Leveraging Data-Driven Infrastructure Management to Facilitate AIOps for Big Data Applications and Operations

Richard McCreadie, John Soldatos, Jonathan Fuerst, Mauricio Fadel Argerich, George Kousiouris, Jean-Didier Totow, Antonio Castillo Nieto, Bernat Quesada Navidad, Dimosthenis Kyriazis, Craig Macdonald, and Iadh Ounis

Abstract As institutions increasingly shift to distributed and containerized application deployments on remote heterogeneous cloud/cluster infrastructures, the cost and difficulty of efficiently managing and maintaining data-intensive applications have risen. A new emerging solution to this issue is Data-Driven Infrastructure Management (DDIM), where the decisions regarding the management of resources are taken based on data aspects and operations (both on the infrastructure and on the application levels). This chapter will introduce readers to the core concepts underpinning DDIM, based on experience gained from development of the Kubernetes-based BigDataStack DDIM platform (https://bigdatastack.eu/). This chapter involves multiple important BDV topics, including development, deployment, and operations for cluster/cloud-based big data applications, as well as data-driven analytics and artificial intelligence for smart automated infrastructure self-management. Readers will gain important insights into how next-generation

R. McCreadie (✉) · J. Soldatos · C. Macdonald · I. Ounis
University of Glasgow, Glasgow, United Kingdom
e-mail: richard.mcCreadie@glasgow.ac.uk; john.soldatos@glasgow.ac.uk; craig.macdonald@glasgow.ac.uk; iadh.ounis@glasgow.ac.uk

J. Fuerst · M. F. Argerich
NEC Laboratories Europe, Heidelberg, Germany
e-mail: jonathan.fuerst@neclab.eu; mauricio.fadel@neclab.eu

J.-D. Totow · D. Kyriazis
University of Piraeus, Pireas, Greece
e-mail: totow@unipi.gr; dimos@unipi.gr

G. Kousiouris
Harokopio University of Athens, Moschato, Greece
e-mail: gkousiou@hua.gr

A. C. Nieto · B. Q. Navidad
ATOS/ATOS Worldline, Chennai, India
e-mail: antonio.castillo@atos.net; bernat.quesada@worldline.com

E. Curry et al. (eds.), *Technologies and Applications for Big Data Value*,
https://doi.org/10.1007/978-3-030-78307-5_7

DDIM platforms function, as well as how they can be used in practical deployments to improve quality of service for Big Data Applications.

This chapter relates to the technical priority Data Processing Architectures of the European Big Data Value Strategic Research & Innovation Agenda [33], as well as the Data Processing Architectures horizontal and Engineering and DevOps for building Big Data Value vertical concerns. The chapter relates to the Reasoning and Decision Making cross-sectorial technology enablers of the AI, Data and Robotics Strategic Research, Innovation & Deployment Agenda [34].

Keywords Data processing architectures · Engineering and DevOps for big data

1 Introduction to Data-Driven Infrastructure

For nearly a decade, advances in cloud computing and infrastructure virtualization have revolutionized the development, deployment, and operation of enterprise applications. As a prominent example, the advent of containers and operating system (OS) virtualization facilitates the packaging of complex applications within isolated environments, in ways that raise the abstraction level towards application developers, as well as boosting cost effectiveness and deployment flexibility [32]. Likewise, microservice architectures enable the provision of applications through composite services that can be developed and deployed independently by different IT teams [8]. In this context, modern industrial organizations are realizing a gradual shift from conventional static and fragmented physical systems to more dynamic cloud-based environments that combine resources from different on-premises and cloud environments. As a result of better application isolation and virtualized environments, basic semi-autonomous management of applications has become possible. Indeed, current cloud/cluster management platforms can natively move applications across physical machines in response to hardware failures as well as perform scaling actions based on simple rules. However, while useful, such basic autonomous decision making is insufficient given increasingly prevalent complex big data applications [31]. In particular, such applications rely on complex interdependent service ecosystems and stress the underlying hardware to its limits and whose properties and workloads can vary greatly based on the changing state of the world [4].

Hence, there is an increasing need for smarter infrastructure management solutions, which will be able to collect, process, analyse, and correlate data from different systems, modules, and applications that comprise modern virtualized infrastructures. In this direction, recent works have developed and demonstrated novel Data-Driven Infrastructure Management (DDIM) solutions. For instance, experimental DDIM solutions exist that process infrastructure data streams to detect errors and failures in physical systems (e.g. [16, 29]). In other cases, more advanced data mining techniques like machine learning have been used to detect anomalies in the operation of cloud infrastructures (e.g. [10]). This is complemented by works on data-driven performance management of cloud infrastructures [9] and resources [5]. However, these solutions address individual optimization and resilience concerns, where a more holistic Data-Driven Infrastructure Management approach is required.

This chapter introduces a holistic DDIM approach, based on outcomes of BigDataStack [15], a new end-to-end DDIM solution. The main contributions and innovations of the presented DDIM approach are the following: (i) Data-oriented modelling of applications and operations by analysing and predicting the corresponding data flows and required data services, their interdependencies with the application micro-services and the required underlying resources for the respective data services and operations. Allowing the identification of the applications data-related properties and their data needs, it enables the provision of specific performance and quality guarantees. (ii) Infrastructure management decisions based on the data aspects and the data operations governing and affecting the interdependencies between storage, compute, and network resources, going beyond the consideration of only computational requirements. The proposed DDIM leverages AI and machine learning techniques to enable more efficient and more adaptive management of the infrastructure, known as the AIOps (Artificial Intelligence Operations) paradigm, considering the applications, resources, and data properties across all resource management decisions (e.g. deployment configurations optimizing data operations, orchestration of application and data services, storage and analytics distribution across resources, etc.). (iii) Optimized runtime adaptations for complex data-intensive applications throughout their full lifecycle, from the detection of fault and performance issues to the (re)configuration of the infrastructure towards optimal Quality of Service (QoS) according to data properties. The latter is achieved through techniques that enable monitoring of all aspects (i.e. applications, analytics, data operations, and resources) and enforcement of optimal runtime adaptation actions through dynamic orchestration that addresses not only resources but also data operations and data services adaptations. The proposed DDIM approach has been deployed and validated in the context of three enterprise application environments with pragmatic workloads for different vertical industries, including retail, insurance, and shipping.

The rest of the chapter is structured to incrementally introduce the key building blocks that enable DDIM in BigDataStack. In particular, Sect. 2 introduces the core modelling of user applications, their environment, as well as additional concepts needed to realize DDIM. Section 3 discusses how profiling of user applications can be performed prior to putting them in production. In Sect. 4, we discuss how to monitor applications in production, as well as measure quality of service. Section 5 introduces AIOps decision making within BigDataStack, while Sect. 6 discusses how to operationalize the decisions made. Finally, we provide an illustrative example of DDIM in action within BigDataStack for a real use case in Sect. 7.

2 Modelling Data-Driven Applications

At a fundamental level, DDIM is concerned with altering the deployment of a user's application (to maintain some target state or quality of service objectives) [14, 22]. As such, the first question that needs to be asked is 'what is a user application?' From a practical perspective, a user application is comprised of one or more programs, each needing a compatible environment to run within. However, we cannot assume any language or framework is being used if a general solution is needed. To solve this, the programs comprising the user's application and associated environments need to be encapsulated into packaged units that are readily deployable without additional manual effort. There are two common solutions to this, namely virtual machines and containers [26]. For the purposes of DDIM, containers are generally a better solution, as they have a smaller footprint, have fewer computational overheads and are faster to deploy/alter at runtime [26]. We assume container-based deployment for the remainder of this chapter.

Given container-based deployment, we can now model the user application in terms of containers. It is good practice for a container to be mono-task, that is each container runs only a single program, as this simplifies smarter scheduling on the physical hardware [3]. A user application is then comprised of a series of containers, each performing a different role. DDIM systems then configure, deploy, and maintain these containers over large hardware clusters or clouds. There are a range of commercial and open-source container management solutions currently available, such as Docker Swarm and Kubernetes. The primary function of these solutions is to schedule containers onto cloud or cluster infrastructures. This involves finding machines with sufficient resources for each container, copying the container (image) to those machine(s), mounting any required attached storage resources, setting up networks for communication, starting the container(s), and finally monitoring the container states and restarting them if necessary. At the time of writing, the most popular container management solution is the open-source Kubernetes platform, which is what we will assume is being used moving forward. We discuss the most important Kubernetes concepts for DDIM systems below.

2.1 Application Modelling Concepts

Pods, Deployments, and Jobs For the purposes of modelling the user application in a containerized cloud/cluster, it is reasonable to consider an application to be comprised of a series of 'Pods', where a pod is comprised of one or more containers. A pod abstraction here exists to provide a means to group multiple containers into a single unit that can be deployed and managed together. In our experience, it is useful to distinguish pods along two dimensions: *lifespan* and *statefulness*. First, considering pod lifespan, 'continuous' pods are those that are expected to run indefinitely, representing permanent services which may be user-facing (e.g. a web host). Meanwhile, 'finite' pods are those that are aimed at performing a task, and will end once that task is complete (e.g. a batch learning or analytics task). Continuous pods in effect have an implied Service-Level Objective, that is that they must be kept running regardless of changes to the underlying infrastructure (e.g. due to hardware failures), while finite pods do not. In Kubernetes, continuous pods are managed using ‘Deployment’ objects, while finite pods are represented as ‘Job’ objects. Second, considering statefulness, a pod can be stateless meaning that it does not retain any data between requests made to it. This type of pod is the easiest to manage, it holds no critical data that could be lost if the pod needs to be restarted or moved and can often be replicated without issue. On the other hand, stateful pods maintain or build up data over time, which is lost if the pod fails or is killed. As such, the ‘cost’ of altering the configuration of an application that is comprised of stateful pods can be high, as data is lost when those pods are moved or restarted. In this case, the lost data needs to either be regenerated requiring more time and computational power or may simply be unrecoverable if the underlying input that created the data is no longer available. For this reason, it is recommended that application architects design their system to use only stateless pods where possible.

Services and Routes When scheduling a pod, a machine with the needed resources is only selected at runtime, meaning the network address of that pod cannot be known before then. Furthermore, that address may not be static, as changes in the infrastructure environment may result in the pod being lost/deleted and then a new copy spawned on a different physical node. This complicates the configuration of user applications, as it is commonplace for user programs to expect to be preconfigured with static URLs or IP addresses when two components need to talk to one another. A ‘Service’ is the solution to this issue, as it is a special entity in Kubernetes in that it has a static URL. Traffic directed at a service will then be forwarded to one or more pods based on a service-to-pod mapping, which is updated over time if changes occur and can also be used for load-balancing requests across multiple copies of a pod. A service can be paired with a ‘Route’ object to produce an external end-point, enabling requests from the outside world to reach a pod.

Volumes and Volume Claims Containers by their nature are transient, that is their state is lost when they exit. Hence, most pods need some form of persistent storage to write to, for example for writing the final output of a batch operation or as a means to achieve statelessness by reading/writing all state information to an external store. This is handled in Kubernetes using volumes. A volume is simply a definition of a directory within a storage medium that can be mounted to one or more containers, such as an NFS directory or distributed file system like HDFS. However, the available storage amounts and storage types available will vary from cluster to cluster, and as such it is not good practice to directly specify a volume, as this ties the pod to only clusters with that exact volume. Instead, Volume Claims exist, which represent a generic request for a desired amount and type of storage. If a pod specifies a volume claim, Kubernetes will attempt to automatically provide the requested amount and types of storage from its available pool of volumes. In this way, an application can still obtain storage, even if the application owner does not know what exact storage volumes are available on the target cluster.

2.2 Data-Driven Infrastructure Management Concepts

In this section, we will describe additional concepts that are required for DDIM systems to function based on experience from developing BigDataStack, namely Pod Level Objectives, Resource Templates, Workloads, and Actions.

Pod Level Objectives For DDIM systems to meaningfully function, they need a set of objectives to achieve, representing the needs of the application owner. Given the application modelling discussed above, we can consider that an application component, represented by a running pod (and created via a Deployment or Job), could have zero or more objectives associated with it. In the literature, such objectives are typically referred to as Service-Level Objectives (SLOs) [24]. However, this may be somewhat confusing as in containerized environments a ‘Service’ means something different, as such we will instead refer to these as Pod-Level Objectives (PLOs). A PLO defines a quality of service (QoS) target for a pod to achieve, such as ‘cost less than 1.2 U.S. dollars per hour’, or ‘provide response time less than 300 ms’. A QoS target is comprised of three parts: (1) a metric (e.g. response time), (2) a target value (e.g. 300 ms), and (3) a comparator (e.g. less than). Note an implicit assumption here is that the specified metric is measurable for the pod, either because the pod exports it (e.g. response time), or it can be calculated for the pod by a different component (e.g. cost). If so, a PLO can be checked by comparing the current measured metric value against the QoS target, resulting in a pass or failure. PLO failures are the primary drivers of DDIM systems, as they indicate that changes in the user application or data infrastructure are needed.

Resource Templates To launch a Deployment or Job in a cluster or cloud environment, sufficient computational resources need to be provided, that is CPU capacity, system memory, and potentially GPUs or other specialized hardware [6]. The modelling of resources assigned to a pod is a critical part of DDIM systems, as a lack (or in some cases excess) of such resources is the primary cause of PLO failures. Moreover, the resource allocation for individual pods are often a variable that the DDIM system has control over and hence can manage automatically. In theory, resources, such as allocated system memory, are continuous variables, where any value could be set up to a maximum defined by the target cluster. However, predefined Resource Templates that instead define aggregate ‘bundles’ of resources for a fixed cost are very common, such as Amazon EC2 Instances. Resource Templates exist as they both simplify the resource selection process for the application owner, while also enabling the cluster owner to divide their available resources in a modular fashion. A basic Resource Template needs to specify CPU capacity and system memory for a pod, as all pods require some amount of these. A Resource Template may optionally list more specialized hardware, such as Nvidia GPUs or Quantum cores based on the requirements of the application and cluster/cloud support available.

Workloads Another factor that DDIM systems often need to consider is the application environment. Continuous applications will typically be serving requests, either from users or other applications. Meanwhile, finite applications most commonly will be concerned with processing very large datasets. We can consider these environmental factors as sources of workload that is placed on pods, that is they quantify the properties of the input to those pods over time. In the case of continuous applications, this might manifest in the number of API calls being made per second and may vary over time (e.g. for user-facing applications distinct day-night cycles are commonly observable). On the other hand, for finite applications, the size of the dataset or database table(s) being processed can be considered to define the workload. Some types of DDIM systems will model such workloads and may even condition their PLOs upon them, for example if the number of requests is less than 500 per second, then response latency should be less than 100 ms.

Actions The final aspect of a user application that is critical to enable DDIM systems is how they can be altered. For a DDIM system to function, it requires a finite set of actions that it can perform for an application. In effect, these actions form a ‘toolbox’ that the DDIM system can use to rectify PLO failures. Relatively simple actions in containerized environments might include adding/removing replicas or altering the Resource Template for a pod. However, as applications become more complex, associated actions often require multiple steps. For example, scaling a data-intensive API service might involve first replicating an underlying database to provide increased throughput, followed by a reconfiguration step joining the new database instance into the swarm, followed by a scaling action on the API pods to make use of the increased capacity. Moreover, as actions become more complex, the time taken to complete them will grow, hence DDIM systems need to track what actions are currently in progress and their state to enable intelligent decision making.

3 Application Performance Modelling

Much of the emphasis for DDIM systems is on how to manage user applications (semi-)automatically post-deployment. However, some types of DDIM systems, including BigDataStack, support optional functionality to analyse the user application pre-deployment. The core concept here is to gain some understanding of the expected properties of the application, which can be used later to enable more intelligent decision making. From a practical perspective, this is achieved via benchmark tooling, whereby application components can be temporarily deployed with resource templates, subjected to a variety of predefined workloads, in a parameter sweep fashion, and their performance characteristics measured and recorded. This enables the quantification at later stages of the effects of a workload on the QoS metrics of this service.

How Does Benchmarking Function? Fundamentally, benchmarking has three main aspects: (1) the deployment of part or all of the user application with a defined set of resources; (2) the creation of a workload for that application; and (3) measurement of the observed performance characteristics. Deployment in a containerized environment is greatly simplified, as the containers needing tested can be deployed as pods directly upon the infrastructure if correctly configured. On the other hand, generating a workload for the application is more complicated. For continuous pods, typically an external service is needed to generate artificial requests, for example to simulate user traffic patterns. Meanwhile, for finite pods, use of a standard dataset or data sample is used for benchmarking. Finally, measurement of the performance characteristics of an application typically comes in three forms: application exported metrics, infrastructure reported metrics for the application, and metrics about the infrastructure itself. As the name suggests, application exported metrics are metrics directly reported by the application itself based on logging integrated into it by the application engineer, for example request processing time for a website host. Infrastructure metrics about the application represent monitoring the management platform (e.g. Kubernetes) that is passively performing, such as pod-level CPU and system memory consumption. Finally, infrastructure metrics (which are not typically available on public clouds) can provide information about the wider state of the cluster/cloud providing insights into how busy it is. Metrics here might include node-level CPU and memory allocation and information about node-to-node network traffic.

Benchmarking Tools in BigDataStack BigDataStack incorporates a Benchmarking-as-a-Service framework (Flexibench), developed in the context of the project, that exploits baseline tools such as Apache Jmeter and OLTPBench and orchestrates their operation towards a target endpoint (database or otherwise). Flexibench retrieves the necessary setup (type of client to use, workload to launch, desired rate of requests, etc.) via a REST-based interface or a UI-based interface and undertakes their execution. Multiple features are supported such as parallel versus isolated execution of the experiment, trace-driven load injection (based on historical data files), inclusion of the defined data service in the process, or load injection towards an external datapoint. The tool offers REST interfaces for test monitoring and result retrieval and is based on Node-RED, a visual flow programming environment of node.js.

Predictive Benchmarking While undertaking actual benchmarking is the most accurate way to determine the performance characteristics for an application, it can be costly and time consuming to implement as the application needs to be physically deployed and the parameter search space may be extensive. An alternative to this is *predictive benchmarking*. The idea is to use machine learning to estimate what the outcome of a benchmark run would look like, by considering the benchmarking outcomes from other similar application deployments. BigDataStack also supports the creation of predictive benchmarking models via Flexibench through the same UI- or REST-based environment, therefore integrating the two processes (result acquisition and model creation). The creation is performed following a REST- or UI-based request, in which the test series is defined, as well as other parameters that are necessary for result acquisition (such as the identifiers of related services to use for predictions). The framework retrieves the relevant training data and launches a containerized version of an automated model creation algorithm defined in [13].

The resultant model is subsequently validated, and the respective results are made available for the user to examine via the UI. Following, the model can be queried if the relevant input values are supplied (e.g. type and size of workload, resource used, etc.) and the predicted QoS metric (e.g. response time, throughput, etc.) will be returned to the user.

4 Metric Collection and Quality of Service Monitoring

To enable the validation of the application PLOs that act as the triggers for DDIM, constant quantitative performance measurement for pods is required. Therefore, metrics must be collected, stored, and exposed. Indeed, the performance of applications running on platforms like BigDataStack is impacted by many factors, such as infrastructure state, data transaction speeds, and application resourcing. For this reason, the monitoring engine of BigDataStack was developed using a triple monitoring approach. By triple monitoring we mean the collection of performance indicators from: (1) the infrastructure, (2) data transactions, and (3) applications exported metrics.

Metric Collection Strategies Any metric monitoring system can be considered as comprised of multiple agents and a manager. The agents perform measurements and prepare metrics for collection by the manager, which might be a pod within the user application, a database, or Kubernetes itself. Most commonly, the manager periodically requests metrics from the agents (known as polling) via standardized metric end-points. An interval (scraping interval) then defines how frequently the metrics are collected. An alternative approach is to have the agents directly post metric updates to the manager or an intermediate storage location (known as the push method). This can be useful for applications or services that do not/cannot expose a metrics end-point that a manager can connect to. BigDataStack supports both polling and push methods for data collection. The measurement data collected by the manager is then typically held in a time-series database, enabling persistent storage of performance history over time.

Triple Monitoring in BigDataStack BigDataStack's monitoring solution is based on Prometheus, which is the official monitoring tool of Kubernetes. Prometheus requires a definition of target endpoints exposing metrics in its configuration file. However, manually defining this for the many applications that are managed by BigDataStack is infeasible. Instead, BigDataStack exploits the service discovery functionality of Prometheus to automatically configure metric collection from new pods as they are deployed. Under the hood, in this scenario Prometheus periodically communicates with the Kubernetes API to retrieve a list of port end-points exposed by running pods in the BigDataStack managed namespaces, and checks them to see if they export metrics in a format that Prometheus can understand. Some applications may wish to control their own metric collection via their own Prometheus instance. In this context, the triple monitoring engine is able to aggregate metrics collection from multiple Prometheus instances concurrently using Thanos. The monitoring engine of BigDataStack adopts a federated model where several Prometheus instances can be added dynamically for reducing the total scraping duration, thus allowing the collection of very large volumes of data points from a large number of sources. This is coupled with metrics compression by aggregation, minimizing the overhead when working with Big Data applications. For the purposes of DDIM, three different groups of metrics are collected by the triple monitoring engine: infrastructure information, data operations information (i.e. data produced, exchanged, and analysed by applications), and all the data involved in database transactions and object storage operations. Since these metrics are produced by applications with different purposes, specifications, functionalities, and technologies, the triple monitoring engine integrates a data sanitizer to prepare incoming measurements such that they conform with a standard specification. The triple monitoring engine also provides an output REST API for exposing data to all services, as well as a publish/subscription service that enables the streaming consumption of measurements by other applications.

Quality of Service Evaluation Within a DDIM system, the core reason to collect measurements about a user application and infrastructure is to enable evaluation of application Quality of Service (QoS). QoS represents the degree to which the application is meeting the user needs. More precisely, QoS evaluation is concerned with guaranteeing the compliance of a given KPI (Key Performance Indicator) as defined in one or more PLOs (Pod-Level Objectives), typically for a given time period or window. When a user requests a service from BigDataStack, a minimum QoS is agreed between the user and the system, expressed as PLOs. At runtime, metric data is collected by the Triple Monitoring Engine and evaluated against these PLOs to determine whether the application is (or in some cases soon will be) failing to meet the user needs. Such failures can then be used to trigger orchestration actions aimed at rectifying such failures, as discussed in the next section.

5 Automated Decision Making

Having discussed the monitoring of user applications and how to determine when failures have occurred that need to be rectified, we next discuss how DDIM systems can solve these issues through automatic service orchestration.

QoS as an Optimization Problem Service orchestration in the context of Big Data has usually the goal of a Quality of Service (QoS) or Quality of Experience (QoE) sensitive optimization [27]. This optimization problem, under varying contexts, is complex and considered an NP-hard problem [17]. Previous approaches have tackled this optimization problem with heuristics [7, 30] or genetic algorithms [2] that aim to find near-optimal configurations and service compositions, such as configurations respecting overall QoS/QoE constraints, while maximizing a QoS utility function. The composition of services in current cloud-edge Big Data/AI applications usually follows a pipeline pattern in which stream and batch data (potentially recorded at the edge) is processed by multiple components in order to derive the desired results. These new pipelines add new requirements and challenges to service orchestration as they inherently contain complex trade-offs between computation and performance (e.g. resulting accuracy), and errors introduced in early components cascade through the overall pipeline, affecting end-to-end performance and making it impossible to treat the problem as an independent orchestration of components. Initial approaches have addressed the orchestration problem with reasoning across the pipeline components in a probabilistic manner, allowing the user to manually decide the adequate trade-off [25].

We model QoS as a constrained optimization problem. Specifically, (1) we model requirements as constraints, for example to process documents with an end-to-end latency less or equal than 1 s or to run at a cost of less or equal than 10 $ per hour, and (2) we model service performance, such as precision, accuracy, or battery consumption, as objective. There is an important difference between the objective and the constraints: whereas the constraints define a minimum or maximum value for the variable involved (e.g. latency, cost, etc.), the objective does not have a minimum or maximum value expected. In this way, we can define the service requirements as:

$$\begin{aligned} \underset{\theta}{\text{maximize}} \quad & O(\theta) \\ \text{subject to} \quad & c_i(\theta) \leq C_i,\ i = 1, \ldots, N \end{aligned}$$

where:

- θ: is the configuration of parameters used for all of the operators.
- $O(\theta)$: represents the objective of the service, which is determined by the configuration of parameters used.
- $c_i(\theta)$: is a constraint to the service (such as latency), also determined by θ.
- C_i: is the constraint target (e.g. 1 s).
- N: is the total number of constraints.

The developer is in charge of defining the service requirements along with the metrics to monitor them, as well as the parameters that can be adapted and the values they can assume. During runtime, the system is in charge of finding the best configuration of parameter values that maximize (or minimize) the objective while respecting the constraints.

Optimization via Reinforcement Learning Recently, reinforcement learning (RL) has been successfully applied to node selection for execution [19] as well as optimization of overall pipelines using, among others, meta-reasoning techniques to ensure an overall optimization of the pipeline [1, 18, 21]. A key issue with RL-based approaches is the bootstrapping problem, that is how to obtain sufficient performance from the first moment the agent begins to operate. A simple solution is to explore the state space randomly, but this approach is usually time consuming and costly when the state/action space is large, as illustrated in [1]. An alternative approach is to gain experience more cheaply and faster via a sandbox simulation. With enough computational resources, it is possible to produce large volumes of experience data in a short time period, but it is difficult to ensure that the simulated experiences are realistic enough to reflect an actual cloud/cluster environment. To reduce the cost of training RL agents, some works have examined how to leverage external knowledge to improve their exploration efficiency. For example, in [11, 23] prior knowledge like pretrained models and policies are used to bootstrap the exploration phase of an RL agent. However, this type of prior knowledge still originates in previous training and is limited by the availability of such data. In BigDataStack, we leverage Reinforcement Learning in conjunction with expert knowledge to drive service orchestration decisions, as discussed next.

Dynamic Orchestrator in BigDataStack The Dynamic Orchestrator (DO) works alongside the Triple Monitoring Engine (TME) to monitor and trigger the redeployment of BigDataStack applications during runtime to ensure they comply with their Service-Level Objectives (SLOs). The DO receives and manages monitoring requests when a new application or service is deployed into the BigDataStack platform, informing the TME and the Quality of Service (QoS) component what metrics and PLOs should be monitored. When any violation to these PLOs occur, the QoS informs the DO, and the DO is in charge of deciding what redeployment change is necessary, if any. In BigDataStack, we developed flexible orchestration logic that can be applied to any kind of application by applying Reinforcement Learning (RL) that leverages external knowledge to bootstrap the exploration phase of the RL agent. We call this method Tutor4RL. For Tutor4RL, we have modified the RL framework by adding a component we call the Tutor. The tutor possesses external knowledge and helps the agent to improve its decisions, especially in the initial phase of learning when the agent is inexperienced. In each step, the tutor takes as input the state of the environment and outputs the action to take, in a similar way to the agent's policy. However, the tutor is implemented as a series of programmable functions that can be defined by domain experts and interacts with the agent during the training phase. We call these functions knowledge functions and they can be of two types:

- *Constraint functions:* are programmable functions that constrain the selection of actions in a given state, 'disabling' certain options that must not be taken by the agent. For example, if the developer of the application has decided a maximum budget for the application, even if the application load is high and this could be fixed by adding more resources to the deployment, this should not be done if the budget of the user has already reached its maximum.
- *Guide functions:* are programmable functions that express domain heuristics that the agent will use to guide its decisions, especially in moments of high uncertainty, for example at the start of the learning process or when an unseen state is given. Each guide function takes the current RL state and reward as the inputs and then outputs a vector to represent the weight of each preferred action according to the encoded domain knowledge. For example, a developer could

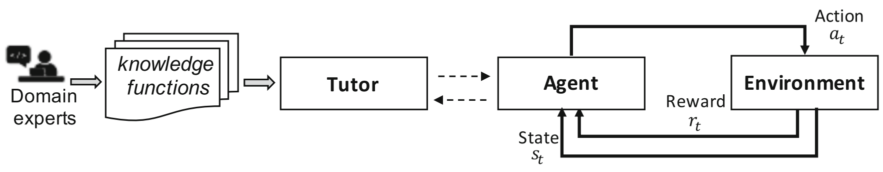

Fig. 1 Overall Working of Tutor4RL

create a guide function that detects the number of current users for an application, and if the number is higher than a certain threshold, more resources might be deployed for the application (Fig. 1).

The benefit coming from using Tutor4RL is twofold. First, during training, the tutor enables a faster bootstrapping to a reasonable performance level. Furthermore, the experience generated by the tutor is important because it provides examples of good behaviour, as it already uses domain knowledge for its decisions. Second, the knowledge the tutor provides does not need to be perfect or extensive. The tutor might have partial knowledge about the environment, that is know what should be done in certain cases only, or might not have a perfectly accurate knowledge about what actions should be taken for a given state. Instead, the tutor provides some 'rules of thumb' the agent can follow during training, and based on experience, the agent can improve upon the decisions of the tutor, achieving a higher reward than it.

Learning What Actions to Take Within BigDataStack, the application engineer (i.e. the domain expert) defines sample guide and constraint functions for the learner. These functions encode domain knowledge of the developer that can guide decision making. Indeed, for some applications these guide functions will be sufficient to manage the application without further effort. On the other hand, for cases where the guide functions are insufficient, reinforcement learning can be enabled. During RF training (i.e. pre-production), the RL agent will experiment with the different available alteration actions that can be performed (discussed in the next section), learning how each affects the metrics tracked by the triple monitoring engine, as well as the downstream PLOs. After a period of exploration, the RF agent can be deployed with the application in production, where it will intelligently manage the application by triggering the optimal alteration action or actions in response to PLO failures.

How Effective is Automatic Service Orchestration? We have compared Tutor4RL performance against vanilla DQN [20] in a scenario where the DO is in charge of controlling two metrics: cost per hour (which varies according to resources used by application) and response time. These are two opposite objectives: if we increase the use of resources, the response time decreases but the cost per hour increases, and if we decrease the use of resources, the opposite is true. However, the SLOs specify thresholds for each metric: cost per hour should be less or equal to \$0.03 and response time should be less than 200 ms. The DO must find the sweet spot that satisfies these two SLOs as long as the application allows it. In

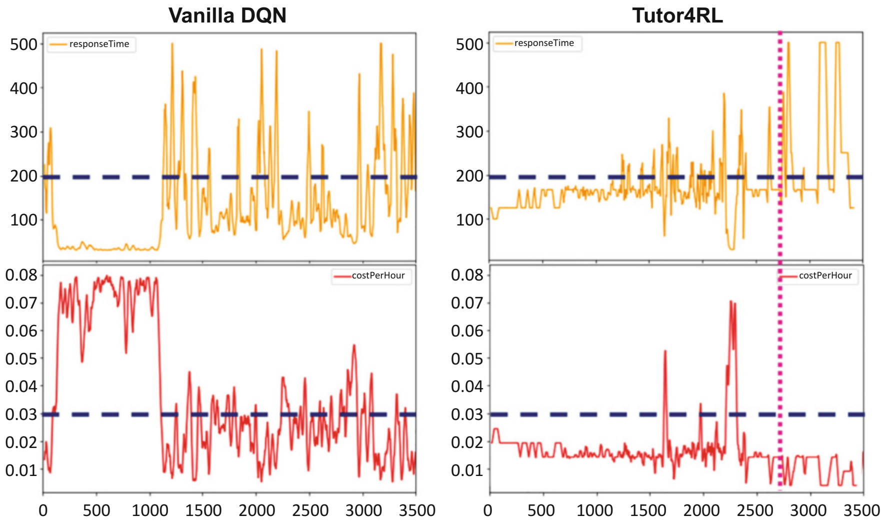

Fig. 2 Tutor4RL Performance compared to Vanilla DQN [20]. DO performance to manage 2 SLOs: costPerHour <0.03 and responseTime <200. Vanilla DQN is shown on the left, while Tutor4RL, with 2 guides and 1 constrain, is shown on the right. The horizontal blue dashed lines show the SLO threshold for the metrics and the pink dotted line show the moment in which guides are not used anymore

fact, it might happen that the application load is too high, and then there is no way of satisfying both SLOs, in these cases the DO behaviour will tend to find the configuration that violates the SLOs proportionally less. However, we believe these are corner cases in which even a human might not be sure what to do, and therefore we have not evaluated the DO's performance in these situations. In Fig. 2, we see the performance of the vanilla DQN agent (left) and the Tutor4RL agent (right) for managing this scenario with two SLOs. Note that on the images we have marked with horizontal lines the thresholds for SLOs and with a vertical line, the moment in which the guide functions from the Tutor are not used anymore, until that point the functions are used on and off with a diminishing frequency from 0.9 to 0. As we can see, the Tutor4RL agent performs better than the vanilla agent by achieving a better satisfaction of SLOs. We still see that once the guides are completely abandoned, the agent commits some mistakes, but it can quickly correct its error. We can avoid this by adding constraints such as not changing the deployment configuration if no SLO is violated, but we wanted to show a case in which the agent is free in its actions and therefore show its learned behaviour better.

6 Operationalizing Application Alterations

Once the decision to alter the user application (as an attempt to rectify a failing PLO) has been made, and the action to perform has been decided, the DDIM needs to operationalize that alteration. In this section we discuss the types of alterations that are possible, as well as how these can be encoded as actions within BigDataStack.

6.1 Types of Alteration Actions

Depending on the type and complexity of the deployed application, as well as the level of permissions that the DDIM has with the infrastructure management system (e.g. Kubernetes), there can be a wide range of alteration actions that might be performed. A useful way to structure actions is based on what problem they aim to solve. In general, there are four common types of problems that can arise as follows:

Insufficient Resources for a Pod Based on unexpectedly poor performance reported by a pod in conjunction with maximized utilization for that pod, the DDIM might decide that a pod needs more resources. This can be solved by the allocation of a different (in terms of resources) pod, which actually refers to a larger Resource Template for that pod. Notably, in Kubernetes this is a destructive action, that is it will involve the pod being killed. For continuous applications, this is typically not an issue, as the pods involved are stateless. However, for Jobs performing this type of operation might result in all progress up to that point being lost. In either scenario, the alteration action involves the launching of a new copy of the target pod with the new larger Resource Template, then halting the previous pod once the new one reaches running status.

Insufficient Application Capacity In this scenario, one or more pods may be reporting unexpectedly poor performance, but the resources are not saturated for any of the application pods. This would indicate that the existing pods are working correctly, but they are unable to keep up with the current workload. The solution here is to scale-up the application to increase its capacity, if supported by the application. For simple applications, this might only involve increasing the number of replicas for one of the pods, which is a non-destructive action (load-balancing across the replicas can be handled by a Service automatically). However, for complex applications this may require multiple steps and can be destructive. For example, consider the scaling of an Apache Spark Streaming application. First, the number of Spark workers needs to be increased, which can be handled by a simple replication factor change on the worker pods. These new workers will be automatically added to the 'Spark Cluster' and will show ready for work. However, the streaming application running on that Spark cluster will not automatically scale to use the additional workers. Instead, the application must be killed and then resubmitted to the Spark cluster before the workers will be allocated to the application.

Insufficient Data Availability Not all issues that might cause PLO failures are necessarily application-related. In this scenario, the DDIM might observe unexpectedly poor performance for the application and at the same time a saturation of resources for one or more data infrastructure components in use by that application (e.g. a database). In this case, the most efficient response would be to scale the data infrastructure components (in the example above, the database) to provide the required additional capacity. This is typically a costly action to perform, in terms of both time and resources. First, appropriate data storage volumes need to be created to hold the replicated data. Second, new infrastructure pods need to be started, and when they are ready, the associated data needs to be imported. Note that this import process is typically performed from some archive service, not via copying from currently running data infrastructure instances, as those are already overloaded by application requests.

Insufficient Network Bandwidth The last case represents failures in the networking/communication infrastructure. Within distributed cluster environments, pod-to-pod as well as external service-to-pod communications are channelled across physical communication links, which can become saturated. A classical example of this is a Denial of Service attack, where external servers attempt to overload an application with requests. While there is no easy fix for this type of failure, specific DDIM setups can control traffic prioritization. In this case, some portion of low-priority in-flight network traffic will be dropped to free up bandwidth for high-priority traffic. This is achieved by re-configuring the cluster's network routing policies based on detected in-bound workloads.

6.2 *Considerations When Modelling Alteration Actions*

Given the above adaptation scenarios and their possible solutions, it is clear that DDIM solutions require the means to perform complex alteration actions at runtime. There are four key aspects that need to be considered when modelling these alteration actions:

Sequencing and State Dependencies First, a single alteration action can be a complex affair that involves multiple steps. Moreover, the individual steps may have dependencies that require progression to wait until particular application states are achieved. For example, for a scaling action on the data infrastructure, the alteration action is not complete when the new infrastructure pods have started, but rather once the data has finished importing. Hence, an action must be seen as a composite set of lower-level operations that together form the desired action, where both the application and operations have states that can be tracked to determine when new operations can start (as well as when the action as a whole is complete).

Actions Are Application (Type) Dependent Second, it is worth stating that the available alteration action set is not the same across applications. While there are standard operations that are built into management platforms like Kubernetes for controlling factors such as pod replication, just because an operation is technically valid, it does not mean that performing the operation would be efficient for the current application. For example, for deployments where state loss is not an issue, the option to perform destructive runtime resource template changes may be desirable. But such an option might not be effective or cost too much for jobs where progress is reset when the underlying pod is restarted, even if it is possible to do so [28]. Moreover, any reasonably complex application will need support for multistage alteration actions that are not supported natively by existing management platforms. On the other hand, it is notable that applications that are of a similar type, for example those that use a common framework like Apache Spark or Ray, may be able to share actions, enabling common action sets to be shared among similar applications.

Available Actions Can Change Over Time The set of available actions for an application may change depending on that application's state, or the states of associated actions being performed. In the simplest case, once an alteration action is triggered, it makes sense to remove that action from the available set until it completes, as the DDIM system should wait to see the outcome of that alteration before attempting the same action again. Meanwhile, in more complex scenarios, the application engineer might want more control over what actions the DDIM system will consider under different conditions, effectively providing the DDIM system expert knowledge of what actions are reasonable given different application states.

Actions Are Data-Dependent Finally, the potential different actions to be applied heavily depend on the data: Data volumes might highlight the need for altering the data services settings tackling both storage (e.g. dynamic split or dynamic migration of data regions) and analytics (e.g. triggering of scalability of a real-time complex event processing engine). In this context, the DDIM system should monitor and account for the data operations and the related workloads (of data-intensive applications) and trigger the optimum adaptation actions during runtime - including deployment configurations and orchestration decisions.

6.3 Alteration Actions in BigDataStack

BigDataStack delivers a solution for enabling complex alteration actions: the Realization Engine. At its core, the Realization Engine has three roles: (1) to act as a central point of reference by storing all application-related information, (2) to maintain up-to-date state information about the application alteration actions, and (3) to enable triggering and subsequent operationalization of alteration actions for each application. Figure 3 illustrates the application model within BigDataStack. In particular, under this model, the user account or 'owner' owns one or more applications and can also define metrics. A single application has a state, zero or

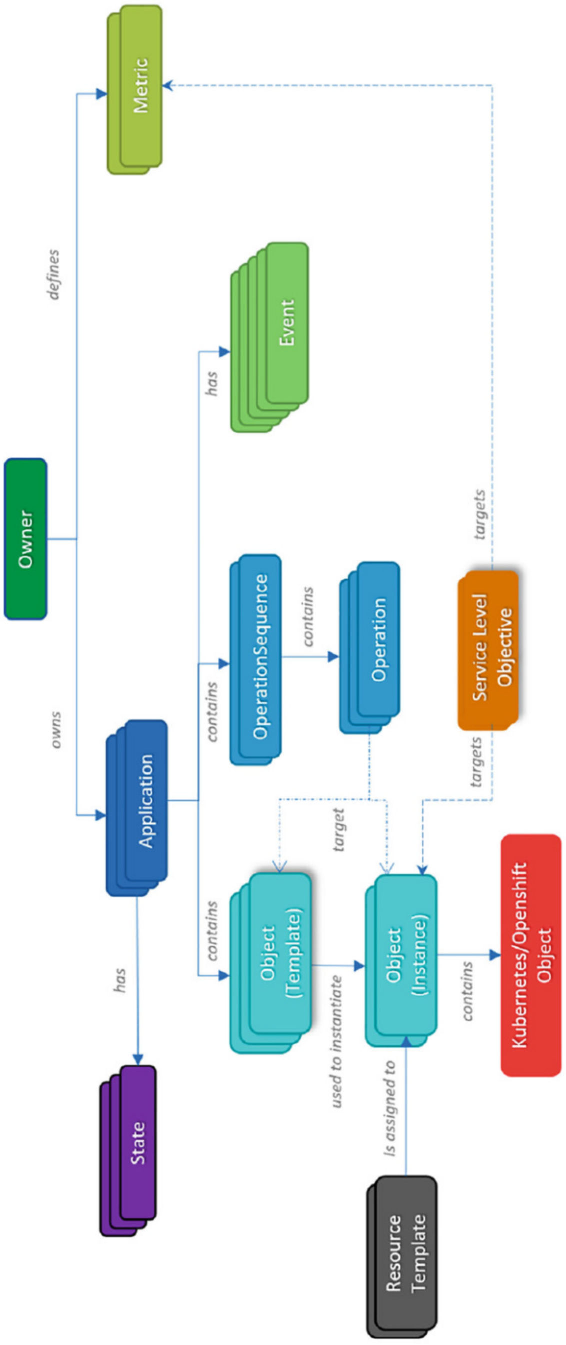

Fig. 3 BigDataStack realization engine application model

more object (templates) representing the different components of the application, zero or more operation sequences representing actions that can be performed for the application, and a series of events generated about the application. An object template (application component) can be instantiated multiple times, producing object instances. Object instances may have an associated resource template describing the resources assigned to that object. An object instance contains a definition of an underlying Kubernetes or OpenShift object that contains the deployment information. Operation sequences represent actions to perform on the application and contain multiple atomic operations. An operation targets either an object template or instance, performing alteration or deployment actions upon it. Service-level objectives can be attached to an object instance, which tracks a metric exported by or about that object.

In this way, alteration actions and their stages have an explicit representation, that is as Operation Sequences and Operations, respectively, which are associated with an application. The application engineer can define any number of operation sequences for their application by specifying the list of operations to perform and their configuration (e.g. the target object(s) for that operation), and can condition the availability of each operation sequence upon the application's current state. An Operation conceptionally performs a single change to a BigDataStack Object. Examples of operations include: *Deploy*, *Execute Command On*, *Build*, *Delete*, and *Wait For*. When an operation sequence is triggered within the Realization Engine, internally this first takes the operation sequence template and generates an instance from it that can be run and monitored separately. The Realization Engine then creates a new Pod object on the Kubernetes cluster to run the operation sequence targeting this new instance. Once the Pod object has been created, the responsibility for that operation sequence is passed to the Pod. Once the new Pod reaches running state, it will first load the target operation sequence instance, and subsequently it will process each operation within the sequence in order. State updates are reported to and stored by the Realization Engine within the operation sequence instance itself, enabling BigDataStack mechanisms to track progress for it.

7 Example Use-Case: Live Grocery Recommendation

Finally, having discussed all of the concepts and components of DDIM systems, in this section we summarize how this comes together in BigDataStack to enable a use case: the connected consumer. The connected consumer use case utilizes the BigDataStack environment to implement and offer a recommender system for the grocery market. All of the data that are used for training the analytic algorithms of the use case are corporate data provided by one of the top food retailers companies in Spain. The goal from a DDIM perspective is to host and manage all aspects of the underlying grocery recommendation system.

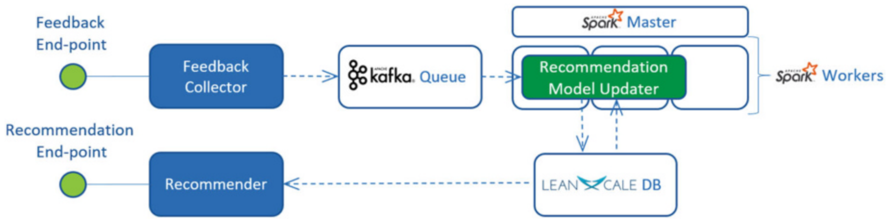

Fig. 4 Overview of the connected consumer grocery recommender

The Grocery Recommendation System Fig. 4 provides an illustration of the grocery recommendation system used by the use case. Each box in this diagram represents a pod. From an external perspective, this system has two end-points: recommendations, which respond with recommended products for a user, and feedback, which receives click and purchase events generated by the user. When a logged-in user opens a homepage on the grocery company store-front, a request is sent to recommendations end-point, which retrieves cached grocery recommendations for that user from a transactional database (in the specific case, LeanXcale [12] has been used as a transactional database). Meanwhile, when a user clicks or purchases a product, an event is sent to the feedback end-point, which reformats the data and sends it via Kafka queue into the main recommendation update component. This is a continuous application component that runs in parallel over multiple Apache Spark workers, which upon receiving item feedback for a user, updates their cached recommendations in real time based on that feedback.

Metrics, Pod Level Objectives, and Workload Within this application, the user cares about three main factors: (1) the response time for recommendations (e.g. less than 100 ms), (2) the delay between feedback being recorded and when the user's recommendations will finish updating (e.g. less than 1 s) and (3) the total cost of the system (e.g. less than 2 US dollars per hour). The volume of both recommendation requests and feedback varies over the course of each day (following the day/night cycle for mainland Europe), with periodic bursts of activity that correspond to flash sales.

Available Actions To achieve these goals, the DDIM system has access to the following actions it can take: (1) increase/decrease replicas for the Feedback Collector, Kafka and/or the Recommender, (2) increase/decrease table replication within the transactional database, and (3) increase the number of Spark workers the recommendation update service has access to.

A Day of Data-Driven Infrastructure Management In the early hours of the morning, the DDIM system will have the application running in a minimal configuration (typically only one instance of each pod) to minimize cost (0.5 USD/hour) when there is little traffic. Around 7am, the workload begins to increase, as online shoppers order groceries before starting work. The triple monitoring engine reports that response times and feedback updates are still within acceptable bounds. At

8:30am, quality of service monitoring reports a POS failure on recommendation updates of 1.1 s on average. The dynamic orchestrator determines based on resource usage that the bottleneck is in the recommendation updater. It triggers an enlargement of the spark cluster, adding an additional worker, and once ready performs a rapid restart of the recommendation updater such that it now leverages both workers. The POS failure is rectified, although cost per hour has increased (0.6 USD/hour). As traffic approaches its peak around mid-day, a second POS failure is reported this time on average recommendation response time (115 ms). In response, the dynamic orchestrator increases the replication factor of the recommender component. After a short delay to collect new measurements, the POS failure is still not resolved, and hence the dynamic orchestrator instructs the database to increase its table replication on the assumption that is where the bottleneck is occurring. This process takes around 10 min to complete, during which the dynamic orchestrator refrains from taking further action for that POS failure. Once the change action has completed, the response time decreases once again, and the POS failure is resolved. Cost per hour is now 1.2 USD/hour. After 6pm, the workloads decrease, and in response, the RL agent within the dynamic orchestrator experiments with decreasing the replication on the recommender to save cost, which results in a POS failure on recommendation response time and so rolls back the change. It tries again a couple of hours later, which does not result in any POS failure. As workloads continue to decrease as the day ends, the dynamic orchestrator instructs the database and recommendation updater to reduce their table replication and number of workers respectively.

8 Conclusions

The future of infrastructure management will be data driven, leveraging recent advances in Big Data Analytics and AI technologies to provide exceptional automation and optimization in the management of diverse virtualized software-defined cloud resources. This chapter has introduced the building blocks of data-driven infrastructure management (DDIM) systems in general and a new DDIM platform, BigDataStack, in particular. Contrary to state-of-the-art DDIM systems that focus on specific optimization aspects (e.g. fault detection or resource allocation), the BigDataStack platform takes a holistic, end-to-end approach to optimizing the configuration of cloud environments. Specifically, BigDataStack-based DDIM includes cutting-edge functionalities in three complementary areas:

- **Deep Application and Action Modelling**: Through deep data-oriented modelling, BigDataStack maintains a more complete view of both data services, their corresponding data flows and the alteration/action space for applications, backed by a metric and state monitoring system that is both efficient and scalable for use with high-parallelism Big Data applications. Indeed, a key take-home message of BigDataStack is that DDIM systems need to not just model applications, but the actions that can be performed on them and have the ability to monitor and interpret the impacts from those actions.

- **Intelligent AIOps Decision Making**: As cluster/cloud infrastructures have become better instrumented and easier to manipulate programmatically, it is now possible to have AI agents learn how to manage such infrastructures as well as humans can (for a fraction of the cost). AI-based decision making enables fast and adaptive management of the infrastructure based on real-time data about the running applications, cluster resources, and data being processed.
- **Complex Multistage Action Management**: BigDataStack provides an atomic operation set from which a wide variety of complex actions can be constructed. Once defined, such actions form templates that can be used to drive fully automated complex runtime adaptations by the AIOps system, enabling end-to-end automated DDIM.

BigDataStack has been validated and evaluated in different real-life environments, including retail, insurance, and shipping, illustrating the efficiency, cost-effectiveness, and flexibility of its DDIM approach. Overall, BigDataStack has provided a novel AIOps showcase, which demonstrates the potential of DDIM for monitoring, analysing, and optimizing the deployment of cloud applications. Moving forward, we aim to extend BigDataStack with support for novel types of cloud computing resources and services, such as the Function-as-a-Service (FaaS) paradigm. Indeed, via FaaS support, we believe big data value chains can be more efficiently enabled, as function-level DDIM is easier to reason about than broader application-level DDIM, increasing precision while also reducing the time-to-convergence to an effective configuration. We also plan to investigate the real-world impact of service performance degradation vs. the cost of maintaining services at different quality of service levels, with the aim of developing future AI models for optimizing this trade-off for the business owner.

Acknowledgments The research leading to these results has received funding from the European Community's Horizon 2020 research and innovation programme under grant agreement n° 779747 (BigDataStack).

References

1. Argerich, M. F., Cheng, B., & Fürst, J. (2019). Reinforcement learning based orchestration for elastic services. In Proceedings of the 2019 IEEE 5th World Forum on Internet of Things (WF-IoT), IEEE (pp. 352–357).
2. Canfora, G., Di Penta, M., Esposito, R., & Villani, M. L. (2005). An approach for qos-aware service composition based on genetic algorithms. In Proceedings of the 7th annual conference on Genetic and evolutionary computation (pp. 1069–1075).
3. Chung, A., Park, J. W., & Ganger, G. R. (2018). Stratus: Cost-aware container scheduling in the public cloud. In Proceedings of the ACM symposium on cloud computing (pp. 121–134).
4. Demchenko, Y., Filiposka, S., Tuminauskas, R., Mishev, A., Baumann, K., Regvart, D., & Breach, T. (2015). Enabling automated network services provisioning for cloud based applications using zero touch provisioning. In 2015 IEEE/ACM 8th international conference on utility and cloud computing (UCC). IEEE (pp. 458–464).

5. Eramo, V., Cianfrani, A., Catena, T., Polverini, M., & Lavacca, F. (2019). Reconfiguration of cloud and bandwidth resources in NFV architectures based on segment routing control/data plane. In Proceedings of the 2019 21st international conference on transparent optical networks (ICTON), IEEE (pp. 1–5).
6. Fard, M. V., Sahafi, A., Rahmani, A. M., & Mashhadi, P. S. (2020). Resource allocation mechanisms in cloud computing: A systematic literature review. *IET Software*.
7. Fürst, J., Argerich, M. F., Cheng, B., & Papageorgiou, A. (2018). Elastic services for edge computing. In Proceedings of the 2018 14th international conference on network and service management (CNSM) , IEEE (pp. 358–362).
8. Gan, Y., & Delimitrou, C. (2018). The architectural implications of cloud microservices. *IEEE Computer Architecture Letters, 17*(2), 155–158.
9. Grabarnik, G. Y., Tortonesi, M., & Shwartz, L. (2016). Data-driven cloud-based it services performance forecasting. In Proceedings of the 2016 IEEE international conference on big data (Big Data), IEEE (pp. 2081–2086).
10. Gulenko, A., Wallschläger, M., Schmidt, F., Kao, O., & Liu, F. (2016). Evaluating machine learning algorithms for anomaly detection in clouds. In Proceedings of the 2016 IEEE international conference on big data (Big Data), IEEE (pp. 2716–2721).
11. Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., Horgan, D., Quan, J., Sendonaris, A., Dulac-Arnold, G., et al. (2017). Deep q-learning from demonstrations. arXiv preprint arXiv:1704.03732.
12. Kolev, B., Levchenko, O., Pacitti, E., Valduriez, P., Vilaça, R., Gonçalves, R. C., Jiménez-Peris, R., & Kranas, P. (2018). Parallel Polyglot query processing on heterogeneous cloud data stores with LeanXcale. In IEEE BigData, Seattle, United States, IEEE (p. 10).
13. Kousiouris, G., Menychtas, A., Kyriazis, D., Konstanteli, K., Gogouvitis, S. V., Katsaros, G., & Varvarigou, T. A. (2012). Parametric design and performance analysis of a decoupled service-oriented prediction framework based on embedded numerical software. *IEEE Transactions on Services Computing, 6*(4), 511–524.
14. Kraemer, A., Maziero, C., Richard, O., & Trystram, D. (2018). Reducing the number of response time service level objective violations by a cloud-hpc convergence scheduler. *Concurrency and Computation: Practice and Experience, 30*(12), e4352.
15. Kyriazis, D., Doulkeridis, C., Gouvas, P., Jimenez-Peris, R., Ferrer, A. J., Kallipolitis, L., Kranas, P., Kousiouris, G., Macdonald, C., McCreadie, R., et al. (2018). Bigdatastack: A holistic data-driven stack for big data applications and operations. In Proceedings of the 2018 IEEE international congress on big data (BigData Congress), IEEE (pp. 237–241).
16. Lin, Q., Hsieh, K., Dang, Y., Zhang, H., Sui, K., Xu, Y., Lou, J.-G., Li, C., Wu, Y., Yao, R., et al. (2018). Predicting node failure in cloud service systems. In Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering (pp. 480–490).
17. Mabrouk, N. B., Beauche, S., Kuznetsova, E., Georgantas, N., & Issarny, V. (2009). Qos-aware service composition in dynamic service oriented environments. In ACM/IFIP/USENIX international conference on distributed systems platforms and open distributed processing, Springer (pp. 123–142)
18. Mao, H., Alizadeh, M., Menache, I., & Kandula, S. (2016). Resource management with deep reinforcement learning. In Proceedings of the 15th ACM workshop on hot topics in networks (pp. 50–56).
19. Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., & Dean, J. (2017). Device placement optimization with reinforcement learning. arXiv preprint arXiv:1706.04972.
20. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature, 518*(7540), 529–533.
21. Modi, A., Dey, D., Agarwal, A., Swaminathan, A., Nushi, B., Andrist, S., & Horvitz, E. (2019). Metareasoning in modular software systems: On-the-fly configuration using reinforcement learning with rich contextual representations. arXiv preprint arXiv:1905.05179.

22. Mohamed, M., Anya, O., Sakairi, T., Tata, S., Mandagere, N., & Ludwig, H. (2016). The RSLA framework: Monitoring and enforcement of service level agreements for cloud services. In Proceedings of the 2016 IEEE international conference on services computing (SCC), IEEE (pp. 625–632).
23. Moreno, D. L., Regueiro, C. V., Iglesias, R., & Barro, S. (2004). Using prior knowledge to improve reinforcement learning in mobile robotics. In Proceedings of the Towards Autonomous Robotics Systems. University of Essex, UK.
24. Nastic, S., Morichetta, A., Pusztai, T., Dustdar, S., Ding, X., Vij, D., & Xiong, Y. (2020). SLOC: Service level objectives for next generation cloud computing. *IEEE Internet Computing, 24*(3), 39–50.
25. Raman, K., Swaminathan, A., Gehrke, J., & Joachims, T. (2013). Beyond myopic inference in big data pipelines. In Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 86–94).
26. Sharma, P., Chaufournier, L., Shenoy, P., & Tay, Y. (2016). Containers and virtual machines at scale: A comparative study. In Proceedings of the 17th international Middleware conference (pp. 1–13).
27. Syu, Y., Ma, S.-P., Kuo, J.-Y., & FanJiang, Y.-Y. (2012). A survey on automated service composition methods and related techniques. In Proceedings of the 2012 IEEE ninth international conference on services computing, IEEE (pp. 290–297).
28. Voorsluys, W., Broberg, J., Venugopal, S., & Buyya, R. (2009). Cost of virtual machine live migration in clouds: A performance evaluation. In IEEE international conference on cloud computing, Springer (pp. 254–265).
29. Xu, Y., Sui, K., Yao, R., Zhang, H., Lin, Q., Dang, Y., Li, P., Jiang, K., Zhang, W., Lou, J.-G., et al. (2018). Improving service availability of cloud systems by predicting disk error. *2018* {USENIX} *Annual Technical Conference* ({USENIX} {ATC}, *18*), 481–494.
30. Yu, T., Zhang, Y., & Lin, K.-J. (2007). Efficient algorithms for web services selection with end-to-end QOS constraints. *ACM Transactions on the Web (TWEB), 1*(1), 6–es.
31. Zhang, D., Han, S., Dang, Y., Lou, J.-G., Zhang, H., & Xie, T. (2013). Software analytics in practice. *IEEE Software, 30*(5), 30–37.
32. Zhu, H., & Bayley, I. (2018). If docker is the answer, what is the question? In *Proceedings of the 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, IEEE (pp. 152–163).
33. Zillner, S., Curry, E., Metzger, A., Auer, S., & Seidl, R. (2017). European big data value strategic research & innovation agenda. In Big Data Value Association.
34. Zillner, S., Bisset, D., Milano, M., Curry, E., Södergård, C., Tuikka, T., et al. (2020). Strategic research, innovation and deployment agenda: AI, data and robotics partnership.