A Service-based Framework for Building and Executing Epidemic Simulation Applications in the Cloud

Nikos Parlavantzas, Linh Manh Pham, Christine Morin, Sandie Arnoux, Gael Beaunée, Luyuan Qi, Philippe Gontier, Pauline Ezanno

HAL Id: hal-02312313
https://hal.science/hal-02312313

RESEARCH ARTICLE

A Service-based Framework for Building and Executing Epidemic Simulation Applications in the Cloud

Nikos Parlavantzas*[1] | Linh Manh Pham*[1,3] | Christine Morin[1] | Sandie Arnoux[2] | Gaël Beaunée[2] | Luyuan Qi[2] | Philippe Gontier[2] | Pauline Ezanno[2]

[1]Univ Rennes, Inria, CNRS, IRISA, France

[2]BIOEPAR, INRA, Oniris, Université Bretagne Loire, France

[3]University of Engineering and Technology, Vietnam National University, Hanoi, Vietnam

Correspondence

Nikos Parlavantzas, IRISA, Campus universitaire de Beaulieu, 35042 Rennes CEDEX, France
*Email: nikos.parlavantzas@irisa.fr

Linh Manh Pham, University of Engineering and Technology, Vietnam National University, Hanoi, Vietnam
*Email: linhmp@vnu.edu.vn

Abstract

The cloud has emerged as an attractive platform for resource-intensive scientific applications, such as epidemic simulators. However, building and executing such applications in the cloud presents multiple challenges, including exploiting elasticity, handling failures, and simplifying multi-cloud deployment. To address these challenges, this paper proposes a novel, service-based framework called DiFFuSE that enables simulation applications with a bag-of-tasks structure to fully exploit cloud platforms. The paper describes how the framework is applied to restructure two legacy applications, simulating the spread of bovine viral diarrhea virus and Mycobacterium avium subspecies paratuberculosis, into elastic, cloud-native applications. Experimental results show that the framework enhances application performance and allows exploring different cost-performance trade-offs while supporting automatic failure handling and elastic resource acquisition from multiple clouds.

KEYWORDS:
Cloud computing; High Performance Computing; Simulation models; Epidemic simulation

1 | INTRODUCTION

As global transport networks continue to expand, the risk of infectious disease transmission grows, making research in pathogen spread on humans and animals increasingly important. An essential tool for studying pathogen spread is computer simulation based on mechanistic modelling. However, running epidemic simulations at large scale is time consuming for multiple reasons. First, the simulation models can capture fine-grained dynamics, considering different population scales (individual, population, meta-population) and accounting for multiple layouts (e.g., combining population and infection dynamics with farm management decisions) .[1] Second, some processes can be data-driven and exploit large amounts of real-life observed data, mostly concerning the population dynamics of hosts. Third, understanding the underlying biological system requires performing and comparing many simulation scenarios, e.g., through a model sensitivity analysis. Finally, producing reliable results demands large numbers of simulation runs when models are stochastic.

Speeding up the execution of large-scale simulations requires parallelism. Parallelism can be implemented at the single-computer level using multiple cores and processors as well as the distributed level using multiple computers interconnected through a network. There are currently three main options for obtaining the necessary computing resources for running simulations in parallel. The first option is using dedicated, high-end multicore systems or clusters. The limitation is that such infrastructures are typically shared among many local users, resulting in long waiting times. At the same time, extending such infrastructures to increase user satisfaction is time consuming and costly. The second option is grid computing, which allows sharing computers from multiple clusters in multiple sites. The main limitation of grids is the heterogeneity of the software stacks in each cluster, which makes application deployment difficult. Moreover, grid-based applications are assigned a statically-fixed number of resources, leading to poor resource utilisation or reduced performance in the face of dynamic variations in resource demand.

The third option for obtaining computing resources is cloud computing, which enables users to rapidly obtain and release practically unlimited amounts of resources while being charged only for actual resource usage. Thanks to virtualisation, cloud computing enables applications to have their own software stacks and to run reliably in various environments. Cloud computing thus overcomes the limitations of the previous two options, bringing immediate economic and agility benefits, and lowering the barrier to running complex simulations. Despite the attractiveness of cloud computing, converting epidemic simulation applications to become cloud-native applications presents several challenges.

A main challenge is facilitating the deployment and operation of simulation applications. Indeed, using clouds requires considerable manual effort for obtaining cloud resources, configuring and launching multiple runs of the simulation on these resources, making input data available, and collecting and processing simulation results. This effort becomes more complex when the application is deployed on multiple clouds, which is useful for avoiding dependence on a single vendor, enhancing availability, and taking advantage of lower resource prices or specialised resource capabilities. Another challenge is exploiting cloud elasticity to optimise the performance and cost of simulation applications. Indeed, simulation applications may require different amounts of resources at different times. For instance, the initialisation phase may require fewer compute resources than the main processing phase, thus making it more efficient to scale up the resource allocation at run time. A third challenge is handling failures, which are unavoidable when executing long-running, distributed applications, such as epidemic simulators, on the cloud. Handling failures is essential for avoiding data loss and ensuring the correct and efficient completion of the simulation.

To address these challenges, this paper presents a framework, called DiFFuSE, that enables simulation applications to fully exploit cloud platforms. The framework assumes that the core of the application's computation takes the form of a bag of independent tasks, in which tasks are simulation runs and can be executed in parallel. Specifically, the paper makes two main contributions. First, it describes the DiFFuSE framework for building and executing pathogen spread simulators. The main novelty of DiFFuSE lies in its service-based architecture; simulation applications are decomposed into services that can be deployed across multiple clouds and independently scaled, while the framework transparently handles service failures. Second, the paper presents the application of DiFFuSE to building cloud-native simulators of the spread of bovine viral diarrhea virus (BVDV) and of Mycobacterium avium subsp. paratuberculosis (MAP) and uses these applications to evaluate the benefits of the DiFFuSE framework. The paper completes and extends our previous work on the DiFFuSE framework.[2,3] Specifically, the paper provides further details on the operation of the framework, including the allocation of simulation runs and failure handling. Furthermore, the paper adds the case study of the MAP application and reports new experiments to provide a thorough evaluation of the framework using both case studies.

The rest of this paper is organised as follows. Section 2 discusses related work. Section 3 briefly presents the BVDV and MAP case studies, the simulations used throughout the paper as practical examples. Section 4 details the architecture of DiFFuSE and its application in the case studies. Section 5 reports a set of experiments using the case studies, and Section 6 concludes the paper.

2 | RELATED WORK

This section examines related research that focuses on exploiting large amounts of computing power to execute epidemic simulations.

Eriksson et al.[1] use OpenMP to parallelise pandemic simulations, taking advantage of multi-core processors. The evaluation demonstrates that performance can be improved by dynamically switching between single- and multi-core modes as the computational load varies. This work does not consider parallelisation at the distributed level. Bhatele et al.[4] introduce EpiSimdemics, a highly scalable, parallel code written in Charm++ using agent-based modelling to simulate disease outbreak over large, realistic, co-evolving interaction networks. The paper presents an implementation of EpiSimdemics simulating the spread of influenza on different supercomputers. The authors argue that EpiSimdemics obtains five times greater speedup than the second fastest parallel code in the field. Similarly to the work by Eriksson et al.[1], the authors do not consider parallelisation at the distributed level.

Many solutions exploit parallelisation at the distributed level in order to enhance performance. Martin and Carretero[5] present an MPI-based epidemic simulator, EpiGraph, used to study the propagation of infectious diseases between regions due to people movements. EpiGraph can be executed efficiently both on clusters and shared-memory nodes. Perumalla and Seal[6] present an optimistic, parallel, discrete event execution of a reaction-diffusion simulation of epidemic outbreaks. The simulation scales to 65,536 cores of a large Cray XT5 system with speedup of over 10,000. Rao and Chernyakhovsky[7] present an eco-modeling, bio-simulation environment, called SEARUMS++, for studying the global epidemiology of avian influenza. The environment uses Time Warp synchronised parallel simulations executed on clusters. Bisset et al.[8] present a modelling environment, called Indemics, for supporting complex epidemic simulations, not limited to a specific disease model. Although all previous solutions are scalable, they do not exploit the advantages of the cloud in reducing the cost and in increasing the agility of obtaining resources.

Zou et al.[9] propose using GPU clusters to implement large-scale, contact-network based epidemic simulations. The paper describes optimisation techniques for improving the efficiency of memory accesses and reducing the communication latency between compute nodes. The techniques are tested on a cluster whose compute nodes are equipped with GPUs, and experimental results show that the execution on GPUs can achieve 7.4 to 11.7 times speedup over the execution on CPUs. Holvenstot et al.[10] use general-purpose GPU devices to accelerate an agent-based simulation

of influenza spread. Experimental results show that the GPU implementation achieves significantly greater speedups than a multi-threaded CPU implementation. Although the previous solutions demonstrate the performance benefits of using GPUs, they require purchasing expensive, high-end hardware rather than using clouds, which currently provide GPU-based resources with low prices.

Eriksson et al.[11] present a cloud-based approach for performing simulations of pandemic influenza. The approach relies on the HTCondor architecture in which a central master schedules and distributes jobs and a set of workers receive and execute the jobs. The approach allows adding workers by renting Amazon EC2 virtual machines (VMs). The tests show that the network capacity of the master node can become a bottleneck when the number of workers increases. The authors discuss possible solutions, such as compressing data or choosing nodes connected to high-speed networks. Unlike our work, this work does not exploit elasticity to dynamically handle increases in resource demand.

Sukcharoen et al.[12] propose a cloud-based framework for simulating the spread of epidemic diseases. The framework applies loop decomposition to parallelise the simulation on multiple VMs of a private cloud, built using the Xen Cloud Platform. This framework is restricted to a specific epidemic model, namely, a modified version of the SEIR model (Susceptible, Exposed, Infectious, Recovered) and does not support elasticity. Price et al.[13] present an approach for executing a compute-intensive application for epidemic analysis on cloud resources. The approach uses the Nimbus cloud to obtain resources on demand. This work considers neither elasticity nor the cost of using cloud resources. Haris and Manzoor[14] present a cloud-based framework for simulating the spread of the dengue virus in Pakistan. This framework does not support parallelisation at single-computer level and does not exploit the elasticity of cloud resources.

Work focusing on executing bag-of-tasks applications on the cloud is also related to ours. Benedictis et al.[15] present a framework, called BOT, for developing bag-of-tasks scientific applications on the mOSAIC cloud platform. Similarly to our work, BOT can be used on the resources of any cloud providers, and it includes strategies for fault-tolerance. BOT was evaluated using an application that solves a classical combinatorial optimisation problem (binary knapsack). Agarwal and Prasad[16] present a similar framework for developing bag-of-tasks applications, called AzureBOT. However, this framework targets exclusively a single cloud, that is, the Azure cloud. AzureBOT was evaluated using two applications, an Internet data scraper and a master-slave simulator. Krol et al.[17] discuss a data farming platform, called Scalarm, which relies on the master-worker pattern, and presents its extension for running a molecular dynamics simulation on clusters and on clouds. The evaluations of BOT, AzureBOT, and Scalarm focus on application performance and cost without considering fault-tolerance and elasticity. In addition, unlike our work, frameworks for bag-of-tasks or master-worker applications lack specialised facilities for building simulation applications, making it complex and inefficient to use them for real-world applications. Lacking facilities include managing simulation experiments, coordinating simulation runs, replicating simulation data for resilience and performance, and elastically scaling the computation and data transfer capacities involved in running the simulation.

Much research has focused on algorithms for scheduling bag-of-tasks applications on cloud resources, aiming to optimise application performance or monetary cost.[18] Similarly, such research does not provide a complete solution for building epidemic simulation applications. Nevertheless, such research can form a basis of optimising task scheduling in our work, which currently only applies basic scheduling policies.

To summarise, some of the related work for executing epidemic simulations does not exploit clouds and their associated parallelisation, cost, and agility benefits.[1,4,5,6,7,8,9,10] Other related work exploits clouds but lacks support for elasticity,[11,12,13,14] multi-cloud deployment,[16] fault tolerance,[17] or required facilities beyond bag-of-tasks execution.[15,18] To the best of our knowledge, our work is the first to propose a complete, practical solution for building and executing simulation applications, structured as bags of tasks, across multiple clouds with support for elasticity and fault-tolerance.

3 | CASE STUDIES: BVDV AND MAP SPREAD SIMULATORS

This section discusses the use of simulation for studying two diseases, bovine viral diarrhea and bovine paratuberculosis. These are endemic diseases in cattle, caused respectively by a virus (Bovine Viral Diarrhoea Virus, BVDV) and a bacteria (Mycobacterium avium subsp. paratuberculosis, MAP), that lead to economic losses owing to reproductive disorders, decrease in milk production as well as increases in morbidity and mortality among infected animals.[19,20] To identify the main drivers of the spread of the diseases in a region and to compare a large range of regional control strategies, two distinct original stochastic metapopulation models have been proposed. They are both discrete-time compartment models based on characteristics of dairy cattle herds in Brittany.

The model of BVDV spread accounts for the two main transmission pathways between herds: trade movements of animals carrying the virus and virus transmission during the grazing season due to over-the-fence contacts with animals from neighbouring herds.[21] In addition, the model accounts for the local within-herd virus spread, using as a building block a previous within-herd model.[19,22] Practically, the simulation uses a 2 GB database describing cattle life trajectories in Brittany from 2005 to 2013. This database provides information on the demography of dairy animals (e.g., birth, death, culling rates) and trade movements between farms (e.g., holding farms, dates and reasons of entry/exit). The metapopulation model represents the spread of BVDV within and between 12,750 dairy cattle herds located in Brittany and takes into account 1,056,565 animals movements over the 9-year period.
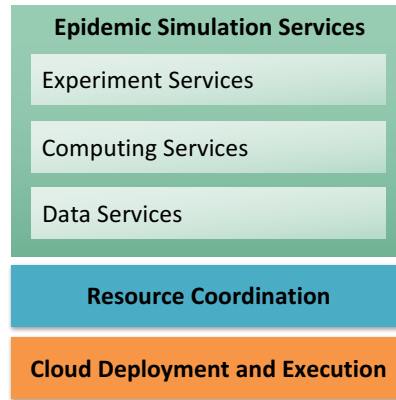
FIGURE 1 Architecture of DiFFuSE

The model of MAP spread [23,24] accounts only for between-herds spread through animal trade movements, which represents the main pathway. The regional model consists in coupling numerous (one per farm) stochastic within-farm epidemiological models [25] through cattle trade movements. Practically, for both animal movements and farming management, pre-processed real observed data (extracted from the same database used for the BVDV model) are plugged into the model. The metapopulation model represents the spread of MAP within and between 12,857 dairy cattle herds located in Brittany, and takes into account 919,304 animals movements over the 9-year period.

These complex simulations pose the challenges of performance and operational costs, already discussed in Section 1 for generic epidemic simulations. Indeed, ensuring accurate predictions requires performing from 200 to 500 repetitions, which is time-consuming and highly susceptible to failures from the human or machine side. Moreover, researchers have to perform simulations for many possible spreading scenarios and carry out sensitivity analyses for many model parameters. As a result, performing such simulations remains beyond the budget of small and medium laboratories and research groups. The BVDV and MAP case studies will be used in demonstrating and evaluating the DiFFuSE framework.

4 | DiFFuSE FRAMEWORK AND APPLICATIONS

Fig. 1 shows the architecture of the DiFFuSE framework, which is structured in three layers. Seen from the bottom up, the layers are increasingly specific to the domain of epidemic simulations. At the bottom is a cloud deployment and execution platform, capable of running any modular application across clouds. Next is the resource coordination layer that supports building services that exchange data using a specific set of provided mechanisms. At the top is the epidemic simulation services layer that supports building epidemic simulation applications as sets of cooperating services. The rest of this section provides details on each layer and on applying the framework in the case studies.

4.1 | Cloud Deployment and Execution

The aim of this layer is to hide the intricacies of cloud technologies (e.g., diverse cloud providers, interfaces, services). Specifically, the layer facilitates deploying an application across multiple clouds and automatically adapting this deployment, when needed (e.g., elastically scaling components to handle workload variations). The layer assumes that applications are structured as interconnected, deployable components and relies on the PaaSage platform [26,27], a holistic solution for automatic deployment and execution of cloud applications.

Briefly, the PaaSage platform operates as follows. First, PaaSage receives as input descriptions of cloud providers (e.g., VM types, prices) and applications (e.g., components and interconnections, objectives), all written in CAMEL, the PaaSage application modelling and execution language. PaaSage then decides how to best deploy the application on available cloud resources and performs the deployment through obtaining VMs, installing and configuring software. At runtime, PaaSage monitors and adapts the deployment to react to events, such as violations of performance objectives, or changes in cloud prices. It can notably trigger elasticity actions (e.g., scale out a component) based on ECA (Event Condition Action) rules defined in CAMEL. Within DiFFuSE, PaaSage receives as input CAMEL descriptions of the simulation services and of available cloud providers and generates and dynamically maintains an appropriate multi-cloud deployment.

4.2 | Resource Coordination

The aim of this layer is to facilitate the coordination of distributed services, communicating over the network. Specifically, the layer enables services to exchange data in a reliable and scalable way. Each service is running as a separate process and can be deployed using the cloud deployment and execution layer discussed previously. Next, we discuss how the resource coordination layer supports data exchange and failure handling.

4.2.1 | Data Exchange

The layer relies on the concepts of resource, resource manager, resource user, resource provider, and resource monitor. A `resource` is any data that can be named and exchanged between services (e.g., experiment configuration, experiment data). A `resource manager` owns and provides access to local resources and is identified by a URL. Resource managers can be linked in a hierarchy in which a resource manager can provide access to the resources of its parent resource manager.

A service can play three roles with regards to a resource: (1) `resource user`, which sends requests to a resource, demanding the associated data, (2) `resource provider`, which receives and replies to such requests, providing access to the resource data, and (3) `resource monitor`, which tracks the state of a resource (e.g., providers are present, providers have failed). A resource supports one of four management styles determining how data are distributed among providers:

- `Single provider management`: the resource has only a single provider offering all the data.

- `Replicated management`: the resource can have multiple providers, each offering the same data. A request to the resource is delivered to any of the providers. This management style can be used to support fault-tolerance; if one provider fails, the requests are handled by the remaining providers. Moreover, it can be used to improve the request response time through balancing the load among providers. A service can obtain the data from an available provider and become itself a provider.

- `Individual management`: the resource can have multiple, independent providers, each offering different data. A resource user can send requests to an identified provider to obtain the data.

- `Collective management`: the resource has a fixed set of providers, each offering a different part of the data. A resource user can obtain the data if all providers are available. If a provider fails, a notification is sent that allows reacting to this failure.

Resource coordination in DiFFuSE is implemented using three C++ classes: the `resourceManager` class owns local resources and interacts with its parent resource manager, the `resourceLocalClient` supports interacting with local resources, and the `resourceClient` class supports interacting with remote resources.

4.2.2 | Failure Handling

Services are distributed and interdependent, which introduces many opportunities for failure:

- Deployment-order failures: services may fail when they try to interact with services that are not yet deployed.

- Cleanup failures: services may fail to release allocated resources when the overall computation ends.

- Service failures: services may fail due to hardware errors or bugs.

DiFFuSE handles deployment-order failures through using the `nanomsg` message library.[28] With classical TCP sockets, a connection to a service fails if the service is not already running. With `nanomsg`, the library transparently uses retry and timeout mechanisms, removing any dependencies on deployment order. Cleanup failures are avoided by ensuring that when a specified service service stops, all other services stop and cleanly release their resources. DiFFuSE handles service failures through the connections between each resource manager (except the root manager) and its parent resource manager. The services on both sides of such connections check that they periodically receive messages from the other side and take appropriate actions when failures are detected. For instance, if the resource manager detects the failure of a remote service that was a resource provider, the manager updates the resource state and notifies all resource users and resource monitors.

4.3 | Epidemic Simulation Services

The aim of this layer is to facilitate building epidemic simulation applications as sets of cooperating services. Specifically, the layer provides reusable design and code for commonly-required functionality in such applications, such as controlling experiments, executing simulation runs, and managing

the flow of simulation data between services. This functionality is provided as a collection of services, built on the resource coordination layer, that can be easily customised and used by developers. Next, we discuss the types of provided services and how they collaborate to execute the simulation, handle associated failures, and transfer data.

4.3.1 | Provided Services

The simulation services fall into three main categories.

- `Data Services`: retrieve data from databases and provide these data as well as related meta-data to other services.

- `Computing Services`: perform the main computations of the simulation, either sequentially or in parallel, using single-computer parallelisation (e.g., using OpenMP). Computing services include services that execute simulation runs (called `worker` services), a service that generates final results (called `final results` service) as well as services that process these results.

- `Experiment Services`: control the life-cycle of experiments. They track the progress of the computations, they handle service failures, and they provide configuration data about the experiments to other services.

DiFFuSE provides three generic C++ classes that serve as templates for developing services of the previously-mentioned categories. Each service defines a routine executed when requests are received and instantiates a resource manager class that provides access to local and remote resources.

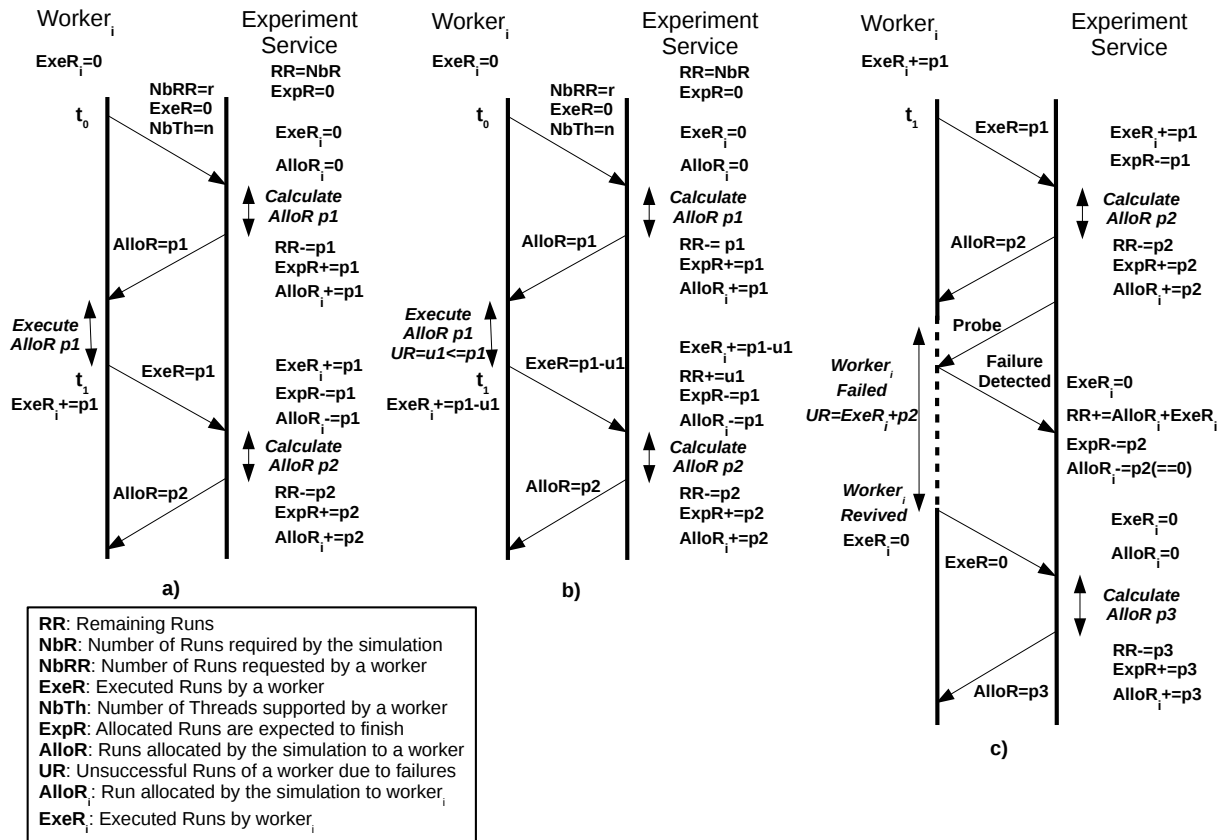### 4.3.2 | Allocation of Simulation Runs

RR: Remaining Runs
NbR: Number of Runs required by the simulation
NbRR: Number of Runs requested by a worker
ExeR: Executed Runs by a worker
NbTh: Number of Threads supported by a worker
ExpR: Allocated Runs are expected to finish
AlloR: Runs allocated by the simulation to a worker
UR: Unsuccessful Runs of a worker due to failures
AlloR$_i$: Run allocated by the simulation to worker$_i$
ExeR$_i$: Executed Runs by worker$_i$

FIGURE 2 Protocol for Allocating Simulation Runs in Different Situations: a) Normal, b) Lost Runs and c) Failed Workers

This section discusses how the experiment service allocates simulation runs to worker services. The allocation assumes that simulation runs are independent, and thus that the overall computation has the structure of a bag of tasks. Moreover, the allocation follows a simple policy that

requires only information on the number of threads supported by the node hosting a worker. Adding more complex policies that exploit richer information on the underlying nodes (e.g., memory size, network bandwidth) and on simulation runs (e.g., expected execution time) could improve application performance. However, the current paper does not focus on optimising performance but on providing a complete solution for running simulations across clouds in a flexible, scalable, and fault-tolerant manner.

The simple policy is shown in Figure 2a. At the beginning (t_0), a $worker_i$ sends a request message to the experiment service to ask for the number of runs to execute. The request contains a suggested number of runs (NbRR), the number of threads (NbTh) supported by the node hosting the worker, and the number of runs executed successfully by the worker after the previous allocation (ExeR). Typically, the NbRR parameter is set to the NbTh value in order to enable parallel execution of the runs; if NbRR is larger than NbTh, some runs will be queued at the worker. The ExeR parameter is used to calculate the cumulative number of executed runs for this worker ($ExeR_i$) at the experiment-service side.

The experiment service maintains the list of executed runs of all participating workers as well as cumulative variables, such as remaining runs (RR) and expected runs (ExpR). RR is the total number of remaining runs to be allocated to workers. ExpR is the number of runs that were allocated, but the experiment service is still waiting for their results. Initially, ExpR is set to zero while RR is set to the number of runs required by the simulation (NbR).

When the experiment service gets the request from a worker, it calculates the number of runs allocated to the worker (AlloR) using Algorithm 4.3.2. The algorithm takes into account the number of runs suggested by the worker (NbRR), the total number of workers (nbOfWorkers) and the remaining number of runs (RR). The algorithm allocates a larger number of runs at the beginning, which decreases as the remaining number of runs decreases. The goal of this strategy is to have all workers finish their computations at approximately the same time. The experiment service then sends the value of AlloR in a reply message to the worker and updates the cumulative variables including the $AlloR_i$ variable, storing the number of runs allocated to that specific worker.

When the worker gets the reply message from the experiment service, it executes the runs allocated to it, and then sends another request message to the execution service. This process repeats until all the simulation runs are allocated. The next subsection includes a discussion of cases when workers fail to execute the allocated runs.

Algorithm 4.3: Pseudocode of Allocation Algorithm

```
int getAllocation (message request) {
  int propose;
  propose = ((RR + nbOfWorkers − 1) / nbOfWorkers)
           * request.NbTh;
  if (propose > request.NbRR) propose = request.NbRR;
  if (propose >= RR) propose = RR;
  else if (propose == 0) propose = RR;
  return propose;
}
AlloR = getAllocation (request);
```

4.3.3 | Handling Worker Failures

This section discusses failures of workers to execute their allocated runs; generic failure handling mechanisms were discussed in Section 4.2.2. The workers are resource providers that provide the partial results resource. The management style of this resource is collective (see 4.2.1), which means that workers offer different parts of results data and any failure to provide the data must be handled. There are two situations when the collective results cannot be provided. The first is when some of the simulation runs allocated to a worker fail to be successfully executed (e.g., due to insufficient memory). In this situation (see Figure 2b), the worker subtracts the amount of unsuccessful runs (UR) from the number of allocated runs and sends the result to the experiment service (t_1), which updates the corresponding variables. Specifically, the amount of the unsuccessful runs is added to RR, and the amount of successful ones is added to $ExeR_i$. The unsuccessful runs will then be reallocated to other workers or again to $worker_i$. The second situation is when the worker service fails completely and thus all its executed runs from the beginning are lost. When the execution service detects this failure (see Figure 2c), it updates the corresponding variables. Specifically, the total number of lost runs of $worker_i$ is added to RR, and $ExeR_i$ is set back to zero. The lost runs will then be reallocated to other workers. If $worker_i$ is later revived, all its variables are set to zero. The worker will still be recognised by the experiment service thanks to a serial identification number. The worker will send requests to obtain simulation runs, and the process continues as before. Similarly with the allocation policy of Section 4.3.2, more optimised failure-handling policies could be added (e.g., using replication), but this is beyond the scope of the current paper.

4.3.4 | Data Management

In general, services exchange data through resources, which also serve as the synchronisation mechanism. For instance, in the case of collective management, a service acting as a resource user must wait for all providers to be available before obtaining the data. To transfer or process data, services do not need to use disk storage, increasing application performance. In addition, services do not need to update data, avoiding consistency issues.

The typical data flow in an epidemic simulation application is as follows. A data service provides access to input data (e.g., animal demography) contained in a local file or database. If further services are configured as providers of the same resource (i.e., following the replicated management style), they contact the initial service and obtain the complete data in a serialised form over the network. When the data are available at all providers, the experiment service is notified, and the workers start receiving simulation runs, following the previously-described allocation policy. Next, the workers obtain data from the data services, as needed, and execute the allocated simulation runs. A special case is when workers obtain data from other workers acting as resource providers; this configuration can reduce data transfer times in multi-cloud environments, as demonstrated in Section 5.3.2. In executing the simulation runs, the workers produce partial results. The workers store these results in memory, acting as providers of the partial results resource (following the collective management style). If there are lost runs, these runs are re-executed, as seen in Section 4.3.3, and the corresponding partial results are reproduced. When all runs are successfully executed, all providers of the partial results resource are available. This enables the final results service to gather the partial results and produce the final results. These data can then be obtained by other services for further analysis or visualisation.

4.4 | The BVDV and MAP spread simulators

The legacy BVDV and MAP spread simulators were transformed to cloud-based applications based on the DiFFuSE framework. Specifically, the original BVDV code (around 4600 lines of C++ code) was sequential and was first modified to add single-computer parallelism using OpenMP. The original MAP code (around 14000 lines of C++ code) already supported single-computer parallelism using OpenMP. In both cases, the services to be used in the target version were then identified and implemented by extending the generic C++ classes and integrating the existing code. Most of the effort was spent on implementing the data services, which required decomposing the original data model into separate pieces of data (e.g., herds, movements) managed by independent services. Implementing computation services required minor modifications for adapting the original simulation computation code and inserting it into the generic class. The experiment services were largely based on the generic class.

In total, 6.3% of the original BVDV code and 1.78% of the original MAP code were modified in order to convert them to cloud-based applications. In the final DiFFuSE-based BVDV application, the proportion of the code that is specific to the BVDV spread simulation (rather than reusable code) is around 17.7%. In the final MAP application, this proportion is around 5.93%. These measures provide evidence of the limited effort required for applying DiFFuSE and of the added value of the DiFFuSE reusable code.

The services used in the DiFFuSE-based applications are shown in Fig. 3. The data services (`Herd`, `Move` and `Neighbour`) provide descriptions of herds, movements, and herd neighbours applying replication to improve failure tolerance and responsiveness. Each `Worker` retrieves descriptions from the data services and configuration data from `Experiment` and interacts with `Experiment` to compute partial simulation results. `Final Result` gets the partial results from workers and computes the final results, while `Plot Final Result` generates a graphical view. Finally, `Experiment` controls the life-cycle of an experiment.

The services are deployed through the cloud deployment and execution layer based on PaaSage. Figure 4 shows a possible deployment of the BVDV application in a multi-cloud environment.

5 | EVALUATION

We conducted a set of experiments to evaluate DiFFuSE in terms of performance, cost effectiveness, failure handling and elasticity in single or multi-cloud environments. The BVDV and MAP spread simulations are used throughout the experiments as the reference applications. We ran experiments with the BVDV application in three cloud environments: Amazon EC2, GWDG[29], and an OpenStack deployment on Grid5000.[30] Specifically, OpenStack was deployed on nodes of the Parasilo Grid5000 cluster at Inria in Rennes. We ran experiments with the MAP application in Google Compute Cloud and two OpenStack deployments on Grid5000, one on nodes of the Paranoia cluster in Rennes and one on the Econome cluster in Nantes. All experiments were repeated 5 times and we show the averages.
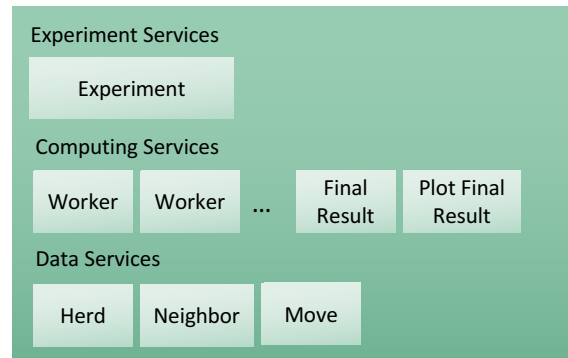
FIGURE 3 Services of the BVDV and MAP spread simulators
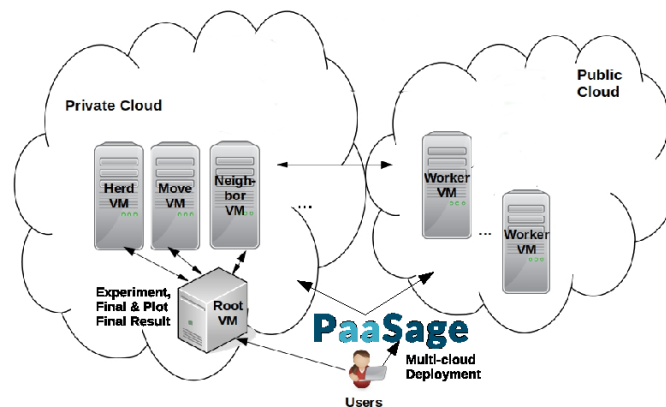

FIGURE 4 The BVDV application deployed in the cloud

5.1 | Performance and Cost

5.1.1 | Experiment 1

This experiment evaluates the execution time and cost of running the DiFFuSE-based BVDV simulation in a private cloud and in a public cloud. As a private cloud, we use our OpenStack-based cloud in Grid5000, where each worker service is packed into one VM with 13 cores and 100 GB memory. The cost is calculated as the total price of all used VMs multiplied by the running time. Although the VMs in our private cloud are free, we use the price of a similar VM type in the Amazon EC2 cloud as the reference price (in $/hour); specifically, we use the price of the r4.4xlarge VM type in the EU (Ireland) region.

Fig. 5 shows the time and cost to conduct 20, 40, 100, 200 and 500 runs in the private cloud. The execution time is almost linear in the number of runs. With 500 runs, the execution times are notably 268.36 and 127.76 minutes with 4 and 9 worker services. For comparison, according to experiments done at INRA, performing 500 runs with the sequential legacy code takes around 1980 minutes (33 hours) in a server of the BioEpAR [31] research unit equipped with 13 cores and 100 GB memory. This server has the same configuration with the used VMs, but note that the legacy code uses only 1 core. The figure also shows that the performance increases when the number of workers rises from 4 to 9, and the effect is greater with increasing number of runs. The cost slightly increases when the number of workers increases from 4 to 9.

As a public cloud, we use Amazon EC2. As each run of the simulation needs at least 35 GBs of memory, we consider two scenarios, each using a different type of memory-optimised EC2 VM: r4.xlarge (4 cores, 30.5 GB RAM + 16 GB swap) and r4.2xlarge (8 cores, 61 GB RAM). In a third scenario, we execute an OpenMP-based BVDV spread simulator, not integrated into DiFFuSE, on a single EC2 VM of r4.2xlarge type. The results are also compared with a fourth, base scenario, which executes the legacy, sequential BVDV code on the previously-mentioned BioEpAR server.
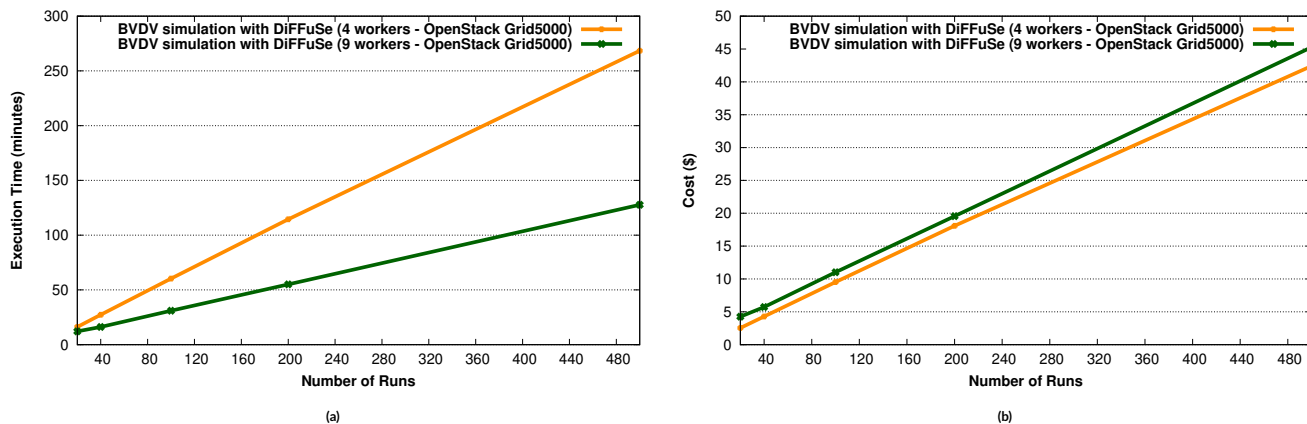
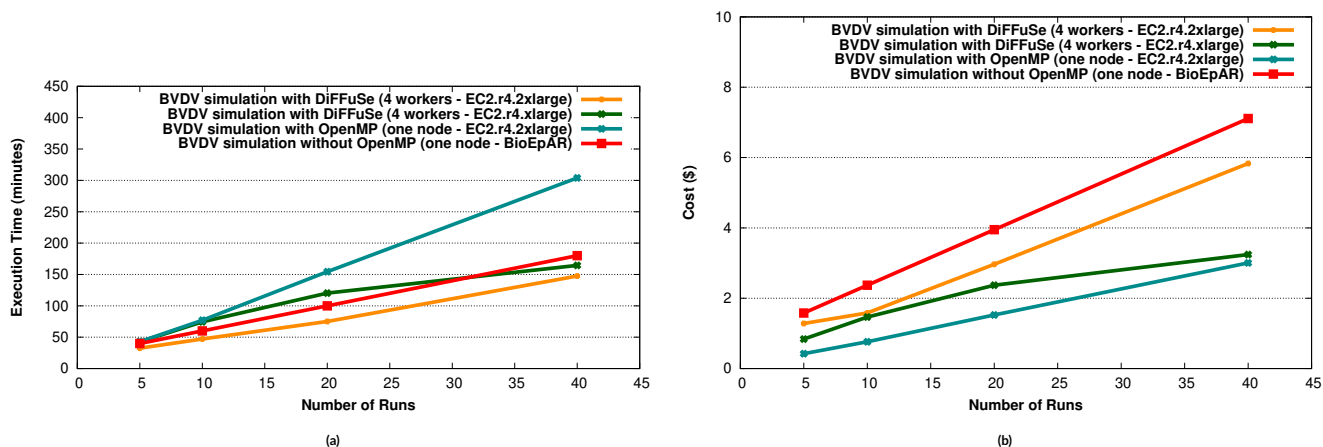**FIGURE 5** BVDV in Private Cloud: Number of Runs vs. a) Execution Time, b) Cost

FIGURE 6 BVDV in Public Cloud: Number of Runs vs. a) Execution Time, b) Cost

Fig. 6 shows the time and cost to conduct 5, 10, 20 and 40 runs with coloured lines representing the 4 scenarios. We see that the DiFFuSE-based simulator with 4 workers of the r4.2xlarge type shows the best performance. In particular, the simulator using r4.2xlarge outperforms the one using r4.xlarge. The reason is that r4.xlarge does not have enough RAM (< 35 GB), thus having to consume its swap space in both data loading and transferring phases. This leads to 60% slower time in these phases than in the case of r4.2xlarge, according to our measurements. The simulator using r4.xlarge, the lower configuration, performs better than the legacy code in the base scenario for more than 32 runs. The OpenMP-based simulator shows the worst performance owing to the high parallelisation overhead for the given number of runs. Regarding cost, the OpenMP-based, single-node scenario has the lowest cost for up to 40 runs. The r4.2xlarge scenario costs more than the r4.xlarge scenario, except with 10 runs, where their costs are almost equal. The base BioEpAR scenario has the highest cost.

This experiment validates that using DiFFuSE reduces execution time and cost compared to the legacy BVDV simulator solution, thus enabling scientists to perform simulations previously considered as impractical or prohibitively expensive. Moreover, the experiment demonstrates the flexibility in the deployment of DiFFuSE-based applications, which can use different number of workers on different types of VMs in private or public clouds.

5.1.2 | Experiment 2

This experiment evaluates the execution time and cost of running the DiFFuSE-based MAP simulation in various private and public cloud environments. As a public cloud, we use Google Compute Engine. As private clouds, we use two OpenStack deployments on Grid5000, one on nodes of the Paranoia cluster in Rennes and one on the Econome cluster in Nantes. In the private clouds, each worker service is packed into one VM with 20 cores and 100 GB memory. We use the price of the same VM type in Google Compute Engine as the reference price increased by 30% to

reflect the typically higher unit costs of private clouds. We also assume that the price of VMs in the Rennes cluster is 5% higher than that of VMs in Nantes. As a base scenario, we use the execution of the legacy MAP code, parallelised using OpenMP, on a BioEpAR server equipped with 20 cores and 64 GB memory.
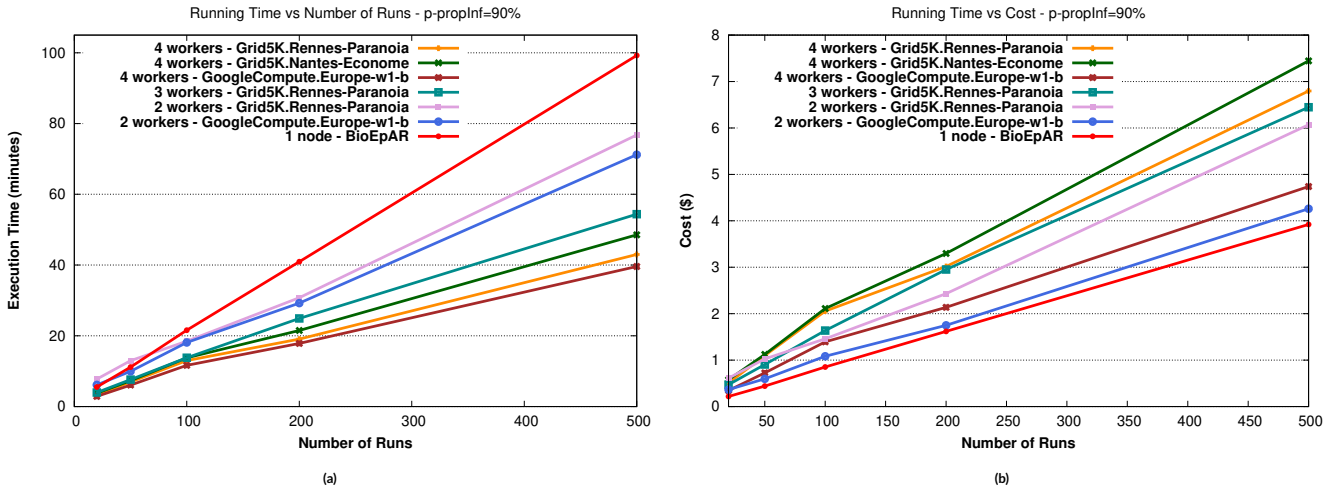

FIGURE 7 MAP in various clouds: Number of Runs vs. a) Execution Time, b) Cost

Fig. 7 shows the time and cost to conduct 20, 50, 100, 200 and 500 runs of the simulation with the proportion of infected herds (p-propInf) set to 90%. Similar results were obtained with this parameter set to lower values (60% and 30%), which reduces the simulation duration. The figure shows that the base BioEpAR scenario has lower performance than the other scenarios, except when conducting less than 80 runs with 2 workers in Grid5000 Rennes. It also shows that performance improves when the number of workers increases. The scenario with 4 workers in Google Compute Engine shows the best performance.

This experiment validates the performance and cost advantage of using DiFFuSE compared to the legacy MAP simulator solution. Moreover, it provides further evidence of the supported deployment flexibility, which allows users to make different cost-performance trade-offs by controlling the number and types of used cloud resources.

5.1.3 | Experiment 3

This experiment evaluates the impact of increasing the number of workers on the cost and execution time. Fig. 8 shows the execution time and cost of running the BVDV simulator with an increasing number of workers (from 1 to 4) for 5 and 20 runs. As expected, the execution time decreases and then levels off. The cost keeps increasing as the number of workers increases. Fig. 9 and 10 show the execution time and cost of running the MAP simulator with an increasing number of workers in a private and a public cloud. Again, the execution time decreases and then levels off. The cost typically increases, except when passing from 4 to 5 workers in Google Compute Engine. In this case, the effect of the reduced execution time on total cost is greater than that of the added VM.

This experiment validates that deploying DiFFuSE-based simulators in diverse cloud environments provides expected scalability properties, and thus that DiFFuSE can effectively exploit cloud resources to perform growing amounts of computation.

5.2 | Failure Handling

5.2.1 | Experiment 1

This experiment evaluates the failure handling capabilities of DiFFuSE. We use the DiFFuSE-based application for BVDV spread simulation with three workers packed in VMs of our OpenStack private cloud (see Section 5.1.1). In one scenario, one of the workers fails early and the others get the burden of redoing the runs until there are zero remaining runs. In the second scenario, the failed worker is revived and takes on further work, if the number of remaining runs is larger than zero. Before the recovery of the failed worker, the other workers recalculate the lost blocks and get other blocks from the experiment service if they are free. We set the NbRR parameter to 2, which means that a worker can receive a maximum of two simulation runs. The simulation is repeated 21 times.
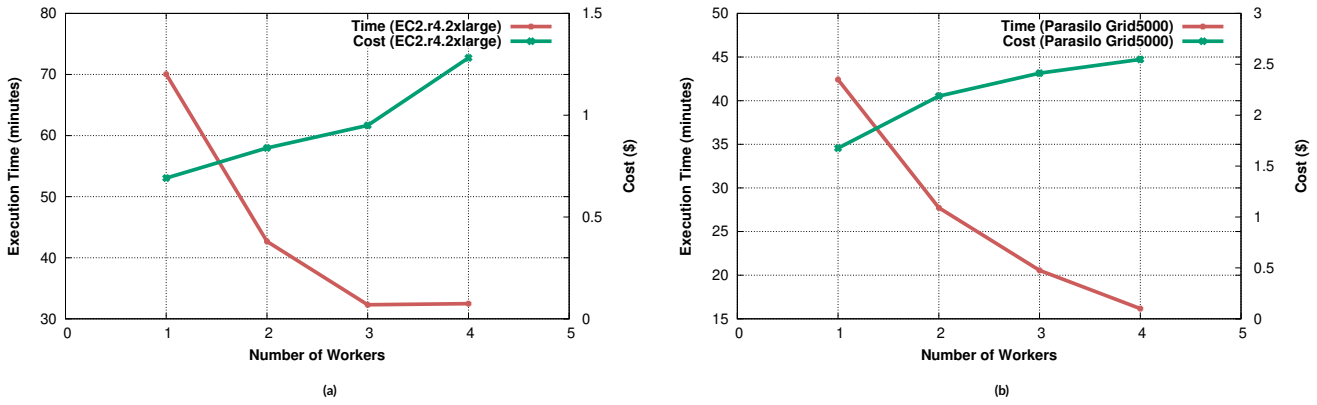
FIGURE 8 Execution Time and Cost to run the BVDV spread simulator: a) 5 runs b) 20 runs
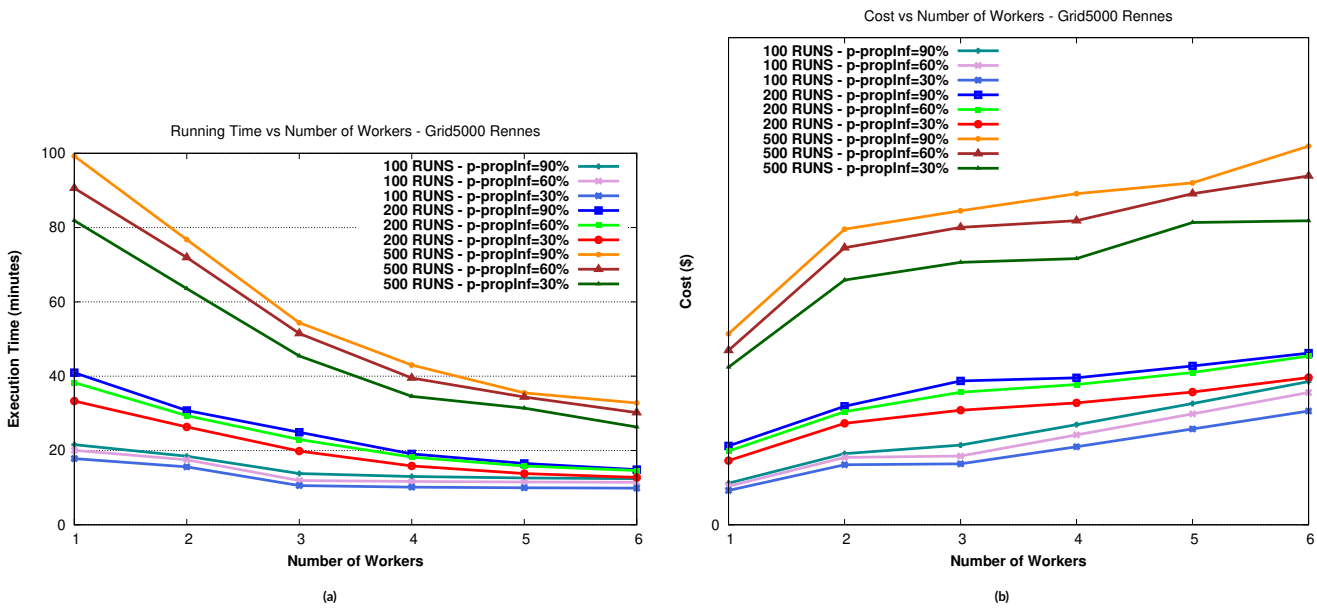

FIGURE 9 Execution Time (a) and Cost (b) to run the MAP spread simulation with different number of workers in OpenStack cloud

Fig. 11a and 12a demonstrate the number of runs distributed to the three workers over simulation time in the two scenarios. Worker 3 is the failed worker (around the 7th minute) which never recovers in scenario 1 and recovers in scenario 2 (around the 94th minute). The results in Fig. 11b and Fig. 12b show that number of remaining runs are distributed, executed and finished faster at the end of the experiment in scenario 2. This makes the distribution of runs about 23 minutes sooner than scenario 1, which reduces the total execution time. The experiment validates that DiFFuSE enables the simulator to dynamically adapt to service failures and recoveries.

5.2.2 | Experiment 2

This experiment evaluates the management of additional types of failures. Specifically, it considers also situations where workers fail to execute a certain number of simulation runs without crashing completely. These are situations described in Figure 2b. The previous experiment considered only situations described in Figure 2c. The setup of the experiment is similar to that of the previous experiment (Section 5.2.1) with 21 simulation runs distributed to 3 workers in VMs of our OpenStack private cloud. We observe how the runs are reallocated in four scenarios: when a worker fails and never revives (*WF*), when a worker loses one run (*L1*) or two runs (*L2*) and when a worker fails completely but is later revived (*RL*). Specifically, the failed worker is worker 3 and failure/loss and recovery follow the same timing as in the previous experiment. The results of this experiment are shown in Table 1. Specifically, the table presents the number of runs allocated to each worker for every scenario and for different values of NbRR
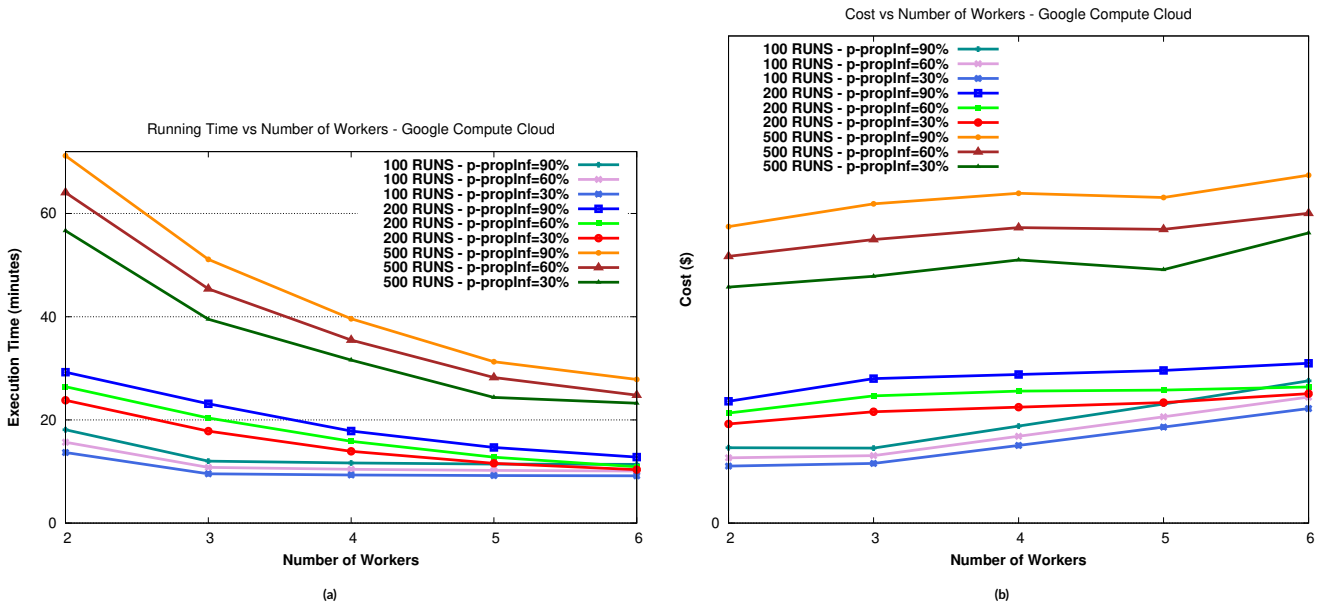
FIGURE 10 Execution Time (a) and Cost (b) to run the MAP spread simulation with different number of workers in Google Compute cloud
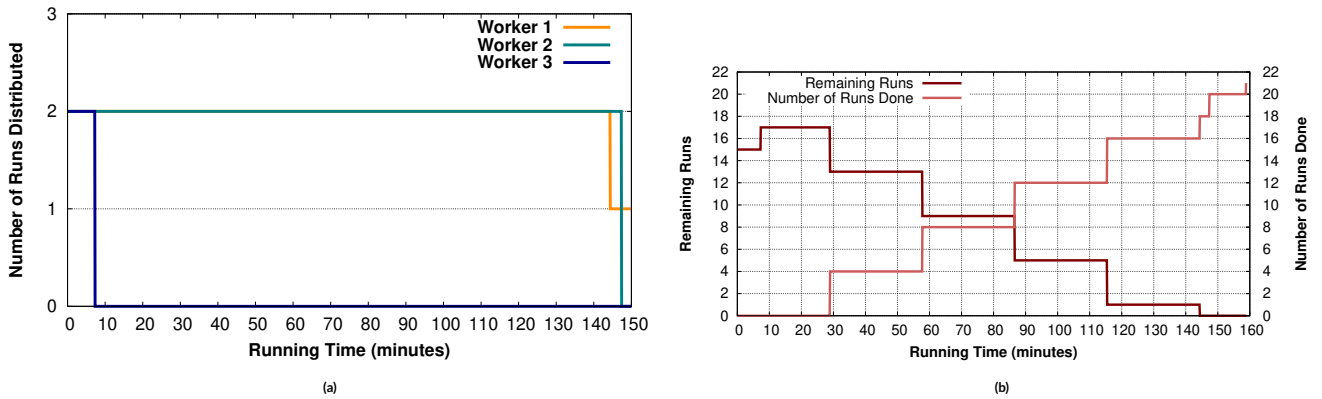

FIGURE 11 DiFFuSE with failure handling - Scenario 1

(2, 4 and 8). The table also shows the maximum number of runs allocated to a worker (*MAX*) which determines the execution time of the scenario (the lower the MAX, the better).

We observe that the value of MAX is best in scenarios L1 and L2 that achieve better distributions of runs. The value of MAX is not affected by the NbRR parameter. When the worker fails without revival (scenario WF), larger values of NbRR lead to worse values of MAX. The reason is that a greater number of runs has to be redistributed to surviving workers. Inversely, when the worker fails but recovers (scenario RL), larger values of NbRR lead to better values of MAX since the recovered worker obtains more runs to execute. The experiment validates that DiFFuSE can handle a wide range of failure scenarios.

5.3 | Multi-Clouds

5.3.1 | Experiment 1

This experiment evaluates the support provided by DiFFuSE for multi-cloud deployment. We deploy the services of the BVDV spread simulator in 2 VMs (r4.2xlarge and m4.4xlarge) of Amazon EC2 and 1 VM (m2.xlarge - 4 cores 32 GB RAM + 16 GB swap) of GWDG, which is a cloud-serving research community of Göttingen University and the Max Planck Society in Germany. The price of the GWDG m2.xlarge VM type is derived from the EC2 r4.xlarge VM type with an identical configuration. All three workers must collectively execute 20 runs. Table 2 shows statistics for this
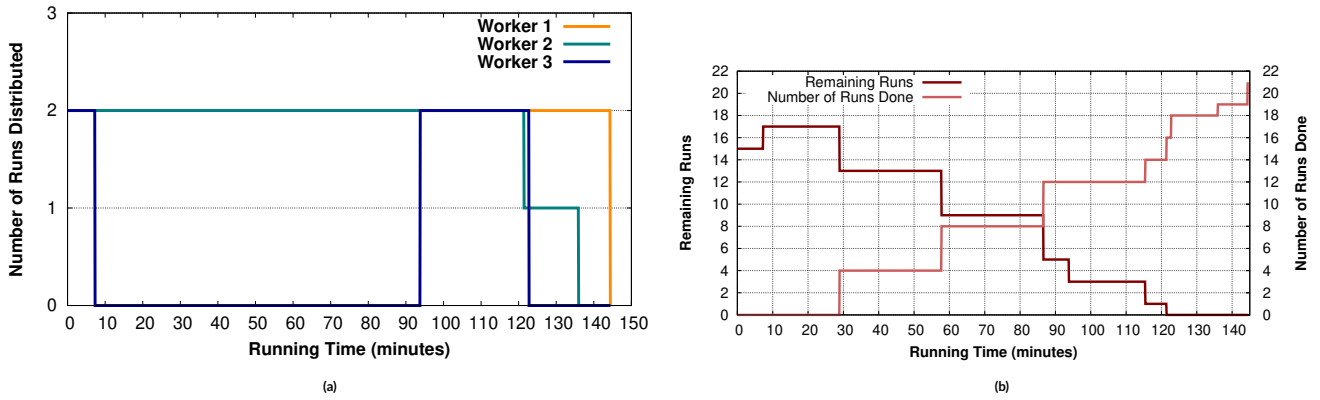
FIGURE 12 DiFFuSE with failure handling - Scenario 2

TABLE 1 DISTRIBUTION OF RUNS AMONG WORKERS

NbRR	2				4				8			
	WF	L1	L2	RL	WF	L1	L2	RL	WF	L1	L2	RL
Worker 1	11	8	8	10	12	8	8	9	13	8	8	8
Worker 2	10	6	7	9	9	6	7	8	8	8	8	8
Worker 3	0	7	6	2	0	7	6	4	0	5	5	5
MAX	11	8	8	10	12	8	8	9	13	8	8	8

TABLE 2 THE BVDV SPREAD SIMULATOR IN A HETEROGENEOUS MULTI-CLOUD ENVIRONMENT

BVDV spread simulator - Total runs 20							
VM Type	# Cores	# Runs	Execution Time	Total CPU Time	Total Cost	R_CPU	Cost per Run
ec2_m4_4xlarge	16	10	32.62(m)	14591.30(s)	0.651$	46.60%	0.065$
ec2_r4_2xlarge	8	8	36.10(m)	11115.20(s)	0.435$	64.15%	0.054$
gwdg_m2_xlarge	4	2	27.93(m)	2706.91(s)	0.217$	40.38%	0.108$

experiment. Total CPU time is the amount of time spent by all VM cores for simulation processing. R_CPU is the exploitation rate of the VM cores, calculated as follows:

$$R\_CPU = \frac{TotalCPUTime * 100}{ExecutionTime * NumberofCores} \tag{1}$$

From the table, we see that the experiment service distributes more blocks of runs to VMs with better configuration (e.g., 10 for ec2_m4_4xlarge and only 2 for gwdg_m2_xlarge). Still, the time to finish 10 runs in ec2_m4_4xlarge is not much different than the time to finish 2 runs in gwdg_m2_xlarge. The cost per run of ec2_m4_4xlarge is lower than that of gwdg_m2_xlarge (0.065$ and 0.108$). While ec2_m4_4xlarge and gwdg_m2_xlarge exploited about 40% to 45% of the potential capacity of their cores, ec2_r4_2xlarge used its cores more efficiently (64.15%). This experiment validates the ability of DiFFuSE to simultaneously exploit diverse VMs from multiple clouds and shows that the choice of VM types has a strong impact on the resulting cost. Moreover, it validates the ability of the simple allocation policy (see Section 4.3.2) to balance the computation load among heterogeneous VMs.

5.3.2 | Experiment 2

This experiment evaluates the impact of network performance on application execution time in multi-cloud deployments. In such contexts, components of the simulation application are deployed at different data centers, located at different areas and even different continents. As a result, network characteristics can have a strong effect on application performance.

The setup of this experiment is similar to Figure 4 with data services in our OpenStack-Grid5000 private cloud and worker services in the EC2 public cloud. The workers are configured to be in the same EC2 availability zone in order to reduce network latency. The data services provide data on animals, herds, movements and neighbours. When a worker is initialised, the worker obtains these data causing a traffic of about 2GBs from the private cloud to the public cloud over a wide-area Internet connection. This is repeated for every worker that joins the simulator, having a negative effect on the performance of the BVDV simulator.

To reduce this negative effect, one can take advantage of the replication feature supported by DiFFuSE. Specifically, the data resources provided by data services are configured to support the replicated management style and the worker services are configured to play both user and provider roles for these resources (see Section 4.2.1). This enables the worker that first obtains the data to provide replicas of the data to other workers. In this way, data transfer traffic is isolated within the boundary of the public cloud, which provides high network performance.

We compare data transfer times for the BVDV simulator when data replication is enabled (**E**) and disabled (**D**) for different numbers of workers (4, 8, 12, 16 workers) processing a total of 40 simulation runs. Fig. 13 shows that the time is lower when the replication feature is used, regardless of the number of workers. The amount of data transferred inside (*I*) and outside (*O*) the public cloud is shown in Fig. 14. Regardless of the number of workers, the portion of the traffic inside the public cloud is greatly increased when replicated management is enabled. This experiment validates that the replication feature of DiFFuSE helps reducing data transfer times in multi-cloud deployments.
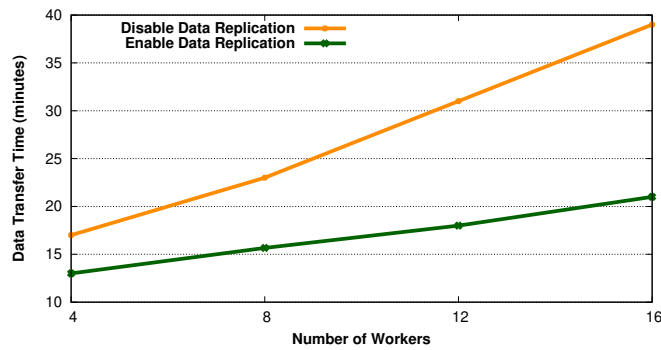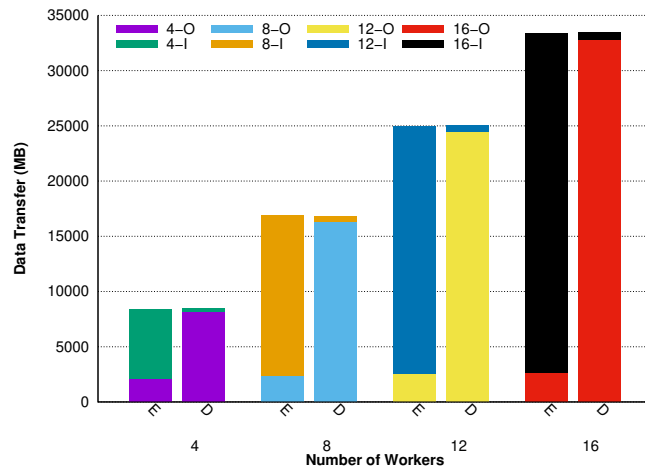

FIGURE 13 Number of Workers Vs Data Transfer Time

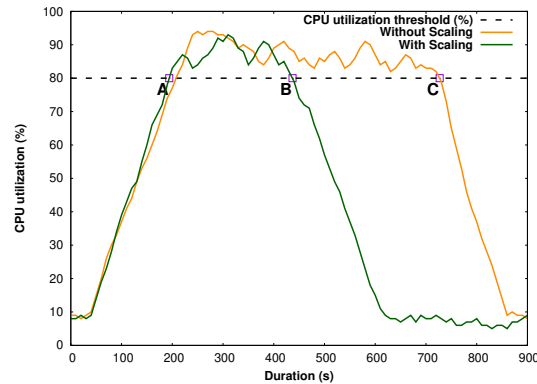**FIGURE 14** Number of Workers Vs Data Transfer

FIGURE 15 CPU utilisation (%) of the BVDV data service in response to 20 workers (EC2.r4.xlarge)

5.4 | Elasticity

This experiment evaluates the elasticity support provided by DiFFuSE. We use a scenario in which several dozens of worker services are simultaneously accessing a data service to load initial data, which causes the data service to become a bottleneck. We compare the performance of the simulator with and without the elasticity feature, which enables scaling out the data service.

Fig. 15 shows the CPU utilisation (%) of the data service in response to 20 worker services. All services are deployed in EC2 r4.xlarge VMs. The green and orange lines represent the cases with and without elasticity. The scale-out action is triggered if the CPU utilisation of the data service goes over 80% (the point A), which results in adding one more VM with the data service (green line from point A to B). Without elasticity, the single data service is busy with processing requests from the workers (orange line from point A to C). Importantly, the duration of processing data requests with elasticity is about 30% shorter than without elasticity, showing the usefulness of elasticity support in DiFFuSE.

6 | CONCLUSION

The main goal of this work is to facilitate building and executing epidemic simulation applications in the cloud. To this end, the paper proposes DiFFuSE, a framework for structuring such applications as cloud-native applications composed of a set of distributed, collaborating services, which are independently developed, deployed and scaled in multi-cloud environments. The framework provides (1) mechanisms that allow services to exchange data in a reliable and scalable way and to automatically handle failures, (2) reusable code for managing experiments, simulation data, and computations following the bag-of-tasks model, and (3) tools for configuring the deployment and auto-scaling of services across clouds in a declarative way.

The paper also describes the use of DiFFuSE to restructure two legacy applications (simulators of BVDV and MAP spread) into elastic, cloud-native applications. Based on these applications, a comprehensive set of experiments were performed, demonstrating the advantages of the framework. These advantages are outlined next. DiFFuSE can exploit variable numbers and types of cloud resources, allowing making different trade-offs between performance and cost. It can automatically handle a wide range of failure and recovery scenarios, reducing the time to complete the simulation. DiFFuSE allows using resources from multiple clouds; notably, it allows combining private resources with those of public clouds, enabling, for instance, small institutions, such as local centers for disease control and prevention, to find cost-optimal resource allocations. Moreover, the replication support of DiFFuSE helps reducing data transfer times in multi-cloud deployments. Finally, DiFFuSE allows elastically modifying resource allocations to adapt to dynamic variations in resource demand; notably, it allows automatically and independently scaling the data service to avoid performance bottlenecks.

We intend to continue this work in three dimensions. First, we will apply DiFFuSE to further simulation applications, which will produce useful feedback for improving the framework. Second, we will refine the basic allocation and failure-handling policies in order to optimise the performance of an application deployment, even in the case of failures. Third, we will extend DiFFuSE with policies for intelligent deployment optimisation, integrated into the PaaSage platform.[27] This will provide us with a fully automatic solution for identifying the best initial deployment for a simulation application and for dynamically modifying this deployment to react to environment changes.

ACKNOWLEDGMENTS

References

1. Eriksson H, Timpka T, Spreco A, Dahlstrom O, Stromgren M, Holm E. Dynamic Multicore Processing for Pandemic Influenza Simulation. In: *AMIA Annual Symposium Proceedings*; 2016: 534-540.

2. Pham LM, Parlavantzas N, Morin C, et al. DiFFuSE, a Distributed Framework for Cloud-Based Epidemic Simulations: A Case Study in Modelling the Spread of Bovine Viral Diarrhea Virus. In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*; 2017: 304-313

3. Pham LM, Parlavantzas N, Morin C, et al. DiFFuSE, a Distributed Framework for Cloud-based Epidemic Simulations: a Case Study in Modelling the Spread of Bovine Viral Diarrhea Virus. Research Report RR-9094, Inria Rennes - Bretagne Atlantique; Rennes: 2017.

4. Bhatele A, Yeom JS, Jain N, et al. Massively Parallel Simulations of Spread of Infectious Diseases over Realistic Social Networks. In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*; 2017: 689-694

5. G. Martin MM, Carretero J. Towards Efficient Large Scale Epidemiological Simulations in EpiGraph. *Parallel Computing* 2015; 42(C): 88–102.

6. Perumalla KS, Seal SK. Reversible Parallel Discrete-Event Execution of Large-Scale Epidemic Outbreak Models. In: *2010 IEEE Workshop on Principles of Advanced and Distributed Simulation*; 2010: 1-8

7. Rao DM, Chernyakhovsky A. Parallel simulation of the global epidemiology of Avian Influenza. In: *2008 Winter Simulation Conference*; 2008: 1583-1591

8. Bisset KR, Chen J, Deodhar S, Feng X, Ma Y, Marathe MV. Indemics: An Interactive High-performance Computing Framework for Data-intensive Epidemic Modeling. *ACM Transactions on Modeling and Computer Simulation* 2014; 24(1): 4:1–4:32. doi: 10.1145/2501602

9. Zou P, Lu sY, Wu dL, Chen lL, Yao pY. Epidemic simulation of a large-scale social contact network on GPU clusters. *SIMULATION* 2013; 89(10): 1154-1172. doi: 10.1177/0037549713482026

10. Holvenstot P, Prieto D, Doncker dE. GPGPU parallelization of self-calibrating agent-based influenza outbreak simulation. In: *2014 IEEE High Performance Extreme Computing Conference (HPEC)*; 2014: 1-6

11. Eriksson H, Raciti M, Basile M, et al. A Cloud-Based Simulation Architecture for Pandemic Influenza Simulation. In: *AMIA Annual Symposium Proceedings*; 2011: 364-73.

12. Sukcharoen P, Pumma S, Mongkolsermporn O, Achalakul T, Li X. Design and analysis of a cloud-based epidemic simulation framework. In: *2012 9th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*; 2012: 1-4.

13. Price RC, Pettey W, Freeman T, et al. SaTScan on a Cloud: On-Demand Large Scale Spatial Analysis of Epidemics. In: *Online Journal of Public Health Informatics; 2(1): ojphi.v2i1.2910*; 2010.

14. Haris M, Manzoor MS. Spatiotemporal Study of Dengue Virus Infection via Cloud Based Framework. *International Journal of Advanced Research in Computer Science and Software Engineering* 2016; 6(6): 155-161.

15. Benedictis AD, Rak M, Turtur M, Villano U. A Framework for Cloud-aware Development of Bag-of-tasks Scientific Applications. *International Journal of Grid and Utility Computing* 2016; 7(2): 130–140.

16. Agarwal D, Prasad SK. AzureBOT: A Framework for Bag-of-Tasks Applications on the Azure Cloud Platform. In: *2013 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum*; 2013: 2139-2146.

17. Król D, Orzechowski M, Kitowski J, Niethammer C, Sulisto A, Wafai A. A Cloud-Based Data Farming Platform for Molecular Dynamics Simulations. In: *UCC 2014*; 2014: 579-584

18. Thai L, Varghese B, Barker A. A survey and taxonomy of resource optimisation for executing bag-of-task applications on public clouds. *Future Generation Computer Systems* 2018; 82: 1 - 11.

19. Ezanno P, Fourichon C, Viet AF, Seegers H. Sensitivity analysis to identify key-parameters in modelling the spread of bovine viral diarrhoea virus in a dairy herd. *Preventive Veterinary Medicine* 2007; 80(1): 49–64.

20. Garcia A, Shalloo L. Invited review: The economic impact and control of paratuberculosis in cattle. *Journal of dairy science* 2015; 98(8): 5019–5039.

21. Qi L, Vergu E, Dutta BL, Ezanno P. Modeling bovine viral diarrhea virus (BVDV) spread between dairy cattle farms at a regional scale: relative contribution of two between-herd transmission pathways.; 2017. Presented at the Society for Veterinary Epidemiology and Preventive Medicine (SVEPM), Inverness, GB.

22. Ezanno P, Fourichon C, Seegers H. Influence of herd structure and type of virus introduction on the spread of bovine viral diarrhoea virus (BVDV) within a dairy herd. *Veterinary Research* 2008; 39(5): 39.

23. Beaunée G, Vergu E, Ezanno P. Modelling of paratuberculosis spread between dairy cattle farms at a regional scale. *Veterinary Research* 2015; 46(1): 295.

24. Beaunée G, Vergu E, Joly A, Ezanno P. Controlling bovine paratuberculosis at a regional scale: Towards a decision modelling tool. *Journal of theoretical biology* 2017; 435: 157–183.

25. Marcé C, Ezanno P, Seegers H, Pfeiffer D, Fourichon C. Within-herd contact structure and transmission of Mycobacterium avium subspecies paratuberculosis in a persistently infected dairy cattle herd. *Preventive veterinary medicine* 2011; 100(2): 116–125.

26. PaaSage. https://www.paasage.eu [21 May 2019]; .

27. Parlavantzas N, Pham LM, Sinha A, Morin C. Cost-Effective Reconfiguration for Multi-Cloud Applications. In: *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*; 2018: 521-528

28. Nanomsg. http://nanomsg.org [21 May 2019]; .

29. GWDG. https://www.gwdg.de [21 May 2019]; .

30. Grid5000. https://www.grid5000.fr [21 May 2019]; .

31. BioEpAR. https://www6.angers-nantes.inra.fr/bioepar [15 July 2019]; .