Institute of Architecture of Application Systems

Data Flow Dependent Component Placement of Data Processing Cloud Applications

Michael Zimmermann, Uwe Breitenbücher, Kálmán Képes,
Frank Leymann, and Benjamin Weder

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{zimmermann, breitenbuecher, kepes, leymann, weder}@iaas.uni-stuttgart.de

BibTeX:

```
@inproceedings{Zimmermann2020_ComponentPlacement,
  author    = {Michael Zimmermann and Uwe Breitenb{\"u}cher and K{\'a}lm{\'a}n
               K{\'e}pes and Frank Leymann and Benjamin Weder},
  title     = {{Data Flow Dependent Component Placement of Data Processing
                Cloud Applications}},
  booktitle = {Proceedings of the 2020 IEEE International Conference on Cloud
               Engineering (IC2E 2020)},
  year      = 2020,
  month     = apr,
  pages     = {83--94},
  isbn      = {978-1-7281-1099-8},
  doi       = {10.1109/IC2E48712.2020.00016},
  publisher = {IEEE Computer Society}
}
```


Universität Stuttgart
Germany

Data Flow Dependent Component Placement of Data Processing Cloud Applications

Michael Zimmermann, Uwe Breitenbücher, Kálmán Képes, Frank Leymann, Benjamin Weder
Institute of Architecture of Application Systems, University of Stuttgart, Stuttgart, Germany
[firstname.lastname]@iaas.uni-stuttgart.de

***Abstract*—With the ongoing advances in the area of cloud computing, Internet of Things, Industry 4.0, and the increasing prevalence of cyber-physical systems and devices equipped with sensors, the amount of data generated every second is rising steadily. Thereby, the gathering of data and the creation of added value from this data is getting easier and easier. However, the increasing volume of data stored in the cloud leads to new challenges. Analytics software and scalable platforms are required to evaluate the data distributed all over the internet. But with distributed applications and large data sets to be handled, the network becomes a bottleneck. Therefore, in this work, we present an approach to automatically improve the deployment of such applications regarding the placement of data processing components dependent on the data flow of the application. To show the practical feasibility of our approach, we implemented a prototype based on the open-source ecosystem OpenTOSCA. Moreover, we evaluated our prototype using various scenarios.**

***Index Terms*—Application Deployment, Deployment Automation, Data Locality, Data Flow, TOSCA**

I. Introduction

Advances in Internet of Things (IoT) [1], cloud computing [2], and the increased usage of sensor-equipped devices and Cyber-Physical Systems (CPS) [3], have led to a significantly increased amount of data stored in the cloud [4]. A wide range of domains can benefit from new opportunities, such as cost-savings through pay-per-use models, for example, mobility [5], health care [6], energy management [7], and scientific computing in general [8]. Also in the area of Industry 4.0, new use cases can be realized, for example, by deploying the processing logic as close as possible to the data sources [9], [10]. In order to enable data analysis providing new insights and optimization potential, for instance, in manufacturing processes, data from different sources and locations need to be integrated, analyzed, and compared. However, with the increasing amount of stored data, the network is becoming a bottleneck and the movement of data from one location to another becomes a problem [11], [12]. Thus, components need to be distributed over different locations, e.g., private, public, and edge clouds [13], depending on the specific requirements as well as the locality of the data to be analyzed. However, the distribution of applications leads to various challenges: required middleware and application components have to be deployed and configured in order to obtain the required data and to enable the communication between each other [14]. In cases where the application and the overall deployment process are highly complex, a manual deployment approach is error-prone, time-consuming, and therefore, not efficient [15], [16].

The deployment and management of applications can be automated using various deployment technologies, such as (i) provider-specific systems, (ii) provider-independent, but platform-specific technologies, (iii) general-purpose technologies [17], or furthermore, (iv) provider- and technology-agnostic standards, such as the *Topology and Orchestration Specification for Cloud Applications (TOSCA)* [18]–[20].

However, the placement of application components regarding their locality can vary depending on different characteristics, for example, requirements of the data and the data processing components. For instance, if the data from different data sets, located in different data centers or countries should be analyzed, dependent on the respective size of the data and if the data can be preprocessed in order to reduce the overall size, for example, by aggregating a time series, it can make a significant difference where the processing logic is located [21]. And thus, how much data needs to be moved from one location to another. Furthermore, differences in data privacy policies or laws may also lead to certain deployment locations or prohibit some [22]. For complex applications, processing multiple distributed data sets, the manual effort for evaluating all generally possible deployment solutions regarding the placement of components and data sets is immense.

In this paper, we tackle these challenges. We present an approach, that automatically improves the deployment of distributed applications regarding the placement of data processing components and data sets by taking the following factors into account: (i) the size of the data sets, (ii) the data flow of the application, and (iii) further characteristics, such as data privacy requirements, performance requirements, and the *Data Factor (DF)* [23]. The *Data Factor* describes, if and to what extent a data processing function is reducing or inflating the volume of the processed data, and therefore, how much data needs to be shipped from one location to another after the function was applied to the data. The goal is to reduce the amount of data that has to be transferred over a long distance via the internet. Moreover, besides the placement decision-making, our approach supports the general deployment of applications by automatically (i) generating an incomplete deployment model based on the data flow of an application, (ii) completing this deployment model based on the results of the placement algorithms for each component, and (iii) deploying the data processing application itself. To validate the practical feasibility of our approach, we present a prototypical implementation based on the TOSCA standard.

The remainder of this paper is structured as follows. In Section II, background, fundamentals, and a motivational scenario are introduced. In Section III, the method of our approach is described step by step. Our algorithms are presented in Section IV. In Section V, our approach is validated by a prototypical implementation and evaluated. Section VI discusses related work and Section VII concludes this paper.

II. Fundamentals, Background and Motivational Scenario

In the following subsections, we first introduce some basic information about deployment models, required for understanding the remainder of this paper. Moreover, we introduce concepts of having the data to be processed as close as possible to the components processing the data, called data locality [24], [25]. For example, in cloud computing, the higher the data locality, the less data needs to be transferred over the network. Since data locality is not a new idea, we present an overview of different areas and domains in which data locality concepts are researched and applied. Furthermore, we present a motivational scenario for our proposed approach.

A. Deployment Models and Topologies

Since the manual deployment of applications is too error-prone, time-consuming, and requires an immense level of technical expertise [15], [16], the automation of application deployment is essential. Various deployment automation systems exist, supporting the automated and model-based deployment of applications [26]. Technically, deployment models can be separated into *declarative deployment models* as well as *imperative deployment models* [27]. Declarative deployment models describe the desired structure of an application that shall be deployed. These models are based on directed graphs and consists of components, their relations, as well as properties [27]. They are also referred as topology-based deployment models, or just short as *topology* of an application. Imperative deployment models, on the other hand, describe all the single deployment tasks required for the deployment of an application. For declarative deployment models, these activities can be inferred by a runtime based on the topology of the application [28]. Since a variety of deployment systems support declarative deployment models [26], the presented approach is based on this kind of deployment models.

Figure 1 depicts an exemplary declarative deployment model consisting of four main *components*: *SourceDB*, *Processor*, *ResultDB*, and *Monitor*. Components have *component types*, e.g., *Grafana* or *Java-App*. They are reusable and define the semantics of a component. Components are connected using typed *relations*, e.g., *hostedOn* or *connectsTo*. For instance, *Monitor* and *ResultDB* are hosted on an *UbuntuVM* provided by *AWS-EC2*. In order to get the data, the *Processor* connects to the *SourceDB*. *Properties* are used to provide additional information, for example, the endpoint of *OpenStack* and the credentials for *AWS-EC2*. Additionally, as stated by the property *State*, the left stack consisting of *SourceDB*, *MySQL-DBMS*, and *UbuntuVM* is already running on *OpenStack*.

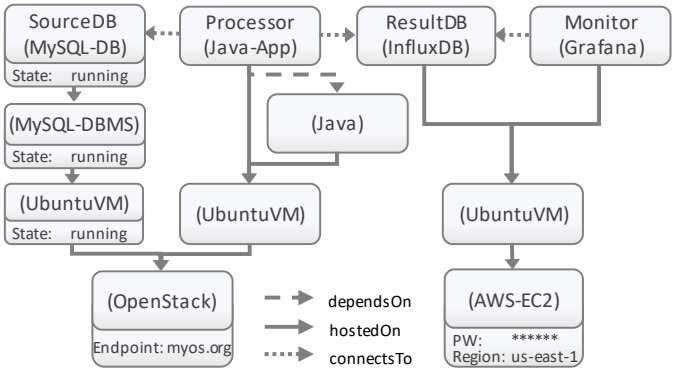

Fig. 1. Exemplary declarative deployment model.

B. Data Locality

The basic idea of data locality is to have the data as close as possible to the function processing this data [24], [25], for example, on the same physical node. Data locality can be achieved by the two paradigms of *function shipping* and *data shipping* [9], [29]. Function shipping is applied when the functionality is shipped or provisioned close to the data to be processed. Data shipping, on the other side, is applied when the data is shipped to the functionality processing the data. In the following we want to give a first brief overview of areas in which the concept of data locality has been researched.

Cornell et al. [30] examined the performance of coupled multi-system database systems using a *function request shipping* approach and a *data sharing approach*. In the first approach, the database is partitioned among the multi-system, and only the function requests are shipped between them. In the latter approach, the idea is to share the data among the multi-system by sharing a common database at disk level. The main goal of both approaches is to avoid transferring an unnecessary amount of data between the single systems.

In the area of microprocessors and microcomputers, research was conducted about merging processing and memory into a single chip in order to reduce overall data movement, and therefore, increase the overall performance [31]–[33].

MapReduce [34] presents a programming model addressing these data locality issues in order to improve the performance of data-intensive applications in the area of cloud computing and Big Data [25]. MapReduce simplifies the development and execution of large-scale data processing jobs, by decomposing and distributing the tasks on distributed file systems and gathering as well as integrating the results afterwards again. It is used, e.g., by Google File System [35] and Hadoop Distributed File System [36] to handle large amounts of data.

While in Big Data analytics, the data processing is typically performed in centralized ways, for example, in a Big Data warehouse, in edge analytics [37] the data processing is done near the active device or even directly on the device itself. Especially in IoT with lots of connected devices, this approach is often used in order to reduce the effort and required time of sending device data into the cloud or an on-premise server.

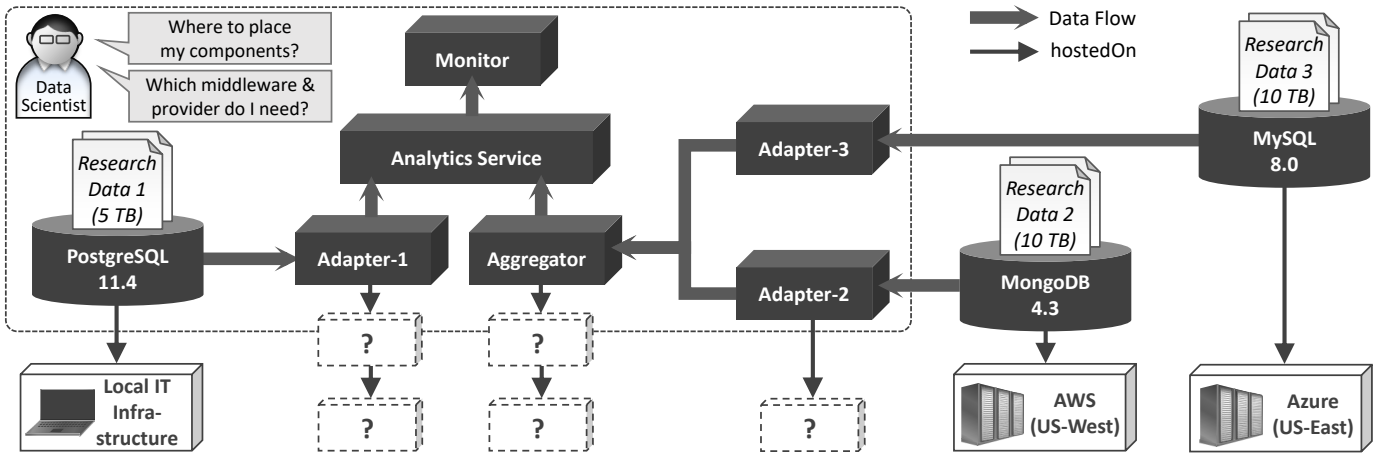

Fig. 2. Motivating scenario of an application for analyzing data from different locations.

In high-performance computing systems, shipping the data often is the preferred option, since in such systems, typically performing CPU-intensive computations, the effort of transferring the data to the computing units is low compared to the computing time [38]. However, within the context of Big Data, IoT, cloud, and edge computing, the approach of transferring large amounts of data to the computing components, usually would be significantly slower than moving the computing components as near as possible to the data to be processed [12].

Furthermore, in the area of scientific workflows, especially when they are greatly data-intensive, location as well as distribution depending on the size of the data to be processed are acknowledged research challenges to be solved [39], [40].

As this brief overview shows, there are different approaches for the placement problem proposed in the literature, located in different domains and based on different assumptions and optimization goals. A more precise differentiation of our work to existing related work is given in the related work section.

C. Motivating Scenario

For emphasizing the motivation of this research work, a motivating scenario for placing components of data processing cloud applications based on their data flow is presented in this section. The scenario illustrates the connection of different data sets to be analyzed with the components to analyze them.

In Figure 2, the exemplary data processing cloud application is depicted, consisting of data processing components and various data sets to be analyzed. Besides the single components for processing the data and the data sets, the figure also shows the data flow of the overall application. On the right side, two different data sets are shown: *Research Data 2* stored in a *MongoDB* database and *Research Data 3* stored in a *MySQL* database. Both databases containing the research data sets are already available and are hosted on the two cloud providers *AWS* and *Azure*. While Research Data 2 has a volume of 10 terabytes and is stored in the region *US-West*, Research Data 3 has a volume of 10 terabytes as well, but is stored in the region *US-East*. Moreover, both databases are managed by third-parties, which are only granting public read access to them. On the left side of the figure, the data scientist is depicted. He has gathered 5 terabytes of data (*Research Data 1*), stored in a *PostgreSQL*, hosted on a local infrastructure, for example, a private cloud or some legacy system. The other components of the depicted application, for example, the three *Adapters* for getting the data, the *Aggregator* for aggregating *Research Data 2* and *Research Data 3*, the *Analytics Service* for analyzing the data, and the *Monitor* for showing the results, are developed and managed by the scientist as well. The goal of the data scientist is to analyze and compare the two data sets stored in the US with his locally available data set.

However, since the both data sets available in the US together are 20 terabytes in size, manually downloading the data from the US in order to process and analyze the data locally would be time-consuming. Provisioning the entire application in a local environment and pulling the data from the databases would be time-consuming as well. Likewise, provisioning all required components and uploading 5 terabytes of data near to the data in the US would be time-consuming and not efficient. Since a huge amount of data needs to be transferred from different locations to a common location, in these scenarios, as described in Subsection II-B, the network can be seen as a serious bottleneck, regarding the performance of the application. Depending on the data flow as well as the characteristics of the single components (pre-)processing the data, for example, if they are reducing or inflating the data, the performance of the application can be improved by distributing the components. Data reducing processing steps are, for example, steps aggregating time series or selecting only subsets, for example, by removing irrelevant or redundant data from the source data set. Data inflating are processing steps interpolating missing values or unzipping archives, for example. Thus, the goal of our approach is to place data processing components and data sets to be provisioned dependent on the characteristics of every single component as well as the data flow of the overall application. And therefore, reducing the transferred data in the system.

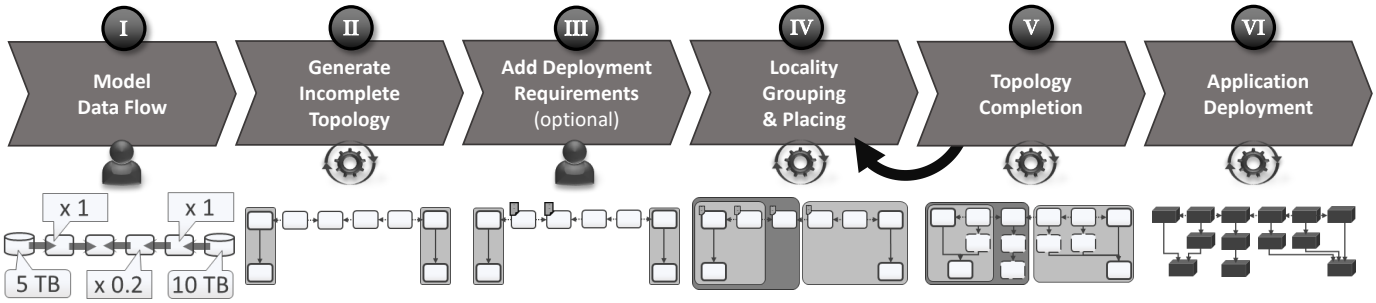

Fig. 3. Overview of the method for placing the components of data processing cloud applications.

III. Method for Placing Components of Data Processing Applications

In this section, we present our method for placing the components of data processing cloud applications regarding their location. An overview of the method is depicted in Figure 3. The goal is to distribute the components of data processing applications based on the data flow of the overall application and the characteristics of the single components, in order to reduce the amount of data that needs to be moved from one location to another over a long distance via the network.

A. Model Data Flow

In the first step, a data flow model representing (i) the data sources, (ii) the components of the data processing application, (iii) the data flow between the data sources and these components, and (iv) characteristics, such as the size of the data sets and the *Data Factor (DF)* has to be created. Again, the Data Factor describes, if and to what extent a data processing component (i) reduces the data it processes ($DF < 1$), (ii) inflates it ($DF > 1$), or (iii) is neutral regarding the size of the data ($DF = 1$). This factor is required in order, for example, to place a component reducing the size of data that needs to be transferred to another location, as near as possible to the data it should process. On the other hand, a component inflating the size of data should be placed as near as possible to its target processing location. In case of a data size neutral component, the component can be placed flexible, as it is not affecting the amount of data that needs to be transferred.

For modeling the data flow, we use a pipes-and-filters [41] oriented approach. A pipes-and-filters based model is a directed graph consisting of nodes and edges. The nodes are called filters and the edges are called pipes. Filters have input and output data and are processing steps, i.e., components, for example, to aggregate or interpolate data. Pipes connect filters and describe the direction of the data flow. An example of such a model is shown in Figure 3 below step one. By using pipes-and-filters as well as the annotated Data Factor for each filter, the data flow of the overall application can be modeled.

B. Generate Incomplete Topology

In the second step of our method, an incomplete topology, i.e., a declarative deployment model with open requirements, is generated using the data flow model from the previous step.

Based on the names of the components of the data flow model, either, (i) an already existing and matching component can be found in a repository or (ii) a corresponding component needs to be created and added to the topology. For example, in our motivation scenario, data processing components, such as the Analytics Service or Adapter-1 are mapped to concrete executable implementations. For instance, the Analytics Service may be implemented as a Java application able to run on an Apache Flink framework and the adapter as a simple Python script, which can read the data from the Research Data 1 data source. The data flow itself is transformed by adding a corresponding relation for each data flow edge to the topology. These relations are set to the type of *connectsTo*, which defines a connection between two components or a component and a data source. The direction of relations is based on whether a component pulls or pushes data, which can be annotated in the data flow. For example, the adapter components in our motivating scenario initiate the communication with the databases and pull the data from them. Therefore, the adapters will be the sources of the *connectsTo* relations between them and the databases. After adding all components and relations to the topology, each data source component is automatically annotated with an initial locality group, specifying that the data set it contains is available at the specified location. Furthermore, this defines, that the data sources are already running on some hosting environment and don't have to be provisioned first. The processing components, however, have to be provisioned either close to these data sources as the processed data may be reduced by them, or close to another processing component in cases where the data is inflated.

C. Add Deployment Requirements

In the optional step three, requirements regarding the deployment of the components themselves can be specified. Required computing power, storage capacity, as well as other restrictions in the selection of possible hosting locations are such requirements. For instance, that some data and the components processing this data are only allowed to be hosted in a specific country or on a companies' private cloud, because of data privacy policies or other legal regulations. In our motivating scenario, for example, the data processing components Analytics Service and Aggregator may require particular computing power, such as CPU and RAM from

the hosting systems. Additionally, the Adapter-1 component for reading the data from the data source Research Data 1 may have the requirement, that it must be deployed, e.g., on a specific private cloud because of data privacy policies or laws.

D. Locality Grouping and Placing

In the fourth step, the placement of the data processing components in relation to the data source components is calculated. As input, the data flow model created in the first step, as well as the incomplete deployment model along with the specified requirements and initial locality groups from steps two and three are required. Based on the data flow, the Data Factor, and the specified component deployment requirements, the processing components are grouped together, describing that these components should be deployed close to each other. In our motivating scenario, for example, the Aggregator should be placed near to the Adapters 2 and 3, as it reduces the amount of data to be transferred. When all processing components are assigned to a locality group, at the end of this step, each group is also assigned to a specific provider which is able to host the components and fulfill all specified requirements. In Section IV, the algorithms to calculate the locality groups and place the components accordingly in an automated manner, are described in detail.

E. Topology Completion

In step five, the so far incomplete deployment model is completed based on the location and provider assignments performed in step four. This can be done manually or automated by using a completion approach for deployment models, as proposed, for example, in [42]–[44]. Typically, these approaches recursively iterate over all components with open requirements and search for matching components fulfilling these requirements. After the matching components are added to the incomplete deployment model and therefore, all requirements should be fulfilled, the deployment model is assumed to be deployable. In our motivating scenario, for example, the Adapter 2 may be implemented using Java and has a requirement for a Java runtime, which in turn has a requirement for a virtual machine. Therefore, the Java component with the requirement could be replaced by a stack containing the Java component, the Java runtime, and a virtual machine hosted by some cloud provider. If the grouping of step four does not allow a successful completion, the problematic component, e.g., a cloud provider, not supporting the computing power or privacy requirements, is blacklisted. Furthermore, the grouping is done again, without using this specific cloud provider.

F. Application Deployment

In step six of our method, the application is deployed in an automated manner using a declarative deployment system. This step is depending on the used declarative deployment model, i.e., the used modeling language must be supported by the deployment system. For our proposed method, any deployment system supporting a declarative deployment model enabling to model components and their relations is sufficient.

IV. Algorithms

For automating step four of our method, we propose a series of five algorithms. First, we describe the high-level algorithm and afterwards details about how to assign a location to a component based on the data flow. Finally, an algorithm is presented to select concrete providers for the components with assigned locations, based on requirements, such as privacy.

Algorithm 1 determineLocalityAndPlacement(dfm, d)

```
1:  // dfm => data flow model from step one
2:  // d => incomplete deployment model from step three
3:  while (assignmentPossible(d)) do
4:      assignToLocations(d, dfm)
5:      assignToProviders(d)
6:      try
7:          return completeModel(d)
8:      catch (fault)
9:          deleteLocationAssignments(d)
10:         addProvidersToBlackList(d, fault)
11:     end try
12: end while
13: throw fault
```

The high-level algorithm of our approach is described in Algorithm 1. It gets the data flow model *dfm* from step one and the incomplete deployment model *d*, which is generated in step two and enriched in step three, as input. The algorithm loops until a completed deployment model is created or no further valid provider assignment can be found (line 3). For this, the components are assigned to locations where they should reside to enable an efficient data flow during runtime (line 4). This location assignment of components is described in detail in Algorithm 2. Then, the components are assigned to providers, which provide infrastructure for the selected locations, based on requirements like privacy (line 5). The assignment to providers is described in Algorithm 5. Afterwards, the incomplete deployment model can be completed using the provider assignments (line 7). An example of such a completion was given in Subsection III-E. Furthermore, we extended the topology completion algorithm introduced by Hirmer et al. [42] to return useful fault messages if a completion fails, which are utilized by our algorithms. If the completion is successful, the algorithm terminates and returns the completed deployment model, which can be deployed in step six of our method. If the calculated provider assignment is not possible due to unfulfilled requirements, the completion throws a fault with the components and selected providers that led to the error. The location and provider assignments are deleted from the deployment model and the invalid providers are added to the blacklists of the components (lines 9 and 10). If the blacklists lead to the case that no further provider assignment is possible (line 3), the algorithm throws a fault message (line 13) and the user has to adapt the model manually. However, this case can only occur if none of the available providers can fulfill the requirements of a component, which would lead to an undeployable deployment model.

Algorithm 2 assignToLocations(dfm, d)

```
1: // dfm => data flow model from step one
2: // d => incomplete deployment model from step three
3: if (|{c | c ∈ components(d) : hasLocation(c)}| <= 1)
   then
4:     return assignAllComponentsToSameLocation(d)
5: end if
6: while (∃c ∈ components(d) : ¬hasLocation(c)) do
7:     assignComponent(dfm,getNextUnassignedComp(d))
8: end while
```

Algorithm 2 is used to assign components to locations based on the data flow and the Data Factor. The algorithm gets the data flow model and the incomplete deployment model as input. First, it checks if one or less components have already an assigned location (line 3). For this, the functions *components*, which returns the set of components in a deployment model and *hasLocation*, that checks if a component has a location label attached, are used. Components with an already assigned location could, for example, be the Research Data in Figure 2, that is stored at a certain location, or a component which has to be placed at a dedicated location, due to legal regulations. In the case, that just one or even none component is already assigned to a location, it is most efficient to assign all components to the same location, as this avoids to transfer data between different locations (line 4). Otherwise, an unassigned component is selected as long as there are such components (line 6) and assigned to a location (line 7) using Algorithm 3.

Algorithm 3 assignComponent(dfm, c)

```
1: // dfm => data flow model from step one
2: // c => selected component to assign a location to
3: locations := {}
4: for (pipe ∈ dfm : target(pipe) = c) do
5:     if (¬hasLocation(source(pipe))) then
6:         return
7:     end if
8:     label := getLocation(source(pipe))
9:     if (∃loc ∈ locations : π1(loc) = label) then
10:        new := (label, π2(loc) + pipeDataSize(pipe))
11:        locations ← locations \ {loc} ∪ {new}
12:    else
13:        new := (label, pipeDataSize(pipe))
14:        locations ← locations ∪ {new}
15:    end if
16: end for
17: for (loc ∈ locations : max(π2(loc)) do
18:    // is there a provider supporting the requirements
19:    // and locality group which is not blacklisted
20:    if (providerAvailable(c, π1(loc))) then
21:        setLocation(c, π1(loc))
22:        return
23:    end if
24:    locations ← locations \ {loc}
25: end for
```

Algorithm 3 gets the data flow model and the selected component c, which needs to be assigned as input. The component c must not have an incoming pipe for which the source is not yet assigned to enable the determination of the location from which the most data flows to component c. Therefore, all pipes which have component c as target are iterated (line 4) and if one of the sources has no assigned location, the algorithm terminates without an assignment (lines 5 to 7). In this case, the algorithm is invoked by Algorithm 2 with another component in the next iteration. There is always at least one component that fulfills this condition, as per assumption, the input data flow models are not allowed to contain loops. If the source has a location assigned, the name of the location is retrieved and it is checked if a tuple with the same location name was already added to the *locations* variable before (lines 8 and 9). Then, the data flow size for the current pipe is calculated using Algorithm 4 and added to the sum of flows from the old tuple, which is replaced by a new tuple with the same location name and the calculated sum (lines 10 and 11). Otherwise, a new tuple with the location name and the data flow size of the current pipe is created and added to locations (lines 13 and 14). Thus, locations contains the data size that needs to be transferred to the component c grouped by the location where the data comes from. Hence, the assignment of c to the location with the maximum data size in locations is most efficient in terms of data flow, as this leads to the minimal amount of data that needs to be transferred to c between different locations. For our location assignment, we assume, that the transfer of data within a single location, for instance, within the same data center is faster than between different locations. If this assignment is not possible, because there is no provider available in the location that is not on the black list of the component c, the location is removed from the set and the new maximum is calculated (lines 17 to 25).

Algorithm 4 pipeDataSize(pipe)

```
1: // pipe => pipe to calculate the data size flowing through
2: if (hasIncomingPipe(source(pipe))) then
3:     dataSize := 0
4:     for (p ∈ getIncomingPipes(source(pipe))) do
5:         dataSize ← dataSize + pipeDataSize(p)
6:     end for
7:     return dataSize * getDataFactor(source(pipe))
8: end if
9: return getDataSourceSize(source(pipe))
```

To determine the data size that needs to be transferred over a given pipe, Algorithm 4 uses the Data Factor and the data size of the transitively connected data sources. If the source filter of the given pipe has no incoming pipes, it is assumed to be a data source and therefore, its attribute specifying the size can be directly returned (line 9). Otherwise, the algorithm iterates over all incoming pipes (line 4), recursively calls the *pipeDataSize* function in order to get the amount of data that needs to be transferred over the pipe (line 5), and multiplies the result with the Data Factor of the source filter (line 7).

Algorithm 5 assignToProviders(d)

```
1: // d => deployment model with assigned locations
2: providers := {}
3: for (c ∈ components(d)) do
4:     for (p ∈ getViableProviders(c)) do
5:         if (∃prov ∈ providers : π1(prov) = p) then
6:             new := (p, π2(prov) ∪ {c})
7:             providers ← providers \ {prov} ∪ {new}
8:         else
9:             providers ← providers ∪ {(p, {c})}
10:        end if
11:    end for
12: end for
13: while (∃c1 ∈ components(d) : ¬hasProvider(c1)) do
14:    nextProv := p ∈ {prov ∈ providers :
       max(|{c | c ∈ π2(prov) : ¬hasProvider(c)}|)}
15:    for (c2 ∈ π2(nextProv) : ¬hasProvider(c2)) do
16:        setProvider(c2, π1(nextProv))
17:    end for
18: end while
```

After assigning components to locations, they have to be placed on a concrete provider that supports this location to enable the completion of the deployment model and the deployment of the application. This provider assignment is performed by Algorithm 5, which gets the incomplete deployment model *d* as input. First, it is determined which of the available providers supports which components that need to be assigned. For this, the algorithm iterates over each component in the deployment model d (line 3) and retrieves all providers that support the current component by calling the function *getViableProviders* (line 4). The function *getViableProviders* checks for each available provider in the assigned location, if the provider fulfills the non-functional requirements of the component. For example, it checks if the provider conforms to the privacy requirements of the component or if it can provide enough computing resources. However, this function can be extended with arbitrary selection logic for currently not considered requirements, for example, costs of the required resources or reliability [45]. If the provider was already added to the variable *providers*, the component is added to the set of supported components in the provider tuple (lines 5 to 7). Otherwise, a new tuple for the provider is created and only the current component is added to the set of supported components (lines 9). Afterwards, the components are assigned to a provider until all components of the deployment model have a valid assignment (lines 13 to 18). For this, the next provider is determined, which supports the most components that are not yet assigned (line 14) and these components are then assigned to the provider (line 15 to 17). Therefore, as many components as possible are assigned to the same provider to profit from a faster connection within the same data center or possible discounts. After all components are assigned to a provider, the still incomplete deployment model can now be completed accordingly to the results of this algorithm (cf. line 7 in Algorithm 1). Afterwards the application can be deployed.

V. Prototype, Validation and Evaluation

The concepts, presented in this paper build upon the *Essential Deployment Metamodel (EDMM)*, providing a technology-independent baseline for deployment automation research and a common understanding of declarative deployment models [17]. In the course of a systematic review, Wurster et al. [17] derived the essential parts supported by declarative deployment automation technologies and showed how they can be mapped to EDMM. In order to validate the practical feasibility of our approach, we use the deployment modeling language TOSCA [18], [19] for the following reasons: (i) it provides a vendor- and technology-agnostic modeling language, (ii) it is fully compliant with EDMM [17], and (iii) it is ontologically extensible [26]. Moreover, for implementing our prototype we extended the open-source OpenTOSCA ecosystem [46], which provides a tool-chain for modeling, orchestrating, and managing cloud and IoT applications.

In this section, we first describe the mapping from EDMM to TOSCA. Moreover, we present details of our prototypical implementation, for example, the system architecture containing the main components of the prototype as well as our used language for modeling data flows. Furthermore, we evaluated our prototype regarding its performance in different scenarios. For example, with different sized deployment models regarding the amount of contained, and thus, to be placed components and with different amounts of available providers.

A. Mapping to TOSCA

In EDMM, only a subset of entities of the TOSCA standard is used: Deployment models are called *service templates* in TOSCA, while components are called *node templates* and relations are called *relationship templates*. Accordingly, component types are called *node types* and relation types are called *relationship types* in TOSCA. Furthermore, the TOSCA standard defines some normative types for relations, for example, *hostedOn* and *connectsTo*, that compliant deployment systems have to support. In order to add additional information to relationship types as well as node types, for example, the username and password of a database or the port of a web server, in TOSCA so-called *properties* can be specified for node types as well as relationship types. Artifacts in EDMM are mapped to *deployment artifacts* in TOSCA. Deployment artifacts contain the business logic, therefore, they are required for the execution of a component. Deployment artifacts can be implemented using various technologies, for example, a Python file implementing an analytic service. Furthermore, operations in EDMM are mapped to parameterizable *management operations* in TOSCA. For example, a cloud provider or hypervisor node type usually provides management operations to create and terminate virtual machines. They are implemented by implementation artifacts, which can be implemented using various technologies as well. For instance, as a web service, a simple shell script, or by using a configuration management technology, such as Ansible, Chef, or Puppet.

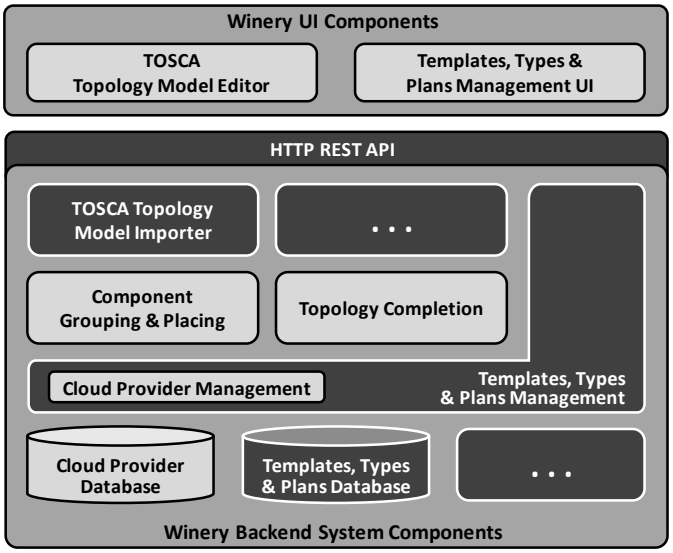

Fig. 4. Overview of the Winery architecture. Newly added and adapted components for implementing our prototype are highlighted in light grey.

B. System Architecture and Implementation Details

Winery is a graphical modeling tool for TOSCA and is part of the OpenTOSCA ecosystem [46], [47]. Figure 4 gives an overview of the Winery architecture highlighting newly added and adapted components (light grey). The both UI components *TOSCA Topology Model Editor* and *Templates, Types & Plans Management UI* enable the modeling of deployment models and the management of TOSCA elements, e.g., node types and relationship types. The Winery back-end has an *HTTP REST API* to enable the communication with the UI components. Components of the back-end are, e.g., the *TOSCA Topology Model Importer* for importing TOSCA-based deployment models. The TOSCA elements, and all further artifacts, are stored in corresponding databases, such as the *Templates, Types & Plans Database*. For the sake of simplicity, other components, e.g., an artifact generator or an exporter for exporting the executable deployment models are omitted here. In the middle of the architecture, there is the management layer, called *Templates, Types & Plans Management* enabling the management of templates, types, and other artifacts.

For implementing our approach, we adapted and extended Winery. In order to select cloud providers based on their location, for example, *US-West* and *US-East*, we enriched the *Cloud Provider Database* to be able to support this filter functionality. In this database, all components offered by cloud providers are grouped together by using TOSCA-compliant node templates as well as namespaces. Accordingly, the *Cloud Provider Management* component was adapted in order to reflect these changes. Our algorithms presented in Section IV are part of the newly developed and added *Component Grouping & Placing* component. The *Topology Completion* component was adapted as well, according to our changes proposed in Section IV. In order to enable the user to upload a data flow model and, therefore, to start the placing and completion process, the both *Winery UI Components* were also extended. After uploading the data flow model, it is parsed and an initial incomplete TOSCA-based deployment model is created based on the data flow model (cf. step two in Section III). Afterwards, the user can either modify the generated deployment model manually, for example, by adding additional components or specifying further requirements, or start the algorithms for the automated grouping, placing, as well as completion of the deployment model. After the algorithms are finished, the resulting deployment model is stored as a new service template in Winery, from where it can be exported as a *Cloud Service Archive (CSAR)*, which is a packaging format defined by the TOSCA standard. Such a CSAR can be deployed using a TOSCA-compliant deployment system, such as the open-source OpenTOSCA container[1] [48].

Listing 1 shows a simplified example of our pipes-and-filters oriented model for defining data flows. The model is XML-based and supports the both main elements *Pipe* and *Filter*. Filters have an *id* and a *type* and can contain properties, such as the *DataSize* in case of a database or the *DataFactor* in case of a processing component. Further properties can be specified, which are then adopted into the generated TOSCA deployment model, for example, a location where a component should be deployed or a database is already deployed. Pipes have an *id* as well as a *dataTransferType*, specifying the type of the connection, i.e., if the data is pulled from another component or database, or pushed to it. In our data flow model, the direction of the data flow is defined by specifying the *Source* and *Target* of the pipe. This information is important, since in TOSCA, a *connectsTo* relation only specifies which component initiates the connection, but not the direction of the data flow. Our prototypical implementation extending the functionality of Winery as well as further exemplary data flow models and generated deployment models are available on GitHub[2].

```
1  <DataFlow id="ExampleDataFlow" ...>
2     <Filters>
3        <Filter id="RD3" type="MySQL-DB">
4           <Properties>
5              <DataSize unit="TB">10</DataSize>
6              ...
7           </Properties>
8        </Filter>
9        <Filter id="Adapter-3" type="...">
10          <Properties>
11             <DataFactor>0.2</DataFactor>
12             ...
13          </Properties>
14       </Filter>
15       ...
16    </Filters>
17    <Pipes>
18       <Pipe id="P1" dataTransferType="pull">
19          <Source>RD3</Source>
20          <Target>Adapter-3</Target>
21       </Pipe>
22       ...
23    </Pipes>
24 </DataFlow>
```

Listing 1. Exemplary data flow definition.

[1]https://github.com/OpenTOSCA/container/
[2]https://github.com/OpenTOSCA/winery/

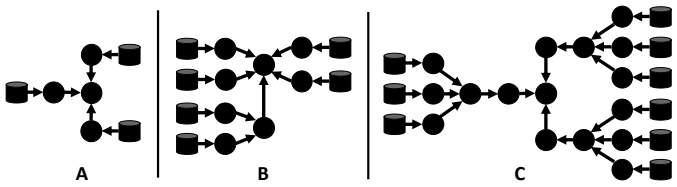

Fig. 5. Schematic structure of the three applications used for the evaluation.

C. Evaluation

In this section, we present the results of our evaluation regarding the time, required to determine the placement of the components of an application using our placement algorithm. For this purpose, we modeled three differently sized exemplary data processing applications. The first application contains 3 data sources and 4 data processing components, that need to be placed. The second application contains 6 data sources and 8 data processing components. And the third application contains 9 data sources and 16 data processing components. The schematic structure of these three exemplary applications used for evaluating our prototype is depicted in Figure 5. Furthermore, since the time required to place the components is also highly depending on the amount of possibilities where the components can be placed at, we created three scenarios. One scenario with 5 providers, each supporting 2 locations. A second scenario with 10 providers, each supporting 10 locations. And a third scenario with 100 providers, each supporting 10 locations. Therefore, in the first scenario, there are 10 potential placements for each component, in the second scenario, there are 100 potential placements for each component, and in the third scenario, there are 1000 potential placements for each component of the modeled applications.

Table I shows the results of our measurements. The median based on 10 measurements for each case is calculated. The table shows, that the amount of available providers and locations has the greatest influence on the required time of our placing algorithm, and not the size of the topologies. But, the influence of the size of the topologies increases with the number of possible placements. However, since our approach was developed for use in data-intensive applications, e.g., with several terabytes of data to be processed, the initial time for determining the placement of the components is rather short compared to the assumed runtime of such applications.

TABLE I
REQUIRED TIME TO PLACE ALL COMPONENTS

	Topology A (3D & 4C)*	**Topology B (6D & 8C)***	**Topology C (9D & 16C)***
5 Provider each with 2 Locations	494 ms	513 ms	524 ms
10 Provider each with 10 Locations	1043 ms	1080 ms	1148 ms
100 Provider each with 10 Locations	6117 ms	7052 ms	7169 ms

* D = data sources; C = data processing components

VI. RELATED WORK

Various placement approaches and algorithms have been proposed in the literature as surveyed in [49]–[51]. However, they are based on different optimization goals and assumptions, and thus, they are hardly comparable to each other. Therefore, in this section, we complete our discussion about related work, which we already discussed partially in Section II, and try to differentiate our approach from it.

The existing placement solutions aim to optimize various objectives, for example, resource usage [52]–[54], availability and reliability [55]–[57], energy consumption [58]–[60], costs [61]–[63], or response time [64]–[66]. Moreover, there are mono-objective approaches, such as [55] or [66] as well as multi-objective approaches, such as [52] or [63] available. Furthermore, the existing approaches rely on various methodologies, e.g., graph-theoretic methods [67]–[69], greedy algorithms [70]–[72], mathematical optimization approaches [73]–[75], as well as different kinds of heuristic approaches [76]–[78]. Our presented work can be classified as a graph-based and greedy approach. In the following, some research work with focus on the transfer of data are presented in more detail.

Many of the existing approaches, for example, the proposals of Lakshmanan et al. [79] and Gedik et al. [71], are designed specifically for clustered environments, with an assumed network latency of almost zero. Thus, these kind of approaches are not really suitable for cases, where the network latency can have a big impact on the performance and execution time of an application. For example, when components of data-intensive applications need to be distributed over various locations.

Other related work considering the network (e.g., [69], [80]–[82]), are trying to minimize the amount of exchanged data between computing nodes. While Fischer et al. [69] used a graph partitioning technique for this, Eidenbenz et al. [80] proposed a heuristic algorithm, also considering the transfer cost. Moreover, Aniello et al. [81] and Xu et. al [82] used a greedy best-fit heuristic in order to minimize the traffic between nodes. In [81], the main goal is to co-locate operators on a single node, based on the amount of communication between them. In [82], the location of the operators are assigned in descending order of incoming and outgoing traffic. In the area of sensor networks, Abrams and Liu [70] are using greedy placement heuristics in order to optimize the placement cost and reduce the amount of required bandwidth by considering tree-structured application graphs. Gu et al. [83] investigated how the communication cost for data stream processing applications in geo-distributed data centers can be minimized. Therefore, they explored inter-data center traffic cost diversities and considered virtual machine placement, proposing a heuristic-based algorithm. Regarding geo-distributed data stream processing systems, Zhu et al. [68] explicitly takes communication delays for their placement into account. However, in their proposed approach, they assume, that a node can only host a single operator at most, which seems to be an unrealistic assumption. With SpanEdge [84], Sajjad et al. proposed an approach especially for fog com-

puting, enabling programmers to specify the parts of their application that should be placed close to the data sources. The goal is to reduce the bandwidth consumption and the response latency. In order to optimize the provisioning resource costs, with MIST [75], Arkian et al. presented a data analytics scheme for placing IoT applications on fog resources as well.

While these introduced papers present different interesting approaches and algorithms for optimizing the placement regarding various optimizing goals, yet still the problem of a practical solution supporting the modeling, placement, deployment, as well as managing of arbitrary cloud and IoT applications remains without considering the applications' structure, its data flow, the placement of its components, and the deployment of the application itself in a holistic manner.

There are also practical oriented research works available about optimizing the distribution of cloud applications, e.g., the MOCCA method by Leymann et al. [85], the optimal distribution framework by Andrikopoulos et al. [86], and CloudMIG by Frey and Hasselbring [87]. Typically, they are optimizing based on the providers' offerings and mainly taking factors, such as costs or availability into account and are not considering relevant data processing factors, e.g., the data flow of the application in order to improve the performance of it.

With MOCHA [23], Rodríguez-Martínez and Roussopoulos presented a database middleware system designed to interconnect data sources distributed over the network. The goal of MOCHA is to improve performance by reducing the amount of data that needs to be transferred. Therefore, data reducing queries are moved to the location where the data resides, whereas in case of data inflating queries the data to be processed is moved to the query. However, besides the point that their approach is limited to databases, only selective queries stored in its catalog can be shipped by MOCHA, whereas in our approach, by using TOSCA-based deployment models, any component is supported to be shipped and deployed.

Regarding the transformation into executable TOSCA models, Hirmer and Mitschang [88] proposed an approach to transform non-executable data mashup plans into an executable format, by using categorized patterns and requirements, for example, robustness, security, and time sensitiveness. While their approach focuses on data processing and integration scenarios, their goal is not to improve the performance of the application by placing components depending on factors, for example, data size or data flow, but to ease the integration and execution of data-intensive cloud applications in general.

To our best knowledge, this work is the first deployment model-based approach, that enables the automated placement of components of data processing cloud and IoT applications in a practical way, that not only improves the performance of such applications depending on its data flow, but also supports its lifecycle, i.e., modeling, deployment and management of it. Moreover, in contrast to many mono-objective approaches, our proposed approach also supports the specification of further deployment restrictions and requirements, for example, restricting the deployment according to data privacy regulations.

VII. Conclusion and Future Work

In this paper, we presented a practical approach for automatically placing components of data processing cloud applications, among others, based on the data flow and different characteristics of the single components. Therefore, a novel method as well as algorithms supporting this method were presented. The approach aims to improve the performance of distributed applications, by placing data processing components as close as possible to the data sets that should be processed. The approach is validated by a prototypical implementation based on the TOSCA standard and the open-source ecosystem OpenTOSCA. However, since TOSCA can be mapped to the Essential Deployment Metamodel, our approach is not only restricted to TOSCA but can be applied to any other declarative deployment modeling language. We evaluated our prototype under different scenarios and investigated the impact of the size of the deployment models and the amount of available cloud providers on the performance of the prototype.

We plan to extend the approach to support more characteristics for the placement, for example, the current transfer speed between locations. In order to determine the transfer speed or transfer quality in general between different locations, we plan to generate some kind of testing deployment model first, for measuring the connection quality in between preselected locations. Moreover, we want to integrate an additional step about provisioning huge amounts of data into our method.

Acknowledgments

This work was partially funded by the BMWi project *Industrial Communication for Factories – IC4F* (01MA17008G), the DFG project *DiStOPT* (252975529), and the DFG's Excellence Initiative project *SimTech* (EXC 2075 – 390740016).

References

[1] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787 – 2805, 2010.

[2] F. Leymann, "Cloud Computing: The Next Revolution in IT," in *Proceedings of the 52th Photogrammetric Week*. Wichmann Verlag, Sep. 2009, pp. 3–12.

[3] V. Gunes, S. Peter, T. Givargis, and F. Vahid, "A Survey on Concepts, Applications, and Challenges in Cyber-Physical Systems," *KSII Transactions on Internet and Information Systems*, 2014.

[4] R. L. Villars, C. W. Olofson, and M. Eastwood, "Big data: What It Is and Why You Should Care," *White Paper, IDC*, vol. 14, pp. 1–14, 2011.

[5] Y. Guo, X. Hu, B. Hu, J. Cheng, M. Zhou, and R. Y. K. Kwok, "Mobile Cyber Physical Systems: Current Challenges and Future Networking Applications," *IEEE Access*, vol. 6, pp. 12 360–12 368, 2017.

[6] S. A. Haque, S. M. Aziz, and M. Rahman, "Review of Cyber-Physical System in Healthcare," *International Journal of Distributed Sensor Networks*, vol. 10, no. 4, p. 217415, 2014.

[7] F. Shrouf and G. Miragliotta, "Energy management based on Internet of Things: Practices and framework for adoption in production management," *Journal of Cleaner Production*, 2015.

[8] A. Szalay, "Extreme Data-Intensive Scientific Computing," *Computing in Science & Engineering*, vol. 13, no. 6, pp. 34–41, 2011.

[9] M. Falkenthal, U. Breitenbücher, M. Christ, C. Endres, A. W. Kempa-Liehr, F. Leymann, and M. Zimmermann, "Towards Function and Data Shipping in Manufacturing Environments : How Cloud Technologies leverage the 4th Industrial Revolution," in *Proceedings of the 10th Advanced Summer School on Service Oriented Computing*. IBM Research Division, 2016, pp. 16–25.

[10] M. Zimmermann, U. Breitenbücher, M. Falkenthal, F. Leymann, and K. Saatkamp, "Standards-based Function Shipping - How to use TOSCA for Shipping and Executing Data Analytics Software in Remote Manufacturing Environments," in *Proceedings of the 2017 IEEE 21st International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE Computer Society, 2017, pp. 50–60.

[11] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on Big Data," *Information Sciences*, vol. 275, pp. 314–347, 2014.

[12] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A View of Cloud Computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[13] M. Satyanarayanan, "The Emergence of Edge Computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[14] M. Zimmermann, U. Breitenbücher, and F. Leymann, "A Method and Programming Model for Developing Interacting Cloud Applications Based on the TOSCA Standard," in *Enterprise Information Systems*, ser. Lecture Notes in Business Information Processing, vol. 321. Springer International Publishing, 2018, pp. 265–290.

[15] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and J. Wettinger, "Integrated Cloud Application Provisioning: Interconnecting Service-Centric and Script-Centric Management Technologies," in *On the Move to Meaningful Internet Systems: OTM 2013 Conferences (CoopIS 2013)*. Springer, Sep. 2013, pp. 130–148.

[16] T. Eilam, M. H. Kalantar, A. V. Konstantinou, G. Pacifici, J. A. Pershing, and A. Agrawal, "Managing the Configuration Complexity of Distributed Applications in Internet Data Centers," *Communications Magazine*, vol. 44, no. 3, pp. 166–177, 2006.

[17] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, and J. Soldani, "The Essential Deployment Metamodel: A Systematic Review of Deployment Automation Technologies," *SICS Software-Intensive Cyber-Physical Systems*, 2019.

[18] OASIS, *Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0*, Organization for the Advancement of Structured Information Standards (OASIS), 2013.

[19] ——, *TOSCA Simple Profile in YAML Version 1.2*, Organization for the Advancement of Structured Information Standards (OASIS), 2019.

[20] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, "TOSCA: Portable Automated Deployment and Management of Cloud Applications," in *Advanced Web Services*. Springer, 2014, pp. 527–549.

[21] U. Srivastava, K. Munagala, and J. Widom, "Operator Placement for In-network Stream Query Processing," in *Proceedings of the Twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM, 2005, pp. 250–258.

[22] Z. Mahmood, "Data Location and Security Issues in Cloud Computing," in *2011 International Conference on Emerging Intelligent Data and Web Technologies*. IEEE, 2011, pp. 49–54.

[23] M. Rodríguez-Martínez and N. Roussopoulos, "MOCHA: A Self-extensible Database Middleware System for Distributed Data Sources," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. ACM, 2000, pp. 213–224.

[24] M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm," in *ACM Sigplan Notices*, vol. 26, no. 6. ACM, 1991, pp. 30–44.

[25] Z. Guo, G. Fox, and M. Zhou, "Investigation of Data Locality in MapReduce," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid)*. IEEE Computer Society, 2012, pp. 419–426.

[26] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, and G. Kappel, "A Systematic Review of Cloud Modeling Languages," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–38, 2018.

[27] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, and J. Wettinger, "Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications," in *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services, Feb. 2017, pp. 22–27.

[28] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, and J. Wettinger, "Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA," in *International Conference on Cloud Engineering (IC2E 2014)*. IEEE, Mar. 2014, pp. 87–96.

[29] M. J. Franklin, B. T. Jónsson, and D. Kossmann, "Performance Tradeoffs for Client-server Query Processing," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. ACM, 1996, pp. 149–160.

[30] D. W. Cornell, D. M. Dias, and S. Y. Philip, "On Multisystem Coupling Through Function Request Shipping," *IEEE Transactions on Software Engineering*, no. 10, pp. 1006–1017, 1986.

[31] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A Case for Intelligent RAM," *IEEE micro*, vol. 17, no. 2, pp. 34–44, 1997.

[32] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. Mckenzie, "Computational RAM: Implementing Processors in Memory," *IEEE Design Test of Computers*, vol. 16, no. 1, pp. 32–41, 1999.

[33] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-Data Processing: Insights from a MICRO-46 Workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, 2014.

[34] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[35] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, 2003, pp. 20–43.

[36] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST '10)*. IEEE Computer Society, 2010, pp. 1–10.

[37] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos, "Edge analytics in the internet of things," *IEEE Pervasive Computing*, vol. 14, no. 2, pp. 24–31, 2015.

[38] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. Tuecke, and I. Foster, "Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing," in *2001 18th IEEE Symposium on Mass Storage Systems and Technologies*. IEEE, 2001, pp. 13–28.

[39] E. Deelman and A. Chervenak, "Data Management Challenges of Data-Intensive Scientific Workflows," in *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, 2008, pp. 687–692.

[40] E. Deelman, T. Peterka, I. Altintas, C. D. Carothers, K. K. van Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, and J. Vetter, "The future of scientific workflows," *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 159–175, 2018.

[41] R. Meunier, "The Pipes and Filters Architecture," in *Pattern Languages of Program Design*, 1995, pp. 427–440.

[42] P. Hirmer, U. Breitenbücher, T. Binz, and F. Leymann, "Automatic Topology Completion of TOSCA-based Cloud Applications," in *GI-Jahrestagung*. GI, 2014, vol. P-251, pp. 247–258.

[43] K. Saatkamp, U. Breitenbücher, O. Kopp, and F. Leymann, "Topology Splitting and Matching for Multi-Cloud Deployments," in *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*. SciTePress, 2017, pp. 247–258.

[44] A. Panarello, U. Breitenbücher, F. Leymann, A. Puliafito, and M. Zimmermann, "Automating the Deployment of Multi-Cloud Applications in Federated Cloud Environments," in *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS'16)*. ICST, 2017, p. 194–201.

[45] Z. U. Rehman, F. K. Hussain, and O. K. Hussain, "Towards Multi-criteria Cloud Service Selection," in *2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*, 2011, pp. 44–48.

[46] U. Breitenbücher, C. Endres, K. Képes, O. Kopp, F. Leymann, S. Wagner, J. Wettinger, and M. Zimmermann, "The OpenTOSCA Ecosystem - Concepts & Tools," *European Space project on Smart Systems, Big Data, Future Internet -Towards Serving the Grand Societal Challenges -Volume 1: EPS Rome*, pp. 112–130, 2016.

[47] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, "Winery – A Modeling Tool for TOSCA-based Cloud Applications," in *Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC)*. Springer, 2013, pp. 700–704.

[48] M. Zimmermann, F. W. Baumann, M. Falkenthal, F. Leymann, and U. Odefey, "Automating the Provisioning and Integration of Analytics Tools with Data Resources in Industrial Environments using OpenTOSCA," in *Proceedings of the 2017 IEEE 21st International Enterprise Distributed Object Computing Conference Workshops and Demonstrations (EDOCW 2017)*. IEEE Computer Society, 2017, pp. 3–7.

[49] A. S. Alashaikh and E. A. Alanazi, "A Survey of Preferences in Virtual Machine Placement," *CoRR*, vol. abs/1907.07778, 2019.

[50] A. Brogi, S. Forti, C. Guerrero, and I. Lera, "How to Place Your Apps in the Fog - State of the Art and Open Challenges," *CoRR*, vol. abs/1901.05717, 2019.

[51] A. Laghrissi and T. Taleb, "A Survey on the Placement of Virtual Resources and Virtual Network Functions," *IEEE Communications Surveys Tutorials*, vol. 21, no. 2, pp. 1409–1434, 2019.

[52] Y. Gao, H. Guan, Z. Qi, Y. Hou, and L. Liu, "A multi-objective ant colony system algorithm for virtual machine placement in cloud computing," *Journal of Computer and System Sciences*, vol. 79, no. 8, pp. 1230 – 1242, 2013.

[53] Q. Zheng, R. Li, X. Li, N. Shah, J. Zhang, F. Tian, K.-M. Chao, and J. Li, "Virtual machine consolidated placement based on multi-objective biogeography-based optimization," *Future Generation Computer Systems*, vol. 54, pp. 95 – 122, 2016.

[54] X. Li, Z. Qian, S. Lu, and J. Wu, "Energy efficient virtual machine placement algorithm with balanced and improved resource utilization in a data center," *Mathematical and Computer Modelling*, vol. 58, no. 5, pp. 1222 – 1235, 2013.

[55] E. Bin, O. Biran, O. Boni, E. Hadad, E. K. Kolodner, Y. Moatti, and D. H. Lorenz, "Guaranteeing High Availability Goals for Virtual Machine Placement," in *2011 31st International Conference on Distributed Computing Systems*. IEEE, 2011, pp. 700–709.

[56] A. Zhou, S. Wang, B. Cheng, Z. Zheng, F. Yang, R. N. Chang, M. R. Lyu, and R. Buyya, "Cloud Service Reliability Enhancement via Virtual Machine Placement Optimization," *IEEE Transactions on Services Computing*, vol. 10, no. 6, pp. 902–913, 2017.

[57] F. Machida, Masahiro Kawato, and Y. Maeno, "Redundant Virtual Machine Placement for Fault-tolerant Consolidated Server Clusters," in *2010 IEEE Network Operations and Management Symposium - NOMS 2010*. IEEE, 2010, pp. 32–39.

[58] Z. Cao and S. Dong, "An energy-aware heuristic framework for virtual machine consolidation in Cloud computing," *The Journal of Supercomputing*, vol. 69, no. 1, pp. 429–451, 2014.

[59] X. Fu and C. Zhou, "Virtual machine selection and placement for dynamic consolidation in Cloud computing environment," *Frontiers of Computer Science*, vol. 9, no. 2, pp. 322–330, 2015.

[60] G. Wu, M. Tang, Y.-C. Tian, and W. Li, "Energy-Efficient Virtual Machine Placement in Data Centers by Genetic Algorithm," in *Neural Information Processing*, T. Huang, Z. Zeng, C. Li, and C. S. Leung, Eds. Springer, 2012, pp. 315–323.

[61] J. Araujo, P. Maciel, E. Andrade, G. Callou, V. Alves, and P. Cunha, "Decision making in cloud environments: an approach based on multiple-criteria decision analysis and stochastic models," *Journal of Cloud Computing*, vol. 7, no. 1, p. 7, 2018.

[62] V. Cardellini, E. Casalicchio, F. Lo Presti, and L. Silvestri, "SLA-aware Resource Management for Application Service Providers in the Cloud," in *2011 First International Symposium on Network Cloud Computing and Applications*. IEEE, 2011, pp. 20–27.

[63] H. N. Van, F. D. Tran, and J.-M. Menaud, "Autonomic virtual resource management for service hosting platforms," in *2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*. IEEE, 2009, pp. 1–8.

[64] M. Alicherry and T. V. Lakshman, "Optimizing Data Access Latencies in Cloud Systems by Intelligent Virtual Machine Placement," in *2013 Proceedings IEEE INFOCOM*. IEEE, 2013, pp. 647–655.

[65] J. T. Piao and J. Yan, "A Network-aware Virtual Machine Placement and Migration Approach in Cloud Computing," in *2010 Ninth International Conference on Grid and Cloud Computing*. IEEE, 2010, pp. 87–92.

[66] J. Kuo, H. Yang, and M. Tsai, "Optimal Approximation Algorithm of Virtual Machine Placement for Data Latency Minimization in Cloud Systems," in *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. IEEE, 2014, pp. 1303–1311.

[67] J. Li, A. Deshpande, and S. Khuller, "Minimizing Communication Cost in Distributed Multi-query Processing," in *25th International Conference on Data Engineering*. IEEE, 2009, pp. 772–783.

[68] Q. Zhu and G. Agrawal, "Resource Allocation for Distributed Streaming Applications," in *37th International Conference on Parallel Processing*. IEEE, 2008, pp. 414–421.

[69] L. Fischer, T. Scharrenbach, and A. Bernstein, "Scalable Linked Data Stream Processing via Network-Aware Workload Scheduling," in *Proceedings of the 9th International Conference on Scalable Semantic Web Knowledge Base Systems*, vol. 1046, 2013, pp. 81–96.

[70] Z. Abrams and Jie Liu, "Greedy is Good: On Service Tree Placement for In-Network Stream Processing," in *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*. IEEE, 2006, pp. 72–72.

[71] B. Gedik, H. G. Özsema, and Ö. Öztürk, "Pipelined fission for stream programs with dynamic selectivity and partitioned state," *Journal of Parallel and Distributed Computing*, vol. 96, pp. 106–120, 2016.

[72] T. Li, J. Tang, and J. Xu, "A predictive scheduling framework for fast and distributed stream data processing," in *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 2015, pp. 333–338.

[73] S. Rizou, F. Dürr, and K. Rothermel, "Solving the Multi-Operator Placement Problem in Large-Scale Operator Networks," in *2010 Proceedings of 19th International Conference on Computer Communications and Networks*. IEEE, 2010, pp. 1–6.

[74] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal Operator Placement for Distributed Stream Processing Applications," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems (DEBS '16)*. ACM, 2016, pp. 69–80.

[75] H. R. Arkian, A. Diyanat, and A. Pourkhalili, "MIST: Fog-based data analytics scheme with cost-efficient resource provisioning for IoT crowdsensing applications," *Journal of Network and Computer Applications*, vol. 82, pp. 152 – 165, 2017.

[76] X. Meng, V. Pappas, and L. Zhang, "Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement," in *2010 Proceedings IEEE INFOCOM*. IEEE, 2010, pp. 1–9.

[77] A. Chatzistergiou and S. D. Viglas, "Fast Heuristics for Near-Optimal Task Allocation in Data Stream Processing over Clusters," in *Proceedings of the 23rd ACM International Conference on Information and Knowledge Management (CIKM '14)*. ACM, 2014, pp. 1579–1588.

[78] P. Smirnov, M. Melnik, and D. Nasonov, "Performance-aware scheduling of streaming applications using genetic algorithm," *Procedia Computer Science*, vol. 108, pp. 2240–2249, 2017.

[79] G. T. Lakshmanan, Y. Li, and R. Strom, "Placement of Replicated Tasks for Distributed Stream Processing Systems," in *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems (DEBS '10)*. ACM, 2010, pp. 128–139.

[80] R. Eidenbenz and T. Locher, "Task Allocation for Distributed Stream Processing," in *The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[81] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive Online Scheduling in Storm," in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems (DEBS '13)*. ACM, 2013, pp. 207–218.

[82] J. Xu, Z. Chen, J. Tang, and S. Su, "T-Storm: Traffic-Aware Online Scheduling in Storm," in *2014 IEEE 34th International Conference on Distributed Computing Systems*. IEEE, 2014, pp. 535–544.

[83] L. Gu, D. Zeng, S. Guo, Y. Xiang, and J. Hu, "A General Communication Cost Optimization Framework for Big Data Stream Processing in Geo-Distributed Data Centers," *IEEE Transactions on Computers*, vol. 65, no. 1, pp. 19–29, 2016.

[84] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov, "SpanEdge: Towards Unifying Stream Processing over Central and Near-the-Edge Data Centers," in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2016, pp. 168–178.

[85] F. Leymann, C. Fehling, R. Mietzner, A. Nowak, and S. Dustdar, "Moving Applications to the Cloud: An Approach based on Application Model Enrichment," *International Journal of Cooperative Information Systems*, vol. 20, no. 3, pp. 307–356, 2011.

[86] V. Andrikopoulos, S. Gómez Sáez, F. Leymann, and J. Wettinger, "Optimal Distribution of Applications in the Cloud," in *Proceedings of the 26th International Conference on Advanced Information Systems Engineering (CAiSE)*. Springer, Jun. 2014, pp. 75–90.

[87] S. Frey and W. Hasselbring, "The CloudMIG Approach: Model-Based Migration of Software Systems to Cloud-Optimized Applications," *International Journal On Advances in Software*, vol. 4, no. 3&4, pp. 342–353, 2011.

[88] P. Hirmer and B. Mitschang, "FlexMash – Flexible Data Mashups Based on Pattern-Based Model Transformation," in *Rapid Mashup Development Tools*. Springer, 2016, pp. 12–30.