J Netw Syst Manage (2014) 22:517–558
DOI 10.1007/s10922-013-9265-5

Cost-Effective Feature Placement of Customizable Multi-Tenant Applications in the Cloud

Hendrik Moens · Eddy Truyen · Stefan Walraven ·
Wouter Joosen · Bart Dhoedt · Filip De Turck

Received: 7 May 2012 / Revised: 14 January 2013 / Accepted: 21 January 2013 /
Published online: 3 February 2013

Abstract Cloud computing technologies can be used to more flexibly provision application resources. By exploiting multi-tenancy, instances can be shared between users, lowering the cost of providing applications. A weakness of current cloud offerings however, is the difficulty of creating customizable applications that retain these advantages. In this article, we define a feature-based cloud resource management model, making use of Software Product Line Engineering techniques, where applications are composed of feature instances using a service-oriented architecture. We focus on how resources can be allocated in a cost-effective way within this model, a problem which we refer to as the feature placement problem. A formal description of this problem, that can be used to allocate resources in a cost-effective way, is provided. We take both the cost of failure to place features, and the cost of using servers into account, making it possible to take energy costs or the cost of public cloud infrastructure into consideration during the placement calculation. Four algorithms that can be used to solve the feature placement problem are defined. We evaluate the algorithm solutions, comparing them with the optimal solution determined using an integer linear problem solver, and evaluating the execution times of the algorithms, making use of both generated inputs and a use case based on three applications. We show that, using our approach a higher degree of multi-tenancy can be achieved, and that for the considered scenarios, taking the relationships between features into account and using application-oriented placement performs 25–40 % better than a purely feature-oriented placement.

H. Moens (✉) · B. Dhoedt · F. De Turck
Department of Information Technology, iMinds-Ghent University, Gaston Crommenlaan 8/201,
9050 Ghent, Belgium
e-mail: hendrik.moens@intec.ugent.be

E. Truyen · S. Walraven · W. Joosen
DistriNet Research Group, Department of Computer Science, iMinds-Katholieke Universiteit
Leuven, Celestijnenlaan 200A, 3001 Heverlee, Belgium

Keywords Distributed computing · Cloud computing · SPLE · Application placement

1 Introduction

In recent years, there has been an increasing interest in cloud computing [1]. By moving applications to cloud platforms, and making use of multi-tenancy, where multiple end users utilize the same application instances and hardware, administrators can consolidate hardware and save costs. Cloud-hosted applications can also react faster to sudden changes in demand. Different obstacles to the widespread adoption of cloud computing do however still exist. One of the issues with contemporary cloud Software as a Service (SaaS) offerings is that the applications generally offer a one-size-fits-all package, with only limited customizability. Often it is only possible to add minor changes using configuration changes. Software customizability, where entirely separate code paths are executed in different software versions, significantly changing the behavior of applications, is difficult to add to SaaS applications.

Often, applications must however be tailored for specific customer needs, offering similar but slightly differing functionality for different end users. The CUSTOMSS [2] project seeks to create solutions to develop, deploy and manage highly customizable software and services on multi-tenant cloud infrastructures, by incorporating management of the variability of applications into the cloud platform itself. Within the project we focus on applications from three domains: (1) document processing, in which large batches of documents are processed and managed using a web interface; (2) medical information management, where medical data and patient information are stored and processed; and (3) medical communication systems, where communication between patients and nurses is coordinated based on a management system using advanced ontologies. While we focus our evaluation on these three use cases, the presented approach could be applied to all cloud-based applications that require high variability, provided the applications can be split into interacting components. The techniques can either be applied on top of an Infrastructure as a Service (IaaS) or Platform as a Service (PaaS) platform, or can be integrated into existing PaaS platforms. In this article, we will discuss how customizable multi-tenant applications can be managed by a cloud platform.

Software Product Line Engineering (SPLE) [3] concepts are often used to develop customizable applications. In this approach, the software is modeled as a collection of features. By selecting and deselecting features, different software variants can be created. Features themselves are organized by relating them to each other in a *feature model*. These techniques can however not easily be adapted to a cloud context, as most approaches in SPLE focus on the development of statically configured products, where changes are compiled into the application. In this approach, all variations are instantiated and compiled before a product is delivered to customers and, once the decisions are made, it is difficult for users to alter them. When used in a cloud context, this implies that every software variant would be an entirely separate application, making it impossible to use multi-tenancy where instances are shared between users if these users do not use the same variant, thereby greatly reducing the potential cost savings of a migration to the cloud.

An alternative approach [4], where the software is split up into sepnt feature instances using a Service-Oriented Architecture (SOA), and where the individual services are multi-tenant alleviates this shortcoming, but in this case, some services risk being underutilized, especially if many features and variants exist. Furthermore, as these services are dependent on each other, failure of a single service can result in performance degradations for the entire application, which can not be taken into account by current cloud resource allocation mechanisms.

By adding variability information to the applications running on a cloud platform, and managing variability at this platform level, developing highly customizable SaaS applications becomes easier. One very important functionality of the platform, is to decide which applications are executed where. This is known as the application placement problem [5, 6]. Current application placement techniques are however inadequate for this purpose, as they do not take relationships between services, introduced by variability modeling, into account. For our purposes, the placement must take the variability of the managed applications into account, ensuring all application components are allocated sufficient resources. Furthermore, the cost of using servers for running applications must be taken into account as well, as this makes it possible to either minimize the carbon footprint of the managed cloud, or to reduce costs when part of the application is executed on public cloud infrastructure. Within this article, we consider two costs: the cost of failing to provision capacity for application components (determined e.g. by a service level agreement) and the operational cost of using a server.

In this article, we focus on the design of algorithms for placing high-variability applications on cloud infrastructure, extending the methods and evaluations from our previous work [7], adding energy efficiency and server usage costs, and incorporating relations between applications components. The applications are composed from a set of multi-tenant feature instances using a SOA. For this purpose we designed a variation of the application placement problem [8], which we refer to as the *feature placement problem*. An overview is shown in Fig. 1. The resulting feature placement determines which servers will execute which feature instances, taking into account the datacenter server configuration, applications to be placed, and the feature model of which the applications are instantiations. A single feature instance is capable of serving multiple applications, ensuring applications composed of a set of features are themselves multi-tenant. We address the following research questions: (1) How can we define the feature placement problem, and what information is required to define it? (2) How can the feature placement problem be formally modeled? (3) Which algorithms can be designed to solve this problem in an efficient way? and (4) Which performance is achieved by the algorithms compared to the optimal solution, in terms of placement quality and execution speed, and what is the impact of model parameters on the obtained performance?

The contribution of this article is two-fold: (1) we describe how SPLE techniques can be combined with cloud application placement techniques to facilitate the management of high-variability applications; and (2) we formally define the feature

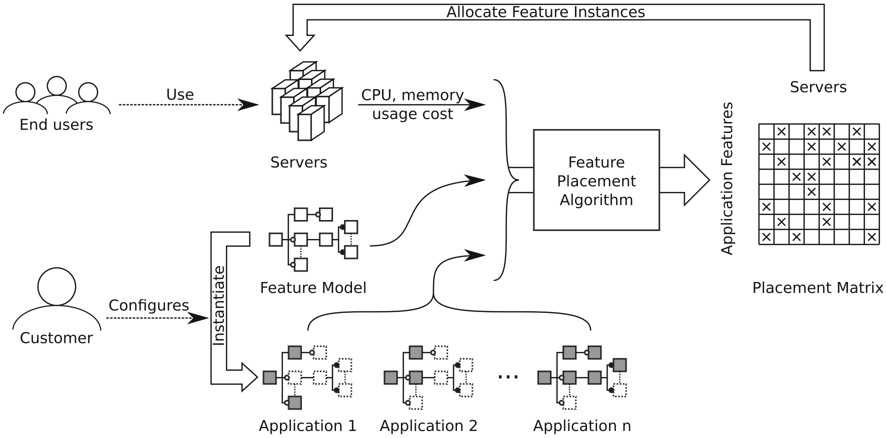


Fig. 1 A schematic representation of the feature placement problem. Application instantiate a feature model. Features instances are placed on physical servers and can be used by multiple applications

placement problem, define optimal and heuristic algorithms, and evaluate them. In the next section, we will discuss related work. Afterwards, in Sect. 3 we explain the feature modeling approach, and how it can be applied to cloud applications. We then formally define the feature placement problem in Sect. 4. This is followed by Sect. 5, where we outline different approaches to solve the placement problem. In Sect. 6 we describe the set-up of the evaluation. Subsequently, in Sect. 7 we evaluate the algorithms. Both the quality of the results of the algorithms, and their execution speeds are discussed. Finally, Sect. 8 contains our conclusions.

2 Related Work

This work builds on two research areas: SPLE and feature-oriented application development, and application placement.

2.1 Software Product Line Engineering

SPLE is used to manage the variability of applications, making it easier to build and manage groups of similar applications, with different feature sets. Managing a separate codebase for every software variant family would introduce a large overhead. Instead, only a single codebase is used, in which the variability is managed using SPLE techniques. Research has been done on configuration policies and methodologies to support customizations of SaaS. In Zhang et al. [9], discuss a policy-based framework for publishing customization options of web services and building customizations on top of this, enabling clients to build their own customizations. They however do not take multi-tenancy and runtime aspects into account, nor do they propose a software development methodology to create the customizable applications. Sun et al. [10] propose an approach choosing configuration over customization to create modifiable applications, and propose a software development methodology to develop such applications. We, by contrast, focus on the customization aspect by using SPLE methods in combination with a SOA development approach. In Mietzner et al. [4] an approach for modeling customizable applications built using SOA is described. The application is linked to a feature model, allowing automatic generation of deployment scripts. Our approach is similar in its use of SOA in the proposed development approach. We however focus on the resource allocation of customizable applications, proposing optimal and heuristic algorithms to determine where to run specific features. Recent work in the SPLE community [11–14] further progresses towards the development of customizable SaaS applications, but mainly focuses on the design-time variability of these applications, and not on their runtime management. Work on the dynamic adaptation of SOA applications was conducted in [15], but it does not address how these applications must be placed on physical infrastructure.

2.2 Application Placement

The application placement problem is used within clouds and clusters to determine which services to execute on which servers, and has previously been formally described [5, 6, 8, 16–18]. Many different approaches to application placement in clouds have been developed over the recent years. Specific requirements have however led to the creation of many extensions to the application placement problem, each focusing on different parameters. Whalley and Steinder [19] extended a Virtual Machine (VM) management system to take into account the complexities of software licensing. In a similar way, Breitgand and Epstein [20] added the consideration of Service Level Agreements (SLAs) to the placement problem. The consideration of energy consumption and carbon emissions was added in [21] using a system that works in parallel with existing datacenter brokering systems. We extend the generic application placement problem formulation to place the features of applications in a cloud environment. Our approach further differs from the traditional application placement problem formulation and its variants, as we consider an application to be a set of interacting services, and not just a single service. By contrast to the existing work surrounding application placement, our placement approach not only takes these services, but also the relations between them into account during the placement calculation.

The algorithm we describe within this article has similarities with the linear application placement algorithm described in [16]. Our work however adds the concept of software variability. Furthermore, our application-based feature placement algorithm aims to place all application components at once and adds a backtracking phase to the algorithm if placement of an application fails, lowering the cost of placements.

Energy efficiency and server usage costs are incorporated in an application placement system in [22]. The authors however focus on the placement at a VM level, while our approach focuses on managing multi-tenant applications where multiple applications can make use of a single instance, meaning more fine-grained control is needed. Furthermore, our algorithm also adds explicit support for software variability. This enables the management system to dynamically fill in undecided variability, known as open variation points, at runtime.

In [23], the concept of application component placement is introduced, where applications consisting of separate components are placed within datacenters, and an integer linear programming algorithm to solve the problem is introduced. Our approach similarly focuses on applications consisting of multiple components, but we by contrast add support for multi-tenancy, making it possible for multiple tenants to make use of individual application components. Additionally, we also take relations between application components, modeled using SPLE, into account during the placement.

We have previously discussed the runtime management of, and resource allocation for highly customizable applications [7]. In this article we extend the problem description, generalizing the inputs, and add a server use cost, ensuring energy efficiency and hybrid cloud scenarios can be taken into account. We also incorporate requirements that improve the problem applicability, ensuring the algorithm can better handle scenarios where memory requirements increase when the loads increase, and situations where features depend on each other to function. We also present and evaluate an improved, application-centric placement algorithm yielding more cost-effective resource allocations than the algorithms described in our previous work

In literature, most application placement algorithms make use of specific resources, usually taking into account CPU and memory limitations [5, 8, 24, 25], application bandwidth requirements [26], or generalized load-dependent and load-independent resources [27]. Our approach generalizes these inputs, as done in [27], but goes further by allowing for the definition of multiple resources. This is achieved by making use of concepts we previously described in [28], enabling the management of high-variability applications with heterogeneous resource demands.

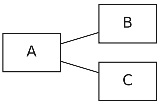
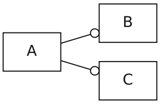
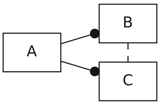
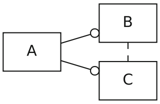
In [29, 30], a management system for services composed of multiple VMs is presented. These works focus on the definition and deployment of composed cloud services. Our work is complementary with this approach, as it focuses on the relations between the different services using SPLE techniques and not on how these relations are represented. We also focus on the physical location where the instances are executed, rather than how they are deployed.

3 Feature Placement Concepts

Using SPLE, an application is modeled as a collection of features and relations between these features. The features are then linked to actual code modules or configuration files. Sometimes the inclusion of a feature can imply the inclusion or exclusion of other features, which is represented using relations in the feature model. A software variant can then be generated by selecting and excluding features from this feature model.

To facilitate reasoning on these relations, feature models are often created in a hierarchical fashion. Table 1 contains the different relation types, a description, a graphical representation, based on the notation used in [31], and a formal notation

Table 1 Graphical representation of feature models, description of relations, and formal representation

	Mandatory If the parent is selected the child must be selected as well **Mandatory**(f_A, f_B) **Mandatory**(f_A, f_C)
	Optional If the parent is selected the optional children can be selected **Optional**(f_A, f_B) **Optional**(f_A, f_C)
	Alternative If the parent is selected exactly one of the child nodes must be selected **Alternative**(f_A, {f_B, f_C})
	Or If the parent is selected at least one of the child nodes must be selected **Or**(f_A, {f_B, f_C})

The nodes on the left are parent features, those on the right are child features

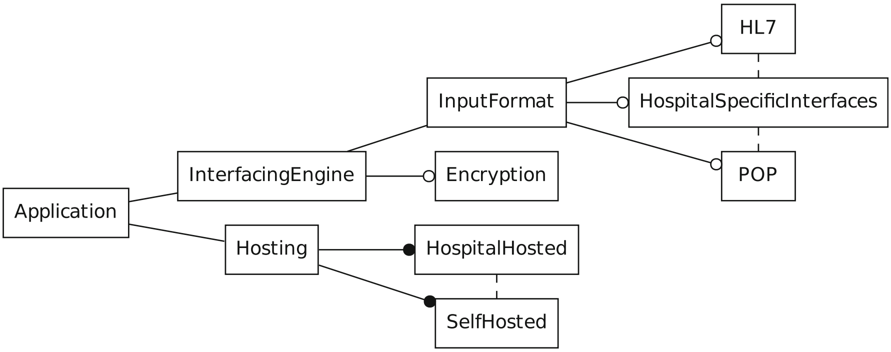


Fig. 2 A feature model fragment for a medical data processing application

which will be used later on in this article. An example feature model is shown in Fig. 2. The figure shows an illustrative fragment of the feature model for a medical data processing application. The application contains an *interfacing engine* feature to connect to individual hospitals, which is capable of handling input in one or more different formats. Additional *encryption* can optionally be added to the interfacing engine. Finally, parts of the application can be hosted at the hospital or they can be hosted by the application provider. An application created for a hospital using their own datacenter and a hospital specific interface will differ significantly from the application created for a hospital using public cloud infrastructure and a standard medical data interface.

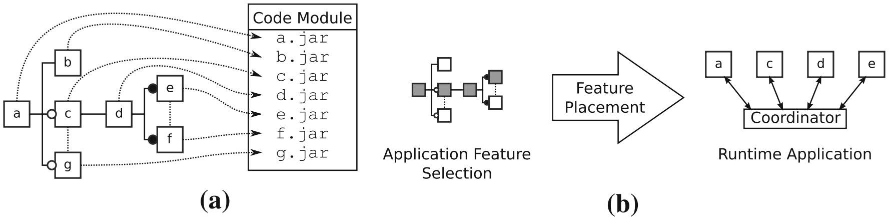


Fig. 3 Features are associated with code modules. Applications containing the features are created by instantiating these features and linking them together. **a** Associating code modules with features. **b** Deploying applications composed of feature instances

Sometimes a feature can be implemented by simply updating configuration files. This could for example be changing the logo of an application. More complicated changes can be created by adding code changes. The most complicated changes lead to completely different modules being used by the applications. The first method is variation by configuration, the latter two variation types are referred to as customization [10]. In this article, we only consider customization, which leads to the creation of applications that are different at the code level. Configuration-based features can already be adapted into a cloud context using existing software development techniques [10]. The feature models used further on in this article will only contain features that cause changes at a code level in the deployed services.

The development of applications will be driven using the feature model, building an application using a SOA, in which the individual services map to the different features defined in the feature model. An example of this is shown in Fig. 3a. Deploying the application then comes down to allocating feature instances and connecting them either to each other or by using a coordinator component, illustrated in Fig. 3b. To determine where these instances are placed, a feature placement algorithm is used. We assume that the individual services are multi-tenant and can serve multiple applications. In our use cases, the various feature instances are developed, managed and tested by the platform provider, ensuring the components can be trusted and that calls to a service respect the tenant resource limits. This in turn minimizes performance interference between tenants which can be caused by sharing a feature instance between different tenants. The allocation of the different feature instances, taking into account relations as defined by the feature model, is the main focus of this article. We assume the application has already been split up into components, and that data isolation issues are resolved using existing techniques [32, 33]. We also assume the performance of the various components has been evaluated using performance models such as those in [34], possibly grouping components that often communicate together to guarantee they are colocated, which ensures good performance is achieved.

When configuring a SPLE application, part of the variability can be left undecided, creating open variation points [4]. When two applications with different feature configurations exist, and some have open variation points, this information can be used to reduce the cost of the full placement. This makes it particularly interesting to take these points into account during the placement of applications.

An application with, e.g., regular availability requirements can use high availability instances when such instances exist with remaining capacity, rather than creating new instances with lower reliability, effectively lowering the total resource usage. This is further illustrated in Fig. 4, where two applications are shown. The first application uses Feature D, and the second application requires either Feature B, C, or D. If the placement function is unaware of these open variation points, it would simply choose the cheapest alternative, while a choice for the, possibly more expensive Feature D might be preferable as this choice reduces the total number of feature instances used in the application.

The inputs and outputs of the feature placement problem are shown in Fig. 5. Input for the placement problem comes from three sources: the servers, on which the application are executed, the feature model, that defines the structure of applications that are to be placed, and the applications themselves. More specifically:

- The *servers* contain resources, such as CPU, memory, disk space and bandwidth. Each of these are limited, and it is impossible to allocate more of these resources

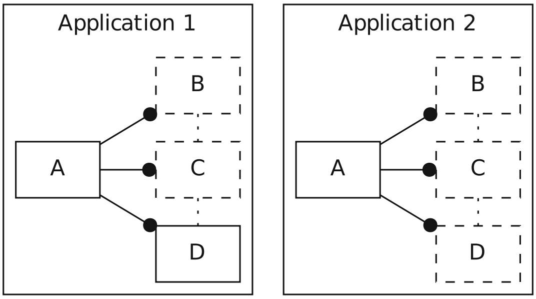


Fig. 4 Different feature model selections for two applications. Features with a *solid border* are selected, features with a *dotted border* are undecided and remain open variation points. By selecting Feature D for Application 2 during placement, the total resource requirement of both applications can potentially be decreased

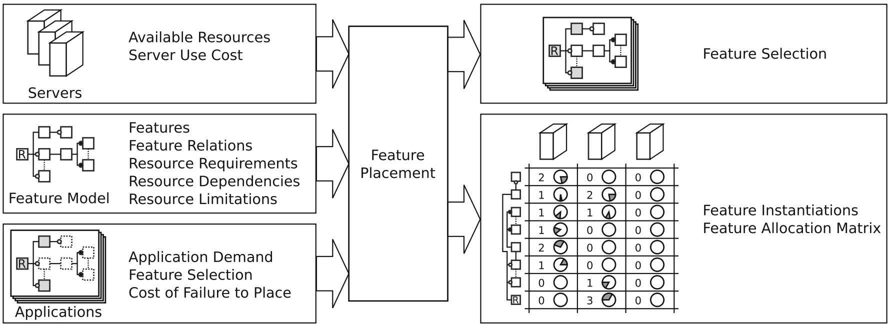


Fig. 5 A detailed overview of the feature placement inputs and outputs. We use a small example feature model to illustrate the inputs. The root of this feature model is marked using the *letter R*

to feature instances than available. Using a server also incurs a server use cost. This is the operational cost of using the server, and can represent an energy cost, to determine an energy-efficient placement, or an instance cost in a hybrid or public cloud environment.

- The *feature model* describes the different features, and the relations between them. These must be expressed, ensuring they can be taken into account during the placement process. For every feature, instance resource requirements are needed. The different feature instances may also impact each others resource requirements. For example, the addition of a security feature can increase the load on other services, that then have to use encryption in their communications. To get a realistic view of the actual resource need of features, the impact features have on each others resource demands is added as an input. Finally, it is possible that a single instance of a feature, with a fixed amount of e.g. memory, can only use a limited amount of resources, e.g. CPU. If this is the case, these limitations are also added as an input of the feature placement problem. Then, multiple instances of a feature can be instantiated on a single server to handle higher resource demands.
- Each *application* is an instantiation of the feature model, with a specific selection of features. Applications add three parameters to the feature placement: (1) the demand, that varies depending on the load on the application, (2) a feature selection, that indicates the selected and excluded features, and (3) the cost of failing to place the application. In some cases an additional cost for the failure to provision specific features can be added, for example a feature providing a minimal service. If so, this feature failure cost is also added as an input. The cost of failing applications and features can either be an actual economical cost, defined in a SLA, or it can be an estimated cost such as the potential cost of losing customers due to a bad service.

Using these inputs, the feature placement will generate two outputs:

- For every application, a *feature selection* will be returned. This contains all the features that were selected in the feature selection input variable, but any remaining open variation points are filled in.
- A *placement*, that contains for every server the number of instances of a feature that are executed on it, and the amount of resources allocated to them. Each instance of a feature uses part of the available resources on the server on which it is executed (represented using pie charts in Fig. 5). When no services are allocated on a server, it can be turned off, reducing the operational cost of the placement.

When a resource conflict occurs, and more resources are needed by applications than available, the algorithm handles this conflict by choosing the best configuration based on its resulting cost. In this case, some of the application features will not be placed. The cost used within the optimization is composed of the cost incurred by a failure to place applications and the cost of using a server, with the cost of failure of placing applications typically significantly larger than the cost of using servers. An optimal solution to the feature placement problem is a placement that minimizes the total cost.

A feature placement algorithm will be used as one of the central components of a cloud management system. The system architecture of this management system contains three components that are responsible for determining feature placement inputs: (1) a staging environment where new configurations are tested, and where the impact of features on other features can be measured in a controlled environment; (2) a monitoring system, that can be used to dynamically improve estimated demand and impacts during execution; and (3) an admission controller, limiting the number of applications admitted into the system.

4 Formal Problem Description

The variables used in the model are listed in Table 2. We begin by discussing the optimization objective. This is followed by a description of the input variables. Three variable types can be distinguished: input variables, decision variables and auxiliary variables. Finally we will discuss the constraints used within the model.

4.1 Optimization Objective

The objective of a placement algorithm is to minimize two costs: the cost of failure to place an application or feature, which we refer to as the cost of non-realized demand, and the cost of using servers, referred to as the server use cost. When multiple applications contend for resources, the configuration with the lowest cost according to this objective function will be chosen.

More formally, the goal of the model is to minimize the cost, C, of the placement. This cost is determined by two factors: the cost of non-realized demand, C_D, which is incurred due to failure to place applications, and a server use cost, C_U which is incurred when servers are used. We express this cost using Eq. (1).

$$C = C_D + C_U \tag{1}$$

C_D is defined in Eq. (2).

$$C_D = \sum_{a \in A} \left(p_a \times C^V(a) + \sum_{f \in \mathrm{sel(a)}} p_{f,a} \times C^V(f,a) \right) \tag{2}$$

Equation (2) uses binary variables to indicate when the provisioning of features or applications fail, and multiplies these binary variables with the cost that this failure causes. The variable $p_{f,a}$ takes on value 1 if the feature f of an application a is not provisioned sufficient resources, and 0 otherwise. Similarly, a binary variable p_a is used to express the failure of *any* feature of an application a. To determine the total costs, these binary variables are then combined with the cost of failing to provision individual features $C^V(f, a)$, and the cost of failing to provision an application $C^V(a)$.

Table 2 The different symbols used in Sect. 4

Symbol	Type	Description
Input variables		
Γ		The set of considered resource types (eg. memory, CPU, bandwidth)
Γ_s	$\wp(\Gamma)$	Resource types for which the demand is strict: they must be allocated for each feature instance for the instance to be usable
$\Gamma_{\bar{s}}$	$\wp(\Gamma)$	The resource types for which the demand is non-strict: the goal of the optimization process it to allocate as much of this demand as possible, but a configuration in which not all of these resources have been allocated is still valid
S		The set of servers
Ra_s^{γ}	$[0, +\infty)$	The available resources on a server s for a resource type $\gamma \in \Gamma$
$\mathscr{F}$		The feature model used by the applications
F		The set of features contained in $\mathscr{F}$
$\mathscr{R}$		The set of relations contained in $\mathscr{F}$, using relations as described in Table 1
A		The set of applications
sel (a)	$\wp(F)$	The features that must be included for application a
excl (a)	$\wp(F)$	The features that must not be included for application a
$FI^{\gamma}_{f\_1}(f_2)$	$[0, +\infty)$	The impact on the resource requirement for feature f_2 if feature f_1 is included in the selected features of an application, for a resource type $\gamma \in \Gamma_{\bar{s}}$
D_a	$(0, +\infty)$	The demand for an application a
IR_f^{γ}	$[0, +\infty)$	The resource requirement of a single instance of a feature f for a resource type $\gamma \in \Gamma_s$
L_f^{γ}	$(0, +\infty)$	The instance limitations indicate the maximum amount of non-strict resource type γ that can be allocated to a single instance of a feature f
$C^V(f, a)$	$[0, +\infty)$	The cost of failing to place a feature f for an application a
$C^V(a)$	$[0, +\infty)$	The cost of failing to place an application a
$C^U(s)$	$[0, +\infty)$	The cost of using a server s.
Decision variables		
$M^{\gamma}_{s,f,a}$	$[0, +\infty)$	The amount of a resource $\gamma \in \Gamma_{\bar{s}}$ to be allocated for a given server s, feature f and application a
$\Phi_{f,a}$	$\{0,1\}$	A binary variable indicating whether application a includes feature f
$IC_{s,f}$	$\mathbb{N}$	The instance count is an integer variable, indicating how many instances of a feature f are instantiated on a server s
Auxiliary variables		
$AI^{\gamma}_{f,a}$	$[0, +\infty)$	The application impact, containing the actual resource impact per feature f of a specific application a, for a resource $\gamma \in \Gamma_{\bar{s}}$
p_a	$\{0,1\}$	A binary variable that has value 1 if an application a is not correctly placed, that is when any of its features are not placed
$p_{f,a}$	$\{0,1\}$	A variable that has value 1 when the resource demand of a single feature f of an application a is not placed
$p^{\gamma}_{f,a}$	$\{0,1\}$	A variable indicating whether the resource demand of a single feature f of an application a is not placed, for a specific resource $\gamma \in \Gamma_{\bar{s}}$
U_s	$\{0,1\}$	A binary variable indicating whether a server s is used

Note that within our approach, any feature can fail, including those that are considered mandatory; feature failure is handled by assigning a cost to it. This is done to ensure the feasibility of results: by enforcing the inclusion of selected features using constraints, some inputs could lead to an infeasible result to which no solution exists. It is better for a single application or feature to fail, than for there not to be a feasible solution at all. More importantly, if no feasible solution can be determined, it is important that the application or feature that fails incurs the lowest possible cost.

The cost of using a server is expressed in Eq. (3). The equation makes use of a server usage cost $C^U(s)$, denoting the cost of using a server, and binary variables U_s indicating whether a server is used.

$$C_U = \sum_{s \in S} U_s \times C^U(s) \tag{3}$$

4.2 Input Variables

In literature, application placement techniques are generally designed to place application instances on servers, ensuring a global CPU demand is met. Each of these application instances requires a fixed amount of memory for it to work. Some works, however, make use of different resource types, e.g. bandwidth [26], or sometimes the resource types are abstracted [27]. To ensure maximum applicability of the formal model, we will define it making use of two generalized resource types, and we will allow multiple resources of these types to occur:

- The first resource type is associated with individual instances, and these resources are needed to create a valid instance. Every instance needs exactly the right amount of these resources to function correctly. We refer to these resources as *strict resources*, as a given amount of them is needed to create a valid feature instance. This in turn implies these requirements are enforced as *constraints*. The memory resource, in a VM placement scenario, has this behavior, as an instance needs a fixed amount of memory to run. Similarly, disk space is also a resource of this type, as each VM requires disk space for its image. In some cases, a fixed amount of bandwidth is required per instance, making it a resource of this type.
- The second resource type behaves differently. For these resource, there is a *global demand*, that must be fulfilled, and fulfilling as much of this demand as possible is the goal of the optimization process. To succeed, instances must be created that handle part of the resource demand. We refer to these resources as *non-strict resources*. The traditional example of this resource requirement is the CPU demand, that is often used in application placement. In some cases other resource types can occur, such as bandwidth, if fulfilling a given bandwidth demand is the optimization goal.

We have previously made a similar distinction between resources in [28], and a similar approach was used in [27]. Within the model, we define Γ as the collection of all resource types, and we use Γ_s and $\Gamma_{\overline{s}}$ to denote strict and non-strict resource types respectively. Note that in practice less strict resources than needed could be allocated to an instance: a virtual machine can for example function with less memory at the cost of performance degradation. Characterizing this performance degradation is however service-specific, and as every service is used by multiple applications this performance degradation impacts the quality of multiple applications. By using conservative fixed resource requirement estimates, these issues can be avoided. For these reasons, strict resources are defined as a fixed value requirement within this article.

Sometimes a pure separation between the two resource types is difficult to achieve, as an increase in for example CPU use can sometimes cause increasing memory utilization. To linearize this problem, we will introduce instance limitations further in this section. These ensure a limit is added to the amount of work a single instance can process, ensuring that memory-intensive applications can also be modeled using this formulation.

Each problem also has a set of servers S with an amount of available resources. For a server $s \in S$ the available resources are given by Ra_s^γ, for the different resource types $\gamma \in \Gamma$. The goal of the optimization is to allocate the required non-strict resources for applications at a minimal cost, while ensuring the created instances have the required strict resources they require to execute.

The problem statement also contains a set of applications A that must be placed. Each of the applications is a specific instantiation of a global feature model $\mathscr{F}$. This feature model contains a set of features F and a collection of relations $\mathscr{R}$, formally describing the feature model tree. The possible relations are described in Sect. 3 and Table 1. This approach still allows the placement of entirely distinct application types with separate feature models $\mathscr{F}_i$ by creating a set containing the roots of every feature model, R, and linking these different feature models in a global feature model by the addition of a new root feature r and a relation **Alternative**(r, R). This ensures that each application executed on the cloud is an instance of exactly one of the separate feature models, and that an arbitrary amount of different application types can be placed using the model. An example of this will be shown in Sect. 7.3.2.

Every application $a \in A$ contains a set $\mathrm{sel}(a) \in F$ with features that must be selected in the application and a set $\mathrm{excl}(a) \in F$, containing features that must be excluded. The configuration of both is assumed to be valid according to $\mathscr{F}$. Features contained in neither set are considered open variation points as described in Sect. 3.

It is possible for features to impact the resource needs of other features. For instance, adding the *encryption* feature to the application in Fig. 2 can increase the CPU load on the *interfacing engine*, and applications hosted by the application provider will require more CPU resources than applications partially hosted at the client site. We assume that applications with similar feature selections will have similar load characteristics, as this is the case for the three application use cases, and represent this using a feature impact matrix FI. $FI_{f_1}^{\gamma}(f_2)$ represents the impact of feature f_1 on feature f_2 for a non-strict resource type γ. The resource requirement of a feature f can be expressed using the feature's impact on itself $FI_f^\gamma(f)$. By including a feature f, it's own feature impact $FI_f^\gamma(f)$ is added, representing the resource requirement of the feature itself, and it's impact is added to all other features f' for which $FI_f^{\gamma}(f') \neq 0$. When two applications make use of the same feature, they will both require resources allocated to this feature, and thus both resource requirements will be counted to determine the total resource demand for this feature. As the demand for an application varies in time, we also add a D_a variable denoting the user demand for an application a. This variable impacts the resource need for the entire application. If load characteristics can vary for individual applications, the approach could be extended ensuring a *FI* matrix is defined per-application, but in such a case detailed measurements would be needed to determine this matrix for every individual application. Every instance of a feature f also requires a specific amount of strict resources IR_f^{γ}.

In many situations, it is unrealistic to assume that a single instance with limited strict resources allocated to it, would be able to use an unlimited amount of non-strict resources. Because of this, we introduce resource limitations: A single instance of a feature f cannot use more than L_f^{γ} of non-strict resource $\gamma \in \Gamma_s$. E.g. an application component that is memory intensive will have a low limit, ensuring only a limited amount of CPU can be used by it. These limitations make the model more applicable to real-life applications, ensuring the ratio between allocated non-strict and strict resource types remains realistic.

The optimization process is used to minimize the cost of failed placement and the server use cost. Two variables are needed to represent the cost of failing to place specific features and applications:

- The cost of failing to reserve the capacity for a specific feature f of an application a is given by $C^V(f, a)$. This can be used if failure of specific features needs to be taken into account.
- The cost of failing to reserve the capacity for *any* feature of an application is given by $C^V(a)$.

Finally, for every server s, the cost of using the server $C^U(s)$ can be determined. This cost can be the energy cost of using the server, or the cost of using a server from a remote IaaS provider. This parameter allows the system to take energy-efficiency of the cloud into account, and could also be used to differentiate between the cost of using the local datacenter and a remote IaaS cloud in a hybrid cloud scenario.

4.3 Decision Variables

The output of the formulation is a placement, indicating which applications are executed where, and the amount of resources allocated for each of these applications. The resource types behave differently, leading to two separate expressions. For strict resource types, we determine how often an application is instantiated on a server, a value that can be used to determine the required strict resource requirement. As the amount of non-strict resources that can be allocated for a given feature can be limited, it is possible for there to be multiple instances of a feature on a single server. For non-strict resources we make use of a matrix representing the amount of resources allocated on a server. This yields two variables:

- The variable $IC_{s,f}$ determines the number of instances of feature f on server s. This integer variable can be used to determine the total strict resource usage of features on servers, by multiplying it with the, fixed, per-instance resource requirements IR.
- For non-strict resources we use an allocation matrix M. For a server s, feature f, and application a, $M^{\gamma}_{s,f,a}$ contains the amount of non-strict resources of a type γ that need to be allocated.

Another output is the feature selection matrix Φ, indicating which applications are selected and excluded for a given application. For application a and feature f, $\Phi_{f,a} = 1$ if the application contains the feature and $\Phi_{f,a} = 0$ if it does not. At the start of the algorithm this matrix can be partially filled in by using sel (a) and excl (a), the remaining features are assigned values during the optimization process.

4.4 Auxiliary Variables

Up until now, we have not yet defined a variable that determines the actual resource requirement of a feature of an application. For this, we define the application impact matrix $AI^{\gamma}_{f,a}$ which contains, for every feature f of application a, the actual resource requirement for a given non-strict resource γ. This matrix can be constructed using the selected features and the feature impact matrix.

A set of binary variables is needed to express whether an application is correctly provisioned. We do this by introducing variables denoting application failure, feature failure, and the failure to provision specific resource demands for an application:

- For every application a there is a variable p_a, indicating whether the provisioning of an application has failed. If $p_a = 1$, a feature of a exists that has not been allocated sufficient non-strict resources.
- For every application a and feature f there is a variable $p_{f,a}$. This variable indicates whether a specific feature of the application is insufficiently provisioned.
- For every application a, feature f and non-strict resource γ, there is a variable $p^{\gamma}_{f,a}$, which has value 1 when too few resources of type γ were allocated for a feature of an application.

Finally, a collection of variables is needed to determine whether a server is active. For every server s there is a binary variable U_s, indicating whether the server is used. If $U_s = 0$ the server is not used and can be turned off.

4.5 Constraint Details

In the following sections we will discuss the different constraints included in the model.

4.5.1 Feature-Based Constraints

The feature selection matrix Φ is used to indicate whether a feature f is present in an application a, $\Phi_{f,a}$ being 1 if f is included in a, and 0 if it is not. For an application a

we add the constraints $\Phi_{f,a} = 1$ if $f \in \mathrm{sel}\,(a)$ and $\Phi_{f,a} = 0$ if $f \in \mathrm{excl}\,(a)$. If the feature does not occur in either set, the value of $\Phi_{f,a}$ remains undecided, creating open variation points, which will be filled in during the optimization process.

The relations between features $\mathscr{R}$ must also be converted into constraints. Elements of $\mathscr{R}$ define relations between individual features. As the constraints of the feature model affect all applications, they must be applied to all application features in the feature selection matrix. Because of this, we define $f_i = \Phi_{i,*}$ a row of the feature selection matrix. We describe the conversion for the relation types to constraints in Table 3. This conversion is required as the logical constraints defined by the relations must be converted into linear expressions for them to be formally used within the model. When we, for example, apply this conversion to the **Alternative**(f_A, $\{f_B, f_C\}$) relation, this yields the constraint $\Phi_{f_A,a} = \Phi_{f_B,a} + \Phi_{f_C,a}$, for every $a \in A$.

4.5.2 Application Resource Requirement Constraints

Each feature f can have resource requirements, but it can also impact resource requirements of other features. If feature f is selected, its impact matrix, FI_f^γ will be added to the total resource requirement for the application. A feature f_i can only affect a feature f_j if f_i requires f_j according to the feature model. Otherwise the feature impact matrix would be able to add feature constraints not included in the feature model, which could in turn lead to inconsistencies.

Using the selected features Φ and the feature impact matrices FI_f^γ, an application impact matrix $AI_{f,\,a}^\gamma$ can be constructed. This application impact matrix, expressed in Eq. (4), displays the resource requirements for individual features f, of an application a, for a given non-strict resource γ, and additionally takes into account the global application demand variable D_a for the application.

$$AI_{f,a}^\gamma = D_a \times \sum_{f' \in F} \Phi_{f',a} \times FI_{f'}^\gamma(f) \tag{4}$$

4.5.3 Resource Constraints

Resource constraints are expressed for every server s, but this is done differently for strict and non-strict resources. For non-strict resources, the used resources are expressed using the allocation matrix M^γ, of which the requirement is aggregated

Table 3 Conversion of relations in the feature model $\mathscr{F}$ to constraints

Relation	Conversion
Mandatory(f_A, f_B)	$f_A = f_B$
Optional(f_A, f_B)	$f_A \geq f_B$
Alternative(f_A, $\{f_B, f_C\}$)	$f_A = f_B + f_C$
Or(f_A, $\{f_B, f_C\}$)	$f_A \geq f_B$
	$f_A \geq f_C$
	$f_A \leq f_B + f_C$

over all features and applications. This is done, for every $\gamma \in \Gamma_{\bar{s}}$, in Eq. (5). Strict resource limitations follow from the instance count *IC* for the service, indicating the number of times a service is allocated, and the required amount of strict resources per-instance, as shown in Eq. (6), which is added for every $\gamma \in \Gamma_s$.

$$\sum_{f \in F} \sum_{a \in A} M^{\gamma}_{s,f,a} \leq Ra^{\gamma}_{s} \tag{5}$$

$$\sum_{f \in F} IR^{\gamma}_{f} \times IC_{s,f} \leq Ra^{\gamma}_{s} \tag{6}$$

As discussed earlier in Sect. 4.2, we assume that single feature instances are only capable of using limited amounts of resources. This is expressed using Eq. (7). The equation expresses that the total resource allocation, for a non-strict resource type γ, of a given feature *f*, on server *s*, must not exceed the amount of resources the instances can handle.

$$\sum_{a \in A} M^{\gamma}_{s,f,a} \leq L^{\gamma}_{f} \times IC_{s,f} \tag{7}$$

4.5.4 Application Provisioning Constraints

Additional constraints are needed to ensure the variables $p^{\gamma}_{s,f,a}$, $p_{f,a}$ and p_f, introduced in Sect. 4.4, correctly express whether the application and features are insufficiently provisioned. Logically, we can express the $p^{\gamma}_{f,a}$ this using Eq. (8):

$$p^{\gamma}_{f,a} \equiv \sum_{s \in S} M^{\gamma}_{f,s,a} < AI^{\gamma}_{a,f} \tag{8}$$

This statement can be turned into constraints using the transformation of Eqs. (9–10), with $x \in \{0, 1\}$, and $\mathbf{M}$ a number larger than any possible value of $\mathbf{expr}$. If $x = 0$, it follows from Eq. (10) that $\mathbf{expr} \leq 0$, while $x = 1$ yields the constraint $\mathbf{expr} \leq \mathbf{M}$, which is always true. Consequently, this transformation holds only in optimizations where the objective function value improves when $x = 0$, which is the case here as a placement in which no applications fail ($p^{\gamma}_{f,a} = 0$) is preferred by the optimization objective function.

$$x \equiv \mathbf{expr} > 0 \tag{9}$$

$$\mathbf{expr} \leq x \times \mathbf{M} \tag{10}$$

Applying the transformation to Eq. (8), $\mathbf{expr} = AI^{\gamma}_{a,f} - \sum_{s \in S} M^{\gamma}_{f,s,a}$. To determine a minimal value for $\mathbf{M}$, we must find a maximum value for the first term, and a minimal value for the second term of $\mathbf{expr}$. For the first term, the definition of *AI*, Eq. (4), can be used with an application that contains all features. For the second term, an empty allocation matrix can be used. This leads to $\mathbf{M} = 1 + \sum_{f' \in F} FI^{\gamma}_{f'}(f)$, ensuring $\mathbf{M} > \mathbf{expr}$ for all possible values of $\mathbf{expr}$.

Once the different $p^{\gamma}_{f,\,a}$ variables are determined, we can use these to determine the value of $p_{f,\,a}$ by expressing, for all of the non-strict resource types γ, that $p_{f,\,a} \geq p^{\gamma}_{f,\,a}$, as the failure for a single resource type ($p^{\gamma}_{f,\,a} = 1$) implies the failure of the entire feature ($f_{f,\ a} = 1$). We also add the constraint $p_a \geq p_{f,a}$ for every feature f and application a, using a similar logic.

4.5.5 Cascading Failure of Features

Child features are dependent on their parent features, and require the parent feature to be selected for them to be used. This implies that, should the parent feature fail, the child feature will fail as well. This is easy to add to the model by, for every parent feature f and child feature c related in the feature model, and every application a, adding the following constraint:

$$p_{f,a} \leq p_{c,a} \tag{11}$$

Equation (11) expresses that if a parent feature fails for an application, the child features must fail as well.

4.5.6 Server Usage Constraints

The variable U_s expresses whether a server s is used. Logically, a server is used if any resource $r \in \Gamma$ is allocated on the server. We express this using Eq. (12).

$$U_s \equiv TSU_s \neq 0 \tag{12}$$

$$TSU_s = \sum_{\gamma \in \Gamma_{\bar{s}}} \sum_{f \in F} \sum_{a \in A} M^{\gamma}_{f,s,a} + \sum_{\gamma \in \Gamma_s} \sum_{f \in F} IR^{\gamma}_{f} \times IC_{s,f} \tag{13}$$

Equation (13) describes the total server use (TSU) for a server s, and calculates the sum of all resources used on the server. This adds values for all non-strict resource types, by summing them over the allocation matrix M, and for all the strict resource types by multiplying the instance counts IC with the instance requirements IR. This summation adds elements with different unit types, so the actual resulting value is of little use, but as soon as a single resource is used on the server, TSU_s will be non-zero, ensuring $U_s = 1$.

The transformation from Eqs. (14) to (15) transforms these logical statements into constraints, and only holds if **expr** is non-negative, which is the case here as negative resource requirements are impossible. If $x = 1$, then **expr** $\leq$ **M**, which is always true. If $x = 0$, it follows that **expr** ≤ 0, which taking into account that **expr** ≥ 0 implies that **expr** $= 0$. Again, this transformation holds only if the placement quality benefits when $x = 0$, as otherwise this option will not necessarily be taken, but this is the case as switching off servers ($U_s = 0$) lowers the cost of execution.

$$x \equiv \mathbf{expr} \neq 0 \tag{14}$$

$$\mathbf{expr} \leq x \times \mathbf{M} \tag{15}$$

Like in the previous section, we can determine a minimal value for **M**, again by finding a maximal value for **expr**. Here this can be done by observing that **expr** equals the sum of all resources used on a server, which can never be larger than the sum of all available resources. Thus, we choose $\mathbf{M} = 1 + \sum_{\gamma \in \Gamma} Ra_s^{\gamma}$.

5 Solution Techniques

We consider an optimal algorithm, based on an Integer Linear Programming (ILP) solver, and several heuristic algorithms to solve the feature placement problem.

5.1 Integer Linear Programming (ILP)

The formulation, discussed in the previous section, can be used to define an ILP. This program can be solved using a commercial ILP solver, and yields the optimal problem solution using Simplex and Branch and Bound algorithms. As the model contains integer values, the ILP algorithm can not be run in polynomial-bound execution time. Therefore, we will define heuristic algorithms that approximate the optimal solution generated by the ILP solver.

5.2 Heuristic Algorithms

We first define a single meta-heuristic, consisting of two recursive functions: an inner function **placeFeature**, placing individual features and a **place** function that does the actual feature placement. The meta-heuristic as we define it makes use of four functions that are left open. We then present different approaches for filling in these functions. The combination of the algorithm with different function

Table 4 The different functions used in Sect. 5.2

Function	Description
place	This recursive function forms the main part of the feature placement algorithm, and is responsible for placing a collection of features on a collection of servers
placeFeature	This function is responsible for placing a single feature on a collection of servers
featureConversion	A function used to fill in open variation points in feature models
groupStrategy	A function determining whether an application features are placed at once or in multiple steps
featureOrder	Determines the order in which features or applications are placed
serverOrder	The order in which servers are considered during placement

implementations can be used to define different algorithms with varying performance and properties. The different functions used in this section are shown in Table 4.

Algorithm 1, describes the **placeFeature** function, responsible for the placement of a single feature. As input, this function requires different parameters: (1) the problem configuration *P*, containing all the input variables of the formal problem formulation, (2) the instance count matrix $IC_{f,s}$, which contains the number of instances of a features each server has, (3) the placement matrix $M^{\gamma}_{s,f,a}$, which specifies the amount of resources allocated to a feature and application on a server, (4) the feature *f* that must be placed, and (5) a list *Servers* containing all the servers in the system and their remaining resource capacities.

```
Data: problem $P$
Data: Instance Count for a feature on a server $IC_{s,f}$
Data: current placement matrix $M^{\gamma}_{s,f,a}$
Data: a feature $f$ of an application $a$ to place
Data: list of servers with remaining resources $Servers$
sort $Servers$ using serverOrder;
$s \leftarrow$ findServer($f, Servers$);
if no s found then
    return $\emptyset$;
else
    if no remaining capacity for $f$ on $s$ then
        $IC_{s,f} = IC_{s,f} + 1$;
    end
    Update $M_{s,f,a}$ for all resource types;
    Adjust remaining resources on $s$;
    if all non-strict resource demand placed then
        return ($IC, M, Servers$);
    else
        Update $f$, decreasing its resource demand;
        return placeFeature($P, IC, M, f, Servers$);
    end
end
```

Algorithm 1: The **placeFeature** function used by the algorithm.

The algorithm sorts the list, based on a given **serverOrder**, and uses a **findServer** operation to find the first server *s* in the sorted list on which either a feature instance exists with remaining free space, or on which enough resources remain to create a new instance of the feature. In the latter case, a new instance is created. The **serverOrder**, which determines the order in which servers are considered, is essential for the performance of the algorithm, and will be elaborated on later on in the article. Subsequently, the maximum amount of resources possible, taking into account instance resource limitations, are allocated for the feature that is to be placed, by adding them to $M^{\gamma}_{s,f,a}$. The server information of *s* is also updated, to reflect the decrease in available resources on the server. If the entire feature *f* is placed, the updated allocation *IC* and *M* is returned, along with the updated server list *Servers* are returned. If the feature is not fully placed yet, the **placeFeature** function is repeated recursively, and is given as an argument the residual demand of feature *f*. The **placeFeature** function will always either return a placement where the feature is placed in its entirety, or not placed at all.

The main body of the heuristic is listed in Algorithm 2, which displays the **place** function. The function is responsible for placing a list of applications or features. It requires five parameters: (1) the problem model description *P*, (2) the instance count *IC*, (3) the current placement matrix *M*, (4) a list *Servers*, containing all the servers, (5) a list *AppFeatures*, of which every entry is either an application or a feature, and (6) a collection *Failed* containing the applications for which the placement of the application as a whole has failed. The first four parameters are also used for the **placeFeature** function. The fifth parameter determines the order in which features and applications are added, and makes it possible to place features as either applications, or as individual instances. The sixth parameter maintains a list of applications and features that could not be placed successfully.

The formulation of the **place** function makes use of two additional functions:

1. The **dependingFeatures** (*f*) function returns the set of all features that depend on the feature *f*. All the features present in the subtree with root *f* of the feature model tree, except the feature *f* itself, are included in this set.
2. The **dependentFeatures** (*f*) function is the opposite of the previous relation, and returns the collection of all features upon which the feature *f* is dependent. This set can be constructed by, within the feature model tree, selecting the parent feature of *f*, and subsequently recursively adding all of the parent features of the features present in the set.

The **place** function starts by choosing the first element of the *AppFeatures* list. If this element is a feature, it first checks whether the feature should be added. If any feature upon which the feature depends has already failed to be placed for this application, the selected feature is not placed, as it would automatically fail because of the cascading of failure constraint described in Eq. (11). If the application has already failed to be placed, and there is no additional cost for the failure of this feature or any of the child features, the feature is not placed either. This rule is added, as placing these features would increase both the load on the system and the cost of used servers, without decreasing the cost of failed placement. If neither condition is met, the algorithm continues by using the **placeFeature** function to place the feature on the infrastructure. The **place** function is then repeated with the remaining elements of the *AppFeatures* list and, if the feature was correctly placed, the server configuration

```
Data: problem $P$
Data: Instance Count for a feature on a server $IC_{s,f}$
Data: current placement matrix $M^{\gamma}_{s,f,a}$
Data: list of servers with remaining resources $Servers$
Data: list of applications and features to place $AppFeature$
Data: collection of failed application placements $Failed$
if $AppFeature$ is empty then
    return $(IC_{s,f}, M^{\gamma}_{s,f,a}, Servers)$
else
    $fa \leftarrow$ take first element of $AppFeature$;
    $AppFeatures' \leftarrow$ tail of $AppFeature$ list;
    if $fa$ is a feature $f$ of an application $a$ then
        $failedDependent \leftarrow Failed \cap$ dependentFeatures$(f)$;
        $failCost \leftarrow C^V(f,a) + \sum_{f' \in \mathbf{dependingFeatures}(f)} C^V(f',a)$;
        if $failedDependent \neq \emptyset$ then
            Do not place feature;
        else if $a \in Failed \wedge failCost = 0$ then
            Do not place feature;
        else
            $(IC', M', Servers') \leftarrow$ placeFeature$(P, IC, M, Servers, f)$;
        end
        if feature $fa$ placed then
            place$(P, IC', M', AppFeature', Servers', Failed)$;
        else
            place$(P, IC, M, AppFeature', Servers, Failed \cup fa)$;
        end
    else if $fa$ is an application then
        $features \leftarrow$ features of $fa$;
        sort $features$ using featureOrder;
        $(IC', M', Servers') \leftarrow$ place$(P, IC, M, features, Servers)$;
        if $fa$ is correctly provisioned then
            place$(P, IC', M', features, Servers', Failed)$;
        else
            $AppFeatures' \leftarrow AppFeatures' + features$;
            sort $AppFeatures'$ using featureOrder;
            place$(P, IC, M, AppFeatures', Servers, Failed \cup fa)$;
        end
    end
end
```

Algorithm 2: The **place** function.

returned by the **placeFeature** function. Otherwise the initial server configuration is reused.

If the head element of the *AppFeatures* list is an application, the list of all features in the application will be determined, and this list will be placed by recursively calling the **placeFeature** function. If this succeeds, and all the features of the application can be placed, the algorithm will continue by processing the tail of the *AppFeatures* list. If this fails, the changes are undone, and the individual features of the applications are added to the *AppFeatures* list, which will be sorted again, and then placed using a recursive call to the **placeFeature** function. This ensures that, if an application can not be placed in its entirety, the algorithm will still make an effort to place individual application features. As the cost of failing to place the application is incurred by this, only features that further add to the cost of failure will still be considered for placement.

Algorithm 3 shows how the initial parameters are generated, and contains the complete feature placement algorithm.

```
Data: problem $P$
$features \leftarrow$ featureConversion($P$);
$list \leftarrow$ groupStrategy($P, features$);
$AppFeature \leftarrow$ sort $list$ using featureOrder;
$Servers \leftarrow$ sort servers in $P$ using serverOrder;
$IC_{s,f} \leftarrow 0$;
$M^{\gamma}_{s,f,a} \leftarrow 0$;
$Failed \leftarrow \emptyset$;
execute place($P, IC, M, AppFeature, Servers, Failed$);
```

Algorithm 3: The feature placement algorithm.

The algorithm contains four components that we have not yet elaborated on. At the start of the algorithm, the open variation points of the feature model are filled in using a **featureConversion** function. This function ensures that for every application, all features are either selected or excluded, eliminating open variation points. The **groupStrategy** is used to determine whether all the application features should be considered as a whole, or whether they should be placed independently. The result of this function is a list containing a mix of features and applications: the *AppFeature* list. The **featureOrder** is used to sort the *AppFeature* list, and can compare features and applications to determine the order in which they are placed. Finally, the order in which servers are considered is determined by the **serverOrder** function. The effectiveness of the meta-heuristic is largely determined by the **featureConversion**, **groupStrategy**, **featureOrder**, and **serverOrder** functions. We will now present different implementations for these functions.

5.2.1 Feature Ordering

The order in which features are considered significantly impacts the quality of the final result, as it determines which features are placed first, and thus assigns a priority to the features. We make use of an application-based ordering, where applications and features with a higher cost of failure are placed first. For applications, we define the cost of failure as the sum of feature failure costs, and the application failure costs. For features, we define this cost as the sum of the cost of failure of the feature, the application, and the cost of failure of all features dependent on the feature. When according to this ordering no preference is achieved, we consider the number of instances required to place the feature or application. The instance requiring the smallest number of instances is placed first.

5.2.2 Grouping Strategies

As explained above, the list *AppFeature* can contain either entire applications, individual application features or a combination of both. The **groupStrategy** function determines for each application present in the problem definition whether it must be considered as a whole, or as a group of features. We consider two versions:

- Feature grouping, where every application is split up into features, and the features are placed independently. This corresponds to the approach we previously described in [7].
- Application-based grouping, where applications are always grouped, their features are placed at the same time. Should the placement of an application fail, the algorithm will still try to place individual features, as described in the Algorithm 2.

It is important to note that in both cases, the algorithm will place multi-tenant feature instances, and allocate part of their capacity to the placed applications. Using application-based grouping, the algorithm will however start by trying to place all of the feature instances of a given application at once.

5.2.3 Server Ordering

We consider two different server orderings:

- Instance Based (IB) ordering, which orders servers according to the best fit for the feature *f* that is to be placed. This ordering prefers servers that have instances of the feature placed on it, that are not fully utilized by the current allocation. If multiple servers comply, the server with the best fit will be selected. If, using this approach, two servers score the same, the server with the lowest utilization cost is used.
- Cost Based (CB) ordering, where servers are ordered according to their utilization cost.

Note that the IB ordering of nodes changes for every invocation of the place method, whereas the CB ordering does not change. This ensures the sort in Algorithm 1 does not have to be executed for the CB ordering. Both of the approaches take the cost of using servers, C^U, into account, but only in the IB case is it the primary selection criterion.

5.2.4 Feature Model Conversion

The **featureConversion** function is used to fill in open variation points. This function determines the features that must be included in the placed applications. We make use of an approach in which the cheapest feature combination in terms of resource requirements is determined in two steps. First, ten cheap combinations of feature models are determined for every application. As the number of combinations increases exponentially, at each point in time the list of possibilities is shortened. We have shown before that shortening to ten elements is sufficient for improving the placement [7]. This can be determined as soon as an application is added, rather than when application placement is executed. Within the evaluation section, we will refer to this as the preparation step of the algorithm.

Secondly, when the list of all applications is known, the best total configuration is determined. This is done by incrementally iterating all applications, and creating a partial list containing a the best configuration for the subset of applications that has already been considered. In every step, an application is added, and each of its ten feature combinations is combined with the list of best applications from before. From the resulting collection, the ten best elements are retained and passed on to the next iteration.

5.2.5 Heuristic Algorithms

The described functions can be combined with the meta-heuristic to create different algorithms. For this article, we use the two grouping strategies, *feature* and *application* based, and the two server orderings, *IB* and *CB*. This creates four algorithms: **IB_application** , **IB_feature** , **CB_application** and **CB_feature** .

6 Evaluation Setup Details

We implemented the ILP problem and the heuristics using Scala. The ILP solver uses CPLEX [35] as its back-end. Within the evaluations, we will make use of two types of problem models: (1) problem models based on the three real-life applications studied in the CUSTOMSS project, and (2) problems created using a generator capable of creating a wide range of random problems.

6.1 CUSTOMSS Problem Model

The full model, used by the CUSTOMSS project, is shown in Fig. 6, and contains the features and relations as they are currently defined in the project. The feature names have been replaced by numbers. Each feature entry also contains an estimated CPU requirement and a CPU use limitation (in the form *CPU* = *requirement*/*limit*), and an instance memory requirement (*Memory* = *requirement*). The relations between features are expressed in the format as described in Sect. 3. As discussed earlier, features can impact each other. This is illustrated by the arrows between nodes, the number on the arcs represents the impact on CPU requirement other nodes. For example, the addition of Feature 7 increases the CPU demand for Feature 1 by 100.

The presented model groups the models for the three real-life applications, with application roots Feature 1, Feature 16 and Feature 28 into a single model by adding a new root node, modeling a cloud that executes these distinct applications. As the nodes are grouped using an *Alternative* relation, every application will be an instance of exactly one of the CUSTOMSS applications. This approach for running multiple distinct applications was discussed previously in Sect. 4.2. Note that some of the features do not have any CPU or Memory requirement. These features are either used for grouping other features, improving the structure of the complete feature model, or as they do not create new feature instances but significantly impact

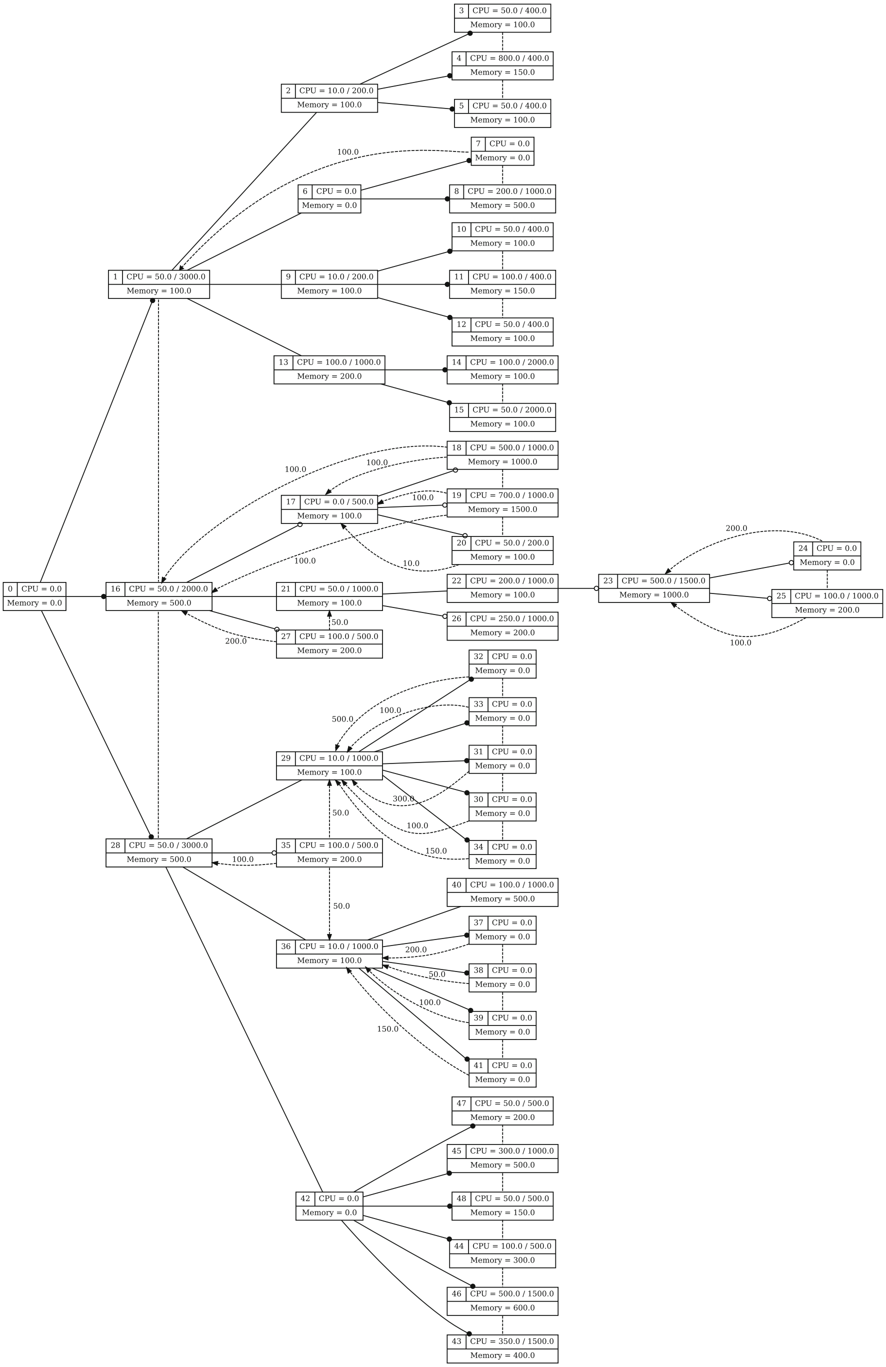


Fig. 6 The combined feature model containing the CUSTOMSS applications. Each entry in this model corresponds to a feature in one of three real-life applications

the demand for other features. When these features are included, they are automatically satisfied, provided that their parent nodes are correctly provisioned.

Applications feature selections are randomly generated by creating random valid selections, where all features are either selected or excluded. Open variation points are then randomly added. The application failure cost is set to 10. Some features are selected in the feature model and are considered as being critical: if they are selected, they must be correctly provisioned, or an additional cost of 5 will apply. The features in the model that can be considered as critical are Features 13, 21, 22, 26 and 40. The energy cost is chosen as 1. This ensures applications will always be placed if possible, the desired behavior, and that, if an application does fail, only its critical features are placed.

In practice, a realistic cost could be determined by utilizing the actual economical cost of failure of applications. This cost would however vary throughout time, based on previous placement performance. Practically, it is better to assign relative costs that are maintained by the management system. In general, the cost of failure of applications is always bigger than the cost of using servers, and specific features exist that significantly increase the cost, on top of application placement failure cost, if they fail to be placed.

6.2 Generated Problem Models

To evaluate the algorithms for differing problem sizes and varying features models, we generated different problem models. These models are similar in structure to those of the applications studied in the CUSTOMSS project, but the number of features in the feature models can be varied.

The generator creates a collection of servers, a feature model, and a set of applications. For the purposes of the evaluation, we use $\Gamma_{\bar{s}} = \{\text{CPU}\}$ and $\Gamma_s = \{\text{Memory}\}$. First, the servers S are generated. For these evaluations we assume a uniform server configuration with 4,000 MiB memory and a 2,000 MHz processor. We also use a uniform server use cost of 1. The costs of failure are chosen relative to this cost.

To create a random feature model $\mathscr{F}$, first, a collection of features F is generated with random memory requirements from a set $\{0.5, 1, 2, 2.5\ GB\}$. Subsequently a feature model tree $\mathscr{R}$ is created. This is done by iteratively selecting nodes that are not in the tree yet and adding them in a relation with a node in the tree as the parent node. To start this process, a random feature is selected as root of the feature tree. There is an equal chance of picking any of the four relation types, and *Alternative* and *Or* relations have between two and six child nodes. Feature models generated in this fashion are similar in structure to those used for the applications in the CUSTOMSS project.

Next, we generate the impact matrix FI^{CPU}. Each feature impacts itself and has a chance of impacting any feature required by it. This is enforced by only letting a feature impact parent features. The CPU impact of a feature on itself is randomly chosen from the set $\{100, 200, 500, 1,000\ MHz\}$, the CPU impact of a feature on a parent feature is added with a probability of 50 %, and chosen randomly from the same set. As stated earlier, we assume a homogeneous host capacity.

Selecting features is done by randomly selecting or excluding features, and checking the validity of the resulting feature model with SAT4J [36], an open source SAT solver. This ensures that the selection is feasible according to feature model $\mathscr{F}$. Features are randomly removed from either the collection of selected features, or from the collection of excluded features. All dependent features are removed as well, ensuring an open variation point is added.

Finally, random applications A are generated using the generated feature selections. Each application and application feature is also assigned costs for failure, randomly chosen from a given set. We use four different scenario's, shown in Table 5 for the evaluations, each with a different application failure cost. The Varying Costs (VC) scenario makes use of varying costs for both application and feature failure, and represents the realistic case where the failure of some applications or features can incur a much larger cost than the failure of others. The Identical Costs (IC) scenario by contrast only considers a single cost for both application and feature failure. Finally, the Application Costs (AC) and Feature Costs (FC) scenarios consider situations in which either only application failure, or only feature failure are considered. Costs are defined relative to each other, the VC scenario representing the case where the costs of different applications differ by a large order of magnitude.

6.3 Evaluation Methodology

We will now discuss the different evaluation strategies, and the different quality metrics used in these evaluations.

6.3.1 Load-Based Evaluation

We use a large number of randomly generated problem models in our evaluations. As each of the randomly generated problem models can have very different properties, we need a common parameter to represent the difficulty of finding a good solution. For this, we use the *problem model load*. The problem model load is determined by filling in the feature model for every application, and determining the cheapest possible application. We sum the CPU load for all features and all applications, to determine the total application demand. We then divide this by the sum of all available resources. This variable is indicative of the problem difficulty, as higher load values imply that it becomes more difficult to place all applications.

Table 5 The costs for the different evaluation scenarios 4

Scenario	Application failure costs	Feature failure costs
Varying Costs (VC)	{2, 4, 8, 16, 32}	{2, 4, 8, 16, 32}
Identical Costs (IC)	{2}	{2}
Application Costs (AC)	{2}	{0}
Feature Costs (FC)	{0}	{2}

Load-based evaluation of feature placement algorithms is done by first generating a large batch of problem models: a model is generated for every value of $(s, a, f) \in \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}^3$, thus creating 1,000 problem models. We subsequently removed all problem models with a load >3. In these cases, it would be preferable to filter applications using admission policies, such as those described in [37], in the management system, ensuring some applications are not accepted by the system.

Due to the nature of ILP solvers, some problems require large amounts of memory or computing time. Additionally, the CPLEX solver allows slight constraint violations, in the order of 10^{-9}, making the solutions to a minority of the problem models violate constraints when the values are rounded. In both cases, the models causing problems are excluded from the test. The placement quality comparisons were performed using the the Stevin Supercomputer Infrastructure at Ghent University, a hardware cluster containing quad core Intel Xeon L5420 nodes with 16 GB ram. This ensures almost no problem models are filtered due to resource constraints.

For the each of the evaluations in this article, we repeat this process three times, retaining on average 150 entries for every test set, most being excluded due to the load limitations.

6.3.2 Execution Time Evaluations

The execution speed evaluations of the algorithms were executed on a Linux server with a Dual-Core AMD Opteron(tm) Processor 2212 with 4GiB of memory, and using Scala version 2.9.0.1. For these evaluations, the different versions of the algorithm are executed for varying server, application and feature count.

6.3.3 Quality Evaluation Metrics

The results of a placement can be evaluated in different ways:

- Cost of Non-Realized Demand (NRD): This metric measures the cost caused by the failure to provision applications. It corresponds to the C_D variable in the formal model, defined in Eq. (2).
- Cost of Non-Realized Demand Simple (NRDs): This measure is similar to NRD, but does not take cascading failure of features into account.
- Full: Measures the total cost of using servers. This measure corresponds to the C_U variable, defined in Eq. (1).

7 Evaluation Results

First, we evaluate the feature-based approach by comparing the degree of multi-tenancy that can be achieved compared to an approach where every variant would be provisioned its own instance. We then evaluate the impact of some of the design decisions, determining the maximal amount of quality that can be achieved by the placement when the various constraints are taken into account, and the importance of including these constraints. Then, we evaluate the placement quality of the algorithms compared to an optimal solution, and finally we evaluate the execution speed of the algorithms.

7.1 Degree of Multi-Tenancy

As discussed earlier, there are two approaches to build high-variability applications in clouds: (1) by generating a binary application for every variant and (2) by splitting the application into separate components, which we have referred to as feature instances. In the former case, tenants can only share an application instance if they require the same variant, in the latter case, individual feature instances are shared by tenants. Note that the application-based grouping in Sect. 5.2.2 still makes use of the second approach, and ensures that the different feature instances are considered at the same time during placement.

To compare the degree of multi-tenancy that can be achieved using an application instance approach to that of a feature based approach, we use the CUSTOMSS problem model and count the number of instances that make use of the same application variant for the former, and the number of applications that make use of the same feature for the latter.

By counting the number of identical applications appearing within the problem models used in the next subsections, we can determine how many tenants make use of the same application. We then sort these values, showing frequently used applications first, and average the results over the different problem models used in the CUSTOMSS model evaluation, which will be discussed more in depth later on in Sect. 7.3.2. The results, shown in Fig. 7a, show that in average problems, 100 different applications must be provisioned that are each used by 5–30 tenants. Many of these instances share less than 10 tenants.

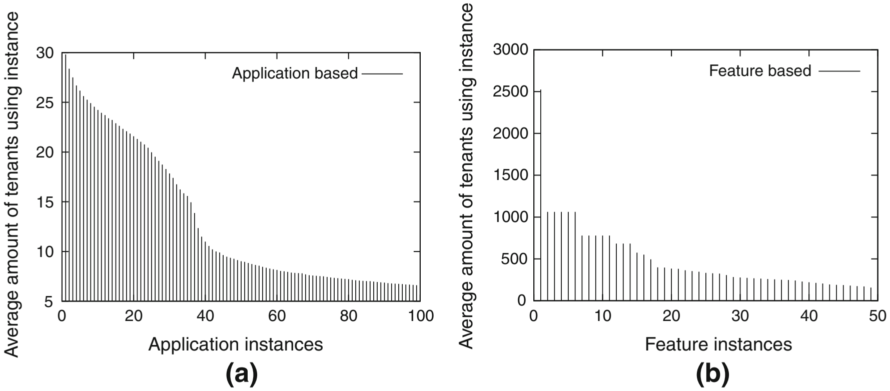


Fig. 7 The maximum number of tenants sharing instances for an application-based approach, where variants are created as monolithic instances, and a feature-based approach, where applications are composed from feature instances. **a** Using application instances. **b** Using feature instances

When the focus is shifted from application instances to feature instances, and we count how many applications require specific features, we see that a much higher degree of multi-tenancy can be achieved, as seen in Fig. 7b. Only 49 feature instances are needed, and each instance is shared between 150 and 2,500 different tenants.

Our more fine-grained approach where applications are composed using multi-tenant services, thus increases the achievable level of multi-tenancy while at the same time decreasing the number of different instances that must be provisioned.

7.2 Impact of the Model Constraints

We will now determine the impact of various constraints that are defined in the model on the maximum amount of quality that can be achieved by the optimization algorithm. Compared to our earlier work [7], three additional constraints have been added: (1) resource limits, expressing the limited amount of resources that can be used by single application instances, (2) the cascading failure of features, which expresses that parent features, upon which the feature relies for its correct execution, need to be correctly provisioned for the feature to be allocated, and (3) the consideration of server usage costs. Each of these additional constraints will have an impact on the complexity of the problem, and on the minimal achievable cost.

We consider three variants of the ILP formulation:

- The first formulation, ILP Simple (ILPs) represents a simple variant of the ILP formulation, where no resource limits, energy requirements or cascading failure are taken into account.
- ILP Requires Parent (ILPrp) is a variant of the ILP formulation that adds the cascading failure of applications, but not the energy requirement of servers nor the resource limitations.
- ILP Requires Parent Limited (ILPrpl) is a variant of the ILP formulation, considering both resource limitations and cascading failure of features.

We will now evaluate the impact of these requirements on the quality of the resulting feature instance allocations using a load-based evaluation using randomly generated problem models. We will do this by comparing the algorithms using the two evaluation functions, NRD and NRDs. The results of these evaluations are shown in Fig. 8a, b.

In Fig. 8a we show the performance of the three variations of the ILP solution with respect to NRDs metric. The addition of the different constraints increases the number of applications that fail. Introducing the cascading failure of features greatly increases the cost of placing applications. This is to be expected, as constraints are added to the ILP formulation that complicate placement, but that are not taken into account by the NRDs evaluation function. Adding resource limitations further increases the cost, as more instances are required to meet the required demand.

When we add the effect of cascading failure in the evaluation, and measure the performance using NRD, the performance of the different ILP formulation changes drastically, as shown in Fig. 8b. Here, the ILP solution performs badly, which is again to be expected as it allocates features with a high cost of failure, without

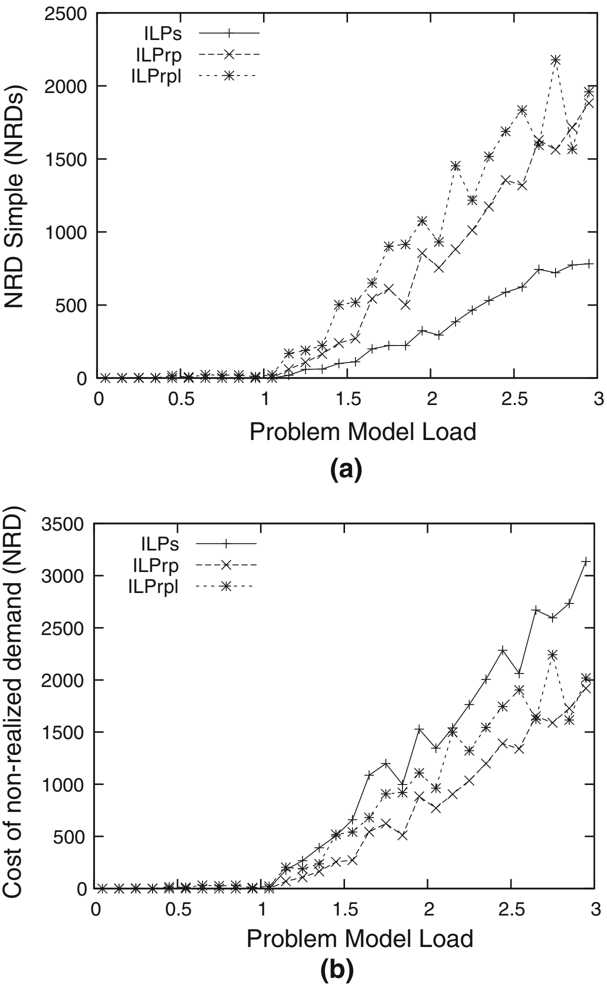


Fig. 8 Performance of ILPs, ILPrp and ILPrpl using different quality evaluation metrics using the VC scenario. **a** Performance of ILPs, ILPrp and ILPrpl using the NRDs metric. **b** Performance of ILPs, ILPrp and ILPrpl using the NRD metric

taking into account whether the parent features are correctly allocated. The performance of both ILPrp and ILPrpl remains identical for both evaluation mechanisms, as no new failed features are introduces.

From this evaluation we can conclude that the use of cascading failure and resource limitations comes at a cost, making it more difficult to find a satisfactory placement on given infrastructure as more requirements are taken into account. This disadvantage is in addition to the increased computational cost incurred by these constraints. If, however, the constraints are required for an accurate representation of the application, they must be considered during placement, as these results show that otherwise the quality of the eventual placement result will be significantly worse.

7.3 Placement Quality

We evaluate the placement quality using both the load-based approach as discussed in the previous section, and the CUSTOMSS model.

7.3.1 Generated Problem Models

We use the load-based evaluation with generated problem models for the different scenarios which we defined in Table 5, and evaluate the performance of the algorithms using the total cost evaluation function (Full) defined earlier. Figure 9a shows that in the Varying Costs (VC) scenario with varying costs, the **IB_application** and **CB_application** algorithms perform significantly better than the **IB_feature** and **CB_feature** algorithms. This indicates that an application-based feature grouping strategy performs well in practice. In both cases, the IB server ordering strategy performs slightly better than the CB approaches, but these differences are less significant.

As shown in Fig. 9b, the application-based approach works remarkably well when both features and applications impact the cost of placing features, as we do in

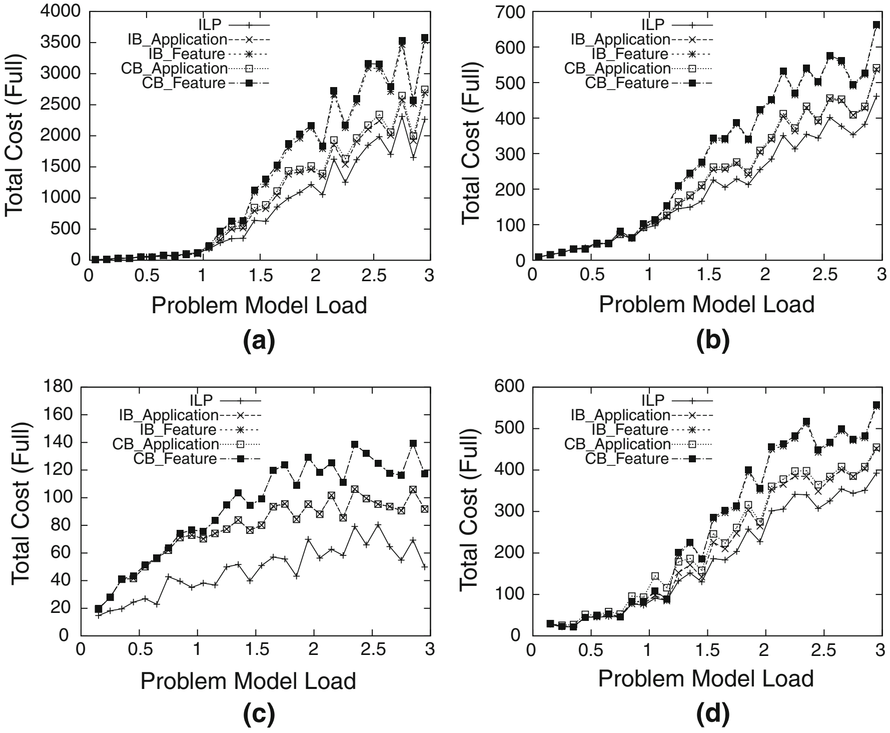


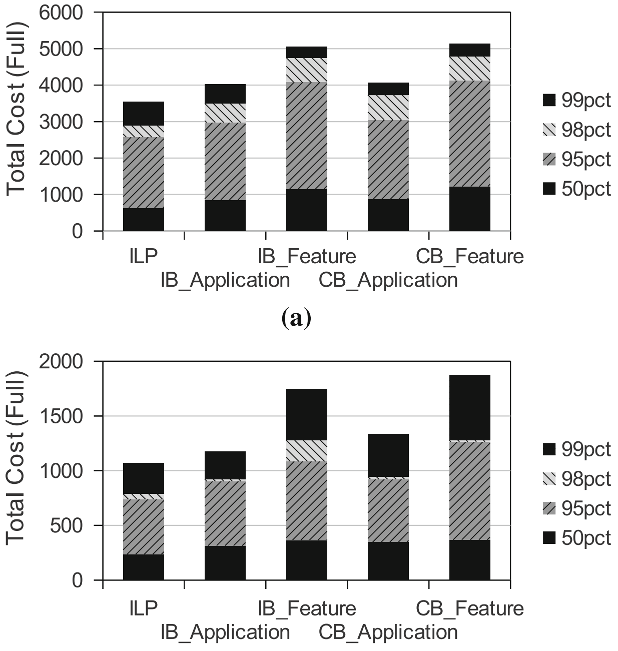
Fig. 9 Performance of the heuristics and the ILP algorithm for the different scenarios. **a** Varying Costs (VC). **b** Identical Costs (IC). **c** Application Costs (AC). **d** Feature Costs (FC)

the IC scenario, even if these costs are kept constant, yielding significant improvements when compared to a purely feature-based approach.

The results for the AC scenario are shown in Fig. 9c. In this scenario, the different algorithm variations all perform more or less the same, and significantly worse than the ILP results. While this may seem counterintuitive, considering the feature placement focuses on applications in their entirety, this is not unexpected: this phenomenon is caused by the homogeneous nature of the different applications, which makes each application equally important in the placement, making it difficult to decide on which applications to exclude.

Even when no costs for application failure are taken into account, as in the FC scenario, an application-based approach again performs best, as we show in Fig. 9d. It is however to be noted that the combination of a purely CB approach for servers, combined with an application-based approach for grouping, can sometimes result in bad performance, as seen in the [1, 1.2] region of the plot.

As explained previously, these evaluations make use of randomly generated problem models. While a load-based approach groups elements according to their difficulty, variations in feature models can still cause large differences in the eventual quality of the placement. We show the percentiles for the VC scenario in Fig. 10. Figure 10a shows the percentiles for the entire test set, with loads between 0 and 3. Figure 10b shows the percentiles for the problem entries with loads in [1,



(a)

(b)

Fig. 10 Percentiles for the performance of the heuristic and ILP algorithms for the VC scenario. **a** Using all datapoints. **b** Using datapoints with load between 1 and 1.4

1.4], an interesting region as it represents a situation that could potentially occur when application demand spikes. In both charts, the tendencies explained previously reoccur: an application-based approach performs better than a feature-based approach, and an instance based order for servers performs better than a purely cost based approach. On average, the application-based approach performs ±25 % better than a feature-based approach, in the [1, 1.4] region, while when all results are considered, this difference increases to ±42 %. Using a cost-based approach rather than a feature-based approach also slightly increases placement quality, by ±12 in the [1, 1.4] range, but only by a negligable ±1 % when all evaluation results are included.

It is noticeable that, globally, the instance-based approach has a similar worst-case performance as the cost-based approach, but its occurs less often. In an overload situation, the difference between both approaches is only noticeable in the 99th percentile. It is of note that, for loads in the [0.5, 1] range, not shown here, the different algorithms often perform slightly better than the ILP-based algorithm due to the assignment of non-integer values to integer variables that occurs in CPLEX.

The results in this section demonstrate that an application-based approach to feature placement, where the algorithm tries to place all of the features of applications at once, performs significantly better compared to a feature-based approach, where the features are considered separately. Note that in both cases, the placed feature instances are multi-tenant services and shared between applications, as discussed previously. When servers are selected, it is best to take into account how well applications fit on the server, as it is done with the with the IB approach, but the improvements of this choice compared to a purely cost-based approach are limited, and this change only impacts problem models in the 98th and 99th percentiles.

7.3.2 CUSTOMSS Problem Model

Using the CUSTOMSS model, we assessed the costs when the number of servers is varied using the total cost evaluation function (Full) and using the **CB_application** algorithm. Figure 11 shows the quality of placement considering varying application and server counts. These graphs were generated by using the CUSTOMSS feature model, and creating applications as described in Sect. 7. To generate the problem for n applications, a single application is generated and added to the problem generated using $n - 1$ applications. Because of this, the graph shows the impact of iteratively adding applications and servers to a cloud.

We see that as applications are added, the cost for failed placement increases. The plot consist of two intersecting planes. One plane is nearly flat, with only a slight slope as application counts increase, as this ensures more servers that need to execute applications, increasing the server use cost. The second plane shows a steeper increase in costs, as in these points application failure occurs, incurring a larger cost. The intersection of the planes shows the point at which too many applications are allocated, and a cost of failure is incurred.

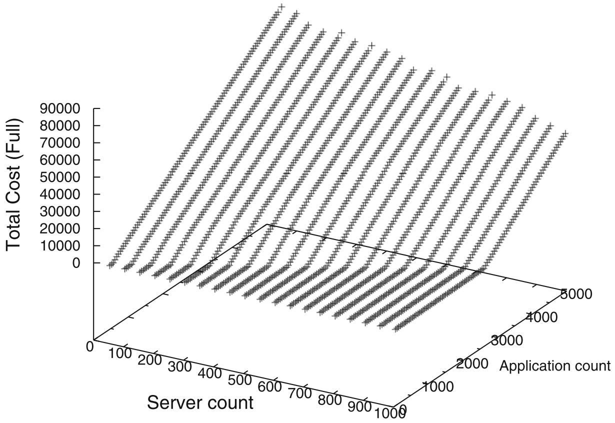


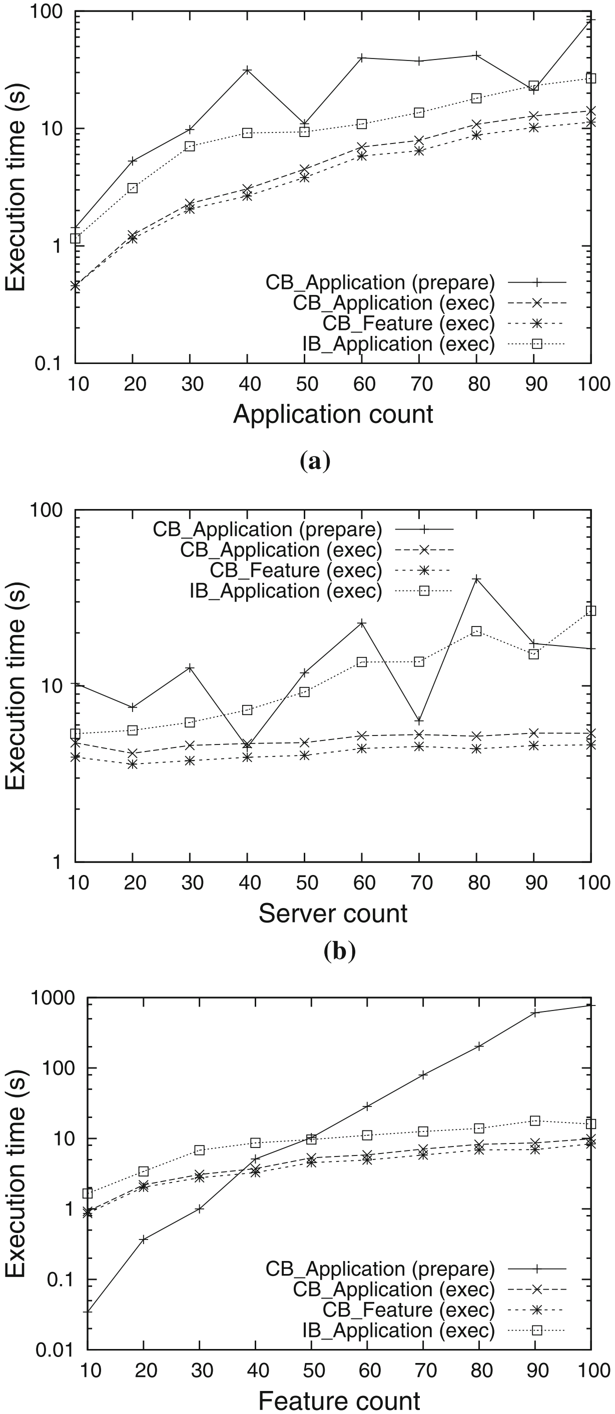
Fig. 11 The quality of feature placement as a function of the number of servers and applications

7.4 Execution Speed Evaluation

We consider the execution speed of the algorithm for increasing application counts, server counts and feature counts, using randomly generated problem models. In the graphs, we separate the total execution time of the algorithms into two parts: a preparation time and an execution time. Each data point is an average of 20 executions using randomly generated feature models with the parameters discussed in the previous section. The preparation time is the part of the computation that can be executed when an application is added, and is needed only once. This mainly comprises of the time required to create feature model configurations using the feature model conversion as discussed in Sect. 5.2.4. We only show the preparation cost for the **CB_application** algorithm, but these costs are identical for the other three algorithms. The execution time is the time required to execute feature placement, provided the preparation step has been executed in advance. This step must be executed when applications are actually running, to take changing application demands and the addition of new applications into account.

As shown in Fig. 12a, increasing the number of applications increases the execution duration of the algorithm in a more or less linear fashion. It is notable that the application-based approach performs a bit worse than a feature-based approach, as some applications will be considered twice by it, once in an application-based fashion and once in a per-feature order, while the feature-based approach only considers each feature once. Similarly, an IB approach to server ordering also increases the execution duration w.r.t. a CB approach, as it requires an additional sort operation. As the number of applications doubles, so does the preparation duration, as this preparation runs for each application. Such a trend is noticeable in the plot, but the preparation duration does vary significantly from problem model to problem model.

In Fig. 12b, we show the performance of the algorithm in the face of varying server counts. Once more, the high variability in preparation execution time is demonstrated, as for each data point the same number of applications are


Execution time (s)
100
10
1
0.1
10 20 30 40 50 60 70 80 90 100
Application count
CB_Application (prepare)
CB_Application (exec)
CB_Feature (exec)
IB_Application (exec)
Execution time (s)
100
10
1
10 20 30 40 50 60 70 80 90 100
Server count
CB_Application (prepare)
CB_Application (exec)
CB_Feature (exec)
IB_Application (exec)
Execution time (s)
1000
100
10
1
0.1
0.01
10 20 30 40 50 60 70 80 90 100
Feature count
CB_Application (prepare)
CB_Application (exec)
CB_Feature (exec)
IB_Application (exec)

(a)

(b)

(c)

◀**Fig. 12** The execution speed of the feature placement algorithm as a function of varying application counts, server counts and feature counts. **a** Varying application counts, using $|S| = 50$ and $|F| = 50$. **b** Varying server counts, using $|F| = 50$ and $|A| = 50$. **c** Varying feature counts, using $|S| = 50$ and $|A| = 50$

considered. We notice that the CB algorithms are largely independent from the number of servers considered. This opposed to the IB approach, for which the required computational time increases with the server count.

Figure 12c demonstrates the execution speed considering varying feature counts. Here we notice a significant increase in preparation time as feature counts increase, but only limited impact on the execution time of the algorithms. We again observe that an application based approach requires more time to execute, as do the IB algorithms.

Feature models become more complicated to manage as the number of features increases, especially as within this article we only consider customization changes, in which a change implies the use of a different code module that must be maintained. Because of this, we do not expect the models to become prohibitively large, ensuring the preparation duration will remain acceptable. Furthermore, this has little impact on the execution time of the algorithm. We can conclude that the CB algorithms scale well in terms of application and server counts, and that, due to the possibility of preparing applications before execution, increasing feature counts can be managed as well. The IB algorithms do not scale as well when server counts increase as the CB approach. This implies that, while the IB algorithms perform slightly better than the CB algorithms, it can be preferable to make use of the latter in large server configurations to improve the speed with which placements can be determined. The presented algorithms still make use of a centralized approach, implying they could become a bottleneck as the size of the cloud increases. To address this, techniques such as those we presented in [28] could be used to increase the scalability of the algorithms in larger clouds, by reusing the centralized algorithms within a hierarchically structured management infrastructure.

8 Conclusions

In this article, an approach for managing highly customizable applications using feature modeling and SPLE techniques was presented. We first presented the feature placement problem, determining the different inputs, outputs and requirements, which we subsequently formalized. Then, heuristics were developed and compared to the optimal ILP-based algorithm. In this evaluation we used the feature models from existing applications, ensuring the presented techniques are applicable to realistic cases, and using generated feature models, ensuring the performance remains similar for different cases.

For the considered cases, using feature instances rather than application instances greatly increased the achievable level of multi-tenancy. In the former approach, each instance can be shared between up to at least 150, while in the latter approach some instances can only be used to serve ±5 tenants. We found that an application-centric approach to feature placement, where the services corresponding to application features are placed at once, performs 25–40 % better than a feature-based approach, where the features are placed independently without taking their relations into account. We also conclude that an approach where servers are chosen based on a best fit approach performs best, albeit with a penalty to execution times. For three out of four scenarios, the application-based approach to feature placement performs close to the optimal algorithm, failing only when no differences between applications occur. The presented heuristics scale well, with execution times remaining under 10 s for the considered cases.

In future work we will extend the discussed approach to achieve dynamic application placement, and we will incorporate the designed algorithms in a cloud management platform as a proof-of-concept.

Acknowledgments Hendrik Moens is funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). This research is partly funded by the iMinds CUSTOMSS [2] project. This work was carried out using the Stevin Supercomputer Infrastructure at Ghent University.

References

1. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. Future Gener. Comput. Syst. **25**(6), 599 (2009). doi:10.1016/j.future.2008.12.001
2. CUSTOMSS: CUSTOMization of Software Services in the cloud (2013) http://www.iminds.be/en/projects/overview-projects/p/detail/customss. Accessed on 1/2013
3. Pohl, K., Böckle, G., Linden, F.: Software product line engineering: foundations, principles, and techniques. Springer, New York (2005)
4. Mietzner, R., Metzger, A., Leymann, F., Pohl, K.: In: ICSE Workshop on Principles of Engineering Service Oriented Systems, (IEEE, 2009), vol. 215483, pp. 18–25. doi:10.1109/PESOS.2009.5068815
5. Wuhib, F., Stadler, R., Spreitzer, M.: In: Proceedings of the 6th International Conference on Network and Service Management (CNSM 2010), pp. 1–8 (2010)
6. Li, Y., Chen, F.H., Sun, X., Zhou, M.H., Jiao, W.P., Cao, D.G., Mei, H.: Self-adaptive resource management for large-scale shared clusters. Sci. Technol. **25**, 945 (2009). doi:10.1007/s11390-010-1075-6
7. Moens, H., Truyen, E., Walraven, S., Joosen, W., Dhoedt, B., Turck, F.D.: In: Proceedings of the 13th Network Operations and Management Symposium (NOMS 2012), pp. 17–24 (2012). doi:10.1109/NOMS.2012.6211878
8. Tang, C., Steinder, M., Spreitzer, M., Pacifici, G.: In: Proceedings of the 16th international conference on World Wide Web, pp. 331–340 (2007). doi:10.1145/1242572.1242618
9. Zhang, K., Zhang, X., Sun, W., Liang, H., Huang, Y., Zeng, L., Liu, X.: In: 9th IEEE International Conference on E-Commerce Technology & The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (CEC-EEE) (IEEE, 2007), pp. 123–130. doi:10.1109/CEC-EEE.2007.9
10. Sun, W., Zhang, X., Guo,C.J., Sun, P., Su, H.: In: IEEE Congress on Services Part II (services-2) (IEEE, 2008), pp. 18–24. doi:10.1109/SERVICES-2.2008.29
11. Abu-Matar, M., Gomaa, H.: In: Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture (WICSA 2011) (IEEE, 2011), pp. 302–309. doi:10.1109/WICSA.2011.47
12. Abu-Matar, M., Gomaa, H.: In: Proceedings of the 15th International Software Product Line Conference (SPLC2011), pp. 110–119 (2011). doi:10.1109/SPLC.2011.26
13. Ruehl, S.T., Andelfinger, U.: In: Proceedings of the 15th International Software Product Line Conference (SPLC 2011), pp. 16:1–16:4 (2011)
14. Alférez, G.H., Pelechano, V.: In: Proceedings of the 15th International Software Product Line Conference (SPLC 2011), pp. 100–109 (2011). doi:10.1109/SPLC.2011.21

15. Gomaa, H., Hashimoto, K., Kim, M., Malek, S., Menascé, D.A.: In: Proceedings of the 2010 ACM Symposium on Applied Computing (SAC 2010), pp. 462–469. ACM Press, New York, New York, USA (2010). doi:10.1145/1774088.1774185
16. Urgaonkar, B., Rosenberg, A.L., Shenoy, P.: Application placement on a cluster of servers. Int. J. Found. Comput. Sci. **18**(05), 1023 (2007). doi:10.1142/S012905410700511X
17. Adam, C., Stadler, R.: Service middleware for self-managing large-scale systems. IEEE Trans. Netw. Serv. Manag. **4**(3), 50 (2007). doi:10.1109/TNSM.2007.021103
18. Rolia, J., Andrzejak, A., Arlitt, M.: In: Self-Managing Distributed Systems: 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2003) (Springer, 2004), pp. 118–129. doi:10.1007/978-3-540-39671-0_11
19. Whalley, I., Steinder, M.L: In: Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2012), pp. 169–176 (2011)
20. Breitgand, D., Epstein, A.: In: Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011), pp. 161–168 (2011). doi:10.1109/INM.2011.5990687
21. Peoples, C., Parr, G., McClean, S.: In: 3rd IEEE/IFIP International Workshop on Management of the Future Internet (ManFI), pp. 1246–1253 (2011). doi:10.1109/INM.2011.5990573
22. Xu, J., Fortes, J.A.B.: In: 2010 IEEE/ACM International Conference on Green Computing and Communications & International Conference on Cyber, Physical and Social Computing (IEEE, 2010), pp. 179–188. doi:10.1109/GreenCom-CPSCom.2010.137
23. Zhu, X., Santos, C., Beyer, D., Ward, J., Singhal, S.: Automated application component placement in data centers using mathematical programming. Int. J. Netw. Manag. **18**(6), 467 (2008)
24. Carrera, D., Steinder, M., Whalley, I., Torres, J., Ayguadé, E.: In: Proceedings of the 11th Network Operations and Management Symposium (NOMS 2008) (IEEE, 2008), pp. 9–16. doi:10.1109/NOMS.2008.4575111
25. Karve, A., Kimbrel, T., Pacifici, G., Spreitzer, M., Steinder, M., Sviridenko, M., Tantawi, A.: In: Proceedings of the 15th international conference on World Wide Web (ACM, 2006), pp. 595–604. doi:10.1145/1135777.1135865
26. Low, C.: Decentralised application placement . Future Gener. Comput. Syst. **21**(2), 281 (2005). doi:10.1016/j.future.2003.10.003
27. Kimbrel, T., Steinder, M., Sviridenko, M., Tantawi, A.: In: Proceedings of the 4th international conference on Experimental and Efficient Algorithms, pp. 391–402 (2005). doi:10.1007/11427186_34
28. Moens, H., Famaey, J., Latré, S., Dhoedt, B., De Turck, F.: In: Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011), pp. 137–144 (2011). doi:10.1109/INM.2011.5990684
29. Rodero-Merino, L., Vaquero, L.M., Gil, V., Galán, F., Fontán, J., Montero, R.S., Llorente, I.M.: From infrastructure delivery to service management in clouds. Future Gener. Comput. Syst. **26**(8), 1226 (2010). doi:10.1016/j.future.2010.02.013
30. Chapman, C., Emmerich, W., Marquez, F.G., Clayman, S., Galis, A.: In: Proceedings of the 12th IEEE/IFIP Network Operations and Management Symposium Workshops (NOMS 2010) (IEEE, 2010), pp. 327–334. doi:10.1109/NOMSW.2010.5486555
31. pure-systems GmbH, pure::variants User's Guide, version 3.0 edn. http://www.pure-systems.com/Documentation.116.0.html. Accessed on 12/2012
32. Guo, C.J., Sun, W., Huang, Y., Wang, Z.H., Gao, B.: In: Proceedings of the 9th IEEE International Conference on E-Commerce and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007. CEC/EEE 2007, pp. 551–558 (2007)
33. Cai, H., Wang, N., Zhou, M.J.: In: Proceedings of the 6th World Congress on Services (SERVICES-1), 2010, pp. 40–47 (2010). doi:10.1109/SERVICES.2010.48
34. Hajjat, M., Sun, X., Sung, Y.W.E., Maltz, D., Rao, S., Sripanidkulchai, K., Tawarmalani, M.: Cloudward bound: planning for the beneficial migration of enterprise applications to the cloud. SIGCOMM Comput. Commun. Rev. **40**(4), 243 (2010)
35. IBM ILOG CPLEX 12.2. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer (2011)
36. SAT4J 2.2.2. http://www.sat4j.org (2011)
37. Leontiou, N., Dechouniotis, D., Denazis, S.: In: Proceedings of the 6th International Conference on Network and Service Management (CNSM 2010), pp. 318–321 (2010). doi:10.1109/CNSM.2010.5691214

Author Biographies

Hendrik Moens received a Masters degree in computer science from Ghent University in 2010. He is a Ph.D. student in the Department of Information Technology of the Ghent University, and a member of the network and service management research group. His research interests include the management of large scale cloud computing environments and autonomic management systems.

Eddy Truyen is a postdoctoral researcher in the Department of Computer Science at the Katholieke Universiteit Leuven. His research interests include aspect-oriented middleware, middleware for service-based architectures, dynamic reconfiguration, and support for building reusable aspect frameworks. Truyen received his Ph.D. in computer science from K.U. Leuven.

Stefan Walraven is a Ph.D. research student in the Department of Computer Science at the Katholieke Universiteit Leuven. His research interests include adaptive middleware and middleware for cloud computing. Walraven received his master's degree in computer science from K.U. Leuven.

Wouter Joosen is a full professor in the Department of Computer Science at the Katholieke Universiteit Leuven. His research interests are in aspect-oriented software development (focusing on software architecture and middleware) and in security aspects of software, including security in component frameworks and security architectures. Joosen has a PhD in computer science from K.U. Leuven.

Bart Dhoedt received a Masters degree in Electro-technical Engineering (1990) from Ghent University. His research, addressing the use of micro-optics to realize parallel free space optical interconnects, resulted in a Ph.D. degree in 1995. After a 2-year post-doc in opto-electronics, he became Professor at the Department of Information Technology. His research interests include software engineering, distributed systems, mobile and ubiquitous computing, smart clients, middleware, cloud computing and autonomic systems.

Filip De Turck leads the network and service management research group at the Department of Information Technology of the Ghent University, Belgium and the iMinds. He received his Ph.D. degree from the Ghent University in 2002. He is a full-time professor since October 2006 in the area of telecommunication and software engineering. His main research interests include scalable software architectures for telecommunication network and service management, performance evaluation and design of Cloud management systems.