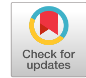
Future Generation Computer Systems 109 (2020) 1–16

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

CCoDaMiC: A framework for Coherent Coordination of Data Migration and Computation platforms



Chinmaya Kumar Dehury, Satish Narayana Srirama *, Tek Raj Chhetri

Mobile & Cloud Lab, Institute of Computer Science, University of Tartu, Tartu 50090, Estonia

A R T I C L E I N F O

Article history:
Received 20 November 2019
Received in revised form 9 February 2020
Accepted 10 March 2020
Available online 18 March 2020

Keywords:
Data pipeline
Data flow management
Serverless computing
Data migration
TOSCA

A B S T R A C T

The amount of data generated by millions of connected IoT sensors and devices is growing exponentially. The need to extract relevant information from this data in modern and future generation computing system, necessitates efficient data handling and processing platforms that can migrate such big data from one location to other locations seamlessly and securely, and can provide a way to preprocess and analyze that data before migrating to the final destination. Various data pipeline architectures have been proposed allowing the data administrator/user to handle the data migration operation efficiently. However, the modern data pipeline architectures do not offer built-in functionalities for ensuring data veracity, which includes data accuracy, trustworthiness and security. Furthermore, allowing the intermediate data to be processed, especially in the serverless computing environment, is becoming a cumbersome task. In order to fill this research gap, this paper introduces an efficient and novel data pipeline architecture, named as CCoDaMiC (Coherent Coordination of Data Migration and Computation), which brings both the data migration operation and its computation together into one place. This also ensures that the data delivered to the next destination/pipeline block is accurate and secure. The proposed framework is implemented in private OpenStack environment and Apache Nifi.

© 2020 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

Cloud maturity and its unprecedented growth in the number of connected devices in the latest years is changing the computing paradigm and this may further increase in future generation digital system. According to Ericsson, by 2023, there will be 8.9 billion mobile subscriptions, 8.3 billion mobile broadband subscriptions, 6.1 billion unique mobile subscribers, and also 1 billion 5G devices for enhanced mobile broadband [1]. It is forecasted to have 39% compound annual growth rate in total monthly mobile data traffic worldwide by 2023 from 15 Exabyte (EB) in 2017 to 107 EB [1].

The impact of the data abundance considering its application areas, especially monitoring and control [2], extends well beyond just business [3]. It is, therefore, necessary to handle these data considering all the changing dynamics to get benefits from it. Handling the data refers to transforming the data from one structure to another, processing the data to retrieve the useful information, and eventually loading the processed or the raw data to the desired location. Based on the situation, these operations may follow different sequences. Different big data platforms are available to implement the above operations.

However, the current big data platforms are not enough matured to efficiently handle the data generated by different smart environments [2,4]. To handle such huge IoT sensor data, cloud computing is mainly used to provide computing and storage supports. Such resource-intensive big data are migrated to cloud computing for processing. For example, Yassine et al. [5] proposed data analytics considering the smart home with cloud computing environment. Similarly, Plageras et al. [6] presented a technology for collection and processing of IoT sensor big data deployed in smart building. The responsibility of the cloud provider is to provide enough storage, computing, and network bandwidth resource. The cloud environments are generally in a far distance that introduces network latency — such issues are directing research and commercial communities towards the introduction of fog and edge computing [7,8]. Hence the data are first moved to fog/edge computing for pre-processing the data followed by moving the data for final processing in cloud computing [9].

The current Data Pipeline (DP) architectures are providing a way to handle the flow of the data from one provider to another provider, e.g., from fog to cloud or from edge devices to nearby fog environments. The DP architectures also provide functionalities to carry out the fundamental pre-processing operations. The fundamental pre-processing operations could be replacing

* Corresponding author.
E-mail addresses: chinmaya.dehury@ut.ee (C.K. Dehury), satish.srirama@ut.ee (S.N. Srirama), tek.raj.chhetri@ut.ee (T.R. Chhetri).

https://doi.org/10.1016/j.future.2020.03.029
0167-739X/© 2020 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

and updating the messages, converting the sensor data from its raw format to human-readable format etc. The implemented technologies such as Apache Nifi allow the developer to add multiple DP modules to the existing set up to handle the data from multiple data sources such as FTP servers, thousands of sensors, application logs, etc. Some of the primary demands of modern applications are the ability to perform the data analytics tasks with high robustness and to scale up or down based on the demand dynamically.

In addition to the scalability, the modern smart environments (such as smart city and smart home) demand a platform not only to handle the data flow/migration but also demand the processing feature while migrating the data, which is one of the major motivations to carry out this research. In extension to this, modern smart environments also demand a way to handle the data that include migration and processing, in a secure manner.

One of the major competitors to data pipeline approach is ETL, acronym of *Extract, Transform, and Load*. ETL mainly focuses on batch data processing, it carries out all the three basic operations for a batch of data before moving to next batch of data. The major challenges in ETL is its ability to perform the given tasks in real-time manner [10]. In contrast to ETL, ELT (extract–load–transform) is used to handle some of the disadvantages of ETL [11]. Unlike ETL, in ELT the data are transformed within the data warehouse, resulting in a reduction in network usage. However, in both ETL and ELT systems, it is hard to handle the real-time events and require a huge computation power. On the other hand, data pipeline addresses the issue associated with ETL and ELT approaches. The major purpose of data pipeline is to carry out the transformation step upon obtaining the data from its source before delivering to the destination [12]. This enables on-the-fly transformation of the data, which encourages the developers to process a huge amount of data in a real-time manner.

Traditionally, processing the data in the cloud mainly done by sending the data and the software package to the virtual machine/instance. This approach of processing the data mainly comes with an inbuilt pitfall of computation latency at the inception of instance. This problem sometimes refers to the cold start problem. Serverless is introduced to mitigate the cold start issue in a cloud environment. Serverless mainly takes advantage of kernel-level virtualization. Docker is one of the de-facto tools facilitating a platform to manage the life-cycle of light-weight virtual machines, also known as a container. Here, users are able to host and execute functions and will be billed for the actual resource the data-processing code is using.

Furthermore, continuing the demand for modern smart environments, handling the data veracity is another crucial feature that modern DP architecture should offer [13,14]. Ensuring data veracity infers the precise, accurate, and trusted data migration and processing [15,16] . Combining the above scenarios, it is observed that the modern smart environments generating a massive amount of streaming and batch data demand a DP architecture that provides a platform to handle the flow and processing of data in a coherent manner ensuring the veracity of the data, which is the main focus of the paper.

The rest of the paper is organized as follows: Section 2 presents the technical background, motivation behind the proposed framework and summarizes our contributions. The related works about the data pipeline are described in Section 3. Section 4 gives the detailed description of proposed data pipeline framework. Section 5 shows the implementation of proposed framework, followed by the concluding remarks and scope for future works in Section 7.

2. Background, motivation, & contribution

DP primarily focuses on the smooth handling of data while it is being transformed, cleaned, grouped, transferred, mapped to a different context, processed, and stored. Several technologies and businesses are adopting the DP approach for efficient handling of a huge amount of streaming and batch data. The commercially available DP technologies provide various basic functionalities, such as pipeline to collect data from various data sources, pipeline to broadcast or multicast the data to the connected pipeline modules, to send the data to remote location over different protocols, to remove the unnecessary information associated with the data, to combine the data based on its characteristics, etc. The essential parts of the DP technologies can be end-points (source and destination), load, automation (time and event-driven) correction, standardization, etc.

The commercially available technologies/products that use DP are Apache Nifi, AWS data pipeline, Kafka, Luigi etc. Apache Nifi [17] allows the developer to automate the flow of data between multiple software systems. Using Luigi [18], on the other hand, the complex pipeline of batch jobs can be built with the ability to resolve dependency conflict, management business workflows, visualization of workflows etc. To handle the movement of the data within the AWS instances and service, a dedicated data pipeline service is offered by AWS to its consumer [19]. This offers two basic components: (a) task runner to execute a specific task such as copying a data object from one virtual machine to another and (b) pipeline scheduler to automate a pipeline based on a specific time or event.

2.1. Motivations

Data pipeline allows a developer to effectively manage the flow of data (or information) from one system to another. Numerous DP frameworks are developed and become available commercially, such as Apache Nifi [17], as discussed before. Apache Nifi, being one of the most popular DP frameworks, allows the data administrator to define flexible data flow schemas and follow the movement of each data packet, but lacks to ensure the availability of accurate data and trustworthiness. Accuracy in this context refers to the similarity of the content at destination and source. In order to handle the delivery of accurate data, the developer needs to give additional effort by attaching a supplementary task to each DP component. This may become a cumbersome and complicated task when the number of DP components increases to hundreds or even larger.

Further, it is essential to verify the authenticity of the data. The existing DP frameworks do not support such functionality. As a result, a third party intruder may corrupt the data during its transmission, making developer clueless of data corruption during transmission. A novel DP framework is introduced to mitigate the aforementioned scenarios of accurate data availability and data authenticity.

In addition to the aforementioned features, the vast amount of real-time data need to be processed in the intermediary locations. For instance, uploading the smart home data to the cloud storage may involve transforming the raw data to the structural and readable form, handling data inconsistency, and missing value prediction. To carry out such preprocessing tasks, the data developer needs to attach the computing environment to each DP component. The developer needs to define when, where, and how the data need to be preprocessed. Further, the developer must ensure that authentic and accurate data are available in the computing environment, making the coupling of flow and processing of data more complex and challenging to achieve.

A huge amount of resources (more than the actual demand) are allocated to process a user's task, which is further minimized

with the introduction of serverless technology. However, combining the serverless platform with the DP framework introduces new research challenges and directions in terms of different aspects, such as delivery of authentic and accurate data, and combining serverless computing environment to specific intermediate DP modules.

2.2. Contributions

Based on the above motivation behind this work, our main contributions to this paper can be summarized as follows:

- In the process of designing the proposed data pipeline architecture, we have integrated both computation (with serverless platform) operation and data migration operation.
- In order to make the migration process smooth with the processing capability on the way, we have designed and implemented a dedicated component that ensures the availability of data at every adjacent end-point.
- Extending the previous contribution, we also have designed and implemented a dedicated component to ensure the data veracity that infers the preciseness, accuracy, and trustworthiness of the data at every end-points.
- We have designed the TOSCA (Topology and Orchestration Specification for Cloud Applications) model for the DP in extension to the existing models.
- A dedicated component is designed to analyze the actual resource requirement for each user's function based on the size of the data and the function itself.
- We have introduced dedicated components that select the serverless platform on-the-fly based on the actual requirement of the user's function's resource requirements.

3. Related works

A number of architectures have been proposed for serverless computing and data pipelines, considering different use cases such as in smart industry [20], edge computing [21], open city [22], etc. In this section, a brief survey on recent data pipeline architectures and serverless platforms is presented.

O'Donovan et al. [20] present a data pipeline architecture for industrial big data. The advantage of the proposed architecture is its ability to smoothen the ingestion of data from industrial sources such as sensors, controllers, etc. However, the architecture lacks the ability to support serverless environment. As a result, the resource requirements of data analytics are not optimized.

To overcome the Elasticity and Resource contention issues, Josep Sampé et al. [23] proposed a middleware that works in between user's storage API and the actual storage node in the cloud. However, it is not clear how the proposed architecture ensures data availability.

Saha et al. [24] proposed a resource management system for serverless cloud computing. The goal was to improve the response time and the latency for the allocation of desired resources to the functions that are invoked through HTTP requests.

Sewak et al. [25] describe serverless and function as a service (FaaS) and their applications, advantages, limitations, etc. Further, the article provides an underlying architecture for the serverless platform. It can be observed that the basic architecture does not provide any means of monitoring the status of actions (aka functions).

In [21], a framework for reorganizing the data pipeline (DP) function is proposed. DP functions refer to the acquisition, processing, and analytics that focuses on IoT, Edge, fog, and Cloud environment. The framework supports handling the real-time data generated by IoT devices. However, it is not clear how the framework ensures the fulfillment of resources from a fog-cloud computing environment. Further, the framework does not support monitoring the status of a data flow.

Gupta et al. [26] used IoT devices in monitoring and controlling electrical devices, primarily focusing on the improvement of performance and their quick fault detection. The IoT devices are also used in home automation by authors in [27]. Vishwakarma et al. [27] focus on minimizing energy consumption while automatizing home activities through different IoT devices. The IoT devices are also used in smart communities [28]. Automation action can further be applied in the agriculture system using IoT devices as proposed by Puranik et al. [29].

An architecture in [22] is proposed to collect, integrate, and enrich the city-wide data collected using thousands of IoT devices. The pre-processed enriched data are republished worldwide as linked data. The architecture also takes care of the missing values and applies a statistical method to fill up those missing values. However, the proposed data pipeline may not be enough to support real-time data. Further, the architecture does not consider the supported computing environment. As a result, this may not be able to handle a large volume of data.

Tang et al. [30] introduced a hierarchical distributed four-layer Fog-based computing paradigm for big data analysis in smart cities with a working prototype to integrate intelligence in fog computing architecture using a case study, smart pipeline monitoring system based on fiber optic and sequential learning algorithms (Hidden Markov Model). The work of the authors focuses on the protection of safety-critical infrastructures from environmental hazards for city-wide infrastructures. In the paper, however, authors miss addressing the challenges of data availability and validity.

Zhang et al. [31] present the work for dealing with big data on the internet of vehicles using cooperative fog computing, to avoid the latency of cloud. To support the efficient data handling of Internet of Vehicles (IoV), authors presented techniques like multi-source data acquisition, cooperative data acquisition, multi-path routing using fog computing for data availability. The author claims that it would reduce the amount of traffic and makes data handling efficient. In the paper, the authors presented an example of file downloading, specifying how it can improve data transmission. But the question of data validity and authorization remains unanswered, what if the data i.e. file downloaded contains invalid or corrupted data.

Cheng et al. [32] proposed a scalable data-centric programming model called fog function and a context-driven orchestration runtime system to enable serverless fog computing to overcome the limitation of fog computing to support dynamic service composition and inefficiency of Functions as-a-Service (Faas) for data-intensive IoT services. The authors also introduced orchestration mechanisms to leverages three types of contexts: data context, system context, and usage context. The authors presented the use case scenario of the proposed fog function with smart parking, which otherwise is difficult to achieve using a service topology programming model. The proposed fog function saves internal data traffic by 95% compared to cloud function and latency by 30% compared to edge function. The focus of it is mostly on bandwidth saving, scalability, and resource management. But unlike the proposed architecture in this paper, it does not possess all the flexibility as our model, which is capable of working with serverless as well as ensure data availability and validity, and is capable of handling large volumes of data.

Branowski et al. [33] presented a three-layer framework, Cookery framework that allows building a data analytics pipeline without knowledge of programming. The proposed framework has its Domain Specific Language (DSL) and supports Function-as-a-Service. Though it supports multiple sources, clearly, it lacks

the robustness that our proposed architecture provides and was developed for scientists without a complete understanding of programming.

Mujezinović et al. [34] proposed a fully serverless and infinitely scalable architecture that is based on producer–consumer patterns leveraging AWS Fargate technology. The proposed architecture for workflow scheduling makes use of docker containers as the worker node to avoid the constraints in the context of their execution environment, time or space, and also supports the high-frequency data, offering multi-cloud capabilities, end-to-end testing, debugging and error handling. The proposed architecture may not support the real-time requirement and also does not specify how data validity and authorization is maintained.

4. Coherent coordination of data migration and computation

In this section, a novel data pipeline framework CCoDaMiC (Coherent Coordination of Data Migration and Computation) is designed that provides the environment to handle the flow and processing of data generated by various environments such as smart city, smart vehicle, smart manufacturing industries, various types of Internet of Thing (IoT) sensors, etc. The main intention behind the design of the CCoDaMiC framework is to provide a coherent environment that would simplify the integration of both the data migration and data computation process, as shown in Fig. 1. Furthermore, the proposed framework is intended to be integrated into the model-driven DevOps framework, in regard to the management of flow of data within the service, in serverless computing [35].

In Fig. 1, the data generated needs two primary operations: storage and computation. Both the operations need assistance from each other, for which different platforms and concepts are introduced. For instance, for the storage of a massive amount of data, ETL (Extract, Transform, Load) concept is used, focusing primarily on three processes: Extraction, Transformation, and Load process. On the other hand, for computation, different computing paradigms such as cloud computing, fog computing, etc. evolved. Being one of the most popular computing paradigms, cloud computing successfully adopted the concept of utility computing. However, as more and more IoT sensors and devices are installed and connected to the internet for gathering the environmental, healthcare, industrial, and daily-life related data, a need for modern technology arises in the commercial and research community. As a result, the data pipeline concept is introduced that focuses on mainly handling the flow of data from one end-point to other end-points. Similarly, the cloud computing concept is extended, and the demands of computing are narrowed down to introduce serverless computing. To the best of our knowledge, both the data pipeline and serverless technologies are developed parallelly without any aim of converging at a certain point, where the developers can avail both the advantages.

This paper mainly focuses on converging the data pipeline and serverless computing, as shown in Fig. 1. The data migration/flow process ensures the developer and user that the data is available at the source or intermediary location. The proposed framework can further check the data accuracy by implementing different existing tools/algorithms such as Message Digest 5 (MD5) program. The framework is further made efficient and secure by incorporating the security features that verify the source of the data. CCoDaMiC mainly interacts with two entities: the external source of the data and the user, who defines what to do with the data, as shown in Fig. 2. The source of the data can be from any smart entities such as smart vehicles, smart city IoT sensors, smart industry, etc.

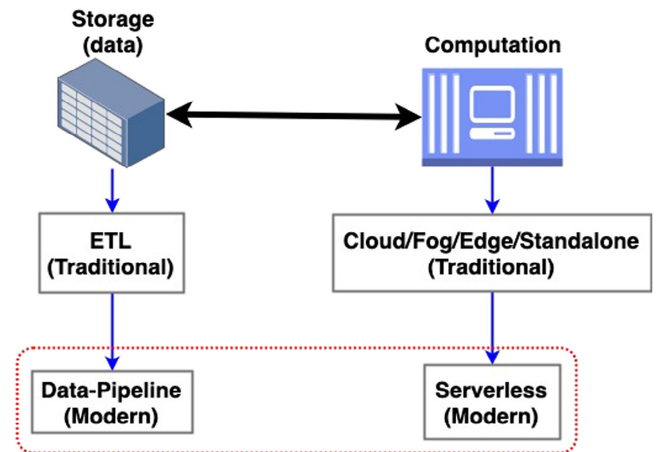


Fig. 1. The concerned areas that need to be focused.

4.1. Data source

The proposed CCoDaMiC framework is designed only to handle the data more securely and efficiently. It is assumed that the data are generated by the surrounding smart environments where thousands of dedicated sensors are deployed. The sensors can be fitted into a smart vehicle to determine the carbon emission by the vehicle, to alert lane-departure, notify if the seat-belt is not on, etc. Sensors in smart industries can be deployed to control the lighting system, establish robot communications, production automation, inside air-quality monitoring, etc. Such thousands of sensors generate a massive amount of data either on a regular time interval or based on specific events. A CCTV in the smart city generates large size data, whereas a humidity sensor generates the data of a few bytes in size. The data generated by the CCTV needs to be analyzed in a real-time manner, on the contrary, the data from humidity sensors can be pre-processed and analyzed in a batch manner. The proposed framework recognizes such types of data.

4.2. CCoDaMiC users

The users are mainly responsible for defining the flow and the processing of the data. All the components within the framework depend on the instruction given by the users through a template. The user's templates are aligned with the TOSCA standard. The TOSCA template contains the information regarding each pipeline, the communication among the pipelines, function metadata to process the intermediate data, information to validate and authorized the data, etc. The framework is enabled to validate and notify the users regarding the correction of the template. This will allow the to prohibit any pipeline interruption that may cause due to the bug within the template.

TOSCA: Topology and Orchestration Specification for Cloud Applications (TOSCA) [36] is a OASIS specification standard used to describe the management of complex applications. The entire application consists of multiple services or nodes. The communication topology of those services within the application is described in a template. The template is nothing but a YAML file. This also allows the developer to provide the management plan for each service, which refers to the general life cycle of the service, provisioning and migration plan of services, etc. The TOSCA template file does not contain the detailed information on the application rather only the meta information as described above.

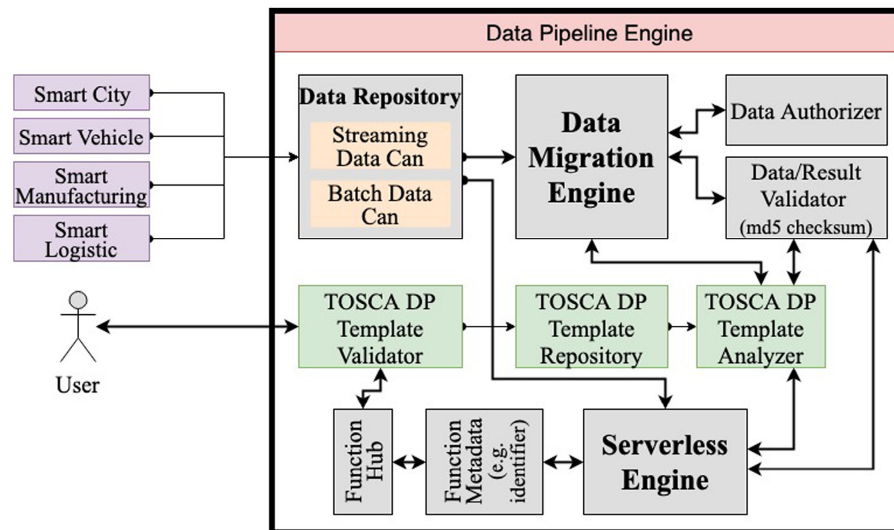


Fig. 2. Proposed CCoDaMiC architecture.

Fig. 2 shows the components of the proposed framework and interactions among them. The major components are the Data Migration Engine, Serverless Engine, Data Repository, etc. Data/result validator, data authorizer, template analyzer, resource analyzer, etc. act as the supporting components that are involved in transferring and processing the data. Data migration is responsible for migrating the data atoms or data packets from one location to others based on the TOSCA template given by the external user. The serverless engine is responsible for providing the computing environment to those data atom. The data repository provides the environment to store the data for both the data migration engine and serverless engine. The detailed description of those components is provided in the following sub-sections.

4.3. Data repository

This section describes the repository of the data upon which other components of the CCoDaMiC framework depend. As the name of this component suggests, the data repository component is responsible for keeping track of all the data atoms. The data atoms can either be batch type or stream type. This component interacts with external data sources, as described in Section 4.1, and acts as a bridge between other elements of CCoDaMiC and external data sources. The data may arrive from the sensors deployed in smart environments, such as smart industry sensors, smart vehicle sensors, air quality monitoring sensors, fire detection sensors, etc. Based on the type of sensors, the data can be batch or stream. Further, based on the nature of the data, two dedicated sub-components are designed: (a) Streaming Data Can, and (b) Batch Data Can. Data Cans are responsible for holding all incoming raw data atoms.

4.4. Data migration engine

Data Migration Engine, as shown in Fig. 3, is responsible for transferring the data from one location to another location. The data can be of batch or streaming type. The detail information regarding each flow of data is provided by the TOSCA template analyzer, which is described in Section 4.6. The detail information includes the source of data, the destination of the data amount of data to migrate, the time to migrate, etc. In order to carry out such tasks, this components consist of three sub-components: *Scheduler*, *Data Transmitter*, and *Data Receiver*, as shown in Fig. 3.

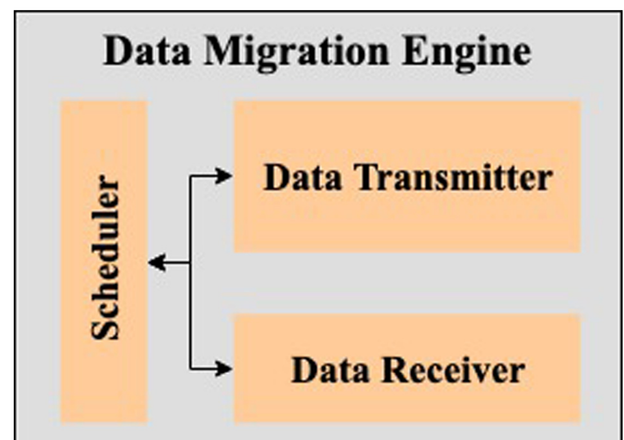


Fig. 3. Details of data migration engine and the interactions among sub-components.

Scheduler: This module mainly answers the question "when to migrate?". The information regarding the time to migrate is extracted from the user given TOSCA template, i.e., TOSCA template analyzer. Based on the type of data, the scheduler may initiate the migration process either on the arrival of data, or the scheduler may need to wait for a specific time interval to initiate the migration process. The initiation of the migration process includes sending the data from the source location and receiving the data at the final or intermediate destination from the remote source. This indicates that the data pipeline framework needs to be deployed on all the locations through which the data may flow. For example, if the data from a smart home needs to be migrated to a remote cloud environment, it needs to be migrated through edge device, gateway, fog environment(s), cloud router, and the specific server(s) as the final destination. In such a scenario, the proposed CCoDaMiC data pipeline framework may need to be installed/deployed on each location.

Data Transmitter: This module is responsible for extracting the required data from the data repository and migrating the data from the source location to the specified next destination. The data transmitter always works in conjunction with the scheduler. Copy of the data is always migrated from the source. This is to ensure that the correct data is delivered to the adjacent location.

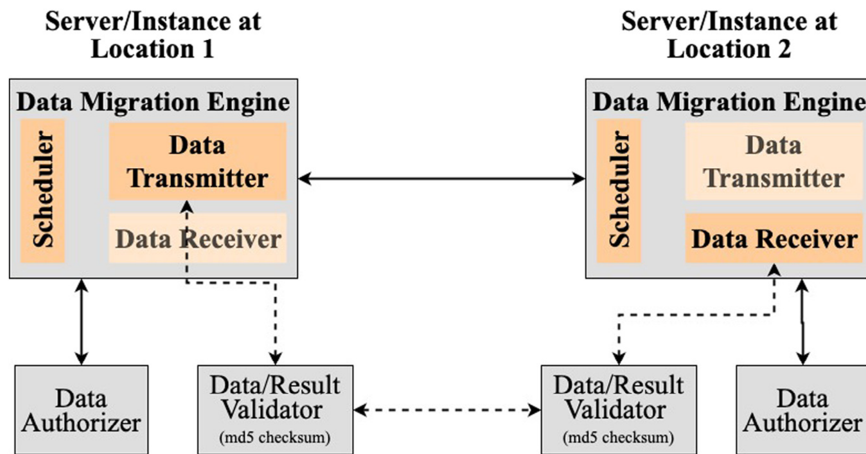


Fig. 4. An example of data migration between two instances at different locations.

On failure of any migration, this module sends another copy of the original. If the migration process is carried out successfully, the original data is deleted. In order to determine the status of the migration, the data transmitter interacts with the Data/result validator, which is described in Section 4.7.

Data Receiver: In conjunction with the data transmitter and the scheduler, the data receiver component is responsible for receiving the data sent by the data transmitter. Upon receiving, the data atoms need to be checked if the data is correct. This is done by the data/result validator, as described in Section 4.7. On failure of receiving accurate data, the data receiver discards the received data and listens to the data transmitter again to receive the accurate data. This process continues until the accurate data is received.

An example of data migration is shown in Fig. 4. Assuming that a set of data atoms need to be migrated from *Location 1* to *Location 2*, let the CCoDaMiC data pipeline framework is installed atop two responsible servers at both the locations. On initiating each data migration process, the data transmitter component at the *Location 1* (i.e., source location) is enabled, and that of the data receiver component is disabled. Similarly, the data transmitter is disabled, and the data receiver is enabled at the *Location 2* (i.e., destination location). On migrating all the specified amount of data atoms, the data validator at both the location will notify the resulting status to the corresponding enabled components. The migration process will be repeated until the accurate data is delivered to the server at the *Location 2* and checked by the data/result validator component.

4.5. Serverless engine

To provide an integrated environment for the purpose of both the migration and processing of data, CCoDaMiC framework is equipped with another major component, known as *Serverless Engine* with the objective to provide a seamless capability to process the data atoms. The data preprocessing artifacts are provided by the users in the TOSCA template, which is in YAML format. Serverless engine communicates with user-provided TOSCA template (see Section 4.6), data/result validator (see Section 4.7), the Function repository (see Section 4.8), and Data repository (see Section 4.3). Major functionalities of the serverless engine include deployment and execution of the function, requesting and leasing the required amount of resource, etc. Serverless engine consists of *Resource Demand Analyzer*, *Function Deployer*, *Function Executor*, *Resource Issuer*, *Credential Manager*, and the *serverless platform providers*, as shown in Fig. 5.

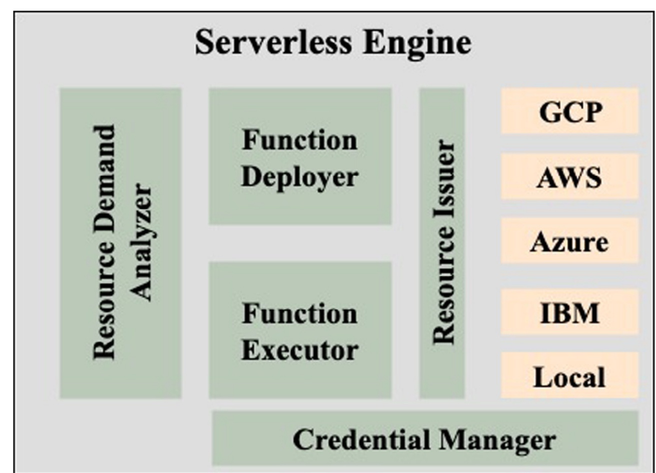


Fig. 5. Details of serverless engine and the subcomponents.

Deployment & execution of functions: As more and more tangible entities are connected to the internet, the data generated from those deployed sensors are increasing significantly. To handle such data with respect to its flow and processing, the modern data pipeline framework should be able to provide a seamless computing platform to pre-process the data as and when required. For processing, the users provide the function defining what and how to process. The function should be invoked from any location. *Function Deployer (FD)* is mainly responsible for the deployment of the function that is mentioned in the TOSCA template. In addition to the function itself, FD also requires the information on actual resource demand of the function. The actual resource demand may vary based on the input data that need to be processed. Further, using the actual resource demand, function code, and the actual data as input, FD deploys the function on the platform decided by the resource issuer. Upon deployment of the function, *Function Executer (FE)* is responsible for invoking the remotely/locally deployed function. The outputs of the function execution are received by the FE component, which is further forwarded to the data/result validator and eventually to the data migration engine.

Resource demand analyzer: As discussed in the above subsections, the resource demand for any function is highly dependent on the size of the input data. The resource demand analyzer is mainly a responsible component to carry out this task. Every

function deployment must go through the resource demand analyzer, where the actual amount of resource would be estimated based on the size of data to be processed. In some commercial serverless environments, registering a function requires the information on maximum resource requirement. In such a scenario, it is not possible to know the size of the unavailable input data. Hence a dynamic system is necessary, where the user's function registration should be done at the serverless environment provider after analyzing the data size and the basic resource demand of the function. The resource demand analyzer can fulfill the task of analyzing actual resource demand in such a dynamic system.

Resource issuer: This component plays a significant role as a bridge between Serverless providers and other components within the Serverless engine component. The resource issuer is responsible for providing the resources to the components responsible for the deployment and invocation process of user-defined functions. CCoDaMiC allows the users only to provide the function information such as its identifier and the information on the data that need to be processed. CCoDaMiC eliminates human intervention in the process of finding a suitable Serverless provider that can fulfill the demand of the function in terms of resources and the time. In addition to the functional requirements, the non-functional requirements, such as time constraints, cost constraints, location constraints, etc. are mentioned in the users' TOSCA template. Based on these requirements, the resource issuer chooses a suitable serverless provider and allows the FD and FE to deploy and invoke the execution process of the function.

Serverless platform provider: This is a core part of the serverless engine, which provides the computing resources to process the data. CCoDaMiC allows the users to attach any number of providers' information. The resource requests mainly sent by the resource issuer. Based on the functional and non-functional requirements, each provider needs to provide resource information.

Credential manager: Resource Issuer sends the resource requirement on behalf of the users to the resource provider. However, to deploy and invoke the functions, the credential manager provides all the related information that authenticates a user. The credential manager must implement a database of resource providers and the users' information.

4.6. TOSCA template

The proposed CCoDaMiC framework supports the modern standard, namely TOSCA, for deployment and management of the cloud application that is consisting of a large number of small microservices. TOSCA specification allows the users to provide the service-related functional and non-functional artifacts in the YAML format file. CCoDaMiC is consisting of three sub-components: *TOSCA DP template validator*, *TOSCA DP template repository*, and *TOSCA template analyzer*. The detailed descriptions of these three modules are given below:

TOSCA DP template validator: In order to avoid any interruption that may occur due to any undefined terminology in the data pipeline template itself, a validation step is incorporated into CCoDaMiC framework as TOSCA DP template validator module. There is a two-way interactive communication between the user and the validator module. The validating module notifies the users if the TOSCA template contains any undefined terminology or word. This also ensures that the function identifier (if any in the data pipeline template) is valid. In addition to the functions metadata, data pipeline may also provide the following information:

- Time to migrate data atom(s): This information notifies the scheduler the time to migrate the collected from one point to other end-point
- Amount of data to migrate: This adds a constraint for the scheduler while initiating any migration or computation process. For instance, the user may trigger the scheduler to migrate the temperature sensor data if the size is greater than or equal to 1 GB in the buffer.
- Source sensor type: Based on the type of sensor that generated the data, the scheduler may migrate the data to a specific location. For instance, the data from CCTV may need to migrate to a location that is equipped with computation resources.
- Data source environment: source environment refers to the location or environment from where the data is generated. for example, data from smart home, data from specific smart industry, etc.
- Non-functional requirements: Non-functional requirements such as cost constraint, location constraint, Service Level Agreement (SLA), etc. mainly used by the resource issuer component while choosing a suitable computing resource provider.
- Source end-point: This indicates the location from which the data should be migrated.
- Target end-point: This refers to the other end-point to which the data should be migrated.
- Process: This indicates if the data needs to be processed before moving to the next end-point.
- Source identifier: This information is used by the data authorizer to ensure that the data is coming from an authentic and genuine source.
- Function identifier: This points to the function which needs to be deployed and invoked in order to process the data.

TOSCA DP template repository: This component helps the user to keep the TOSCA DP template in a repository, which can be retrieved and used in the future. This also helps the users to reuse the templates. The repository can be set up in local or in a remote location. Instead of the user, the validator mainly pushes the valid error-free DP templates with a unique identifier.

TOSCA template analyzer: TOSCA DP template is mainly in a human-readable YAML format. The functional and non-functional requirements, as mentioned above, must be decomposed and forwarded to data migration and serverless engine. The TOSCA template analyzer acts as an instruction interpreter for two main migration and serverless engines. The information related to data migration, such as when to migrate, the amount of data to migrate, where to migrate, etc. are provided to the data migration engine. Further, the information, such as source identifier, are forwarded to the data authorizer to ensure that the data is authenticated and not corrupted. The template analyzer interacts with the data/result validator to verify if the processed data is aligned with a specific standard. For instance, in a data pipeline to handle large images, one DP block could be to resize the image to a specific resolution. The user can provide the resolution information of the resized image within the DP template, which can further be used by the data/result validator to validate the processed image provided by the serverless engine.

The TOSCA template analyzer assists the serverless engine while choosing a suitable serverless environment provider. Artifacts such as cost constraints, location constraints, etc. are provided to the resource issuer. The users' credential information is extracted from the template and forwarded to the credential manager by the TOSCA template analyzer, which can be used while accessing the resource from a chosen resource provider.

4.7. Data validation and authorization

One of the major advantages of the proposed CCoDaMiC data pipeline framework is its ability to ensure the correctness and

authenticity of the data without any human intervention. In order to achieve such functionalities, two dedicated components are designed: *Data Authorizer* and *Data/Result Validator*. The details of these two components are presented in the following subsections. *Data authorizer:* This component is mainly designed to ensure the other components of the framework that the incoming data is from the authorized source. This component must implement two tasks: authentication of the source of data atom(s) and other is the validation of the data integrity. To make the data migration more secure, an authenticated and secure channel needs to be established between two data pipeline blocks in two systems/locations. However, for data integrity, a checksum mechanism needs to be implemented, as described in the next subsection. The basic steps for data authorization is presented in Algorithm 1.

Algorithm 1: Algorithm for data authorizer

```
   Data: ipStream ← input data stream
   Result: Authorized output data stream
1  data ← convertToJSON(ipStream);
2  src ← data["source"];
3  srcList ← getSourceRegistry();
4  if src in srcList then
5  |  data["Authorization"] = "AUTHORIZED";
6  else
7  |  data["Authorization"] = "UNAUTHORIZED";
8  end
9  retData ← dumpJSON(data);
10 return retData;
```

Data/Result Validator: This component is mainly responsible for verifying the incoming data from other sources. A checksum method needs to be implemented to detect if the received data is equal to that of the sent data. This can be implemented for each data atom or a set of data atoms. Ming et al. [37] presented a MD5 based error detection method. In [38], a comparative study on different data error detection and correction methods is presented. Based on the comparative study, it is possible to correct the multiple-bit error on the received data. However, the proposed framework only detects the error and notifies the sender to initiate the data migration process. The basic steps for validating data is presented in Algorithm 2.

Algorithm 2: Data validator algorithm

```
   Data: ipStream ← input data stream
   Result: validated output data stream
1  data ← convertToJSON(ipStream);
2  hash ← md5(data["data"]);
3  if hash = data["hash"] then
4  |  data["validity"] = "VALID_DATA";
5  else
6  |  data["validity"] = "INVALID_DATA";
7  end
8  retData ← dumpJSON(data);
9  return retData;
```

In addition to that, the output of the processed data is also verified by this component. For this, the user needs to provide the artifact or the metadata to verify the result from any intermediate processing. For instance, an intermediate data pipeline block could be to preprocess a dataset by filling up the missing values through some statistical approach. The output/result of this DP block could be verified by checking if any value is missing in the dataset. The user's TOSCA DP template contains the method to verify such output, which is further used by the data/result validator. This is the main reason behind a two-way communication between the serverless engine and the data/result validator components.

4.8. User function

A huge amount of data is generated from a large number of sensors deployed in different smart environments. The existing DP architecture mainly focuses on the migration of data from one end-point to another. CCoDaMiC framework allows the users to bind the computing environment with the data migration environment tightly. This infers that the user needs to provide the function artifacts, which may include the location of the function or the function code itself, function identifier, function size, etc. In order to handle the user-submitted function, CCoDaMiC offers two components: *Function Hub* and *Function Metadata*.

The *function hub* is a repository of the user's functions. Each function should be uniquely identified and should be associated with the user's credentials. The physical existence of the function hub can be at a local system or any public code repository platform. The user's TOSCA DP template must contain either the identifier of the function or the function code itself. In the case of the function code within the template, the function needs to be registered and stored in the function hub. This would allow the user to refer to that function in the new DP template. If the template contains the function identifier information (instead of the function code), the responsibility of the function hub would be to assist the template validator on confirming the existence of the intended function. On communication with the serverless engine, particularly with the resource demand analyzer, the function metadata needs to be passed. The metadata may contain information such as the size of the function, resource demand of the function, the intended data that would be input to the function, etc.

5. Implementation

This section describes the detailed implementation of the proposed CCoDaMiC framework, that provides the environment to handle the flow and processing of huge amount of data generated by the dedicated sensing devices in smart environments such as smart city, smart vehicle, smart manufacturing industries, etc. The experimental setup is first described in Section 5.1, including the details of all the tools that are used, the type of data used, etc. Further, in Section 5.2, the detailed implementation is discussed that includes the implementation of each component of CCoDaMiC.

5.1. Experimental setup

The proposed CCoDaMiC framework in the current version uses different tools and technologies for implementation. This section is dedicated to the experimental setup. CCoDaMiC framework provides a platform to maintain the flow of data in an efficient manner. There are several ways, CCoDaMiC framework can be implemented, either from the very scratch or using the existing basic tools. In the current implementation, the proposed CCoDaMiC framework makes use of the available Apache Nifi [17] software package as a base. Apache Nifi is easy to use, browser-based and reliable platform to process and distribute data [17]. Apache Nifi provides basic functionalities such as sending data from one server/location to other, generating dummy data, converting basic file types, splitting basic data types, routing data, invoking HTTP/FTP protocols, etc. In this implementation, Apache

Nifi version 1.9.2 is installed atop instances in our OpenStack environment (detail specification is given below).

In order to store the data at a particular location, PostgreSQL version 12.0 [39] is installed atop instances in our OpenStack environment (detail specification is given below). This database is considered as a small-scaled data repository. For the implementation of such a database, container-based virtualization technology is used, such as Docker. The Docker Community Edition 18.0.9 is installed inside Ubuntu 19.04 Operating System (OS). The PostgreSQL Driver Version 42.2.8 was used for interacting with the PostgreSQL database. The PostgreSQL being opensource relational database is the reason for selecting it instead of other databases. Moreover, its scalability and adoption by cloud services like Heroku are also one of the reasons for choosing PostgreSQL. Similarly, a NoSQL based storage server, Couchbase Server Enterprise Edition 6.0.2, with two data nodes is used. To speed up our implementation task and take advantage of the Nifi Component, we choose Couchbase as our cloud-based storage, but we could use any other similar database. Python version 3.7 language is used along with Python Flask 1.1.1 and Flask-RESTful 0.3.7. To perform any other tasks like validation, authorization, data processing, etc., we require the help of some programming language. Python is popular among the community and its ability to perform our task with less programming compared to Java; we picked it as a choice of language for implementation. TOSCA specification v1.3 [36] is used by the user to describe the entire data pipeline. In other words, users provide the detail information about the data pipeline that includes (a) the information about every end-points data need to travel through, (b) information about the function to process the data, (c) the information on the constraints that can be used to validate the result from each function, and (d) the additional artifacts such as data input, when to invoke the function, etc. The TOSCA model is mentioned in a YAML file. The components are installed in different Openstack instances. Ubuntu 19.04 OS is installed atop each instance. Each instance is created with a minimum of 4 number of vCPUs, 4 GB of memory, and a minimum of 20 GB of the disk storage system. Considering the above experimental setup, the detailed implementation of each component is discussed in the following subsection.

5.2. Experiment discussion

CCoDaMiC framework allows the users to migrate the data from one location to another by tightening coupling the computing environments. This also facilitates secure data migration and ensure the data and output are valid and from the authorized sources. In order to achieve those goals, the implementation is divided into multiple phases, such as the implementation of data source and its repository, implementation of secure data migration with validation and authorization, implementation of computation (serverless) engine, etc.

5.2.1. Data repository implementation

In the implementation, the first step is to receive the data from the external sources. The data can either be batch or stream data. Instead of real sensing devices, we are generating similar data using Nifi by invoking the external python script. Nifi allows the users to generate the random data using the *GenerateFlowFile* processor. The data are generated at a specific time interval. By customizing the time interval, we are able to simulate the batch data and streaming data generation. The data are first generated and stored in files, where the size of the data files is kept constant at 5 Bytes. Further, to distinguish the batch data and streaming data, a batch of 1 data file will be considered as streaming data, whereas the batch of more than 1 data file is considered as batch data. Each data file may contain random text or may follow a specific pattern. However, in the implementation of CCoDaMiC framework, the content of each data file is kept constant and is used to initiate the task of data generation by sending a request to another Nifi component *InvokeHTTP*. This Nifi component *InvokeHTTP*, invoke external python script that generate random data. Considering the real implementation scenario where data sometimes might come from invalid source, we in our script also generate the invalid data source. Moreover, data in real implementation comes from sensor and communication from sensor is usually done via REST approach. To depict this situation the python script for data generation is implemented as a REST API using python FLASK and FLASKRestful. The deployment of it is done using a Docker container. Algorithm 3 provides the basic steps used in the random data generator.

Algorithm 3: Random Data Generator

```
   Result: Random data with hash value, timestamp, and
           source
 1 authorized_source ← {Set of sensing devices} ;
 2 src ← getRandSrc(authorized_source);
 3 randStr ← generateRandomString(size=1MB) ;
 4 strHash ← generateMD5hash(randStr);
 5 crrTime ← GetCurrentTimeStamp();
 6 generated_data = {
 7      "source": src,
 8      "data_gen_time": currTime,
 9      "hash": strHash,
10      "data": randStr,
11 };
12 return generated_data;
```

Further, each data file is preprocessed to convert the input JSON data to Nifi attribute. To achieve this, *EvaluateJsonPath* Nifi processor is used, as shown in Fig. 6. In order to implement the data repository, as discussed in Fig. 2, we have implemented PostgreSQL database. Instead of deploying this database directly inside the Openstack instance, we are using the container-based virtualization technology, i.e., Docker, inside the instance. This allows easy setup and portability of the PostgreSQL database. This database serves as the data repository to the CCoDaMiC framework. The preprocessed data originally generated by invoking python REST API using *InvokeHTTP* processor are further pushed to this data repository using Nifi *PutSQL* processor. Nifi *PutSQL* processor interacts with the remote containerized data repository by taking the help of the PostgreSQL driver as specified in the experimental setup. Below is the code snippet used in Nifi *PutSQL* processor to push the data to the remote data repository.

```
INSERT INTO repository
  (source, data, hashkey )
  VALUES (
    '${source}',
    '${data}',
    '${hashkey}'
  );
```

A screenshot of the database is presented in Fig. 7. The database contains information in mainly three different data fields: source, data, and a hash key of data.

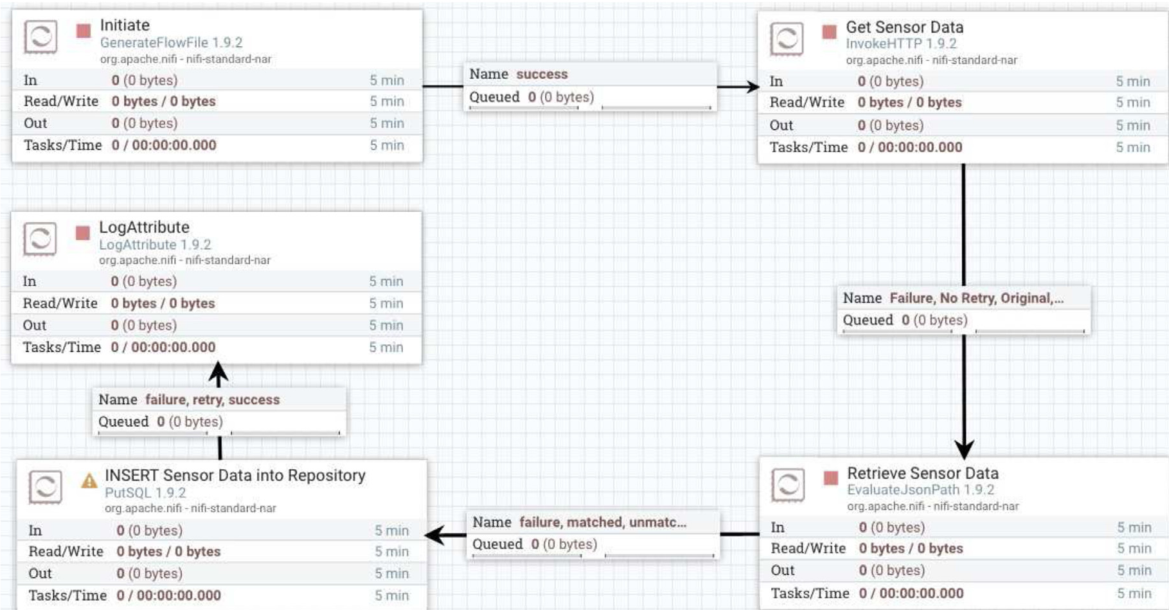


Fig. 6. Abstract architecture of implementation for data repository.

```
test=# CREATE TABLE repository(id serial PRIMARY KEY, source VARCHAR(250), data VARCHAR(250), hashkey VARCHAR(250));
CREATE TABLE
test=# select * from repository;
id |      source                   |                  data                            |          hashkey
---+-------------------------------+--------------------------------------------------+----------------------------------
1  | SENSOR_HUM00H212  | 9XLR13N9PK5CNR44LSM8AAFT1Z986VCDDYJTE1L2      | 791BDC88F7EC3C46E831347D64CD60E1
2  | SENSOR_HUM00H212  | WVFOC9EWOUTKSVP7U85ZU7MSN98V4IN10F0BXAX7      | 5B9A8FE9D2748F64DF3C99EC06E27FBB
3  | SENSOR_TEMP00X912 | 14EZT0JZYMCTSB6H8EK051SNQG7DCYMHP86MRNKQ      | AB04F3603E289BD8C60EBE496CD0ABC4
4  | SENSOR_HUM00H212  | DOD2CTNTZMTDXTT4Z61PXNNUY6SHYCLR3LZQHCH1      | C0B23411DC124D47681266DC98DAD794
5  | SENSOR_TEMP00X912 | 3KQZJ8K9R817TFDE45DO88EEGK2GVW003T1V61XC      | EAABF0BDEA05C6FF7FCF51F94E5C0A44
6  | SENSOR_HUM00H212  | V91SM94UTEWJQ2Q51K41Q1IMDYPMPMKTHVWVI676      | 6148A4EAF6C745C79867E92FA516250F
7  | SENSOR_HUM00H212  | 87I9WQYBHEM653CCRCQNYXZRQXZSQR6EEUXK5MIM      | 2397F269A5664E51E78DB8832E7F3BD1
8  | SENSOR_TEMP00X912 | KGOISLHLHTT7T1CCARAKXB0BRDXB3PAY2CG5EAOE      | 01FA5B548B1899DAAE166EF33432F74D
9  | SENSOR_CAM00S212  | ANG9W0TACZ4746VP302NGIN8PCBBGKHUSQH74P2J      | 3FED188BA012C732B3B239CEA9F9AB3F
10 | SENSOR_TEMP00X912 | 6OE5DUCBHOYXQ2121B12HTT0NUOZ4KYURLVFUEZI      | 1EEBF8590F9AF87E82C5AE684EDE7C4C
11 | SENSOR_CAM00S212  | 56Q37DETH1EYFO3LXSI75E5M4IVOJP9U5N69AZ5W      | BA02D1ABB9FF381C21092B7306ECC654
12 | SENSOR_HUM00H212  | 8MJ9LW61LACSEZJVWT9YKU69WEOQ0LRR0EYDPAES      | D8AA384FC1424878F6551826A1CA31B0
13 | SENSOR_HUM00H212  | GKTPZDDBPVSQ58SJ2SPTRG4Z1953DAQXXHH0F7IK      | 1DFAA08148552F9E1B8BAB88A9B3B27F
14 | SENSOR_TEMP00X912 | 4A4TFXOPFW9RD5094FCIGBJZMCQ3BL9QZ810ZBMV      | C1F746037D0229DB38B09EE2D3DAA9FF
15 | SENSOR_TEMP00X912 | QSHY9FY5X07KW2LK7873ZWYABUH6P5WIXN7BFDL4      | C3C90D896272C3AB4E2EDE1968794694
```

Fig. 7. A screenshot of data in CCoDaMiC framework data repository.

5.2.2. Implementing data migration engine with validation and authorization

Upon storing the data in the data repository, there are always two possible operations: (1) the data is migrated to another location or server, (2) the data is forwarded to the server engine, followed by a data migration engine. For data migration, based on the requirement, different NiFi processor can be used, such as *ExecuteSQL*, *RemoteProcessGroup*, etc. In the implementation of the CCoDaMiC framework, *ExecuteSQL* Nifi processor is used in different instances to migrate the data from a remote data repository. The implementation of this is shown in Fig. 8. In this case, *ExecuteSQL* Nifi processor uses DBCPConnectionPool. This provides information regarding communicating with the remote data repository. The basic information that is used for remote data repository connection includes URL to the database, database driver, the login credentials, etc. *RemoteProcessGroup* Nifi processor is another basic component available in Nifi to send the data from one server to other servers. The minimum amount of information the user needs to provide to *RemoteProcessGroup* Nifi processor is the hostname of the destination server and the protocol to be used to migrate the data.

The advantage of the CCoDaMiC framework can be realized with its ability to validate and authorize the source of the data. In the case of validation, the MD5 checksum of the incoming data is generated and compared with its original checksum, as shown in Fig. 8 and Algorithm 2. This task of checking MD5 checksum for validation is performed by the external python script. Before performing this task, the intermediate data transformation is carried out, converting the object form of data retrieved from the data repository to the JSON format. The reason for this is because handling JSON data is much easier than the object in our case. The data is then passed to the serverless engine. Serverless Engine receives all data, even if it fails the validation and authorization. We could have easily discarded those data, but we wanted to keep track of the invalid data as well. The invalid data are marked as invalid to identify it later. This becomes important in the case of the real environment where failure is evident. It is also critical from the security point of view, where it provides information about the data sources and data, both valid and invalid. This available information could be used to gain more insights by analyzing the data and helps to take the necessary effective security measures. On each migration, the validator component will be invoked. A dedicated python script is implemented that is mainly

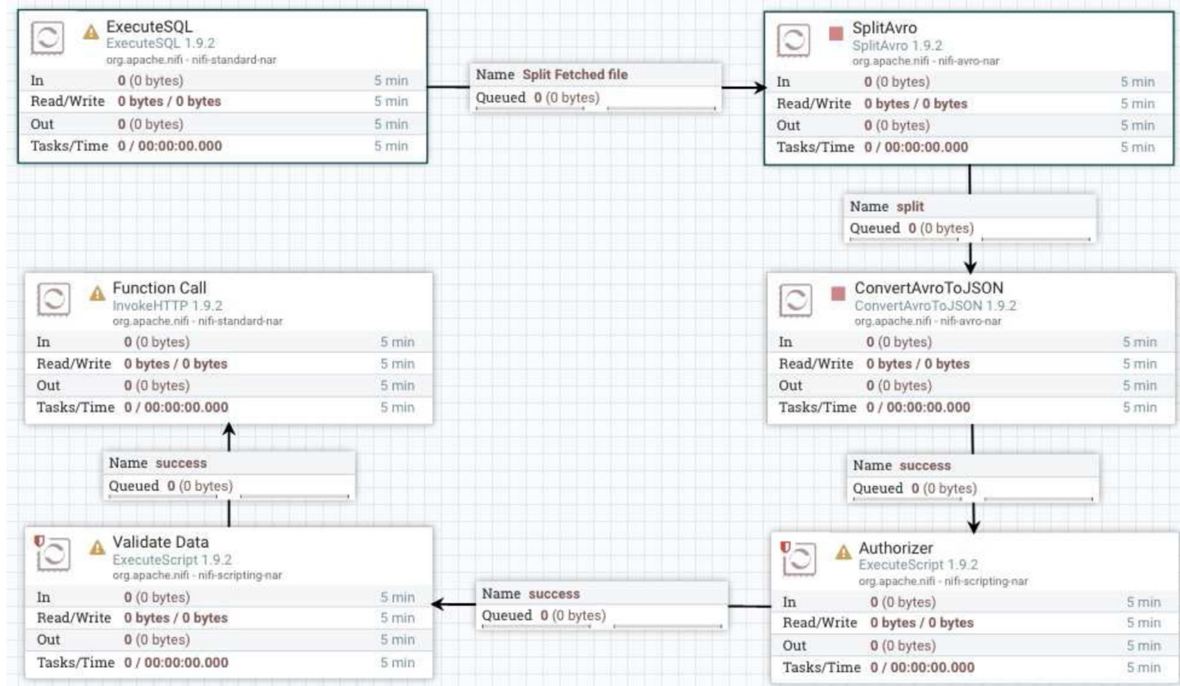


Fig. 8. Abstract architecture of implementation for retrieving the data from data repository.

responsible for validating the JSON data. Below in Algorithm 2 is the basic steps followed in validating the received.

The job of validator in Algorithm 2, is to compare the received hash value with that of the received data. The comparison result is further attached to the data for future reference.

For authorization of the data, additional information such as source device is included in the data. It is assumed that the data migration engine at each location has the synchronized registry of source devices. As on the current implementation of the CCoDaMiC framework, the data are authorized only by verifying the presence of the same source in the source registry. Algorithm 1, presents the basic steps followed in this implementation for data authorization purpose.

Based on the authenticity, the data either will be further forwarded to the next destination or will be handed over to the data migration engine. Based on the users' template, OpenFaaS function, or the serverless engine will be invoked.

5.2.3. Implementing serverless engine

The major component of the proposed CCoDaMiC framework is its ability to integrate the computing platform, especially the serverless environment, to the data migration engine, as shown in Fig. 9. On arrival of the data, based on the users' requirements and the resource demand, the predefined function will be invoked. The serverless engine consists of the capability to interact with external services like Google Cloud Services, Amazon Web Services, Microsoft Azure, and even local services. The interaction to external services is made for different purposes, and one such is storage. In our implementation, we have demonstrated this interaction capability by storing data in the Couchbase Server that is set up and run in a separate container using docker. In this implementation to demonstrate the ability of the serverless engine, we developed a function using python flask and FLASKRestful that accepts the data in the JSON format and modify the content of existing data and send the result back.

In the implementation, *invokeHTTP* Nifi processor is used to invoke a remote function, as shown in Fig. 9. This is a generic component, which can be used to invoke any remote function from any provider. The basic information required is the URL, method to invoke the function, etc. However, Nifi provides a dedicated processor to invoke the function of a specific FaaS provider. For example, in our implementation, *putLambda* Nifi processor is used to invoking the Amazon Lambda function. This required basic information, such as the lambda function name, the region of the function, and the credential information. Unlike, generic *invokeHTTP*, *putLambda* processor does not require any specific URL to invoke the lambda function. The standard HTTP communication with POST request is used to invoke the function. The received data is then stored in the two-node Couchbase Server that is setup using the docker and in our implementation acts as a remote cloud service. The latest TOSCA standard v1.3 [36] does not provide any specification for service lifecycle management with the serverless platform. However, in our implementation, we use basic information only to indicate if the data need remote function invocation.

One of the major challenges/drawbacks in serverless implementation is its cold start. This issue arises if the function is invoked for the first time or was idle for a longer period of time. Further, another major challenge in implementing serverless platform is ensuring that the function is available with the selected cloud provider. In case of unavailability of the function, it has to go through the deployment process, which contributes towards the cold start of the function execution. In the implementation, it is also hard to maintain the uniformity of function input and its performance due to diverse underlined platforms with the cloud providers.

5.2.4. Design of TOSCA model

The latest TOSCA specification v1.3 [36] provides a broader environment to model and design the entire lifecycle of a service in human-readable format. However this does not offer the users to design the model of data pipeline. In this paper, the model that is in YAML file, is referred to as Data Pipeline Service (DPS) template. Dedicated TOSCA models are designed to implement the data pipeline. Currently, the model mainly consists of two node types: (a) node type for configuring the Apache

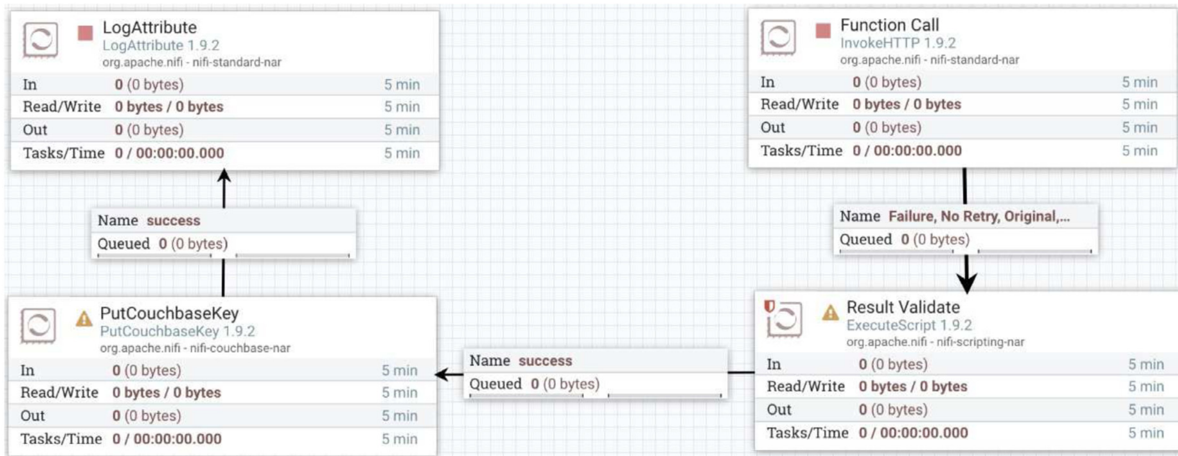


Fig. 9. Abstract architecture of implementation for serverless platform.

```
node_types:
  ccodamic.nodes.apache.nifi.Nifi:
    derived_from: tosca.nodes.SoftwareComponent
    properties:
      component_version:
        description: The version of Apache Nifi
        required: true
      port:
        description: The port exposed by Apache Nifi
        required: true
        default: 8080
    capabilities:
      host:
        type: tosca.capabilities.Container
        valid_source_types: [ ccodamic.nodes.apache.nifi.NifiPipeline ]
    interfaces:
      Standard:
        type: tosca.interfaces.node.lifecycle.Standard
        create:
          inputs:
            tarball_version: { default: { get_property: [ SELF, component_version ] } }
          implementation: files/create.yml
        start:
          implementation: files/start.yml
        stop:
          implementation: files/stop.yml
        configure:
          implementation: files/configure.yml
        delete:
          inputs:
            tarball_version: { default: { get_property: [ SELF, component_version ] } }
          implementation: files/delete.yml
```

Fig. 10. TOSCA model for Nifi.

Nifi *ccodamic.nodes.apache.Nifi.Nifi*, (b) node type for implementing the lifecycle of the Nifi data pipeline, i.e. *ccodamic.nodes.apache.Nifi.NifiPipeline*.

TOSCA model for Nifi: Nifi node type i.e. *ccodamic.nodes.apache.Nifi.Nifi* is derived from the *SoftwareComponent* node type of the TOSCA, as shown in Fig. 10. This node type has two properties: component version and the port number which can be used to access from the remote host. Component version refers to the version of the Apache Nifi. The capability of this node type is hosting the data pipeline blocks that are of type *ccodamic.nodes.apache.Nifi.NifiPipeline*, which is described in later subsection. This follows the standard lifecycle offered by the current TOSCA standard. While creating the Nifi node, the only input user need to provide is the version of the Apache Nifi. In the

implementation of the proposed framework, Apache Nifi v1.9.2 is used.

TOSCA model for Nifi pipeline: The TOSCA model for Nifi pipeline is presented in Fig. 11: the first part is in Fig. 11a and the second part is in Fig. 11b. Each data pipeline must be of type *ccodamic.nodes.apache.Nifi.NifiPipeline*. In the current version of the implementation, as shown in Fig. 11a, each node of *ccodamic.nodes.apache.Nifi.NifiPipeline* type has *id* as the attribute. This mainly used to uniquely identify the data pipeline block. This value of the *id* is retrieved once the data pipeline block is created and deployed. Each data pipeline requires (a) the *host* information where the data pipeline needs to be created, and (b) the connection information, which infers that the data pipeline may need to connect to another data pipeline of the same node type. For this, we have use *ConnectsTo* relationship type. To establish the connection between two data pipeline, the node type must also provide the capability to connect to another data pipeline. The *connect* capability is derived from the TOSCA *EndPoint* capability type.

Each data pipeline node follows the lifecycle provided the TOSCA standard, i.e. *tosca.interfaces.node. lifecycle.Standard*, as shown in Fig. 11b. *template_file* and *template_name* are provided while creating a data pipeline. *template_file* refers to the xml representation of the data pipeline containing the detailed information on the data pipeline. *template_name* is the name of the template mentioned in the template file (in .xml format). To create a data pipeline, on a specific host/server, the template file is uploaded to the host. This file also contains the function name, which can be used to deploy and invoke remote function provided by the SaaS provider. Further, Nifi uses the template name to create and deploy the data pipeline block. Upon creation of a data pipeline, Nifi assigns an unique ID, which is further used to update the *id* attribute. The *id* attribute is used in further lifecycle stages, such as start, configure, delete, and stop. The complete source code can be found online in [40].

6. Performance evaluation

Upon successful implementation, we have evaluated the performance of the proposed CCoDaMiC framework. In doing so, we have mainly checked the execution overhead of the data validation and data authorizer components. The data migration time is observed with and without those two features/components.

Fig. 12a represents the average data migration time with data validation and authorizer component. The value in X-axis represents the number of data atoms and ranges from 1 through 10. The size of each data atom is kept constant at 1 MB. The values in Y-axis represent the average migration time of the data atom while migrating from the source to the final destination. It is observed that, the data validation and authorizer components introduce additional overhead to the data migration operation. However, it ensures the data veracity to the users, which infers the data accuracy and the trustworthiness. In Fig. 12a, it is observed that an average of 1.33 s is required to send 10 number of data atoms of total size 10 MB, when both the components are used in CCoDaMiC framework.

The additional overhead on time required to migrate the data can be realized by comparing with the time without those components. Figs. 12b and 12c represent the observed data migration time without data validation and authorizer components, respectively. It is observed that the average migration time decreases to 0.35 s and 1.21 s when a single data atom and 10 data atoms are migrated without the data validation component, respectively. Similarly, only a 3.7% decrease in migration time is observed while migrating 10 data atoms from source to destination without the data authorizer component, which is significantly less, as shown in Fig. 12c. In Figs. 12b and 12c, authorizer and validator components are used, respectively.

The similar effects of data validation and data authorizer components have been observed on the data migration engine while migrating multiple batches of data atoms in Fig. 13a. The number of batches in X-axis ranges from 10 to 100. The average size of a batch of data atom is kept constant at 10 MB. Y-axis represents the average migration time of the data from the source location to the final destination location. The unit for migration time calculation is second. It is observed that migrating 100 batches of data from source to destination with the data/result validator and the data authorizer components take an average of 5.33 s, as shown in Fig. 13a.

13b and 13c show the time required to migrate the data without data validator component and data authorizer component, respectively. By deactivating the data validator component, 100 batches of data atoms take an average of 3.6 s, as shown in Fig. 13b. Similarly, migrating the same number of batches of data atoms from source to the final destination takes 4.96 s, which is 6.9% less from 5.33 s, without the data authorizer component, as in Fig. 13c. However, in the absence of such components, CCoDaMiC framework cannot ensure the data veracity to the users' data, which is very crucial in implementing an efficient and secure data pipeline framework.

To fulfill the demand from the real-life applications the proposed CCoDaMiC framework provides a coherent environment that would simplify the integration of both the data migration and data computation process. From the small-scale implementation the architecture shown in Fig. 2, the scalability feature of CCoDaMiC can be observed as all the components are loosely coupled and can be configured with large number of other components.

7. Conclusions and future works

In this paper, the problem of integration of the data pipeline and computation platform is addressed. The modern data pipeline architectures are investigated, taking smart environments into consideration. It is concluded that the lack of essential features such as data validation and authorization facilities makes the modern data pipeline architectures not mature enough for modern smart environments. Upon investigating further, it is also observed that the existing architecture supports only the migration of data from one location to others and does not support the tight and smooth integration of the computation platform. Moving towards addressing such a problem, we have presented a novel framework CCoDaMiC (Coherent Coordination of Data Migration and Computation). Different components are carefully designed for secure data migration and providing a platform for the developer to process the data in the intermediate location before arriving at the destination.

Some of the major advantages of using CCoDaMiC framework are its ability to incorporate the serverless platform, which not only allows the developer to process the data but also provides a way to validate and authorize the source of the data. CCoDaMiC also introduces TOSCA extension for modeling the data pipeline. Different components are introduced that ensure the secure and accurate transmission of the data. The major disadvantage of the current version of the proposed framework is its dependency on the performance of the external cloud provider. Performance degradation of the cloud provider may bring huge computational latency to the entire data pipeline. One potential solution to this problem could be applying a fault-tolerant strategy, where multiple serverless platforms can be engaged to perform a single task.

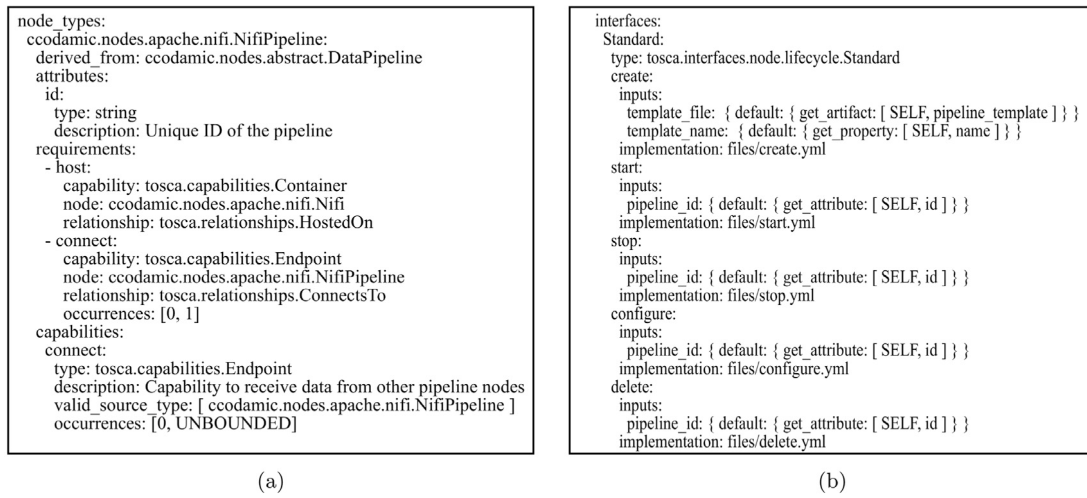


Fig. 11. TOSCA model for Nifi data pipeline.

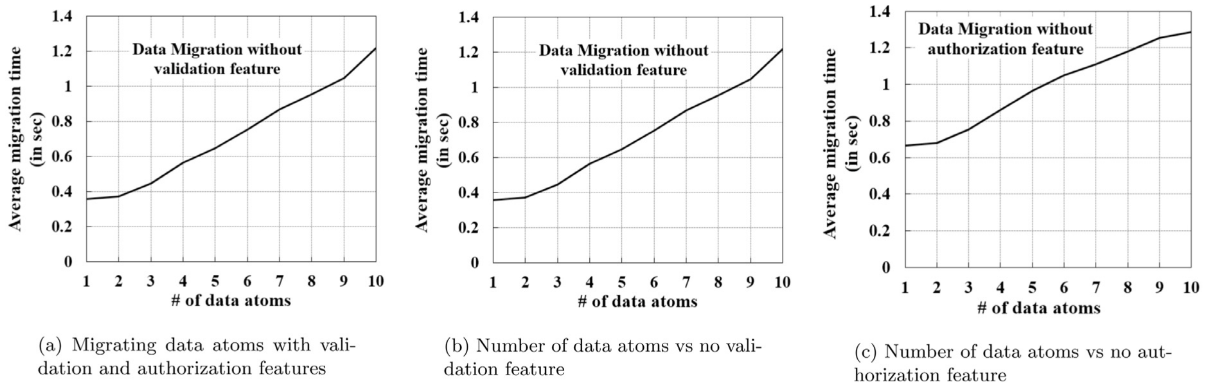


Fig. 12. Effect validation and authorization feature on migrating data atoms.

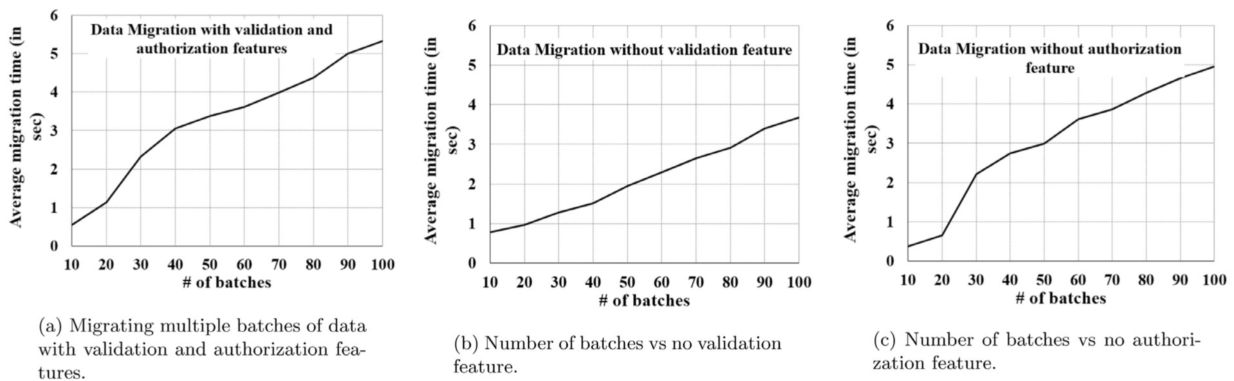


Fig. 13. Effect validation and authorization feature on multiple batches of data.

Further development of the TOSCA templates and the template analyzer to facilitates a way for the developer to manage the data flow and the serverless environment during the whole journey of each data atom would be a part of our future works. The future works also include the design of TOSCA template analyzer that can efficiently and accurately decide when to perform migration and computation operations. In future, it is also essential to evaluate the performance of the proposed CCoDaMiC framework in a large-scale experimental test-bed.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Chinmaya Kumar Dehury: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Resources, Writing - original draft, Writing - review & editing, Visualization. **Satish Narayana Srirama:** Supervision, Funding acquisition, Conceptualization, Methodology, Validation, Formal analysis, Writing - review & editing. **Tek Raj Chhetri:** Software, Validation, Formal analysis, Investigation, Data curation, Writing - review & editing, Visualization.

Acknowledgment

This work is partially funded by the European Union's Horizon 2020 research and innovation project RADON (825040).

References

[1] E. Mobility, Ericsson mobility report, 2018, https://www.ericsson.com/assets/local/mobility-report/documents/2018/ericsson-mobility-report-june-2018.pdf, [Online; accessed 21-October-2019].
[2] I. Lee, K. Lee, The internet of things (iot): Applications, investments, and challenges for enterprises, Bus. Horiz. (ISSN: 0007-6813) 58 (4) (2015) 431–440.
[3] S. Lohr, The age of big data, New York Times 11 (2012) (2012).
[4] Z. Ning, J. Huang, X. Wang, Vehicular fog computing: Enabling real-time traffic management for smart cities, IEEE Wirel. Commun. 26 (1) (2019) 87–93.
[5] A. Yassine, S. Singh, M.S. Hossain, G. Muhammad, Iot big data analytics for smart homes with fog and cloud computing, Future Gener. Comput. Syst. (ISSN: 0167-739X) 91 (2019) 563–573.
[6] A.P. Plageras, K.E. Psannis, C. Stergiou, H. Wang, B. Gupta, Efficient iot-based sensor big data collection–processing and analysis in smart buildings, Future Gener. Comput. Syst. (ISSN: 0167-739X) 82 (2018) 349–357.
[7] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, J.P. Jue, All one needs to know about fog computing and related edge computing paradigms: A complete survey, J. Syst. Archit. (ISSN: 1383-7621) 98 (2019) 289–330.
[8] C.K. Dehury, S.N. Srirama, Personalized service delivery using reinforcement learning in fog and cloud environment, in: The 21st International Conference on Information Integration and Web-Based Applications & Services (iiWAS2019), 2019, pp. 524–531, http://dx.doi.org/10.1145/3366030.3366055.
[9] R. Buyya, S.N. Srirama, Fog and Edge Computing: Principles and Paradigms, John Wiley & Sons, 2019.
[10] A. Sabtu, N.F.M. Azmi, N.N.A. Sjarif, S.A. Ismail, O.M. Yusop, H. Sarkan, S. Chuprat, The challenges of extract, transform and loading (etl) system implementation for near real-time environment, in: 2017 International Conference on Research and Innovation in Information Systems (ICRIIS), IEEE, 2017, pp. 1–5.
[11] V. Ranjan, A Comparative Study Between ETL (Extract, Transform, Load) and ELT (Extract, Load and Transform) Approach for Loading Data into Data Warehouse, Tech. rep., 2009, viewed 2010-03-05, http://www.ecst.csuchico.edu/~juliano/csci693.
[12] J. Meacham, M. Harris, G. Brodman, L. Cuthriell, H. Korus, B. Toth, J. Hsiao, M. Elliot, B. Schimpf, M. Garland, et al., History Preserving Data Pipeline System and Method, Google Patents, 2016, US Patent 9, 229, 952.
[13] J. Darmont, S. Loudcher, Utilizing Big Data Paradigms for Business Intelligence, IGI Global, 2018.
[14] S. Bhattacharjee, Practical Industrial Internet of Things Security: A practitioner's guide to securing connected industries, Packt, 2018.
[15] V. Kale, Big Data Computing: a Guide for Business and Technology Managers, Chapman and Hall/CRC, 2016.
[16] S.C. Öner, O.H. Yüregir, Optimizing Big Data Management and Industrial Systems With Intelligent Techniques, IGI Global, 2018.
[17] Apache nifi documentation, 2019, https://nifi.apache.org/, [Online; accessed 21-October-2019].
[18] Spotify/luigi, 2020, https://github.com/spotify/luigi, [Online; accessed 21-Jan-2020].
[19] What is aws data pipeline? - aws data pipeline, 2020, https://docs.aws.amazon.com/datapipeline/latest/DeveloperGuide/what-is-datapipeline.html, [Online; accessed 24-Jan-2020].
[20] P. O'Donovan, K. Leahy, K. Bruton, D.T. O'Sullivan, An industrial big data pipeline for data-driven analytics maintenance applications in large-scale smart manufacturing facilities, J. Big Data 2 (1) (2015) 25.
[21] H. Tianfield, Towards edge-cloud computing, in: 2018 IEEE International Conference on Big Data (Big Data), 2018, pp. 4883–4885.
[22] S. Bischof, A. Harth, B. Kämpgen, A. Polleres, P. Schneider, Enriching integrated statistical open city data by combining equational knowledge and missing value imputation, J. Web Semant. (ISSN: 1570-8268) 48 (2018) 22–47.
[23] J. Sampé, M. Sánchez-Artigas, P. García-López, G. París, Data-driven serverless functions for object storage, in: Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, in: Middleware '17, ACM, New York, NY, USA, 2017, pp. 121–133.
[24] A. Saha, S. Jindal, Emars: Efficient management and allocation of resources in serverless, in: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), IEEE, 2018, pp. 827–830.
[25] M. Sewak, S. Singh, Winning in the era of serverless computing and function as a service, in: 2018 3rd International Conference for Convergence in Technology (I2CT), IEEE, 2018, pp. 1–5.
[26] A.K. Gupta, R. Johari, Iot based electrical device surveillance and control system, in: 2019 4th International Conference on Internet of Things: Smart Innovation and Usages (IoT-SIU), 2019, pp. 1–5, http://dx.doi.org/10.1109/IoT-SIU.2019.8777342.
[27] S.K. Vishwakarma, P. Upadhyaya, B. Kumari, A.K. Mishra, Smart energy efficient home automation system using iot, in: 2019 4th International Conference on Internet of Things: Smart Innovation and Usages (IoT-SIU), 2019, pp. 1–4, http://dx.doi.org/10.1109/IoT-SIU.2019.8777607.
[28] P. Gupta, C.K. Dehury, P.K. Sahoo, Scheduling IoT data of smart communities in cloud, in: The 12th Workshop on Wireless, Ad Hoc and Sensor Networks, Nantou, Taiwan, 2016.
[29] V. Puranik, Sharmila, A. Ranjan, A. Kumari, Automation in agriculture and iot, in: 2019 4th International Conference on Internet of Things: Smart Innovation and Usages (IoT-SIU), 2019, pp. 1–6, http://dx.doi.org/10.1109/IoT-SIU.2019.8777619.
[30] B. Tang, Z. Chen, G. Hefferman, S. Pei, T. Wei, H. He, Q. Yang, Incorporating intelligence in fog computing for big data analysis in smart cities, IEEE Trans. Ind. Inf. 13 (5) (2017) 2140–2150.
[31] W. Zhang, Z. Zhang, H. Chao, Cooperative fog computing for dealing with big data in the internet of vehicles: Architecture and hierarchical resource management, IEEE Commun. Mag. 55 (12) (2017) 60–67.
[32] B. Cheng, J. Fuerst, G. Solmaz, T. Sanada, Fog function: Serverless fog computing for data intensive iot services, in: 2019 IEEE International Conference on Services Computing (SCC), IEEE, 2019, pp. 28–35.
[33] M. Branowski, A. Belloum, Cookery: A framework for creating data processing pipeline using online services, in: 2018 IEEE 14th International Conference on E-Science (E-Science), IEEE, 2018, pp. 368–369.
[34] A. Mujezinović, V. Ljubović, Serverless architecture for workflow scheduling with unconstrained execution environment, in: 2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), IEEE, 2019, pp. 242–246.
[35] G. Casale, M. Artač, W.-J. van den Heuvel, A. van Hoorn, P. Jakovits, F. Leymann, M. Long, V. Papanikolaou, D. Presenza, A. Russo, S.N. Srirama, D.A. Tamburri, M. Wurster, L. Zhu, Radon: rational decomposition and orchestration for serverless computing, in: SICS Software-Intensive Cyber-Physical Systems, 2019.
[36] C.N. Matt Rutkowski, Chris Lauwers, C. Curescu, TOSCA-simple-profile-YAML-v1.3, 2019, OASIS Committee Specification 01.https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/cs01/TOSCA-Simple-Profile-YAML-v1.3-cs01.html, [18 September 2019].
[37] H. Ming, W. Yan, Md5-based error detection, in: 2009 Pacific-Asia Conference on Circuits, Communications and Systems, 2009, pp. 187–190.
[38] J. Singh, J. Singh, A comparative study of error detection and correction coding techniques, in: 2012 Second International Conference on Advanced Computing Communication Technologies, 2012, pp. 187–189.
[39] PostgreSQL: Documentation: 12: PostgreSQL 12.0 Documentation, 2019, https://www.postgresql.org/docs/12/index.html, [Online; accessed 21-October-2019].
[40] Chinmaya Kumar Dehury, Satish Narayana Srirama, Tek Raj Chhetri, A framework for coherent coordination of data migration and computation platforms, 2019, https://github.com/chinmaya-dehury/CCoDaMiC_framework, [Online; accessed 31-October-2019].

Chinmaya Kumar Dehury received bachelor degree from Sambalpur University, India, in June 2009 and MCA degree from Biju Pattnaik University of Technology, India, in June 2013. He received the PhD Degree in the department of Computer Science and Information Engineering, Chang Gung University, Taiwan. Currently, he is a postdoctoral research fellow in the Mobile & Cloud Lab, Institute of Computer Science, University of Tartu, Estonia. His research interests include scheduling, resource management and fault tolerance problems of Cloud and fog Computing, and the application of

artificial intelligence in cloud management. He is an reviewer to several journals and conferences, such as IEEE TPDS, IEEE JSAC, Wiley Software: Practice and Experience, etc.

Satish Narayana Srirama is a Research Professor and the head of the Mobile & Cloud Lab at the Institute of Computer Science, University of Tartu, Estonia and a Visiting Professor at University of Hyderabad, India. He received his Ph.D. in computer science from RWTH Aachen University, Germany in 2008. His research focuses on cloud computing, mobile web services, mobile cloud, Internet of Things, fog computing, migrating scientific computing and enterprise applications to the cloud and large scale data analytics on the cloud. He is an IEEE senior member, was an Associate Editor of IEEE Transactions in Cloud Computing, is an Editor of Wiley Software: Practice and Experience, a 50 year old Journal, and a program committee member of several international conferences and workshops. Dr. Srirama has co-authored over 140 refereed scientific publications in several international conferences and journals. For further information of Prof. Srirama, please visit: http://kodu.ut.ee/~srirama/.

Tek Raj Chhetri received a B.Tech degree in Computer Science & Engineering from SSN College of Engineering & Technology affiliated to Jawaharlal Nehru Technological University Kakinada, AP, India in 2015. He is currently a Computer Science master's student at the University of Tartu and expected to graduate in June 2020. His interests include distributed systems, Fog & Edge Computing, Cloud computing, and Intelligent Transportation Systems.