López *et al. Journal of Cloud Computing* (2025) 14:23
https://doi.org/10.1186/s13677-025-00748-7

Journal of Cloud Computing

RESEARCH Open Access

Fast autoscaling algorithm for cost optimization of container clusters

José María López[1], Joaquín Entrialgo[2], Manuel García[1], Javier García[1*], José Luis Díaz[1] and Rubén Usamentiaga[1]

Abstract
Container clusters are widely used to execute containerized applications in cloud environments. An essential characteristic implemented by these clusters is autoscaling, which is the ability to automatically adapt the computing resources of a cluster to support variable workloads. Precise adjustment of cluster resources to its workload in each autoscaling operation is essential to control cluster deployment costs. Several resource allocation models have been developed with the objective of cost minimization. However, as the number of containers and virtual machines of the cluster increases, resource allocation problems become too complex, and cannot be solved in reasonable time by existing resource allocation models. In this paper we present FCMA (Fast Container to Machine Allocator), a resource allocation algorithm designed to calculate a suitable allocation of the resources of a cluster in autoscaling operations, to minimize cluster deployment costs. The main motivation for the development of FCMA has been to significantly reduce the solving time of the resource allocation problem compared to a previous state-of-the-art optimal Integer Linear Programming (ILP) model. In addition, FCMA addresses secondary objectives to improve fault tolerance and reduce container and virtual machine recycling costs, load-balancing overloads and container interference. We have conducted an experimental evaluation to assess the effectiveness of FCMA, using the ILP model and two heuristics as a baseline. The experiments show that FCMA is much faster than the ILP model, with an average solving time reduction of two orders of magnitude. This gain in speed does not compromise the quality of the solutions, which have a cost on par with those of the ILP model. In comparison to the heuristics, FCMA achieves similar solving times while consistently delivering more cost-effective solutions.

Keywords Cloud computing, Container clusters, Cost optimization, Cluster autoscaling, Resource allocation

Introduction

Container clusters are widely used to run containerized applications. They are made up of a set of compute nodes managed by a control system, offering significant benefits, such as the automation of application rollouts and rollbacks, self-healing to maintain the containers of applications in the desired state, and automatic scaling of the number of containers of applications in response to changes in their workloads. Relevant systems to deploy container clusters are OpenShift [1], Apache Mesos [2] and Kubernetes [3], the latter being the industry standard.

Container clusters can be deployed using as compute nodes either physical servers or virtual machines (VMs) executed in a cloud environment. The latter approach has the advantage of elasticity, since computing resources (that is, VMs) can be increased or decreased as needed. The most significant cloud providers offer managed services to deploy container clusters, such as Amazon EKS

*Correspondence:
Javier García
javier@uniovi.es
[1]Department of Informatics, University of Oviedo, Campus de Viesques, Gijón 33204, Spain
[2]Department of Informatic Systems, Polytechnical University of Madrid, Campus Sur. Ctra. de Valencia, Km. 7, Madrid 28031, Spain

[4], Google Kubernetes Engine [5], and Azure Kubernetes Services [6].

Autoscaling is a crucial characteristic provided by container clusters. It is the ability of a cluster to automatically and dynamically adapt its computing resources to support a variable workload. In the case of clusters of VMs, two types of virtual entities can be scaled: containers and VMs. The autoscaling of containers adjusts the computing resources allocated to cluster applications without altering the cluster computing resources. In contrast, when autoscaling increases or decreases the number of VMs, the cluster computing resources are modified. In cloud environments, cluster costs depend on the VMs deployed for the cluster, so to minimize costs in autoscaling operations, the autoscaling of both VMs and containers must be taken into account, as described in [7].

The autoscaling process of containers and VMs can be horizontal or vertical. Horizontal autoscaling involves scaling the number of virtual entities (containers or VMs) out or in. Vertical autoscaling implies scaling the computing resources allocated to virtual entities (essentially, the amount of CPU and memory) up or down. In addition, autoscaling can be reactive or predictive. Reactive autoscaling scales virtual entities based on metrics, such as CPU or memory utilization. When the selected metrics reach predefined thresholds, the autoscaling processes is triggered. Alternatively, predictive autoscaling works periodically, forecasting the workload for the next time period, and deploying a suitable allocation of computing resources for the period to satisfy the expected workload.

The operating model of predictive autoscaling is much more complex compared to that of reactive autoscaling. The operating stages commonly used in predictive autoscaling are presented in [7]. A scheduling window is defined, and four stages are carried out for each consecutive window. These stages, depicted in yellow in Fig. 1, are: 1) *Workload forecasting*, 2) *Resource allocation*, 3) *Resource deployment* and 4) *Resource execution*. In the *Workload forecasting* stage, the expected workload for the next scheduling window is predicted. Then the computing resources required to support the expected workload are estimated in the *Resource allocation* stage. The calculated resources are passed to the control system of the cluster that deploys the resources during the *Resource deployment* stage. Lastly, the deployed resources are kept in execution during a whole scheduling window in the *Resource execution* stage.

A key concept of the operating model of predictive autoscaling is the temporal overlapping of stage 4 of the current scheduling window with stages 1, 2 and 3 of the next scheduling window, as shown in Fig. 1. This overlapping implies that the available time to accomplish stages 1, 2 and 3 should not exceed the length of the scheduling window. In order to fulfill this restriction, all actions to minimize the time spent in stages 1, 2 and 3 are extremely beneficial.

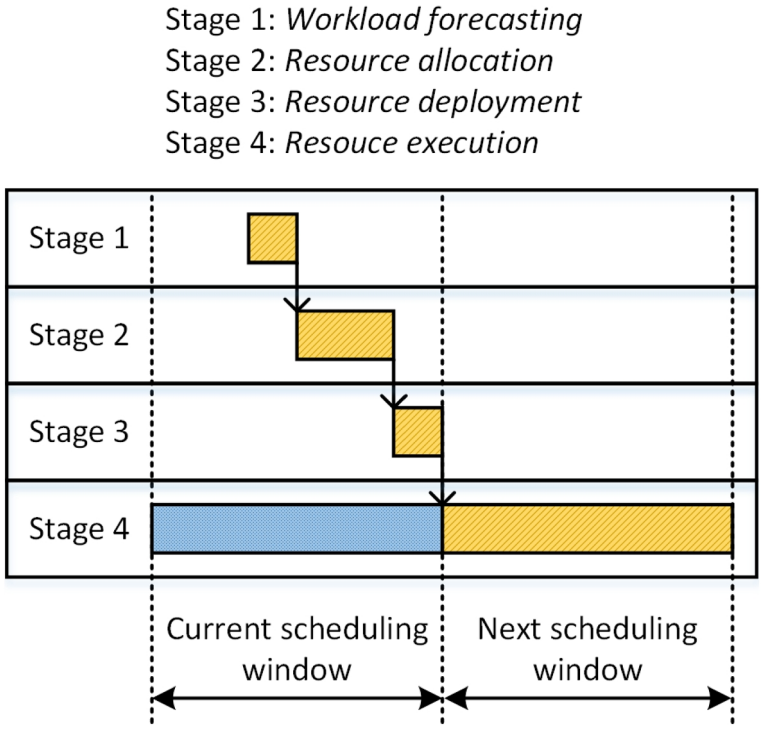

Fig. 1 Predictive autoscaling stages

This paper presents a fast algorithm to solve stage 2 (*Resource allocation*) for the predictive autoscaling of container clusters. This algorithm, called Fast Container to Machine Allocator (FCMA), calculates a suitable allocation of containers and VMs to handle the workload received by a container cluster in a time window, with the primary objective of minimizing cluster deployment cost. For each time window in the service time of the cluster, FCMA determines the number of VMs and containers as well as the computing resources allocated to them. In this way, FCMA operates as a horizontal and vertical autoscaler.

The main motivation for FCMA development was to significantly reduce the time used by the Integer Linear Programming (ILP) model called Conlloovia [7] to solve the allocation problem of containers and VMs, without compromising cost minimization. The experiments carried out with FCMA confirm this reduction in time while maintaining costs. Furthermore, FCMA improves Conlloovia by addressing four secondary objectives that Conlloovia does not take into account. These objectives are: 1) improve fault tolerance, 2) reduce container and VMs recycling costs between successive scheduling windows, 3) reduce load-balancing overloads and 4) reduce container interference. The research presented in this paper offers the following key contributions:

- Development of the FCMA algorithm for the fast allocation of containers and VMs with the primary objective of minimizing the cost of autoscaling operations in container clusters.
- Evaluation of FCMA in comparison with Conlloovia on a range of problems of different sizes shows that FCMA significantly outperforms Conlloovia in terms of solving times, while achieving similar cost results.
- Evaluation of FCMA in comparison with two heuristics, First Fit by Core (FFC) and First Fit by Price (FFP), on a range of problems of different sizes shows that FCMA achieves more cost-effective solutions while maintaining similar solving times.
- Incorporation of additional secondary objectives in FCMA that were not addressed by Conlloovia, thereby enhancing the overall capabilities of the autoscaling procedure.

The rest of the paper is organized as follows. "Related work" section discusses the related work. In "System model and objectives" section, the system model describing the resource allocation problem as well as the objectives of the algorithm are explained. "FCMA algorithm" section provides a detailed description of the algorithm. In "Experimental results" section, the experiments carried out to compare FCMA with Conlloovia and two heuristics are described, and the results obtained are analyzed. This section also carries out a scalability analysis of FCMA. "Practical considerations" section discusses practical considerations regarding the integration of an FCMA-based autoscaler into a Kubernetes cluster. Finally, "Conclusions" section concludes the paper.

Related work

Cloud computing brought significant advancements in deploying computing resources through virtualization, introducing a new paradigm for resource availability. However, it also presents numerous challenges, the trade-off between performance and cost being the most crucial. Subsequently, the advent of containerization, based on lightweight virtualization, has emerged. This technology enhances the flexibility of deploying applications on computational resources, although introducing further challenges. An open problem is to determine the computational resources needed to support a given workload, while fulfilling a certain performance target. This must be done optimally in terms of monetary cost, and the solution should adapt as the workload evolves. In the literature, this challenge is often referred to as scaling or autoscaling, and its nature can vary depending on the specific workload or the criteria employed for optimization. For example, in [8], the authors allocate containers based on various optimization targets. Their method relies on genetic algorithms. The work of [9] encompasses a survey of existing alternatives for managing resource provisioning for a workload represented as microservices on containers. In general, autoscaling is a complex problem that has been widely studied [10].

Our research focuses on resource scaling to meet workload requirements, which are represented as requests per second. The primary optimization criterion is the cost of resources, which includes both VMs and containers. The cost is mainly given by the VMs on which the containers are deployed.

Many studies in the literature address the scaling of required resources, either VMs or containers, as the workload changes. However, few combine the two types of resources, and even fewer consider both horizontal and vertical scaling simultaneously.

In [11], the authors scale the number of VMs and containers horizontally as the workload changes, using a threshold of the response time predicted by a queue model as the trigger to scale. In [12], the author presents a method for autoscaling in geographically distributed cloud clusters. The approach initially scales containers horizontally, and in cases of resource shortages, new VMs are deployed to support the workload, implementing horizontal scaling at both the container and VM levels.

Less common is the vertical scaling of both VMs and containers, as seen in [13], where the scaling process is triggered by resource utilization levels.

Other works focus on scaling container resources by increasing both the number of replicas (horizontal scaling) and their capacity (vertical scaling). For instance, in the works of [14, 15] the autoscaling process reacts to workload changes when a controlled performance metric crosses a given threshold. In contrast, [16, 17] use predictive techniques to forecast the evolution of the controlled performance metric and conduct scaling in advance using reinforcement learning and machine learning, respectively. In a recent evolution of their last work [18], the authors report an important cost reduction and improvement in the QoS applying neural networks in workload prediction.

Recently, some works have explored the use of machine learning to enhance the autoscaling process. For example, in [19] a cloud-native autoscaling mechanism is developed for microservices, utilizing a convolutional workload predictor capable of detecting bursts within the forecast, which improves scaling accuracy. In [20], a container-based microservices autoscaling mechanism is presented, using joint workload prediction (CPU-memory) with the Random Forest method. Based on this prediction, the autoscaler determines the number of replicas (horizontal scaling, based on CPU forecast), and the amount of resources (vertical scaling, based on memory forecast) assigned to each container. Similarly in [21], the authors propose an autoscaling process in Kubernetes based on workload prediction using machine learning and burst detection. Based on these predictions, they introduce a hybrid autoscaler that increases the number of pods (horizontal scaling) and also increases the resources allocated to each pod (vertical scaling). Lastly, in [22] based on a mixed predictive-reactive workload prediction module, the autoscaling is carried out by a learning model that selects the most suitable configuration for the given conditions.

More comprehensive studies such as [23–26], address the scaling of both VMs and containers. In [23], the authors analyzed the challenge of scaling across different virtualization layers, a concept they refer to as Co-Scale. In [24] scaling is applied to both VMs and containers, using horizontal and vertical scaling. Their method evaluates several scaling policies based on workload predictions. The hierarchical model presented in [25], synchronizes the scaling of VMs and containers, and develops a simulation tool to assess its effectiveness. Finally, in [26] the scaling process is conducted in a heterogeneous VM environment, determining the number and size of containers based on workload predictions.

Previous solutions often rely on heuristics to conduct the scaling process. However, other works have developed mathematical optimization models to guide the scaling. In [27], the authors develop a multi-objective Integer Linear Programming (ILP) method to find the optimal allocation of containers over a set of given homogeneous or heterogeneous nodes. Other works carry out a whole scaling process optimizing both the VMs and containers, as seen in [28–31]. These methods predict how the workload evolves and solve the optimization model to obtain the necessary resources to guarantee the established service levels. The main disadvantage of these approaches is the complexity of the model and the high resolution time required for a long time analysis interval.

In a previous work, [7] developed the Conlloovia optimization model for the joint autoscaling of containers and VMs for cost optimization in container clusters. Conlloovia simplified some aspects of previous state-of-the-art models to compute solutions in reasonable times, making it suitable for use in real systems. However, when problem sizes are too large or the available time to compute allocations is too short, even Conlloovia may not be fast enough to find a solution because of memory limitations. FCMA overcomes these issues by providing a new methodology to reduce the problem size, making it more applicable than Conlloovia. This methodology combines ILP methods and heuristics to reach a near-optimal solution quickly enough, even under the most demanding conditions.

System model and objectives

The problem addressed in this research essentially involves the optimal allocation of application containers on cloud nodes to meet load requirements, with the objective of cost minimization.

The notation used in the model is summarized in Table 1. There are N applications denoted by A_a, where $a \in [1, N]$. Each application is a memory-less service that receives requests and provides a timely response. Let w_a be the workload for application A_a, which can be obtained through measurement or prediction. Workload is assumed to be expressed in requests per second (rps).

Applications run on nodes of cloud providers, which are implemented as virtual or bare-metal machines. Cloud providers rent out different types of nodes, so that each client can select the most appropriate ones for their applications. The types of nodes are called *instance classes* by some providers, and this will be the term used in this paper. Instance classes differ in the number of CPU cores, memory and price. There are M instance classes, which are denoted by I_i, where $i \in [1, M]$. These instance classes may come from one or multiple cloud providers. Each instance class I_i defines a number of available cores, c_i, a memory

Table 1 Notation

Inputs		
Apps	N	Number of applications
	A_a	Application, $a \in [1, N]$
	w_a	Workload of application A_a (rps)
Nodes	M	Number of instance classes
	I_i	Instance class, $i \in [1, M]$
	c_i	Number of cores of instance class I_i
	m_i	Memory of instance class I_i (GB)
	p_i	Price of instance class I_i ($/h)
Containers	$C_{k,a,i}$	Container class *k* of application A_a running on a node of instance class I_i
	$c_{k,a,i}$	Cores required by container class $C_{k,a,i}$
	$m_{k,a,i}$	Memory required by container class $C_{k,a,i}$ (GB)
	$r_{k,a,i}$	Performance (rps) of container class $C_{k,a,i}$
Outputs (unknowns)		
	X_i	Number of running nodes of instance class I_i
	$Y_{k,a,i}$	Number of running $C_{k,a,i}$ containers

capacity, m_i, in gigabytes, and a price p_i expressed in dollars per hour.

Applications are deployed using types of containers called *container classes*. Each application can be implemented using different container classes denoted by $C_{k,a,i}$, where index *k* refers to one of the container classes of A_a, and *i* refers to the instance class I_i where it can be run. Every container class reserves computational resources as CPU, represented by $c_{k,a,i}$ (cores), and memory, represented by $m_{k,a,i}$ (gigabytes), from the node of instance class I_i in which it is deployed. Additionally, each container of container class $C_{k,a,i}$ is assumed to serve $r_{k,a,i}$ requests per second of application A_a. That number of rps defines the container performance.

The algorithms presented in this paper can be applied to a single deployment of applications, but they can also be a component of an autoscaler that looks for the best container to machine allocation at predefined scheduling windows.

The main objective is to minimize the cost of a set of applications taking into account the following considerations:

- One application can be deployed on one or multiple nodes, of the same or different instance class, using multiple container classes.
- The accumulated performance of all the containers associated to an application must be high enough to process the application workload.
- Every node in the system must have enough capacity to satisfy CPU and memory demands of all the containers running on it.

The minimization problem can have multiple solutions, all with the same cost. Therefore, some secondary objectives are proposed:

- Improve fault tolerance. A parameter called *Single Failure Maximum Performance Loss* (SFMPL) is defined for each application, denoting the maximum performance loss when a single node fails, relative to the total application performance. This limit should be fulfilled provided it does not increment cost.
- Reduce container and VM recycling costs in the context of autoscaling. These costs are generated by the change in the number and types of VMs and containers in the transition between the current and the next scheduling window. The greater the number of containers and VMs that change, the higher the cost of recycling. Because of this, solutions with a low number of container classes and instance classes are preferred to solutions with the same cost but a higher number of container classes and instance classes. Although container and VM recycling algorithms will be part of a future work, FCMA algorithms have been devised to facilitate these recycling operations.
- Reduce load-balancing overloads. Two solutions with the same cost can involve either many "small" containers or a few "big" containers. In the first case inter-node and intra-node load-balancing overloads can be high, so the second solution may be more suitable.
- Reduce container interference. Containers are a form of virtualization that isolates computational resources, so that the performance of a container is independent of other containers running in the system. Although this approach holds reasonably well with a small number of containers, as the number increases, the actual behavior deviates from the ideal, leading to performance losses. In this regard, a solution that uses fewer containers per node at the same cost is preferable.

Some of the secondary objectives may be contradictory. For example, reducing the number of nodes reduces load-balancing overload, but has a negative impact on fault tolerance since in this case the failure of a single node produces a higher performance loss.

FCMA algorithm

This section begins by introducing two key concepts: *minimum-size container*, in "Minimum-size container" section, and *instance class family*, in "Instance class families" section. Next, "Illustrative example" section uses these concepts to present an example that will be used to illustrate the different phases of the FCMA algorithm. An overview of these phases are presented in "Overview of FCMA phases" section, and each of these phases is developed in depth in "Phase 1. Partial ILP problem", "Phase 2. Node aggregation", "Phase 3. Container allocation" and "Phase 4. Container aggregation" sections. Finally, "FCMA computational complexity" section analyses the computational complexity of FCMA.

Minimum-size container

Containers are a runtime deployment technology that isolates applications or parts of them, allowing, among other things, to assign each container the amount of computational resources it needs. These resources are basically CPU, expressed as an integer or fractional part of cores (millicores), and a certain amount of memory. This paper focuses on CPU-bound applications, where execution times are highly dependent on the CPU resources allocated. Applications with significant memory or I/O demands fall outside the scope of this study. In this context, CPU and memory resources behave differently. For memory, there is a minimum amount below which the container cannot execute. However, once this minimum is exceeded, a higher value does not usually affect the container's operation, so it does not usually improve its response times. Regarding the CPU, the situation is entirely different. Assigning more millicores to the container usually reduces response times until certain limit is reached. For example, a container running a single-threaded application can improve its response time when its number of millicores is increased but only up to 1000 millicores.

Next, the assumptions that support the concept of minimum-size container are formally established.

Assumption 1. The response of a container to a sequence of requests is independent of the memory allocated to that container, as long as it is sufficient to meet its demands at all times. Therefore, increasing the memory allocated to the container beyond that value provides no benefit, as experimentally shown in [32, 33].

Assumption 2. The response time of a container is a monotonically decreasing function of its number of cores. This assumption essentially states that container performance cannot decrease as the number of cores increases. This assumption is trivially true for single-threaded applications, as a single thread cannot use more than 1000 millicores, and below that point, the response time remains proportional to the allocated millicores. For multi-threaded applications, it is still a reasonable assumption since additional cores provide more opportunities for parallelism, and thus shorter response times. However, Amdahl's law predicts diminishing returns, and resource contention-such as cache or memory bandwidth limitations-can degrade performance [33]. Nevertheless, this effect depends on the number of threads rather than strictly on the number of cores. Therefore, if the application is properly configured, the assumption still holds.

Assumption 2 does not define valid response times. Response times could be deemed valid when all response times are below a certain maximum, or simply are within the 95$^{\text{th}}$ percentile.

Figure 2 depicts an example of the relationship between the response time of a single request, t_R, and the number of cores of the container, taking into account the previous assumptions. There is a minimum number of cores, $\text{cores}_{\text{min}}$, below which the response time exceeds a maximum allowable value, $t_{R\text{max}}$. The value of $\text{cores}_{\text{min}}$ can be obtained by performing a series of experiments with a single request using a fractional value of cores, millicores (mcores), that is incremented progressively while response time is measured. Experiments are repeated until the response time for the given mcores is just below t_{Rmax}. The performance of the minimum-size container in requests per second (rps) can be obtained assuming a sequence of requests equally spaced t_{Rmax} seconds, i.e, a performance of $1/t_{Rmax}$ rps.

Using n minimum-size container replicas provides n times the performance of an individual container, and in addition, they can process at least n simultaneous requests within an acceptable time.

The minimum-size container memory must also be determined. This can be carried out by measuring the memory consumption of the container when t_{Rmax} is reached. For example, in the case of Docker containers, memory consumption can be obtained using the `docker stats` command.

The minimum-size container is one particular case among the possible container classes $C_{k,a,i}$, and we will use the subindex $k = 0$ for it. Table 2 summarizes this notation. FCMA uses minimum-size containers to better match container performance to application load, thereby reducing overcapacity and, consequently, deployment costs.

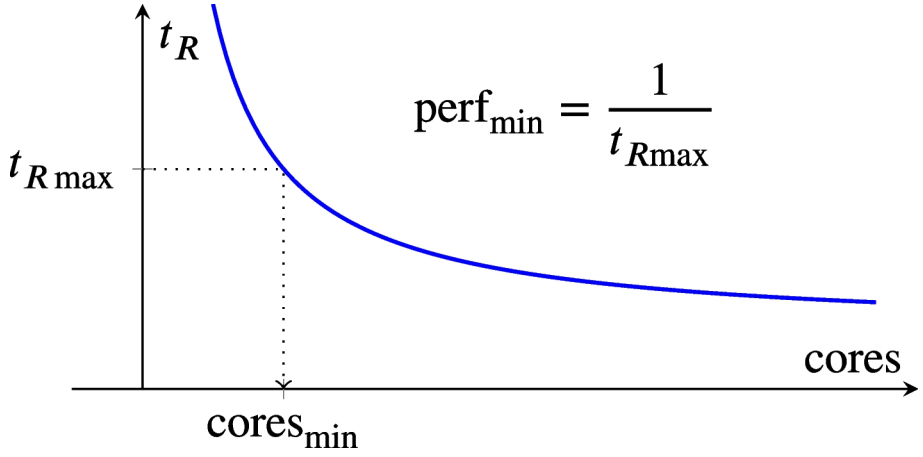

Fig. 2 Determining the minimum number of cores for a container

Table 2 Minimum-size container notation

$C_{0,a,i}$	Minimum-size container (MSC) for application A_a on instance class I_i
$c_{0,a,i}$	$\text{cores}_{\min}$ required by MSC $C_{0,a,i}$
$m_{0,a,i}$	$\text{mem}_{\min}$ required by MSC $C_{0,a,i}$ (GB)
$r_{0,a,i}$	Performance $= 1/t_{R\,\max}$ (rps) of MSC $C_{0,a,i}$

Instance class families

One of the sources of complexity in the problem of cost minimization is the high number of available instance classes. A single provider can offer close to a thousand instance classes to choose from. Most instance classes are types of VMs grouped by the cloud provider using diverse criteria such as the characteristics of the underlying hardware (that is, the type of processor, memory technology, etc.) and the memory-to-core ratio. For example, in the AWS provider, the group of instances called M6a uses an AMD EPYC 7R13 as the underlying processor and a memory-to-core ratio of 8. On the other hand, the group called R6a uses the same processor, but a memory-to-core ratio of 16.

In this paper we introduce the concept of instance class family, defined as the collection of all the instance classes that use the same underlying hardware. Using the previous example, all the instance classes of the groups M6a and R6a belong to the same instance class family. However, each instance class in an instance class family has a different amount of computing resources. For example, the m6a.large instance class has 1 core and 8 GB whereas the r6a.2xlarge has 4 cores and 64 GB. Despite the differences in resources, any container with an identical number of cores and memory deployed on the two types of instance classes provides the same performance, because the underlying hardware (and therefore, the computing speed) is identical in both cases.

Working with instance class families will help reduce the problem size, as will be shown in "Phase 1. Partial ILP problem" section.

Illustrative example

In order to better illustrate the problem that FCMA solves, and the particular sub-problems considered in each phase of the algorithm, an example is presented in Table 3.

This example works with two families of instance classes, called A and B in order to avoid using commercial names. Each family contains 16 instance classes, which can be classified into two groups, one with a memory-to-core ratio of 4, and the other with a memory-to-core ratio of 16. The name of each instance class is composed of two letters and a number. The first letter identifies the family of the instance class, the second letter identifies the memory-to-core ratio (C for 4, and M for 16), and the number specifies the number of cores of the instance class. So, for example AC8 is an instance class of family A, with 8 cores and 32 GB of RAM (4 GB per core).

Instance class family B provide similar instance classes in terms of cores and memory, but may be implemented from processors with different microarchitecture and therefore different performance and prices. Note that prices shown in Table 3 are prices per core so, to obtain the price of an instance class the value in the table must be multiplied by the number of cores. For example, the price of an AC4 is $4 \times 0.1 = 0.4$ \$/hour.

The table also shows the characteristics of the applications and the minimum-size containers. Three applications, A_1, A_2 and A_3, appear in the example, each one with its workload in rps. Also, for each combination of application and family, a minimum-size container is specified. Its parameters (millicores, memory and performance) are also listed in the table. Despite the *i* subindex,

Table 3 Example of a cost minimization problem

Instance class family	Instance classes I_i	Price p_i/c_i ($/hour/core)
A	AC1, AC2, AC4, AC8, AC18, AC24, AC36, AC48	0.10
	mem-to-core ratio = 4	
	AM1, AM2, AM4, AM8, AM18, AM24, AM36, AM48	0.40
	mem-to-core ratio = 16	
B	BC1, BC2, BC4, BC8, BC18, BC24, BC36, BC48	0.07
	mem-to-core ratio = 4	
	BM1, BM2, BM4, BM8, BM18, BM24, BM36, BM48	0.11
	mem-to-core ratio = 16	

Apps.		Minimum-size containers $C_{0,a,i}$			
A_a	w_a (rps)	Family	$c_{0,a,i}$ (mcores)	$m_{0,a,i}$ (GB)	$r_{0,a,i}$ (rps)
A_1	3	A	600	0.95	0.50
		B	800	0.20	0.40
A_2	60	A	5000	17.45	0.40
		B	7600	15.10	3.00
A_3	45	A	1500	8.20	2.00
		B	1200	6.40	2.50

it is not necessary to specify these data for the 16 instance classes of each instance class family, because the values are the same for all instances in the same family. This not only simplifies the problem statement but also greatly reduces the solving time.

Overview of FCMA phases

Figure 3 presents the four phases of the FCMA algorithm. They will be briefly described using the previous example as a reference. Figure 4 is a graphical representation of the input data for the example problem, and the output results of each phase.

Phase 1. Solving the partial ILP problem. A simplified version of the problem is formulated and solved using ILP. The following assumptions are applied in this version:

- Memory constraints are ignored, assuming that all nodes have sufficient memory to host all containers.
- Core capacity is aggregated among nodes of the same family. For example, in four nodes of an instance class with 2 cores each, it is assumed that any combination of containers requiring a total capacity less or equal than 8 cores can be allocated.

The inputs for this phase are all possible instance classes, the application workloads, and the minimum-size containers for each application-family pair. Figure 4a depicts each instance class using a box, whose size is proportional to the number of cores. Blue and orange are used for the families A and B, respectively. Rectangles are used to depict the minimum-size containers, filled in blue or orange depending on the family, and outlined with a different color per application: gray, red and yellow for applications A_1, A_2 and A_3, respectively. The area of the fill is proportional to the number of cores required by the container.

This first approximation to the problem yields a number of nodes for each instance class and a number of application's minimum containers to be allocated in those nodes. The result for the previous example at the output of phase 1 is the following:

- Nodes: 4 AC1 with a cost of $4 \times 1 \times 0.1 = 0.4$ \$/hour, and 10 BC18 with a cost of $10 \times 18 \times 0.07 = 12.6$ \$/hour, shown in the upper part of Fig. 4b.
- Minimum-size containers: 6 for application A_1 in family A, 20 for application A_2 in family B and 18 for

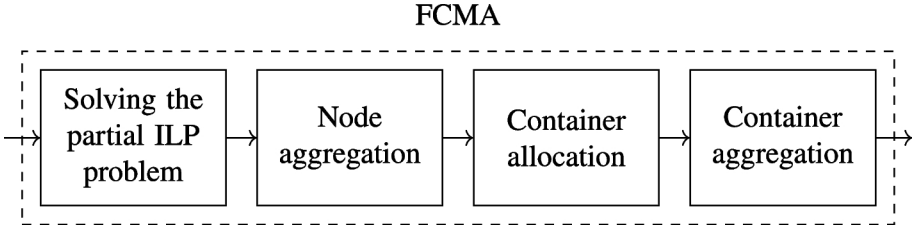

Fig. 3 FCMA phases

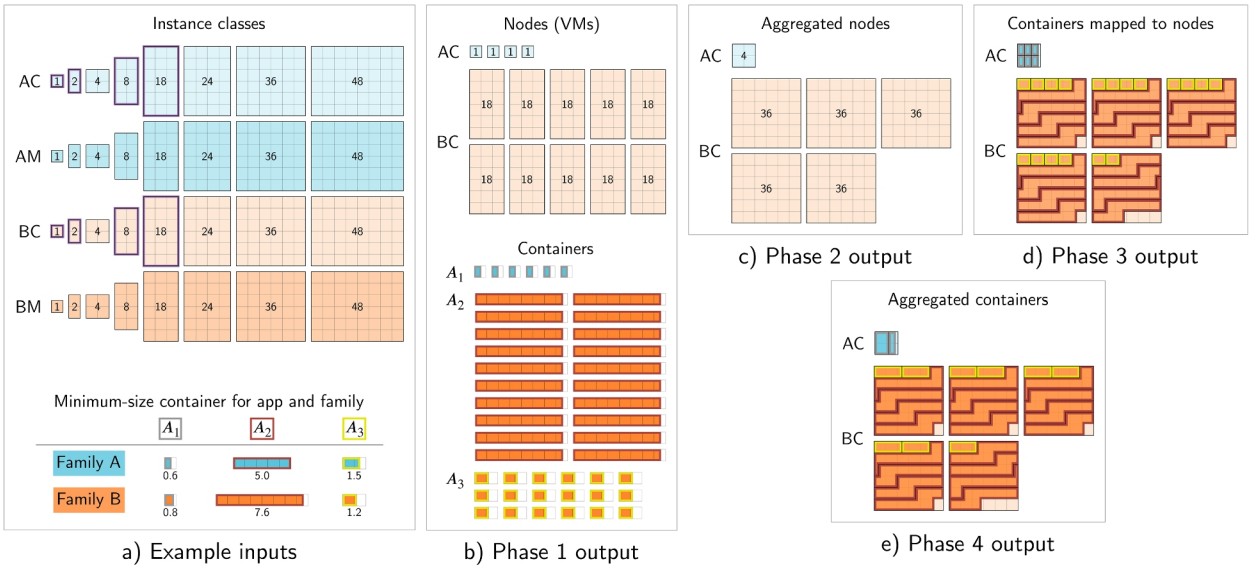

Fig. 4 Graphical representation of the problem solved in each phase

application A_3 in family B, shown in the lower part of Fig. 4b.

The FCMA solution at the output of this phase may not be feasible, as indeed is the case with the given example, since the 6 containers with 600 mcores each cannot be mapped to 4 nodes with 1000 mcores each. This will be solved in next phases.

Phase 2. Node aggregation. This phase aims to aggregate the capacity of all the nodes within the same family from phase 1 by replacing the original set of nodes with another set from the same family. The resulting set consists of larger nodes with the same total capacity in terms of cores and memory. This aggregation facilitates the work of the container allocation in the next phase. As shown in Fig. 4c, after this phase, the four AC1 nodes are replaced by one AC4 node with the same cost. In addition, the ten BC18 nodes are replaced by five BC36 nodes. With this transformation, the six minimum containers for A_1 can now be assigned to nodes from family A with no cost increase. More information can be found in "Phase 2. Node aggregation" section.

Phase 3. Container allocation. Containers are allocated to nodes in this phase using a First Fit Decreasing (FFD) allocator, [34], with custom modifications. In the most favorable scenario, the FFD algorithm is able to assign the containers to nodes in such a way that the sum of cores and memory of the containers assigned to each node does not exceed the node capacity. However, it could happen that FFD cannot perform the allocation, especially in the case of applications with high memory requirements. In that case, it may be necessary to promote some nodes to others with higher capacity and price, choosing the cheapest option. Another alternative to node promotion is adding nodes. Details of this phase can be found in "Phase 3. Container allocation" section.

In the example neither promotion nor node addition is required. Results are (see Fig. 4d):

- 6 minimum-size containers for A_1 in node AC4.
- 4 minimum-size containers for A_2 in each of the 5 BC36 nodes.
- 4 minimum-size containers for A_3 in the first 4 BC36 nodes and 2 in the last BC36 node.

Phase 4. Container aggregation. Minimum-size container performances may be small compared to application workload. In this case a high number of containers is required to handle all the application workload. A large number of containers not only increases overhead on load-balancers, but also leads to a greater interference between containers. For this reason, the minimum-size containers of the same application running on the same node can be grouped to form larger containers. Container aggregation is driven by parameters that restrict the allowable aggregations, as explained in "Phase 4. Container aggregation" section.

For example, these parameters could dictate that for application A_1 the allowed container aggregations are 2 and 4, while A_3 containers can only be aggregated by 2, and A_2 containers cannot be aggregated. With these restrictions, as Fig. 4e shows, the 6 minimum-size containers of application A_1, with 600 mcores each, allocated to the AC4 node, are aggregated into 1 container with

2400 mcores and 1 container with 1200 mcores, while providing the same total performance in rps. Regarding application A_3, the 18 containers of 1200 mcores and 2.5 rps, allocated in BC36 nodes, are aggregated into 9 containers of 2400 mcores and 5 rps. Finally, containers of application A_2, remain as they were at the end of previous phase.

Container aggregation also has beneficial effects on memory consumption. Typically, increasing the number of mcores associated with a container results in little or no increase in memory usage. Applied to application A_1 , this would mean that the 6 minimum-size containers would consume $6 \times 0.95 = 5.7$ GB, while after aggregation in two containers they would consume approximately $2 \times 0.95 = 1.9$ GB, just one-third.

Phase 1. Partial ILP problem

This section presents the mathematical formulation of the partial ILP problem together with a set of simplifications that can further reduce the size of the problem without compromising the optimality of the solution.

Mathematical formulation

The notation for the formulation of the problem has already been introduced in Tables 1 and 2. The problem is defined by the following expression:

$$\text{Minimize} \sum_{i=1}^{M} p_i \cdot X_i \tag{1}$$

Under the following constraints:

$$\sum_{i=1}^{M} Y_{0,a,i} \cdot r_{0,a,i} \geq w_a \qquad \forall a \in [1, N] \tag{2}$$

$$\sum_{a=1}^{N} c_{0,a,i} \cdot Y_{0,a,i} \leq X_i \cdot c_i \qquad \forall i \in [1, M] \tag{3}$$

$$c_{0,a,i} \cdot Y_{0,a,i} \leq c_i \cdot Y_{0,a,i} \qquad \begin{array}{l} \forall i \in [1, M], a \in [1, N] \\ \text{with } X_i, Y_{0,a,i} \in \mathbb{N} \end{array} \tag{4}$$

Equation (1) defines the objective function, which is the cost of all the nodes in the deployment. The constraints in Eq. (2) establish that the sum of performances of minimum-size containers from each application must be greater than or equal to the application workload. $Y_{0,a,i}$ is the number of minimum-size containers of application A_a running on nodes of instance class I_i. Term $r_{0,a,i}$ is the performance of the minimum-size container for application A_a running on a node of instance class I_i. Term w_a is the load to be processed by application A_a. Performances and loads are expressed in the same units, requests per second.

Equation (3) imposes the restriction that instance class I_i can allocate containers whose sum of cores must be less than or equal to the aggregated capacity in cores of all the nodes of the instance class I_i, that is, $X_i \cdot c_i$.

Finally, restrictions in Eq. (4) prevent any container from being allocated in a node with insufficient cores. Without this restriction, a container requiring 3 cores could be allocated to 3 nodes with 1 core each.

The containers in these equations are the minimum-size containers for each application. However, the equations are fully applicable to containers of other sizes.

Simplifications

The number of variables in FCMA's partial ILP problem can be reduced by employing several simplifications that makes the problem size smaller without altering its solution. Three different strategies are used. The first strategy removes from the input parameters those instance classes that cannot be part of an optimal solution. This strategy is used in simplifications 1 and 2. The second strategy, implemented by simplification 3, eliminates instance classes that can be replaced by smaller ones without affecting the optimality of the solution. The last strategy, implemented by simplification 4, removes from the problem those variables that are known to be null (they must be zero in the optimal solution).

Simplification 1. Remove extended memory instances. If two instance classes in the same family are identical except for the amount of memory, the one with more memory and therefore higher cost will not be part of the solution to the partial ILP problem. The reason is simple: memory is disregarded in the partial problem, and for any instance class with extended memory, there exists an instance class with the same core capacity and performance at a lower cost. Therefore, the latter would be part of the optimal solution. Applying this simplification to the example, all instance classes AM and BM cannot appear in the solution, so they can be removed from the problem.

Simplification 2. Remove instance classes with insufficient cores. When an instance class I_i does not have enough cores to allocate the minimum-size container of all the applications, the associated variable X_i must be zero in the solution. Consequently, instance class I_i is removed from the partial ILP problem.

Simplification 3. Remove instance classes multiples of each other. This simplification is related to instance classes within the same family that have resources and prices multiples of those of another instance class. In

the example, instance class AC2 has double the CPU resources and price of instance class AC1, so it can be removed. The following theorem develops a simplification based on this proportionality.

Theorem 1 Let I_i and I_j be two instance classes within the same family, where I_j has k times more cores than instance class I_i, with k being a natural number greater than 1. Suppose that the cost of instance class I_j is k times the cost of instance class I_i, and all the containers that fit in instance class I_j also fit in instance class I_i.

It holds that there exists an optimal solution to the partial ILP problem in which instance class I_j does not appear.

Proof Let us start with an optimal solution that includes nodes of instance class I_j and perform a number of transformations.

Firstly, the number of nodes of instance class I_i is increased to become $X'_i = (X_i + k \cdot X_j)$ and the nodes of instance class I_j are removed. This transformation does not change the cost.

Secondly, all the containers in instance class I_j are moved to instance class I_i. Thus, instance class I_i now allocates $Y'_{0,a,i} = (Y_{0,a,i} + Y_{0,a,j})$ containers of A_a. These new values $Y'_{0,a,i}$ fulfill the three sets of restrictions of the partial ILP problem. Restrictions in Eq. (2) are satisfied because the number of containers as well as their performance remains unchanged after the movement. Restrictions in Eq. (3) are satisfied because the newly created k nodes of instance class I_i have the same aggregated capacity as the nodes removed from instance class I_j, so it is enough for the moved containers. Finally, restrictions in Eq. (4) are fulfilled since all the containers that fit in nodes of instance class I_j also fit in nodes of instance class I_i.

Therefore, the solution after applying the two transformations is valid and has the same cost as the optimal solution so, it is proved to be also optimal.□

The results of simplifications 2 and 3 are easier to understand if they are applied sequentially to each application to obtain different subsets of allowable instance classes. The final result of the simplifications will be the union of these subsets. For example, application A_1 fits into instance class AC1 (1 core), and thus the other AC instance classes can be removed for this application, because all of them have a number of cores multiple of 1. Application A_2 removes AC1, AC2 and AC4 using simplification 2, because they do not have enough cores. Then, AC24, AC36 and AC48 can be removed using simplification 3, because AC24 is a multiple of AC8, AC36 is a multiple of AC18 and AC48 is a multiple of AC8. Finally, the minimum-size container of A_3 requires 1.5 cores, so AC1 can be removed using simplification 2, but AC2 remains. Instance classes AC4, AC8, AC18, AC36 and AC48, which are multiple of AC2, can be removed using simplification 3.

Performing the union of the surviving instance classes for each application, the result is AC1, AC2, AC8, and AC18 for family A, and BC1, BC2, BC8, and BC18 for family B. These instance classes are marked with a bolder outline in Fig. 4a.

Simplification 4. Remove null $Y_{0,a,i}$ variables. When any of the previous instance classes, I_i, does not have enough cores for the minimum-size container of an application, A_a, the corresponding variable $Y_{0,a,i}$ is eliminated from the problem. In this way, in addition to reducing the number of variables $Y_{0,a,i}$, the third set of constraints of the linear programming problem, given by Eq. (4), is also eliminated. In our example, variables $Y_{0,a,i}$ for A_2 and instance classes AC1, AC2, AC4, BC1, BC2 and BC4 are removed because they do not have enough cores to allocate at least one A_2 container.

After all the simplifications, the partial ILP problem associated to the example works with 18 $Y_{0,a,i}$ variables and 8 X_i variables. An optimal solver like Conlloovia would solve the complete ILP problem with 224 variables.

The cost obtained from the solution of the partial ILP problem is a lower bound, since the three sets of restrictions of the partial ILP problem are necessary but not sufficient conditions for the complete ILP problem.

Phase 2. Node aggregation

Node aggregation consists in replacing a subset of nodes in the initial solution with a single node providing the same aggregated number of cores. The aggregation is limited to nodes of the same family and the same memory-to-core ratio, which ensures two important properties:

- The cost before and after aggregation will be the same, because the cost per core is the same.
- If a feasible allocation of containers to nodes exists before aggregation, there will also exist a feasible allocation of containers to nodes after aggregation.

To illustrate how the second property is guaranteed, consider the example shown in Fig. 5, in which two AC8 nodes plus one AC2 node are aggregated into a single AC18 node. This figure also shows a possible allocation before the aggregation, in which numbers on the left indicates the allocated mcores and the shaded area indicates free mcores. If the allocation before aggregation is feasible, a trivial feasible allocation exists after aggregation into one AC18 node, as shown in the figure.

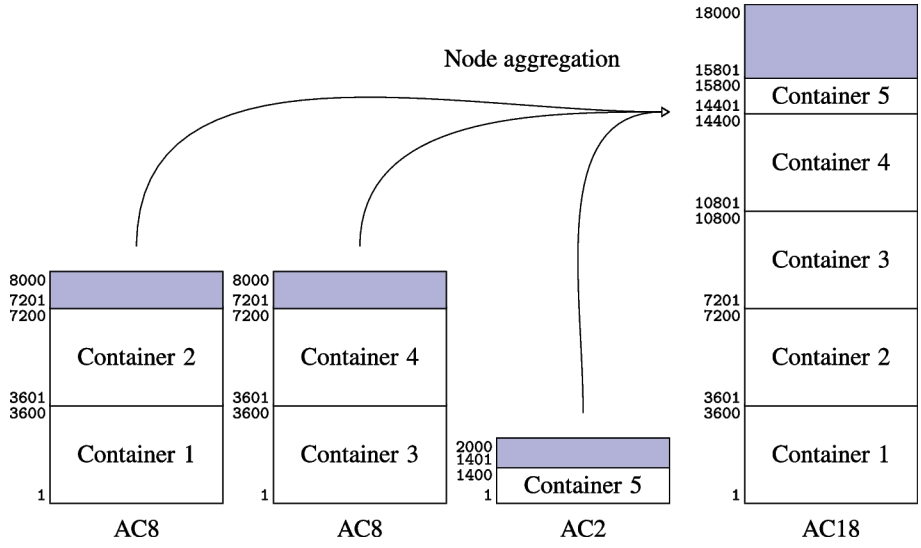

Fig. 5 Example of node aggregation

Node aggregation does not compromise the feasibility of the solution, does not increase its cost, and can provide important benefits in the allocation phase. As seen previously, after completing the first FCMA phase on the example introduced in "Illustrative example" section, the solution includes four AC1 nodes that should allocate six minimum-size containers for application A_1 , but only four of them can be allocated. If the four AC1 nodes were replaced by one AC4 node, the cost would not change, but now the six A_1 containers can be allocated. This node aggregation technique facilitates the allocation of containers to nodes, avoiding or reducing cost increases that may arise later during the allocation phase.

The aggregation of AC1 nodes into AC4 nodes in the previous example was straightforward. However, in many cases, aggregation is more complex due to the numerous possible aggregations. The goal is to find the aggregation that results in the minimum number of final nodes.

This section provides a systematic way to find that optimal aggregation in two steps. First, all possible aggregations of nodes that produce another node in the same family are found. These are called *aggregation sets* and the method for finding them is presented in "Aggregation sets" section. Next, groups of aggregation sets are used to create an ILP problem that minimizes the total number of final nodes for a given set of initial nodes, as explained in "Optimal node aggregation" section.

Aggregation sets

For the node aggregation phase, this research introduces the concept of the aggregation set. An aggregation set represents a possible aggregation of two or more nodes belonging to one or more instance classes of a family to obtain another node of a different instance class of the same family. The number of cores of the final node is the sum of the cores of the starting nodes. All the possible aggregation sets for a family can be represented by a graph. Figure 6 represents the possible aggregation sets for the instance class family A used in the problem example of Table 3. In the graph, nodes represent instance classes, identified with an index *i* and the number of cores, which is the only relevant parameter for the node aggregation phase, is indicated. Each arrow in the graph represents a participation of a node in an aggregation set and it is labeled with a tuple (*j*, *k*), where index *j* identifies the instance class for the node obtained after the aggregation and index *k* refers to one of the multiple aggregation sets that can yield that kind of node.

To better explain the graph, one node of instance class I_7, with 36-cores, can be obtained by aggregating smaller nodes in three different ways, represented by arrows of three different colors in the figure, labelled (7, 1), (7, 2) and (7, 3). Arrows with the same label and color are part of the same aggregation set. For example, the first aggregation set is represented by a single red arrow labelled (7, 1), and gives one 36-core node by aggregating two 18-core nodes. The arrow origin has a multiplier indicating the number of source nodes aggregated to obtain a destination node. The second aggregation set yielding one 36-core node corresponds to blue arrows labelled (7, 2). There are three (7, 2) blue arrows that aggregate one node with 4 cores, one node with 8 cores, and one node with 24 cores, to yield one 36-core node. The third aggregation set is represented by green arrows labelled (7, 3).

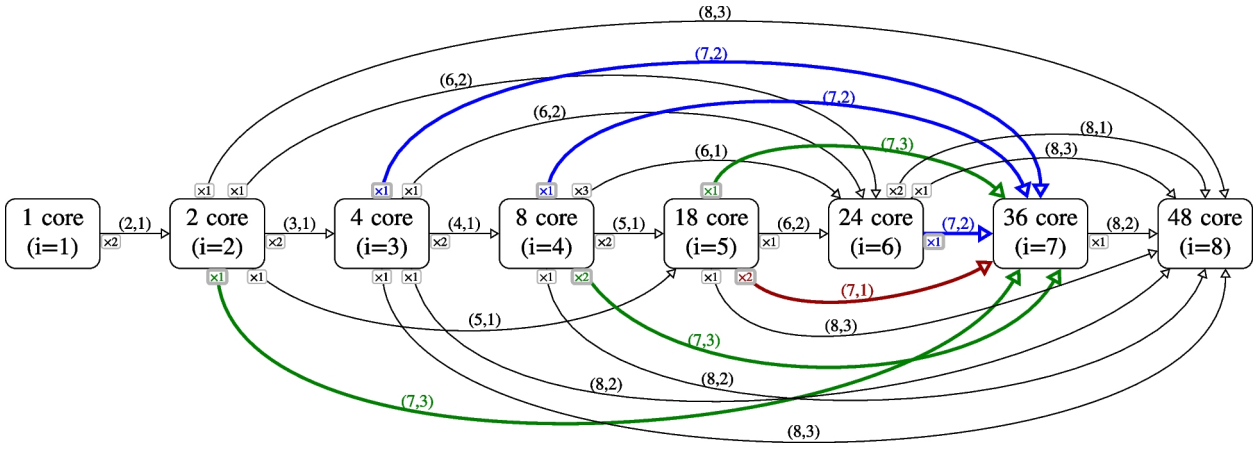

Fig. 6 Aggregation sets for the instance class families in the example

The graph shown in Fig. 6 is not trivial to obtain, and therefore an algorithm is needed. Moreover, it is convenient to reduce the number of aggregation sets as much as possible, to help reduce the size of the ILP problem to be solved in "Optimal node aggregation" section. In general, to find all the aggregation sets that can produce an instance class I_j, all possible sums of cores from smaller instance classes that result in the number of cores c_j of the final instance class need to be analyzed. This means finding multiplier values v_i which satisfy equation $\sum_{i<j} v_i \cdot c_i = c_j$. For example, for $j = 7$ (instance class AC36 in Fig. 6) instance classes AC1, AC2, AC4, AC8, AC18 and AC24 should be considered. The problem can be stated as finding vectors of multipliers $\vec{V} = (v_1, v_2, v_3, v_4, v_5, v_6)$ that satisfy the dot product condition $\vec{V} \cdot (1, 2, 4, 8, 18, 24) = 36$, where (1, 2, 4, 8, 18, 24) is the vector of cores c_i of the smaller instance classes $i < j$. The resulting equation is a linear Diophantine equation that can be solved by brute force or using ILP.

In the context of this paper, where many of the coefficients of the equation are multiples of each other, a brute force algorithm with a limited number of combinations proves efficient. The combinations to check can be reduced by considering the coefficient values and the equivalence of some solutions.

For example, the maximum value of v_1 to be analyzed is the quotient of the integer division $\lfloor 36/1 \rfloor = 36$. Using the same reasoning with the rest of the multipliers, the number of multiplier vectors to analyze is $\lfloor 36/1 \rfloor \times \lfloor 36/2 \rfloor \times \lfloor 36/4 \rfloor \times \lfloor 36/8 \rfloor \times \lfloor 36/18 \rfloor \times \lfloor 36/24 \rfloor = 46656$.

This number can be further reduced by considering that the node aggregation process will consist of a sequence of chained aggregations that ultimately produce a final number of nodes. If several sequences of aggregations produce the same final nodes, it is sufficient to consider only one of them. Let us consider two instance classes I_i and I_k that participate in the aggregation to obtain nodes of instance class I_j, such that c_k is multiple of c_i, with multiplicity m, i.e. $c_k = m \cdot c_i$. In this case, trying values up to $(m-1)$ for multiplier v_i does not restrict the solutions to the node aggregation problem. The result of any aggregation set giving a node of instance class I_j with $v_i = n \geq m$ would be the same as that obtained by replacing n by the reminder of the integer division $(n \mod m)$, and incrementing v_k in the quotient of that division, $\lfloor n/m \rfloor$. This transformation introduces a previous additional aggregation of the n nodes of class I_i into $\lfloor n/m \rfloor$ nodes of class I_k, leaving a remainder of $(n \mod m)$ nodes of I_i. The $\lfloor n/m \rfloor$ nodes of class I_k coming from the previous additional aggregation are added to the initial I_k nodes. Since the remainder cannot be greater than $(m-1)$, the maximum value to try for v_i is $(m-1)$.

For example, instance classes I_2 and I_3 in Fig. 6 have $c_i = 2$ and $c_k = 4$ cores, respectively, which are multiples of each other, $m = c_k/c_i = 2$. Thus, the maximum value to try for v_2 is $(2-1) = 1$, so only two values (0,1) need to be checked for v_2. For a higher value, such as $v_2 = 3$ it is possible to perform a previous aggregation of two I_2 nodes to obtain one I_3 node, leaving one node I_2. The resulting aggregated I_3 node will be added to other I_3 nodes and will participate in next aggregations.

Therefore, multipliers v_1, v_2, v_3, v_4, v_5, v_6 to obtain AC36 nodes by aggregation can be restricted to (0,1), (0,1), (0,1), (0,1,2), (0, 1, 2) and (0, 1), respectively, which gives a total of only 144 combinations to check in the Diophantine equation. Of these, only three fulfill equation $\sum_{i<7} v_i \cdot c_i = 36$, which are the aggregation sets (7, 1), (7, 2) and (7, 3) in Fig. 6.

Aggregation sets could be calculated in advance for each instance class family so that its calculation does not increase the time required to perform node aggregations.

Optimal node aggregation

Once the possible aggregation sets have been obtained, the next problem is to calculate the number of times each aggregation set is applied to transform an input set of nodes $\{X_i\}$, X_i being the number of nodes of instance class I_i, into an output set of nodes $\{X'_i\}$. The objective of the transformation is to minimize the total number of nodes in $\{X'_i\}$.

For example, for the aggregation sets shown in Fig. 6, $\{X_i\}$ ranges from $i = 1$ to 8, because there are 8 instance classes being considered. The previous analysis found 12 aggregation sets in that family, denoted by labels (2,1), (3,1), (4,1), (5,1), (6,1), (6,2), (7,1), (7,2), (7,3), (8,1), (8,2), and (8,3). Let us consider the following input set of nodes: 12 AC2 nodes, 3 AC8 nodes and 8 AC18 nodes. This means that $\{X_i\} = \{0, 12, 0, 3, 8, 0, 0, 0\}$. The problem to solve is to determine the output set of nodes $\{X'_i\}$ following the aggregation sets previously listed.

To formalize the problem, an integer $n_{j,k}$ must be associated with each aggregation set in Fig. 6. This integer is unknown in advance. After the problem is solved, it will represent the number of times the aggregation set (*j*, *k*) is applied, leading to a node of instance class I_j. In addition, each aggregation set (*j*, *k*) has a collection of multipliers $v_{j,k,i}$ for $i = 1, \ldots, j-1$ that provide the number of nodes of smaller instance classes I_i that participate in the aggregation set (*j*, *k*).

The number of nodes before and after the aggregation are related by the following expression:

$$X'_i = X_i + \sum_{k=1}^{P_i} n_{i,k} - \sum_{j=i+1}^{M} \sum_{k=1}^{P_j} n_{j,k} \cdot v_{j,k,i} \qquad (5)$$

where P_i and P_j are the number of aggregation sets that yield nodes of instance classes I_i and I_j, respectively. In addition, the *n* variables are the unknowns in the equations. The formula simply states that the total number of nodes of instance class I_i after the aggregation, X'_i, is equal to the sum of the nodes before the aggregation, X_i, plus the number of nodes that are aggregated into nodes of instance class I_i, minus the number of nodes of instance class I_i that are aggregated into nodes of larger instance classes.

While $n_{i,j}$ are the unknowns in this problem, the values of X_i are given inputs, and the values of $v_{j,k,i}$ are known from the aggregation sets found previously.

Applying Eq. 5 to the aggregation graph of Fig. 6 results in:

$$\begin{aligned}
X'_1 &= X_1 - 2n_{2,1} \\
X'_2 &= X_2 + n_{2,1} - 2n_{3,1} - 1n_{5,1} - 1n_{6,2} - 1n_{7,3} - 1n_{8,3} \\
X'_3 &= X_3 + n_{3,1} - 2n_{4,1} - 1n_{6,2} - 1n_{7,2} - 1n_{8,2} - 1n_{8,3} \\
X'_4 &= X_4 + n_{4,1} - 2n_{5,1} - 3n_{6,1} - 1n_{7,2} - 2n_{7,3} - 1n_{8,2} \\
X'_5 &= X_5 + n_{5,1} - 1n_{6,2} - 2n_{7,1} - 1n_{7,3} - 1n_{8,3} \\
X'_6 &= X_6 + n_{6_1} + n_{6,2} - 1n_{7,2} - 2n_{8,1} - 1n_{8,3} \\
X'_7 &= X_7 + n_{7,1} + n_{7,2} + n_{7,3} - 1n_{8,2} \\
X'_8 &= X_8 + n_{8,1} + n_{8,2} + n_{8,3}
\end{aligned} \qquad (6)$$

The objective is to minimize the number of nodes after the aggregation, i.e., minimize $\sum X'_i$. As the values X_i are constants, the problem becomes:

$$\begin{aligned}
\text{Minimize } (&-n_{2,1} - n_{3,1} - n_{4,1} - 2n_{5,1} - 2n_{6,1} - 2n_{6,2} + \\
&- n_{7,1} - 2n_{7,2} - 3n_{7,3} - n_{8,1} - 2n_{8,2} - 3n_{8,3})
\end{aligned}$$

Generally, this can be expressed as:

$$\text{Minimize} \quad \sum_{i=2}^{M} \sum_{k=1}^{P_i} \left(1 - \sum_{j=1}^{i-1} v_{i,k,j}\right) n_{i,k} \qquad (7)$$

The restrictions of the ILP problems are $X'_i \geq 0$ for $i \in [1, M-1]$, i.e, in general:

$$X_i + \sum_{k=1}^{P_i} n_{i,k} - \sum_{j>i}^{M} \sum_{k=1}^{P_j} n_{j,k} \cdot v_{j,k.i} \geq 0 \quad \forall i \in [1, M-1] \qquad (8)$$

The ILP problem has as many natural variables $n_{i,k}$ as aggregation sets, 12 for the aggregation sets of Fig. 6. The number of restrictions is equal to the number of instance classes minus 1, i.e., 7 in the example. The time required to solve an ILP problem of that size is negligible. The problem is so small because the number of cores in the different instance classes are often multiples of each other. In the worst-case scenario, where these numbers are co-prime, the number of variables and restrictions are significantly larger. Conversely, if the number of cores in the instance classes were powers of 2, the solution is trivial.

Solving this ILP problem for the input set of nodes $\{X_i\} = \{$ 0, 12, 0, 3, 8, 0, 0, 0 $\}$, previously presented, the values for $n_{i,j}$ in the optimal solution are: $n_{2,1} = 0$, $n_{3,1} = 6$, $n_{4,1} = 1$, $n_{5,1} = 0$, $n_{6,1} = 0$, $n_{6,2} = 0$, $n_{7,1} = 4$, $n_{7,2} = 0, n_{7,3} = 0, n_{8,1} = 0, n_{8,2} = 4$ and $n_{8,3} = 0$. Moving these values to Eq. (6) results in the output set of nodes $\{X'_i\} = \{0, 0, 0, 0, 0, 0, 0, 4\}$. Therefore, after the aggregation, the initial 23 nodes are transformed into four AC48 nodes with the same number of total cores, performance and price. Furthermore, if a feasible container-to-node allocation (disregarding memory requirements) exists for the initial set nodes, there will also be a feasible allocation for the final set of nodes.

Phase 3. Container allocation

```
Input: nodes, apps
sort nodes by cores (ascending)
sort apps by minimum-size container cores (descending)
foreach app ∈ apps do
    app_num_c ← number of app minimum-size containers
    app_sfmpl_max_c ← compute max number of app containers from SFMPL
    first_attempt_nodes ← nodes
    second_attempt_nodes ← [ ]
    third_attempt_nodes ← [ ]
    // First attempt: meet SFMPL requirement while optimizing memory
    foreach node ∈ first_attempt_nodes do
        allocatable_c ← compute number of containers that fit in node
        if allocatable_c ≥ app_sfmpl_max_c then
            to_allocate ← min(app_sfmpl_max_c, app_num_c) allocate(node,
              app, to_allocate)
            app_num_c ← app_num_c − to_allocate
            third_attempt_nodes.append(node)
        else
            second_attempt_nodes.append(node)
        if app_num_c = 0 then
            go to the next app
    // Second attempt: meet SFMPL requirement without optimizing memory
    sort second_attempt_nodes by allocatable containers (descending)
    foreach node ∈ second_attempt_nodes do
        app_num_c ← app_num_c − allocate(node, app, app_num_c)
        if app_num_c = 0 then
            go to the next app
    // Third attempt: allocate without SFMPL restrictions
    foreach node ∈ third_attempt_nodes do
        app_num_c ← app_num_c − allocate(node, app, app_num_c)
        if app_num_c = 0 then
            go to the next app
    // Fourth attempt: promote current nodes to get extra capacity
    while there are feasible promotions do
        node ← find cheapest promoted node that allocates 1 container
        app_num_c = app_num_c − allocate(node, app, app_num_c)
        if app_num_c = 0 then
            go to next app
    // Fifth attempt: No promotion is possible, so add new nodes
    new_nodes ← find cheapest added nodes to allocate app_num_c containers
    foreach node ∈ new_nodes do
        app_num_c ← app_num_c − allocate(node, app, app_num_c)
```

Algorithm 1. Allocation algorithm

After applying the first two phases of FCMA to the problem, the result is a set of nodes of various instance classes and containers to be allocated for each instance class family. In Phase 3 containers are allocated to nodes. This phase will also take into account the memory requirements of the containers and the memory capacity of the nodes, which previous phases did not consider. In addition, the allocation algorithm attempts to provide some degree of fault tolerance to the system, whenever this does not increment the cost of the solution. Algorithm 1 provides a high-level description of the process for allocating application containers to nodes within a single instance class family.

To achieve these goals, a modified First Fit Decreasing algorithm (FFD) is used. Containers within each instance class family are first sorted by decreasing size (number of mcores), while the nodes in the solution are sorted by increasing number of mcores. Then they are allocated to the first node in which they fit. The concept of fitting must be carefully defined to account for CPU and memory requirements, as well as fault tolerance. The memory checking is explained in "Memory checking" section. The fault tolerance is controlled by a parameter called SFMPL, which is defined in "Fault tolerance" section. It may be impossible to allocate some containers to any of the nodes. In this case, some nodes must be "promoted" to a higher capacity, or new nodes added to the system, following the procedure explained in "Node promotion and addition" section. These node promotions and additions will increase the cost. To address all these requirements, containers are allocated to nodes using up to five attempts in the following order:

1. First attempt. Allocate the maximum number of containers to nodes in compliance with the application's SFMPL fault tolerance parameter.
2. Second attempt. Allocate as many containers as possible to the remaining nodes, i.e., those that did not receive the maximum number of containers. Allocating fewer application containers per node increases memory usage per container (after future container aggregation).
3. Third attempt. Allocate the remaining containers to the nodes that initially received the maximum number of containers. Although the fault tolerance parameter is no longer met during this attempt, the system cost does not increase.
4. Fourth attempt. Promote nodes to increase their capacity. This promotion increases both CPU and memory capacities, but also the cost. If the nodes belong to the highest-capacity instance class within the family, further promotions are not possible, leading to the fifth and last attempt.
5. Fifth attempt. The only remaining solution is to add new nodes to the system, increasing the cost. This attempt is always successful, as there is at least one instance that is capable of accommodating a single application container.

Function `allocate()` is called several times in the algorithm. It receives three parameters: the node in which one or more containers will be allocated, the application to which the containers belong, and the maximum number of minimal-size containers for that application to allocate. It tries to allocate as much containers as possible, up to the number received as third parameter, and returns the number of containers that are successfully allocated. CPU and memory parameters are checked to perform this operation.

"Memory checking" section explains how memory requirements are checked, including the influence on memory requirements of potential container aggregations to be performed in the fourth FCMA phase. "Fault tolerance" section describes how the algorithm is modified to improve application fault tolerance without increasing system costs. "Node promotion and addition" section explains how node promotion and addition are used to solve allocation problems when no node has enough capacity to allocate a container.

Memory checking

The algorithm needs to check at several points whether a container fits into a node, in particular when computing the value for `allocatable_c` and when calling the `allocate()` method.

To decide if a container fits into a node, the algorithm must check if the free CPU and free memory of the node is sufficient to hold the container. For the CPU, this is simply a matter of comparing the free cores of the node with the cores required by the container. However, in the case of memory, this check may be more complex, because of the possibility of container aggregation.

When a node holds multiple containers of the same application, these containers could be aggregated to form larger containers, whenever the resulting performance is not compromised. Although this aggregation is not performed in this phase, it must be considered when checking memory requirements, because usually the amount of memory required by the aggregated container is less than the sum of the memory requirements of the individual containers. For example, two containers of an application with 400 mcores and 100 MB each could be

aggregated to form one container with 800 mcores and an amount of memory lower than the sum, 200 MB. This aggregation should provide the final container with performance not inferior to the sum of performances of the two initial containers. One of the beneficial effects of this aggregation is the reduction in container memory requirements.

The added complexity of container aggregation during the allocation process comes from considering the containers of the same application previously allocated, in order to determine the memory requirements. For example, if there were four allocated containers of an application and one new container of that application to be allocated, it would be necessary to evaluate the memory requirements associated with the aggregation of five containers of that application. This evaluation is carried out by simulating the aggregations performed by the aggregation algorithm, shown in "Phase 4. Container aggregation" section. Container aggregation does not affect the FFD algorithm, it simply establishes the conditions under which a container fits into a node.

Memory checking can be disregarded when container memory-to-core ratios are low relative to node memory-to-core ratios. More formally, if the memory-to-core ratio of all the containers to be allocated to a set of nodes within a family is lower than the memory-to-cores ratio of all the nodes of the family, memory checking can be ignored because memory requirements will always be met as long as CPU requirements are met.

Fault tolerance

A parameter called Single Failure Maximum Performance Loss (SFMPL) must be defined by the analyst for each application, indicating the expected maximum performance loss caused by the failure of one node. For example, an SFMPL value of 0.25 indicates that the failure of one node causes a maximum performance loss of 25%. The SFMPL value imposes a maximum number of containers for each application within a node. Let A_a be an application with a requirement of w_a rps that is allocated to a node of instance class I_i using minimum-size containers of performance $r_{0,a,i}$. The maximum number of containers in nodes of that instance class is given by

$$\text{Max. containers}_{a,i} = \lfloor \text{SFMPL}_a \cdot w_a / r_{0,a,i} \rfloor \quad (9)$$

which corresponds to variable `app_sfmpl_max_c` in Algorithm 1.

The allocation algorithm in the first three attempts allocates groups of containers from the same application. In the first attempt, groups containing the maximum number of containers, as calculated in Eq. 9, are allocated to nodes. Allocating the maximum number of containers ensures the fulfillment of the application fault tolerance restriction, SFMPL, while reducing container memory requirements after container aggregation.

The second attempt works with nodes that were unable to allocate the maximum number of containers. On each of these nodes, as many containers as possible are allocated. Fault tolerance restrictions continue to be met, but memory savings are lower.

When the first two attempts fail to allocate the application containers, continuing to fulfill the fault tolerance restriction would require increasing the system's computational capabilities, which would increase the cost. Since minimizing cost takes precedence over fault tolerance, containers are allocated to the nodes from the first attempt, and the application will no longer meet the fault tolerance restriction.

Node promotion and addition

Node promotion is an operation that modifies the capacity of a node in the system so that it can allocate at least one container that would otherwise be impossible to allocate in any node. This impossibility arises from an insufficient amount of CPU and/or memory in any node of the system, but not from SFMPL limits, as previously explained. The promotion algorithm is simple: one of the nodes must be replaced with another of higher capacity in the same family, and among all possibilities, the one with the lowest cost increase should be chosen.

For example, if the system has two nodes: one AC2 with 0.5 free cores and one AC8 with 0.4 free cores. A container that requires one core and low memory cannot fit. The cheapest promotion for AC2 is upgrading it to AC4, and for AC8, upgrading it to AC18. The first option adds 2 cores, and the second one adds 10. Since A instances cost is proportional to the number of cores, as shown in Table 3, the first promotion is the cheapest option.

If all nodes in the system are of the largest instance class within the family, e.g. AC48, no promotion is possible, and the only viable option would be the addition of new nodes, using the following procedure. Instance classes in the family are sorted by decreasing number of containers they can allocate, removing those that can allocate the same number of containers at a higher cost.

Starting with the first instance class, the number of nodes for the instance class is calculated as the integer quotient of the number of remaining containers divided by the number of containers it can allocate. This number of nodes is added to the system. The procedure is then repeated with the reduced number of remaining containers and instance classes.

Note that node addition is always possible. For example, instead of promoting one AC2 node to AC4, it is also possible to add another AC2 node. However, node addition will be used only when no promotion is possible because node addition usually adds small nodes to the system, which can fragment the available capacity across many nodes. This fragmentation makes it harder to allocate the remaining containers, thus increasing the cost.

Once the allocation of containers to nodes is completed, a final optimization is implemented, which can revert the last promotion performed in all the nodes, if a cheaper solution based on node addition is found. To do this, the following procedure is followed. Firstly, the state of the current solution is saved, including its cost. Secondly, the last promotion performed in each node is undone, and as a result some containers become unallocated. Next, a new problem is formulated to allocate these containers to nodes, and solved again using phases 1, 2 and 3, but without this last optimization step. Finally, if the cost of allocating the remaining containers is higher than the cost of the promotion, the saved solution state is recovered.

The constraints of the partial ILP problem presented in "Mathematical formulation" section are less restrictive than those of the complete ILP problem. Therefore, the solution provided by the partial ILP problem is optimal when neither promotions nor node additions occur. This is the case in the example, in which the output of phase 3 is shown in Fig. 4d.

Phase 4. Container aggregation

This is the fourth and final phase of the algorithm, in which the minimum-size containers of an application allocated to a node can be replaced by containers of larger size in terms of cores, reducing the number of containers as long as performance is not affected.

The requirement to preserve the performance of the application deployment imposes limits to the aggregation of its minimum-size containers. For example, a single-process and single-thread application cannot be aggregated beyond one core, because the performance of the final container after aggregation would be lower than the sum of the performance of the initial containers. For multi-threaded applications, the limit is determined by the number of threads beyond which performance is no longer proportional to the number of threads. Furthermore, it is not advisable to carry out aggregations that yield too many different container types. The reason is that the greater the number of container types, the more difficult it would be to recycle containers between successive scheduling windows, and as a result, cluster deployment costs would increase.

Given that the aggregation algorithms lack detailed knowledge about internal application implementations and how performance degrades with aggregation, and taking into account the restriction of not generating a large number of different container types, the analyst must provide specific aggregation levels for each application and instance class family that cannot be exceeded.

For instance, if a minimum-size container with 150 mcores and 2 rps in a family, aggregation levels of 2, 6, and 12 were established by the analyst, the following aggregations would be permissible:

- 2 containers can be aggregated to form 1 container with 300 mcores and 4 rps.
- 6 containers can be aggregated to form 1 container with 900 mcores and 12 rps.
- 12 containers can be aggregated to form 1 container with 1800 mcores and 24 rps. Note that this case implies that the application internally uses a multi-threaded implementation, to maintain proportionality between mcores and performance.

To solve this phase, all containers of an application that are allocated to the same node are considered for aggregation, using the different aggregation levels provided by the analyst. Of all the possible aggregations, those that minimize the number of final containers are selected.

For example, if a node contains 50 identical minimum-size containers, using aggregation levels 2, 6 and 12, several aggregations are possible, such as using aggregation level 12 four times and aggregation level 2 once, or using aggregation level 2 twenty-five times. The first option is preferable because it results in 5 containers, while the second results in 25.

An ILP problem could be formulated to optimally solve this phase. However, container aggregation is performed many times as part of the FFD algorithm used in phase 3, so a faster approach is preferred, shown in Algorithm 2.

```
Input: num: number of identical minimum-size containers; aggs: list of
       aggregation levels; container: minimum-size container properties
Output: List of aggregated containers

agregated_containers ← [ ]
sort aggs (descending)
for agg in aggs do
    num_aggregated ← ⌊num/agg⌋
    num ← num mod agg
    if num_aggregated > 0 then
        new_container ← container with (container.cores × agg) cores, memory
          ≤ (container.mem × agg) and performace (container.perf × agg)
        agregated_containers.append(new_container)
agregated_containers.append(num minimum-size containers)
return agregated_containers
```

Algorithm 2. Aggregate app containers

The algorithm relies on using the maximum level of aggregation as many times as possible, and with the remaining containers, repeating the process at the next level of aggregation. This process continues until all aggregation levels have been tried. Applying this idea to the previous example of 50 minimum-size containers, with 150 mcores each, and assuming aggregation levels 2, 6 and 12, at aggregation level 12 the algorithm will obtain $\lfloor 50/12 \rfloor = 4$ containers of 1800 mcores. With the $50 - 4 \cdot 12 = 2$ remaining containers the same procedure is used with the next aggregation level, i.e, 6. In this case $\lfloor 2/6 \rfloor = 0$, which means this aggregation level is not used. Finally, with the remaining 2 containers and the next aggregation level, i.e. 2, $\lfloor 2/2 \rfloor = 1$ container of 300 mcores is obtained. If there were remaining containers after the division by the lowest aggregation level, those containers could not be aggregated.

In addition to memory savings, container aggregation offers the following advantages:

- It reduces interference between containers. When a large number of containers run on a single machine, their performance deviates from the ideal. By aggregating containers, interference is reduced leading to better overall performance.
- It reduces load-balancing costs: A large number of containers within a node implies higher computational costs for distributing the load among the different containers.

FCMA computational complexity

This section theoretically analyses the relationship between the FCMA execution time and the problem parameters. FCMA consists of four phases, as described in "Phase 1. Partial ILP problem", "Phase 2. Node aggregation", "Phase 2. Node aggregation" and "Phase 4. Container aggregation" sections, which are graphically represented in Fig. 3. Next, the computational complexity of each phase is analysed.

1. Partial ILP Problem ("Phase 1. Partial ILP problem" section). It is well known that ILP is an NP-complete problem ([35], p.245) and therefore generally has exponential complexity in the number of integer variables, though it is polynomial in other parameters such as the number of constraints and the size of the coefficients [36]. In our case, the integer variables are the number of nodes of each instance class X_i and the number of minimum-size containers for each application-instance class pair $Y_{0,a,i}$. The number of instance classes is denoted by M and the number of applications by N, making the number of integer variables $M + N \cdot M$.
   While the worst-case complexity remains exponential in $N \cdot M$, efficient algorithms of type Branch and Cut often achieve average-case or best-case runtimes that are orders of magnitude lower ([37], p.364). This explains why this FCMA phase can obtain solutions in reasonable time even for large problems, as shown in the Experimental results section. Moreover, FCMA operates with a model that drastically reduces the number variables in the ILP problem compared to baseline approaches like Conlloovia, making FCMA much faster. In particular:

   - The ILP problem in the first FCMA phase involves a few integer variables, X_i, in comparison to the large number of equivalent boolean variables used in Conlloovia.

- FCMA further reduces the number of X_i variables through simplifications based on the concept of instance class families.
- The number of variables in Conlloovia's ILP problem increases with the workload, whereas in FCMA the number of variables remains independent of the workload.
- The container aggregation concept introduced in FCMA allows for the use of a single container per application and instance class in the ILP problem, reducing the number of $Y_{0,a,i}$ variables.

2. Node aggregation ("Phase 2. Node aggregation" section). Node aggregation is solved as one ILP problem on each instance class family. The number of variables is the number of node aggregation sets, *S*, whereas the number of constraints is the number of instance classes in the family, *I*. Worst-case complexity is again exponential, but considering the usual values of *S* and *I*, typically in the order of 10, as shown in the example of "Aggregation sets" section, the execution time of this phase can be disregarded in practice.
3. Container allocation ("Phase 3. Container allocation" section). The inputs of this phase differ from those of the initial problem. In the initial problem, we had *N* different applications and *M* different instance classes. However, after phase 1 and 2 are completed, the result is a set of *C* different minimum-size containers and a set of *V* virtual machines (nodes). The number *C* of different containers can be related to the initial problem parameters through the number of applications *N*, the workload of each application, and the performance of the minimum-size containers. If we denote by *W* the maximum workload among applications and by *R* the minimum performance among minimum-size containers, then $C \approx N \cdot W/R$.
   The number *V* of virtual machines (nodes) depends on the capacity in cores of the nodes and the cores required by all the minimum-size containers. If we denote by *K* the quotient of the average capacity in cores of the nodes and the average number of cores required by the minimum-size containers, then $V \approx C/K$.
   Phase 3 consists basically of two nested loops, as presented in Algorithm 1. The outer loop iterates over applications, and for each one, to allocate all of its containers, it is necessary to loop over nodes. Considering only the first, second and third attempts, the worst-case complexity of phase 3 is $O(C \cdot V)$. Fourth and fifth attempts require iterating again over the nodes, but also on the *M* instance classes to get additional capacity to allocate the remaining containers, so it increases the complexity to $O(C \cdot V \cdot M)$. However, note that attempts 4 and 5 are often unnecessary, and in that case the complexity reduces to $O(C \cdot V)$.
4. Container aggregation ("Phase 4. Container aggregation" section). All the *V* nodes of the solution are traversed, and for each node, a sequence of divisions is performed, one per aggregation level, to obtain the optimal aggregations. Denoting by *A* the number of aggregation levels, the worst-case complexity of this phase is $O(A \cdot V)$, which can be ignored when it is compared to the worst-case complexity of the allocation phase, since *A* is usually very small.

Although worst-case complexity is interesting from a theoretical perspective, in practice, the worst-case analysis is extremely pessimistic. "Experimental results" section includes an experimental study on FCMA behaviour, summarized in Fig. 10, which depicts FCMA computation time as a function of $(M \cdot N)$ and the product of the number of containers and nodes, $(V \cdot C)$.

Experimental results

A series of experiments was conducted to assess the effectiveness of FCMA. The evaluation includes comparisons with three baseline techniques. First, Conlloovia [7], a state-of-the-art method that solves the same problem as FCMA using linear programming, providing a relevant point of comparison. Other techniques based on linear programming are not directly comparable, as they address different problem formulations. The two other baselines are heuristic algorithms based on the First Fit strategy: First Fit by Core (FFC), which prioritizes instance classes with more cores and allocates containers to VMs from the largest container class until the workload is satisfied, and First Fit by Price (FFP), which follows the same allocation process but prioritizes instance classes by ascending price to minimize cost.

The hypotheses to test are that FCMA:

- Finds solutions faster than Conlloovia while maintaining comparable times to the heuristics FFC and FFP.
- Successfully finds solutions in scenarios where Conlloovia fails due to problem size.
- Achieves solutions with costs comparable to those of Conlloovia when Conlloovia identifies the optimal solution.
- Achieves solutions with costs better than those obtained by FFC and FFP.
- Outperforms Conlloovia, FFC and FFP with respect to the secondary objectives defined in "System model and objectives" section.

Table 4 Variables defining the different scenarios

Variable	Base values	Multipliers
Number of instance class families	1, 4	-
Number of applications	1, 2, 5, 15, 30	-
Minimum-size containers variables		
Number of cores	0.120, 3	0.75, 0.9, 1, 1.15, 1.2
Memory-to-core ratios	2, 8	0.75, 0.9, 1, 1.15, 1.2
Performance in rps	0.02, 0.4	0.04, 0.08, 0.16, 0.32, 0.4, 0.8, 1.1, 2.1, 4

- Works efficiently with large container clusters, maintaining competitive execution times as the problem size increases.

Experimental setup

A set of 80 synthetic scenarios was generated to represent a wide range of situations with varying problem sizes. Each scenario is defined by a combination of the base values shown in Table 4.

For parameters of instance class families (number of cores, memory and price), real values from an AWS region were used. The values where obtained from the on-demand EC2 pricing page[1]. The number of instance class families, as shown in Table 4, is 1 (c5 instance types from AWS) or 4 (c5, c6 g, c6i and c7a), including also in each family the extended memory instances types (e.g., m5 and r5 for c5). When one family is used, there are 32 instance classes, while when four families are used, there are 105 instance classes.

The number of applications may be any of the base values in the second row of Table 4.

In each scenario, the number of cores required by the minimum-size container of each application is chosen from the base values in the third row of Table 4. In order to use different number of cores for each application, one of the five multipliers shown in the last column of the third row of the table is randomly selected for each application and then applied to the base value. When the base number of cores is 0.120, the aggregation levels are established to 1, 2, 4, and 8, while when the base number of cores is 3, the aggregation levels are established to 1 and 2.

Memory-to-core ratios are used to simulate different memory requirements relative to the number of cores for each application. A memory multiplier is selected from the five values shown in the fourth row of Table 4 and then applied to the base value. The result is multiplied by the number of cores of the container to obtain a different memory value in GB for the container classes.

The problem size and the solution depend on the relation between container performance and application workloads. Therefore, the workload for each application is set to 1 rps to simplify the scenario specification by only varying the performance of the containers. Firstly, one of the two base performance in Table 4 is used in each scenario. Next, a random multiplier for each application performance from the 9 possibilities shown in the last row of the table is applied to the base value.

A Python package was developed and open-sourced to apply FCMA[2]. The code required to reproduce the experiments, including datasets and workloads generated following the description above, has also been open-sourced[3].

Each scenario was solved using FCMA and the baseline methods. All experiments were run on an Intel Core i7 -9700 K CPU at 3.60GHz with 64 GiB of RAM. COIN CBC 2.10.3 was used as the LP solver. CBC has two mechanisms for controlling when to abort the search for the optimal solution: one based on a relative optimality gap and another based on a time limit. The relative optimality gap is defined as the relative difference between the best feasible solution found and a bound of the optimal solution. This parameter was set to 0.02, i.e., the solver stops when it is at most 2% away from the bound of the optimal solution. The limit on the solving time was set to 600 seconds. If a solution within 2% of the bound is not found in this time, the solver returns the best feasible solution found, if available, along with the bound.

In the case of FCMA, parameter tuning is limited to the configuration of the ILP solver, as there are no additional algorithm-specific parameters. The value of the relative optimality gap was set to 0.02 based on experimentation. In a series of experiments, we observed that decreasing this value below 0.02 led to marginal quality gains at the expense of significantly longer computation times. Regarding the solving time limit, it was set to 600 seconds, as this is a reasonable value given that the scheduling window is expected to be much longer in most practical scenarios.

[1] https://aws.amazon.com/es/ec2/pricing/on-demand/

[2] https://github.com/asi-uniovi/fcma/

[3] https://github.com/asi-uniovi/fcma_exp/

The main metrics used to compare the four techniques are:

- Cost of the solution: measured as the ratio of the cost obtained by each technique to the cost obtained by FCMA.
- Total time to create and solve the problem: measured in seconds.
- Applicability: the number of scenarios in which each technique finds an optimal solution, a feasible solution, or fails to find a solution. The results for Conlloovia are obtained directly from the solver. For FCMA, FFC and FFP, which are heuristic techniques, a solution is considered optimal when its cost matches the cost found by Conlloovia and Conlloovia determined that it was optimal, or when the solution found is equal to the lower bound. A solution is considered feasible when optimality is not guaranteed, although it might be optimal.

As FCMA has four secondary goals, four additional metrics are defined:

- Fault tolerance: proportion of applications fulfilling the Single Failure Maximum Performance Loss (SFMPL) criteria. The SFMPL was set to 0.5 for all applications in all scenarios.
- Node recycling: to compute this metric, each scenario is solved twice, the second one with a 20% higher workload. The number of cores that are allocated to nodes of the same instance class in both allocations are counted and divided by the total number of cores. This metric represents the proportion of cores that are reused between two consecutive allocations.
- Load balancing reduction: calculated by determining the inverse of the number of nodes running each application. The load balancing metric is the average of these inverse values across all applications. This metric is inversely related to the load-balancing overload among nodes.
- Container isolation: calculated by first determining the inverse of the number of containers allocated to each node. The container isolation metric is the average of these inverse values across all nodes. This metric is inversely related to container interference.

The metrics were designed to be easily interpreted: in all cases, higher values are better, 0 is the worst value and 1, the best. Although Conlloovia, FFC and FFP do not take these metrics into account in the optimization process, they can be calculated for the solutions they obtain.

Results

Figure 7 shows the relative solution costs of Conlloovia, FFC, and FFP with respect to FCMA, plotted on a logarithmic scale (Y-axis) against the problem size (X-axis), also in logarithmic scale. The problem size corresponds to the number of variables in the Conlloovia ILP formulation, comprising Boolean variables to represent potential VM allocations and integer variables to specify the number of containers for each container class within each VM. The number of variables scales with several factors, including the number of applications, instance classes, container classes, and the performance ratio of each container class within an instance class relative to the workload managed by the system. The precise formulation is detailed in [7].

In Fig. 7, a cost ratio of 1 (i.e., 10^0 in the figure) indicates that both the technique used in the ratio and FCMA have found solutions with the same cost. When the ratio exceeds 1, the technique yields a solution that costs more than FCMA; conversely, when the ratio is below 1, the technique finds a solution that costs less than the

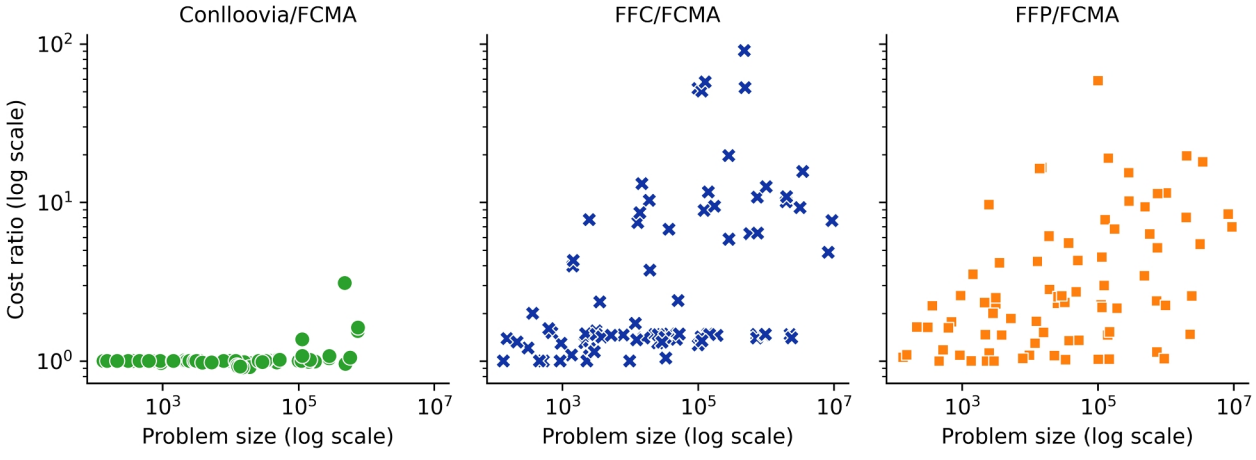

Fig. 7 Comparison of costs between the different techniques using the cost obtained by FCMA as baseline

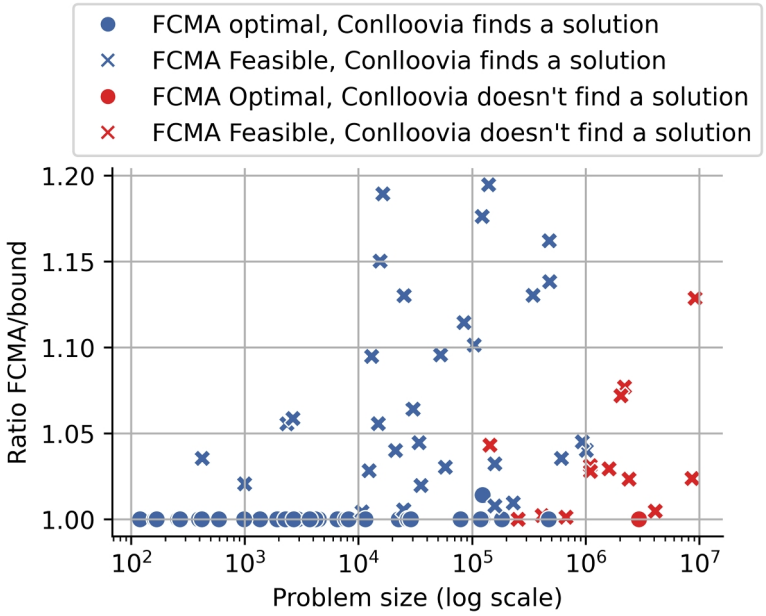

Fig. 8 Comparison of costs between FCMA and a lower bound

Table 5 Number of scenarios for each status

	FCMA	Conlloovia	FFC	FFP
Optimal	36 (45.0%)	37 (46.2%)	6 (7.5%)	5 (6.2%)
Feasible	44 (55.0%)	29 (36.2%)	74 (92.5%)	75 (93.8%)
No solution	0 (0.0%)	14 (17.5%)	0 (0.0%)	0 (0.0%)

solution found by FCMA. This latter scenario occurs only for Conlloovia, and the difference is less than 14% (i.e., a ratio greater than 0.86, which is not visible in the figure due to the scale). As the problem size increases, the cost ratio also rises, particularly for FFC and FFP, suggesting that these heuristics perform significantly worse than FCMA. This poor performance is especially pronounced in cases where the most expensive VM delivers lower performance, as these scenarios can mislead the heuristics. While such cases are not very common, they may arise, for instance, when a machine equipped with advanced hardware is allocated but the workload does not leverage the capabilities of that resource. For Conlloovia, the maximum observed ratio is 2.79, indicating that, in this case, the non-optimal solution found by Conlloovia is almost three times worse than that found by FCMA.

Furthermore, the figure shows no points beyond 10^6 on the X-axis for Conlloovia, indicating that for these large problem sizes, Conlloovia fails to find a solution, so the ratio cannot be plotted. In these cases, FCMA finds a solution and is significantly better than FFC and FFP.

To evaluate how far from the optimal solution FCMA can be, Fig. 8 shows the ratio of the cost obtained by FCMA to the bound of the optimal solution. This plot highlights in red the situations where the CBC solver for Conlloovia did not find a feasible solution. In the worst-case scenario, FCMA offers a solution that is approximately 20% worse than the bound, meaning it is no more than 20% worse than the optimal solution. This result is very relevant considering the complexity of the problem and the low solving time achieved by FCMA.

Table 5 summarizes the comparative applicability of the different techniques. In 14 scenarios (17.5%), FCMA found a feasible solution where Conlloovia could not. In addition, FCMA found the optimal solution much more frequently than FFC and FFP.

Figure 9 and Table 6 compare the solving times of the different techniques as the problem size increases, using logarithmic scales on both axes. As shown in the table, the mean time for FCMA is 0.88 seconds, in the same range as the mean times of FFC and FFP, and in contrast to the considerably higher 383.82 seconds required by Conlloovia. The maximum time for FCMA is 28.73 seconds, which occurs in a scenario with 30 applications, more than 500 nodes in the solution and a cost of approximately 2600 $/h in AWS. In this scenario, Conlloovia does not find a solution and the cost of the solutions found by FFC and FFP are about 3000 and 3500 $/h respectively. This highlights the substantial efficiency advantage of FCMA over Conlloovia in solving larger problems.

Regarding the secondary goals, Table 7 presents the average metric values for each technique, ranging from 0 (worst) to 1 (best), with standard deviations shown in parentheses. FCMA achieves equal or better fault tolerance compared to Conlloovia and FFC, although it is outperformed by FFP, which achieves higher fault tolerance by using more VMs, incurring additional costs. For

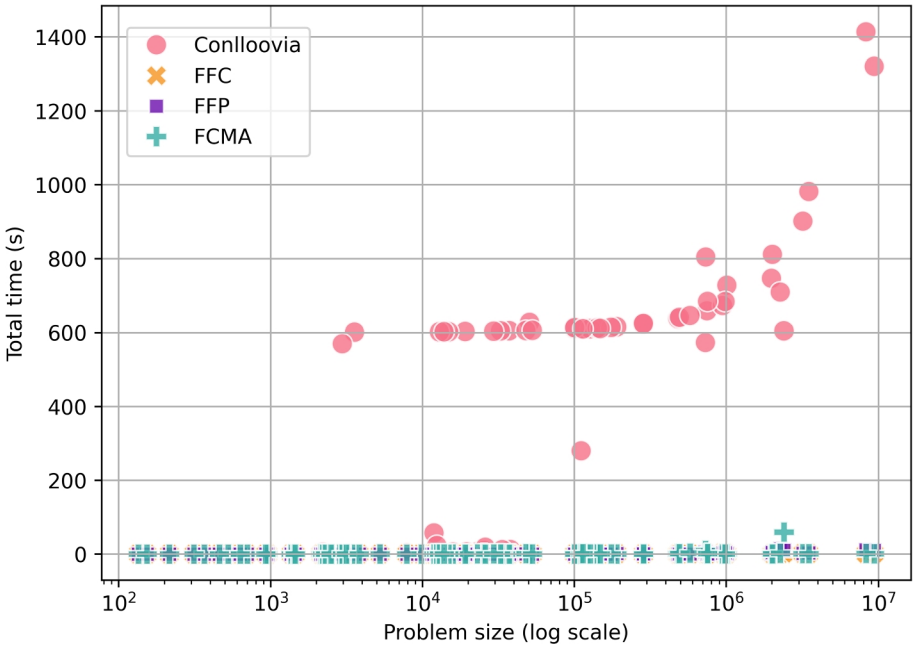

Fig. 9 Comparison of solving times

Table 6 Comparison of total times

Technique	mean (s)	max (s)
Conlloovia	383.82	1390.17
FFC	0.50	4.89
FFP	3.25	75.90
FCMA	0.88	28.73

Table 7 Summary of metrics related to secondary objectives for each technique

Metric	Conlloovia	FFC	FFP	FCMA
Container isolation	0.31 (0.24)	0.07 (0.12)	0.62 (0.40)	0.31 (0.25)
Node recycling	0.27 (0.39)	0.37 (0.46)	0.09 (0.29)	0.72 (0.34)
Load balancing reduction	0.38 (0.34)	0.49 (0.42)	0.09 (0.21)	0.58 (0.31)
Fault tolerance	0.25 (0.36)	0.26 (0.37)	0.81 (0.30)	0.42 (0.42)

Each cell shows the average value with the standard deviation in parentheses. All metrics range from 0 (worst) to 1 (best)

node recycling, FCMA significantly outperforms all other techniques. Similarly, the load balancing reduction metric indicates superior performance for FCMA. Finally, in terms of container isolation, FCMA surpasses Conlloovia and FFC, while FFP achieves the highest values by using more VMs, which incurs additional costs.

In addition to the 80 scenarios discussed above, an additional set of 1,000 scenarios was generated to evaluate how FCMA's solving time depends on the problem size. These 1,000 scenarios were designed to evaluate FCMA's scalability with a larger number of applications (up to 100). In each scenario, all other problem parameters were randomly generated to comprehensively sample the variables influencing the algorithm's performance. The instance classes used were taken from the four AWS families indicated in Table 4 and can reach up to 105 different instances. The process of generating these scenarios, along with the resulting dataset, is documented in the previously linked paper repository.

Figure 10 illustrates how the solving time increases with the two main sources of computational complexity identified in "FCMA computational complexity" section: $N \cdot M$, the number of applications multiplied by the number of instance classes, which affects the partial ILP problem; and $V \cdot C$, the number of minimum-size containers multiplied by the number of virtual machines (nodes), which impacts the container allocation phase. The size and colour of the points in the plot represent the total solving time across all phases.

As in the previous experiments, the ILP solver was configured with a 600-second time limit and a relative optimality gap of 0.02. Data points marked with an 'x' indicate cases where FCMA did not find a solution within this time. Note that the solving time includes all phases, not just the ILP problems.

Figure 10 shows that the main limiting factor for FCMA is $N \cdot M$. Even when $V \cdot C$ exceeds 10^4, FCMA still finds a solution as long as $N \cdot M$ remains below 1,500. Additionally, the figure clearly illustrates how solving time increases significantly with larger values of $N \cdot M$ (indicated by the larger marker sizes) and how FCMA is more likely to fail to find a solution (blue 'x' markers). This confirms that the bottleneck is the partial ILP problem rather than the container allocation phase,

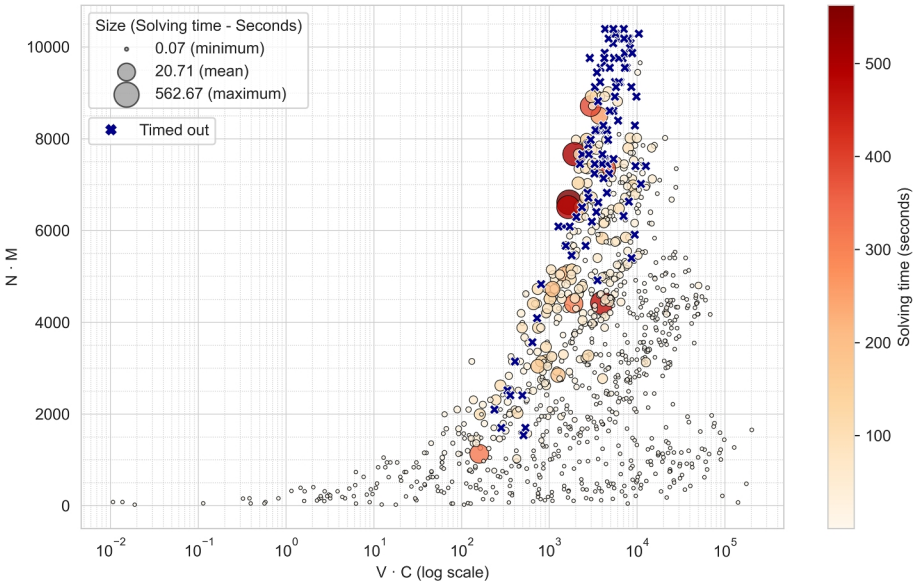

Fig. 10 Analysis of FCMA solving time depending on the problem size. The scatter plot illustrates the relationship between solving time (seconds, color-coded and point size scaled) and two key scaling factors: the product of the number of applications and instance classes ($N \cdot M$, Y-axis), and the product of the number of virtual machines and containers ($V \cdot C$, X-axis, log scale). Larger and darker points indicate longer solving times. 'x' markers denote scenarios that timed out after 600 seconds. The figure demonstrates the dominant influence of $N \cdot M$ on solving time

Table 8 FCMA Problem Solving Success Rate by $N \cdot M$ Range

$N \cdot M$	Solved (%)	Number of problems generated
[0, 1000)	100.00	217
[1000, 2000)	97.92	144
[2000, 3000)	96.23	106
[3000, 4000)	98.39	124
[4000, 5000)	97.86	140
[5000, 6000)	93.24	74
[6000, 7000)	79.03	62
[7000, 8000)	68.25	63
[8000, 9000)	57.14	28
[9000, 10000)	11.11	27
[10000, 11000)	6.67	15

This table shows the percentage of problems solved within 600 seconds and the total problems generated for different $N \cdot M$ ranges. Note the decreasing success rate with increasing $N \cdot M$

despite the former being implemented in C and the latter in Python, where performance improvements could be more easily achieved.

Table 8 presents the percentage of problems solved under 600 seconds across different ranges of $N \cdot M$ values. The first column specifies the range of $N \cdot M$, the second column reports the percentage of solved problems within each range, and the third column indicates the total number of problems generated in that range. The results show that as $N \cdot M$ increases, the success rate generally decreases, with nearly all problems being solved for $N \cdot M < 1,000$, while only a small fraction are solved for $N \cdot M > 9,000$.

In summary, FCMA provides an allocation method for the autoscaling problem that offers a better trade-off compared to the alternatives. It achieves similar costs to Conlloovia when Conlloovia can solve the problem and find the optimal solution, but with faster solving times. Additionally, FCMA yields lower costs than FFC and FFP without incurring longer solving times. Furthermore, it achieves equal or superior performance in secondary metrics, except for container isolation and fault tolerance in FFP, which incurs a higher cost. Lastly, it can scale to very large cluster containers, being limited mainly by the product of the number of applications and the number of possible instance classes to use.

Practical considerations

In this section, a comparison between FCMA and the autoscaling mechanism used in the industry standard Kubernetes is discussed. Additionally, a possible integration of an FCMA-based autoscaler into a Kubernetes cluster is considered.

FCMA is designed to be integrated in predictive autoscalers, which operate following the stages depicted in Fig. 1. However, Kubernetes implements a reactive autoscaler that triggers scaling operations when specified metrics reach predefined thresholds. Kubernetes primarily uses CPU utilization as the key metric to drive the autoscaling process.

The Kubernetes autoscaler aims to maintain cluster resources at reasonable levels to meet the quality-of-service requirements of cluster applications while avoiding

resource wastage. As a result, cluster costs can be kept under control. However, Kubernetes' autoscaling algorithms are not explicitly designed for cost optimization. Therefore, they presumably cannot manage cluster deployment costs as precisely as other cost-minimization algorithms, such as FCMA. Further research should be conducted to compare the cluster deployment costs generated by Kubernetes' autoscaling algoritms and FCMA.

In Kubernetes, autoscaling is implemented through two components: the Horizontal Pod Autoscaler (HPA) [38] and the Cluster Autoscaler (CA) [39]. The former is responsible for scaling containers within the nodes of a cluster, while the latter scales nodes out or in to adjust the cluster computing capacity to match container demand. The Horizontal Pod Autoscaler has a single, standard implementation across all Kubernetes clusters. However, the Cluster Autoscaler requires a specific implementation for each cloud provider, as it must leverage the provider's node autoscaling services. Custom implementations of the Cluster Autoscaler are available for the Kubernetes services of major cloud providers, including Amazon EKS, Google Kubernetes Engine, and Azure Kubernetes Service.

An FCMA-based autoescaler could be implemented and integrated into the Kubernetes service of any cloud provider. Each provider would require a specific implementation of this autoscaler, which would consist of a single component that utilizes the cloud provider's autoscaling services to scale nodes and the Kubernetes APIs to deploy or terminate containers as needed. It is important to highlight that before implementing FCMA, the autoscaling algorithm should be enhanced with an additional mechanism to manage transitions between successive scheduling windows. This mechanism would determine which nodes and containers should be maintained or removed between windows, and which new nodes and containers should be created. This challenge is currently being addressed as part of ongoing work.

Conclusions

In this work, we have developed FCMA, an algorithm that calculates an optimal or near-optimal resource allocation solution to the autoscaling problem in container clusters. This algorithm has two main advantages over previously existing approaches: its extremely low solving time, and its applicability to complex systems. These characteristics are achieved by decomposing the resource allocation problem into four incremental phases that are solved by combining mathematical optimization methods and heuristics approaches.

Although the main objective of the methodology is cost optimization, FCMA also takes into account other important secondary objectives during the resolution process.

The results presented in the previous section show that FCMA is not only much faster than Conlloovia, but also more applicable, finding feasible solutions to problems that Conlloovia cannot address. Nevertheless, it may be worth exploring refinements of FCMA to handle problems involving even larger numbers of applications and instance classes. In addition, while Conlloovia may occasionally produce lower-cost solutions, FCMA achieves comparable costs in most cases, or even better results when Conlloovia cannot find the optimal solution. Furthermore, FCMA outperforms Conlloovia in metrics related to fault tolerance, node recycling, load balancing reduction, and container isolation. These findings suggest that FCMA represents a significant advance in the field.

When comparing FCMA with other heuristics such as FFC and FFP, it is observed that the solving times are, in many cases, similar, but FCMA provides solutions with significantly lower costs.

Future work will focus on the optimization of the transition between the allocations generated for successive scheduling windows. To achieve this optimization, a methodology that minimizes the differences between the allocations obtained by FCMA for successive scheduling windows must be developed.

Authors' contributions

José María López: algorithm conceptualization, software tool development, writing the original draft, and reviewing and editing the final manuscript. Joaquín Entrialgo: software tool development, experimentation, writing the original draft, and reviewing and editing the final manuscript. Manuel García: writing the original draft, and reviewing and editing the final manuscript. Javier García: writing the original draft, and reviewing and editing the final manuscript. José Luis Díaz: software tool testing, writing the original draft, and reviewing and editing the final manuscript. Rubén Usamentiaga: Funding acquisition.

Funding

This research was funded by the project PID2021-124383OB-I00 of the Spanish National Plan for Research, Development and Innovation from the Spanish Ministerio de Ciencia e Innovación.

Data availability

Data and code are available in public repositories in the following links which are also mentioned in the manuscript. https://github.com/asi-uniovi/fcma/. https://github.com/asi-uniovi/fcma_exp/.

Declarations

Competing interests

The authors declare no competing interests.

Received: 20 December 2024 / Accepted: 16 April 2025
Published online: 06 May 2025

References

1. RedHat (2024) Openshift. https://www.redhat.com/en/resources/openshift-container-platform-datasheet/. Accessed 09 May 2024
2. Apache Software Foundation (2024) Apache mesos. https://mesos.apache.org/. Accessed 09 May 2024
3. Cloud Native Computing Foundation (2024) Kubernetes. https://kubernetes.io/. Accessed 09 May 2024

4. AWS (2024) Amazon elastic kubernetes service. https://aws.amazon.com/eks/. Accessed 09 May 2024
5. Google (2024) Google kubernetes engine. https://cloud.google.com/kubernetes-engine/. Accessed 09 May 2024
6. Microsoft (2024) Azure kubernetes services. https://azure.microsoft.com/en-us/products/kubernetes-service/. Accessed 09 May 2024
7. Entrialgo J, García M, García J, López JM, Díaz JL (2024) Joint Autoscaling of Containers and Virtual Machines for Cost Optimization in Container Clusters. J Grid Comput 22(1):17. https://doi.org/10.1007/s10723-023-09732-4
8. Guerrero C, Lera I, Juiz C (2018) Genetic Algorithm for Multi-Objective Optimization of Container Allocation in Cloud Architecture. J Grid Comput 16(1):113–135. https://doi.org/10.1007/s10723-017-9419-x
9. Qassem LMA, Stouraitis T, Damiani E, Elfadel IM (2024) Containerized Microservices: A Survey of Resource Management Frameworks. IEEE Trans Netw Serv Manag 1–1. https://doi.org/10.1109/TNSM.2024.3388633
10. Dogani J, Namvar R, Khunjush F (2023) Auto-scaling techniques in container-based cloud and edge/fog computing: Taxonomy and survey. Comput Commun 209:120–150. https://doi.org/10.1016/j.comcom.2023.06.010
11. Kiss T, DesLauriers J, Gesmier G, Terstyanszky G, Pierantoni G, Oun OA, Taylor SJ, Anagnostou A, Kovacs J (2019) A cloud-agnostic queuing system to support the implementation of deadline-based application execution policies. Futur Gener Comput Syst 101:99–111. https://doi.org/10.1016/j.future.2019.05.062
12. Aldwyan Y (2021) Intelligent scaling of container-based web applications in geographically distributed clouds. PhD thesis, University of Melbourne, Parkville
13. Al-Dhuraibi Y, Zalila F, Djarallah N, Merle P (2018) Coordinating Vertical Elasticity of both Containers and Virtual Machines:. In: Proceedings of the 8th International Conference on Cloud Computing and Services Science. SCITEPRESS - Science and Technology Publications, Funchal, pp 322–329. https://doi.org/10.5220/0006652403220329
14. Yadav MP, Pal N, Yadav DK (2021) Resource provisioning for containerized applications. Clust Comput 24(4):2819–2840. https://doi.org/10.1007/s10586-021-03293-5
15. Quattrocchi G, Incerto E, Pinciroli R, Trubiani C, Baresi L (2024) Autoscaling Solutions for Cloud Applications under Dynamic Workloads. IEEE Trans Serv Comput 1–17. https://doi.org/10.1109/TSC.2024.3354062
16. Rossi F, Nardelli M, Cardellini V (2019) Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). IEEE, Milan, pp 329–338. https://doi.org/10.1109/CLOUD.2019.00061
17. Shafi N, Abdullah M, Iqbal W, Erradi A, Bukhari F (2024) Cdascaler: a cost-effective dynamic autoscaling approach for containerized microservices. Clust Comput. https://doi.org/10.1007/s10586-023-04228-y
18. Shafi N, Abdullah M, Iqbal W, Bukhari F (2025) DIMA: machine learning based dynamic infrastructure management for containerized applications. Computing 107(3):88. https://doi.org/10.1007/s00607-025-01445-8
19. Jeong B, Jeong YS (2025) ARAScaler: Adaptive Resource Autoscaling Scheme Using ETimeMixer for Efficient Cloud-Native Computing. IEEE Trans Serv Comput 18(1):72–84. https://doi.org/10.1109/TSC.2024.3522815
20. Al Qassem LM, Stouraitis T, Damiani E, Elfadel IAM (2023) Proactive Random-Forest Autoscaler for Microservice Resource Allocation. IEEE Access 11:2570–2585. https://doi.org/10.1109/ACCESS.2023.3234021
21. Mogal AK, Sonaje VP (2024) Predictive Autoscaling for Containerized Applications Using Machine Learning. In: 2024 1st International Conference on Cognitive, Green and Ubiquitous Computing (IC-CGU), IEEE, Bhubaneswar, pp 1–6. https://doi.org/10.1109/IC-CGU58078.2024.10530773
22. Wang Y, Chandra A, Weissman J (2024) Jingle: IoT-Informed Autoscaling for Efficient Resource Management in Edge Computing. In: 2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid). IEEE, Philadelphia, pp 395–407. https://doi.org/10.1109/CCGrid59990.2024.00052
23. Podolskiy V, Jindal A, Gerndt M (2019) Multilayered Autoscaling Performance Evaluation: Can Virtual Machines and Containers Co-Scale? Int J Appl Math Comput Sci 29(2):227–244. https://doi.org/10.2478/amcs-2019-0017
24. Ramirez YM, Podolskiy V, Gerndt M (2019) Capacity-Driven Scaling Schedules Derivation for Coordinated Elasticity of Containers and Virtual Machines. In: 2019 IEEE International Conference on Autonomic Computing (ICAC). IEEE, Umea, pp 177–186. https://doi.org/10.1109/ICAC.2019.00029
25. Podolskiy V (2021) Predictive autoscaling for multilayered cloud deployments. PhD thesis, Technische Universität München. https://mediatum.ub.tum.de/1615040
26. Tran MN, Kim Y (2024) Optimized resource usage with hybrid auto-scaling system for knative serverless edge computing. Futur Gener Comput Syst 152:304–316. https://doi.org/10.1016/j.future.2023.11.010
27. Asensio A, Masip-Bruin X, Garcia J, Sánchez S (2021) On the optimality of Concurrent Container Clusters Scheduling over heterogeneous smart environments. Futur Gener Comput Syst 118:157–169. https://doi.org/10.1016/j.future.2021.01.003
28. Hoenisch P, Weber I, Schulte S, Zhu L, Fekete A (2015) Four-Fold Auto-Scaling on a Contemporary Deployment Platform Using Docker Containers. In: Barros A, Grigori D, Narendra NC, Dam HK (eds) Service-Oriented Computing, vol 9435. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 316–323. https://doi.org/10.1007/978-3-662-48616-0_20
29. Nardelli M, Cardellini V, Casalicchio E (2018) Multi-Level Elastic Deployment of Containerized Applications in Geo-Distributed Environments. In: 2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud). IEEE, Barcelona, pp 1–8. https://doi.org/10.1109/FiCloud.2018.00009
30. Wais A (2021) Optimizing Container Elasticity for Microservices in Hybrid Clouds. Master's thesis, Institut für Information Systems Engineering, TU Wien. https://doi.org/10.34726/HSS.2021.77600
31. Sheganaku G, Schulte S, Waibel P, Weber I (2023) Cost-efficient auto-scaling of container-based elastic processes. Futur Gener Comput Syst 138:296–312. https://doi.org/10.1016/j.future.2022.09.001
32. Lahmann G, McCann T, Lloyd W (2018) Container Memory Allocation Discrepancies: An Investigation on Memory Utilization Gaps for Container-Based Application Deployments. In: 2018 IEEE International Conference on Cloud Engineering (IC2E). IEEE Computer Society, Los Alamitos, pp 404–405. https://doi.org/10.1109/IC2E.2018.00076
33. Wang Q, Kar S, Mishra P, Linduff C, Izard R, Anjam K, Barrineau G, Zulfiqar J, Wang KC (2023) Container resource allocation versus performance of data-intensive applications on different cloud servers. https://arxiv.org/abs/2311.07818
34. Baker BS (1985) A new proof for the first-fit decreasing bin-packing algorithm. J Algorithms 6(1):49–70. https://doi.org/10.1016/0196-6774(85)90018-5
35. Schrijver A (1998) Theory of linear and integer programming. Wiley, Chichester
36. Lenstra HW Jr (1983) Integer programming with a fixed number of variables. Math Oper Res 8(4):538–548
37. Conforti M, Cornuéjols G, Zambelli G (2014) Integer programming models. Springer, Cham. https://doi.org/10.1007/978-3-319-11008-0
38. Cloud Native Computing Foundation (2025) Horizontal pod autoscaler. https://kubernetes.io/docs/reference/kubernetes-api/workload-resources/horizontal-pod-autoscaler-v2/. Accessed 03 Mar 2025
39. Cloud Native Computing Foundation (2025) Cluster autoscaler. https://github.com/kubernetes/autoscaler. Accessed 03 Mar 2025

Publisher's Note