoT scenarios, not just our concrete use case.

Edge Data Collection in Upilio. Upilio uses the AWS Greengrass platform to enable the execution of Lambda functions both in the cloud and at the edge. At every campus location, at least one Raspberry PI device serves as a data collector. Using the Greengrass platform provides two advantages: first, we can avoid reimplementing functionality that can be useful both in the cloud and the edge. For example, we can perform some simple anomaly detection both in the cloud and at the edge, using Arima (cf. [22] for details on anomaly detection in Upilio). The ability to perform monitoring or analysis operations at the edge is crucial for detecting unusual operating conditions even when Internet connectivity is lost or would have too high a latency. While Upilio does not drive any automatic control systems, this capability would be indispensable in such a case. Another crucial advantage afforded by Greengrass is that it enables us to update both the function implementation and the configuration files from the cloud. Furthermore, we can deploy updated code or configuration files reliably without implementing our own update mechanism, which, for instance, enables us to update the collection frequency in selected buildings without the need to access the RPI computers, which are mostly co-located with sensors in utility rooms and tunnels that are difficult to access and access-controlled.

Note that these two requirements of being able to perform latency-sensitive operations and being able to operate when Internet connectivity is lost is shared in many other IoT scenarios. Furthermore, configuration and code updates from the cloud are an important requirement in many scenarios as well, therefore being able to support them in Upilio makes the results reported in Sect. 4 generalizable to a large class of IoT scenarios.

For our data collection, we execute two processing functions at the RPIs only: a `SensorReading` function, which reads the measurements via the local network from all sensors listed in a configuration file and a `PackageAssembly` function, which assembles sensor measurements into larger data packets to send to the Upilio cloud backend. Both are triggered periodically by an auxiliary timer task also implemented as a Lambda function.

The *TimerTask* implementation initializes the reading of the sensors, whereby sensor-specific data such as IP address, device ID, register address,

and resource type are specified in a YAML [32] enabling Upilio to read different resources at different time intervals. *SensorReading* is a generic component that is specialized for the specific sensor type to acquire its data values. Finally, the *PackageAssembly* potentially aggregates multiple sensor readings before sending them to the backend allowing for tradeoffs between the number of data transmissions and freshness. In the case of network problems, the packets that are not confirmed from the backend are persisted locally and sent again when the connection is re-established.

3.2 Data Ingress APIs

Upilio provides two APIs for ingesting data into the system: the real-time data ingress consisting of data collector computers at the edge (the RPIs in our current setup), which transmit sensor data to Upilio as the measurements are being produced. A second interface consists of a simple file drop mechanism, which allows for the upload of current and historical data. This second mechanism is helpful for two purposes. First, it allows for the integration of data collected before real-time instrumentation as well as for backfilling data that was not transmitted in real-time, e.g., due to more extended network connectivity issues.

Real-time Data Ingest API and Data Processing. The sensor data is sent from the RPIs to the Upilio backend, which is running in the AWS cloud. Our real-time data ingest API is built on top of the publisher-subscriber system that is part of the AWS Simple Notification Service (SNS) service [12]. For each resource type (electricity, gas, water, weather data, among others) there is a corresponding SNS topic to which data of that type is sent (published) by the RPIs running the edge sites. SQS queues [13] are configured as consumers of the messages posted. Simple Queue Service (SQS) is a managed queuing service, which can be used to decouple various components of a microservices architecture, such as those employed by Upilio. Lambda functions can be configured to be triggered by data becoming available at an SQS queue. Our primary Lambda function, `SqsToDdb`, which is responsible for the data ingest, is triggered by data becoming available at any of the SQS queues that correspond to the SNS topics. `SqsToDdb` is configured to run with 128 MB, and for the sensor data we currently process, executes on average for ca. 600 ms.

We use DynamoDB [29], a highly scalable key-value store with low write and read latencies, to store all our sensor data. For the batch write that the `SqsToDdb` function performs, we experience average latencies of only 7–8 ms. Due to the low latency and high scalability properties, DynamoDB, and similar key-value stores are very popular in IoT scenarios, where data volume, scalability, and latency usually make relational databases impractical choices.

3.3 Serverless Frontend

Figure 1 shows a schematic representation of the Upilio serverless frontend. The key to a serverless web-based graphical user interface without operating a perma-

nently running server lies in the fact that AWS Simple Storage Service (S3) permits making buckets word readable and that AWS buckets are addressable via web URLs. The Upilio start page with some static content and Javascript is located under a specific S3 bucket address. That page shows a login screen and requires the user to log in to our systems. The user is authenticated using the Amazon Cognito [9] identity service, in which we create a user pool to control access to the Upilio frontend system. All accesses to frontend pages (all hosted in S3) are access-controlled using that system, which hands out an access token after successful authentication.

Once the user is logged on, they see a dashboard like the ones shown in Fig. 2. On the navigation pane on the left, the user can select an overview over all connected sites (campuses), a live data view, or the analytics dashboard (shown in Fig. 2). Alternatively, the user can update their account settings or log out. To perform Upilio operations, e.g., ask the system to display specific data and perform analytics operations, we have designed a REST-interface, which was implemented using the Chalice [6] framework and the Amazon API gateway [7]. Chalice is a collection of libraries and tools to make the development of serverless micro-architecture applications easier. The Amazon API Gateway is an AWS service that makes the operation of REST-ful web APIs possible without operating your own server.

The live data view is generated via the AppSync component. AppSync is an AWS service that provides an API compatible with the open-source GraphQL [20] query language for querying and displaying graphical data, originally developed at Facebook. Upilio uses GraphQL to query DynamoDB for the sensors' current values selected in the dashboard.

Figure 2 shows the dashboard view when the user selects the option to perform OLAP-style analytics, explained in detail in Sect. 3.4. In Fig. 2 the user has selected a slice of data from the main campus (HAW), the combined sensors of all photovoltaic production at that site (labeled 'A Gesamte Leistung PAC...') together with the global radiation, measured by the campus weather station (labeled 'Globalstrahlung'), and a time range from November 11th, 00:00 h to November 12th, 00:00 h. Since the two metrics produce values of very different magnitudes, the dashboard user also selected the option to normalize the data so that they can be displayed at the same scale. The graph very neatly illustrates that the photovoltaic installation was working as expected that day, as the electric power generated almost perfectly overlays with the global radiation, i.e., sunshine present during that day.[2]

3.4 Serverless Analytics

OLAP-style analytics in Upilio is also implemented, relying entirely on the serverless paradigm. As reported in Sect. 4, this can be done efficiently and at

[2] In fact, Upilio uses the difference between these two normalized values to detect anomalies in the functioning of the PV inverters. Details can be found in [22].

a low cost for Upilio's use cases. Crucial in that effort is the efficient implementation of aggregation operations since they can be costly when performed inefficiently. We chose two implementation approaches: first, we use a simple aggregation approach that computes aggregations that are likely to be requested by our system's users, namely aggregations along the time dimension aggregated in buckets of daily, monthly, and yearly intervals. Then, as a second approach, we implemented the well-known blocked-range-sum algorithm [21] by Ho et al., and evaluated the performance of both (cf. Sect. 4).

Simple Data Aggregation. For the simple data aggregation implementation, a separate DynamoDB table is created for the three aggregation levels "daily", "monthly", and "yearly", which are updated on the fly using an AWS Lambda aggregation function. This function is triggered by DynamoDB Streams [3], a DynamoDB service that provides a chronologically ordered sequence of item-level changes in each DynamoDB table. The timestamp determines each granularity level's corresponding index for each record in the stream. It serves as the primary partition key for the corresponding table. For example, the timestamp is converted into the number of days passed since the beginning of UNIX time (January 1st, 1970) for the daily aggregation table. If there already is an entry in that granularity bucket, its value is read and updated with the sum of the new sensor value and the prior sum; otherwise, the new value becomes the initial bucket value. To reduce write costs, all items from the DynamoDB stream are processed first, and then the updated sum values are written back to DynamoDB in one batched write operation.

As a more sophisticated alternative for computing aggregations that also provides a mechanism of trading of aggregation speed versus additional storage, we also implemented the Blocked Range-Sum algorithm developed by Ho et al. We evaluate its cost implications in Sect. 4.2.

4 Experience and Experimental Evaluation

In this section, we first report some general experiences we had when building and operating Upilio answering whether the serverless paradigm supports building performant and cost-effective applications in the IoT domain. Then we evaluate the cost and performance of the analytics operations.

4.1 General Observations

Confirming results by others, for instance, by Lee et al. [24], we were able to observe easy deployment and provisioning of Upilio thanks to the serverless model. We used AWS Cloud Formation templates [17] to define and deploy the components of Upilio, e.g., the DynamoDB tables, SQS queues, or Lambda functions. Cloud Formation templates are JSON files that define the AWS cloud components and their connections in a scripting language. They can be used with a command-line tool to bring up, update or turn off AWS cloud components. In

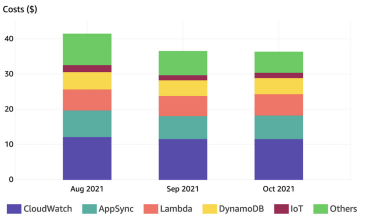

Fig. 3. The graph shows three typical months for Upilio's operational cloud computing costs. The "others" category comprises SNS, Greengrass, SQS, and S3 services.

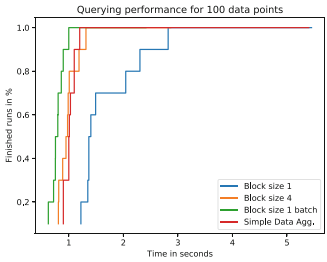

Fig. 4. Latencies of executing queries using the pywren engine. The queries execute within an acceptable time frame, however, the pywren engine introduces some additional latency due to the launching and warmup cost for lambda functions and the data storage in S3

one experiment, we were thus able to bring up another instance of Upilio at the simple execution of a single CLI command resulting in a running instance ready for data from the edge in approximately 15 min.

4.2 Experimental Results

General Operational Costs. Besides the benefits of easier deployment of provisioning, for our university environment and its amount of data, our usual operating costs for the data acquisition storage and the (at this point) low frontend usage are on the order of 35–50$US per month. Figure 3 shows operational costs for Upilio for a typical three-month period.

First, overall cloud computing costs for the current Upilio deployment with roughy 100 sensors reporting data in minute intervals and usual analytic dashboard use is usually around forty dollars a month. Five services, Cloudwatch, AppSync, Lambda, DynamoDB, and AWS Greengrass (reported as IoT in the AWS Cost Explorer graph in Fig. 3) account for approximately 85% of the total cost, and the other services that Upilio relies on, namely SNS, SQS, and S3 combined account for only 15%. Cloudwatch [8] is AWS' service monitoring service, which we use to monitor the correct functioning of Upilio to be notified, for instance, if less than the expected number of sensor data packets arrive at the Upilio cloud API.

The AppSync costs are incurred by frontend use when users display the current live data. For the current Upilio dashboard use with a low number, i.e., approximately 10 h of usage per week, these costs are under ten dollars per month. DynamoDB costs generally vary between two and eight dollars, the variation due to different frontend usage patterns: a higher number of analysis

Table 1. Data aggregation cost for all dimensions, block size of 4.

Block Size 4				
	Cost in $ per month			
Devices	Items/Month	DynamoDB	Lambda	Total
10	432000	2.91	0.0	2.91
100	4320000	2.91	0.0	2.91
1000	43200000	49.68	0.0	49.68

Table 2. Data aggregation cost for a block size of 1 (worst case) for 100,000 dimensions.

Block Size 1				
	Cost in $ per month			
Devices	Items/Month	DynamoDB	Lambda	Total
10	432000	2.91	0.0	2.91
100	4320000	2.91	0.0	2.91
1000	43200000	18.56	0.0	18.56

operations or the use of operations resulting in more read operations will increase the DynamoDB costs.

The Upilio computation costs, i.e., operations performed by the Lambda FaaS service was approximately five dollars for the three service months in the graph. Overall, the operational costs we have found are small, particularly considering the overall functionality and the reliability of Upilio, which directly benefits from AWS' 99.99% availability of resources in a single region.

For comparison, assuming we could operate Upilio on a single server, the hardware depreciation cost alone would already be higher. With electricity and IT personnel costs, this relation tips even more in favor of a cloud-hosted serverless architecture and, almost certainly, with higher availability and reliability of the cloud-based solution.[3]

Evaluating Analytics Operations. Besides the empirically observed typical operation costs, we also evaluated the costs incurred specifically for supporting fast online analytics operations. As mentioned in Sect. 3.4, we implemented the blocked-range-sum algorithm by Ho et al. [21] in addition to the simple aggregation algorithm that aggregates daily, monthly, and yearly values to make the answering of range queries fast in the time dimension. Note that the costs for the simple aggregation algorithm are included in the graph in Fig. 3.

For our implementation of Ho et al.'s algorithm, we evaluated how the storage and Lambda computation costs changed when varying the block aggregation block size and number of sensors. In Ho's algorithm, the larger the block size, the less storage is needed for pre-aggregation at the expense of more on-the-fly costs when answering queries. Therefore, a block size of one is the worst case in terms of required storage and precomputation costs.

The cost calculations are derived from processing the data packets typically produced by our concentrator nodes, i.e., We assume that each sensor is read out once per minute and that the system is running 24/7. We assume an average length of 30 days per month to calculate the monthly costs, which leads to 43200 measurement points per sensor. We use a singular Lambda instance for data processing, which is triggered by DynamoDB Streams. The trigger is configured so that the Lambda function is only triggered with a batch size of

[3] Assuming a $5000 purchase cost and a five-year depreciation, the monthly cost would come to $83.

10 times the number of sensors, 1000 for our main university campus, which reads out about 100 sensors. We are considering the worst case of computing all possible aggregations, i.e., we aggregate values over 473200 possible dimensions, consisting of four (time) granularity levels, 13 units, 13 buildings, 1 domain, 7 resource types, and 100 device ids ($= 4 \times 13 \times 13 \times 1 \times 7 \times 100$). Furthermore, we make the worst-case assumption that all dimensions are updated in every batch and that all device IDs occur in every dimension. In reality, most of these aggregations would never be requested by users of our system, and as such, the cost calculations represent a worst-case scenario. While we might know a priori for some aggregations to be helpful in practice, we cannot know what the system's users will do in practice. We consider implementing one approach to start computing a particular aggregation only once requested, thereby adapting to the typical usage dynamically. Furthermore, to save space, precomputing an aggregation could be stopped again if that aggregation is not used for a certain amount of time; we plan to implement and evaluate this approach in the future.

However, even for the worst-case scenario, the monthly costs were only about $10 for the processing and persistence of the data with a block size of 1 for our university scenario. Table 2 shows the costs for a varying number of sensors and when aggregating not for all but a fixed 100,000 dimensions. When more sensors are present and more data is produced, we keep the implementation cost-efficient by making the batch size, which triggers Lambda processing, a function of the system's number of sensors. We use a linear relationship multiplying the number of sensors with 10 to set the batch size. While this choice of batch size saves computation cost, it also guarantees that the aggregations are fresh: N sensors produce N data points per minute in our environment so that aggregation data will be no more than 10 min old.

As mentioned, using a block size of one creates the worst-case for pre-aggregation storage and computation costs. Table 1 shows the costs when computing the blocked range sums using a block size of 4. The table shows that if we use a block size of 4, we could even compute and store all possible aggregations at an acceptable cost. However, even in this case, adapting the system to actual usage patterns in the frontend would also save costs, as outlined above.

Besides the monetary cost of operating Upilio, performance in terms of latency is also essential. Therefore, user queries should be executed with negligible latency. To evaluate the effectiveness of the aggregation algorithms, we performed various queries over different lengths. The experiments compare the simple data aggregation and the blocked range-sum algorithm with block sizes 1 and 4. Our experimental setup used the Pywren engine to execute multiple lambda functions that combine the appropriate aggregation values to answer user queries. The median query time was from 1 to 2 s, sufficient for our current use cases (cf. Fig. 4). However, as others have observed before, launching Lambda functions via Pywren introduces some startup latency, not least because it uses S3 buckets to store code and data. S3 has a much higher latency for data access than Dynamo DB. Therefore, as part of future work, we want to evaluate

triggering the lambda functions directly from our frontend without using pywren and "keeping them warm," e.g., using a timer mechanism.

5 Conclusions

In building Upilio, we set out to examine if a scalable, cost-effective, and easily portable data collection and analysis platform for IoT sensor data can be built relying entirely on the serverless paradigm and using only off-the-shelf cloud computing building blocks. Upilio is sufficiently similar to other IoT data analytics scenarios that the results demonstrated in this paper should generalize to similar systems.

We have confirmed that developers can focus on designing and implementing functionality specific to their problem domain thanks to the flexible deployment model provided by a serverless platform like AWS Lambda. We found that the serverless paradigm enables creating scalable and performant systems without investing much time, money, or effort. Moreover, except for storage cost, which is very low, Upilio operating costs are proportionate with usage. In addition, performing operations mainly in the cloud was not a limitation for our system's data analytics use cases. Therefore, we have corroborated that creating applications using the serverless paradigm is particularly alluring for "small players" since a comprehensive system can be built with a minimal upfront cost.

In the future, we would like to perform a more detailed evaluation of the analytics operations, especially within the live system, i.e., based on typical workload demands and frontend operations.

Acknowledgements. We would like to thank the following students who have contributed to the implementation of various aspects of Upilio and its precursor: L. Brand, F. Huber, K. Kreitmeier, P. Loibl, W. Paintner, F. Saacke. P. Sacher, and L. Vögl. In addition, we thank the anonymous reviewers, whose valuable suggestions have helped us improve the final version of this paper.

References

1. Agache, A., et al.: Firecracker: lightweight virtualization for serverless applications. In: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2020), pp. 419–434 (2020)
2. Agarwal, A., Slee, M., Kwiatkowski, M.: Thrift: scalable cross-language services implementation. Technical report, Facebook (2007). http://thrift.apache.org/static/files/thrift-20070401.pdf
3. Amazon Web Services Inc: AWS DynamoDB Streams. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html. Accessed 11 Sept 2019
4. Apache OpenWhisk, February 2021. https://opemwhisk.apache.org/
5. Aslanpour, M.S., et al.: Serverless edge computing: vision and challenges. In: 2021 Australasian Computer Science Week Multiconference, ACSW 2021. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3437378.3444367

6. AWS: chalice a framework for writing serverless applications, November 2020. https://aws.github.io/chalice/index
7. Amazon API Gateway, February 2021. https://aws.amazon.com/api-gateway/
8. Amazon Cloudwatch, January 2019. https://aws.amazon.com/cloudwatch/
9. Amazon Cognito, February 2021. https://aws.amazon.com/cognito/
10. Aws Greengrass, February 2021. https://aws.amazon.com/greengrass/
11. AWS Lambda, February 2021. https://aws.amazon.com/lambda/
12. AWS Simple Notification Service, February 2021. https://aws.amazon.com/sns/
13. AWS Simple Queue Service, February 2021. https://aws.amazon.com/sqs/
14. Azure Functions, February 2021. https://azure.microsoft.com/en-us/services/functions/
15. Baresi, L., Filgueira Mendonça, D., Garriga, M.: Empowering low-latency applications through a serverless edge computing architecture. In: De Paoli, F., Schulte, S., Broch Johnsen, E. (eds.) ESOCC 2017. LNCS, vol. 10465, pp. 196–210. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67262-5_15
16. Castro, P., Ishakian, V., Muthusamy, V., Slominski, A.: The rise of serverless computing. Commun. ACM **62**(12), 44–54 (2019)
17. AWS Cloudformation, January 2019. https://aws.amazon.com/cloudformation/
18. Eivy, A., Weinman, J.: Be wary of the economics of "serverless" cloud computing. IEEE Cloud Comput. **4**(2), 6–12 (2017). https://doi.org/10.1109/MCC.2017.32
19. Google Cloud Functions, February 2021. https://cloud.google.com/functions/
20. Graphql, November 2020. https://graphql.org/
21. Ho, C.T., Agrawal, R., Megiddo, N., Srikant, R.: Range queries in OLAP data cubes. ACM SIGMOD Rec. **26**(2), 73–88 (1997)
22. Huber, F., Mock, M.: Toci: computational intelligence in an energy management system. In: 2020 IEEE Symposium Series on Computational Intelligence (SSCI). IEEE (2020)
23. IBM OpenWhisk, February 2021. https://developer.ibm.com/openwhisk/
24. Lee, H., Satyam, K., Fox, G.: Evaluation of production serverless computing environments. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 442–450. IEEE (2018)
25. Nastic, S., et al.: A serverless real-time data analytics platform for edge computing. IEEE Internet Comput. **21**(4), 64–71 (2017)
26. Pelle, I., Czentye, J., Dóka, J., Kern, A., Gerő, B.P., Sonkoly, B.: Operating latency sensitive applications on public serverless edge cloud platforms. IEEE Internet Things J. **8**(10), 7954–7972 (2021). https://doi.org/10.1109/JIOT.2020.3042428
27. Pfandzelter, T., Bermbach, D.: tinyFaaS: a lightweight FaaS platform for edge environments. In: 2020 IEEE International Conference on Fog Computing (ICFC), pp. 17–24 (2020). https://doi.org/10.1109/ICFC49376.2020.00011
28. Raspberry Pi Foundation: Raspberry Pi 4. https://www.raspberrypi.org/products/raspberry-pi-4-model-b/. Accessed 25 Aug 2019
29. Sivasubramanian, S.: Amazon dynamoDB: a seamlessly scalable non-relational database service. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 729–730 (2012)
30. Sreekanti, V., et al.: Cloudburst: Stateful functions-as-a-service. Proc. VLDB Endow. **13**(12), 2438–2452 (2020). https://doi.org/10.14778/3407790.3407836
31. Wang, B., Ali-Eldin, A., Shenoy, P.: Lass: running latency sensitive serverless computations at the edge. In: Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2021, pp. 239–251. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3431379.3460646
32. YAML: YAML Ain't Markup Language, February 2021. https://yaml.org/