A Scalable Data Integration Architecture for Smart Cities: Implementation and Evaluation

Murilo B. Ribeiro, Kelly R. Braghetto

Department of Computer Science – Institute of Mathematics and Statistics
University of São Paulo (USP)
{muriloribeiro,kellyrb}@ime.usp.br

Abstract. The collection, processing, and analysis of data generated by varied sources can help us better understand the functioning and demands of the cities. However, developing efficient solutions to explore urban data is challenging due to the large volume, heterogeneity, and lack of accessibility and integration of this kind of data. In this work, we identify the main requirements of a data integration system to support decision-making in cities, focusing on its challenges. We analyze some existing data integration solutions, to uncover their features and limitations. Based on these results, we propose a new microservice architecture to support the development of software platforms for integrating smart cities' heterogeneous data and a guideline to assess their performance. We also present details of a proof-of-concept implementation of the proposed architecture and its performance evaluation. The results demonstrate that the platform can scale horizontally to handle the highly dynamic demands of a smart city while maintaining low response times.

Categories and Subject Descriptors: H.2.8 [**Database Management**]: Database Applications; H.3.4 [**Information Storage and Retrieval**]: Systems and Software; H.3.5 [**Information Storage and Retrieval**]: Online Information Services

Keywords: Smart Cities, Data Integration, Data Management

1. INTRODUCTION

The smart cities concept has emerged from the idea of applying information and communication technology to make cities more efficient. There is no clear and consensual definition for smart cities, but one of their main characteristics is the use of technology to promote environmental sustainability and improve the quality of life for citizens [Albino et al. 2015; Al Nuaimi et al. 2015].

The urban data, i.e. data gathered in the cities, has a paramount importance in the building of smart cities. The increasing availability of electronic devices with sensing capacity and computational power, capable of receiving and sending information, causes a large amount of data from different sources and in different structures to be continuously produced in cities. Cities also accumulate data generated by government entities, citizens and systems.

[Zheng et al. 2014] classify urban data into five categories: (i) urban mobility data such as traffic data [Yue et al. 2016], displacement, and mobile telephony; (ii) geographic data such as information about the transport network, road network, and areas of interest; (iii) social media data such as text, photos, and videos; (iv) environmental data such as those from weather and energy consumption; and (v) data from other sources not necessarily related to the urban context such as public service, Health [Klemm et al. 2016], and Economy.

This research is part of the INCT of the Future Internet for Smart Cities funded by CNPq proc. 465446/2014-0, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, FAPESP proc. 14/50937-1, and FAPESP proc. 15/24485-9.

The collection, cleaning, integration, transformation, and analysis of large amounts of data generated by different sources can help us to have a better understanding of the deficiencies of cities. Urban data can be used to assist the evidence-based decision-making and the development of public policies aimed at making the best use of the available resources and the improvement in the quality of life of the population. For example, the crossing of records of the municipal health department with sociodemographic, meteorological, and social networks data can be used to monitor and prevent the evolution of endemic diseases such as Dengue.

In order to gather and extract value from urban data, smart cities rely on sophisticated hardware and software infrastructures. [Silva et al. 2018] described these infrastructures as a layered architecture where the lowest layer, *Sensing Layer*, refers to the available sources, detection and data collection. The second layer, *Transmission Layer*, is responsible for the data transmission using wireless technology (3G, 4G, 5G, Bluetooth), wired or via satellite. The third layer, *Data Management Layer*, is considered the brain of any smart city as it is responsible for the storage, organization, manipulation, and analysis of data; event management; and decision making. The fourth layer, *Application Layer*, is responsible for providing services to citizens based on the data management layer.

The development of solutions to explore urban data faces difficulties due to the lack of accessibility and integration of data [Raghavan et al. 2020]. This happens because many systems are built in silos: they are closed and developed for specific needs. Data integration systems for smart cities have already been the subject of several research works [Psyllidis et al. 2015; Consoli et al. 2015; Cheng et al. 2015; Rathore et al. 2016; Hashem et al. 2016; Costa and Santos 2017; Mehmood et al. 2019]. Despite these works present some approaches for the different stages of data integration (such as ingesting, processing, storing, analyzing, and visualizing data), there are still open issues and place for improvement in such systems. Examples of issues are the insufficient support for metadata management and the lack of data query facilities for non-specialist users, which makes the discovery and the reuse of urban data more difficult.

In this work, we analyze some of the data integration solutions for smart cities reported in scientific literature, to uncover their features and limitations. Two main research questions have guided our analyses:

(1) What are the main challenges and issues identified by the researchers for data integration in smart cities?
(2) What are the functional and non-functional requirements of a software platform for data integration in smart cities?

Based on this analysis, we propose a microservices architecture to guide the development of software platforms for integrating smart cities' heterogeneous data and facilitating its use. This architecture was designed to support all the required services (i.e. data ingestion, metadata management, data processing, data analysis, and data visualization) while providing scalability, availability, security, and privacy. We also present a guideline to assess performance of systems that implement the proposed architecture. The guideline follows the Cloud Evaluation Experiment Methodology (CEEM) [Li et al. 2013], used for systematically evaluating cloud services' performance through experiments.

This article is an extended version of the work presented in [Ribeiro and Braghetto 2021]. Here, we present as new contributions the description of a proof-of-concept implementation of the core microservices of the proposed architecture and results of their experimental performance evaluation. The results showed that the architecture keeps response times and latency in acceptable ranges and is capable of scale horizontally to handle a varying workload.

The remainder of this paper is organized as follows. Section 2 analyzes the related works, identifying the requirements of the data integration platforms. Section 3 introduces our microservices architecture, providing details of each one of its services. The guideline for the performance evalu-

ation of implementations of the architecture is presented in Section 4. We present in Section 5 the implementation details of the proof of concept and the results of the load and scalability experiments. Finally, Section 6 presents the concluding remarks.

2. SOFTWARE PLATFORMS FOR DATA INTEGRATION IN SMART CITIES

This section discuss some representative research works that present different software approaches to data integration in smart cities. First, we describe the existing platforms and systems. After, we present the functional and non-functional requirements extracted from the analysis of these works, and discuss the challenges and issues identified.

2.1 Related Work

We have searched Google Scholar[1] for works published from 2015 until 2021 using the search string: (("data integration" or "semantic data integration" or "data warehouse" or "data lake" or "big data") and "smart cities"). This search string covers both conceptual works (i.e., discussing approaches and proposing data models) and practical works (i.e., that present system architectures and tools). Among the found practical works, we observed that most are domain-specific solutions for some smart city application. Despite the large number of publications related to data integration in smart cities, only a few works present general-purpose data integration software solutions that can support the various types of applications a smart city may have. This section analyzes the general-purpose solutions only.

We have analyzed the abstract, keywords, and the introduction (when necessary) of each of the returned papers to filter those that present software architectures to integrate heterogeneous data in smart cities. From this filtering, we reached the works of [Psyllidis et al. 2015], [Consoli et al. 2015], [Cheng et al. 2015], [Rathore et al. 2016], [Hashem et al. 2016], [Costa and Santos 2017], and [Mehmood et al. 2019]. Most of them developed distributed, multi-tiered systems, capable of handling data both in batch and real time, and supporting a large variety of services for applications and final users.

[Psyllidis et al. 2015] developed SocialGlass, a web platform that offers resources for analysis, integration, and visualization of heterogeneous urban data in order to assist in urban planning and decision-making. The SocialGlass architecture is divided into three main modules: ingestion, integration and exploration. The ingestion module refers to the acquisition, cleaning, and processing of social and sensor data. The integration module is responsible for enabling interoperability between different data sources. To achieve that, an ontology-based knowledge representation model was developed, which represents urban systems, the relationships between them, and the corresponding data sources. The exploration module offers a map-based web interface for data visualization and exploration, making it possible to obtain insights about spatial and temporal parameters of the urban context. Details of the technologies used in the implementation were not provided. The system does not support data processing and access via external platforms.

[Consoli et al. 2015] presented an ontology integration approach using Linked Data (a set of practices for publishing and connecting data on the Web). In their work, each dataset was converted to an RDF (Resource Description Framework) data model using custom processes. With the help of domain experts, ontologies were generated for each dataset, to achieve conceptual interoperability. Data and ontology are accessible by querying the SPARQL API. The ingestion, processing, and visualization of data were out of the scope of their work.

[Cheng et al. 2015] proposed a Big Data architecture integrated with the IoT SmartSantander experimental test environment. This architecture is divided into four main modules: (1) data collection,

[1] `https://scholar.google.com/` (Retrieved: 05/20/2022)

(2) data storage, (3) data processing and analysis, and (4) API for communicating with external applications. The data collection module is represented by a broker, which is responsible for receiving data from different sources. A NoSQL database is used for data storage. Data processing and analysis are done in batch or stream by using a distributed computing tool. The module for communicating with external applications has a RESTful API to allow external applications to make simple queries, complex queries, and subscriptions. A simple query might request aggregated results about the latest status of all sensors, while a complex query might request aggregated results about historical data within a specified time frame. Subscription is the mechanism used for apps to receive notifications with the latest results, preventing the apps from querying the data all the time. This architecture does not feature a visualization module for stored data and metadata, making it difficult for public managers to use it.

[Rathore et al. 2016] proposed a system for collecting, aggregating, filtering, sorting, pre-processing, computing, and decision-making using the *Data Lake* approach combined with *Data Warehouse*. The proposed system is divided into four layers: (1) data generation and collection, (2) data transmission, (3) data management and processing, and (4) data analysis. The first and second layers are responsible for collecting data using sensors and transferring the data to the storage platform; therefore, they are in a lower level than the services considered in this work. The third layer, data management and processing, use a distributed file system (HDFS – Hadoop Distributed File System) and distributed computing tools for real-time data processing. For historical data, the authors suggested the use of a tool for Data Warehouses (Apache Hive[2]) and distributed databases (Apache HBase[3]). The fourth layer is composed of several applications, each one for a different type of planning. The architecture does not support metadata management and data analysis.

Similarly, the smart cities Big Data architecture proposed by [Hashem et al. 2016] is divided into four layers. The first is composed of sources and transferring of data, while the second is responsible for storing the data in a distributed and fault-tolerant database. In the latter, the stored data is processed according to the queries received using a parallel and distributed processing programming model. The third layer, intelligent analytics, was designed to support the use of machine learning and data mining to extract patterns and knowledge from large amounts of data. The last layer is made up of applications that use the stored data for varied purposes, such as intelligent management of public resources.

[Costa and Santos 2017] presented an approach to design and implement a Big Data Warehouse in the context of smart cities, with a repository that stores data in raw format. The proposed architecture is divided into four major modules responsible for data collection, preparation and enrichment, storage and access, analysis and visualization. Data can be collected in real-time using a broker (Apache Kafka[4]), or in batch using an ETL tool, e.g. Talend[5] and HDFS Upload. The data collected in batch is directly stored in files on a distributed system, prepared and enriched using a distributed computing tool (Apache Spark[6]), and then stored in a *Data Warehouse* (Apache Hive). Data collected in real time is also stored in a distributed file system, prepared and enriched using a distributed computing tool, and later stored in a distributed database (Apache Cassandra[7]). The stored data can be accessed by a distributed SQL query tool (Presto[8]) and by a data visualization tool. The proposed model was implemented in the SusCity research project and was used to analyze data collected in the city of Lisbon. The solution does not support metadata management and accessing data via external platforms.

Similarly, [Mehmood et al. 2019] proposed an architecture divided into five modules responsible for data collection, ingestion, storage, exploration and analysis, and visualization. For data ingestion,

[2] `https://hive.apache.org` (Retrieved: 05/20/2022)
[3] `https://hbase.apache.org` (Retrieved: 05/20/2022)
[4] `https://kafka.apache.org` (Retrieved: 05/20/2022)
[5] `https://www.talend.com` (Retrieved: 05/20/2022)
[6] `https://spark.apache.org` (Retrieved: 05/20/2022)
[7] `https://cassandra.apache.org` (Retrieved: 05/20/2022)
[8] `https://prestodb.io` (Retrieved: 05/20/2022)

they proposed the use of a stream processing tool (Apache Flume[9]) with storage in a distributed file system (HDFS). Data analysis and exploration were performed using a distributed indexing tool (Apache Solr[10]) and distributed computing (Apache Spark). For data visualization, a SQL query web tool (Hue[11]) and the Matplotlib[12] library were used. The presented metadata management requires data uniformity, making it difficult to analyze the data from the different sectors of the city.

2.2 Requirements for Data Integration Software Platforms

2.2.1 *Functional Requirements.* The main goal of a platform for data integration in smart cities is to facilitate the development of applications that use data combined from different sources. To this end, most of the analyzed platforms implement requirements for data ingestion, processing, analysis, visualization, and data sharing. Table I provides an overview of how the related works cover these functional requirements. Each requirement is described in the sequence.

Table I. Functional Requirements

	Ingestion	Metadata	Processing	Machine Learning	Analysis and Visualization	External Access
[Psyllidis et al. 2015]	X	X			X	
[Consoli et al. 2015]		X				X
[Cheng et al. 2015]	X		X			X
[Rathore et al. 2016]	X		X	X		
[Hashem et al. 2016]	X		X	X		X
[Costa and Santos 2017]	X		X		X	
[Mehmood et al. 2019]	X	X	X		X	

Data ingestion is the process of importing real-time or batch data into the storage platform. Data can come from different sources, in different formats, such as CSV, TXT, JSON, and others.

Metadata management is the process of collecting and managing information about data stored on the platform. Metadata must contain information about the semantics and structure of data collections. Metadata should also keep information about the mappings needed to standardize data and guarantee backward compatibility, in order to enable data integration and ease of use.

The data arriving at the platform may be inaccurate, incomplete, inconsistent, or redundant. Additionally, this data may need aggregation, filtering, or analysis before enabling knowledge discovery. Thus, platforms must offer resources for creating and executing **data processing** procedures.

Extracting knowledge and insights of data is of paramount importance to enable better decision-making in cities and support the implementation of efficient public policies. Therefore, data integration platforms must enable the creation, maintenance, and execution of custom **machine learning** models.

Data analysis and visualization refer to the presentation of data in user-friendly graphical formats, to help users understand the behavior of cities and the use of resources. A data integration platform must offer features to support the creation of custom reports and dashboards for managers to convert data into knowledge.

The **external access** refers to the possibility of external systems consuming the data stored on the platform, allowing the development of new applications to improve the services provided to the

[9] `https://flume.apache.org` (Retrieved: 05/20/2022)
[10] `https://solr.apache.org` (Retrieved: 05/20/2022)
[11] `https://gethue.com` (Retrieved: 05/20/2022)
[12] `https://matplotlib.org` (Retrieved: 05/20/2022)

population or to optimize the use of available resources. Only authorized users or systems should have access to the data.

2.2.2 *Non-functional Requirements.* Table II shows the non-functional requirements mentioned in the related work of Section 2.1. We describe each one of the requirements in the sequence.

Table II. Non-functional Requirements

	Scalability	Availability	Security and Privacy
[Psyllidis et al. 2015]	X		
[Consoli et al. 2015]			
[Cheng et al. 2015]	X		
[Rathore et al. 2016]	X		X
[Hashem et al. 2016]		X	
[Costa and Santos 2017]			X
[Mehmood et al. 2019]	X	X	

Scalability refers to the ability to increase or decrease computational resources according to the need of the system. Scalability can be vertical, meaning adding (removing) resources to (from) a single node; or horizontal, when adding (removing) nodes to (from) a distributed system.

Availability refers to the ability of the platform to be resilient to hardware, software, and power failures to keep services available for as long as possible.

Security and privacy refer to restrict access to the stored data to authorized users and systems, preventing leakage and misuse of information. It also includes the platform's ability to comply with data protection policies so that sensitive data is properly anonymized and secured.

The implementation of these requirements imposes challenges due to the large volume of data and the heterogeneity of sources and formats. According to [Cheng et al. 2015], scalability is an important issue as the amount of data greatly increases over time, with the availability of new services and technology, and also with the population growth. Security and privacy are important issues as well since there is a lot of sensitive data being collected in smart cities, and the cyber attack attempts become each day more frequent [Cheng et al. 2015; Hashem et al. 2016].

3. A MICROSERVICE ARCHITECTURE FOR DATA INTEGRATION IN SMART CITIES

A microservice architecture is a distributed application where all its modules are small services, each of them being a cohesive, independent process that communicates with others via lightweight mechanisms [Dragoni et al. 2017].

We adopted the microservice architectural style in the development of the system architecture proposed in this work in order to achieve scalability and evolvability. Moreover, we have followed the same set of design principles adopted and evaluated in the development of InterSCity, an open-source platform for smart cities [Del Esposte et al. 2017; Del Esposte et al. 2019]. In the following, we briefly describe these principles, which are aligned with the microservices patterns [Taibi et al. 2018]:

—Modularity: consists of dividing the system into smaller functional units (microservices) that communicate using lightweight APIs;

—Distributed models and data: each microservice has its own database and models, allowing the use of the technology that best adapts to each context;

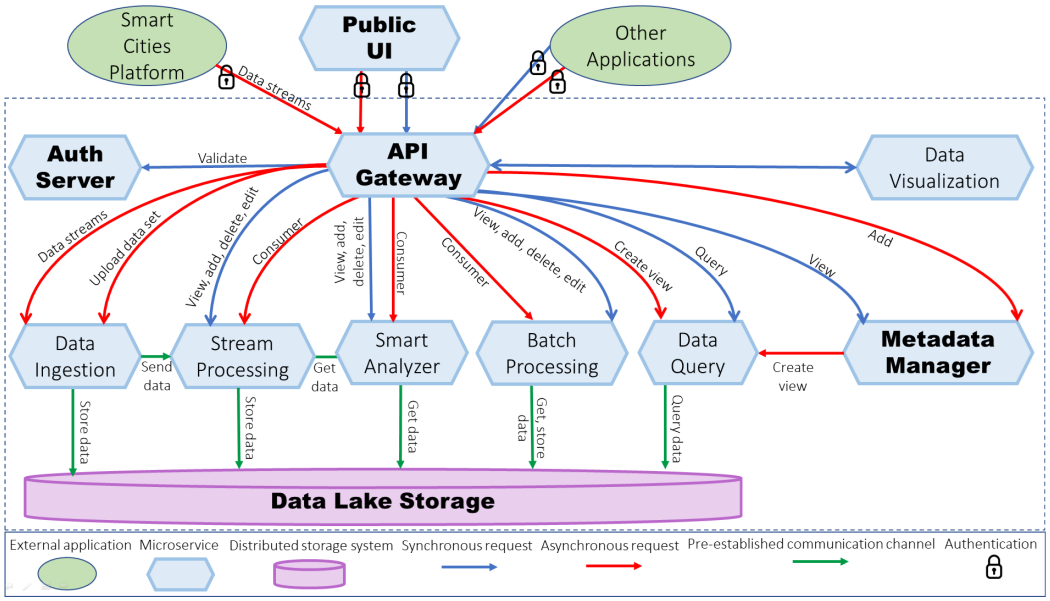

Fig. 1. The proposed software architecture for data integration in smart cities. Microservices with features not provided by the solutions of the related work are highlighted in bold.

—Decentralized evolution: microservices must be autonomous, with well-defined boundaries and communication APIs so that they can evolve and be maintained independently. Microservices must be able to scale independently, possibly using different strategies;

—Reuse of open source projects: reusing software components allows to increase software productivity, economy and reliability;

—Adoption of open standards: open standards aim to provide interoperability at different levels, avoiding technology and vendor lock-in;

—Asynchronous communication: whenever possible, services should use asynchronous communication, to avoid blocking in synchronous request-response interactions, provide low latency, and allow scalability. This can be made through notifications, the publish-subscribe design pattern, and event-based communication strategies;

—Stateless service: services should be stateless to allow any service instance to respond to any request, facilitating load distribution, elasticity, and scalability. State data, such as context and session data, should be separated to be managed by an external component whenever possible.

Considering the microservice architectural style and the design principles adopted, we divided the proposed system into a distributed file system (DFS) and seven microservices: Data Ingestion, Metadata Management, Data Query, User Management, and Public UI. Figure 1 shows the diagram of the proposed architecture. This architecture was initially derived from the architectures of related works, and then it was significantly improved to address the requirements and issues discussed in Section 2.2.

3.1 Components of the Architecture

The lowest-level component in the proposed architecture is the **Data Lake Storage**, a distributed storage system responsible for storing data as close as possible to its original format, so that end-users do not need to understand details of how data is stored in order to be able to use it. To prevent the storage system from being a bottleneck for query engines, data must be stored in a standardized, compressed format that facilitates analytical querying and reduces storage size and cost, but information cannot be discarded or lost. The most used open-source distributed file system

is HDFS. Data can be standardized into Avro[13] or Parquet[14] format and compressed using Snappy or Gzip compression.

The **Data Ingestion** microservice is responsible for asynchronously consuming data from the message queues, converting the original data to a standardized and compressed file format, and then sending it to **Data Lake Storage** by using a pre-established communication channel. It is necessary to have at least two message queues, one for real-time data (i.e. from sensors and social networks) and one for batch data. The separation of queues is important to enable the prioritization of the ingestion of data in real-time and to improve the scalability of the system. This microservice can be implemented using Kafka, RabbitMQ[15] and Spark streaming.

The **Metadata Manager** is a metadata catalog with features for registering data sources and information about data schema, data origin, privacy policies, schema versioning, and data mapping rules. It is also responsible for requesting the creation or updating of the data view for the **Data Query** microservice whenever a new version of the metadata is created. The latter is important to make data from different schema versions compatible, so that even legacy data can be easily discovered and used by the authorized persons. The metadata model must follow the World Wide Web Consortium (W3C) standards. We suggest the adoption of standard models such as the RDF vocabulary Data Catalog Vocabulary (DCAT)[16]. DCAT facilitates the consumption and aggregation of metadata from multiple catalogs. It is integrated with other standards, such as Schema.org[17] and PROV Ontology (PROV-O[18]). The catalog is maintained in the microservice's own database (preferably NoSQL, to facilitate the storage of metadata in JSON).

The **Data Query** is the microservice responsible for processing the creation orders of visualization of data stored in **Metadata Manager** and synchronously executing SQL queries on the data in **Data Lake Storage**. Query statistics such as execution frequency and response time should be stored to allow the use of automatic indexing and caching techniques to speed up the access to frequently used data. Softwares like Hive and Spark SQL can be used to implement this microservice.

The analysis and processing of data in real-time are made by the **Stream Processing** microservice. This microservice enables the execution of tasks on data as soon it enters the platform. It provides an interface for developers to create and manage their jobs. Results of the data processing must be stored, enabling their use by other applications. This microservice can be implemented using tools such as Kafka and Spark Streaming.

The **Smart Analyzer** microservice provides tools to support data mining and the creation, management, and execution of machine learning models on the datasets in the **Data Lake Storage** and data streams provided by the **Stream Processing**. It can generate notification events in a message queue for consumption by other applications. To implement this microservice, Spark ML and Scikit Learn[19] can be used.

Batch Processing is responsible for enabling the processing of large datasets stored in **Data Lake Storage**, providing an interface for the creation, management, and execution of tasks. The processing results can be saved in **Data Lake Storage** or published in a message queue so that they can be consulted by APIs and visualization tools, or consumed by other applications. For batch data processing, MapReduce can be used.

Data Visualization supports graphical interfaces for presenting and analyzing data stored in **Data Lake Storage** and data generated by **Stream Processing** and **Smart Analyzer**. It allows users

[13] `https://avro.apache.org/` (Retrieved: 05/20/2022)
[14] `https://parquet.apache.org/` (Retrieved: 05/20/2022)
[15] `https://www.rabbitmq.com` (Retrieved: 05/20/2022)
[16] `https://www.w3.org/TR/vocab-dcat-3` (Retrieved: 05/20/2022)
[17] `https://schema.org/` (Retrieved: 05/20/2022)
[18] `https://www.w3.org/TR/prov-o` (Retrieved: 05/20/2022)
[19] `https://scikit-learn.org/` (Retrieved: 05/20/2022)

to create reports and dashboards. This service can be made available using tools such as Apache Superset[20] or Kibana[21].

The **API Gateway** provides for external applications a single access point (with load balancing) to the others microservices. The communication between the **API Gateway** and the external applications must use an encrypted communication channel. Furthermore, every request must consult the **Auth Server** microservice, which authenticates the user and generates a cryptographic token to be used in future requests. After successful authentication, the request is enriched with user information and forwarded to the targeted microservice. The gateway can be implemented using Apache Knox[22] or Kong[23], for example. The **Auth Server** can be implemented using Apache Syncope[24] or Auth0[25].

The **Public UI** is responsible for enabling users to access the web interface of other microservices in a single web interface. This way, each microservice offers the type of interface that suits it best. This microservice is public and uses a single access point, the **API Gateway**, with an encrypted communication channel and identified user. Among the technologies that can be used to implement the Web interface and integrate it with the functionalities provided by other microservices are NodeJS[26], React[27], and HTML5.

There are several concerns that an implementation of these microservices have to address. For example, Data Ingestion must be able to handle large data volumes and heterogeneity. Metadata Management must couple with structural and semantic data changes, to provide data compatibilization. Data Query must provide good performance (with automatic indexing, caching, etc.). API Gateway must balance load to handle requests in the most efficient manner. All the microservices must be scalable and fault-tolerant, while ensuring data security and privacy.

To increase the productivity and reliability of the platform's development, we suggest reusing free software in its implementation, mainly those that have an active community of developers; support to stable versions; evolutionary versions; a rich documentation; as well as integration with security and data privacy tools. To facilitate the operation and maintenance of the platform, we suggest the adoption of DevOps best practices, such as continuous integration, continuous delivery, continuous deployment, and monitoring.

3.2 Comparison with Related Architectures

Our architecture supports functionalities for data ingestion and physical integration using approaches similar to those of the works analyzed in Section 2. However, it extends the related works by supporting some unique features, such as: a single point of access to microservices by external applications; an authentication and access authorization service; a centralizing interface for the services; the creation of new data collections based on existing ones; and compatibilization of data in collections that have suffered structural or semantic changes over time, using the metadata modification history and mapping rules.

[Mehmood et al. 2019] had also proposed using metadata to support the data integration. Their architecture works with data models to provide unified vocabulary among data sources and align syntactic and semantic differences, demanding the definition of data models for each city sector (e.g., environmental, social, and economic data). In our work, we made a more comprehensive proposal for metadata management, using DCAT to describe the data sources. Moreover, we do not enforce uniformization in data ingestion. The data is stored as it comes from the source. Then, it can be

[20] `https://superset.apache.org` (Retrieved: 05/20/2022)
[21] `https://www.elastic.co/pt/kibana` (Retrieved: 05/20/2022)
[22] `https://knox.apache.org` (Retrieved: 05/20/2022)
[23] `https://konghq.com/kong` (Retrieved: 05/20/2022)
[24] `https://syncope.apache.org` (Retrieved: 05/20/2022)
[25] `https://auth0.com` (Retrieved: 05/20/2022)
[26] `https://nodejs.org` (Retrieved: 05/20/2022)
[27] `https://reactjs.org` (Retrieved: 05/20/2022)

compatibilized when it is queried. Views of compatibilized data can be materialized to speed up queries.

Another distinguishable feature of the data integration platform presented in this work (compared to those analyzed in Section 2.1) is its microservice architecture. This type of architecture minimizes the dependency between the components, enables independent development and scalability of the services, and facilitates the platform's implementation, testing, and maintenance.

4. GUIDELINES FOR PERFORMANCE EVALUATION

To evaluate the performance of a system that implements the proposed architecture, the Cloud Evaluation Experiment Methodology (CEEM) [Li et al. 2013] can be used. CEEM is a methodology for systematically evaluating the performance of cloud services by experiments, which can be easily replicated or extended to any environment.

The methodology proposes ten steps to evaluate a service: (1) *Requirement Recognition* – define the problem and objectives of the assessment; (2) *Service Feature Identification* – identify the services and features to be evaluated; (3) *Metrics and Benchmarks Listing* – list the metrics and benchmarks that can be used; (4) *Metrics and Benchmarks Selection* – select the appropriate metrics and benchmarks for evaluation; (5) *Experimental Factors Listing* – list factors that may impact in the experiments' evaluation; (6) *Experimental Factors Selection* – select the factors to be studied and define the acceptance criteria; (7) *Experimental Design* – design the experiments based on the previous steps; (8) *Experimental Implementation* – prepare the test environment and run the designed experiments; (9) *Experimental Analysis* – analyze and statistically interpret the experimental results; and (10) *Conclusion and Reporting*.

In the following, we present a guideline for applying CEEM to evaluate the performance of microservices of our data integration software architecture.

4.1 Requirement Recognition and Service Feature Identification

We want to assess the individual capacity of each microservice to function under both normal and above-normal workload conditions. In particular, we want to evaluate the effectiveness of self-scalability to support an increase in the number of users or simultaneous requests for the microservice's main functionalities, while keeping an acceptable quality of service.

The main functionalities to be evaluated through the experiments are: ingestion of data streams and batches received by the API in the **Data Ingestion** microservice; creating, querying, and compatibilizing metadata in the **Metadata Manager** microservice; and recovery of data from the **Data Lake Storage** in the **Data Query** microservice.

The number of users and requests per time interval to be supported by a microservice instance must be defined according to the smart city platform to which the system is coupled. These values will be used as parameters for the execution and analysis of the experiments. Therefore, they must be defined for each microservice to be analyzed.

This guideline does not include experimental scenarios for the **API Gateway**, **Auth Server**, **Stream Processing**, **Batch Processing**, **Smart Analyzer** and **Data Visualization** microservices because we assume that open-source tools with assessed good performance will be used in their implementation. The **Public UI** will not be considered either because it is a front-end application, thus it impacts the system's general performance less than the other components.

4.2 Metrics and Benchmarks Listing and Selection

In this analysis, we will consider the catalog of metrics presented by [Li et al. 2012b]. Four metrics of the catalog are particularly appropriate to assess the performance of the microservices: CPU utilization, RAM utilization, latency, and the number of requests processed per time interval. These metrics enable us to validate whether the developed system is capable of serving the expected number of users and requests with low latency.

4.3 Experimental Factors Listing and Selection

[Li et al. 2012a] point out the operating system and container manager versions as experimental factors to be considered. CPU clock speed and number of cores, type and capacity of RAM memory, and storage capacity are important factors as well, since both software and hardware changes can affect the performance results. To avoid the microservices' instances competing for computing resources, we suggestt running them in containers with limited resources.

4.4 Experimental Design and Implementation

This section describes three experiments designed to evaluate the microservices' performance. Each one of the experiments must be run with three different configurations:

4.4.1 *Configuration 1.* Execution with a single instance of the microservice with auto-scaling disabled, and a workload that gradually increases over time, until reaching the maximum size expected for the system. The goal in this configuration is to measure the microservice's performance under normal workloads.

4.4.2 *Configuration 2.* Execution of the microservice with auto-scaling enabled, and a workload varying between the lower and the upper limit values supported by a given fixed number of instances. In this configuration, the goal is to assess the capacity of the microservice to self-adjust to the current demand, increasing or decreasing the number of instances according to the variations in the workload.

4.4.3 *Configuration 3.* Execution with a single instance of the microservice with auto-scaling disabled, and a workload that gradually increases over time (both in the number of users and in the number of concurrent requests per time unit), until CPU or RAM usage is close to 100%. In this configuration, the goal is to identify the maximum number of users and concurrent requests that a single instance can handle.

During the execution of the experiments, information about the CPU and RAM usage, request processing time, and number of messages processed per time unit (throughput) must be collected and recorded. The number of replications of each experiment should be defined taking into account the resources available, and the desired sensitivity and confidence of the performance indexes to be obtained from the measurements. Generally, the sensitivity increases with the number of replications of the experiment.

4.4.4 *Experiment 1 – Data Intake Latency.* Start a single instance of the **Data Ingestion** microservice and trigger real-time or batch data provisioning, varying the number of requests over time according to the configuration being executed.

4.4.5 *Experiment 2 – Response Time For Operations on Metadata.* Start a single instance of the **Metadata Manager** microservice and simulate the simultaneous execution of metadata creation, query, and compatibilization requests for different data collections, varying the number of simulated users and requests over time according to the experiment configuration being executed.

4.4.6 *Experiment 3 – Response Time of Data Queries.* To evaluate the **Data Query**, first initialize the **Data Lake Storage** with data from different collections and sources until reaching a considerable percentage of use of its storage capacity, to make possible the evaluation of the response time of queries over a large volume of data. Then, for each data collection stored, create a view in the **Data Query**. After this initialization, launch an instance of **Data Query**. Use a simulator to generate and execute queries with filters and random aggregations over the pre-existing views, varying the number of simulated users and requests over time according to the experiment configuration being executed.

4.5 Experimental Analysis

To analyze data collected in the experiments and drawn conclusions, it is strongly recommended the use of statistical methods to ensure robustness [Li et al. 2013]. Nevertheless, even simple graphical tools (e.g. dot plots, histograms, and box plots) showing the response time, throughput, CPU usage, and RAM usage over time may help to visualize how well a microservice self-adjust to workload variations.

5. AN IMPLEMENTATION OF THE ARCHITECTURE AND ITS EXPERIMENTAL EVALUATION

In this section, we describe a proof-of-concept implementation of the core of the architecture introduced in Section 3, entirely built using open-source tools. The source code of this implementation is available at `https://gitlab.com/interscity/data-integration` (Retrieved: 05/20/2022). The performance of the system was evaluated following the guidelines presented in Section 4.

5.1 Implementation Details

The architecture has as foundation the **Data Lake Storage** for storing historical data. This service was made available in an HDFS cluster consisting of a NameNode and one or more DataNodes[28].

To offer capabilities to transmit data to the platform, we implemented a **Data Ingestion** service using the Go programming language[29], a language focused on simplicity, reliability, and efficiency, used to build robust-performance scalable software. The **Data Ingestion** is composed of the **DataIngestionProducer** and **DataIngestionConsumer** microservices. Segregating **Data Ingestion** into two microservices makes it possible to reduce the response time for clients and improve the scalability of the service.

DataIngestionProducer provides a RESTful API for sending data collections in JSON or CSV format and publish them into the topic **data-batch** of Kafka, where they are maintained and tuned to deliver real-time responses to clients. Kafka is a high-performance messaging platform that handles real-time streaming for fast, scalable operations. The real-time data is published in Kafka's **data-stream** topic. **DataIngestionConsumer** consumes data from Kafka and store it in the **Data Lake Storage**. It gets data from Kafka topics in microbatches and inserts it into HDFS.

We also implemented for **Metadata Management** a RESTful API that receives the metadata of a data collection in JSON format, inserts it in MongoDB[30] database, and generates a notification of a new version in Kafka's **metadata-version** topic. The notifications in this topic are consumed, for example, by the component responsible to provide data visualization in the **Data Query** microservice.

The model of the mandatory metadata is based on the Data Catalog Vocabulary (DCAT) and has attributes to identify the organizations that generated the data and to describe the data schemas. The

[28] In an HDFS cluster, a NameNode is a master server that manages the file system namespace and access to files by clients, while the DataNodes manage storage attached to the nodes.
[29] `https://go.dev/` (Retrieved: 05/20/2022)
[30] `https://www.mongodb.com` (Retrieved: 05/20/2022)

model also has attributes to support metadata version management, such as version ID and version creation date, in order to register how schemas evolve over time. Figure 2 shows the data model of the database of the **Metadata Management**.

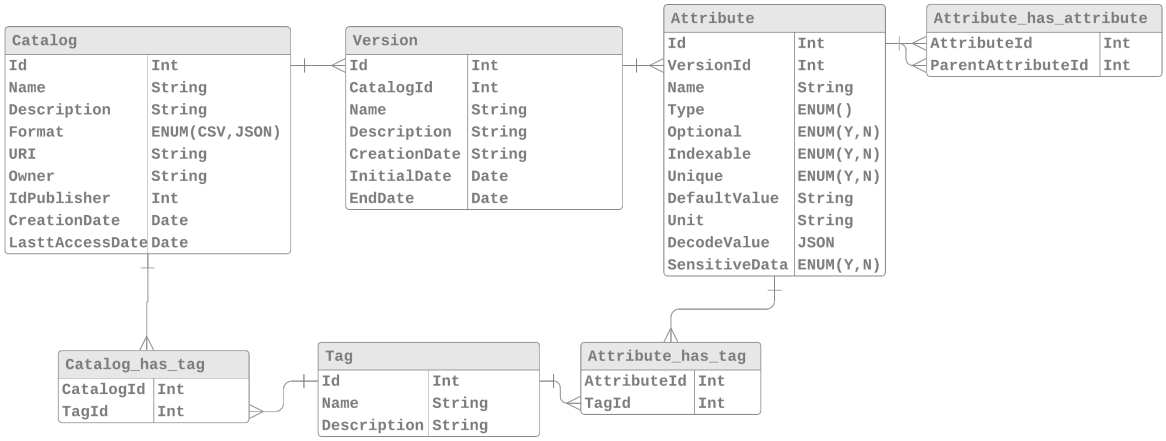

Fig. 2. Data model used for metadata management.

The **API Gateway** was made available as a Kong container. Kong is an open-source distributed and scalable gateway that acts as middleware between clients and API-centric applications. In our platform, it redirects all requests to the internal microservices so that clients have a single access point. The **API Gateway** receives all requests, determines which microservice instance should respond to each request (based on the URI and load balancing rules), and then forwards the request.

All implemented microservices are available as Docker containers[31]. The open-source tools used in the project (i.e., Kafka, Kong, and MongoDB) were also used in containers provided by the community. We used Kubernetes[32] to automatize the deployment of the Docker containers in a cluster of virtual machines in a cloud platform and to provide individual scaling for the services, to appropriately support the workload fluctuations.

The other microservices of the proposed architecture were not included in the proof-of-concept.

5.2 Performance Evaluation and Analysis

5.2.1 *Data Ingestion.* The experiments were executed in Docker containers deployed in virtual machines in the Digital Ocean[33] cloud platform. Each service instance was hosted on its own virtual machine, ensuring a fixed amount of machine resources per service. To run the experiment, we used an Ubuntu 20.04 (LTS) x64 machine with a shared CPU, 25GB of SSD, and 2GB of RAM for each instanced service. All machines were provisioned in the same region (New York).

To measure the latency of the **DataIngestionProducer** and **DataIngestionConsumer** microservices, we performed an isolated experiment for each microservice using a single active instance and with autoscaling disabled. We ran each experiment for five minutes and repeated it 15 times in order to minimize the effects of uncontrollable variables inherent to the environment in which the experiments were performed. The experiment was performed using real-world data, the Travel Sensors[34]

[31] `https://www.docker.com` (Retrieved: 05/20/2022)
[32] `https://kubernetes.io` (Retrieved: 05/20/2022)
[33] `https://digitalocean.com/` (Retrieved: 05/20/2022)
[34] `https://catalog.data.gov/dataset/travel-sensors` (Retrieved: 05/20/2022)

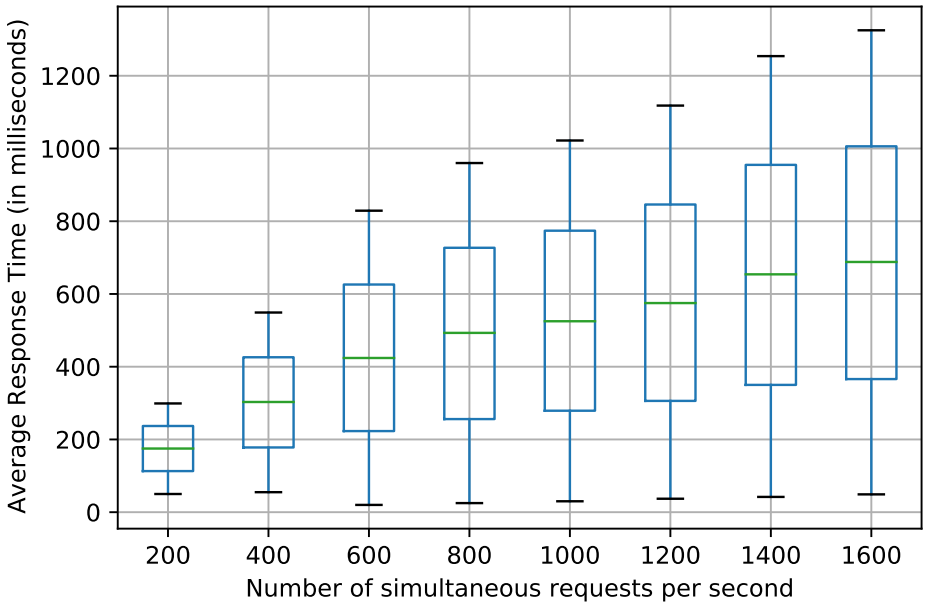

Fig. 3. DataIngestionProducer average response time degradation as the number of simultaneous requests increases.

dataset from the city of Austin (Texas, USA). This dataset contains information about travel sensors used by the City of Austin Department of Transportation to monitor traffic conditions across the city.

To analyze the degradation of **DataIngestionProducer**, we ran a load test simulating concurrent requests with Apache Jmeter[35], for workloads ranging from 200 to 1600 requests per second, and collected the response time from the client's point of view. Figure 3 shows the response time degradation as the number of concurrent requests increases. The response time, in this case, is the difference between the time of arrival of the request in the **DataIngestionProducer** and the time of arrival of the response to client. Each box-plot in Figure 3 summarizes the results of the 15 rounds of the experiment using a fixed number of requests per second. From each round, an average response time was calculated. The box-plot shows the minimum, first quartile, median (green line), third quartile, and the maximum of the rounds' average response times. The best median was 187 milliseconds for 200 concurrent requests. The median average response time remained below 1 second for all test scenarios, which is an excellent result according to [Del Esposte et al. 2019].

To evaluate the **DataIngestionConsumer**, we entered 960 thousand messages of the same size (4096 bytes) in the topic **data-stream** of Kafka. We started an instance of the microservice and let it run for 5 minutes. Figure 4 shows how the latency varied in each one of the 15 rounds of the experiment. The latency is the difference between the timestamp of the arrival of the message in the **DataIngestionConsumer** and the timestamp of its insertion in HDFS. The highest median latency was 8 milliseconds and the lowest 7 milliseconds.

We run Experiment 1 with Configuration 2 (described in Section 4.4) to evaluate the horizontal scalability of **DataIngestionProducer** and **DataIngestionConsumer**. We started an instance of each microservice and triggered simultaneous requests, ranging from 0 to 500 requests per second. Every time a container reached an average of 30% of CPU usage during the last 30-second interval, a new container of the microservice was instantiated. Contrarily, every time the average CPU usage of all the containers of a microservice during the last 30-second interval became less than 20%, one

[35] `https://jmeter.apache.org` (Retrieved: 05/20/2022)

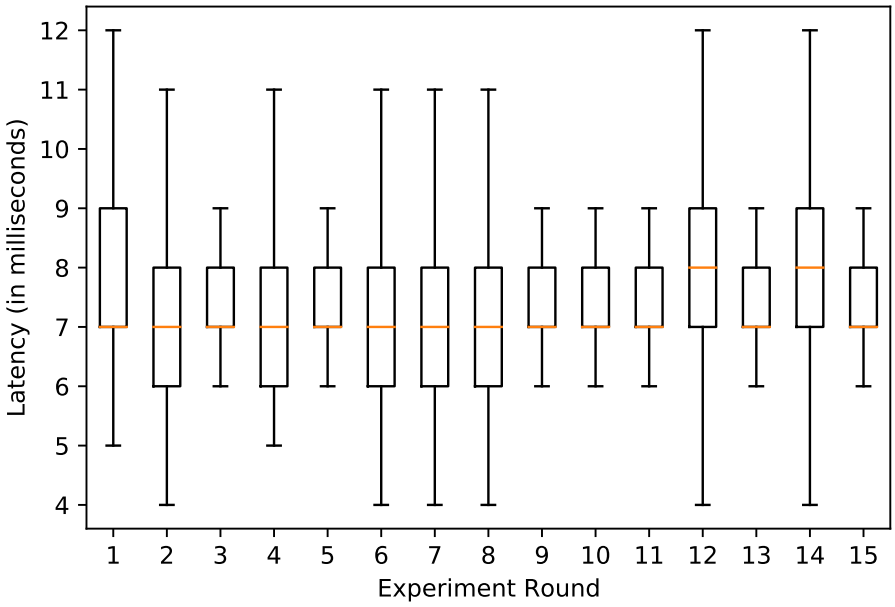

Fig. 4. DataIngestionConsumer latencies in all the 15 rounds of the experiment.

of the containers was destroyed and its workload was transferred to the remaining containers. Thus, the system increased or decreased the number of containers per service by balancing the workload to match the desired average CPU usage in the 30-second interval. Figure 5 shows the creation and destruction of the **DataIngestionConsumer** microservice containers due to the application of the auto-scaling strategy in a single round of the experiment. Initially, each service had only one container. The number of containers varied between 1 and 4 as a response of the elasticity mechanism to the demand fluctuations.

6. CONCLUSION

The contribution of this work is twofold. First, it provides a panorama of the requirements of data integration for smart cities and state-of-the-art solutions. Second, it presents an architecture to help researchers and developers to approach the implementation of these requirements and also guidance to assess its performance.

We have identified functional and non-functional requirements of data integration in smart cities and the challenges involved in their implementation by analyzing the related literature. Then, we have proposed a microservices architecture for a data integration platform that meets these requirements. We have also specified the software components needed to implement them. Our software architecture extends those of related work by providing a solution for metadata management that keeps the history of changes in the structure and semantics of attributes, to enable the compatibilization of data in queries. Another differentiated feature of the architecture is the API Gateway, which provides a secure data access point for external applications (through encrypted and authenticated communication channels).

Following the CEEM methodology, we have designed a set of experiments that can be used to evaluate the performance of the microservices of the architecture under both normal and above-normal workload conditions. With these experiments, one can assess the effectiveness of self-scalability to keep acceptable quality of service while the number of users and requests vary over time.

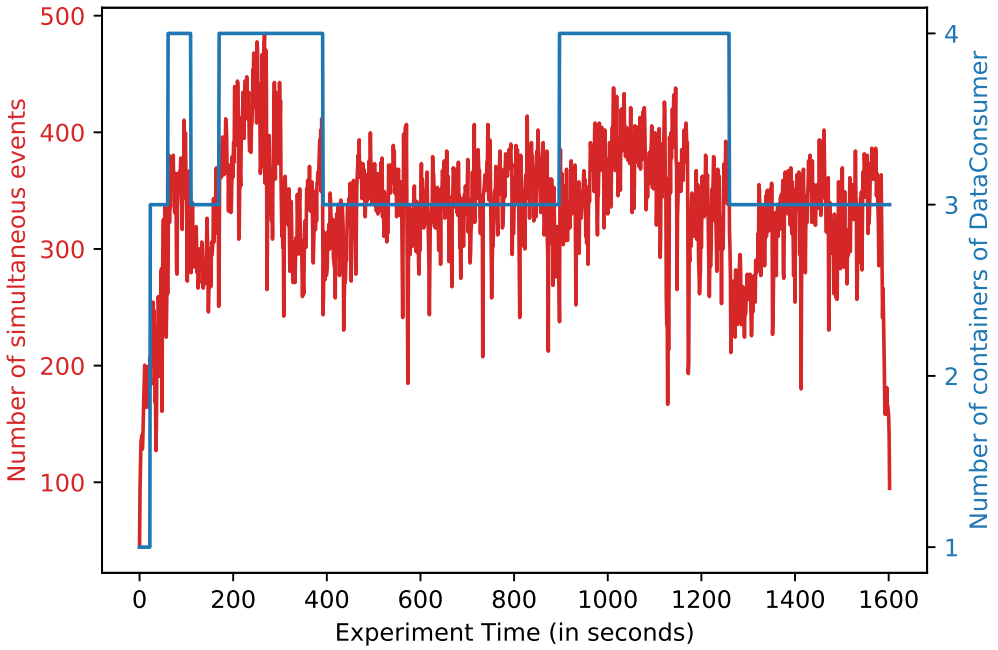

Fig. 5. DataIngestionConsumer auto-scaling (variation of the number of containers of the microservice in function of the workload fluctuation).

We have implemented a proof of concept of the core microservices of the proposed architecture using open-source tools. In the experimental performance evaluation of the platform, the implemented microservices (**DataIngestionProducer** and **DataIngestionConsumer**) presented response times within the expected range and had horizontally scaled according to the workload fluctuations. These results show the architecture can meet the performance demands of a smart city. We are currently working on implementing the remaining components of the architecture in InterSCity, an open source platform for smart cities.

REFERENCES

Al Nuaimi, E., Al Neyadi, H., Mohamed, N., and Al-Jaroodi, J. Applications of big data to smart cities. *Journal of Internet Services and Applications* 6 (1): 25, 2015.

Albino, V., Berardi, U., and Dangelico, R. M. Smart cities: Definitions, dimensions, performance, and initiatives. *Journal of Urban Technology* 22 (1): 3–21, 2015.

Cheng, B., Longo, S., Cirillo, F., Bauer, M., and Kovacs, E. Building a big data platform for smart cities: Experience and lessons from Santander. In *2015 IEEE International Congress on Big Data*. IEEE BigData 2015. IEEE, New York, pp. 592–599, 2015.

Consoli, S., Mongiovic, M., Nuzzolese, A. G., Peroni, S., Presutti, V., Reforgiato Recupero, D., and Spampinato, D. A smart city data model based on semantics best practice and principles. In *Proceedings of the 24th International Conference on World Wide Web*. WWW '15 Companion. Association for Computing Machinery, New York, NY, USA, pp. 1395–1400, 2015.

Costa, C. and Santos, M. Y. The SusCity big data warehousing approach for smart cities. In *Proceedings of the 21st international database engineering & applications symposium*. IDEAS 2017. Association for Computing Machinery, New York, NY, USA, pp. 264–273, 2017.

Del Esposte, A. d. M., Santana, E. F., Kanashiro, L., Costa, F. M., Braghetto, K. R., Lago, N., and Kon, F. Design and evaluation of a scalable smart city software platform with large-scale simulations. *Future Generation Computer Systems* vol. 93, pp. 427 – 441, 2019.

Del Esposte, A. M., Kon, F., M. Costa, F., and Lago, N. Interscity: A scalable microservice-based open source platform for smart cities. In *Proceedings of the 6th International Conference on Smart Cities and Green ICT Systems*. SMARTGREENS 2017. SCITEPRESS - Science and Technology Publications, Lda, Setubal, PRT, pp. 35–46, 2017.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. Microservices: Yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer (Eds.). Springer International Publishing, Cham, pp. 195–216, 2017.

Hashem, I. A. T., Chang, V., Anuar, N. B., Adewole, K., Yaqoob, I., Gani, A., Ahmed, E., and Chiroma, H. The role of big data in smart city. *International Journal of Information Management* 36 (5): 748 – 758, 2016.

Klemm, P., Lawonn, K., Glaser, S., Niemann, U., Hegenscheid, K., Volzke, H., and Preim, B. 3d regression heat map analysis of population study data. *IEEE Transactions on Visualization & Computer Graphics* 22 (01): 81–90, jan, 2016.

Li, Z., OBrien, L., and Zhang, H. Ceem: A practical methodology for cloud services evaluation. In *2013 IEEE Ninth World Congress on Services*. IEEE, Santa Clara, CA, USA, pp. 44–51, 2013.

Li, Z., O'Brien, L., Zhang, H., and Cai, R. A factor framework for experimental design for performance evaluation of commercial cloud services. In *4th IEEE Intl. Conf. on Cloud Computing Technology and Science Proceedings*. IEEE, Taipei, Taiwan, pp. 169–176, 2012a.

Li, Z., O'Brien, L., Zhang, H., and Cai, R. On a catalogue of metrics for evaluating commercial cloud services. In *2012 ACM/IEEE 13th International Conference on Grid Computing*. IEEE, Beijing, China, pp. 164–173, 2012b.

Mehmood, H., Gilman, E., Cortes, M., Kostakos, P., Byrne, A., Valta, K., Tekes, S., and Riekki, J. Implementing big data lake for heterogeneous data sources. In *IEEE 35th Intl. Conference on Data Engineering Workshops (ICDEW 2019)*. IEEE, Macao, China, pp. 37–44, 2019.

Psyllidis, A., Bozzon, A., Bocconi, S., and Titos Bolivar, C. A platform for urban analytics and semantic data integration in city planning. In *Computer-Aided Architectural Design Futures. The Next City - New Technologies and the Future of the Built Environment*. CAAD Futures 2015. Springer Berlin Heidelberg, São Paulo, Brazil, pp. 21–36, 2015.

Raghavan, S., Simon, B. Y. L., Lee, Y. L., Tan, W. L., and Kee, K. K. Data integration for smart cities: Opportunities and challenges. In *Computational Science and Technology*, R. Alfred, Y. Lim, H. Haviluddin, and C. K. On (Eds.). Springer, Singapore, pp. 393–403, 2020.

Rathore, M. M., Ahmad, A., Paul, A., and Rho, S. Urban planning and building smart cities based on the internet of things using big data analytics. *Computer Networks* vol. 101, pp. 63 – 80, 2016.

Ribeiro, M. and Braghetto, K. A data integration architecture for smart cities. In *Anais do XXXVI Simpósio Brasileiro de Bancos de Dados*. SBC, Porto Alegre, RS, Brasil, pp. 205–216, 2021.

Silva, B. N., Khan, M., and Han, K. Towards sustainable smart cities: A review of trends, architectures, components, and open challenges in smart cities. *Sustainable Cities and Society* vol. 38, pp. 697 – 713, 2018.

Taibi, D., Lenarduzzi, V., and Pahl, C. Architectural patterns for microservices: a systematic mapping study. In *CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, Lda., Funchal, Madeira, Portugal, pp. 221–232, 2018.

Yue, M., Fan, L., and Shahabi, C. Inferring traffic incident start time with loop sensor data. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. CIKM '16. Association for Computing Machinery, New York, NY, USA, pp. 2481–2484, 2016.

Zheng, Y., Capra, L., Wolfson, O., and Yang, H. Urban computing: Concepts, methodologies, and applications. *ACM Trans. Intell. Syst. Technol.* 5 (3): 1–55, sep, 2014.