Efficiënt resourcebeheer in een 'multi-tenant'-cloudomgeving

Efficient Resource Management in a Multi-Tenant Cloud Environment

Pieter-Jan Maenhaut

Promotoren: prof. dr. ir. F. De Turck, prof. dr. B. Volckaert
Proefschrift ingediend tot het behalen van de graad van
Doctor in de ingenieurswetenschappen: computerwetenschappen

UNIVERSITEIT
GENT

Vakgroep Informatietechnologie
Voorzitter: prof. dr. ir. B. Dhoedt
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2018 - 2019

ISBN 978-94-6355-250-9
NUR 986, 988
Wettelijk depot: D/2019/10.500/58

Ghent University
Faculty of Engineering and Architecture
Department of Information Technology

imec
Internet Technology and Data Science Lab

Examination Board:

prof. dr. ir. Filip De Turck	(advisor)
prof. dr. Bruno Volckaert	(advisor)
prof. dr. Hennie De Schepper	(chair)
prof. dr. Veerle Ongenae	(secretary)
dr. Marinos Charalambides	
dr. ir. Hendrik Moens	
prof. dr. ir. Pieter Simoens	
dr. ir. Eddy Truyen	

Dissertation for acquiring the degree of
Doctor of Computer Science Engineering

Dankwoord

"So long, and thanks for all the fish."

– Douglas Adams (1984)

Ondanks het feit dat dit de eerste volle paragrafen zijn, zijn het vermoedelijk de laatste woorden die ik in dit boek zal schrijven. Een boek dat het resultaat is van een hele reis, met vele gebeurtenissen die zeker en vast het vermelden waard zijn, op zich al genoeg stof om een boek over te schrijven. Gelukkig hoefde ik die reis nooit alleen te maken, maar was ik onderweg steeds vergezeld door een handvol personen.

Het dankwoord is dan ook vaak een plaats waar je oneindige reeksen van namen kan terugvinden, meer bepaald de namen van alle personen die rechtstreeks of onrechtstreeks bijgedragen hebben tot de realisatie van dit boek. Nu zal ik meteen de lezer moeten teleurstellen, aangezien ik besloten heb om hier geen eindeloze lijstjes te maken. De reden hiervoor is voor de hand liggend; aangezien ik de voorbije jaren op minstens vier bureaus gewerkt heb, en voor minstens even veel instellingen, zouden de lijstjes van collega's en ex-collega's gewoon te lang worden, en het risico is groot dat ik onbewust een paar namen zou vergeten. Maar toch wil ik vier personen expliciet bij naam vermelden, en niet toevallig zijn het de namen die je op de meeste publicaties opgesomd in het eerste hoofdstuk terug zal vinden. Maar laat me hiervoor eerst even terug keren in de tijd, meer bepaald naar het najaar van 2011.

Na een kort intermezzo in de privésector, begon ik eind september 2011 aan het brugprogramma tot Master of Science in de ingenieurswetenschappen: computerwetenschappen aan de Universiteit Gent. Een programma dat normaal 2 jaar zou duren, en waaraan ik deels begonnen was omdat ik een sterke interesse had om nadien een doctoraat te starten. Het eerste semester van mijn tweede 'studententijd' was behoorlijk goed gevuld met diverse projecten (ondertussen is het aantal projecten in die opleiding gelukkig sterk gereduceerd), en voor één van deze projecten had Hendrik Moens het geluk (of ongeluk) om de groep waartoe ik behoorde te mogen begeleiden. Hendrik was op dat moment zelf begonnen aan een doctoraat, en tijdens het verloop van het project kwam mijn interesse toevallig ter sprake. Het duurde dan ook niet lang tot ik - ergens midden in de examenperiode van het eerste semester - met hem en Filip De Turck samenzat in een vergaderlokaal op de Zuiderpoort, en toen ging de bal aan het rollen.

Tijdens dit eerste overleg was ik verheugd om te horen dat er zeker mogelijkheden waren om een doctoraat te starten, en aangezien ik reeds een masterdiploma

op zak had hoefde ik zelfs de opleiding waaraan ik pas begonnen was niet af te maken. Op datzelfde moment waren ze binnen mijn 'oude' opleiding, de opleiding tot industrieel ingenieur informatica, toevallig net op zoek naar een praktijkassistent voor een tijdelijke onderwijsopdracht - het woord tijdelijk mag achteraf gezien met een korrel zout genomen worden - en na een kort telefoontje met Veerle Ongenae begon ik in februari 2012 als praktijkassistent aan de (toen nog) Hogeschool Gent, gecombineerd met een doctoraat aan de Universiteit Gent binnen de onderzoeksgroep IBCN.

We springen even enkele jaren vooruit in de tijd. Na het succesvol behalen van zijn doctoraat, kreeg Hendrik een bijzonder interessante aanbieding, en had hij besloten om de universiteit te verlaten. Dit was allesbehalve evident, aangezien Hendrik sinds het begin heel nauw bij mijn onderzoek betrokken was, en hij dan ook mijn eerste aanspreekpunt was geworden voor alles wat te maken had met mijn doctoraat. Gelukkig kon ik toen terecht bij Bruno Volckaert, die de rol van Hendrik zou overnemen, en zo ben ik aangekomen bij de laatste naam van dit beperkte lijstje.

Het is vermoedelijk overbodig om te zeggen dat dit boek er nooit geweest was zonder deze vier personen, en het lijkt me dan ook maar gepast om hen bij deze expliciet te bedanken. Dankjewel Hendrik, om mij warm te maken voor een doctoraatsonderzoek, om me gedurende de eerste jaren op weg te helpen, en dankjewel voor de aangename samenwerking tijdens projecten en voor de vele fijne reizen waarin we een stukje van de wereld konden ontdekken. Ik herinner me nog goed hoe je het steeds grappig vond dat je 'een student moest begeleiden die ouder is dan jou', maar bij deze zal ik de studententijd hopelijk voor goed en altijd achter mij laten! Dankjewel Filip, om mij de kans te geven om aan deze reis te beginnen, en om steeds constructieve feedback en ideeën aan te leveren. Je gaf mij enerzijds altijd voldoende vrijheid, maar anderzijds stond je steeds klaar om naar mij te luisteren wanneer nodig. Dankjewel Veerle, om mij de kans te bieden om betrokken te zijn binnen de opleidingen industrieel ingenieur, en voor de fijne en vaak nauwe samenwerking gedurende de voorbije jaren. En dankjewel Bruno, om de taak van Hendrik verder te zetten, en me uiteindelijk tot aan de eindstreep te begeleiden. Ook bij jou kon ik steeds terecht, vaak zelfs ongeacht het tijdstip van de dag (of nacht).

En uiteraard wil ik ook al die andere personen bedanken, de namen die ik bewust niet in lijstjes wil zetten, maar die zeker ook hun steentje bijgedragen hebben tot dit doctoraat. Dankjewel aan alle huidige en ex-collega's, op zowel de Hogeschool Gent en de Universiteit Gent, alsook aan alle personen die achter de schermen werken om alles in goede banen te leiden!

Gent, juli 2019
Pieter-Jan Maenhaut

Table of Contents

List of Figures

List of Tables

List of Acronyms

A

ABAC Attribute-Based Access Control
AF Allocation Factor
API Application Programming Interface
AWS Amazon Web Services

C

CAS Cluster Agent Service
CDN Content Delivery Network
CMS Cluster Management Service

F

FFD First-Fit Decreasing

H

HFFD Hierarchical First-Fit Decreasing
HGD Hierarchical Greedy Decreasing
HPC High-Performance Computing
HTTPS Hypertext Transfer Protocol Secure

I

IaaS Infrastructure as a Service
ILP Integer Linear Programming
IoT Internet of Things

L

LLDP Link Layer Discovery Protocol

M

MC Medical Communications

N

NDP Neighbor Discovery Protocol
NFV Network Functions Virtualization
NIST National Institute of Standards and Technology

O

OS Operating System

P

P2P Peer-to-Peer
PaaS Platform as a Service
PHP Hypertext Preprocessor
PoC Proof of Concept
PXE Preboot Execution Environment

Q

QoS Quality of Service

R

RD Reallocation Delta
REST Representational State Transfer
RPiaaS Raspberry Pi as a Service

S

SaaS Software as a Service
SDN Software-Defined Networking
SDS Software-Defined Storage
SLA Service-Level Agreement
SLO Service-Level Objective
SOA Service-Oriented Architecture
SOAP Simple Object Access Protocol

T

TDS Tenant-Defined Storage

U

URI Uniform Resource Identifier

V

VM Virtual Machine
VoIP Voice over IP
VPN Virtual Private Network

Samenvatting
– Summary in Dutch –

De laatste jaren kende cloud computing een enorme groei, net als het aantal applicaties dat gebruikmaakt van deze cloud. Eén van de belangrijkste voordelen van cloud computing is dat het de uitrol en het beheer van IT oplossingen voor bedrijven eenvoudiger maakt, aangezien het de uitgaven voor het kopen, onderhouden en upgraden van hardware en software overbodig maakt. In plaats daarvan kunnen bedrijven de benodigde hoeveelheid computationele bronnen huren van een cloudprovider, en een cloudomgeving biedt een bijna oneindige hoeveelheid bronnen, waardoor applicaties uitgerold in de cloud kunnen schalen gebaseerd op de huidige of verwachte toekomstige vraag. Voor cloudgebruikers is de kostprijs typisch gebaseerd op basis van de hoeveelheid toegewezen bronnen, waardoor een efficiënt gebruik van de toegewezen bronnen aangewezen is om onnodige huurkosten te voorkomen. Cloudproviders daarentegen moeten een haalbare toewijzing bepalen van de gevraagde bronnen over de fysieke hardware, waarbij ze trachten om de benodigde hoeveelheid fysieke hardware te minimaliseren om de operationele kosten te verminderen, maar tegelijkertijd moeten ze doelstellingen afgesproken met de cloudgebruikers blijven garanderen.

Een van de belangrijkste kenmerken van cloud computing is multi-tenancy, aangezien meerdere cloudgebruikers (tenants) typisch gebruikmaken van dezelfde fysieke hardware. Applicaties uitgerold in een cloudomgeving kunnen echter ook genieten van de voordelen van multi-tenancy, door meerdere eindgebruikers te bedienen via een enkele instantie. Multi-tenancy kan dus leiden tot een hogere schaalbaarheid, en een efficiënter gebruik van de beschikbare computationele bronnen, maar een multi-tenant omgeving moet wel zorgen voor een duidelijke scheiding van data en performantie tussen de verschillende tenants.

Dit proefschrift heeft tot doel het ontwerpen en bestuderen van technologieën en technieken die gebruikt kunnen worden voor een efficiënt beheer van bronnen in een multi-tenant cloudomgeving, zowel vanuit het standpunt van onderliggende infrastructuur als van de applicaties uitgerold in een cloudomgeving. Concreet werden vier belangrijke uitdagingen beschouwd:

1. Er is nood aan een manier om applicaties uit te rollen in een cloudomgeving. Een cloudplatform kan enkele beperkingen opleggen, en sommige cloudomgevingen zijn beter geschikt dan anderen voor een bepaald type applicatie. Bovendien dienen applicaties uitgerold in een cloudomgeving de beschik-

bare bronnen zo efficiënt mogelijk te gebruiken.

2. Een strategie voor het bereiken van een hoge schaalbaarheid en een hoge graad van gebruik van middelen is noodzakelijk, maar deze strategie moet voldoende isolatie garanderen tussen de verschillende tenants. Zowel het aantal applicaties als de noden voor individuele applicaties zullen in de loop van de tijd wijzigen, waardoor er wijzigingen nodig zullen zijn in het huidige toewijzingsschema. Migraties van uitgerolde applicaties moeten echter geminimaliseerd worden, aangezien dergelijke operaties zowel kostelijk als tijdsintensief zijn.

3. Voor de validatie van resource management strategieën is een praktische aanpak wenselijk. Evaluatie van dergelijke strategieën door middel van een echt cloudplatform is echter zowel kostelijk als tijdsrovend. Simulaties kunnen gebruikt worden voor de initiële evaluatie, maar op een gegeven moment moeten experimenten op fysieke hardware ook overwogen worden om vertrouwen te krijgen in de voorgestelde strategie.

4. Cloud computing evolueert constant, en nieuwe technologieën en modellen winnen aan populariteit. In deze context is het aangewezen om te onderzoeken hoe de recente evoluties een impact hebben op het beheer van bronnen in een cloudomgeving.

Om de bovengenoemde uitdagingen aan te pakken, wordt er eerst een aanpak voor het uitrollen van applicaties in een cloudomgeving opgesteld en geverifieerd met behulp van twee praktische cases. De voorgestelde aanpak bestaat uit twee delen, een strategie voor de migratie van applicaties naar een cloudomgeving en een strategie voor het toevoegen van multi-tenancy aan een bestaande applicatie. De voorgestelde aanpak is proactief, aangezien deze mogelijke toekomstige risico's identificeert en elimineert, bijvoorbeeld door beveiligingsrisico's te beperken en de architectuur te analyseren met betrekking tot de schaalbaarheid. Door de voorgestelde aanpak te volgen, kan het gebruik van de beschikbare cloudbronnen gemaximaliseerd worden, en de huurkosten geminimaliseerd. De migratie van bestaande applicaties naar de cloud brengt kosten met zich mee, en sommige delen van de applicatie zullen aangepast of herschreven moeten worden. De langetermijnvoordelen van een cloudmigratie kunnen echter gemakkelijk opwegen tegen de kosten nodig voor het implementeren van de beschreven wijzigingen.

Vervolgens wordt er een schaalbaar systeem beschreven voor het beheer van opslagbronnen in een multi-tenant omgeving. Een multi-tenant omgeving moet zowel scheiding van data als scheiding van performantie garanderen voor elke tenant, en migratie van data in de loop van de tijd moet geminimaliseerd worden aangezien dit zowel een dure als tijdsrovende operatie is. In de voorgestelde oplossing worden de tenants georganiseerd met behulp van een hiërarchisch model, en wordt er een dynamisch algoritme gebruikt om een haalbare toewijzing van de data over een set opslagbronnen te bepalen. Het hiërarchisch model maakt de toewijzing van gegevens vanuit het perspectief van de tenant mogelijk, en waarborgt een duidelijke scheiding van tenants, rekeninghoudend met de individuele eisen.

Een belangrijke uitdaging bij onderzoek naar resource management binnen cloudomgevingen is de validatie van nieuwe strategieën in een praktische omgeving. Een algemene aanpak wordt voorgesteld, die het belang van experimentele validatie benadrukt. Experimenten uitvoeren op een bestaand cloudplatform is echter kostelijk en tijdsrovend, en niet altijd mogelijk. Bijgevolg wordt het ontwerp en de implementatie van 'Raspberry Pi as a Service' (RPiaaS) voorgesteld, een low-cost embedded cloud-testbed dat opgebouwd is uit Raspberry Pi borden. De voorgestelde aanpak wordt vervolgens geïllustreerd door het valideren van het schaalbare opslagsysteem dat eerder werd geïntroduceerd bovenop het RPiaaS-testbed. De verkregen resultaten van de experimentele evaluatie worden vergeleken met de resultaten die zijn verkregen met behulp van een op maat ontwikkelde simulatietool. Hoewel de resultaten van beide evaluaties erg gelijkaardig zijn, bood het RPiaaS-testbed de mogelijkheid om andere bruikbare waarden te meten, en werden er ook nieuwe inzichten geboden met betrekking tot de voorgestelde strategie.

Tenslotte wordt een uitgebreid overzicht gepresenteerd van de huidige stand van zaken met betrekking tot resource management in de brede zin van cloud computing, complementair aan bestaande enquêtes in de literatuur. In dit overzicht ligt de focus op hoe recent gepubliceerd onderzoek zich aanpast aan de hedendaagse evoluties binnen cloud computing, meer bepaald de opkomst van nieuwe implementatiemodellen zoals edge en fog computing, en het gebruik van containers als een lichtgewicht alternatief voor virtuele machines. Hoewel het merendeel van het onderzoek zich nog steeds richt op het beheer van virtuele machines binnen een traditionele cloudomgeving, is het duidelijk dat de recente evoluties vele problemen kunnen aanpakken van bestaande benaderingen voor resource management. Daarom worden, ter afsluiting van dit proefschrift, er nog enkele uitdagingen en opportuniteiten voor beheer van bronnen in een toekomstige cloud voorgesteld.

Summary

Over recent years, cloud computing has seen an enormous growth, as well as the number of applications deployed on top of it. One of the main benefits of cloud computing is that it simplifies the management and deployment of IT solutions for enterprises, as it eliminates the capital expense of buying, maintaining and upgrading hardware and software. Instead, enterprises can rent the required amount of computational resources from a cloud provider, and a cloud environment offers a near infinite amount of resources, allowing the deployed applications to scale up or down based on the current or expected future demand. Cloud users are typically charged based on the amount of allocated resources, introducing the need for an efficient usage of the allocated resources to avoid unnecessary rental costs. Cloud providers on the other hand need to determine a feasible allocation of the requested resources over the physical hardware, minimizing the amount of physical hardware required to reduce the operational costs while still guaranteeing the objectives agreed upon with the cloud users.

One of the main characteristics of cloud computing is multi-tenancy, as multiple cloud users (tenants) are typically served by the same set of hardware. Applications deployed on top of the cloud however can also benefit from the concepts introduced by multi-tenancy, by serving multiple end users by a single application instance. Multi-tenancy therefore can help to achieve a higher scalability, and a more efficient usage of the available resources, but a multi-tenant environment should guarantee data and performance isolation between tenants.

This dissertation aims to investigate technologies and management techniques that enable efficient resource management in a multi-tenant cloud environment, from the perspective of both the infrastructure and the applications deployed on top of it. Specifically, four main challenges were considered:

1. An approach for the deployment of applications in the cloud is required. The cloud platform could introduce some restrictions, and some environments might be better suited than others for a specific type of application. Furthermore, applications deployed on top of a cloud environment should achieve an efficient usage of the available cloud resources.

2. A strategy for achieving high scalability and a high level of resource utilization is needed that still guarantees isolation between tenants. The number of applications as well as the demand for individual applications will change over time, requiring changes to the current allocation scheme. Migration of

deployed applications should however be minimized, as such operations are both costly and time consuming.

3. For the validation of resource management strategies, a practical approach is desirable. Evaluation of such strategies on top of a real cloud platform however is both costly and time consuming. Simulations can be used for the initial evaluation, but at some point experimental evaluation using real hardware should also be considered to gain confidence in the proposed strategy.

4. The cloud is evolving, and new technologies and models are gaining popularity. In this context, it is recommended to study how the recent evolutions impact resource management in a cloud environment.

To tackle the above challenges, an approach for deploying applications in a cloud environment is first proposed and verified using two real world use cases. The proposed approach consists of two main parts, a strategy for migrating applications to a cloud environment and a strategy for incorporating support for multi-tenancy in existing applications. The presented approach is proactive, as it includes identifying and eliminating possible future risks, for example by mitigating security risks and analyzing the architecture regarding its scalability. By following the presented approach, the utilization of the available cloud resources can be maximized, while minimizing the rental costs. Migrating legacy software to the cloud comes at a cost, and some application components may need to be modified or rewritten. However, the long-term benefits of a cloud migration can easily outweigh the costs of implementing the described changes.

Next, a scalable system for the allocation of storage resources in a multi-tenant environment is presented. A multi-tenant environment should guarantee both data separation and performance isolation towards every tenant, and migration of tenant data over time should be minimized as this is both an expensive and time consuming operation. In the presented solution, tenants are hierarchically structured, and a dynamic resource allocation algorithm is used to determine a feasible allocation of the tenant data over a set of storage resources. The hierarchical structure enables the allocation of data from the tenant's perspective, guaranteeing a clear isolation of tenants, and taking custom tenant characteristics into account.

A main challenge with research targeting resource management for cloud environments is the validation of new strategies in practice. A generic approach is presented, illustrating the importance of experimental validation. Executing experiments on an existing cloud platform is however both costly and time consuming, and not always possible. As a result, the design and implementation of Raspberry Pi as a Service (RPiaaS) is described, a low-cost embedded cloud testbed which was built using Raspberry Pi nodes. The presented approach is then illustrated by validating the scalable storage system previously introduced on top of the RPiaaS testbed. The obtained results from the experimental evaluation are then compared to the results obtained using a custom developed simulation tool. Although the results from both evaluations were very similar, the experiments executed on the

RPiaaS testbed allowed for the measurement of other useful metrics, and also offered some new insights regarding the proposed strategy.

Finally, an extensive overview of the current state of the art regarding resource management within the broad sense of cloud computing is presented, complementary to existing surveys in literature. In this overview, the focus is on how recently published research is adapting to the latest evolutions within cloud computing, being the rise of new deployment models such as edge and fog computing, and the use of containers as a lightweight alternative for virtual machines. Although the majority of research is still focusing on the management of virtual machines within a traditional cloud environment, it is clear that the recent evolutions could tackle many issues with existing resource management approaches. As a result, and to conclude this dissertation, several challenges and opportunities for resource management in a future cloud are proposed.

1 Introduction

"First to mind when asked what 'the Cloud' is, a majority respond it's either an actual cloud, the sky, or something related to weather."

– Citrix Cloud Survey Guide (August 2012)

Cloud computing has seen an enormous growth over recent years, as well as the number of applications deployed on top of it. We might not realize it, but most people are already using 'the Cloud' on a daily basis. Whether it is for reading and sending e-mails, for storing and sharing pictures and documents online, or when using mobile applications, chances are high that some data is stored and/or processed in the cloud. Nowadays, most modern operating systems installed on personal computers or mobile devices are even using multiple cloud services under the hood, for basic tasks like the initial activation and registration to more advanced features such as using built-in voice assistants. As a matter of fact, for any application that requires an active internet connection, it is very likely that a cloud environment is involved.

1.1 A Shift to the Cloud

When cloud computing was introduced, one of its main selling points was that it would simplify the management and deployment of IT solutions for enterprises [1] [1].

[1] In this dissertation, the bibliography is distributed across the different chapters.

Table 1.1: Global data center workloads and compute instances. (Source: Cisco Global Cloud Index: Forecast and Methodology, 2016–2021 [2])

	2016	**2017**	**2018**	**2019**	**2020**	**2021**
Traditional Data Center	42.1M 17%	41.4M 14%	40.8M 11%	39.1M 9%	36.2M 7%	32.9M 6%
Cloud Data Center	199.4M 83%	262.4M 86%	331.0M 89%	393.3M 91%	459.2M 93%	533.7M 94%

Cloud computing eliminates the capital expense of buying, maintaining and upgrading hardware and software, as well as the cost of maintaining on-site data centers, and the electricity required for powering and cooling equipment. Instead, enterprises can rent the required amount of computational resources from a cloud provider, and all of this can be done with a few clicks, without requiring any human interaction. The cloud is also an ideal environment for enterprises with growing or fluctuating demands, as the elasticity offered by the cloud makes it easy to scale up or down. Billing happens at the end of a predefined period, and is based on the actual usage, so that enterprises only pay for what they really need, without having to make a prediction about future growth.

Furthermore, by moving to the cloud, enterprises no longer have to worry about aspects such as backups and recovery, security and availability, as these are now the responsibility of the cloud provider. Cloud services are typically offered using a well defined agreement, consisting of several objectives (e.g. regarding the guaranteed availability of the cloud services, a maximum allowed response time for service requests, incident response times, back-up and disaster recovery, etc.), and it is the provider's task to comply with these objectives.

Because of these advantages, over recent years there has been a significant shift towards the cloud [2], as illustrated in Table 1.1 and Figure 1.1. In this table and the corresponding figure, an estimation is given for the total number of global data center workloads and compute instances (in millions) per year for both traditional data center and cloud environments. In 2018, 89% of all workloads and compute instances executed within a data center were already executed in a cloud environment, and it is expected that this ratio will only increase in the future. While the number of workloads and compute instances inside traditional data center environments slowly decreases, there is an exponential increase when it comes to workloads and compute instances executed inside cloud environments, illustrating the massive growth of cloud applications.

However, this wide adoption of cloud computing introduces several challenges when it comes to efficient management of the available cloud resources. Consumers that are renting cloud resources from a cloud provider need to ensure that

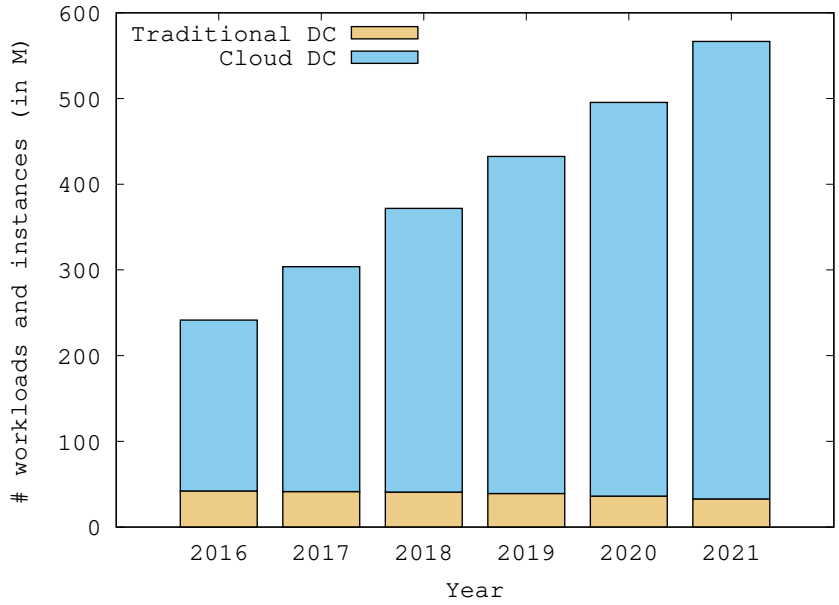


Figure 1.1: Global data center workloads and compute instances in millions. (Source: Cisco Global Cloud Index: Forecast and Methodology, 2016–2021 [2])

they can put the allocated resources to good use to avoid unnecessary rental costs. Cloud providers on the other hand need to determine a feasible allocation of the different consumers over their physical hardware, minimizing the required number of physical servers in order to reduce the energy consumption of the data center, without violating the objectives agreed upon with the consumers. This results in a more effective use of the hardware resources, and therefore helps to accommodate as many consumers as possible, and minimizes the operational costs. A cloud data center also typically consists of a heterogeneous set of resources, and hosts a wide variety of applications deployed by multiple consumers, with each application having its own requirements and characteristics. Ideally, both the set of resources and the different applications should be managed in an autonomous way, meaning that the environment can react to changes by adjusting the current resource allocation scheme or even migrating existing applications, preferably without requiring any human interaction. The heterogeneity of both the hardware and software aspects however increases the difficulty for autonomous management of the often quite complex cloud environments.

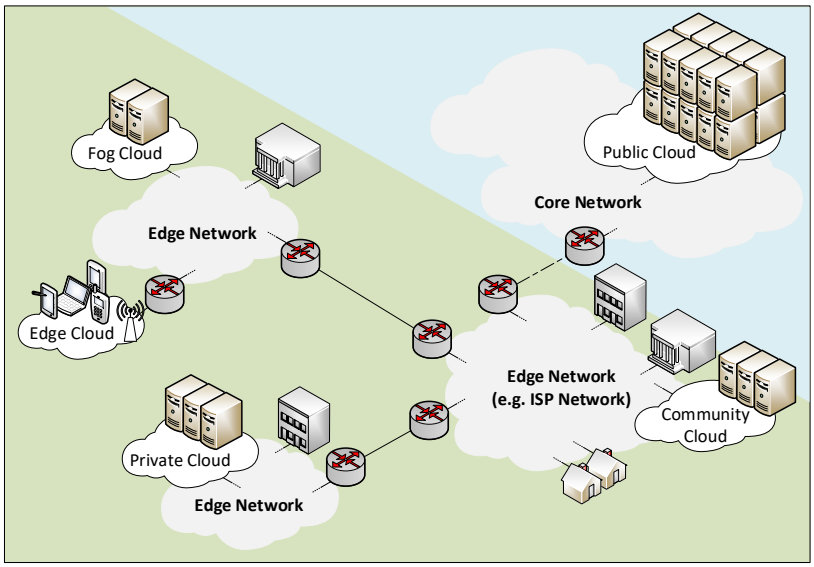


Figure 1.2: An overview of the typical deployment models used with cloud computing.

1.2 Definitions and Terminology

Before diving into the challenges, this section provides a definition of the most important concepts used throughout this dissertation.

- **Cloud computing**: while there are many definitions possible, the National Institute of Standards and Technology (NIST) published a technical report in 2011 in which they provided the following general definition [3]: *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.*

- **Cloud service model**: a cloud service model describes how providers offer their services to the consumers. The NIST defined three main service models for cloud computing, being Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) [3]. These models offer an increasing level of abstraction. With IaaS, (often virtual) resources are leased to the consumer, giving the consumer the highest level of control. A PaaS provider offers a development environment to application devel-

opers, which typically consists of a toolkit and standards for the development of cloud applications. A SaaS provider offers application software and databases to the consumers. Apart from some customization options, SaaS consumers have limited control over the software or cloud environment, and SaaS applications are typically deployed on top of an IaaS or PaaS cloud platform.

- **Cloud deployment model**: a cloud deployment model represents a specific type of cloud environment, with main characteristics being the ownership, the size, and the level of access of the cloud environment. The NIST defined four main deployment models [3]. With a private cloud, the cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers. A community cloud is similar to a private cloud, but the infrastructure is provisioned for exclusive use by a specific community of consumers. A public cloud is provisioned for open use by the general public, and is usually fully accessible over the public internet. A hybrid cloud is a composition of two or more distinct cloud infrastructures using different deployment models. A private cloud is typically limited in size, whereas a public cloud environment easily consists of ten thousands of servers. Recently, two new deployment models are gaining popularity, being edge and fog computing, which aim to bring the cloud closer to the end users. These new models are often used in combination with less powerful devices, such as mobile devices and Internet of Things (IoT) devices. Figure 1.2 provides a graphical overview of the most popular cloud deployment models.

- **Virtualization**: cloud computing is mainly built on top of virtualization, as cloud providers typically provide virtual resources to the consumers. Virtualization makes an abstraction of physical hardware, turning physical resources into logical ones. Virtualized resources provide identical functionality compared to their physical counterparts. They, however, provide more flexibility, as multiple virtualized resource bundles can be executed on a single physical machine. Popular virtualization technologies include the use of Virtual Machines (VMs) and containers [4].

- **Virtual Machine**: a VM is an emulation of a computer system. A virtual machine provides the same functionality of a physical computer, and uses virtual resources for the execution. Typically, multiple VMs are emulated on top of a single physical machine, and a hypervisor is used for managing the allocation and sharing of the physical resources over the different VMs. A VM typically runs the full software stack, meaning that an Operating System (OS) is deployed on top of the VM, and the required software is installed on top of this OS.

- **Container**: recently, container technologies have emerged as an alternative for VMs [4]. A container instance also provides an emulated environment for executing software, but the major difference with VMs is that the corresponding container image typically has no OS installed. Instead, all container instances deployed on a single machine are running directly on the OS kernel of the machine, often referred to as OS-level virtualization. As a result, container images are much smaller in size. A typical container image is a few hundreds megabytes, whereas a similar VM with the same applications installed will easily consume a few gigabytes as it contains the full OS.

- **Multi-tenancy**: with multi-tenancy, a set of resources is shared by different tenants, in order to achieve a higher level of resource utilization. In this context, a tenant can either be a single user or a group of users, typically belonging to a single organization. With software multi-tenancy for example, a single application instance is used to serve multiple tenants. Multi-tenancy is also possible at the infrastructure level, and this is in fact one of the main characteristics of cloud computing, as the physical resources (e.g. servers within a data center) are shared by multiple tenants (the consumers).

- **Resource Management**: resource management is a broad term, which refers to all required functionality for the allocation, provisioning, profiling, monitoring and pricing of (virtual) resources [5]. When deploying applications in a cloud environment, the consumer needs to determine the required amount of virtual resources, and the cloud provider needs to determine a feasible allocation over the set of physical resources available within the data center. In an elastic cloud environment, the allocated amount of resources can change dynamically based on the current demand. By monitoring and profiling the resource utilization, an estimate can be made regarding the future demand, which can be taken into account for resource scheduling. In a public cloud environment, the cloud provider needs to determine the price billed to the consumers based on the actual resource usage, and the consumer could charge the end users for using the SaaS applications deployed on top of the cloud environment.

Apart from the above definitions, Armbrust et al. defined three main actors within Cloud Computing [6]:

- The **cloud provider** or infrastructure provider manages a physical data center, and offers (typically virtualized) resources to the cloud users, either as IaaS or PaaS instances.

- The **cloud user** rents virtual resources from the cloud provider to deploy its cloud applications, which he provides (typically as SaaS) to the end users.

- The **end user** uses the SaaS applications provided by the cloud user. The end user generates workloads that are processed using cloud resources.

1.3 Challenges

Applications can benefit from the many advantages of cloud computing, with main benefits being the elimination of up-front costs, the near infinite capacity on-demand, and pricing based on the actual usage. Deploying applications in a cloud environment can therefore help to reduce the operational costs, but only if the application uses the available resources efficiently. Efficient resource management is thus of paramount importance, and not only for the cloud user. For the cloud provider, it can help to minimize the power consumption, as unprovisioned hardware can be put in standby or even turned off. This contributes to the reduction of the energy footprint of the data center, which is one of the main goals of green cloud computing, and can also reduce the operational cost of the data center. In addition, when more consumers can be served by the same set of physical hardware, the cloud provider can offer its instances at a lower price.

This dissertation investigates how cloud resources can be efficiently managed, from both the perspective of the cloud user who wants to deploy its applications on top of a cloud environment (application-level) as from the perspective of the cloud provider who needs to determine a feasible allocation of the requested resources over the physical hardware (infrastructure-level resource management). In this context, four main challenges have been addressed in this dissertation.

Challenge #1: Design and deployment of applications in a multi-tenant cloud environment. Deploying an application inside a cloud environment is slightly different from a deployment within a traditional data center environment. The cloud platform could introduce some restrictions, requiring several changes to the design and implementation of the application. Some cloud platforms might also be better suited than others for hosting a given type of application. A key enabler of cloud computing is the offered elasticity, meaning that applications or individual application components can scale up or down based on the actual or expected future demand. This however requires that the cloud elasticity is taken into account during the design of the application, as the application should be able to handle synchronization and conflicts in data when more than one instance is deployed. Furthermore, as cloud users are typically charged based on the actual resource usage, efficient use of available cloud resources is highly recommended to avoid unnecessary rental costs.

Challenge #2: Achieving high scalability and a high level of resource utilization while guaranteeing tenant isolation. When deploying applications on top of the cloud, the optimal amount of resources needs to be determined to reduce the rental costs, as well as the preferred geographical location. Given the requirements

of the different cloud users and their applications, the cloud provider needs to determine an optimal mapping for the requested bundles of resources over its set of physical hardware, in order to accommodate as many consumers as possible. In this context, a static allocation scheme can be used for the initial allocation of cloud resources. However, as the number of applications deployed on top of the cloud will change over time, as well as the demand for individual applications, some changes in the allocation scheme might be required to avoid over- or under-provisioning. This introduces the need for a dynamic allocation scheme, which aims to achieve a high utilization of the available resources and therefore a high scalability of the cloud environment. To reduce its operational costs, the cloud provider will typically try to minimize the amount of provisioned computational resources, but without violating the objectives described in the agreements with the cloud users. If the allocation scheme changes over time, some applications might need to be migrated to a different physical server, which could lead to a downtime. And if the applications to be migrated are using a large amount of data storage, the re-allocation of the application data can become both expensive and time consuming. Furthermore, as the cloud is typically a multi-tenant environment, the cloud platform should guarantee sufficient isolation between the different tenants. A tenant user should not be able to access data belonging to a different tenant, and the performance of the cloud for a single tenant should not be influenced by other tenants. In general, an efficient resource allocation scheme should provide enough flexibility to cope with changes, minimizing migrations over time, and should determine a feasible tradeoff between achieving a high level of resource utilization and therefore high scalability versus guaranteeing sufficient isolation between the different tenants.

Challenge #3: Cost-effective and reproducible validation of resource management approaches. When designing a resource allocation scheme, or a resource management approach in general, a major challenge is how to validate this new approach in practice. Ideally, the proposed solution should be validated using a real cloud environment, using a realistic data set over a large time period. Unfortunately, running such experiments on top of a cloud environment is both expensive and time consuming. This is especially true for the design and fine-tuning of new resource allocation strategies, as this often requires multiple incremental iterations of experiments using several cloud instances. Especially when experiments fail during execution, for example due to hardware constraints or a faulty algorithm, this can quickly ramp up the cost. Many cloud computing environments also have some important limitations, as cloud users rarely have full control over the underlying hardware resources. This level of control over the hardware is often one of the requirements for resource allocation strategies aiming at the physical hardware level. As a result, simulations are often used instead for the experimental validation of resource management approaches. When using a simulator, a full cloud

environment is emulated in software, and large scenarios can be quickly simulated within reasonable time. For this type of evaluation, no real cloud hardware is required, but simulators often have their limitations [7]. In general, simulations can be effectively used as a prototyping mechanism to provide a rough idea of how a particular algorithm may perform, but it is very difficult to verify if the simulation environment is an accurate representation of a real world data center environment. Therefore, experimental validation using real hardware should also be considered, but this introduces the need for a testbed that offers enough flexibility and ideally allows for low-cost and reproducible execution of experiments.

Challenge #4: Adapt to recent evolutions within cloud computing. The world of information technology is constantly evolving, and so is cloud computing. A recent trend within cloud computing is the uprise of new types of clouds, such as edge and fog computing. The cloud is no longer limited to the centrally hosted data center, accessible from a laptop or desktop computer with a broadband internet connection, but lightweight devices such as mobile phones and Internet of Things (IoT) devices can also benefit from the infinite amount of resources offered by the cloud. These devices can offload computationally intensive tasks to a centrally hosted cloud, and by installing dedicated hardware at the edge of the network, close to the end user devices, the latency can be reduced, as well as the consumed network bandwidth towards the public cloud. When it comes to virtualization, containers are gaining popularity, due to the minimal overhead compared to traditional virtual machines and the offered portability. Traditional resource management strategies however are typically designed for the allocation and migration of virtual machines within a traditional cloud environment, so the question arises how these strategies can be adapted to support these new trends.

1.4 Outline

This dissertation is composed of a number of publications that were realized within the scope of this PhD. The selected publications provide an integral and consistent overview of the work performed. The different research contributions are detailed in Section 1.5 and the complete list of publications that resulted from this work is presented in Section 1.6. This section provides an overview of the remainder of this dissertation, and explains how the different chapters are linked together. A schematic overview of how the chapters (Ch.) and appendices (App.) are related to each other and to the research contributions is depicted in Figure 1.3. Table 1.2 illustrates how the chapters of this dissertation relate to the challenges listed in Section 1.3.

Chapter 2 investigates how existing applications can benefit from migration to a cloud environment. If the application will be used by multiple end users, it can be beneficial to add multi-tenancy to the application and serve different ten-

Table 1.2: An overview of the contributions per chapter in this dissertation.

	Ch. 2	Ch. 3	Ch. 4	Ch. 5	App. A
Challenge #1	•			•	
Challenge #2	•	•		•	•
Challenge #3			•		
Challenge #4				•	

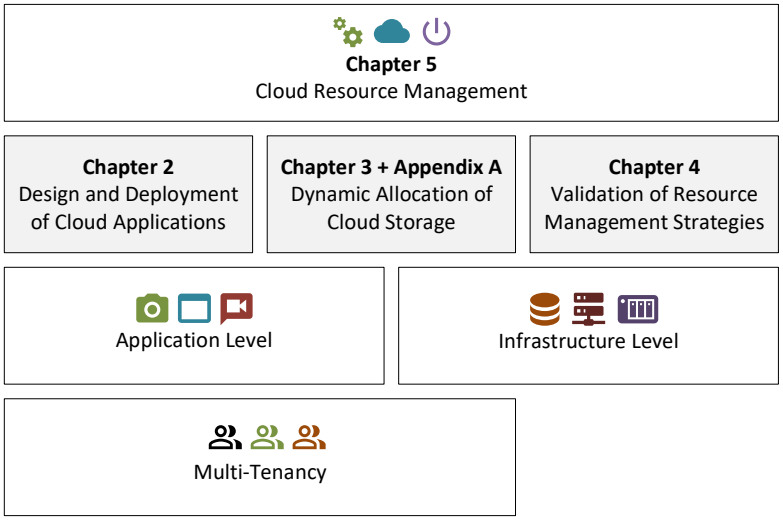


Figure 1.3: Schematic overview of the different chapters in this dissertation.

ants by a single application instance, instead of deploying a separate instance for every tenant. A summary is provided of both the steps required to migrate existing applications to a public cloud environment, and to incorporate support for multi-tenancy in these applications. A generic approach is presented and verified by means of two case studies. Both case studies are subject to stringent security and performance constraints, which need to be taken into account during the migration. Migrating legacy software to the cloud comes at a cost, and some application components may need to be modified or rewritten. However, by following the migration approach presented in this chapter, the benefits of a cloud migration could outweigh the costs of implementing the described changes, and incorporating application-level multi-tenancy will result in a higher and more efficient usage of the available cloud resources, which, in the long term, can lead to a significant reduction in operational costs.

Once the application is deployed on top of a cloud environment, it can scale up or down based on the current or expected future demand. This will however require the (de-)provisioning of additional resources over time. In Chapter 3 a

generic approach for the dynamic allocation of cloud resources in a multi-tenant environment is presented, which specifically focuses on the allocation of storage resources. A multi-tenant environment should guarantee both data separation and performance isolation towards every tenant, and migration of tenant data over time should be minimized as this is both an expensive and time consuming operation. In the presented approach, tenants are hierarchically clustered based on multiple scenario-specific characteristics, and allocated to storage resources using a hierarchical bin packing algorithm (static allocation). As the load changes over time, the system responds to these changes by reallocating storage resources when required (dynamic reallocation), but will try to minimize the amount of migrated data and hence the migration cost. The presented approach is generic and can be implemented either at application level, for example for the allocation of storage resources for data-intensive applications, or at infrastructure level, for example for the management of virtual machines and their corresponding virtual disks. An extension of the approach can be found in Appendix A, in which the focus is on the allocation of storage resources for data records stored inside relational databases. The latter can have indexes defined over the data, which speed up search operations, and the influence on the performance of these operations is investigated when the records are divided over multiple separate database instances.

Chapter 4 focuses on the validation of resource management approaches designed for cloud environments. As described in the previous section, experimental validation can be both costly and time consuming, and therefore resource management approaches are often only validated using simulations in software. In this chapter, a general approach for the validation of cloud resource allocation strategies is introduced that is not limited to simulations, and the importance of experimental validation on physical testbeds is illustrated. Furthermore, the design and implementation of Raspberry Pi as a Service (RPiaaS) is presented, a low-cost embedded testbed that is built using Raspberry Pi nodes, which serves as a miniature cloud environment. RPiaaS aims to facilitate the step from simulations towards experimental evaluations on larger cloud testbeds. The presented validation approach is then illustrated by evaluating the resource allocation strategy introduced in Chapter 3 on top of the RPiaaS testbed.

The RPiaaS testbed introduced in Chapter 4 was designed using a microservice architecture, where experiments and all required management services are running inside containers. Furthermore, the testbed can in fact be seen as an edge environment, as it consists of lightweight Raspberry Pi nodes, and it could collaborate with a different large-scale cloud environment for offloading of computational intensive tasks. Fog and edge computing are relatively new concepts, but are recently gaining popularity within cloud computing. The majority of resource management approaches however still focus on the management of virtual machines inside a traditional (private, public, community or hybrid) cloud environment, so the question

arises how these strategies can be adapted for the management of a containerized cloud. Therefore, in Chapter 5 an overview of the current state of the art regarding resource management within the broad sense of cloud computing is provided, complementary to existing surveys in literature. This chapter investigates how research is adapting to the recent evolutions within the cloud, being the adoption of container technology and the rise of fog and edge computing. Finally, this dissertation is concluded by identifying several challenges and possible opportunities for future research.

1.5 Research contributions

In Section 1.3, the problems and challenges for managing resources in a multi-tenant cloud environment are formulated. They are tackled in the remainder of this PhD dissertation for which the outline is given in Section 1.4. To conclude, this section presents an elaborated list of the research contributions made by this dissertation:

- An approach for deploying multi-tenant applications in a public cloud environment. (Chapter 2, mainly addressing *Challenge #1*)
  - A migration strategy, consisting of both the steps required for migrating legacy applications to a public cloud environment and the steps required for incorporating support for multi-tenancy in an existing application.
  - Two case studies based on existing medical software, which illustrate the presented migration strategy in a practical scenario.
  - An overview of the possible advantages and disadvantages of moving application components to the public cloud, and an evaluation of the migration costs versus the possible cost savings in the long term.
- Design and implementation of a scalable storage system for the management of tenant data in a cloud environment. (Chapter 3, mainly addressing *Challenge #2*)
  - Design of a Tenant-Defined Storage system, in which tenants are hierarchically clustered based on multiple characteristics, and storage resources are allocated using a data fusion approach to maximize isolation of tenant data.
  - Introduction of two novel algorithms for the allocation of tenant data, and a dynamic extension of these algorithms for minimizing the number and amount of migrations over time. The presented algorithms are based on a well-known heuristic for the bin packing problem [8], but

are specifically designed for packing items with a hierarchical structure (referred to as hierarchical bin packing).
  - An implementation of the proposed system and the different algorithms, used for fine-tuning and evaluating the performance of the system.
  - An extensive evaluation of the performance of the algorithms, in which both the average resource utilization and the number and amount of migrations over time are measured.

- A general approach for the validation of resource management strategies, and the introduction of a low-cost embedded cloud testbed. (Chapter 4, mainly addressing *Challenge #3*)
  - A general approach for the validation of resource management strategies, which illustrates the importance of experimental validation, complementary to evaluation using simulation tools.
  - The design and implementation of RPiaaS, a low-cost embedded cloud testbed built using Raspberry Pi nodes.
  - A comparison of the performance and cost of the RPiaaS testbed to traditional cloud testbeds.
  - A case study to illustrate the presented approach, in which the storage system introduced in Chapter 3 is implemented and evaluated on top of the RPiaaS testbed, and the experimental results are compared to those obtained through simulations.
- A survey of recent evolutions within cloud computing, and their impact on resource management. (Chapter 5, mainly addressing *Challenge #4*)
  - A summary of cloud, edge and fog computing, and the main virtualization technologies that enable them, being virtual machines and containers.
  - An extensive overview of recent research focusing on resource management within cloud environments, with a special focus on the adoption to recent evolutions.
  - The introduction of several challenges and opportunities for resource management in cloud environments.

1.6 Publications

The research results obtained during this PhD research have been published in scientific journals and presented at a series of international conferences. The following list provides an overview of these publications.

1.6.1 A1: Journal publications indexed by the ISI Web of Science "Science Citation Index Expanded"

1. **P.-J. Maenhaut**, H. Moens, V. Ongenae and F. De Turck. *Migrating legacy software to the cloud: approach and verification by means of two medical software use cases.* Journal of Software: Practice and Experience (SPE), Volume 46, Issue 1, pages 31–54, 2016.

2. **P.-J. Maenhaut**, H. Moens, B. Volckaert, V. Ongenae and F. De Turck. *A dynamic Tenant-Defined Storage system for efficient resource management in cloud applications.* Journal of Network and Computer Applications (JNCA), Volume 93, pages 182–196, 2017.

3. **P.-J. Maenhaut**, B. Volckaert, V. Ongenae and F. De Turck. *Efficient resource management in the cloud: From simulation to experimental validation using a low-cost Raspberry Pi testbed.* Journal of Software: Practice and Experience (SPE), Volume 49, Issue 3, pages 449-477, 2019.

4. **P.-J. Maenhaut**, B. Volckaert, V. Ongenae and F. De Turck. *Resource Management in a Containerized Cloud: Status and Challenges.* Submitted to Springer Journal of Network and Systems Management, 2019.

1.6.2 P1: Proceedings included in the ISI Web of Science "Conference Proceedings Citation Index – Science"

1. **P.-J. Maenhaut**, H. Moens, M. Verheye, P. Verhoeve, S. Walraven, E. Truyen, W. Joosen, V. Ongenae and F. De Turck. *Migrating medical communications software to a multi-tenant cloud environment.* Published in proceedings of the 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), pages 900–903, Ghent, Belgium, 2013.

2. **P.-J. Maenhaut**, H. Moens, M. Decat, J. Bogaerts, B. Lagaisse, W. Joosen, V. Ongenae and F. De Turck. *Characterizing the performance of tenant data management in multi-tenant cloud authorization systems.* Published in proceedings of the 2014 IEEE Network Operations and Management Symposium (NOMS 2014), pages 1–8, Krakow, Poland, 2014.

3. **P.-J. Maenhaut**, H. Moens, V. Ongenae and F. De Turck. *Scalable user data management in multi-tenant cloud environments.* Published in proceedings of the 10th International Conference on Network and Service Management (CNSM 2014), pages 268–271, Rio de Janeiro, Brazil, 2014.

4. **P.-J. Maenhaut**, H. Moens, V. Ongenae and F. De Turck. *Design and evaluation of a hierarchical multi-tenant data management framework for cloud*

applications. Published in proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM 2015), pages 1208–1213, Ottawa, Canada, 2015.

5. **P.-J. Maenhaut**, H. Moens, B. Volckaert, V. Ongenae and F. De Turck. *Design of a hierarchical software-defined storage system for data-intensive multi-tenant cloud applications.* Published in proceedings of the 11th International Conference on Network and Service Management (CNSM 2015), pages 22–28, Barcelona, Spain, 2015.

6. **P.-J. Maenhaut**, H. Moens, B. Volckaert, V. Ongenae and F. De Turck. *A simulation tool for evaluating the constraint-based allocation of storage resources for multi-tenant cloud applications.* Published in proceedings of the 2016 IEEE/IFIP Network Operations and Management Symposium (NOMS 2016), pages 1017–1018, Istanbul, Turkey, 2016.

7. **P.-J. Maenhaut**, B. Volckaert, V. Ongenae and F. De Turck. *RPiaaS: A raspberry pi testbed for validation of cloud resource management strategies.* Published in proceedings of the 2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pages 946–947, Atlanta, GA, USA, 2017.

8. **P.-J. Maenhaut**, H. Moens, B. Volckaert, V. Ongenae and F. De Turck. *Resource allocation in the Cloud: From simulation to experimental validation.* Published in proceedings of the 2017 IEEE 10th International Conference on Cloud Computing (CLOUD 2017), pages 701–704, Honolulu, CA, USA, 2017.

1.6.3 C1: Publications in other international conferences

1. S. Leroux, S. Bohez, **P.-J. Maenhaut**, N. Meheus, P. Simoens and B. Dhoedt. *Fingerprinting encrypted network traffic types using machine learning.* Published in proceedings of the 2018 IEEE/IFIP Network Operations and Management Symposium (NOMS 2018), pages 1–5, Taipei, Taiwan, 2018.

2. J. Moeyersons, **P.-J. Maenhaut**, F. De Turck and B. Volckaert. *Aiding First Incident Responders Using a Decision Support System Based on Live Drone Feeds.* Published in proceedings of Knowledge and Systems Sciences (KSS 2018), pages 87–100, Tokyo, Japan, 2018.

3. J. Moeyersons, B. Verhoeve, **P.-J. Maenhaut**, B. Volckaert and F. De Turck. *Pluggable Drone Imaging Analysis Framework for Mob Detection during Open-air Events.* Published in Proceedings of the 8th International Conference on Pattern Recognition Applications and Methods (ICPRAM 2019), pages 64–72, Prague, Czech Republic, 2019.

4. L. Van Hoye, **P.-J. Maenhaut**, B. Volckaert, T. Wauters and F. De Turck. *Logging mechanism for cross-organizational collaborations using Hyperledger Fabric.* Accepted for publication in Proceedings of the 2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC 2019), Seoul, South Korea, 2019.

References

[1] A. Khajeh-Hosseini, D. Greenwood, and I. Sommerville. *Cloud Migration: A Case Study of Migrating an Enterprise IT System to IaaS*. In 2010 IEEE 3rd International Conference on Cloud Computing, pages 450–457, July 2010. doi:10.1109/CLOUD.2010.37.

[2] *Cisco Global Cloud Index: Forecast and Methodology, 2016-2021*. Technical report, 2018. Available from: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html.

[3] P. Mell and T. Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Technical report, Gaithersburg, MD, United States, 2011.

[4] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay. *Containers and Virtual Machines at Scale: A Comparative Study*. In Proceedings of the 17th International Middleware Conference, Middleware '16, pages 1:1–1:13, New York, NY, USA, 2016. ACM. Available from: http://doi.acm.org/10.1145/2988336.2988337, doi:10.1145/2988336.2988337.

[5] B. Jennings and R. Stadler. *Resource Management in Clouds: Survey and Research Challenges*. Journal of Network and Systems Management, 23(3):567–619, Jul 2015. Available from: https://doi.org/10.1007/s10922-014-9307-7, doi:10.1007/s10922-014-9307-7.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. *A View of Cloud Computing*. Commun. ACM, 53(4):50–58, April 2010. doi:10.1145/1721654.1721672.

[7] A. Ahmed and A. Sabyasachi. *Cloud computing simulators: A detailed survey and future direction*. 2014 IEEE International Advance Computing Conference (IACC), pages 866–872, Gurgaon, India, February 21–22, 2014. https://doi.org/10.1109/IAdCC.2014.6779436. doi:10.1109/IAdCC.2014.6779436.

[8] D. S. Johnson. *Fast algorithms for bin packing*. Journal of Computer and System Sciences, 8(3):272 – 314, 1974. Available from: http://www.sciencedirect.com/science/article/pii/S0022000074800267, doi:https://doi.org/10.1016/S0022-0000(74)80026-7.

2

Migrating Legacy Software to the Cloud: Approach and Verification by means of Two Medical Software Use Cases

In this chapter, we investigate how applications can benefit from deployment in an elastic cloud environment. Adding multi-tenancy to these applications can help to achieve a higher scalability at a lower cost. We describe the steps required to migrate existing applications to a public cloud environment, and the steps required to add multi-tenancy to these applications. This generic approach is then verified by means of two case studies, a commercial medical communications software package mainly used within hospitals for nurse call systems and a schedule planner for managing medical appointments. Both case studies are subject to stringent security and performance constraints, which need to be taken into account during the migration. In our evaluation, we estimate the required investment costs and compare them to the long term benefits of the migration.

⋆ ⋆ ⋆

P.-J. Maenhaut, H. Moens, V. Ongenae and F. De Turck

Published in Wiley Journal of Software: Practice and Experience (SPE), Volume 46, Issue 1, pages 31–54, January 2016.

2.1 Introduction

Cloud computing is a technology that enables elastic, on-demand resource provisioning. Over the last few years many companies have used clouds to build new highly scalable systems. However, legacy applications can also benefit from the advantages of cloud computing, and there is a general trend for moving applications to a cloud infrastructure, consolidating hardware, saving costs and allowing applications to react faster to sudden changes in demands. With the recent evolution of cloud computing [1] and Software as a Service (SaaS) in particular, an elastic, scalable multi-tenant architecture has gained popularity [2]. Elastic systems are able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner. With cloud computing, an optimal usage of available resources is recommended to reduce operating costs, as the infrastructure provider usually charges for the number of instances used. SaaS is a software delivery model in which the software and associated data are centrally hosted on the cloud, and the end-users are typically accessing the software through the browser or by using a thin client. As the number of clients grows, a scalable architecture for both the application and data is needed.

Multi-tenancy [3] enables the serving of multiple clients or tenants by a single application instance. The major benefits include increased utilization of available hardware resources and improved ease of maintenance and deployment. Without a multi-tenant architecture, the cost savings using cloud computing are limited for applications requiring continuous availability, as for every new client (tenant), a separate Virtual Machine (VM) instance would have to be provisioned. This instance must then be available at all times, even if it is only used sporadically. Also, as every tenant has a dedicated instance, some resources would be wasted, especially for smaller clients. Using a multi-tenant architecture, a SaaS application could run on few instances that are shared between the different users, and the number of instances could dynamically grow with the current demand. Smaller tenants could be co-located on a single instance, minimizing costs and maximizing resource utilization.

Therefore, when migrating applications to the cloud, it is recommended to adapt the legacy software to support multi-tenancy. Some changes to the architecture will be necessary, coming at a one-time cost, but this cost is overruled by the long-term benefits. Apart from adapting the legacy software for supporting multi-tenancy, some other changes may be needed to support the migration to a public or hybrid cloud, as every Platform as a Service (PaaS) or Infrastructure as a Service (IaaS) provider will have its own limitations and possibilities.

This chapter proposes an approach for both migrating applications to a hybrid or public cloud, and for incorporating support for multi-tenancy in the existing software with a minimal overhead. We verify our approach using two different

case studies of legacy medical applications which are migrated to the cloud, and discuss the required changes. We describe the advantages and disadvantages of moving components of the software to the public cloud, and evaluate the migration costs.

In the next section of this chapter we will discuss related work. Afterward, in Section 2.3, we will present the approach for both migrating legacy software to the cloud and adding multi-tenancy. We verify this approach in Section 2.4 and Section 2.5 using two different case studies. In Section 2.6, we discuss our approach and present our evaluation results. In Section 2.7, we state our conclusions and discuss avenues for future research.

2.2 Related Work

In previous work [4], we described the steps required to migrate an existing .NET-based application to the Windows Azure public cloud environment, and proposed a specific approach for adding multi-tenancy to the application. In this chapter, we propose a generic migration approach for migrating legacy applications to the cloud. We describe the different steps of our approach in detail, and verify our approach by means of two case studies. In this chapter we also present an extended discussion and evaluation based on the results from the two case studies.

An approach for partially migrating applications to the cloud is presented in [5], together with a model to explore the benefits of a hybrid migration approach. The approach focuses on identifying components to migrate, taking into account various rules such as performance and security. We also focus on migration to a hybrid or public cloud, but extend their approach by going into detail about the complete migration process, and not only selecting the components to migrate. We also present an approach for adding multi-tenancy to the application to optimal benefit from the migration to a public cloud.

When migrating software to the cloud, some choices have to be made. Different cloud computing service models exist, each having its own advantages and limitations. Figure 2.1 provides an overview of the different cloud service models. The legacy software could for example be fully migrated to a public cloud, or a hybrid approach could be used. When it comes to public cloud providers, CloudCmp [6] offers a system for comparing the performance and cost of the different providers. For the implementation, the authors use computation, storage and network metrics. For the storage metrics, they selected some benchmark tasks and measured the response times, throughput, time to consistency and cost per operation.

Cost savings and other organizational benefits and risks of migration to IaaS are discussed in [7]. We however don't limit our approach to migrations to an IaaS provider, but also consider migrations to a PaaS platform. When using an

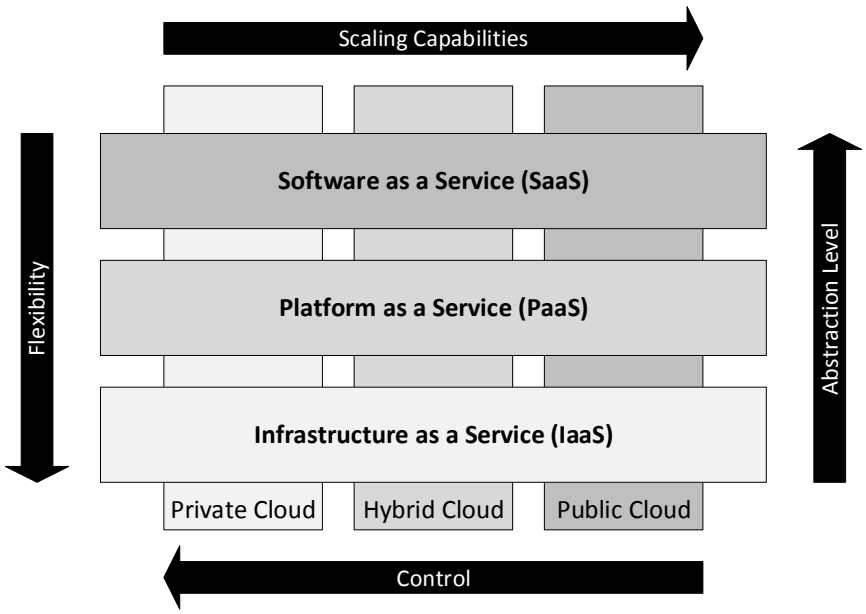


Figure 2.1: An overview of the different cloud service models used in cloud computing.

IaaS provider, the customer has full access to the operating system, middleware and runtime, hosted on a virtual machine. On the other hand, when using a PaaS provider, the customer only manages the application and data, which brings some limitations, such as the selected operating system and supported frameworks and libraries.

In [8] a checklist is presented that can be used to determine whether applications are compatible with a chosen PaaS provider. The approach is evaluated by three case studies where a Java application and two Python applications are migrated to Google App Engine. Three different and representative PaaS platforms are compared in [9], based on a practical case study, with respect to their support for SaaS application development. In this chapter, we focus on how complex applications can be executed on the public cloud, and for our case studies, we go into detail on migrating two different legacy applications. We don't limit our work by determining whether the applications are compatible with the selected provider, but also describe the different steps required in detail. Furthermore, we describe how multi-tenancy can be added, making it possible to better utilize individual application instances.

As our first case study handles a legacy application written in .NET, we selected Windows Azure to host some components of the legacy software. The migration of an on-premise web application to Windows Azure is described in [10],

together with a comparison of the application's performance when deployed to a traditional Windows server versus its deployment to Windows Azure. While the cloud migration of a .NET application requires limited effort, Azure has no built-in support for multi-tenancy, so it must be added during the migration process. In this chapter, we discuss both the steps needed to migrate an application to the cloud, and the steps needed to add multi-tenancy to the application.

To support highly customizable SaaS applications, we use a software product line based customization approach, which we have previously discussed in [11], [12], [13] and [14]. In this approach, variability is modeled by defining multiple features and the relations between them. These features are then associated with separate code modules that are deployed separately. The application is then composed out of these multi-tenant components, resulting in an application that is both customizable and multi-tenant. For changes that do not impact the performance of the application, a multi-tenancy enablement layer can be used, which amongst others can be used for data isolation, feature management and tenant-specific customizations [15].

In Appendix A we focus on the scalability of tenant data in multi-tenant applications and the impact on the performance of the application. The outcome of this research is used in [16] to build an abstraction layer for achieving high scalability for the storage of tenant data. This layer uses data allocation algorithms to determine an acceptable allocation of tenant data to different databases. The presented solution can be used for decoupling the databases and the management of tenant data, two of the steps in the approach presented in this chapter.

2.3 Migration Strategy

In this section, we describe both the steps needed to migrate an existing application to the cloud, and to redesign the application to support multi-tenancy. We start this section by briefly describing the concept of multi-tier architectures, a popular software architecture used by many applications, which we will refer to later in this section. Next we discuss the different steps of our approach, as summarized in Figure 2.2.

Many applications are designed using a multi-tier architecture, where the application is separated into multiple layers. A typical multi-tier architecture consists of 3 layers: the client layer, the business logic layer and the database layer. We refer to this basic layered architecture as the *3-Tier architecture*. Most layered applications will have more than 3 layers, as more layers can be easily added to the architecture if needed. For example, when working with multiple database instances, an extra data access layer can be added between the business logic and database layer, responsible for load balancing and selecting the correct instance. Other architectures are possible, but in the remainder of this section, we will start

(a) Cloud Migration

(b) Adding Multi-Tenancy

Figure 2.2: A summary of the different steps required to migrate an existing application to the cloud, and to add multi-tenancy to the application. The different steps are described in detail in sections 2.3.1 and 2.3.2.

from the *3-Tier architecture*.

2.3.1 Cloud Migration

The process to migrate an existing application to a public or hybrid cloud can be summarized in a few steps, illustrated in Figure 2.2a and described below.

2.3.1.1 Selecting Components

The first step during the planning phase should be to select the components of the software to migrate to the public cloud, as described in [5]. Both components of the business logic layer and the data access layers can be selected. The selection can happen based on the quality attributes of the application, to guarantee the required Quality of Service (QoS) and Service Level Agreements (SLAs). In case the whole application is being migrated, this step is quite straightforward, but when only some components of the application are selected, the architecture might need to be reviewed. Special attention has to be paid to the communication between the different components, as the communication between the dedicated servers and the public cloud might need extra security, extra bandwidth, and usage of standardized protocols. When using a Service-Oriented Architecture (SOA), communication between the different modules could for instance make use of Simple Object Access Protocol (SOAP) or Representational State Transfer (REST) over Hypertext Transfer Protocol Secure (HTTPS). Possible communication between the client layer and the components of the business logic layer should also be secured.

Figure 2.3 illustrates an example of the possible communication between the different components after migration to a hybrid cloud. The components are rep-

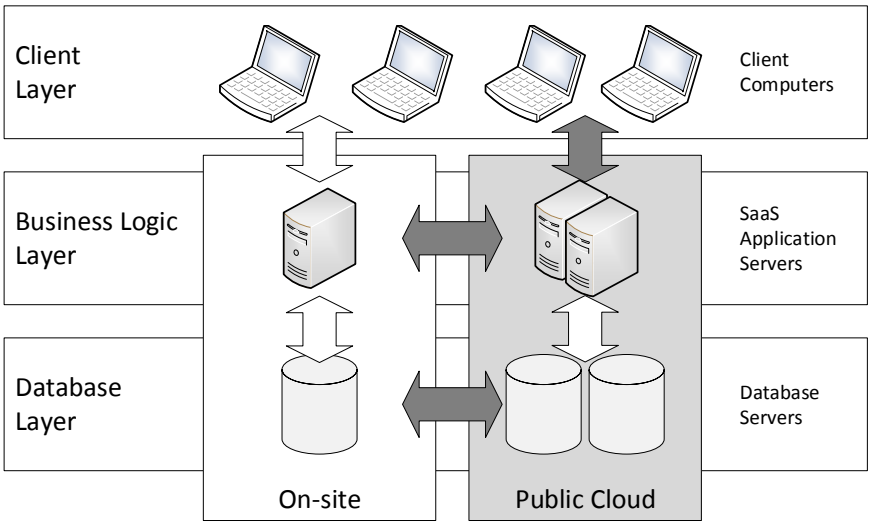


Figure 2.3: An illustrative example of the possible communication between components after migration to a hybrid cloud. Dark arrows denote communication that should be secured.

resented by server instances. Dark arrows denote communication where extra attention has to be paid regarding security and available bandwidth.

2.3.1.2 Determining Provider Compatibility

Apart from selecting the components for migration, some extra changes might be needed for migrating the application to the public cloud. Every public PaaS provider will typically have its own limitations and possibilities, so during the planning phase of the migration, it is best to verify that the provider will support all features of the software. In case no suitable PaaS provider can be found, an IaaS provider could also be selected to host some components of the application, but this again results in more maintenance overhead for the application provider. When comparing different providers, for example by using CloudCmp [6], the balance should be made between the advantages of the selected provider and the overhead due to needed changes to the application. Different public cloud providers should be considered and evaluated, for example by using a small Proof of Concept (PoC), and the advantages of using a PaaS provider should also be weighted against the increased control gained when using an IaaS provider.

2.3.1.3 Determining Impact on Client Network

A side effect of the migration to cloud environments is that communication between some components of the software might need to pass over the Internet, especially when migrating the software to a hybrid cloud. As a result, more traffic bandwidth at the client network might be required. Before deploying the service, it is important to perform an impact analysis whenever the client configurations are changed. We have previously covered this in-depth in [17].

2.3.1.4 Scaling the Application

When an instance is overloaded, extra instances can be added (up-scaling) and removed (down-scaling) in a few steps. This concept is often referred to as the elasticity of the (public) cloud. Some public cloud providers offer out of the box load-balancing and/or scaling, other providers only provide limited load-balancing possibilities, together with an Application Programming Interface (API) to support up-scaling and down-scaling from within the application.

When selecting the components to migrate, it is a good idea to take into account the scalability of the application. Components which should be highly scalable could be good candidates for migration to the public cloud, as this environment offers a quasi unlimited resource pool. The application should also support decoupling of the components and the handling of synchronization and conflicts in data. Possible bottlenecks should be eliminated, as these could break the whole scalability of the application. Reviewing the architecture of the application to better support scalability will bring some overhead, but the advantages on the long term will outweigh this one-time investment.

2.3.2 Multi-Tenancy

In this subsection, the steps required to add multi-tenancy to an existing application are discussed. These steps are also summarized in Figure 2.2b.

2.3.2.1 Decoupling Databases

As multiple tenants will use the same application instance, each tenant will have its own application data stored in a shared or dedicated database instance. Using shared database instances is cheaper, while dedicated databases will lead to better performance and higher security, but at a higher cost. To connect to the correct database, a connection string is associated with each tenant. These connection strings can for example be stored in a shared database.

The application database needs to be decoupled, and support for multi-tenancy needs to be added to the data tables in case of shared instances. Also, the application needs to be modified to support dynamic database binding. An extra

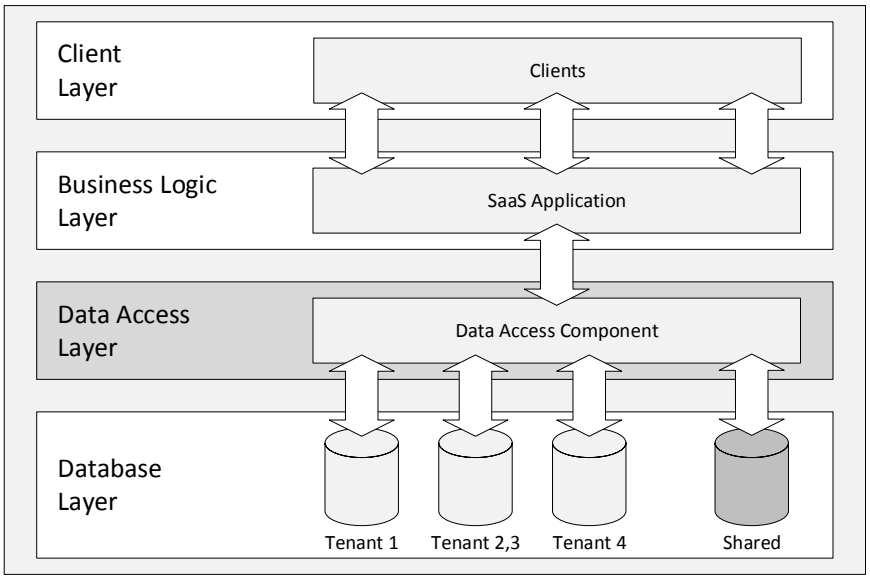


Figure 2.4: Possible architecture of the application after decoupling the databases.

component can be added to the application, the *data access component*, responsible for both the correct handling and access control of all data requests by the application. Figure 2.4 illustrates a possible architecture of the application after decoupling the databases. The data access component is added to a new layer, the data access layer, situated between the business logic layer and the database layer.

For the design of the data access layer, the abstraction layer presented in [16] could be used. This abstraction layer mainly handles the security and isolation of tenant data, and the scalability of the database layer. In our approach, we partition tenant data over multiple database instances based on the tenant. By doing so, tenants can still store their data in a dedicated on-site database instance, for example to comply with regulatory policies on data. Partitioning the data based on the tenant also provides a clear separation of tenant data. For large tenants with a dedicated database instance, the performance of the database will not be influenced by other tenants. As the number of tenants using a shared database instance will be limited, the possible damage due to an information leakage is also minimized. Different SLAs can be provided, based on the scenario of a dedicated or shared database instance.

2.3.2.2 Adding Tenant Configuration Database

A new database, which we refer to as the *tenant configuration database*, needs to be added to store general information about all tenants. The connection strings in-

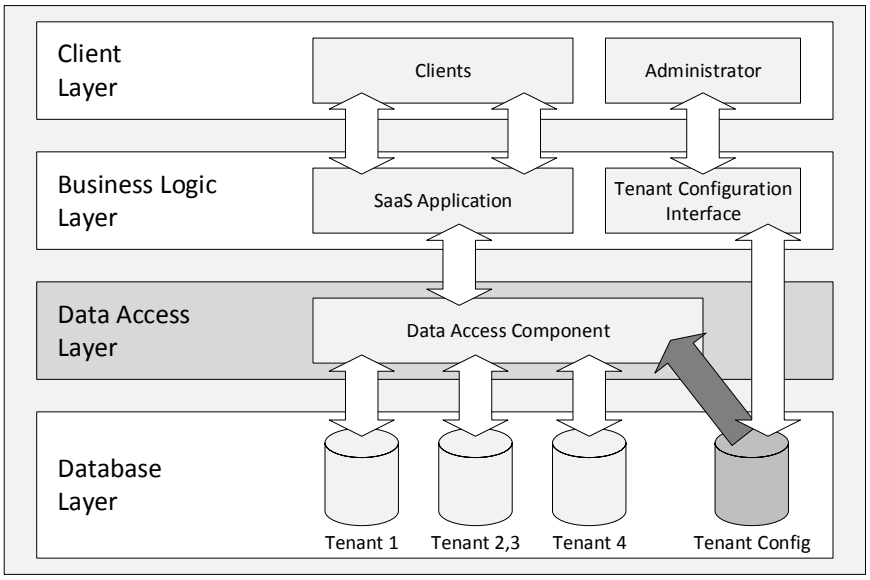


Figure 2.5: Possible architecture of the application after adding the tenant configuration interface.

troduced in the previous steps will be stored in this database, together with specific information and configuration parameters such as billing and contact information, and the selection of features for the tenant as described in Section 2.3.2.4. While this database is shared between all tenants, it only contains minimal information, and is only accessed sporadically as the information inside this database can be cached by the application, so it should not become a bottleneck [18] [17].

2.3.2.3 Providing Tenant Configuration Interface

Adding multi-tenancy to the application makes it possible to more flexibly select the application features used by different clients, as the tenant configuration is stored in the shared *tenant configuration database*. It is however also necessary to create a separate application, the *tenant configuration interface*, which can be used by tenant administrators to modify the tenant configuration. This interface will be used to create, modify and delete tenants in an easy way, and change the configuration of a single tenant, for example the selection and configuration of features and the connection string of the tenant.

Ideally, the *tenant configuration interface* is the only component which has read/write access to the *tenant configuration database*, as the legacy application should only require read access. Figure 2.5 illustrates the changes to the architecture after adding this interface.

2.3.2.4 Dynamic Feature Selection

An application can have multiple features which will be dynamically loaded at start-up. As every tenant can have its own selection of features, and a tenant-specific configuration for these features, a tenant administrator should be able to select and configure these features using the *tenant configuration interface* introduced before.

The application itself needs to support the dynamic selection of features. For example, some features might require additional modules, and the application needs to support dynamic loading (and unloading) of the corresponding modules. Also, the user interface of the application might need to be automatically adapted for the different tenants, based on their configuration, representing the tenant's feature selection. The different features might run on the same instance of the application, or on dedicated machines. In the latter case, feature placement algorithms can be used to determine the optimal solution. We have previously covered this in [11].

2.3.2.5 Managing Tenant Data, Users and Roles

Every tenant using the application will typically have its own data, users and custom roles. The users could be stored in a shared common database or in the tenant database, or the application could support external identity providers. In case the users are stored in a global shared database, or when an external identity provider is used, the application could provide single sign-on scenarios. In case the users are stored in the tenant database, an administrator should be able to create and modify users and their corresponding roles from within the application. By introducing multi-tenancy, a tenant administrator role with permissions to create and modify the tenant configuration using the *tenant configuration interface* is required, different from the administration roles within a single tenant. These tenant administrators can be stored in the shared *tenant configuration database* and should have limited access to the multi-tenant application for every tenant if required. The management of users and roles could be moved to the *tenant configuration interface*, or could stay inside the application, depending on the application's requirement and the software license model. The question arises how and where to store the tenant data and the different users and roles. Different approaches are possible, and we will cover this in-depth in Appendix A.

2.3.2.6 Mitigating Security Risks

A major disadvantage of using multi-tenancy is an increased security risk, as by definition multiple tenants will use the same application instance. These risks can be mitigated in multiple ways:

- **Implementing URL-based filtering of application requests, taking into account the permissions of the user and tenant.** Every tenant can have its own URL, for example by having a customized sub-domain. When a client wants to access the data of a specific tenant, the access module of the application needs to verify if the authenticated user and its corresponding tenant have access to the requested data (the requested URL), to eliminate unauthorized access.

- **Separating the tenant configuration from tenant data.** Because the tenant data is stored in a different database instance as the tenant configuration, it is easier to configure tenant-specific access at the database level. Each tenant will have its own connection string, and the associated credentials will only have access to the tenant's database.

- **Offering single-tenant instances of specific components at a higher cost.** If the above methods are not deemed sufficient, tenants with a huge amount of confidential data can have single-tenant instances at a higher cost. Having a dedicated instance clearly improves security, as the tenant's data is not only virtually but also physically isolated from other tenants. Because the connection strings are stored separately for each tenant in the shared *tenant configuration database*, these connection strings can either point to a shared or dedicated database.

2.4 Case Study: Medical Communications System

2.4.1 Introduction

In this section we verify our presented approach using the case study of a Medical Communications (MC) system. The MC system is responsible for the correct functioning of all communication peripherals located in a medical environment. The central functionality of this system is the *nurse call* system. The basic concept of a *nurse call* system is simple: a call device is located in every room. When a button is pressed on the device, a message is sent to a controller after which nurses are notified of the call. This concept can be enhanced by using ontologies and semantic reasoning to identify the urgency of a call or select the nurses to notify in a more intelligent way [19] [20] [21].

A *nurse call* system consists of many different elements, installed within a hospital. These elements include amongst others (i) end user equipment installed in the rooms, which patients can use to contact hospital personnel, and terminals used by the personnel; (ii) embedded servers, used to communicate between the terminals and management servers; and (iii) servers for logging, registration and visualization. Figure 2.6 illustrates an example of the architecture of a nurse call

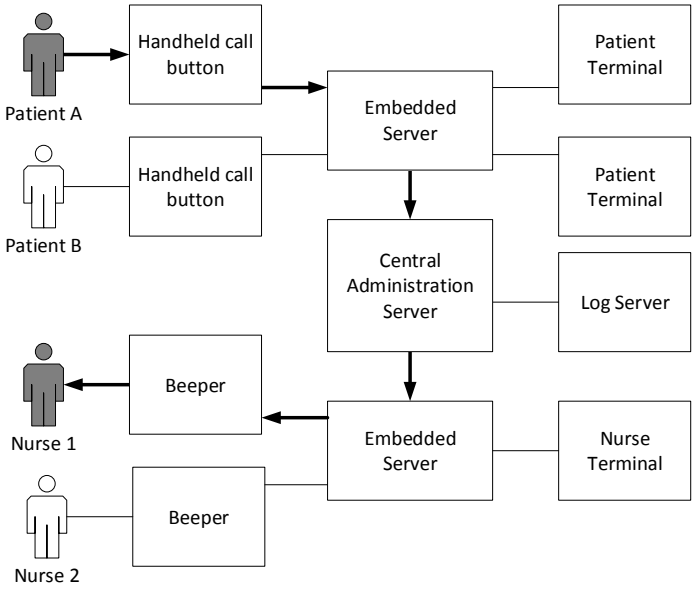


Figure 2.6: An example architecture of a nurse call system, with the communication between the different elements when a patient makes a call.

system, with the possible communication between the different components when a patients calls a nurse shown in arrows.

While the center of the MC application is the *nurse call* system, additional services, such as intercom, video over IP, access control and other health services are being offered as well. Currently, the MC system is installed in multiple locations, ranging from big hospitals to small nursing homes. The cost of installing dedicated servers, and the corresponding maintenance is quite high. Migrating a part of the system to the cloud will minimize the cost, by eliminating the need of many of the dedicated servers, making it possible for smaller hospitals and and nursing homes to afford the system. However, considering the medical use case, the MC application is subject to stringent security and performance constraints, which need to be taken into account when the components to migrate to the cloud are selected.

2.4.2 Cloud migration

2.4.2.1 Selecting Components

The MC software consists of two main components: the *device manager* and the *administration service*. The *administration service* is the main application, and is used to manage the different features and devices installed within the hospital. The *device manager* is a dedicated hardware box, running different modules mainly

written in C++ for communicating with the different peripherals installed within the medical environment. The modules for the different features are dynamically loaded on start-up, and can be configured from the *administration service*. Figure 2.7a shows the initial architecture of the application. The device manager and the different peripherals communicate over Ethernet, using a custom proprietary secure protocol.

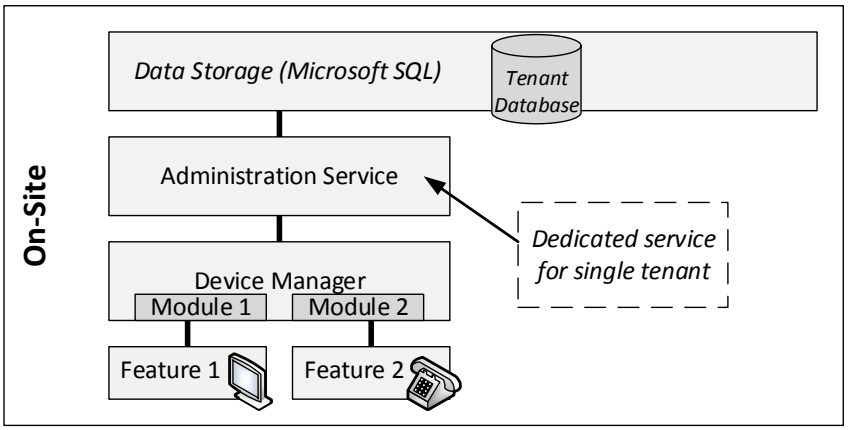
For our PoC, we selected the *administration service* and its corresponding database instances for migration to the public cloud. The MC system has a fallback mechanism, allowing the *device manager* to operate standalone in case the *administration service* is not available. Because the *device manager* communicates directly with the devices, migrating this component to the cloud would be tricky, as the devices need to operate when no connection to the public cloud is available. Passing all communication between the device manager and the peripherals over the public Internet would also result in slow response times, and could make the system unreliable. However, as most of the processing is done in the devices, a single *device manager* will be sufficient to control all devices in a small or medium environment. If the peripherals could be adjusted to work standalone when the device manager is unavailable, migrating the *device manager* could also be an option in the future, but for this PoC, we started focusing only on decoupling the *administration service*.

After adding multi-tenancy to the application and migration to the cloud, a new component is introduced, the *tenant configuration interface*. The reviewed architecture after adding multi-tenancy and migration is shown in Figure 2.7b. Because every tenant has its own features, the user interface of the *administration service* is automatically adapted for the different clients based on the tenant configuration.

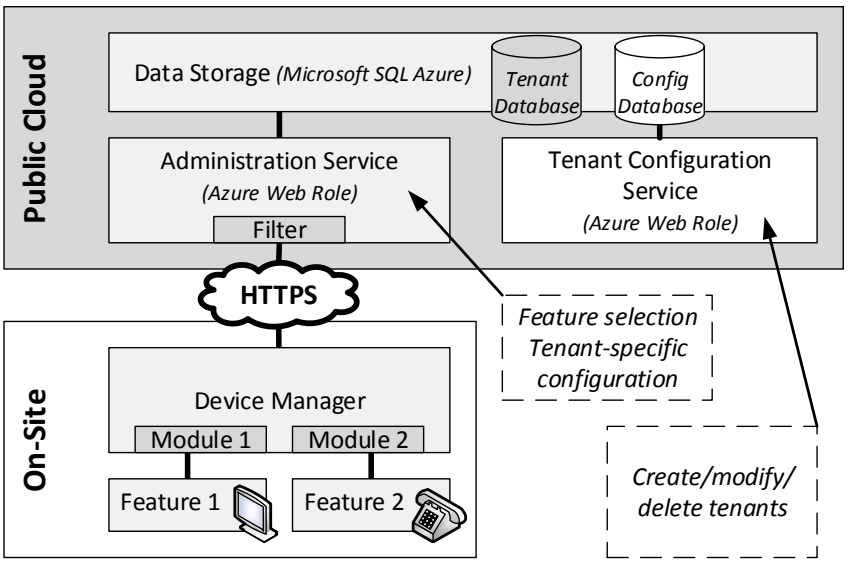
2.4.2.2 Determining Provider Compatibility

As the application is written in .NET, migrating the *administration service* to Microsoft Azure seemed like an evident choice. Microsoft Azure [22] currently offers two roles to choose from when creating an instance, web roles and worker roles, both based on Windows Server. The main difference between these two is that an instance of a web role runs IIS, while an instance of a worker role does not. In addition to the type of instances, Azure offers different sizes for both roles [23]. Table 2.1 gives an overview of the different standard instances available on Azure.

Both the *administration service* and the *tenant configuration interface* will be running on an Azure web role. While preparing the application for migration, these Azure web roles need to be added to the .NET project, and can be tested in the Azure simulator. When using a third party assembly in the project, this assembly should be added as a reference to the project, with the Copy Local property set to true. A nice side effect of this process is that many deprecated libraries were removed or replaced in the project, making it much easier for developers to locally



(a) Initial architecture of the software before moving to the cloud.



(b) Architecture after migration to the cloud and adding support for multi-tenancy.

Figure 2.7: Architecture of the application before and after migration to the cloud and adding support for multi-tenancy.

Table 2.1: Overview of standard instances on Windows Azure.

Name	Virtual Cores	Ram
Extra Small (A0)	Shared	768 MB
Small (A1)	1	1.75 GB
Medium (A2)	2	3.5 GB
Large (A3)	4	7 GB
Extra Large (A4)	8	14 GB

install the application, as they no longer needed to configure and install third-party products on a clean environment before being able to compile and test the application.

The relational databases will be moved to SQL Azure. As a result, the connection strings inside the application should be altered to point to the SQL Azure instance. SQL Azure has some limitations compared to a dedicated Microsoft SQL Server instance, but for most .NET applications, this shouldn't be an issue. Once the application is running correctly in the Azure simulator, the project can be packaged and deployed onto Windows Azure [24] [25].

2.4.2.3 Determining Impact on Client Network

The traffic between the *administration service* and the *device manager* now has to pass the public Internet, and the internal network is also loaded with traffic between the *device manager* and the different peripherals. The total amount of traffic is depending on the selection of features, as some of the features might require more bandwidth. Both the internal network as the public internet connection need to have sufficient bandwidth to support the MC system to operate. The service described in [17] was customized to support this PoC, making it possible to predict if a custom selection of features would be able to run on the client network. For this PoC, the different topologies of the client networks were implemented statically, but we introduced the option to easily replace these static topologies by a dynamically generated topology, which could be generated by tools using existing network discovery protocols, such as Neighbor Discovery Protocol (NDP) and Link Layer Discovery Protocol (LLDP).

2.4.2.4 Scaling the Application

Azure allows the administrator to configure multiple instances with automatic load balancing, which will be required as the number of tenants grow. Recently, limited possibilities were added to Azure for automatic scaling, using the *Autoscaling Application Block* [26]. Alternatively, the creation and deletion of extra instances

can be done manually (or in code) by the customer. Some third party products also exist, like AzureWatch [27], which will handle the scaling automatically, or the SaaS provider can create a customized system, for example by using advanced load prediction.

2.4.3 Multi-Tenancy

2.4.3.1 Decoupling Databases

In the initial single-tenant architecture, there is a dedicated relational database for every instance. The connection string to this database is hard-coded in the configuration file (Web.config). To support multi-tenancy, we introduced dynamic connection strings, stored in the *Tenant Configuration Database*. The connection string in the configuration file was replaced by a connection string to this shared database.

To support both shared and dedicated databases, we added an extra column in the data tables, holding the identification of the tenant (*tenantID*). By doing so, the application itself doesn't need to know if the database is shared or dedicated, as multiple tenants can share the same connection string. The *Data Access Component* introduced in Section 2.3 is now responsible to select the correct tenant's data, for example by filtering on the corresponding *tenantID*.

2.4.3.2 Adding Tenant Configuration Database

The *Tenant Configuration Database* is introduced to store the general information about the different tenants. It holds the connection strings for each tenant, together with some contact and billing information, and the feature selection for the tenant. As the *administration service* only needs to get this information at start-up, read-only access to this database is sufficient for the main application. This also eliminates the risk of tenants modifying the configuration of other tenants. Figure 2.8 illustrates this by giving an overview of the possible communication between actors and components within the system.

2.4.3.3 Providing Tenant Configuration Interface

A new application is introduced, the *Tenant Configuration Interface*, used by tenant administrators (like resellers or the application provider) to setup and configure the different tenants. This application has write access to the *tenant configuration database*, but as only tenant administrators have access to this application, there is no risk of tenants modifying the configuration of other tenants, or even their own configuration, making the system unusable. For this PoC we didn't spend too much time to build a full-blown interface, but in the final version enough time

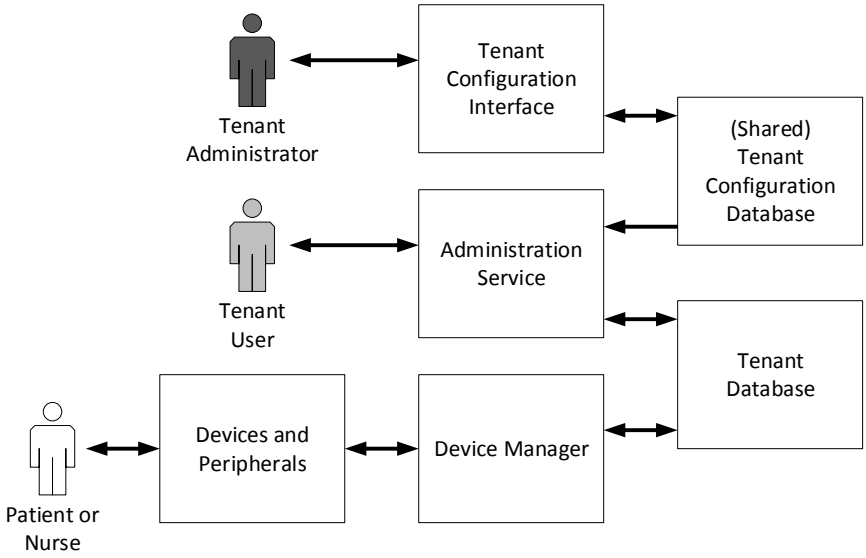


Figure 2.8: An overview of the possible communication between actors and the different components of the medical communications system.

should be spent building this application, as it is a key component in the multi-tenant application which can dramatically minimize the time needed to configure and modify new or existing tenants. The tenant configuration interface was designed as a web application running on an Azure Web Role, but to mitigate security risks, this interface could also be developed as an internal mobile or desktop application, accessing the *tenant configuration database* through web services.

2.4.3.4 Dynamic Feature Selection

The nurse call feature is the core feature of our MC system, but some other features are also implemented, for example voice and video calling between different rooms using Voice over IP (VoIP), and door access control with badges used by the hospital personnel. The selection of features for a single tenant depends on the available hardware and peripherals within the hospital, and the available bandwidth of both the internal and external network. The selection of features and general/technical configuration is done by a tenant administrator through the *tenant configuration interface*, while the tenant-specific configuration of the features can be done by different tenant users through the *administration service*. The initial application (*administration service*) was designed to support dynamic loading of the required libraries and modules at start-up. The modules kept running during the lifetime of the application, but as this application was installed on a dedicated

instance with a lot of available resources, this was not really an issue. Converting the application to a multi-tenant application however introduced some new challenges. As every tenant can have its own selection of features, all features might need to be loaded on the single machine, and if the multi-tenant application is not well designed, some features might even be loaded multiple times. To overcome this issue, some changes are needed to the application:

- The required libraries and modules for a specific tenant are loaded as soon as a user logs in to the *administration service*.
- Libraries and modules should be loaded only once, and hence can be shared between different tenants.
- Loaded libraries and modules should be freed as soon as they are not used anymore, for example after a timeout, to eliminate the usage of unnecessary resources.

2.4.3.5 Managing Tenant Data, Users and Roles

As already indicated in Figure 2.8, there are different users and roles used in the MC system:

- The tenant administrators (application provider, resellers, installers), having access to the *tenant configuration interface*. These users and their corresponding roles are stored in the *tenant configuration database*.
- The tenant users and their corresponding roles (mostly personnel of the different hospitals). Because every tenant can have its own users and roles, these are stored in the *tenant database*.
- The patients don't really require roles, but are in a way guest users of the system. The peripherals however could count as visualized users with customized roles, and can also be stored in the *tenant database*, together with the tenant users and roles.

2.4.3.6 Mitigating Security Risks

Some of the security risks and a way to eliminate these risks are already described in the previous steps. To increase the security, we added URL-filtering to the application, and altered the access module to take into account the requested URL (and hence the identification of the specific tenant) and the authorized user and its corresponding tenant ID. The traffic between the *device manager* and the *administration service* and *tenant database* now passes the public internet and is secured by using HTTPS over SSL/TLS. Every tenant can have a dedicated *tenant*

database, increasing the isolation of data, but this comes at a higher cost. In practice, big hospitals will typically have a dedicated database, and data from smaller nursing homes belonging to the same entity (subtenants of the same tenant) will be co-located in shared databases. This way, we won't be mixing data from subtenants belonging to different tenants, and isolation of data is always guaranteed at tenant level.

2.5 Case Study: Medical Appointments Schedule Planner

2.5.1 Introduction

As a second case study, we migrated a medical appointments schedule planner to public cloud environments. This planner is used by both patients and medical staff to manage their appointments. The software was originally developed as a single-tenant application. Figure 2.9 illustrates the original layered architecture of the application. The end-users (patients) access the web application through the user portal, in order to manage their medical appointments. The application is running on a shared web server, the appointments and patient data are stored in a dedicated database on a shared database server. Medical staff access the application through the admin portal in order to approve and review the requested appointments.

As multiple clients started using the software, multiple independent copies of the software were installed and configured, running different versions, increasing maintenance complexity. Independent copies were deployed on the same shared web server and the average load increased over time, resulting in an increase in page load times due to the large amount of data and the required amount of data processing by the application. For this case study, we added multi-tenancy to the application to optimize the utilization of available hardware resources, and migrated the application to the public cloud environment in order to centralize the management and to increase the scalability. We deployed the application on two different cloud providers in order to compare the performance and the ease of deployment.

2.5.2 Cloud Migration

2.5.2.1 Selecting Components

The schedule planner consists of two main components: the user portal, used by patients to request medical appointments, and the admin portal, used by medical staff to approve and review the requested appointments and to manage their schedule. Both patients and medical staff can synchronize their appointments to their

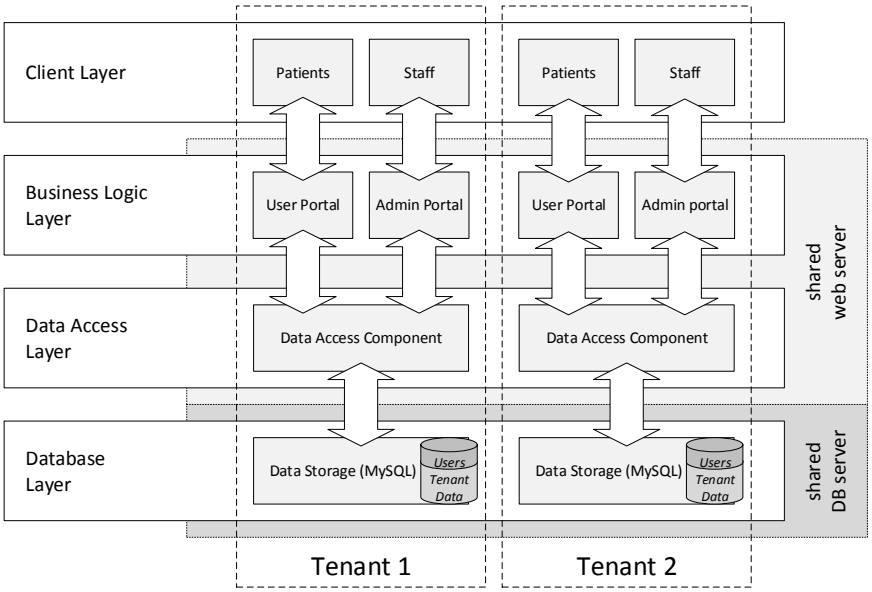


Figure 2.9: Pre-migration single-tenant architecture of the medical appointments schedule planner.

personal calendar using one of the available standard calendar formats, and confirmations and reminders are sent by email or by text message (SMS). Different departments are using this portal, but a department may only access patient information relevant for their appointments. Patients on the other hand can browse and request appointments at the different departments.

For this second case study, we selected the whole application for migration to a public cloud provider. This application is less sensitive for short downtime periods as the MC application from the previous case study, as both patients and medical staff have offline copies of their appointments. Therefore, no additional fallback mechanism is necessary inside the application.

The legacy application was developed to be used by a single medical department or an independent doctor (a single tenant), and independent copies of the software were installed and configured. After adding multi-tenancy to the legacy application, a single instance of the application is now shared between multiple tenants, and a new component is introduced, the *tenant configuration interface*, with a similar functionality as the interface from the previous case study.

2.5.2.2 Determining Provider Compatibility

The legacy web application is developed using HTML5, Hypertext Preprocessor (PHP) and Oracle MySQL for the persistent storage of data, and is executed on

Table 2.2: Overview of the available T2 instance types on Amazon EC2.

Model	Virtual Cores	Ram
t2.micro	1	1 GB
t2.small	1	2 GB
t2.medium	2	4 GB

Table 2.3: Overview of the available M3 instance types on Amazon EC2.

Model	Virtual Cores	Ram
m3.medium	1	3.75 GB
m3.large	2	7.5 GB
m3.xlarge	4	15 GB
m3.2xlarge	8	30 GB

a shared web server. For evaluating our approach, we migrated the application to both a PaaS and IaaS environment. We selected Google AppEngine [28] as PaaS provider as they provide a PHP Runtime Environment, and Amazon EC2 [29] as IaaS provider. Google AppEngine requires more changes to the legacy application as it puts more constraints on applications, while Amazon EC2 requires more maintenance as they provide full control over the virtual machine.

Migrating an application to Amazon EC2 is straightforward. Amazon currently offers two types of EC2 instances, the T2 instances which are burstable performance instances for development environments and early product experiments, and the M3 instances, which provide a good balance of compute, memory, and network resources. The different types of T2 and M3 instances currently available are listed in Tables 2.2 and 2.3 respectively. For our PoC, we selected a t2.micro instance running Ubuntu Server 14.04 for both the database and web server. We configured Apache and MySQL on the instance and deployed the application, and except from some configuration settings, no changes were required in the application.

Migrating the legacy PHP application to Google AppEngine on the other hand required multiple changes to the application as summarized in Table 2.4. First of all, as Google offers Cloud SQL instead of MySQL, some changes are required in the application to connect to the Cloud SQL database instance [30]. The original MySQL database can be exported to a file using a SQL dump, and this file can be used to import the data into a new Cloud SQL database instance. The mysqli extension introduced with PHP version 5.0.0 can still be used to connect to the database, but the connection string differs from a traditional connection string as

Table 2.4: Overview of the most important changes for migration to Google AppEngine.

Item	Description
Relational Data	Migrate MySQL databases to Google Cloud SQL and modify connection strings
Temporary Files	Replace local file storage by storage buckets on Google Cloud Storage
URL Rewriting	Replace `mod_rewrite` by a custom PHP script providing similar functionality

illustrated in [30].

In AppEngine, the local file system that the application is deployed to is not writable. However, if the application needs to write and read files at runtime, AppEngine provides a built-in Google Cloud Storage (GCS) stream wrapper that allows many of the standard PHP file system functions. A PHP application running on AppEngine can read and write files by using buckets as illustrated in [31]. The legacy application was developed using the Smarty PHP Template engine [32], which requires different physical file directories for reading and writing templates and configuration files. As a result, the Smarty engine needs to be reconfigured to use the GCS for storing the compiled templates and files. One major difference between writing to a local disk and writing to GCS is that GCS does not support modifying or appending to a file after closing it. Instead, a new file can be created with the same name, which overwrites the original. For the Smarty PHP Template engine however this is not really an issue as it only creates temporary files which are not modified after creation. Using buckets to store temporary files can have an influence on the performance of the application. There is no straightforward way to measure this impact, as local file storage is not supported by Google AppEngine. In Section 2.6 we however do compare the performance of the application running on Google AppEngine with other environments which are using traditional file storage.

Finally, the legacy application implemented URL rewriting by invoking the `mod_rewrite` module of Apache. As Google AppEngine does not support this module, this functionality has to be simulated through the use of a PHP script referenced from the application's configuration file (app.yaml) that will in turn load the desired script, as described in [33]. The overhead introduced by this script is minimal, as it just parses the requested Uniform Resource Identifier (URI) and executes a simple conditional statement. Google however recommends to rewrite the application to operate without the `mod_rewrite` module, but this requires more effort as more changes to the source code are required.

Once the application is running correctly in the simulated environment of

Google App Engine Launcher (part of the Google App Engine SDK), it can be deployed onto the public cloud.

2.5.2.3 Determining Impact on Client Network

As the original application was already designed to be accessed over the web, and the full application is migrated to the cloud, there is no real impact on the client network after migration to the public cloud.

2.5.2.4 Scaling the Application

Amazon offers CloudWatch [34] to monitor Amazon Web Services (AWS) cloud resources and applications. This service provides a clear insight in the current demand using different metrics such as CPU utilization, data transfers and disk usage activity. Application developers can also create custom metrics, and customize automated actions and alarms. For a production-ready application on Amazon EC2, CloudWatch can be used to provide compliance with specific SLA targets, and to handle the automated scaling of both the computational resources.

Google AppEngine on the other hand has built-in support for high scalability. An application running on AppEngine can be deployed on multiple instances and instances are automatically created or removed depending on the current load. No action is required from the developer, but the developer has more limited control than with Amazon EC2. During our experiments as described in Section 2.6, multiple instances were automatically created.

2.5.3 Multi-Tenancy

After adding multi-tenancy to the application, the original architecture was slightly modified. Figure 2.10 illustrates the modified architecture after adding multi-tenancy and migration to a public cloud provider. These modifications are discussed in detail in the remainder of this section.

2.5.3.1 Decoupling Databases

In the initial single-tenant architecture, every tenant has a dedicated MySQL database on a shared database server. In order to support multi-tenancy, we introduced dynamic connection strings, as in the previous case study. All users are however stored in a single database, separated from the tenant databases. By doing so, a single user can access multiple tenants, and multiple copies of the same user object are eliminated.

As with the previous case study, we added an extra column to the data tables, holding the identification of the tenant (*tenantID*). By doing so, the application supports both shared and dedicated database instances, and multiple tenants can

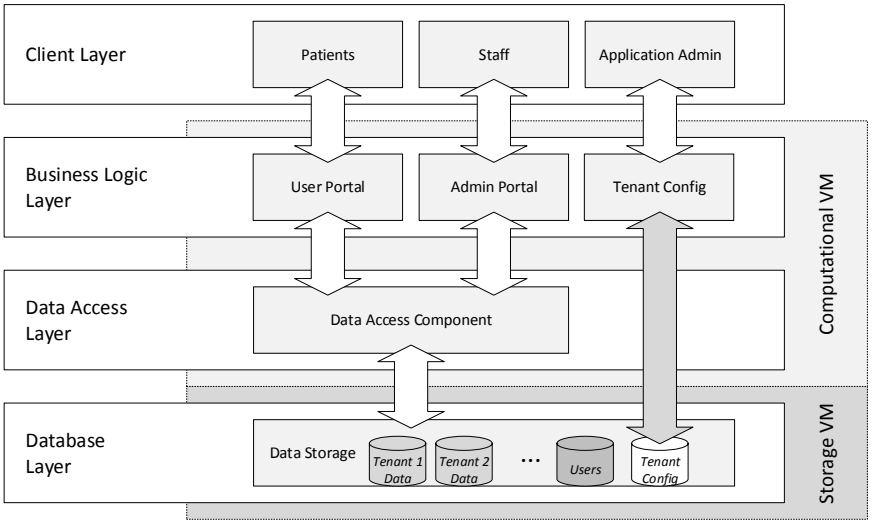


Figure 2.10: Revised architecture of the schedule planner after adding multi-tenancy to the application and migration to the public cloud.

share a single database. The data access component of the data access layer was modified to support the dynamic behavior of the tenant databases, and to filter tenant data based on the *tenantID*. This filtering is required in order to provide transparent isolation of tenant data, especially when multiple tenants are sharing a single database instance.

2.5.3.2 Adding Tenant Configuration Database

The shared *Tenant Configuration Database* contains general information about the different tenants, and a connection string to the database instance where the tenant data is stored. This database is small in size, which is why it is also used to store the different user objects. However, should this database ever become a bottleneck, the user data can easily be decoupled from the general tenant information, as separate connection strings are used for the user database and the tenant configuration database. For our PoC these connection strings refer to the same tenant configuration database instance.

2.5.3.3 Providing Tenant Configuration Interface

A tenant configuration interface was added to the application, and is used to manage the different tenants. As in the previous case study, this configuration interface

communicates directly with the tenant configuration database.

2.5.3.4 Dynamic Feature Selection

Tenants can have optional features enabled, for example notifications by text messages or export options to different calendar formats. A tenant administrator can configure these features through the tenant configuration interface. The feature configuration is stored in the tenant configuration database together with the general tenant information, and both the application's user portal and admin portal take the feature selection of the selected tenant into account.

2.5.3.5 Managing Tenant Data, Users and Roles

The user objects are stored in the shared tenant configuration database, the roles are stored together with the tenant data in a tenant database instance. This instance can either be a dedicated database instance or an instance that is shared between multiple tenants. Relevant medical information belonging to a certain patient is stored together with the role in the tenant database. By doing so, sensitive patient data is inaccessible by other tenants, as all data queries are filtered based on the *tenantID* by the data access component.

By using a single database instance to store all user objects, multiple roles for different tenants can be created for a single user object. This allows for a single sign-on, where the user object is loaded when the user logs in on the application, and the relevant roles are loaded when the user wants to access one of the tenant's restricted pages.

2.5.3.6 Mitigating Security Risks

As mentioned above, sensitive data belonging to a certain patient is stored together with the user role in the tenant database. As all data queries are filtered by the data access object based on the *tenantID*, queries can never return data belonging to different tenants.

Communication between a client computer and the web server is encrypted, as all communication uses HTTPS over SSL. This was already the case with the legacy application.

2.6 Discussion and Evaluation

Moving applications to the cloud and adding multi-tenancy introduces new opportunities for our presented use cases. First of all, there is the increased flexibility and elasticity. When the workload on the application increases, new instances can be created and deployed automatically. Similarly, when the workload decreases,

instances can again be removed. For new customers, deployment times decrease as there is no need to physically install a new server. By using a PaaS platform instead of IaaS, there is no need to install, configure and manage the guest OS, further reducing the deployment times. The hardware maintenance cost is also eliminated as the virtual machines running in the cloud are automatically migrated when the hardware fails. Combining multi-tenancy and migration to a public cloud makes maintenance easier, as the software is deployed centrally, and no on-site intervention is needed, for example to install patches or updates. Adding multi-tenancy to the application also improves the efficiency of resource utilization, decreasing the costs, and eliminates the need for installing and configuring independent copies of the same software, sometimes running different versions of the software. In this section, we will highlight some of the major advantages of the migration, and compare them with the overhead of the migration.

2.6.1 Increased flexibility and elasticity

As the amount of available resources in the cloud is quasi unlimited, application developers don't have to worry about selecting the right amount of resources to host the software. Novel multi-tenant applications can start with a single instance, and the number of instances can grow as the workload increases. In cloud computing, elasticity is defined as the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible. In our approach, we have presented some possibilities for building elastic applications in the step of *Scaling the Application*. For the two case studies, we also started with a single instance, and determined the possibilities to scale the application as the demand grows.

2.6.2 Decreased deployment time

The addition of multi-tenancy to the application and migration to a public cloud yields a significant decrease in deployment times, especially for new tenants. Tables 2.5 and 2.6 illustrate this for the MC software case study by giving an estimation of the needed deployment time for a new tenant, respectively before and after the migration process.

Before migration, a physical server was installed and configured on-site for every new tenant, together with the device manager and peripherals. A local copy of the *administration service* was installed and configured on the dedicated physical server together with a SQL Server instance. A total of 6 person-days was required to perform the installation and configuration of both the server and the *device manager* and peripherals.

Table 2.5: Initial configuration of new tenant before migration: time estimation for the MC software case study.

Task	Time
Install and configure on-site server for application	1 day
Deploy administration service on on-site server	1 day
Configure SQL server and initial tenant database	1 day
Configure on-site hardware (device manager and peripherals)	2 days
Verification and testing	1 day

Table 2.6: Initial configuration of new tenant after migration: time estimation for the MC software case study.

Task	Time
Create new tenant and initial tenant database	1 hour
Impact analysis on client network (automated)	1 hour
Configure on-site hardware (device manager and peripherals)	2 days
Verification and testing	1 day

After migration, the initial configuration time is largely reduced. Only the *device manager* and peripherals need to be installed, and the configuration of the *device manager* can be done remotely by using the multi-tenant *administration service* running on the cloud. As only a new tenant needs to be created, no local copy of the *administration service* needs to be installed and configured. An estimated total of 3.5 person-days is required for the initial setup after migration, mainly for the installation and configuration of the on-site *device manager* and the peripherals, and for full testing.

Migrating the *device manager* to the cloud could further reduce the deployment time, but this however introduces additional challenges which were mentioned before in Section 2.4.2.1. For completeness, Table 2.7 shows an estimation of the initial deployment of the application on Microsoft Azure. This initial deployment needs to be done only once, and not for every new tenant.

2.6.3 Ease of Maintenance

Having the core of the MC software, the *administration service*, hosted in the public cloud makes maintenance a lot easier, as installers no longer need to go on-site to make small configuration changes. Eventually, one could argue that having Virtual Private Network (VPN) connections to the customer sites could also bypass this, but this requires a VPN setup to the hospitals, or public access to the internal

Table 2.7: Initial deployment after migration: time estimation for the MC software case study.

Task	Time
Deploy administration service on Azure	2 hours
Deploy tenant configuration interface on Azure	2 hours
Create initial databases on SQL Azure	2 hours
Create tenant administrators	2 hours

network, which again introduces some security risks, and requires a stable external connection at the client side. Having a single multi-tenant application also has the advantage that every tenant uses the same version of the software, and software updates can be deployed centrally, for all tenants at once. For installing software updates, a second instance can be deployed and configured in an isolated environment on the public cloud, and switched with the current instance once the configuration and testing is done. Software updates and patches for the *device managers* can be pushed from the central *administration service*, as under normal circumstances, the *device manager* has a persistent connection to the *administration service* and will frequently check for updates.

For our second case study, the schedule planner, deploying the multi-tenant application on the public cloud results in similar benefits. Before adding multi-tenancy to the application, different independent versions were deployed, and updating and maintaining the application became harder as the number of tenants grew. After migration to the cloud, only one copy of the multi-tenant application is running on multiple instances in the cloud, and all tenants are running the latest software of the application.

2.6.4 Migration Cost

Migrating software to the cloud and adding multi-tenancy to the application comes at a cost. The architecture and code might need to be changed, and the software needs to be tested thoroughly. Extra attention needs to be paid to the security aspect, as the whole application or some components are now hosted remotely. For the MC software, we spent a total of six person-months to implement the changes described before in Section 2.4. For a production ready application in the cloud, an estimated additional 14 person-months would be required, as summarized in Table 2.8. The remaining tasks mainly focus on adapting the cloud application to support the existing SLAs, investigating possibilities for automatic backup and restore, providing training for installers and full testing.

For the schedule planner, our second case study, only two person-months were

Table 2.8: Tasks to be done for a production ready application in the cloud: time estimation for the MC software case study..

Task	Time
Azure-specific tasks	3 person-months
Explore monitoring options	
Define backup and restore strategies	
Verify SLA constraints on Azure	
Adapt cost model	
Administration Service	8 person-months
Improve security of service and features	
Make complete mapping for all features	
Migrate remaining features to Azure	
Add support for newest hardware nodes	
Study impact of shared vs. dedicated database instances	
Other remaining issues	
Impact Analyzer	1 person-month
Support dynamic generated topologies	
Other tasks	2 person-months
Training and development courses for developers	
Training for installers, retailers, clients	
Full testing of application	

required to implement the changes described in Section 2.5, as this application is less complex than the MC software. Adding multi-tenancy to the legacy application required a bit less than two person-months. For the migration to Amazon EC2 only 1 day was sufficient, whereas for Google AppEngine a few person-days were necessary to implement the required changes. For a production ready application in the cloud, an estimated additional 4 person-months would be required, for finishing the application and full testing.

2.6.5 Remaining Risks

As some components are now hosted on a public cloud, there is an increased security risk. For our first case study, the MC software, extra attention should be paid to the new risks introduced by moving the software to the public cloud. A side-effect of the migration is that the *Administration Service* is now hosted in the public cloud, and could become a bottleneck if the number of tenants grows significantly. Therefore, extra attention should also be paid to the scalability of this service. For our second case study, the main security risk is due to the addition of multi-tenancy, so the application needs to guarantee isolation of sensitive data, for

example by restricting all queries to filter data based on the *tenantID*.

2.6.6 Change in Cost Model

Migrating the software to the cloud brings a change in the cost-model of the MC software. Before the migration, the software and hardware was typically sold as a single package, including the necessary hardware, licenses for the software, initial installation and configuration. As most public providers work with monthly fees, the application provider should adapt the cost-model to reflect this cost model. Instead of selling a one-time license for the software, the end users can pay a monthly fee, depending on the size of the tenant, covering the hosting on Azure and future software changes and updates. The *Tenant Configuration Interface* can be designed to support this cost-model, and could be linked to the financial software. This change in cost model introduces a new opportunity by expanding the customer market, as smaller clients (for example small nursing homes) are now able to start using the software at a lower cost, as the costs of implementing the system can now easier be spread over time, less hardware is required on-premise, and computational resources are shared between multiple tenants.

The schedule planner software was already being sold using a license-based cost model. Adding multi-tenancy and migrating the application to the cloud introduces no visible changes in cost model. Sales prices could however drop as utilization of available resources is optimized by adding multi-tenancy to the application, and the infrastructure cost is reduced by using a public cloud provider.

2.6.7 Performance Comparison

The performance of an application running on the cloud depends on both the selected cloud provider and the selected instance type. We selected the second case study, the medical appointments schedule planner, for evaluating the performance of the selected cloud providers, as this application is fully migrated to the public cloud, whereas the MC application is only partially migrated, making the performance depend on more factors such as the on-site client network topology and capacity and on-site available hardware.

In order to evaluate the performance of the medical appointments schedule planner, we deployed the application to four different environments, as summarized in Table 2.9. We measured the average page generation time, which is the time needed for the PHP interpreter to generate the page, for different pages of the application. This metric does not take into account the network latency, as the generation time is measured at the server side by the PHP interpreter itself. We also measured the end-to-end transaction time, this is the total load time as perceived by the client, as this metric does include the network latency. The schedule

Table 2.9: An overview of the different deployment environments. The mentioned cost values are valid at the time of submission of this article.

Label	Description	Estimated cost
local	A dedicated virtual machine with 1GB Ram and 1vCPU, running on a physical linux server with a Linux server with an Intel Core i7 CPU (2.80 GHz) with 8 GiB of memory.	$\pm$20.00 USD/month + one-time infrastructure cost
shared	The shared web server on which the legacy application was running before migration to the public cloud.	1.82 USD/month
EC2	Amazon EC2 t2.micro instance	14.28 USD/month
AppEngine	Instance running on Google AppEngine	depends on usage

Table 2.10: Average page generation times (in seconds) and standard deviations for 3 test pages over 50 iterations.

	local		shared		EC2		AppEngine	
	$\overline{y}$	σ	$\overline{y}$	σ	$\overline{y}$	σ	$\overline{y}$	σ
Page 1	0.19588	0.00596	3.20709	0.81836	0.19557	0.00437	6.28543	0.84406
Page 2	0.20555	0.00688	3.00282	0.23948	0.19974	0.00372	6.38447	0.91441
Page 3	0.02882	0.00216	0.33663	0.03426	0.02183	0.00039	1.38429	0.56244

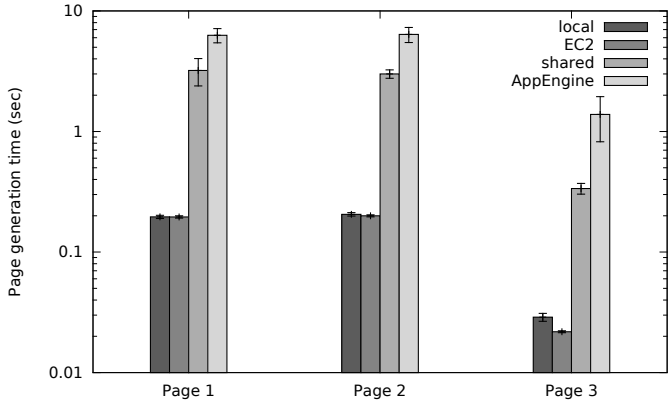


Figure 2.11: A comparison of the average page generation times for 3 test pages over 50 iterations.

Table 2.11: Average end-to-end transaction times (in seconds) and standard deviations for 3 test pages over 50 iterations.

	local		shared		EC2		AppEngine	
	$\overline{y}$	σ	$\overline{y}$	σ	$\overline{y}$	σ	$\overline{y}$	σ
Page 1	0.39160	0.01721	3.39200	0.74018	0.54840	0.10075	6.68200	0.79333
Page 2	0.37520	0.00934	3.11800	0.23113	0.47580	0.08840	7.03000	0.83211
Page 3	0.22260	0.00439	0.43580	0.03662	0.40920	0.03746	1.57200	0.56211

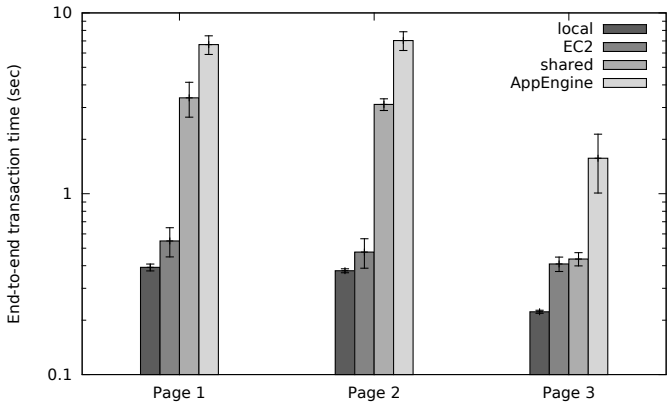


Figure 2.12: A comparison of the average end-to-end transaction times for 3 test pages over 50 iterations.

planner application was configured with a custom database based on data from existing production databases, combining historical data from different tenants over the last 3 years. We selected three specific pages for this experiment. The first two pages perform complex merge operations on the tenant data, as these pages were reported by existing users as being too slow. The third page is a normal page with an average load time. The experiments were executed on the cloud platforms in January 2015. Table 2.10 provides the measured average page generation times together with the standard deviations for the selected test pages over 50 iterations, and Figure 2.11 illustrates the same results graphically. Table 2.11 and Figure 2.12 are similar, but for the end-to end transaction times.

As can be seen from these results the Amazon EC2 Engine provides a good performance, as it is even faster than the local VM, even though we used the lightest instance available, a t2.micro instance. The Google AppEngine on the other hand is rather slow, as it takes up to 7 seconds to generate one of the selected heavy pages. A possible explanation for this is that PHP support by Google AppEngine is

still experimental, and the engine is not yet optimized for PHP. We however would like to note that the performance of Google AppEngine has already improved considerably over the last months, as in September 2014 the similar experiments were executed, and the same page could then take up to 45 seconds to generate. For a production ready application in the cloud, Amazon EC2 will however be selected to host the application, as it currently is a clear winner in the executed experiments.

2.7 Conclusions

Cloud computing and multi-tenancy allow providers to improve the scalability of applications while reducing hosting costs. In this chapter, we presented a generic approach for migrating legacy software to the public cloud, and adding multi-tenancy to the application. We described the different steps needed to convert the dedicated application to a cloud application, and the steps required to add multi-tenancy to the application. We verified our approach using two case studies from medical software: medical communications software and a medical appointments schedule planner. For the MC software, we migrated some components to a public cloud provider, creating a hybrid cloud, whereas for the schedule planner, we did a full migration of the legacy software to two different public cloud providers.

Migrating an application to the public cloud only requires a limited number of changes, while the conversion from a single-tenant to a multi-tenant application requires more steps as the latter requires limited changes to the application architecture. These modifications are however necessary to fully benefit from the opportunities of public cloud computing. We presented a proactive approach by identifying and eliminating possible future risks, for example by mitigating security risks and analyzing the architecture regarding its scalability.

In our evaluation, we described the advantages of both the cloud migration and the addition of multi-tenancy, and took into account the costs of the migration and remaining risks. After migrating the MC software, the time needed for the initial creation of a new tenant is strongly reduced (from 6 to an estimated 3.5 person-days, including the initial setup of the dedicated hardware), and maintenance has become much easier after migration. The reduction in initial costs and management costs also enables supporting smaller clients for which the costs used to be prohibitive. Supporting these additional clients may present new business opportunities in the long-term.

For scenarios where the performance is critical, different public cloud providers should be considered and evaluated, and within a single provider different instance types might exist. For our second use case, Amazon EC2 has a clear advantage over Google AppEngine for running the schedule planner, and yielded even better results than a dedicated virtual machine on a physical Linux server. The ad-

vantages of using a PaaS provider should also be weighted against the increased control gained when using an IaaS provider.

Migrating legacy software to the cloud comes at a cost, and some application components may need to be modified or rewritten. However, by following the multi-step migration approach presented in this chapter, the benefits of a cloud migration could outweigh the costs of implementing the described changes, as can be seen in the evaluation section of this chapter.

References

[1] M. Armbrust, R. Fox, Armandoand Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. *Above the Clouds : A Berkeley View of Cloud Computing*. Technical report, University of California at Berkley, 2009.

[2] D. Schatzberg, J. Appavoo, O. Krieger, and E. Van Hensbergen. *Scalable Elastic Systems Architecture*. In Proceedings of the ASPLOS Runtime Environment/Systems, Layering, and Virtualized Environments (RESoLVE) Workshop, pages 1 – 2, Long Beach, California, March 2011. Available from: http://communities.vmware.com/docs/DOC-14979.

[3] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao. *A Framework for Native Multi-Tenancy Application Development and Management*. In 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., pages 551 – 558, Tokyo, Japan, July 2007.

[4] P.-J. Maenhaut, H. Moens, M. Verheye, P. Verhoeve, S. Walraven, W. Joosen, and V. Ongenae. *Migrating Medical Communications Software to a Multi-Tenant Cloud Environment*. In IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), pages 900–903, May 2013. Available from: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6573107.

[5] M. Hajjat, X. Sun, Y.-w. E. Sung, D. Maltz, S. Rao, K. Sripanidkulchai, and M. Tawarmalani. *Cloudward Bound: Planning for Beneficial Migration of Enterprise Applications to the Cloud*. In Proceedings of the ACM SIGCOMM 2010 conference, pages 243–254, New Delhi, India, 30 August–3 September 2010.

[6] A. Li, X. Yang, S. Kandula, and M. Zhang. *Comparing Public-Cloud Providers*. Internet Computing, IEEE, 15(2):50–53, 2011. doi:10.1109/MIC.2011.36.

[7] A. Khajeh-Hosseini, D. Greenwood, and I. Sommerville. *Cloud Migration: A Case Study of Migrating an Enterprise IT System to IaaS*. In 2010 IEEE 3rd International Conference on Cloud Computing, pages 450–457, Miami, FL, July 2010. IEEE. doi:10.1109/CLOUD.2010.37.

[8] Q. H. Vu and R. Asal. *Legacy Application Migration to the Cloud: Practicability and Methodology*. In 2012 IEEE Eighth World Congress on Services, pages 270–277, Honolulu, HI, June 2012. doi:10.1109/SERVICES.2012.47.

[9] S. Walraven, E. Truyen, and W. Joosen. *Comparing PaaS offerings in light of SaaS development*. Computing, October 2013. Available from: https://lirias.kuleuven.be/handle/123456789/413659, doi:10.1007/s00607-013-0346-9.

[10] P. J. P. da Costa and A. M. R. da Cruz. *Migration to Windows Azure - analysis and comparison*. Procedia Technology, 5(0):93 – 102, 2012. Available from: http://www.sciencedirect.com/science/article/pii/S2212017312004422, doi:10.1016/j.protcy.2012.09.011.

[11] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. *Feature Placement Algorithms for High-Variability Applications in Cloud Environments*. In Proceedings of the 13th Network Operations and Management Symposium (NOMS2012), pages 17–24, Maui, HI, April 2012.

[12] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. *Developing and Managing Customizable Software as a Service Using Feature Model Conversion*. In Proceedings of the 3rd IEEE/IFI Workshop on Cloud Management (CloudMan), pages 1295–1302, Maui, HI, April 2012.

[13] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. *Cost-Effective Feature Placement of Customizable Multi-Tenant Applications in the Cloud*. Journal of Network and Systems Management, pages 1–42, 2013.

[14] H. Moens and F. De Turck. *Feature-based Application Development and Management of Multi-tenant Applications in Clouds*. In Proceedings of the 18th International Software Product Line Conference - Volume 1, SPLC '14, pages 72–81, New York, NY, USA, 2014. ACM. doi:10.1145/2648511.2648519.

[15] S. Walraven, E. Truyen, and W. Joosen. *A Middleware Layer for Flexible and Cost-efficient Multi-Tenant Applications*. In Middleware 2011, pages 370–389, Lisbon, Portugal, December 2011.

[16] P.-J. Maenhaut, H. Moens, V. Ongenae, and F. D. Turck. *Scalable User Data Management in Multi-Tenant Cloud Environments*. In 10th International Conference on Network and Service Management 2014 (CNSM 2014), Rio de Janeiro, Brazil, November 2014.

[17] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. *Network-Aware Impact Determination Algorithms for Service Workflow Deployment in Hybrid Clouds*. In Proceedings of the 8th Conference on Network and Service Managment (CNSM2012), pages 28–36, Las Vegas, USA, October 2012.

[18] P.-J. Maenhaut, H. Moens, M. Decat, J. Bogaert, B. Lagaisse, W. Joosen, V. Ongenae, and F. D. Turck. *Characterizing the Performance of Tenant Data Management in Multi-Tenant Cloud Authorization Systems*. In Proceedings of the 14th Network Operations and Management Symposium (NOMS2014), Krakow, Poland, may 2014.

[19] F. Ongenae, P. Duysburgh, M. Verstraete, N. Sulmon, L. Bleumers, A. Jacobs, A. Ackaert, S. De Zutter, S. Verstichel, and F. De Turck. *User-driven design of a context-aware application: an ambient-intelligent nurse call system*. In 2012 6th International Conference on Pervasive Computing Technologies for Healthcare, pages 205–210, San Diego, CA, USA, May 2012. IEEE. Available from: http://dx.doi.org/10.4108/icst.pervasivehealth.2012.248699.

[20] F. Ongenae, M. Claeys, T. Dupont, W. Kerckhove, P. Verhoeve, T. Dhaene, and F. De Turck. *A probabilistic ontology-based platform for self-learning context-aware healthcare applications*. EXPERT SYSTEMS WITH APPLICATIONS, 40(18):7629–7646, 2013. Available from: http://dx.doi.org/10.1016/j.eswa.2013.07.038.

[21] F. Ongenae, A. Hristoskova, E. Tsiporkova, T. Tourwé, and F. De Turck. *Semantic reasoning for intelligent emergency response applications*. In 10th International Conference on Information Systems for Crisis Response and Management, Abstracts, pages 1–2, Baden-Baden, Germany, May 2013.

[22] *Introducing Windows Azure*, 2013. Last accessed: January 2015. Available from: http://www.windowsazure.com/en-us/develop/net/fundamentals/intro-to-windows-azure/.

[23] *Azure Pricing*, 2013. Last accessed: January 2015. Available from: http://www.windowsazure.com/en-us/pricing/details/cloud-services/.

[24] D. Betts, S. Densmore, M. Narumoto, E. Pace, and M. Woloski. *Moving Applications to the Cloud on Microsoft Windows Azure*. Microsoft ; O'Reilly distributor, 2010.

[25] *How to Create and Deploy a Cloud Service*. Last accessed: January 2015. Available from: http://www.windowsazure.com/en-us/manage/services/cloud-services/how-to-create-and-deploy-a-cloud-service/.

[26] *How to Use the Autoscaling Application Block*. Last accessed: January 2015. Available from: http://www.windowsazure.com/en-us/develop/net/how-to-guides/autoscaling/.

[27] *Monitoring and Autoscaling features for Windows Azure with AzureWatch indepth - AzureWatch*. Last accessed: January 2015. Available from: http://www.paraleap.com/azurewatch.

[28] *Google App Engine - Free Platform-as-a-Service (PaaS)*. Last accessed: January 2015. Available from: https://cloud.google.com/appengine/.

[29] *AWS - Amazon Elastic Compute Cloud (EC2) - Scalable Cloud Hosting*. Last accessed: January 2015. Available from: http://aws.amazon.com/ec2/.

[30] *Using Google Cloud SQL - PHP*. Last accessed: January 2015. Available from: https://cloud.google.com/appengine/docs/php/cloud-sql/.

[31] *What is Google Cloud Storage? - Google Cloud Storage*. Last accessed: January 2015. Available from: https://cloud.google.com/storage/docs/overview.

[32] *PHP Template Engine - Smarty*. Last accessed: January 2015. Available from: http://www.smarty.net/.

[33] *Simulate Apache mod_rewrite routing - PHP*. Last accessed: January 2015. Available from: https://cloud.google.com/appengine/docs/php/config/mod_rewrite.

[34] *Amazon CloudWatch - Cloud & Network Monitoring Services*. Last accessed: January 2015. Available from: http://aws.amazon.com/cloudwatch/.

3

A Dynamic Tenant-Defined Storage System for Efficient Resource Management in Cloud Applications

In Chapter 2, we investigated how applications can benefit from deployment in a multi-tenant cloud environment. To achieve high scalability, these applications can benefit from the concepts introduced by Software-Defined Storage (SDS), by managing the allocation of data storage from the tenant's perspective. A multi-tenant application should guarantee both data separation and performance isolation towards every tenant, and migration of tenant data over time should be minimized as this is both an expensive and time consuming operation. This is also applicable at the infrastructure level, as cloud applications are often deployed inside virtual machines that are using virtual disks as storage. Both the number of virtual disks and their corresponding sizes will grow over time, introducing the need for a dynamic storage allocation approach. In this chapter, we present a generic, dynamic and extensible system for the management of storage resources. In the presented approach, tenants are hierarchically clustered based on multiple scenario-specific characteristics, and allocated to storage resources using a hierarchical bin packing algorithm (static allocation). As the load changes over time, the system responds to these changes by reallocating storage resources when required (dynamic reallocation). We evaluate both the static and dynamic behavior of our system. An extension for the presented system can be found in Appendix A, in which we focus on the distribution of database records over multiple relational

database instances.

⋆ ⋆ ⋆

P.-J. Maenhaut, H. Moens, B. Volckaert, V. Ongenae and F. De Turck

Published in Elsevier Journal of Network and Computer Applications (JNCA), Volume 93, pages 182–196, September 2017.

3.1 Introduction

Cloud computing enables short-term elastic usage of system resources such as processing power and storage, eliminating up-front costs and providing near infinite capacity on-demand [1]. With public cloud computing, infrastructure providers usually apply a cloud pricing model in which the customer is charged based on the actual resource usage (pay-as-you-go pricing model), making optimal usage of the available resources desirable. Multi-tenancy [2] refers to a set of features that enables the sharing of a common set of resources among customers, referred to as tenants. Adding multi-tenancy to cloud applications aids to minimize operation costs as resources are being shared, and also offers higher scalability as there is an increased utilization of the available resources [3–5]. The multi-tenant application can run on a small number of instances that are shared between tenants, and smaller tenants can be co-located on a single instance. In fact, cloud computing itself is a form of multi-tenancy, as all common cloud computing models rely on sharing of resources among multiple clients. Moreover, a mapping can be made between the different models of cloud computing and the different levels of multi-tenancy, as illustrated in Figure 3.1.

A typical multi-tenant cloud application consists of multiple types of resources, such as the computational resources used to execute the application's logic, network resources to provide sufficient bandwidth and storage resources required to store the application's persistent (tenant) data. Software-Defined Storage (SDS) [6] is an evolving concept for the management of data storage from the software's perspective, independent of the underlying hardware. An SDS system manages the policy-based provisioning of data storage, and virtualization is often used to provision the required resources. Multi-tenant applications running on the public cloud can benefit from the concepts introduced by SDS by managing the allocation of tenant data from the tenant's perspective, taking custom tenant policies and preferences into account. Within the cloud, customers currently often have no way to specify their requirements regarding the storage of sensitive tenant data, introducing the need for an extensible storage system for multi-tenant cloud applications.

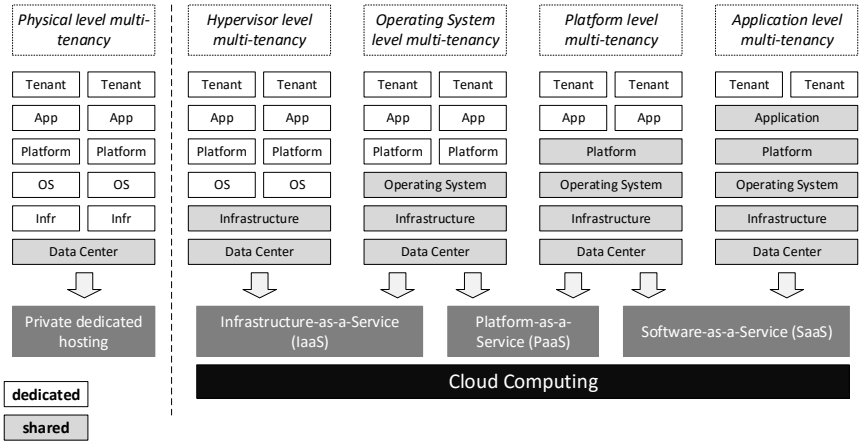


Figure 3.1: Mapping between the different models of cloud computing and the different levels of multi-tenancy.

Besides the intelligent provisioning and management of resources, reallocation of data over time should also be minimized. This is especially true for environments in which the application servers are geographically distributed, making migration of data both expensive and time consuming. Furthermore, although resources are being shared among multiple tenants, the storage system should behave like a private instance towards every tenant by guaranteeing both data separation and performance isolation [4].

In this chapter we present an implementation of an SDS system for the management of storage resources in multi-tenant cloud environments, which we refer to as Tenant-Defined Storage (TDS). The system allocates storage from the tenant's perspective, using novel bin packing approximation algorithms which are optimized for both the static and dynamic allocation. An elastic cloud already provides a good basis for hosting scalable multi-tenant applications that can adapt to changes in demand over time. The allocation of data, especially sensitive tenant data, in compliance with regulatory policies however remains a key hurdle [7, 8]. Tenant data can be stringent to certain data assurance policies in order to meet legal and business data archival requirements, but often customers have no way to specify their requirements. The TDS system should be flexible enough to support these and other policies and constraints, such as guaranteeing the selected Service-Level Agreement (SLA).

In most scenarios, storage resources will be tightly coupled to computational resources. A multi-tenant application running on the cloud could for example consist of multiple Virtual Machine (VM) instances. Initially, a single datacenter or even a single VM could be used to execute the application, but as the number of

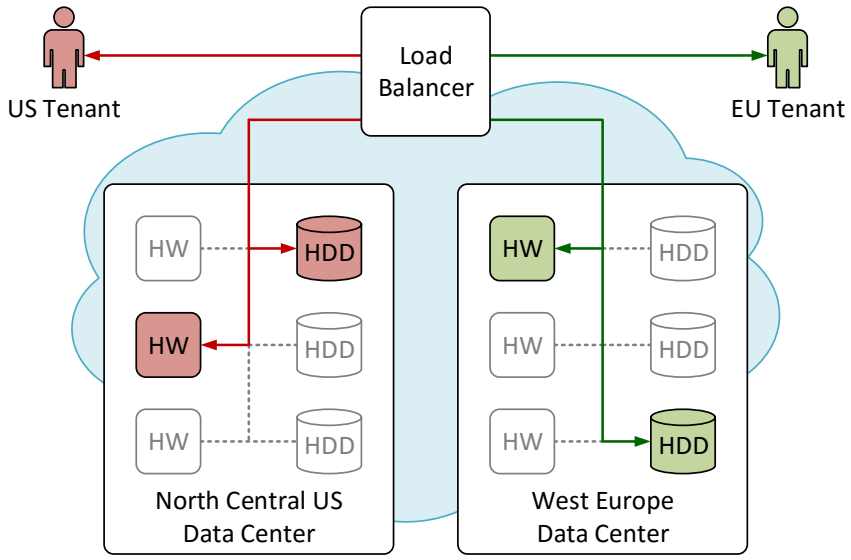


Figure 3.2: Example mapping of tenants to provisioned computational and storage resources for a distributed multi-tenant application running in the cloud.

tenants and therefore the load on the VM increases over time, multiple geographically distributed instances could be provisioned. This might however require the migration of existing VMs to a different location, resulting in the migration of some blob storage, the provisioned virtual disks. Figure 3.2 illustrates this concept. In this example, computational resources are represented by hardware (HW in the figure), these are the physical servers hosting the VMs, and storage resources are represented by disks (HDD), storing the virtual disk images. A tenant user located in the US is redirected to a provisioned VM in the North Central US Data Center, whereas for the European tenant user resources are provisioned in the West Europe Data Center. The tenant data should be stored close to the selected application server, preferably in the same data center, to minimize network latencies.

The remainder of this chapter is structured as follows. In the next section, we discuss related work within the field. Next, in Section 3.3 we provide a general overview of the TDS system and discuss the major design decisions. In Section 3.4 we introduce several static and dynamic bin packing algorithms for the (re)allocation of tenant storage. In Section 3.5 we define the relevant evaluation metrics and present the most significant evaluation results and discuss these in Section 3.6. Finally, in Section 3.7 we state our conclusions and discuss avenues for future research.

3.2 Related Work

3.2.1 Previous Work

In previous work [9, 10], we designed a data management framework which can be used to extend existing multi-tenant cloud applications in order to achieve high scalability of the database layer. This database layer consists of multiple relational databases, and the framework manages the distribution and retrieval of tenant data over the available instances. Furthermore, it guarantees the correct functioning of complex data queries, as aggregate data could be distributed over multiple database instances. In this chapter, we focus on the design of a TDS system for managing the allocation of blob storage instead of relational data, which has been briefly introduced in [11]. Allocating blob storage instead of relational data leads to a very different approach for the allocation of tenant data, with a strong focus on data isolation (as this is an important aspect of multi-tenancy), and the design of a new algorithm based on bin packing techniques. In this chapter we also focus on the supervised clustering of tenants and the migration of tenant data over time and we introduce two novel bin packing approximation algorithms which are optimized for both the static and dynamic allocation of tenant data, together with an extended evaluation.

3.2.2 Data Assurance Policies

Data assurance policies can be used to meet legal and business data archival requirements for both persistent data and records management. Compliance with regulatory policies on data however remains a key hurdle to cloud computing [7, 8]. According to the authors, cloud providers often fail to meet the required policies, and often users have no way to specify their requirements. The authors show how storage services such as AppScale and Cassandra could be extended to support the data handling requirements. Jun Li et al. [12, 13] propose a policy management service that offers scalable management of data assurance policies attached to data objects stored in a cloud environment. With GEODAC [14], the authors provide a policy framework that enables the expression of both the service providers' capabilities and customers' requirements, and enforcement of the agreed-upon policies in service providers' environments.

It is not our goal to develop a new policy framework, but as data assurance policies add additional constraints on the allocation of tenant data, the cloud based TDS system presented in this chapter should be flexible enough to support such and other tenant specific data policies.

3.2.3 Software-Defined and Scalable Storage

Ceph [15] is a distributed object store and file system designed to provide excellent performance, reliability and scalability. The system stores client data as objects within storage pools, and uses the CRUSH [16] algorithm to allocate the objects over placement groups. This algorithm also uses a hierarchical structure, called the hierarchical cluster map, but this hierarchical structure is used to manage the storage devices, which are the leaves of the tree. The main difference with our solution is that CRUSH will distribute data using a pseudo-random function. Each object is mapped to a list of devices on which to store object replica, approximating a uniform probability distribution. We however want to distribute data based on tenant-specific parameters, where every tenant is mapped to a single storage device, taking into account custom tenant policy restrictions. This results in a high isolation of tenant data, which is an important requirement for multi-tenancy. When a single tenant is too big to fit a single storage device, the tenant is split into smaller subtenants following the approach of Section 3.4.2.3.

In [17] the performance of a data cluster based on the Ceph platform with geographically separated nodes is evaluated. The authors focus on achieving high availability, by allocating copies of the data over multiple distributed storage nodes, and measuring the required bandwidth. We mainly focus on the scalability of tenant data and less on high-availability, but the solution proposed in their paper can be used to extend our system to support distributed high-availability.

In the context of SDS, FlexStore [18] is a Software-Defined Energy Adaptive Storage Framework that aims to reduce energy usage of the storage resources while guaranteeing the required performance. The framework consists of a policy engine to enforce the required policies by adjusting the allocation of storage resources. In [19] the authors present an implementation of a Software-Defined Storage Service in a heterogeneous environment. For their Proof of Concept, they integrate HDFC, Ceph and Swift together with a management cloud service running on OpenStack.

Most scalable storage systems follow the data fission approach, as we will describe in detail in Section 3.3.3. For the tenant-based provisioning of data storage, we however prefer the data fusion approach, as we want to cluster related tenants together based on their location and other tenant-specific attributes. As a result, in this chapter we will present a new approach for managing tenant data, based on bin packing. A highly scalable storage solution such as Ceph could however be used for the provisioning of the independent storage pools within a single location.

Although related, SDS is not the same as a Content Delivery Network (CDN). The goal of a CDN is to deliver content to end-users with high availability and high performance. CDN operators either use distributed dedicated servers to replicate and deliver the content, or they make use of a hybrid model, often using Peer-to-Peer (P2P) technology. SDS on the other hand is based on similar concepts

as Software-Defined Networking (SDN), and its main goal is to provide a virtual storage system towards the applications, by abstracting the logical storage services and capabilities from the underlying physical storage systems. An SDS system handles the provisioning and allocation of the application data, taking into account custom data policies, for example based on legal requirements or the selected SLA. The TDS system presented in this chapter is an SDS system, but optimized for the allocation of tenant data in multi-tenant cloud systems.

3.2.4 Bin Packing

The problem of allocating cloud resources can be seen as a bin packing problem. Quite often bin packing approximation algorithms are being used for cloud resource provisioning [20–22]. In [23] for example, the authors present an approach for the adaptive provisioning of cloud resources within a datacenter using bin packing approximation algorithms. In [24], a family of sharing-aware online algorithms for the provisioning of cloud resources are proposed, which are based on existing bin packing approximation algorithms but with a focus on the sharing of volatile resources such as memory.

The TDS system proposed in this chapter also invokes an approximation algorithm for the bin packing problem, but we focus on the management of storage resources, which introduces some additional objectives such as minimizing the number of migrations. Our system is not limited to a single datacenter, but instead we make a clear distinction between inexpensive migrations within a single datacenter, and costly migrations between geographically distributed datacenters. Furthermore, we present a hierarchical system, in which tenants are first organized in a hierarchical structure, and as a result our system uses a hierarchical variant of existing bin packing algorithms, which we will refer to as hierarchical bin packing.

In [25], the authors also introduce a hierarchical bin packing approach for the management of virtual machines within a cloud environment. They however express the current demand as means with standard deviations, instead of given scalar variables, which is a good approach for the management of resources such as CPU load and network bandwidth. Because of this, the authors formulate the problem as a multi-capacity stochastic bin packing problem, and propose a heuristic method to solve this NP-hard problem. As we mainly focus on storage resources, we can work with scalar variables (as the exact usage is known at any moment), but our system can similarly be extended to support other types of resources if required.

In [26] a hierarchical variant of the bin packing problem is studied in which the items to be packed are structured as the leaves of a tree. Just as with classical bin packing, this is an NP-hard problem, but the authors provide approximation algorithms for several cases. Furthermore, in [27] the authors provide some comments on the hierarchical bin packing problem by investigating an existing algorithm, but

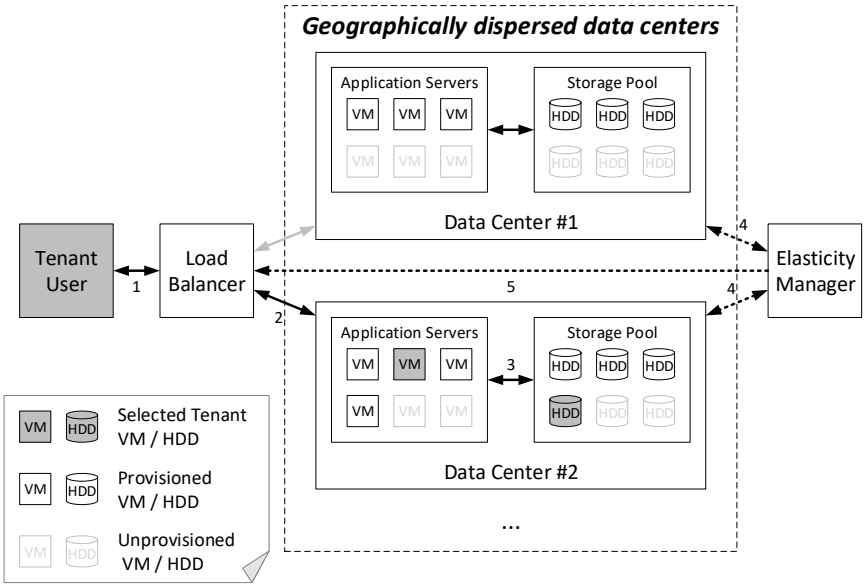


Figure 3.3: General overview of the Tenant-Defined Storage system.

they also only handle cases in which the items to be packed into bins are the leaves of a tree. In our approach, after the hierarchical clustering of tenants, some internal nodes can also have a certain size. As a result, in this chapter we will present two novel approximation algorithms for the hierarchical bin packing problem, in which the items to be packed are not restricted to the leaf nodes, and compare them to existing algorithms in terms of efficiency.

3.3 Design of a Tenant-Defined Storage System

3.3.1 General Overview

The TDS system aims to automatically allocate and reallocate the storage resources required by the different tenants. Figure 3.3 presents a general overview of the system. When a tenant user accesses the multi-tenant application, he first connects to a load balancer (1 in the figure) in order to select one of the available server instances located in a nearby data center (2). To retrieve the application data, the selected application server needs to connect to the corresponding storage pool where the tenant's data is stored (3). This storage pool should be close to the application server, preferable within the same data center. In an elastic cloud environment, both the application servers and storage pools can be provisioned on the fly. The management and provisioning of these resources is the main task of the

elasticity manager (4). To achieve high scalability, this component monitors and evaluates the current load on the provisioned application server instances. As the load increases, additional instances will be provisioned to avoid overload. Similarly, the component also monitors the usage of the provisioned storage pools. If the usage of a single storage pool reaches a certain threshold, an additional storage pool is provisioned and some of the existing tenant data will be reallocated before the storage pool runs out of space. On the other hand, if the load on the application servers or the usage of the storage pools decreases significantly, one or more application servers and/or storage pools should be de-provisioned, requiring the reallocation of some of the tenants, in order to minimize the operating costs. Whenever tenants are reallocated, the elasticity manager also notifies the load balancer (5) to guarantee the correct routing of incoming requests.

In the remainder of this chapter, we will focus on the design of the elasticity manager. Every tenant puts a certain load on the system, requiring a specific amount of resources. This load can be expressed as the required number of CPU cycles, the required amount of memory, storage, or a combination. We however focus on storage resources, as these introduce additional constraints for the (re)allocation such as minimizing the amount of migrations or the total migration size. In this context, the primary task of the elasticity manager is to, given a certain set of tenants with tenant-specific characteristics, find a feasible allocation of tenant data over the set of available storage resources (static allocation). Furthermore, as the amount of tenant data grows over time, reallocation might be required (dynamic allocation), and especially for geographically distributed storage resources, the total amount of data to reallocate (reallocation size) should be minimized as this is both expensive and time consuming.

3.3.2 Hierarchical Clustering of Tenants

In our solution, tenants are hierarchically organized using a tree structure, which we refer to as the tenant tree. There are several reasons to do so. First of all, multi-tenant applications are often used by a number of organizations, the tenants. Large organizations however tend to consist of multiple independent divisions, introducing the need for subtenants or even sub-subtenants and the tenant tree inherently supports this hierarchical structure. Secondly, when the tenants using the application are geographically distributed, it might be good practice to cluster them based on their location, and resources can be allocated from a resource pool close to the tenant. Tenants could also be clustered based on other characteristics, e.g. the selected SLA or other regulatory policies concerning the storage of sensitive data, and these characteristics could define the required type of (physical) hardware. In general, tenants can be clustered based on multiple characteristics, depending on the requirements of the tenant, the possibilities of the application and the infras-

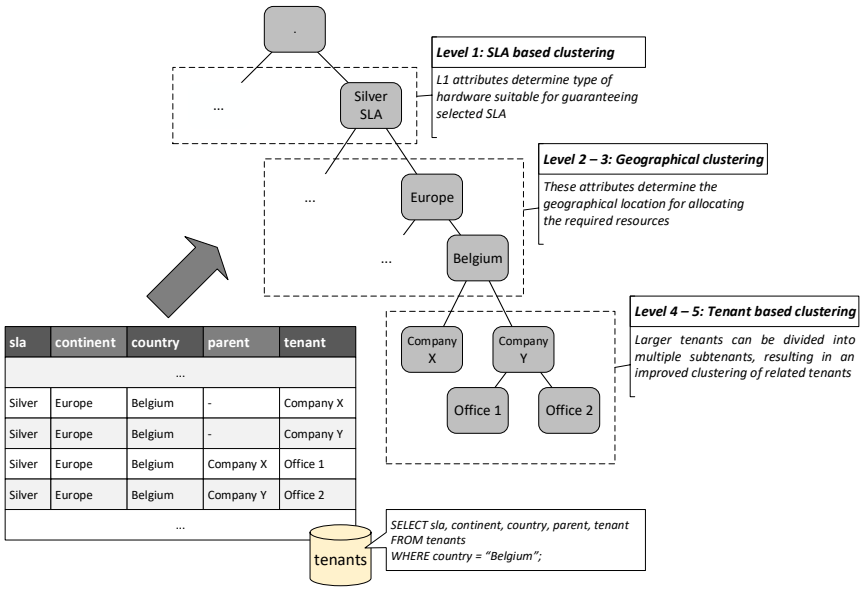


Figure 3.4: Example partial tenant set with their selected characteristics and a part of the corresponding tenant tree. In the tenant tree, tenants are clustered based on multiple characteristics, with more significant characteristics such as the selected SLA at a higher level in the tree structure.

tructure. The goal of the TDS system is to cluster related tenants together while minimizing migrations over time.

In the resulting tenant tree, the most significant characteristics appear at the highest levels of the tree structure, as higher levels have a higher impact on the clustering of tenants. Figure 3.4 illustrates this concept. In the left bottom corner of this figure some example tenants using the application are listed, together with their relevant characteristics. The remainder of the figure illustrates a part of the corresponding tenant tree. In this example, tenants are first clustered based on their SLA, allowing the infrastructure provider to assign tenants to different types of storage resource pools, for example by using high-end hardware with improved replication and higher availability for more stringent SLAs. Next, the tenants are clustered based on their geographical location, allowing the provider to allocate tenants to a resource pool close to their location to minimize network delays. Finally, tenants are clustered based on their internal structure. The conversion from the tenants table to the tenant tree is straightforward and can be achieved by using simple data queries. Once a feasible allocation scheme has been found, the mapping from clusters of tenants towards a geographical location and required type of storage pool is calculated by determining the mayors for every cluster, as we will

explain in detail in Section 3.4.3.3.

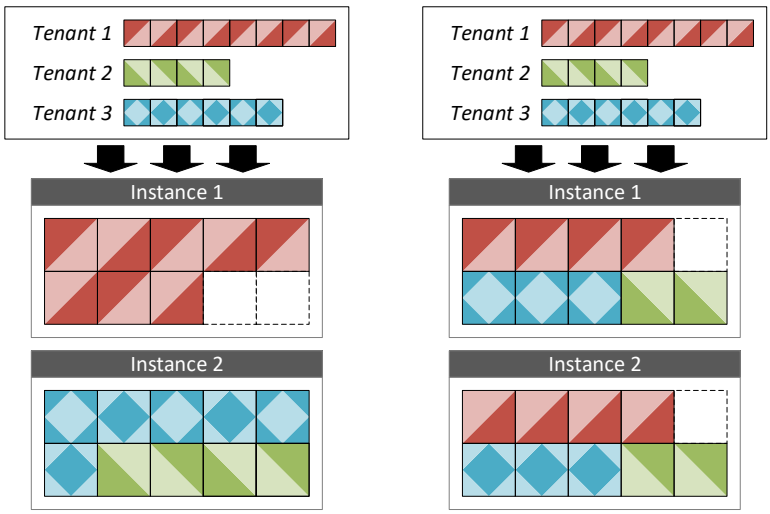
3.3.3 Data Fusion vs Data Fission

When allocating storage resources, for example when developing scalable database systems, there are 2 commonly used approaches, often referred to as data fusion and data fission [28]. The difference between both is how the data is distributed over the available instances. Data fusion refers to the approach where multiple small data granules are combined to provide stringent transactional guarantees on larger data granules, and is often used to achieve better scalability. Data fission or partitioning on the other hand is another approach for achieving high scalability, by splitting large data blocks into relatively independent shards or partitions and providing transactional guarantees only on these shards. Figure 3.5 illustrates both approaches.

Most scalable database systems and data storage solutions such as Ceph [15] follow the data fission approach. The TDS system however aims to cluster related tenants together on a single instance, which is related to the data fusion approach. There are several reasons to do so. First of all, one of the most important aspects of multi-tenancy is the isolation of tenants, for both performance and security. Although multiple tenants share a common set of resources, the instance should always behave as a dedicated instance towards the tenant. By using the data fusion approach, when a large tenant puts a heavy load on the application, the impact will be limited to a single instance, and only few tenants sharing the same instance can be affected. This is opposed to the data fission approach in which all tenants using the application would potentially be impacted. In case of a security leak on one of the instances, the impact is similarly contained to a single instance. Furthermore, when tenants and storage resources are geographically distributed, it would be bad practice to divide data belonging to a single tenant over multiple physical locations, as this will result in high response times when data needs to be aggregated. On the other hand, the data fission approach could still be used within a single resource pool, for example by using Ceph, but on the meta level the data fusion approach is best suited.

3.3.4 Elasticity Manager

Once the tenant tree is constructed, the main task of the elasticity manager is to determine a feasible allocation of the tenants over the available resource pools. After the initial allocation, tenants can however grow in size, requiring reallocation of some of the tenants, either within a single resource pool or between multiple geographically distributed locations. The elasticity manager needs to support this behavior, introducing the need for a dynamic system and thus dynamic algorithms for the (re)allocation of tenants.



(a) Data fusion (b) Data fission

Figure 3.5: Data fusion vs data fission: 2 approaches for achieving high scalability of data storage. Data fission focuses on a uniform distribution of data over the available instances, whereas data fusion provides a strong isolation of data.

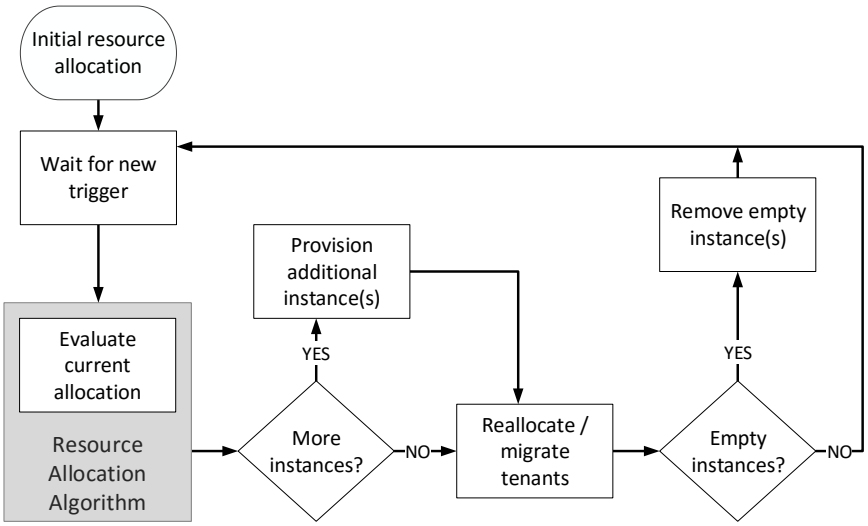


Figure 3.6: The functionality of the elasticity manager, responsible for the dynamic allocation of tenant data over the available resource pools, and the role of the (dynamic) resource allocation algorithm(s).

Figure 3.6 illustrates the functionality of the elasticity manager. As can be seen in this figure, the component invokes a dynamic resource allocation algorithm for the allocation of the different tenants, based on the information of the input tenant tree. Re-evaluation can be triggered by certain events, for example when a new tenant is added to or removed from an existing application, or when some of the provisioned instances are overloaded or underloaded because of an increase or decrease in size of some of the existing tenants.

3.3.5 Resource Allocation Algorithm

Although linear programming could be used to find an optimal solution for the resource allocation problem, its applicability in real systems is often limited due to the complexity. Every additional constraint increases the complexity of the mathematical model, and possibilities for customization are often limited due to the need for linear expressions. Furthermore, large datasets containing a high number of items often result in high (and sometimes unacceptable) execution times. Another possible solution would be to design a permutation based algorithm for finding a (pseudo-)optimal solution, which iterates over all possible allocations. These type of algorithms offer higher flexibility, but at the cost of even higher execution times [9, 10].

The management of storage resources (or cloud resources in general) can also be seen as a bin packing problem, in which the tenants are the items to be packed inside multiple resource pools, the bins, and the objective is to minimize the number of bins in order to minimize operational costs. As the bin packing problem is an NP-hard problem, we prefer to use an approximation algorithm as these often find an acceptable solution within limited time.

As mentioned before, related tenants should be clustered together based on the tenant tree. When using bin packing, this introduces the need for an approximation algorithm for the hierarchical bin packing problem. In the next section, we introduce two novel approximation algorithms for the static allocation of tenant data, each having its own advantages and applicability for certain scenarios, and a dynamic extension for the dynamic reallocation.

3.4 Dynamic Resource Allocation

The goal of the resource allocation algorithms is to find a feasible distribution of the different tenants over a set of resource pools. When using bin packing techniques, the tenants (each having their own size) correspond to the items to be packed into the bins, being the resource pools.

3.4.1 Problem Formulation and the FFD Strategy

The bin packing problem can be formulated as follows: Given a list of n tenants, each having their own tenant size a_i ($1 \leq i \leq n$), and a single resource pool S of size V. The goal is to find a finite number of resource pools B and a B-partition $S_1 \cup ... \cup S_B$ of the set $\{1, ..., n\}$ so that $\sum_{i \in S_k} a_i \leq V$ for all $k = 1, .., B$. A possible solution for this problem is optimal if it has a minimal value for B.

The bin packing problem can also be formulated as an Integer Linear Programming (ILP) model [29], by introducing two binary decision variables:

$$y_i = \begin{cases} 0 & \text{if bin i is used} \\ 1 & \text{otherwise} \end{cases} \qquad (i = 1..u)$$

$$x_{ij} = \begin{cases} 0 & \text{if item j} \in \text{bin i} \\ 1 & \text{otherwise} \end{cases} \qquad (i = 1..u, j = 1..n)$$

In the above formulas, u corresponds to the upper bound on the optimal solution value (e.g. the value found by an approximation algorithm). The objective of the ILP is to find:

$$B = \min \sum_{i=1}^{u} y_i$$

subject to:

$$\sum_{j=1}^{n} a_j x_{ij} \leq V y_i \quad (i = 1..u),$$

$$\sum_{i=1}^{u} x_{ij} = 1 \qquad (j = 1..n),$$

$$y_i \in 0, 1 \qquad (i = 1..u),$$
$$x_{ij} \in 0, 1 \qquad (i = 1..u, j = 1..n).$$

The bin packing problem is an NP-hard problem, but there exist some simple yet very efficient approximation algorithms, such as the First-Fit Decreasing (FFD) strategy. This algorithm first sorts the set of items, based on their size, in decreasing order and then it adds each item to the first available bin that can accommodate the item. If no bin is found, it adds a new bin to the set, and adds the item to this new bin. It has been proven [30] that the algorithm will always find a solution in which there are no more than $11/9 \times OPT + 1$ bins, in which OPT is the number of bins in the optimal solution.

The formal model is sufficient for the static allocation of unrelated tenants, but when applied to a hierarchical dynamic system, two additional objectives can be introduced:

1. As tenants are hierarchically structured, related tenants (e.g. siblings of the same parent node) should be allocated together when possible.

2. For a dynamic system, migrations over time should be minimized, as we mainly focus on the allocation of storage resources.

The FFD strategy is not suitable for the implementation of the elasticity manager of the TDS system, as it is not adjusted to take into account our additional objectives regarding the dynamic behavior of the system. In the evaluation section however, we compare the hierarchical bin packing algorithms to both the FFD strategy and an implementation of the ILP model, as this offers some interesting insights about the overhead of the different algorithms in terms of number of bins and the average bin usage.

3.4.2 Hierarchical Bin Packing

When the items to be packed are hierarchically organized, the bin packing problem is often referred to as hierarchical bin packing [25–27]. For our system, we designed two novel approximation algorithms for this problem, based on the FFD strategy and existing tree methods. Before introducing the algorithms, we first define the structure of a tree node. In the tenant tree, every node (both internal nodes and leafs) is represented by the structure shown in Figure 3.7.

In the representation of a single node, the $nodeSize$ property returns the size of the current node (without child nodes), whereas the $treeSize()$ method returns the calculated total size of the subtree with the current node as root. Nodes can either be virtual ($nodeSize = 0$), used for structuring the tenants, or they can represent a single tenant ($nodeSize \geq 0$). In the example of Figure 3.4, *Europe* is a virtual node, whereas *Office 2* and *Company X* represent tenants using the application.

Tenants however are not restricted to leaf nodes, as some internal nodes can also represent a tenant (with a certain size). This could for example happen when a company is divided into multiple departments, but there is also a central administration managing the whole company, represented in the tenant tree as a parent node with different leaf nodes for the different departments. Another reason to have internal nodes with a certain size is to support the splitting of nodes when a single node is too big to fit a single bin, as we will explain in Section 3.4.2.3.

3.4.2.1 The Hierarchical First-Fit Decreasing Strategy

The pseudo-code of the first algorithm, which we will refer to as the Hierarchical First-Fit Decreasing (HFFD) strategy, is presented in Algorithm 1. This algorithm takes a pointer to the root node as input parameter, instead of a list of tenants, and a pointer to an (initially empty) list of bins, required for the recursion. Note that

```
struct Node {
  String name;          // Tenant identifier
  Node[] children;      // Array of child nodes
  double nodeSize;      // Size of current node
                        // (without children)

  double treeSize();    // Size of (sub)tree
                        // with current node as root
  Node[] separate();    // Returns a list containing
                        // the current node without
                        // children, and all child
                        // nodes as subtrees
  Node* parent(int l); // Returns the (grand-)parent
                        // node at the selected level
                        // of the tenant tree, e.g.
                        // when l=0 the root node
                        // will be returned
};
```

Figure 3.7: Structure of a single node within the tenant tree.

the $bins$ parameter is a reference, meaning that there is only one list of bins, and the reference is shared between all recursive calls.

If the whole (sub)tree can be allocated to a single bin, the algorithm will do so, in a similar way as the FFD strategy. Note however that in this case the (sub)tree will not be split and allocated over multiple bins. If the (sub)tree does not fit a single bin, all child subtrees are sorted in decreasing order based on their $treeSize()$, together with the current root node based on its $nodeSize$. For this, the $separate()$ method of the node structure is very useful, as it returns a list of all subtrees together with the root as a leaf node, which can then easily be sorted based on the $treeSize()$. The algorithm is then invoked recursively for every Node inside this sorted list, together with the pointer to the list of existing bins.

3.4.2.2 The Hierarchical Greedy Decreasing Strategy

Although the HFFD strategy is already feasible for many scenarios, we also propose a second algorithm, which we refer to as the Hierarchical Greedy Decreasing (HGD) strategy. The pseudo-code of this algorithm is presented in Algorithm 2. The algorithm is very similar to the HFFD strategy, but the main difference lies within the allocation of the (sub)tree to a bin. Instead of looking for a suitable existing bin, the (sub)tree is always added to a new bin, which is then added to the list of bins.

algorithm 1: The Hierarchical First-Fit Decreasing (HFFD) Strategy

Input: V, $Node*\ root$, $S[]*\ bins$
Output: $bins$
if $root.treeSize() \leq V$ **then**
for bin in $bins$ **do**
if $root.treeSize()$ fits in bin **then**
Add whole (sub)tree to existing bin
Break the loop
end if
end for
if $root.treeSize()$ did not fit in any available bin **then**
Add whole (sub)tree to a new bin, and add bin to $bins$
end if
else
for $node$ in (sort $root.separate()$ in decreasing order) **do**
Invoke algorithm with $node$ as root and $bins$ as input parameters (recursive call)
end for
end if

algorithm 2: The Hierarchical Greedy Decreasing (HGD) Strategy

Input: V, $Node*\ root$, $S[]*\ bins$
Output: $bins$
if $root.treeSize() \leq V$ **then**
Add whole (sub)tree to a new bin, and add bin to $bins$
else
for $node$ in (sort $root.separate()$ in decreasing order) **do**
Invoke algorithm with $node$ as root and $bins$ as input parameters (recursive call)
end for
end if

The HGD strategy will generally divide the tenants over more bins than when using the FFD or HFFD strategy, hence the term greedy. This however results in fewer reallocations and smaller migration sizes over time, together with an increased level of tenant isolation, making this algorithm very suitable for supporting higher (often referred to as "gold") SLAs.

3.4.2.3 Support for Big Nodes

The presented algorithms work fine if all nodes have a $nodeSize \leq V$, meaning that all nodes fit into a single bin. In some cases however, there might exist some

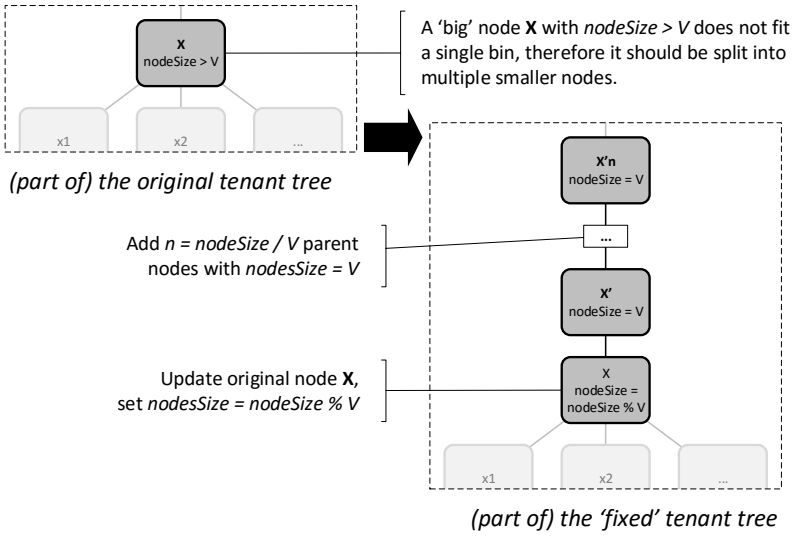


Figure 3.8: Fixing the tenant tree before invoking the algorithm by splitting "big" nodes.

bigger nodes which don't fit a single bin. As a result, the tenant tree is first modified before invoking the algorithm. During this pre-processing step, big nodes are split into multiple small nodes as illustrated in Figure 3.8.

3.4.3 Migration Strategy

The hierarchical bin packing algorithms can be used for the static allocation of storage resources in a cloud system, as they minimize the required amount of resources while taking into account the hierarchical structure of the tenants. In a dynamic scenario however, tenant sizes grow and shrink over time, and at some point the system might need to reallocate some of the tenants in order to avoid over- or under-usage of the resource pools.

In the remainder of this section, we will first introduce two additional parameters useful for implementing this dynamic behavior. Next, we describe how we adjusted the static hierarchical algorithms to further reduce the migration sizes. We conclude this section by describing how migrations are determined between consecutive (re)allocations.

3.4.3.1 Allocation Factor (AF) and Reallocation Delta (RD)

To implement the dynamic behavior, two additional parameters are introduced:

- The Allocation Factor (AF), with $0 < AF < 1$.
- The Reallocation Delta (RD), with $0 < RD < 0.5$.

The first parameter AF indicates the maximum allowed usage of a single storage resource pool during (re)allocation. In this context, the usage of a bin is defined as the amount of resources currently used over the total amount that the bin can accommodate. The AF can therefore be seen as a margin, and without this parameter the allocation algorithms could fill a single bin up to 100%, resulting in an overflow as soon as one of the tenants increases in size. If a single resource pool has a maximum size of MAX, the bin packing algorithms introduced before will be invoked with $V = AF \times MAX$ for the size of a single bin, meaning that there is still some place left in the bins after the allocation.

The second parameter, RD, is used for reallocations. As mentioned before, reallocation can either be triggered by adding or removing a tenant (with a certain size), or when a single bin is either over- or underloaded. It is a delta value, meaning that reallocation will be automatically triggered by the TDS system as soon as one of the resource pools reaches a usage that is either $\geq (AF + RD) \times MAX$ (overloaded) or $\leq (AF - RD) \times MAX$ (underloaded). A lower value for this parameter will trigger reallocation faster, and the value should be low enough to avoid overflows as data should be migrated before the bin reaches a usage of 100%.

To summarize, while the MAX value refers to the maximum number of data blocks that a single bin can accommodate, the AF and RD parameters are used during the (re)allocation of tenant data and the system continuously monitors all bins to detect any overloaded or underloaded instances. For example, if $AF = 0.7$ and $RD = 0.2$ then during every reallocation the system will try to fill every single bin up to 70%, while reallocation will be triggered when a single bin reaches a usage that is either $\geq 90\%$ or $\leq 50\%$. The only exception happens during the first iterations, when the total amount of data blocks is limited ($\leq 0.5 \times MAX$), and therefore underflows are temporary allowed as otherwise it would be impossible to find a feasible allocation scheme.

3.4.3.2 Dynamic Extension for the Bin Packing Algorithms

The hierarchical bin packing algorithms, together with the two additional parameters could already be used for implementing the elasticity manager. Whenever a bin triggers reallocation, the elasticity manager can re-invoke the algorithm using the whole tenant tree, and calculate the migrations. The algorithms however are not yet optimized for minimizing the migration sizes, as by following this approach tenants who are allocated to a bin that is not yet over- or underloaded might also get reallocated as the algorithm tries to minimize the total number of bins.

A possible and effective solution for this problem is to only invoke the algorithm for tenants assigned to one of the over- or underloaded bins. This can be easily achieved using the following steps:

1. Generate a list containing all tenants that are allocated to either an over- or underloaded bin.
2. Filter the set of bins by removing all under- and overloaded bins.
3. Invoke the bin packing algorithm, with both the set of tenants to reallocate and the filtered set of bins as input parameters.

By invoking the algorithm with the filtered set of bins, tenants that need to be reallocated can still be allocated to an existing bin (except for the greedy strategy, as this strategy will always assign the tenants to a new bin), but existing tenants that don't require reallocation will remain in the previously assigned bin.

This dynamic extension can be applied to all bin packing algorithms, and we will refer to the dynamic variant of the bin packing algorithms as the dFFD, dHFFD and dHGD strategies. As we will illustrate in the next section, using the dynamic variant will generally result in a significant decrease in migration sizes.

3.4.3.3 Determining the Migrations

Whenever reallocation is triggered, the TDS system will re-invoke the selected (dynamic) allocation algorithm in order to determine a new feasible distribution of tenant data over the available resource pools. The new distribution will be likely different from the old one, and some tenants might need to be migrated between resource pools. The TDS should be able to migrate these tenants as efficiently as possible, minimizing the total amount of data that should be migrated.

When calculating the migrations, we make a distinction between expensive migrations, for example between geographical distributed locations, and inexpensive migrations within a single location. Within a single location, migrations are fast as data can be migrated within the internal backbone network, typically having a very high bandwidth, and most infrastructure providers will charge no costs for the consumed bandwidth. Between two locations however, migrations will consume more time as there is less available bandwidth, and the provider might charge for the consumed bandwidth.

In order to calculate the shortest migration path, we added the concept of mayors to the bins. Within the tenant tree, every tenant is connected to the root node through a set of parent nodes. We can retrieve this set of parent nodes by using the $parent(int\ l)$ method of the Node structure defined before. A single bin contains one or more nodes of the tenant tree. Within a single bin, we define the mayor for a given level of the tenant tree as the most important node (biggest total size), by using Algorithm 3.

algorithm 3: Calculating the mayor on the selected level of the tenant tree for a given bin

Input: $S * bin, int\ level$
Output: $Node * mayor$
Initialize $Map < Node*, int > parents$
for $node$ in bin **do**
$parents[node.parent[level]] += node.nodeSize$
end for
Return key from $parents$ with the biggest $value$

If we invoke this algorithm for all levels of the tenant tree, we have a list of mayors per level for the bin. This list defines the characteristics of the bin, if we apply this to the example of Figure 3.4, the level 1 mayor defines the type of hardware required to support the SLA and the mayors from level 2 and 3 define the geographical location. To calculate the migrations we compare the mayors for every tenant in every bin for the 'old' allocation scheme (before re-invoking the allocation algorithm) and the 'new' allocation scheme. If a single tenant is assigned to another level 1 mayor after reallocation this signifies that the tenant is migrated to a different resource pool, for example running on different hardware. If we repeat this for all levels, we can construct a list of all tenants that are migrated on level 1, level 2, and so on.

Calculating the mayors per level allows us to make a distinction between costly and inexpensive migrations and the goal of the TDS system is to minimize the migration sizes at the top levels. For example, if we take the tenant tree of Figure 3.4, and the physical locations are defined by the continent (level 2), we know that level 1 and level 2 migrations are expensive, whilst the other migrations are inexpensive as these happen within a single location.

Figure 3.9 illustrates this concept applied to a small example scenario. In this example, all tenants are initially allocated to a single bin. As the two tenants located in Belgium together account for the biggest part of the bin, the mayors for this bin are *Europe* (L1) and *Belgium* (L2). When some tenants increase in size, the bin reaches a usage higher than 0.9 ($= AF + RD$) and reallocation is triggered. After re-invoking the data allocation algorithm, the tenants are now distributed over 2 bins instead of one, and for the newly provisioned bin the mayors are *North America* and the *United States*. If we iterate over all tenants, only tenant *Z* has different mayors after reallocation. As the L1 mayors are different, a L1 migration is required and the tenant will be migrated from a data center in *Europe* to a data center in *North-America*.

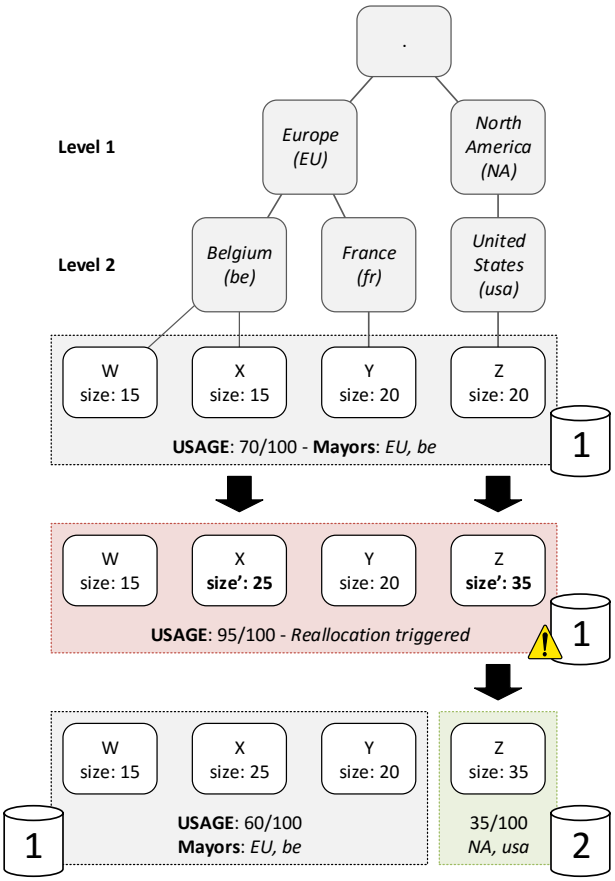


Figure 3.9: Calculating the migrations after some tenants increase in size: example scenario with $MAX = 100$, $AF = 0.6$ and $RD = 0.3$.

3.5 Evaluation Results

3.5.1 Evaluation Metrics

The primary goal of the system is to determine a feasible allocation of tenant data over the available resource pools. In order to minimize the operational costs, a feasible allocation should aim to minimize the number of instances (bins). Furthermore, as we focus on a dynamic system, the number of migrations over time should also be minimized. These two metrics however don't always go together, as more bins will generally result in fewer migrations and vice versa. Finally, we also look at the average distance within the bins, to find out if the system is able to cluster related tenants together (isolation of tenants). In summary, we focused on the following evaluation metrics, which correspond to the objectives described in Section 3.4.1:

- **Average Bin Usage** (static allocation)
  As more bins will mostly result in higher operational costs, the goal of the system is to minimize the number of bins required. For a single bin, we define the bin usage as the sum of the items over the maximum bin size (MAX). If we do this for all bins, we can calculate the average bin usage:

$$\overline{binusage} = \frac{\sum_{k=1}^{B} \frac{\sum_{i \in S_k} a_i}{MAX}}{B} = \frac{\sum_{i=1}^{n} a_i}{B \times MAX}$$

  In the above formula, a_i ($1 \leq i \leq n$) is the size of tenant i, and S_k ($1 \leq k \leq B$) corresponds to a single bin. A higher value for this metric will result in less bins, and an efficient algorithm should achieve an average bin usage close to the selected AF.

- **Migration Size** (dynamic allocation)
  Whenever data is reallocated, we calculate the total migration size, which corresponds to the sum of the sizes for all tenants that are migrated. To normalize the results, we also calculate the relative migration sizes, defined as the migration size during each iteration over the total amount of data in the last iteration. For the dynamic behavior of our system, this is the most significant metric.

- **Average Bin Distance** (hierarchical clustering)
  The system is designed to cluster related tenants together (e.g. siblings of the same parent node), to achieve both data and performance isolation. In the tenant tree, we define the distance between two tenants as the number of edges between the two nodes following the shortest path. The bin distance for a single bin is then defined as the average distance between all tenants

allocated to this bin. If we calculate this for all bins, we can calculate the average bin distance, and a lower value indicates a better clustering and/or isolation of tenants, as a value of 0 for this metric implies that every tenant is allocated to a dedicated bin.

- **Ratio of Shared Bins** (tenant isolation)
  The last evaluation metric is the ratio of shared bins, which corresponds to the number of bins accommodating multiple tenants over the total number of bins. This metric provides some insights about the degree of tenant isolation, as a lower value for this metric indicates a higher number of dedicated storage instances.

3.5.2 Evaluation Setup

We evaluated the different algorithms using the simulator presented in [31]. The simulator is built on top of the elasticity manager of the presented SDS system, but works with a simulated cloud environment. The simulator calculates both the allocations, migrations and average bin distance for every iteration of a given dataset.

3.5.2.1 Case Studies

For our experiments, we used two case studies based on real-life datasets. The first case study is the implementation of a population register, in which for every inhabitant a single data block is stored. The dataset for this case study is based on the yearly population of every town in Flanders, over a period of 7 years, which can be found on the official website of Flanders [32]. In this scenario, every town represents a tenant (with the number of inhabitants corresponding to the number of data blocks that need to be allocated), and the tenants are hierarchically organized based on their geographical location, with the capital city, region and province on the path from the leaf node to the root of the tenant tree. Including the internal nodes, the whole tenant tree for this scenario consists of 946 nodes.

Our second scenario is similar to the first, but the dataset is based on the number of fixed broadband subscribers per country worldwide over the last 16 years, which is available from the World Bank Open Data [33]. For this dataset, every country represents a tenant with the number of subscribers as the required amount of data blocks, and the tenant tree is also based on the geographical location and consists of 222 nodes.

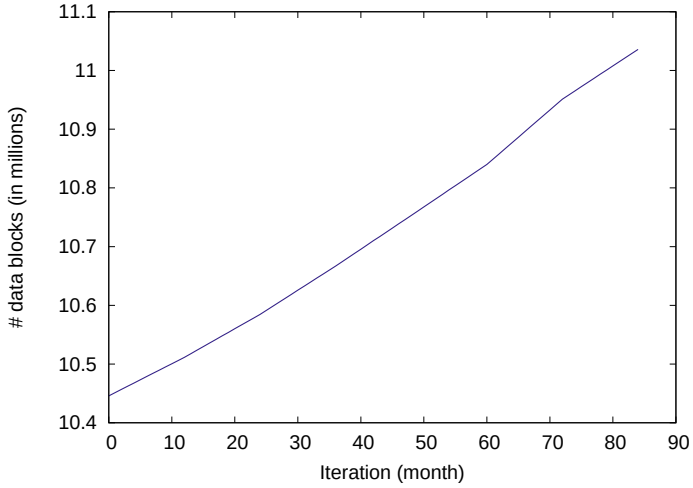
While both scenarios look very similar, there is an important difference. The first scenario is a good example of a slow-growing dataset, as the total amount of inhabitants increases slowly over time, whereas the second scenario is a good example of a fast-growing dataset with exponential growth. These two datasets cover a broad spectrum of possible scenarios. The slow-growing dataset resembles

Table 3.1: Overview of both datasets

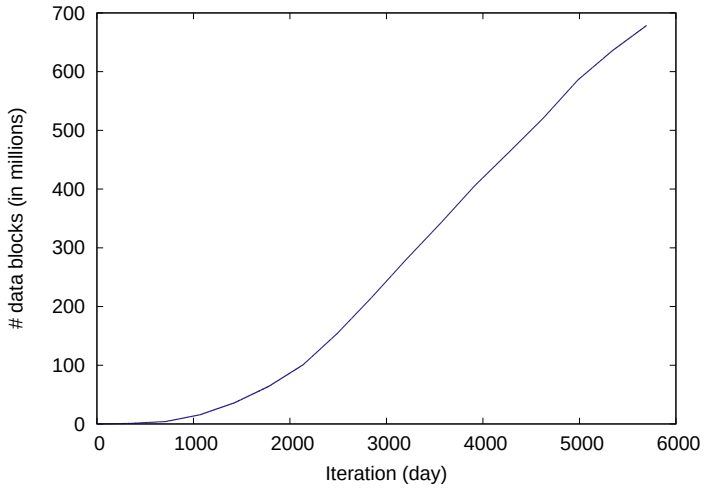
	Slow-Growing Dataset	**Fast-Growing Dataset**
Internal nodes		
$L0$	1 (root)	1 (root)
$L1$	9 (provinces)	7 (continents)
$L2$	43 (regions)	
$L3$	50 (subregions)	
$L4$	254 (main cities)	
Leaf nodes		
	589 (towns, L5)	214 (countries, L2)
Total nodes	946	222
Iterations		
Timespan	$\pm$ 7 years	$\pm$ 16 years
Iterations	monthly	daily
Total iterations	85	5697
Instance physical location defined by		
	region (L2)	continent (L1)
Evaluated values for configurable parameters		
MAX	5×10^6, 10^6, 5×10^5, 10^5 5×10^4, 10^4, 5×10^3	5×10^8, 10^8, 5×10^7, 10^7, 5×10^6, 10^6, 5×10^5
$AF \pm RD$	0.5 ± 0.1, 0.2, 0.3, 0.4 0.6 ± 0.1, 0.2, 0.3 0.7 ± 0.1, 0.2	idem as slow-growing dataset

the expected growth of an established application with a fixed number of tenants with an initial size, and the tenant sizes grow slowly over time. The fast-growing dataset on the other hand resembles a novel popular application, where all tenants have an initial size of 0, and in the initial years not only the individual tenant sizes grow, but also the total number of tenants.

Table 3.1 provides an overview of both datasets but before running the pre-processing step described in Section 3.4.2.3, as this step might introduce additional levels and internal nodes to guarantee that every node can be allocated to a single bin. Figure 3.10 illustrates the total size of both datasets over time. In the remainder of this section, we will refer to both datasets as the slow-growing and fast-growing dataset respectively.



(a) Slow-growing dataset based on the population per town in Flanders with monthly iterations.



(b) Fast-growing dataset based on the number of fixed broadband subscribers per country worldwide with daily iterations.

Figure 3.10: Total number of data blocks for both datasets over time.

Table 3.2: Valid combinations of AF and RD for the fast-growing dataset using all values for MAX

RD \ AF	0.5	0.6	0.7	0.8
0.1	✓	✓	✓	✗
0.2	✓	✓	✗	
0.3	✓	✗		
0.4	✗			

3.5.2.2 Configurable Parameters

For our evaluation, we not only selected different values for the allocation factor (AF) and reallocation delta (RD) described in Section 3.4.3.1, but also for the capacity of a single bin (MAX). For the evaluated case studies, the value MAX refers to the maximum number of data blocks a single storage volume can accommodate, and by selecting different values for the configurable parameters, we cover a very broad spectrum of possible scenarios. An overview of the evaluated values for all parameters is also provided in Table 3.1.

3.5.3 Influence of Configurable Parameters on the Risk of Overflows

While the value of MAX will be defined by the scenario (and type of hardware used), the values for AF and RD can be adjusted. However, not all combinations will be usable. First of all, it should be clear that $AF + RD < 1$, as otherwise the bins would get overfull. But even when this constraint is not violated, the system could still result in overflows, for example when there is an exponential growth in size, resulting in one or more overfull bins before the migration is finished.

To determine the optimal values for both parameters, we ran several experiments using the values described in Table 3.1. Table 3.2 provides an overview of the results.

As can be seen from these results, there were no overflows using daily iterations when $AF + RD \leq 0.8$. We however like to note that when $MAX \geq 10^6$ there were no overflows at all, even for $AF + RD = 0.9$. The value of $AF + RD$ should be as high as possible, as lower values would lead to more bins being used and therefore a waste of resources. In the remainder of this section, we will focus on two combinations for AF and RD which were valid configurations for all algorithms and sizes, 0.5 ± 0.3 and 0.7 ± 0.1.

3.5.4 Average Bin Usage

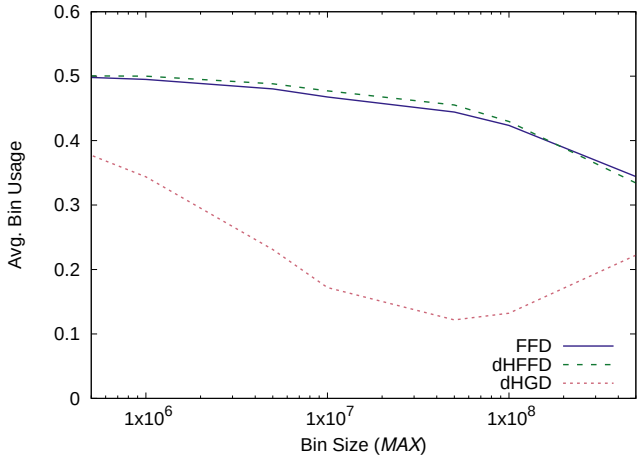
For the static allocation, the bin usage is the most important metric as the primary goal of the algorithm is to minimize the number of storage bins in order to minimize the operational costs. For this metric, we compare the efficiency of our algorithms to the FFD strategy, which we use as benchmark as discussed in Section 3.4.1.

Figure 3.11 illustrates the average bin usage (over all iterations) using the fast-growing dataset, with $AF \pm RD = 0.5 \pm 0.3$ and 0.7 ± 0.1 for the different values of MAX. Figure 3.12 is similar, but illustrates the average bin usage (over all evaluated values for MAX) for the different iterations of the dataset. As can be seen from Figure 3.11, most algorithms will achieve a lower bin usage when using larger bins, which can be explained by the fact that when using larger bins it will be harder to fill the bin and therefore the possibility of wasting more space inside the bin increases. One exception happens when using the dHGD algorithm, for which the bin usage slightly increases when $MAX > 10^8$. This is because when using such large bins, the total number of bins required for allocating all data is limited, and due to the greedy behavior of the dHGD algorithm every subtree is already allocated to a dedicated bin during the first iterations after which no reallocations happen, as we will see clearly in Section 3.5.5, resulting in a higher bin usage. Note that in the plots we only include the dynamic versions of both hierarchical algorithms, as described in Section 3.4.3.2, as the results for the non-dynamic algorithms are almost identical to their dynamic counterparts. In general, dHFFD achieves an average bin usage close to the selected AF, just as FFD, whereas for dHGD the overhead is much bigger (as expected).

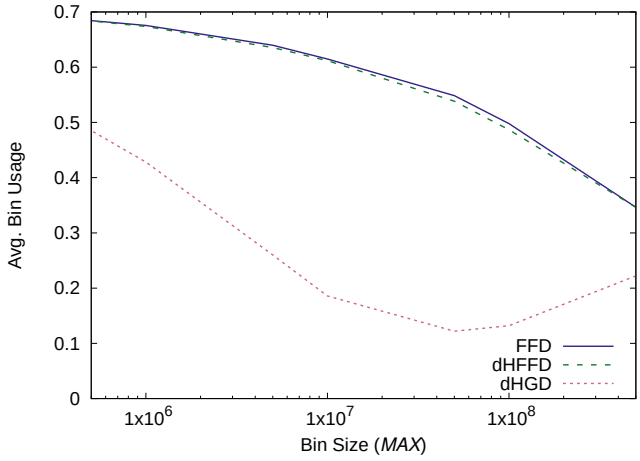
Figure 3.13 on the other hand shows the total number of bins using the fast-growing dataset for 2 different values of MAX. The results also include the total number of bins found by solving the ILP model introduced in Section 3.4.1, implemented using IBM CPLEX Optimization Studio 12.7 [34] with bin size $V = (AF + RD) \times MAX$, the maximum allowed bin usage before reallocation would be triggered. These results also confirm that the dHFFD algorithm achieves very similar results as the FFD algorithm, and the overhead in number of bins is relatively small compared to the optimal solution. We would like to note that dHGD, dHFFD and FFD are invoked with bin size $V = AF \times MAX$ during every reallocation, which explains the difference in overhead between both figures.

3.5.5 Percentage of Migrations and Migration Sizes

Figure 3.14 shows the percentage of migrations (number of migrations over the number of iterations) using the fast-growing dataset, for all evaluated values of MAX. Figure 3.15 shows the average relative migration sizes using the fast-growing dataset, only taking into account iterations in which there actually was a

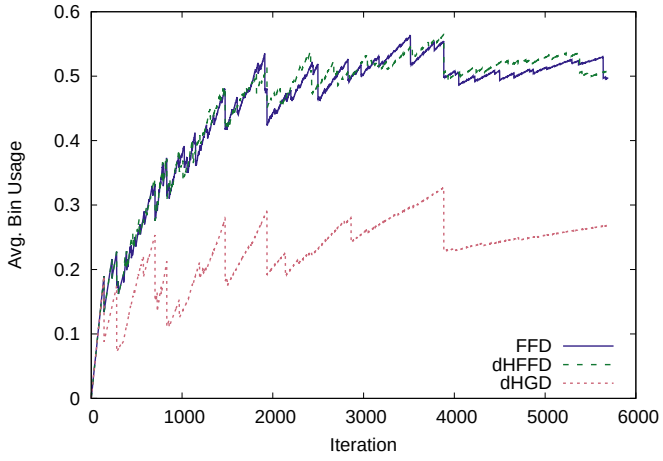


(a) $AF \pm RD = 0.5 \pm 0.3$

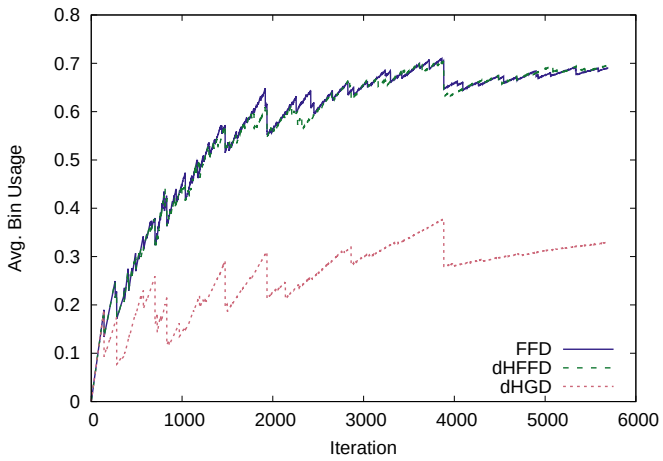


(b) $AF \pm RD = 0.7 \pm 0.1$

Figure 3.11: Average bin usage over all iterations using the fast-growing dataset for the different values of MAX.

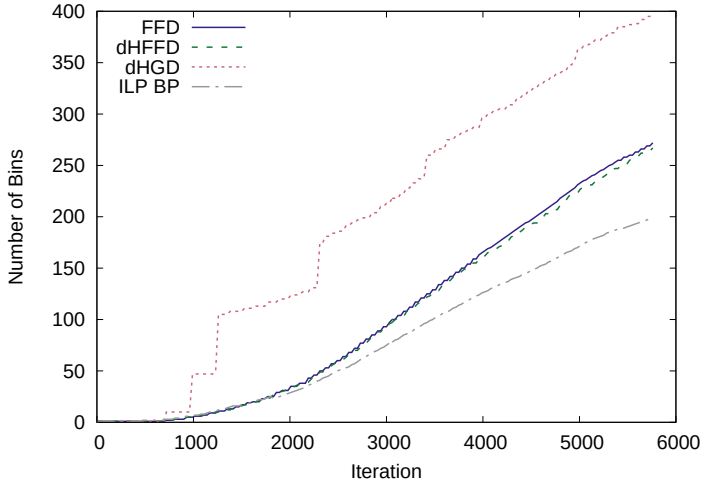


(a) $AF \pm RD = 0.5 \pm 0.3$

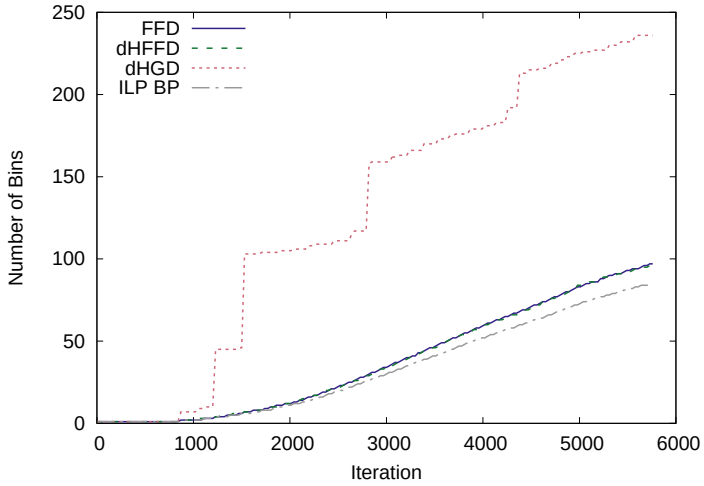


(b) $AF \pm RD = 0.7 \pm 0.1$

Figure 3.12: Average bin usage over all values of MAX using the fast-growing dataset for the different iterations.



(a) $MAX = 5 \times 10^6$, $AF \pm RD = 0.5 \pm 0.3$



(b) $MAX = 10^7$, $AF \pm RD = 0.7 \pm 0.1$

Figure 3.13: Total number of bins using the fast-growing dataset for the different iterations.

migration. As can be seen in Figure 3.14, not every iteration will trigger a migration. By only including actual migrations in Figure 3.15 we get a good estimate about the average amount of data that is migrated during every actual reallocation.

For the HFFD algorithm, migrations are nicely spread over time (not visible in the figures but our experiments confirm this), and the average migration size is already very small thanks to the clustering of related tenants. Furthermore, there are more inexpensive migrations ($L2 + L3$) than expensive migrations ($L1$). The dynamic version of this algorithm, dHFFD, reduces the average migration sizes by a factor 10. The dHGD strategy strongly reduces the number of migrations compared to dHFFD, and there are no $L2$ or $L3$ migrations as the algorithm forces the allocation of tenants to a dedicated instance.

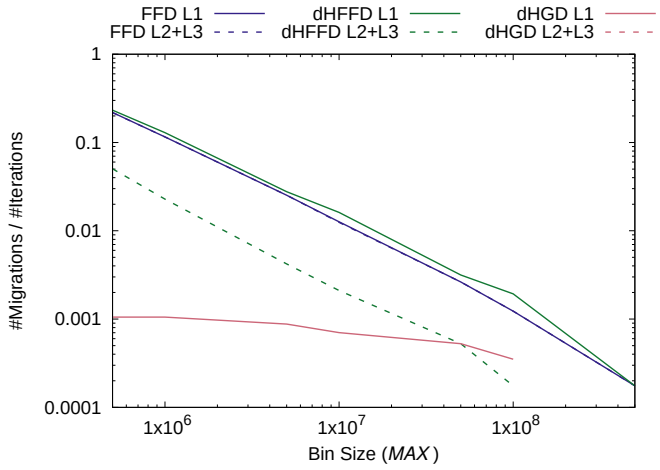
3.5.6 Bin Distance and Ratio of Shared Bins

As mentioned before, the TDS system was designed to cluster related tenants together to have a clear separation of tenants. To evaluate this, we measured the average bin distances over the different iterations for the different algorithms over all executed experiments. Figure 3.16 illustrates the results for the slow-growing dataset over the different iterations. For this metric, the slow-growing dataset is the most interesting dataset as the tenant tree consists of more levels than the tenant tree of the fast-growing dataset. Figure 3.17 on the other hand illustrates the average ratio of shared bins for all algorithms over all values for MAX using the fast-growing dataset.

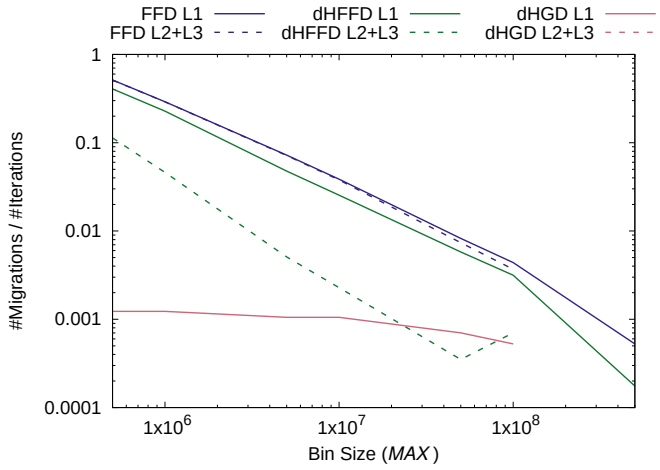
As can be seen from Figure 3.16, the dHFFD algorithm is already performing quite well, as it quickly reaches an average bin distance below 3, signifying that most bins only contain siblings. The dHGD algorithm on the other hand further reduces the average bin distance, but again at the cost of additional bins. Furthermore, both algorithms have a positive effect on the degree of tenant isolation, as can be seen in Figure 3.17.

3.6 Discussion

The FFD strategy is a simple yet efficient approximation algorithm for the bin packing problem, but it does not take the hierarchical structure of the tenants into account. When working with multi-tenant scenarios, clustering of related tenants is important as this will improve both the isolation of performance and tenant data. To achieve this clustering, we introduced two novel hierarchical algorithms for the bin packing problem, HFFD and a greedy variant HGD, and their dynamic counterparts, dHFFD and dHGD. Both algorithms are very similar, but HFFD is designed to minimize the number of bins (similar to FFD), whereas HGD aims to minimize the number of migrations over time. Furthermore, we introduced a

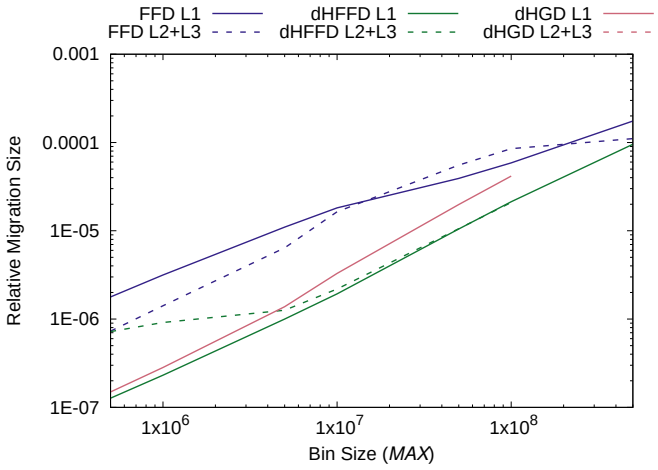


(a) $AF \pm RD = 0.5 \pm 0.3$

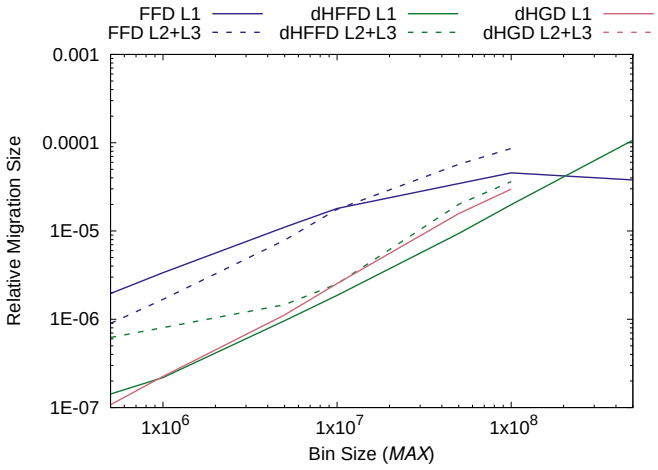


(b) $AF \pm RD = 0.7 \pm 0.1$

Figure 3.14: Average percentage of migrations using the fast-growing dataset for the different bin sizes.



(a) $AF \pm RD = 0.5 \pm 0.3$



(b) $AF \pm RD = 0.7 \pm 0.1$

Figure 3.15: Average relative migration sizes using the fast-growing dataset for the different bin sizes.

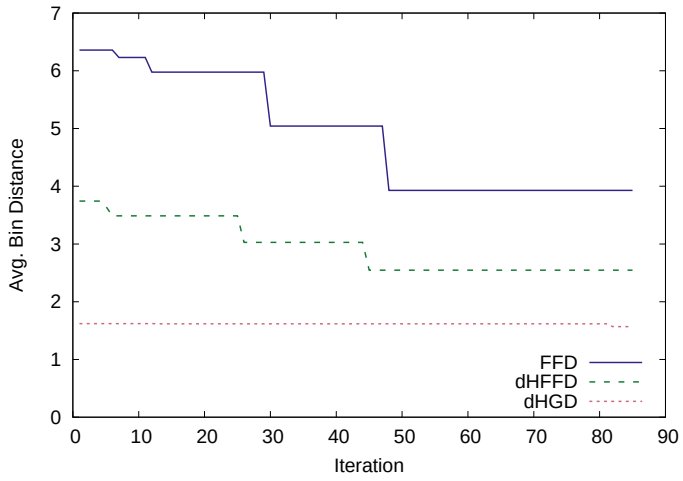


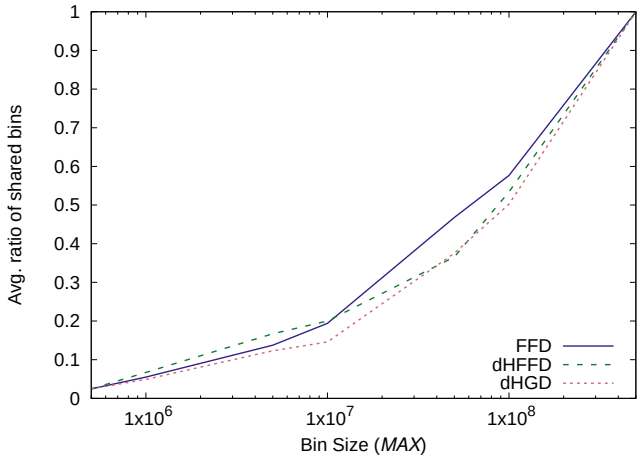
Figure 3.16: Average bin distance for the slow-growing dataset over the different iterations.

dynamic extension for both algorithms to reduce the migration sizes.

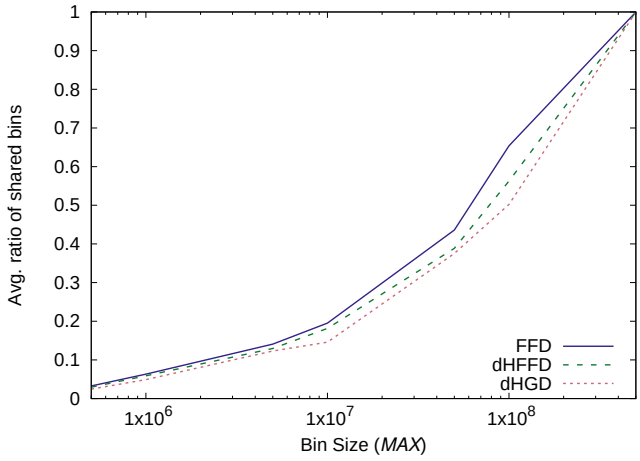
The evaluation results confirm that for the static allocation, dHFFD has a similar bin usage as FFD, whereas dHGD has a much lower bin usage but reduces the number of migrations significantly, and the dynamic variant of the algorithms reduces the migration sizes by about a factor 10. Both dynamic algorithms achieve very low average migration sizes, as on average during every actual migration less than 0.01% of the total amount of data is migrated using either dHFFD or dHGD. When using the greedy algorithm, the number of migrations is very low, as less than 0.2% of the iterations triggered a migration (meaning that on average there was a migration every +500 days), compared to 20% of the iterations triggering a migration (+5 days) when using dHFFD.

The size of a single bin (MAX) will be defined by the scenario and type of hardware used, and in general smaller bins result in higher average bin usages and smaller migration sizes, but they require a lower value for $AF + RD$ when using fixed iterations (for example every day) to avoid overflows. Determining the optimal value for $AF + RD$ in a continuous real-time, real-world scenario is however quite complicated as there are many influencing factors such as the network bandwidth and the future load. Some of them, e.g. the future increase in usage of the different storage instances, are unknown in advance. For a safe scenario with no chance of overflows however, the following approach can be followed.

There are two important factors for the maximum increase in size of a single storage volume over time. The first one is the maximum upstream network



(a) $AF \pm RD = 0.5 \pm 0.3$



(b) $AF \pm RD = 0.7 \pm 0.1$

Figure 3.17: Average ratio of shared bins using the fast-growing dataset for the different bin sizes.

bandwidth towards the storage instance (typically expressed in Mbps). The second factor is the maximum write speed of the storage volume (in MB per second), determined by the type of storage (e.g. traditional hard drives will have a lower maximum write speed than solid state drives) and the used technology (e.g. a premium SAN will have a much higher maximum write speed than a single disk). Both values can be measured, and the maximum increase per second for a single storage volume is the minimum of both values. We will refer to this parameter as MI.

The TDS system periodically checks the usage of each bin to determine if reallocation is required, and we will denote the time period between two consecutive iterations as δt. An overflow will occur when at time t the bin usage is lower than $AF + RD$, while at time $t + \delta t$ the bin usage is higher than 1 (or 100%). The maximum value for $AF + RD$ can therefore be calculated using the following formula:

$$(1 - (AF + RD)_{max}) \times MAX = MI \times \delta t$$

In other words the maximum value for $AF + RD$ corresponds to:

$$(AF + RD)_{max} = 1 - \frac{MI \times \delta t}{MAX}$$

For example, if we are using storage instances with unlimited network bandwidth but a maximum write speed of 100 blocks per second, a single bin has a maximum capacity of 10^6 blocks, and the system polls all storage instances every 1000 seconds, then the maximum value for $AF + RD$ corresponds to:

$$(AR + RD)_{max} = 1 - \frac{100 \times 1000}{10^6} = 0.9$$

Once the system detects an overfull bin, the storage instance can be locked in a read-only state and every change in tenant data can be written to a temporary standby swap storage location. In the meantime, the system can determine a new allocation scheme and migration strategy, followed by the migration of some of the data. Once the data is fully transferred, the changes stored on the temporary swap can be applied to the tenant data. This method is very similar to the approach often used for live migration of virtual machines, such as VMware VMotion [35].

The choice of which algorithm to use for the implementation of the elasticity manager is strongly dependent on the scenario. For most scenarios, dHFFD is the preferred algorithm, as it will reduce the operational costs by minimizing the number of bins, and both the number of migrations as the migration sizes are acceptable. dHGD on the other hand is useful for scenarios in which higher performance is preferred above lower operational costs, as it strongly reduces the number of migrations but at the cost of provisioning additional storage instances. For example, when developing real-time applications with stringent requirements, dHGD is preferred, as migrations can be both expensive and time-consuming. A combination

is also possible, for example when the developed application supports multiple SLAs. In this scenario, dHGD can be used to allocate tenants with higher SLAs, as the cost of provisioning additional instances is compensated by the higher fees for the selected SLA, whereas dHFFD can be used for tenants which prefer a lower SLA at a lower price.

3.7 Conclusions

In this chapter we presented a cloud-based storage system for managing the dynamic allocation of tenant data, which we refer to as the Tenant-Defined Storage system. The system is designed to allocate tenant data from the tenant's perspective, guaranteeing a clear isolation of tenants, and taking custom tenant characteristics into account. In the presented approach, tenants are hierarchically structured using a tree structure, the tenant tree, and the elasticity manager of the system invokes a dynamic resource allocation algorithm to determine a feasible allocation of tenant data over a set of storage resources.

The problem of allocating storage resources for tenant data can be seen as a bin packing problem. As the tenants are hierarchically structured, we introduced two novel hierarchical bin packing approximation algorithms together with a dynamic extension for both algorithms. Similar to the FFD strategy both algorithms are very fast and lightweight, which makes them good candidates for the implementation of a highly scalable TDS system.

Although both algorithms are very similar, they achieve very different results for the static and dynamic allocation of tenant data. dHFFD is optimized for static allocation, as it achieves an average utilization of the provisioned storage instances close to the selected AF, and the overhead in number of bins is very small compared to FFD. For dynamic allocation, migrations over time are nicely spread. During every actual migration on average less than 0.01% of the total amount of data is migrated, consisting for the major part of non-expensive data migrations which can happen within a single data center. The dHGD algorithm on the other hand reduces the number of migrations by a factor 100 compared to dHFFD, making it a valid alternative for real-time scenarios with stringent performance constraints, but this comes at the price of a lower average bin usage and therefore higher operational costs as more storage resources need to be provisioned.

Acknowledgment

The research presented in this chapter is partly funded by the IWT SBO DeCoMAdS [36] project.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. *Above the Clouds: A Berkeley View of Cloud Computing*, Feb 2009. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html.

[2] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao. *A Framework for Native Multi-Tenancy Application Development and Management*. In 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., pages 551 – 558, Tokyo, Japan, July 2007.

[3] A. Azeez, S. Perera, D. Gamage, R. Linton, P. Siriwardana, D. Leelaratne, S. Weerawarana, and P. Fremantle. *Multi-tenant SOA Middleware for Cloud Computing*. In 2010 IEEE 3rd International Conference on Cloud Computing, pages 458–465, July 2010. doi:10.1109/CLOUD.2010.50.

[4] W. T. Tsai and P. Zhong. *Multi-tenancy and Sub-tenancy Architecture in Software-as-a-Service (SaaS)*. In 2014 IEEE 8th International Symposium on Service Oriented System Engineering, pages 128–139, April 2014. doi:10.1109/SOSE.2014.20.

[5] P.-J. Maenhaut, H. Moens, V. Ongenae, and F. De Turck. *Migrating legacy software to the cloud: approach and verification by means of two medical software use cases*. Software: Practice and Experience, 46(1):31–54, 2016. spe.2320. doi:10.1002/spe.2320.

[6] M. Carlson, A. Yoder, L. Schoeb, D. Deel, and C. Prattr. *Software Defined Storage*. Technical report, SNIA, Mar 2014. Available from: http://www.snia.org/sds.

[7] A. Singh and K. Chatterjee. *Cloud security issues and challenges: A survey*. Journal of Network and Computer Applications, 79:88 – 115, 2017. doi:10.1016/j.jnca.2016.11.027.

[8] M. Henze, M. Grossfengels, M. Koprowski, and K. Wehrle. *Towards Data Handling Requirements-Aware Cloud Computing*. In 2013 IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom), volume 2, pages 266 – 269, Bristol, United Kingdom, Dec 2013. doi:10.1109/CloudCom.2013.145.

[9] P.-J. Maenhaut, H. Moens, V. Ongenae, and F. De Turck. *Scalable User Data Management in Multi-Tenant Cloud Environments*. In Proceedings of the 10th International Conference on Network and Service Management

2014 (CNSM2014), pages 268 – 271, Rio de Janeiro, Brazil, Nov 2014. doi:10.1109/CNSM.2014.7014171.

[10] P.-J. Maenhaut, H. Moens, V. Ongenae, and F. De Turck. *Design and Evaluation of a Hierarchical Multi-Tenant Data Management Framework for Cloud Applications*. In 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM), pages 1208 – 1213, Ottawa, Canada, May 2015. doi:10.1109/INM.2015.7140468.

[11] P.-J. Maenhaut, H. Moens, B. Volckaert, V. Ongenae, and F. De Turck. *Design of a Hierarchical Software-Defined Storage System for Data-Intensive Multi-Tenant Cloud Applications*. In 2015 11th International Conference on Network and Service Management (CNSM), pages 22–28, Barcelona, Spain, Nov 2015. doi:10.1109/CNSM.2015.7367334.

[12] J. Li, S. Singhal, R. Swaminathan, and A. Karp. *Managing data retention policies at scale*. In 2011 IFIP/IEEE International Symposium on Integrated Network Management (IM), pages 57 – 64, 2011. doi:10.1109/INM.2011.5990674.

[13] J. Li, S. Singhal, R. Swaminathan, and A. Karp. *Managing Data Retention Policies at Scale*. IEEE Transactions on Network and Service Management, 9(4):393 – 406, 2012. doi:10.1109/TNSM.2012.101612.110203.

[14] J. Li, B. Stephenson, H. Motahari-Nezhad, and S. Singhal. *GEODAC: A Data Assurance Policy Specification and Enforcement Framework for Outsourced Services*. IEEE Transactions on Services Computing, 4(4):340 – 354, 2011. doi:10.1109/TSC.2010.53.

[15] S. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. *Ceph: A Scalable, High-Performance Distributed File System*. In Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06), Nov 2006.

[16] S. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. *CRUSH: Controlled, scalable, decentralized placement of replicated data*. In Proceedings of the 2006 ACM/IEEE Conference on SuperComputing (SC '06), 2006.

[17] D. Malanik and R. Jaek. *The Performance of the Data-Cluster Based on the CEPH Platform with Geographically Separated Nodes*. In Mathematics and Computers in Sciences and in Industry (MCSI), 2014 International Conference on, pages 299 – 307, Sept 2014. doi:10.1109/MCSI.2014.16.

[18] M. Murugan, K. Kant, A. Raghavan, and D. Du. *FlexStore: A Software Defined, Energy Adaptive Distributed Storage Framework*. In 2014 IEEE 22nd

International Symposium on Modelling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), pages 81 – 90, Sep 2014. doi:10.1109/MASCOTS.2014.18.

[19] C.-T. Yang, W.-H. Lien, Y.-C. Shen, and F.-Y. Leu. *Implementation of a Software-Defined Storage Service with Heterogeneous Storage Technologies*. In 2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops (WAINA), pages 102 – 107, Mar 2015. doi:10.1109/WAINA.2015.50.

[20] J. Zhang, H. Huang, and X. Wang. *Resource provision algorithms in cloud computing: A survey*. Journal of Network and Computer Applications, 64:23 – 42, 2016. doi:10.1016/j.jnca.2015.12.018.

[21] M. Masdari, S. S. Nabavi, and V. Ahmadi. *An overview of virtual machine placement schemes in cloud computing*. Journal of Network and Computer Applications, 66:106 – 127, 2016. doi:10.1016/j.jnca.2016.01.011.

[22] R. W. Ahmad, A. Gani, S. H. A. Hamid, M. Shiraz, A. Yousafzai, and F. Xia. *A survey on virtual machine migration and server consolidation frameworks for cloud data centers*. Journal of Network and Computer Applications, 52:11 – 25, 2015. doi:10.1016/j.jnca.2015.02.002".

[23] W. Song, Z. Xiao, Q. Chen, and H. Luo. *Adaptive Resource Provisioning for the Cloud Using Online Bin Packing*. IEEE Transactions on Computers, 63(11):2647 – 2660, 2014. doi:10.1109/TC.2013.148.

[24] S. Rampersaud and D. Grosu. *Sharing-Aware Online Algorithms for Virtual Machine Packing in Cloud Environments*. In 2015 IEEE 8th International Conference on Cloud Computing, pages 718 – 725, June 2015. doi:10.1109/CLOUD.2015.100.

[25] I. Hwang and M. Pedram. *Hierarchical Virtual Machine Consolidation in a Cloud Computing System*. In 2013 IEEE Sixth International Conference on Cloud Computing, pages 196 – 203, June 2013. doi:10.1109/CLOUD.2013.79.

[26] B. Codenotti, G. D. Marco, M. Leoncini, M. Montangero, and M. Santini. *Approximation algorithms for a hierarchically structured bin packing problem*. Information Processing Letters, 89(5):215 – 221, 2004. doi:10.1016/j.ipl.2003.12.001.

[27] T. Lambert, L. Marchal, and B. Uçar. *Comments on the hierarchically structured bin packing problem*. Information Processing Letters, 115(2):306 – 309, 2015. doi:10.1016/j.ipl.2014.10.001.

[28] D. Agrawal, A. El Abbadi, S. Das, and A. J. Elmore. *Database Scalability, Elasticity, and Autonomy in the Cloud*. In Proceedings of the 16th International Conference on Database Systems for Advanced Applications - Volume Part I, DASFAA'11, pages 2–15, Berlin, Heidelberg, 2011. Springer-Verlag. Available from: http://dl.acm.org/citation.cfm?id=1997305.1997308.

[29] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[30] M. Yue. *A simple proof of the inequality FFD (L)* $\leq$ *11/9 OPT (L) + 1,* $\forall$ *L for the FFD bin-packing algorithm*. Acta Mathematicae Applicatae Sinica, 7(4):321–331, 1991. doi:10.1007/BF02009683.

[31] P.-J. Maenhaut, H. Moens, B. Volckaert, V. Ongenae, and F. De Turck. *A Simulation Tool for Evaluating the Constraint-Based Allocation of Storage Resources for Multi-Tenant Cloud Applications*. In NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium, pages 1017–1018, Istanbul, Turkey, April 2016. doi:10.1109/NOMS.2016.7502951.

[32] *FLANDERS.be - the official website of Flanders*. Available from: http://www.flanders.be/en.

[33] *World Bank - Open Data*. Available from: http://data.worldbank.org.

[34] *IBM ILOG CPLEX Optimization Studio*. Available from: http://www-03.ibm.com/software/products/en/ibmilogcpleoptistud.

[35] *VMWare VMotion*. Available from: https://www.vmware.com/pdf/vmotion_datasheet.pdf.

[36] *DeCoMAdS: Deployment and Configuration Middleware for Adaptive Software-as-a-Service of Software Services in the cloud*. Available from: https://distrinet.cs.kuleuven.be/research/projects/DeCoMAdS.

4

Efficient Resource Management in the Cloud: From Simulation to Experimental Validation using a Low-Cost Raspberry Pi Testbed

In Chapter 3 we introduced a dynamic strategy for the allocation of storage resources within multi-tenant cloud environments. The proposed strategy was validated through simulations using a custom developed simulation tool. Resource management strategies however should also consider experimental evaluation using real hardware, as these experiments often result in new insights or they could be used to to more accurately tune the simulation parameters. Unfortunately, running experiments on the public cloud is both costly and time-consuming. This chapter describes a general approach for the validation of cloud resource allocation strategies, illustrating the importance of experimental validation on physical testbeds. Furthermore, the design and implementation of RPiaaS, a low-cost embedded testbed built using Raspberry Pi nodes, is presented. RPiaaS aims to facilitate the step from simulations towards experimental evaluations on larger cloud testbeds, and is designed using a microservice architecture, where experiments and all required management services are running inside containers. The introduced validation approach is then illustrated by evaluating the resource allocation strategy introduced in Chapter 3 on top of the RPiaaS testbed.

⋆ ⋆ ⋆

P.-J. Maenhaut, B. Volckaert, V. Ongenae and F. De Turck

Published in Wiley Journal of Software: Practice and Experience (SPE), Volume 49, Issue 3, pages 449–477, March 2019.

4.1 Introduction

The carbon footprint of data centers increases further as ever more services are hosted in the cloud, leveraging the need for an efficient management of the available resources. By consolidating cloud applications on few physical servers, the remaining servers can be put in standby, resulting in not only a lower energy footprint but also in lower operational costs. Furthermore, efficient resource management can result in higher scalability, as there is a more optimal usage of the available resources.

Over the recent years, a lot of research has been carried out regarding the efficient allocation of cloud resources [1–3], resulting in multiple novel resource allocation strategies. The evaluation of these new strategies however is often performed using only simulations [4], for example by using CloudSim [5] or the more recently developed DISSECT-CF simulator [6]. Although simulators are an important tool for the development and evaluation of new protocols and algorithms for cloud resource management, they have their limitations [7]. CloudSim for example has recently received some critiques [4], for its oversimplified model of I/O processing, the limited communication models, its inaccuracy of communication models and a lack of support for Quality of Service. Furthermore, when using CloudSim, custom extensions are often required, such as CloudSimSDN [8] for validation of SDN-based strategies or ContainerCloudSim [9] for modeling and simulating containers. Simulations are also not standardized, and the applicability of a simulation is strongly dependent on the design of a good data set which corresponds to real world usage, making validation using simulations challenging. According to Barker et al. [10], simulations can be effectively used as a prototyping mechanism to provide a rough idea of how a particular algorithm may perform, but it is very difficult to verify if the simulation environment is an accurate representation of a real world data centre environment. Furthermore, real world data centres are constantly evolving, and are subject to both planned and unplanned changes. If a simulation model is verified at a particular point in time, this will no longer hold if the simulation is rerun at a later point in time.

Therefore, while cloud simulators can be used for the initial evaluation of large batches of experiments, experimental evaluation using real hardware should also

be considered at some point, as these experiments often result in new insights. The evaluated experimental setup could for example introduce additional hardware constraints, which were not taken into account during the design of a new algorithm or simulation tool, or the experiments could be used to more accurately tune the simulation parameters. The use case described in Section 4.6 illustrates this, as the executed experimental evaluation not only allowed for measuring different useful metrics such as the execution times, but also led to important changes to the evaluated allocation strategies. When a new technology emerges, both the simulation tool as the testing environment will require changes to support this technology. However, there is an important difference. As a typical testing environment consists of a set of virtual machines or containers running a common Linux distribution, chances are higher that the required software components are already available, well-documented and tested for this environment. When using a simulation however, the required extensions might still need to be developed, which requires a deep understanding of the underlying mechanics of the simulation tool.

Unfortunately, running experiments on the public cloud is both costly and time-consuming. This is especially true for the design and fine-tuning of new resource allocation strategies, as this often requires multiple incremental iterations of experiments using several cloud instances. Especially when experiments fail during execution, for example due to hardware constraints or a faulty algorithm, this can quickly ramp up the cost. Public cloud computing also has some important limitations, as customers rarely have full control over the underlying hardware resources. This level of control over the hardware is often one of the requirements for resource allocation strategies aiming at the physical hardware level [11, 12]. Users typically have limited control over the physical allocation of the provisioned virtual nodes, and public clouds have their own resource allocation strategies and provisioning mechanisms deeply 'baked in'. A faulty resource allocation experiment could also result in a crash of the environment, which is why it is unlikely that public cloud providers would ever allow this level of access. Therefore, experiments focusing on resource allocation should be executed within an experimental environment, instead of a stable public cloud environment.

When it comes to private or community clouds, large-scale academic testbed environments such as the ones described in Section 4.2.1 are developed in order to support experimentation in a wide variety of research domains and with increased realism compared to simulations. Although these environments allow for large-scale system validation and offer valuable toolsets for experimentation, they have limited infrastructure resource availability as they are heavily used by researchers worldwide, as well as considerable software and hardware maintenance costs. Typically, these testbeds are used for large and mature validation tests and are less suited for small, repetitive tests with highly frequent updates. These environments are typically also rack-mounted and therefore impractical for off-premise

demonstration purposes.

In this chapter, a general approach for the experimental validation of novel cloud resource allocation strategies is presented, together with the design of a low-cost and energy efficient embedded cloud testbed built using Raspberry Pi nodes. This testbed offers an inexpensive and easy-to-use environment for the initial experimental validation of resource allocation strategies, before moving the experiments to a large-scale cloud testbed. The developed software is designed using a microservice architecture, and provides a REST interface for monitoring all relevant metrics, together with a web-based interface for automated configuration and deployment. The testbed is designed to facilitate the step towards experimental evaluation, without the need for having to dive deep into the complex details of the available testbeds.

The remainder of this chapter is structured as follows. The next section provides an overview of related work within the field. In Section 4.3, a general approach for the experimental validation of novel cloud resource allocation strategies is described, highlighting the importance of experimental validation. Section 4.4 introduces the embedded Raspberry Pi testbed, which is evaluated in terms of performance and costs in Section 4.5. In Section 4.6 we illustrate our approach by using the Raspberry Pi testbed for the validation of a custom resource allocation strategy focusing on the allocation of hierarchically structured tenant data. We finish this chapter with our conclusions and avenues for future work in Section 4.7.

4.2 Related Work

This section consists of two subsections. First, we provide a brief overview of existing cloud testbeds that could be used for the experimental validation of resource allocation strategies. This overview includes both platforms that could be used for configuring a new cloud testbed as well as existing large-scale infrastructures that are available to researchers to build their own clouds. Next, we take a look at other microcloud testbeds that have been built using Raspberry Pi nodes and illustrate the main differences with the testbed described in Section 4.4.

4.2.1 Cloud testbeds

A summary of large-scale testbeds is provided in Table 4.1. The Taiwan UniCloud testbed [13] is a community-driven hybrid cloud platform for academics in Taiwan. The main goal of UniCloud is to leverage resources in multiple clouds among different organizations to cope with sudden changes in demand. A self-managing cloud can join the platform to share its resources, while benefitting from other clouds with scale-out capabilities. The proposed platform provides a web portal to operate each cloud via a uniform interface, as well as federated computation

Table 4.1: Overview of large-scale cloud testbeds.

Testbed	**Type**	**Intended Use**
UniCloud [13]	Hybrid cloud platform	Leverage resources in multiple clouds among different organizations to cope with sudden changes in demand.
CloudLab [11]	Distributed cloud infrastructure	Build own cloud for experimenting with cloud architectures, with a deep level of control over the hardware.
Chameleon Cloud [12]	Cloud infrastructure	Create a customized cloud using pre-defined or custom software stacks.
Virtual Wall [14]	Large-scale generic test environment	Advanced network, distributed software and service emulation and evaluation.
Hyperdrive [15]	Highly reconfigurable cloud testbed	Assessing the practical impact of attacks and mitigations on cloud systems.

and storage, by adopting the RESTful APIs of the cloud platforms. Furthermore, the platform supports SLA-based resource provisioning based on the retrieved resource monitoring information. While the architecture of UniCloud is similar to the architecture of Raspberry Pi as a Service (RPiaaS), an important difference is that RPiaaS does not adopt existing APIs, but instead uses a lightweight agent that is installed as a microservice on each node, in order to retrieve the current resource information. The UniCloud platform also tackles inter-cloud issues such as creating VLANs over WAN and live migration across clouds, which could be useful when extending our system to support federation among geographically distributed clusters. One drawback of UniCloud is that users still have to build their own cloud environment(s) to integrate in the platform, which is both costly and time consuming. By adopting the Application Programming Interfaces (APIs) of RPiaaS into the UniCloud platform, it would be possible to add RPiaaS clusters to the UniCloud hybrid cloud platform.

CloudLab [11] is a large-scale distributed infrastructure consisting of almost 15000 cores, distributed across three sites around the United States. It allows researchers to experiment with cloud architectures and the new applications that they enable. It allows a deep level of control over the hardware, such as control and visibility over the virtualization, storage and network layers. CloudLab itself however is not a cloud, but it allows researchers to build their own clouds in an environment that provides a high degree of realism. The Chameleon Cloud [12] is similar to CloudLab, as this is a large-scale cloud testbed consisting of 18000 processor cores across more than 500 cloud nodes with 5 petabytes of storage. Researchers can use the Chameleon testbed to create a customized cloud using either pre-defined or custom software stacks. It provides bare-metal access in order to design, develop, and experiment with innovative virtualization technologies that are not deployed in today's clouds. It also provides a rich set of instrumentation tools, allowing researchers to profile and view their custom environments in a highly granulated way, with detailed traces for all processes. Another example is the iLab.t Virtual Wall [14], a large-scale generic test environment for advanced network, distributed software and service emulation and evaluation. The Virtual Wall consists of over 4000 cores across more than 400 physical servers, which are fully configurable both in terms of their software installation and network topology. CloudLab, the Chameleon Cloud and the iLab.t Virtual Wall allow researchers to create their own cloud, which can be used for the large-scale experimental validation of resource management strategies. However, as we will discuss in Section 4.3, these testbeds are more suitable for larger, mature validation tests and less for initial experiments with a high chance of failure.

Hyperdrive [15] is a highly reconfigurable cloud testbed for experimentally assessing the practical impact of attacks and mitigations on realistic cloud systems, by providing assisted infrastructure setup and configuration. To achieve this, it

Table 4.2: Overview of testbeds built using Raspberry Pi nodes.

Testbed	Node Model	Summary
Abrahamsson et al. [16]	Raspberry Pi 1	Cloud cluster with master node providing configuration and registration.
Pahl et al. [17]	Raspberry Pi 1	Edge PaaS cloud running Docker for use with IoT devices.
Glasgow RPi Cloud [18]	Raspberry Pi 1	Scale model for cloud computing, using LXC containers.
Glasgow RPi Cloud [19]	Raspberry Pi 2 & 3	Updated version, used for evaluating cloud simulators.
Kecskemeti et al. [20]	Raspberry Pi 2	Small-scale cluster used for evaluating DISSECT-CF simulator.

uses a management server that is in charge of the deployment of the hypervisor and the customized images over the platforms. This management server consists of a DHCP server, TFTP Server, HTTP Server, MQTT Server, cloud images and a minimal live Linux OS containing the deployment shell script. The client machines boot over the network using a Preboot Execution Environment (PXE) and fetch the Debian live OS. The architecture of Hyperdrive is very similar to RPiaaS, as the RPiaaS testbed also consists of a management node responsible for the deployment, and several worker nodes who are booting over the network, in order to provide a highly configurable testbed. The main difference with Hyperdrive is that RPiaaS provides a generic cloud testbed that is designed for the validation of resource allocation strategies, whereas the Hyperdrive testbed is tailored for experiments focusing on cloud security attacks and defenses.

4.2.2 Raspberry Pi testbeds

To facilitate the step towards experimental validation, a low-cost Raspberry Pi testbed was built, which can be used for the initial small-scale experimental validation of novel cloud resource allocation strategies. Several other projects exist that have been using Raspberry Pi nodes for building a small-scale cloud testbed, as summarized in Table 4.2. Abrahamsson et al. [16] for example have built a Raspberry Pi cloud cluster consisting of 300 nodes. This cloud is built using Raspberry Pi 1 model B nodes, which is one of the first commercially available Raspberry

models, powered by an ARM 700Mhz CPU and having a maximum of 512 MB of RAM on a single chip. In the testbed, a custom operating system based on Debian is copied onto all individual SD cards, and every node establishes a connection with a master Raspberry Pi node. During startup, the nodes request the configuration files from this master node, and finally they register with the cluster. The master node handles all requests to register and unregister nodes, and monitors all nodes that are currently registered. The testbed was used for implementing a distributed storage system, in which all nodes mount a volume over NFS on a central NAS which is connected to the master node. Updating all memory cards individually however is a cumbersome and time-consuming process, especially given the relatively large number of nodes in the testbed.

Pahl et al. [17] have extended the initial research of Abrahamsson et al. [16] by describing the architecture for an edge PaaS cloud consisting of Raspberry Pi nodes. Combined with IoT devices, computation can then be brought to the edge of the cloud (the Raspberry Pi testbed), instead of sending all data to a central cloud. The authors also use Raspberry Pi 1 model B nodes, and in the proposed architecture every node is running the Docker service to host containers. A Debian 7 image was used to support the core middleware services such as storage and cluster management. For the cluster management, a custom dedicated tool was built for the low-level configuration, monitoring and maintenance of the nodes. A master node handles the (de)registration of the nodes. For the storage management, the authors investigated OpenStack Swift, but as this is quite resource demanding, they eventually also used a central NAS similar to the setup of Abrahamsson et al. [16].

The Glasgow Raspberry Pi Cloud [18] aims to build a scale model for cloud computing infrastructures. This cloud consists of 56 Raspberry Pi 1 nodes running the Raspbian operating system installed on the SD card and every node is hosting three Linux Containers (LXC's) in order to emulate every layer of a cloud stack.

The main differences with the Raspberry Pi testbed described in Section 4.4 are that (i) we focus on building a generic IaaS cloud testbed for the validation of resource allocation strategies, (ii) in which all nodes are mounting a root file system using a network share instead of having the operating system installed on the memory card, and (iii) instead of (de)registering nodes, the master node uses a minimal configuration file containing a list of all nodes in the cluster(s). By using a network share as the root file system, updating the cluster, for example when upgrading the kernel or operating system, is a lot easier as there is no need to update every memory card individually. Furthermore, we also use Docker for hosting the experiments and the required agent services on the different nodes. Our testbed is built using the Raspberry Pi 3 model B, which has a more powerful 1.2GHz 64-bit quad-core ARMv8 CPU and 1GB of RAM, and therefore is better suited for hosting multiple Docker containers.

More recently, the developers of the Glasgow Raspberry Pi Cloud have updated their cloud testbed by replacing the Raspberry Pi nodes by the newer Raspberry Pi 2 and 3 models, and they have used the updated testbed to compare the performance against the corresponding CloudSim models [19]. Their main conclusions are that CloudSim requires a richer set of input features, and that it needs a more complex model for inter-node communication for distributed applications. Therefore, according to the authors, CloudSim currently lacks sufficient accuracy for such experiments, illustrating the importance of experimental validation. The same authors also used their Raspberry Pi Cloud for evaluating two other cloud simulators, GreenCloud and Mininet [21]. For this comparative study they used a cross validation method to compare the predicted performance from both simulators against the actual performance of the Raspberry Pi testbed. According to their findings, the GreenCloud simulator currently does not predict the energy consumption for a micro data center accurately. Mininet on the other hand offers reasonable accuracy in modelling the network performance.

The developer of the DISSECT-CF simulator has recently introduced new models and extensions to estimate the behavior of newer components such as Raspberry Pi nodes in the simulator [20]. In this paper, a Raspberry Pi based cloud is simulated by using the new models. Using a Hadoop-based application scenario, the results from the simulator are compared to results obtained through a real-life system, both environments consisting of 12 Raspberry Pi 2 model B nodes. The authors claim that the results for this experiment are very similar, with a low mean absolute error when using the new models, but further improvements for the simulator are currently still under development.

4.3 Validation of Resource Allocation Strategies

4.3.1 General Approach

When developing a new resource allocation strategy, several steps can be distinguished. Figure 4.1 summarizes the general steps for the design, implementation and validation of novel cloud resource allocation strategies. Initially, a new strategy is designed and implemented. This can be an iterative process, as during the implementation additional constraints may be introduced, requiring modifications to the original design (feedback arrow 1 in the figure). Once the implementation is finished, the strategy should be validated by means of simulations, experimental evaluations on a cloud testbed, or ideally a combination of both. Simulations are often a good start, as these are less costly and less time consuming than experimental evaluations. They can vary from simple unit tests or batch scripts to full simulations of a cloud environment, for example by using one of the simulators listed in [7]. During these simulations, new optimizations can be discovered,

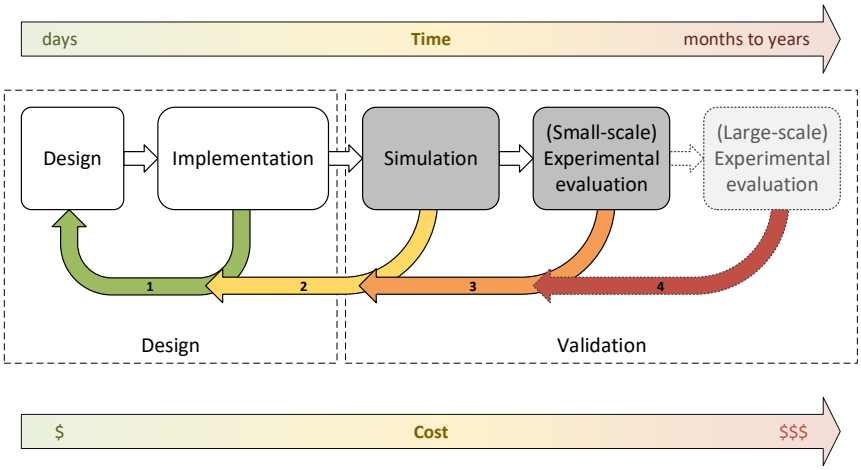


Figure 4.1: General workflow for the design, implementation and validation of cloud resource allocation strategies.

or unforeseen limitations, again requiring changes to the implementation and/or design of the strategy (feedback arrow 2).

Experimental evaluation using real hardware should also be considered as these experiments often result in new insights, and might again require changes to the design and/or implementation (feedback arrow 3). The evaluated setup could for example introduce additional hardware constraints which were not taken into account before, or the experimental results could be used to more accurately fine-tune the configurable parameters. Experiments on physical hardware however are both costly and time-consuming. This is especially true for the design and fine-tuning of new resource allocation strategies, as these often require multiple incremental iterations of experiments using multiple cloud instances. When the executed experiments fail during the execution, for example due to hardware constraints or a faulty algorithm, these experiments can become very costly. Therefore, experiments on relative small-scale testbeds, are initially preferred before doing large-scale experiments.

Although a single powerful server with several virtual machines could be used as a small-scale testbed, this method has some important limitations. First of all, the scalability is limited, as the number of virtual machines that can run in parallel is restricted by the amount of available resources. Ideally, the server should partition its resources into equal shares, and the virtual machines should be clearly isolated so that they do not influence each other. Furthermore, the virtual machines are interconnected using a virtual network, and the customization options for this network are defined by the used virtualization software. Free hypervisors

often have limited features, and are generally installed on top of an operating system, resulting in a noticeable overhead. Commercial bare-metal hypervisors on the other hand come with high licensing costs on top of the required investment for the server hardware. The small-scale testbed introduced in this chapter solves these issues, as it offers a clear isolation between the hosts, allows for a full customization of the network topology, and offers a higher scalability at a lower cost.

Ideally, the new strategy can also be validated on a large-scale testbed. This can for example be done using large-scale academic testbed environments such as the testbeds described in Section 4.2.1. These environments allow for large-scale validation of new strategies, but often have limited resource availability and considerable software and hardware maintenance costs. Alternatively, the new strategy could also be validated using a commercial public cloud provider. Public cloud providers often use either a cost model with a fixed price per instance, or a cost based on the actual usage, making experimental validation using a large number of nodes very costly. Therefore, these large-scale private or public testbeds should be used for large and mature validation tests, and are less suited for small repetitive tests with highly frequent updates.

One possible issue with the presented workflow is that different persons, with a different technical background, might be required during the different steps. While a new strategy is typically designed by a researcher (for example with a mathematical background when the strategy is based on a mathematical model), he or she might require the help of a developer to implement the strategy in a proof-of-concept or within a selected simulation tool. For executing the experiment on a real testbed, a system engineer might be required who has sufficient knowledge about the physical hardware and network topology, in order to get the developed experiment running smoothly on the selected testbed.

4.3.2 Experimental Validation in Practice

Over recent years a lot of research has been done regarding the efficient management of resources in cloud environments. Table 4.3 provides a coarse overview of recent research focusing on resource allocation, and how the described strategies are evaluated (e.g. by means of simulations, small-scale or large-scale experiments or a combination of both). This list consists of publications related to resource management within cloud environments, found using the IEEE Xplore Digital Library [22], which are published between 2015 and 2017. Most of the listed strategies focus on the (static and/or dynamic) allocation of virtual machines, which are often defined by the required amount of CPU cycles, RAM memory, disk space and network bandwidth. Some strategies also take into account the estimated or actual power consumption and required compliance to custom service level agreements.

In this list, it is remarkable that only two strategies were validated by means of experiments on a small-scale testbed, whereas all others were validated through simulations, by using CloudSim or custom developed simulation software. Especially when a custom simulator is used, the quality and credibility of the evaluation results is largely depending on the quality of the simulation software, but more than often sufficient details about how the simulator was implemented are not available.

4.3.3 Towards Embedded Experimental Evaluation

The Raspberry Pi testbed presented in the next section is designed to facilitate the step towards embedded experimentation, by providing a low-cost and easy-to-use small-scale testbed for the initial experimental evaluation of new resource allocation strategies. As illustrated in Table 4.3, most strategies developed within the recent years focus on allocating virtual machines (often referred to as virtual machine packing), and the different resources that are required for hosting the virtual machines (typically CPU, memory, storage and network bandwidth). By providing a simple REST interface for monitoring these resources, the developed algorithms can be easily plugged into the testbed, without having to dive deep into the complex details of advanced cloud platforms such as OpenStack. Instead of virtual machines, the testbed is designed for allocating containers, and the testbed is a perfect fit for validation of strategies aiming at container placement. Container technology has a significantly smaller overhead compared to traditional virtual machines, as the containers are executed directly on the Linux kernel, and the latest Raspberry Pi 3 model B is powerful enough to host multiple containers. Furthermore, when the experiments are designed to be executed within containers, the step towards large-scale testbeds is much easier if these testbeds also offer support for the same container technology, which should not be an issue as most popular container technologies such as Docker can be easily installed on all common operating systems.

4.4 The Raspberry Pi testbed

This section describes the design of the Raspberry Pi testbed, together with RPiaaS, which is used for managing and monitoring the testbed. An initial version of the testbed running RPiaaS was presented during the IEEE INFOCOM 2017 conference [35], but the design and implementation have changed significantly since. All required management services are now running inside Docker containers and all configuration files required by these services are now automatically generated by the master node. In our initial cluster, a Raspberry Pi node was used as master node, and the initial configuration of all services was a time-consuming process. Thanks to the portability of the Docker containers, any Unix-based host can now

Table 4.3: Recent research focusing on cloud resource management.

Publication	Objective	Optimization Method	Validation Method	
			Simul.	Experiments
Liu et al. (2017) [23]	Allocation of VM (job) requests	Deep Reinforcement Learning (DRL)	Custom	-
Portaluri et al. (2017) [24]	Allocation of VM instances based on CPU, RAM, disk usage and consumed network bandwidth	Modified Dijkstra	MATLAB Simulink	-
Maenhaut et al. (2017) [25]	Dynamic allocation of storage resources, based on customizable constraints	Bin Packing	Custom	-
Khoshkholghi et al. (2017) [26]	Dynamic allocation of VMs, taking into account SLA constraints regarding CPU, RAM and bandwidth	Custom algorithms	CloudSim	-
Bi et al. (2017) [27]	Dynamic allocation of VMs, taking into account SLAs, amount of finished and rejected requests and energy consumption	MINLP, Simulated Annealing and Particle Swarm Optimization	Custom	-
Rampersaud et al. (2016) [28, 29]	Allocation of VM instances, taking into account sharing of memory among collocated VMs	Bin Packing	Custom	-
Atrey et al. (2016) [30]	Automatic scaling of SLA-bound SaaS workflows consisting of multiple multi-tenant SaaS services	Custom algorithms	CloudSim	-
Metwally et al. (2015) [31]	Generic model for resource allocation within IaaS clouds	MILP	-	Private cloud (80 servers)
Aral et al. (2015) [32]	Mapping VMs to cloud data centers taking into account resource utilization, VM topologies, performance and resource cost	LAD subgraph isomorphism solver	CloudSim	-
Hieu et al. (2015) [33]	Allocation of VM instances based on current and predicted CPU utilization	Custom algorithm	CloudSim	-
Zhang et al. (2015) [34]	Allocation of VM instances, minimizing resource wastage and power consumption	ILP	Custom	Private cloud (4 servers)

be used as master node, and the initial setup time is heavily reduced as only a central base image and a single configuration file describing the testbed topology need to be created. The developed source code of RPiaaS has been made available to the general public through GitHub [36], and we encourage fellow researchers within the field to try out and customize the code for their own research projects.

4.4.1 Requirements

When building a cloud testbed for the experimental validation of resource allocation strategies, the following requirements should be supported:

- **Easy rollout of custom images**: the Raspberry Pi compute modules are designed to boot the operating system from a memory card installed on the board. In this scenario, updating the operating system on all nodes can be a very cumbersome and time-consuming task, as every memory card needs to be removed, flashed and re-installed. When building a cloud testbed, it should be possible to update the software on the different nodes centrally and only once, and the testbed should then provide a mechanism to deploy this image on all nodes.

- **Easy access towards the current and historical resource usage data**: many resource allocation strategies take decisions based on the actual and/or historical resource usage, in order to select the best suitable node for hosting a certain service. If the testbed provides an easy interface towards this type of information, it should be straightforward to implement the developed algorithms on top of the testbed.

- **Minimal management overhead**: for managing and monitoring the different nodes within the testbed, some additional software will be required that needs to be installed on all nodes. This software should be lightweight, with a low resource usage, leaving enough spare resources on the nodes for executing the experiments.

- **Easy to customize or extend**: every experiment will have different requirements, and it is not straightforward to take into account all possible requirements during the design of the testbed. However, by designing the testbed software in a clean and modular way, it should be easy to extend or customize the testbed to support a wide variety of possible scenarios.

The Raspberry Pi testbed presented in this chapter was designed to support all requirements listed above. For an easy deployment of custom images, the nodes are configured to boot using the network instead of an internal memory card. The testbed provides a dashboard and REST API to monitor the current and historical

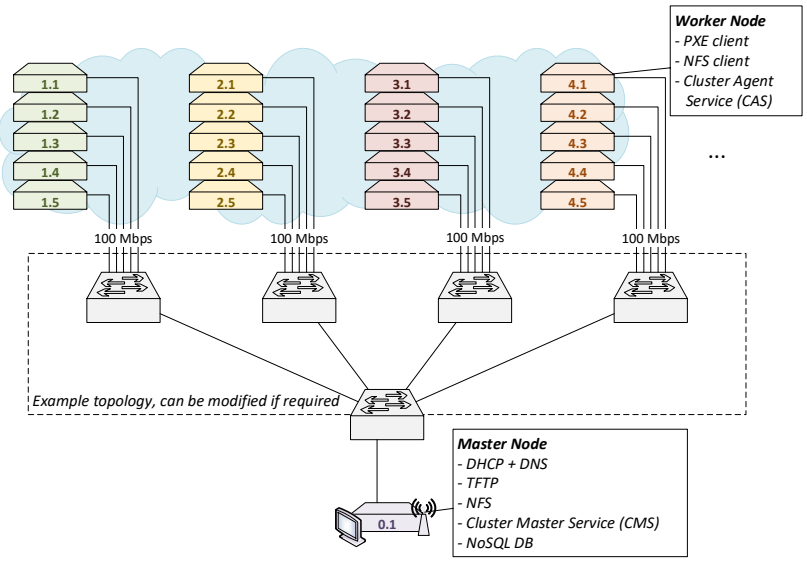


Figure 4.2: Illustrative topology for the Raspberry Pi testbed, consisting of multiple clusters of around 5 nodes.

resource usage. To support this, on every node of the testbed a lightweight service is installed, which is monitoring the local resource usage. The testbed software consists of multiple microservices which are running inside containers, facilitating the deployment of the management services on a heterogeneity of hardware and operating systems, and making it straightforward to customize the software for a wide range of experiments as individual services can be easily replaced.

4.4.2 Architecture

Figure 4.2 provides a coarse overview of a possible Raspberry Pi testbed topology. The testbed consists of multiple worker nodes, aggregated in small clusters of 5 nodes. The testbed was built using the latest Raspberry Pi 3 model B nodes, but any Raspberry Pi model could be used as worker node. Each node in the cluster is interconnected using an Ethernet switch, and the worker nodes are managed through a master node. This master node could either be a Raspberry Pi or any device such as a laptop or desktop running a suitable Linux distribution and the required master services. For managing and monitoring the cluster, RPiaaS was developed, which consists of 2 main components:

- The **Cluster Agent Service (CAS)**, which is deployed onto all worker nodes

and is used to retrieve usage information about the node and to control experiments on this node, and

- The **Cluster Management Service (CMS)**, which is deployed on the master node and is providing access towards the full testbed through an API, together with a web-based dashboard for monitoring and managing the worker nodes.

On the master node, the following services are required:

- A **DHCP** server, which will be used to provision IP addresses to the different worker nodes,
- A **DNS** server, which will be used by all nodes in the cluster,
- A **TFTP** server, used for network booting,
- An **NFS** server, as all nodes will mount the root file system using a network share,
- The **Cluster Management Service (CMS)**, providing user-friendly access towards the nodes, and
- A **NoSQL database system** for storage of historical resource usage data.

Every worker node is running the lightweight CAS for communication with the master node. This service is implemented using Node.js and provides a RESTful API to monitor the resource usage on the node, and to control experiments on the node. The master node is running the CMS, which is implemented using Node.js, Python, HTML, JavaScript and Bash scripting. This service provides a RESTful API towards the whole testbed, together with a web-based dashboard. Table 4.4 summarizes the available API methods which can be used for monitoring the resource utilization on the different worker nodes. In the given path, `{c}` should be replaced by the cluster identifier, whereas `{n}` corresponds to a worker node identifier. Each cluster, and every node inside the cluster should have an unique identifier. All GET requests return a JSON object containing the requested information. Before retrieving information about the cluster or individual nodes, the `ping` method should be invoked first to determine if the selected cluster or worker node is online.

Figure 4.3 shows a small part of the testbed, whereas Figure 4.4 presents a partial screen capture of the dashboard provided by the CMS. The CMS polls all worker nodes using a configurable time interval (e.g. every 60 seconds). During this update, the current resource utilization (CPU load, memory usage and local disk usage) is retrieved from the worker nodes using the RESTful API. This information is then stored in a central database on the master node. By doing so

Table 4.4: Overview of API methods which can be used for monitoring the resource utilization.

Method	Path	Description
GET	/info	Overview of configured clusters
GET	/{c}/info	General information about cluster, including a list of worker nodes
GET	/{c}/{n}/ping	Check if selected node is accessible
	/{c}/{n}/all	Returns all information described below
	/{c}/{n}/uptime	Returns both the system and CAS uptime (in seconds)
	/{c}/{n}/usage/cpu	Returns the actual CPU usage per core together with some information about the CPU
	/{c}/{n}/usage/memory	Returns the actual memory (RAM) usage
	/{c}/{n}/usage/disk	Returns the actual disk usage (based on */data* directory)

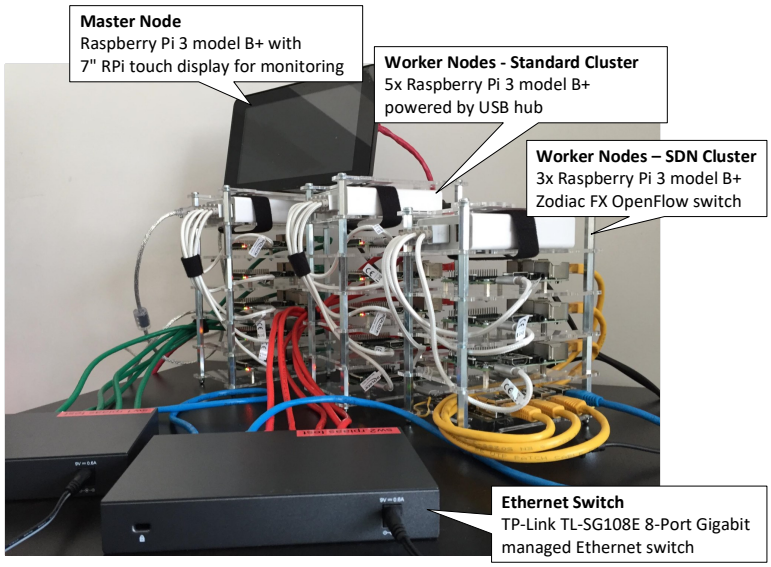


Figure 4.3: A part of the Raspberry Pi testbed, consisting of 2 standard clusters with 5 worker nodes and 1 SDN enabled cluster with 3 worker nodes and a Zodiac FX SDN Controller. In this setup, a Raspberry Pi 3 is used as master node.

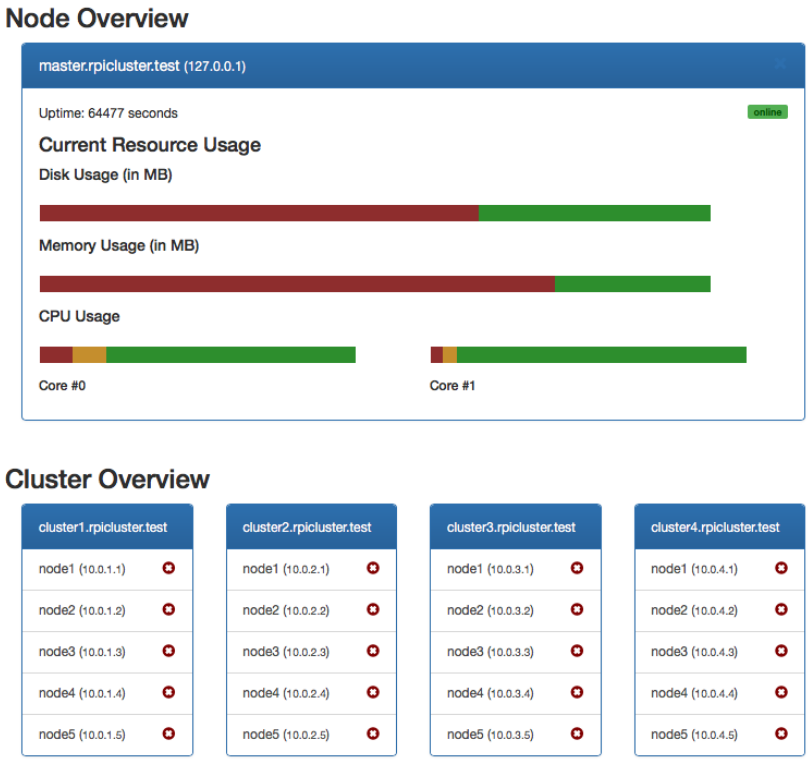


Figure 4.4: Partial screen capture of the dashboard provided by the CMS.

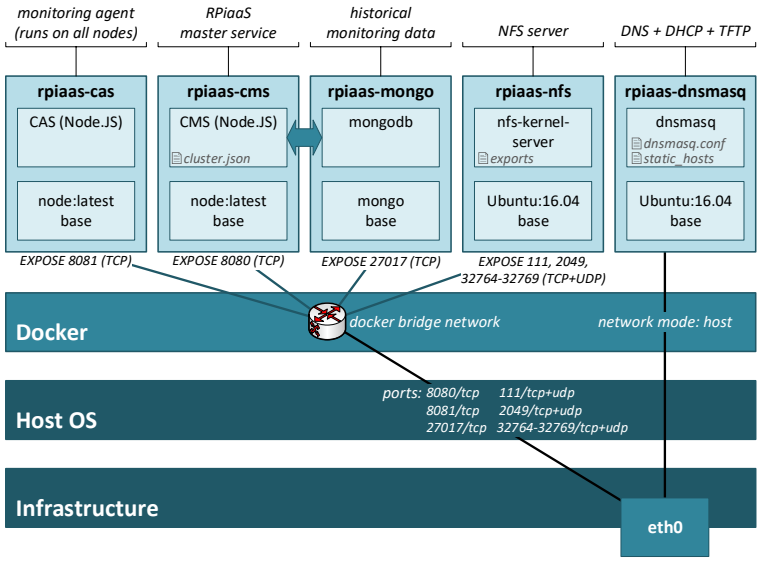


Figure 4.5: Overview of the different RPiaaS microservices, which are running inside Docker containers.

not only the actual resource usage can be displayed on the management interface, but also the historical usage over time for each worker node individually. Both the CMS and CAS are lightweight, consuming a low amount of system resources, as will be illustrated in Section 4.5. Due to the modular design, both services can also be easily extended or customized in order to support a wide range of experiments.

4.4.3 Microservices and Containers

Both RPiaaS services, together with all other required services (e.g. DHCP, NFS) are running as microservices within Docker containers, and all services are deployed using a custom Docker-compose file. This makes it straightforward to deploy RPiaaS on a wide variety of operating systems, and the containers can even be deployed in a Swarm using Kubernetes or Docker Swarm. Separate Docker-compose files are available for deployment on either ARM-based systems (e.g. when a Raspberry Pi is used as master node), or x86/x64-based systems (e.g. for deploying the master services on a Linux-based desktop). Figure 4.5 provides an overview of the different microservices, which are running inside separate Docker containers. The RPiaaS setup consists of 5 different containers:

- **rpiaas-cms** is a Docker container running the RPiaaS CMS described above.

This container is using the latest node image as base image, and a single JSON file is used for the configuration of the testbed. The CMS provides a user-friendly dashboard, together with an API towards the whole testbed. Furthermore, all configuration files required for the other microservices can be automatically generated from within the CMS.

- **rpiaas-mongo** is a Docker container running a MongoDB instance, used for storing the historical resource usage data.
- The **rpiaas-dnsmasq** container hosts an *dnsmasq* instance, responsible for providing DNS, DHCP and TFTP towards the different nodes. As this container needs to listen to DHCP requests on the physical network interface, it is configured to run in the host network mode.
- The **rpiaas-nfs** container hosts an instance of *nfs-kernel-server*, a popular implementation of an NFS server, and is used for serving the base image towards all nodes of the testbed.
- **rpiaas-cas** is a Docker container running the RPiaaS CAS. This is the only service that needs to be deployed onto all nodes, whereas all other services are only deployed on the master node.

While several containers require a custom configuration file, only a single JSON file is required for the initial configuration of the cluster. The CMS container then automatically generates all required configuration files for the other containers. These configuration files can be downloaded using the web-based dashboard, or they can be retrieved by a script using simple *curl* or *wget* requests. Therefore, when launching the services on the master node for the first time, initially only the **rpiaas-cms** container needs to be started, to get the configuration files for the other containers. The configuration files need to be saved in the appropriate directories, after which the other containers can be launched. These steps can be easily automated, for example by using a bash script.

It is worth noting that RPiaaS will only be fully functional on a Unix-based Operating System, as the **rpiaas-dnsmasq** container requires the host network mode. This means that the container shares the network adapter(s) with the physical host. In some Docker distributions, e.g. Docker for Windows or MacOS, the Docker daemon is running inside a virtual machine. As a result, the network interfaces will not be bridged on the physical network adapter(s), but on a virtual network interface connected to the host network using an internal NAT network.

4.4.4 Booting the Worker Nodes

The latest Raspberry Pi 3 model B offers (experimental) support for network booting, using a PXE. However, as older nodes (e.g the original Raspberry Pi and the

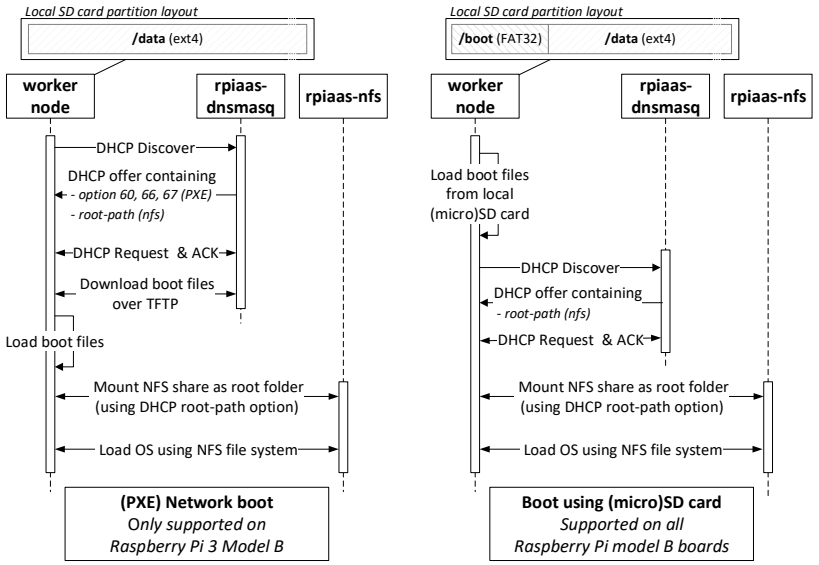


Figure 4.6: A (simplified) comparison of both boot modes. Network booting is only supported on the latest Raspberry Pi 3 model B, and is still experimental.

Raspberry Pi 2 model B) don't support this feature, support for two different boot modes was added, as illustrated in Figure 4.6:

- A full network boot, in which the worker node loads the kernel and boot files from a TFTP server. After loading the kernel and required boot files, the root file system is mounted as an NFS network share. For these worker nodes, the internal SD memory card is only used as local storage (e.g. for storing containers) and neither boot files nor an operating system are located on the memory card.

- Boot from the local memory card, using a network share as root file system. In this mode, the kernel and boot files need to be copied onto a small *FAT32* boot partition of the SD card (e.g. 100MB in size). On this boot partition, the *config.txt* (which is the main configuration file used on a Raspberry Pi upon booting, replacing the BIOS found on conventional computers) is configured to mount the root file system using an NFS network share. After loading the kernel and boot files, the worker node continues to load the operating system using NFS, similar to the previous boot mode. The remainder of the memory card is formatted as a second *ext4* partition, to be used as local storage.

The only difference between both modes is how the kernel and boot files are

loaded. When doing a full network boot (only supported on the Raspberry Pi 3) these are loaded from a TFTP server, making it very easy to update the kernel on all worker nodes as the files are centralized. In the second boot mode, the kernel and other boot files need to be copied onto the memory card, so in order to update the kernel all memory cards need to be updated. Both boot modes however mount the root file system using an NFS network share. As a result, the base image (containing the operating system and required packages) can be updated or modified centrally without the need for removing all memory cards (as long as the new base image uses the same kernel). Section 4.4.5 summarizes the required steps for creating both the boot partition (TFTP or local memory card) and the base image (NFS) in detail.

Using an NFS share as root file system is not something typical for traditional cloud environments, but it is worth noting that cloud data centers do something similar as the physical servers inside a data center typically use iSCSI for mounting LUNs over the network, in order to separate the storage from the computational resources. In this scenario however, the storage network is typically separated from the VM network, and often uses dedicated switches with high-speed fiber connections to achieve a maximum disk throughput. Within the RPiaaS testbed, we could have used iSCSI instead of NFS for mounting the root file system, but this would lead to a more complex setup and configuration, and a higher overhead on the master node. We also did not separate the storage network, but as we will discuss later in this chapter the overhead introduced by NFS will be minimal, because the containers used by the executed experiments will be stored on the local memory card instead of the NFS share.

4.4.5 Preparing a Base Image

In order to create a base image that can be used by all worker nodes, a standalone Raspberry Pi node can be used. After installation of the preferred operating system and all required packages, a full copy of the root filesystem can be transferred over SSH using *rsync*, for example by executing the command from Listing 4.1 on the master node. In this example, the root folder from the standalone worker node (with IP address *node_ip*) is copied using *rsync* to the */nfs/base/* folder on the master node (where the command is executed). Once the root file system is transferred, the */boot* partition of the standalone node can be copied to either the directory on the master node used for TFTP, or to a temporary folder from which we can copy the files back to the boot partition when preparing the memory card for network booting. After copying the files from the */boot* partition, the *cmdline.txt* should be modified as illustrated in Listing 4.2, in order to tell the worker node(s) to mount the root file system using NFS. Note that neither the NFS server or the NFS root path are configured within this file, as these settings are automatically retrieved

Listing 4.1: Transfer the root file system using rsync

```
$ rsync -xa --progress root@node_ip:/ /nfs/base/
```

Listing 4.2: Contents of cmdline.txt *on the* /boot *partition*

```
dwc_otg.lpm_enable=0 console=serial0,115200
  console=tty1 ip=dhcp root=/dev/nfs rw rootwait
  elevator=deadline
```

from the DHCP server upon booting.

Although the CAS can be executed directly on the operating system of the Raspberry Pi, we prefer to install Docker on the worker node(s) and to launch the CAS inside a Docker container. By doing so, other experiments can also be executed inside containers on the nodes, and the worker nodes can even be configured as a Docker cluster. Docker is officially supported on many common operating systems such as Raspbian and can be easily installed on a Raspberry Pi using a single command. It is also recommended to install Docker-compose, as this tool makes it easy to deploy multiple containers at once using a single YAML file. There are various ways to install the Docker-compose tool, but the easiest way is to install this tool using *pip*, a Python based package manager. Listing 4.3 summarizes the commands used for installing both Docker and Docker-compose.

Docker however will fail to start containers when these are stored on an NFS-based file system, as the default OverlayFS storage driver only works with an *ext4* or *xfs* backing filesystem [37]. When starting a container, the Docker daemon will be unable to create an overlay mount for the container, and will return an error message. A workaround could be to use the devicemapper as storage driver instead [38], but this is strongly discouraged by the Docker documentation as it requires configuring direct-lvm in order to avoid very poor performance [39]. Furthermore, it would be bad practice to store all containers on the NFS partition,

Listing 4.3: Installing Docker and Docker-compose on a Raspberry Pi

```
$ curl -sSL https://get.docker.com | sh
$ apt-get -y install python-pip
$ pip install Docker-compose
```

Listing 4.4: Contents of /etc/docker/daemon.json *to configure Docker to store all files inside the* /data/docker *folder*

```
{
  "storage-driver": "overlay2",
  "graph": "/data/docker"
}
```

Listing 4.5: Contents of /etc/fstab *when using network booting with a local memory card*

```
proc            /proc   proc    defaults          0   0
/dev/mmcblk0p1  /boot   vfat    defaults          0   2
/dev/mmcblk0p2  /data   ext4    defaults,noatime  0   1
```

as by doing so the network connection between the worker nodes and the master node could quickly become a bottleneck. A better solution is to configure Docker to store all containers and images (and other related files) on the local memory card instead, for example inside the $/data/docker$ folder. This can be easily done by modifying the Docker daemon file (*/etc/docker/daemon.json*) as illustrated in Listing 4.4.

After transferring the root file system and copying the */boot* partition, the memory card of the standalone node can be replaced by a memory card containing either a single *ext4* partition, or a small boot partition containing the contents from the copied */boot* folder and the remainder of the memory card formatted as a second *ext4* partition, depending on which boot mode from Section 4.4.4 is used. Before booting the worker node, the */etc/fstab* file, containing the necessary information to automate the process of mounting partitions, needs to be modified in order to mount the *ext4* partition as */data* folder on the worker node(s). For example, when using the boot mode with a boot partition on the local memory card, this file can be configured as illustrated in Listing 4.5.

All worker nodes can use a separate copy of the transferred root NFS folder on the master node as root file system, or they can share the same folder. When the executed experiments are running inside Docker containers (which are stored on the local memory card if configured using Listing 4.4), there should be no issues when sharing the NFS folder, but in this scenario it is recommended to create a folder inside $/data$ or even an extra partition on the memory card for storing temporary files such as the */var/log* folder. The new folder can then be linked using a UNIX soft link, or when using an additional partition it can be

automatically mounted by modifying the */etc/fstab* file.

4.5 Performance and Cost Comparison

In this section, we evaluate the performance of the RPiaaS testbed described in Section 4.4, using three benchmark experiments focusing on the CPU performance, disk I/O and memory bandwidth respectively. For each experiment, two versions were implemented: the first is designed to be executed directly on the operating system of the host, whereas for the second version the experiment is executed within a Docker container. Both versions were executed on 4 different environments, being a single Raspberry Pi 3 node, a worker node of the RPiaaS testbed (running the CAS and managed through the CMS), a VM running on a private cloud, and a VM hosted on the Amazon EC2 public cloud environment. These experiments not only provide insights into the performance of Raspberry Pi testbed compared to traditional cloud environments, but they also provide useful insights regarding the overhead introduced by the RPiaaS platform, as well as the overhead introduced by using containers. At the end of this section, we also compare the costs of the RPiaaS testbed to a public and private cloud environment in terms of CAPEX and OPEX.

4.5.1 Benchmark Experiments

4.5.1.1 CPU Performance

The first executed experiment is a CPU benchmark, which is based on the calculation of prime numbers, using the open source Sysbench [40] benchmark suite. In this benchmark, all numbers between 3 and a given maximum number *max_prime* are checked to find out if they are prime numbers, and the total number of prime numbers is returned, as shown in Listing 4.6. This operation is repeated 10000 times, using a given number of threads. The total execution time is measured, as well as the time required per operation.

4.5.1.2 Disk I/O Performance

The second experiment measures the performance of disk I/O, again using the Sysbench [40] benchmark suite. In this test, a predefined number of files are created, up to a given total file size. Some random reads and writes are then executed with an average read/write ratio of 1.5, after which the files are deleted again. As with the previous experiment, this benchmark is also executed using a given number of threads. The total execution time is measured, as well as the time required per operation and the average disk I/O speed.

Listing 4.6: Relevant code from the Sysbench CPU benchmark

```
for(c = 3; c < max_prime; c++) {
  t = sqrt((double)c);
  for(l = 2; l <= t; l++) {
    if (c % l == 0)
      break;
  }
  if (l > t)
    n++;
}
```

Listing 4.7: Overview of the different copy methods used within the mbw benchmark

```
/* First method (MEMCPY) */
memcpy(b, a, array_bytes);

/* Second method (MCBLOCK) */
for(t = array_bytes; t >= block_size;
    t -= block_size, aa += block_size)
  bb = memcpy(bb, aa, block_size);

/* Third method (DUMB) */
for(t = 0; t < asize; t++)
  b[t] = a[t];
```

4.5.1.3 Memory Copy Bandwidth

The final experiment measures the memory copy bandwidth, using the open source tool mbw [41]. In this benchmark, an array of a configurable amount of bytes is copied from one variable to another, using three distinct methods. The first method uses *memcpy* to copy the whole array, the second method also uses *memcpy* but splits the array in blocks of equal size, and the last method copies the array element-wise, as illustrated in Listing 4.7. Each operation is repeated 100 times, and afterwards the measured memory bandwidth is returned for each method individually.

4.5.2 Evaluation Setup

The different benchmarks described above were executed on 4 different environments, as summarized in Table 4.5. For the Raspberry Pi based environments, the

Table 4.5: Overview of the evaluated environments.

	Raspberry Pi 3	RPiaaS node	ESX VM	EC2 t2.micro
Host OS	Raspbian Stretch Lite		Ubuntu Server 16.04 LTS	
CPU	ARM Cortex-A53 1.2GHz (4 cores)		1 vCPU (4 cores)	1 vCPU (1 core)
Memory	1GB LPDDR2 (900 MHz)		1GB vRAM	
Storage	1 node with 4GB MicroSDHC Class 4	1 node with 64GB MicroSDXC Class 10 UHS 1	10GB on SSD	30GB EBS storage

same Raspberry Pi 3 model B is used for both the standalone Raspberry Pi node as the RPiaaS worker node. Two different nodes were configured for both environments, each using a different type of memory card in order to measure the impact on the performance. The first node is using an older MicroSDHC Class 4 memory card, whereas the second node has a MicroSDXC Class 10 with UHS speed class 1 installed. For reference, Table 4.6 provides an overview of the different SD memory card classes available today.

The ESX VM is hosted on a private cloud running VMWare ESX on an IBM BladeCenter H Chassis 8852, consisting of 12 IBM HS22V 7871 blade servers with a 2.67GHz Intel Xeon CPU and 144GB DDR3 RAM each. For the public cloud environment, an EC2 t2.micro instance was selected as this is still one of the most popular instances available today for general purpose computing. By default, the EC2 t2.micro instance uses the Amazon Elastic Block Store (EBS) [42] for persistent storage, and EBS volumes are automatically replicated in order to offer high availability and durability.

Every benchmark experiment was executed using both versions: the first version is executed directly on the operating system of the host, and the second version is executed inside a Docker container. The experiments were first executed on the Raspberry Pi-based environments, and afterwards on the traditional VM environments. Migrating the experiments to the public/private cloud environment was straightforward, especially when using the container-based version, thanks to the portability of the Docker system.

Table 4.6: Overview of SD memory card classes available today.

SD Class	UHS Class	Minimum write speed	Maximum write speed
Class 2	-	2 MB/sec	25 MB/sec
Class 4	-	4 MB/sec	25 MB/sec
Class 6	-	6 MB/sec	25 MB/sec
Class 10	-	10 MB/sec	25 MB/sec
	UHS 1	10 MB/sec	104 MB/sec
	UHS 3	30 MB/sec	312 MB/sec

4.5.3 Evaluation Results

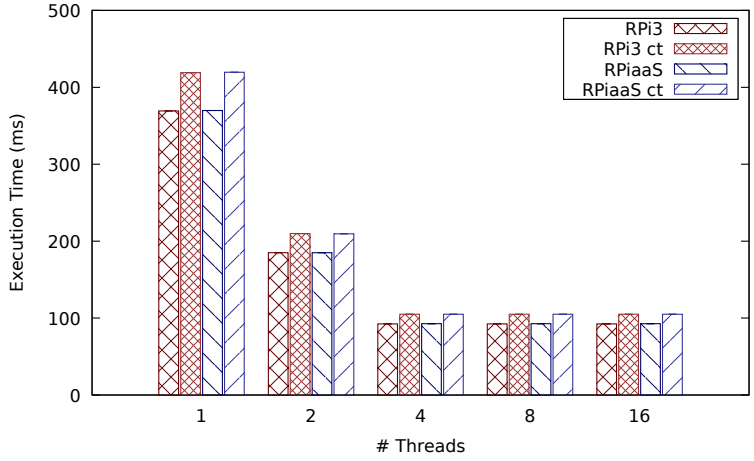
4.5.3.1 CPU Performance

Figure 4.7 illustrates the average execution times of the CPU benchmark for all environments, without and with using Docker containers (*ct* corresponds to the version with containers). For the Raspberry Pi environments, only the results from the nodes using a Class 10 memory card are included. The reason for this is that the results for the nodes with a Class 4 memory card installed are identical, implying that the type of memory card used for persistent storage has no influence on the CPU performance.

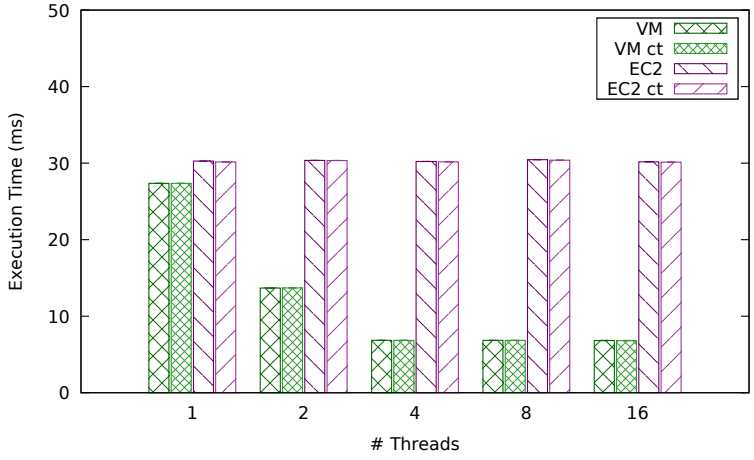
As can be seen from this figure, the performance of the standalone Raspberry Pi and the RPiaaS node are near identical, meaning that the RPiaaS CAS has no impact on the CPU performance of the worker nodes. Using containers however does add a small overhead ($5 - 10\%$) for the Raspberry Pi environments, as these nodes have a much less powerful CPU compared the traditional cloud environments. The overhead is most noticeable when using a small number of threads. This can be easily explained by the fact that the Sysbench CPU experiment is designed to always use the first physical cores on the CPU, and therefore the experiments are always sharing the first core on which the Docker daemon is running. The figures also illustrate that the CPUs of the traditional VMs appear to be around 10 times faster than the CPU of a Raspberry Pi 3, which is an important metric when scaling up experiments before executing them on a large cloud testbed. Finally, the number of threads has no influence on the performance of the EC2 t2.micro instance, which is expected as this type of instance only has a single core.

4.5.3.2 Disk I/O Performance

Figure 4.8 illustrates the average disk I/O speed for all evaluated environments. Some important observations can be made from these results. First of all, for the



(a) Raspberry Pi environments with Class 10 memory card



(b) Traditional VM environments

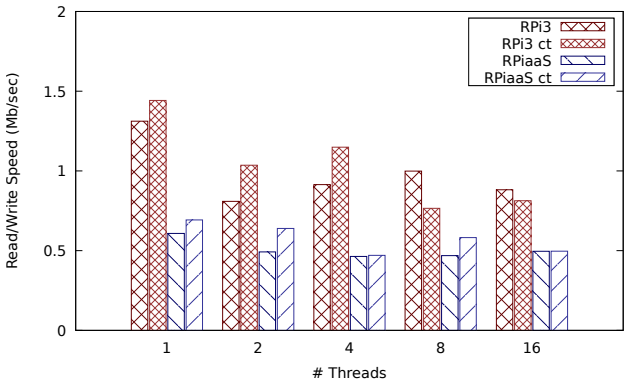
Figure 4.7: Average execution time of the CPU benchmark with $max\_prime = 20000$.

Raspberry Pi environments, the nodes using a Class 4 memory card have significant fluctuations in the measured disk I/O speed, whereas for the nodes with a Class 10 memory card the disk I/O speed appears to be more or less constant. For these environments, RPiaaS also adds some overhead, which is most noticeable for the nodes using the older Class 4 memory cards. For the RPiaaS nodes, some of the data needs to pass over the network to be processed, as the root filesystem is mounted on an NFS share whereas the data from the experiment is written on the local memory card. For the Raspberry Pi environments, using containers however seems to have little impact on the average disk I/O speed, especially for the nodes with a Class 10 memory card. From these results, it is clear that it is worth upgrading to a higher class of memory cards, as these memory cards not only offer higher read/write speeds but also less fluctuations in performance.

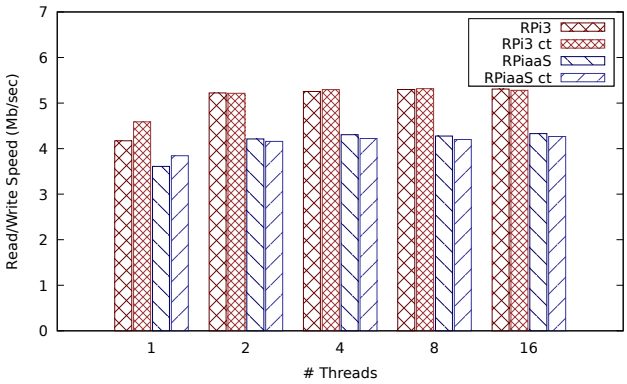
For the private cloud environment, using containers however introduces a small overhead, especially when using a lower number of threads. For the public cloud environment, the measured results might appear odd at first. According to the figure, it seems like the EC2 instance is only performing well when using 1 or 2 threads and no containers. In all other scenarios, the EC2 instance appears to have an average disk I/O speed similar to an SD memory card. This conclusion however is incorrect, as in reality neither using containers or increasing the number of threads will have an impact on the disk I/O speed. The real explanation for these results is that Amazon has some built-in limits regarding its AWS services [43], to prevent users from clogging up resources. For the EBS storage for example, the number of allowed IOPS is limited to $2 * 10^5$. When this limit is exceeded, the performance of the instance is heavily reduced. One iteration of the Sysbench Disk I/O experiment however already performs more than $2 * 10^6$ I/O operations, and all experiments are executed consecutively. Therefore, the defined IOPS limit is quickly reached when executing the experiments. To overcome this limit, individual disk I/O operations could be aggregated into fewer larger operations as much as possible, but this is out of scope for this chapter. It is however worth noting that, when designing experiments for execution on AWS, the service limits should be kept in mind to avoid strange results or even failure of experiments which were running smoothly on other types of testbeds.

4.5.3.3 Memory Copy Bandwidth

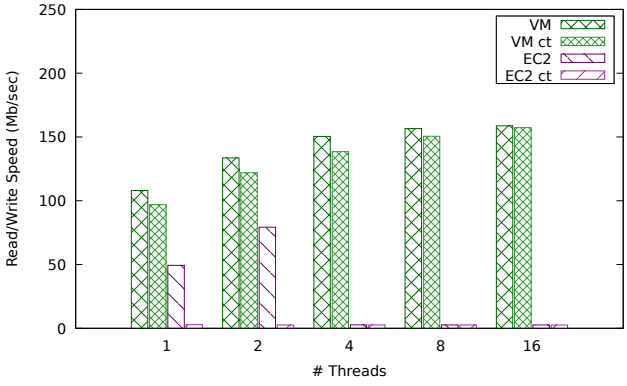
Figure 4.9 and Figure 4.10 illustrate the average memory copy bandwidth for the Raspberry Pi environments and the traditional VM environments respectively, using the different copy methods as described in Listing 4.7. For the Raspberry Pi environments, only the results from the nodes with a Class 10 memory card are included, as the results from the nodes using a Class 4 memory card were similar. As with the CPU benchmark, the type of memory card seems to have no direct impact on the memory bandwidth.



(a) Raspberry Pi, Class 4, total file size = 200MB



(b) Raspberry Pi, Class 10, total file size = 200MB



(c) Traditional VMs, total file size = 1GB

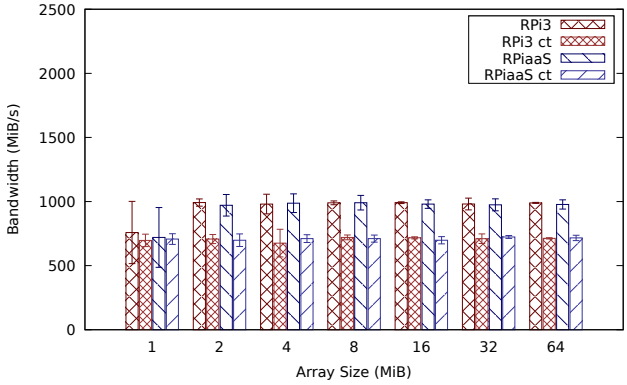
Figure 4.8: Average disk I/O speed as measured by the disk I/O benchmark.

For the Raspberry Pi environments, the results for the standalone node are near identical to the results obtained from the RPiaaS node. Furthermore, when using containers there is a reduction in the measured memory bandwidth of around 25%, but only when using the MEMCPY method. For the other methods, neither RPiaaS or using containers have an influence on the performance. The size of the array to be copied also seems to have no impact on the average memory bandwidth. For the traditional VM environments however, smaller arrays will result in a higher bandwidth. This can be explained by the fact that traditional VMs will have larger memory caches at higher levels, whereas the Raspberry Pi will be forced to write all data directly to the DDR2 RAM. Using containers seems to introduce no noticeable overhead for the traditional VM environments.

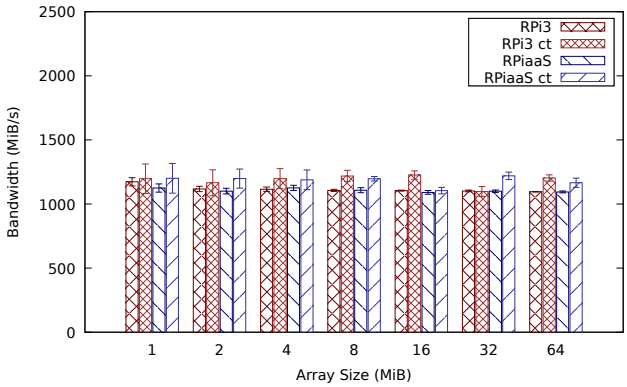
4.5.4 Cost Evaluation

While building a private cloud environment or renting public cloud instances for a longer time interval could quickly become very expensive, building a Raspberry Pi Cluster is much more inexpensive. Table 4.7 illustrates this, as the total price for building a single cluster consisting of 5 worker nodes is less than 500 USD. This cost estimation includes the casing, a 64GB memory card for every node, UTP cables and even a network switch. The total cost can even be further reduced. The stackable casing for example is only a nice to have feature, but not at all necessary. Instead of a 64GB microSDXC card, a microSDHC memory card with a smaller capacity could also be used. As illustrated in the previous section, it is strongly recommended to use memory cards of a higher SD class, as this will result in a significant improvement of the disk I/O speed. The included gigabit network switch is also a bit overkill as the Raspberry Pi nodes only have a 10/100Mbps Ethernet port, and in fact any 100Mbps Ethernet switch could be used. We however preferred the TP-Link TL-SG108E for 2 main reasons. First of all, this switch is managed, meaning that it is possible to define custom Virtual LANs and throttle the bandwidth when required, and it is also possible to monitor the actual network usage on the network switch. Secondly, by using a gigabit switch, a desktop computer with a gigabit Ethernet connection can be used as master node, which comes in handy when multiple worker nodes need to perform a large amount of disk I/O operations on the NFS share.

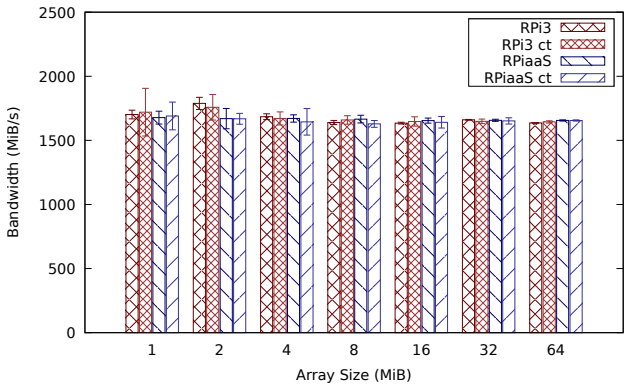
Apart from the small CAPEX, the Raspberry Pi cluster also has a very limited OPEX, as the Raspberry Pi nodes have very low energy usage. All worker nodes are powered by a single 6-port USB hub, which has a maximum current of 650mA and is suitable for an input voltage between 100V and 240V. The Ethernet switch has an input of 220V at 300mA max. These two combined result in a maximum current of 0.95A at 220V, or 209W. In other words, the maximum total energy consumption for a 5 node cluster is about half the usage of a single desktop



(a) MEMCPY

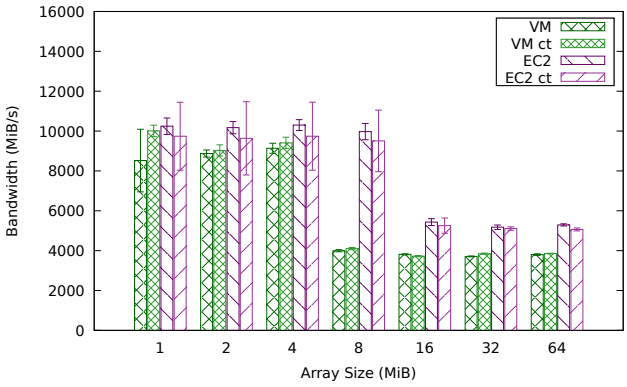


(b) DUMB

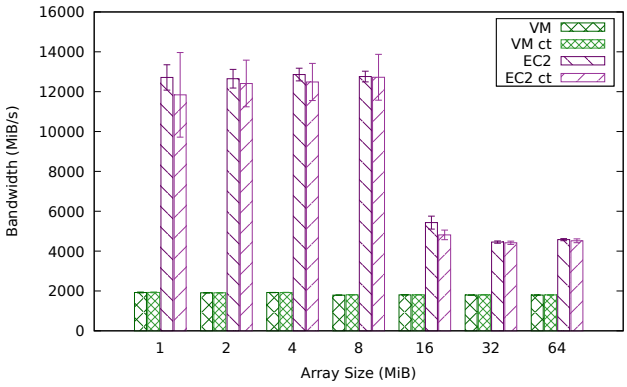


(c) MCBLOCK

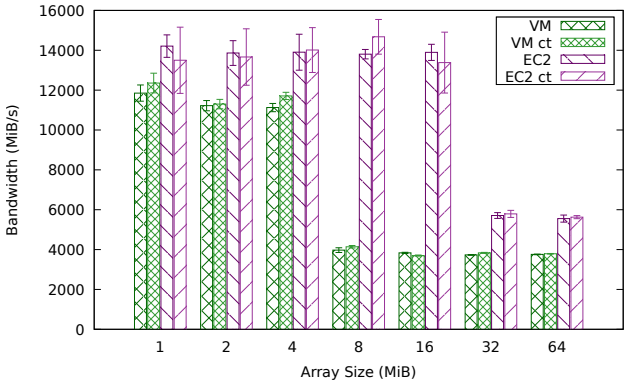
Figure 4.9: Average memory copy bandwidth for the Raspberry Pi environments with a Class 10 memory card.



(a) MEMCPY



(b) DUMB



(c) MCBLOCK

Figure 4.10: Average memory copy bandwidth for the traditional VM environments.

Table 4.7: (CAPEX) Cost of a single cluster (5 worker nodes) in USD.

Item	Unit Price	Amount	Total
Raspberry Pi 3	35	5	175
64GB microSDXC Class 10	25	5	125
Anidées 6-port USB hub	45	1	45
Multi-Pi Stackable Case	15	3	45
Micro USB (power) cable	5	5	25
Ethernet Cable 100MBps	5	5	25
TP-Link TL-SG108E 8-port Gigabit Ethernet Switch	35	1	35
Total price (in USD)			$\pm$ 475

computer with a 450W power supply at 220V. Furthermore, as there are almost no peripherals connected, the actual power usage will be even significant lower, as the USB-hub has a maximum output of 2.5A per USB port, while a Raspberry Pi node without peripherals will only use about 1 to 1.5A. We verified this using a smart plug, and the actual energy consumption for a single cluster (including the Ethernet switch) was between 10W and 15W at 220V. Therefore, it is safe to conclude that the Raspberry Pi cluster is not only low-cost, but also has limited energy consumption, meaning that there is no risk of elevated energy bills when running experiments for multiple days or even months or years.

Building a private cloud is far more expensive, as a single server quickly costs around 2000 USD, excluding any licensing costs, and has a much higher energy consumption (around 500 to 1200W). When no hardware is available, renting instances on a public cloud could be a good alternative, but renting a single t2.micro instance on Amazon EC2 already has a cost of about 10 USD per month.

4.5.5 Discussion

In this section, the presented Raspberry Pi testbed was evaluated using 3 benchmark experiments on 4 environments. Each experiment was executed two times, once directly on the guest operating system and once inside a Docker container. The RPiaaS testbed turned out to be a great tool for the initial evaluation, as once the experiments were running on this environment, the transition to the public and private cloud environment was straightforward, and the same experiments were running on these environments in no time. This is especially true for the version running inside Docker containers, as the only required change was the base image used by the containers. Traditional VM environments require an x86/x64-based image, whereas the Raspberry Pi containers are using an armhf-based image. The

migration however can be easily done by modifying a single line inside the Docker files. The Raspberry Pi testbed therefore is a great tool for the initial development of cloud-based experiments, as when using the testbed failure of experiments is not critical, and experiments can easily be re-executed several times, without incurring high costs. On the private and public cloud however, failure of an experiment is much more of a problem, due to the high cost and/or limited access to these environments. When the experiment executes successfully on the Raspberry Pi cluster, chances are much lower that the same experiment will fail on a traditional cloud testbed. Before running experiments on a public cloud environment however, it is important to take service limits enforced by the provider into account, in order to avoid strange results or even failure of experiments which were running smoothly on other types of testbeds.

Regarding the performance, it is clear that RPiaaS introduces no overhead regarding the CPU and memory performance. It does however introduce some overhead regarding the disk I/O speed, which is due to the fact that the root system is mounted on a network share, and data might need to be transferred over the network when performing I/O operations on the locally installed memory card. The overhead however can be minimized by using a higher class of memory card. In general, for experiments focusing on storage, using a higher class memory card is strongly recommended, as the small additional cost is easily outweighed by the performance gains. RPiaaS consists of several microservices running inside Docker containers, and it is also recommended to run the experiments inside containers, as this makes it easier to migrate the experiments to other testbeds afterwards. Using containers on the Raspberry Pi 3 however introduces a small overhead regarding the CPU performance, which is most noticeable when only a few threads are used. Furthermore, there is also a noticeable overhead regarding the memory copy bandwidth, but only for one out of the three evaluated memory copy methods. For the traditional VM environments, using containers has no significant impact on either the CPU, disk or memory performance.

During the design of RPiaaS, one of our requirements was that the services should introduce little to no overhead, and the results illustrate that we managed to achieve this requirement. When comparing the results from the Raspberry Pi based environments to the results from the private and public cloud environment, it is clear that the traditional VMs offer much better performance regarding CPU, memory and disk I/O. This might seem like a limitation, but when using the Raspberry Pi cluster for the validation of resource management strategies, this can be easily overcome by scaling down the evaluation scenario by an appropriate factor. This was in fact the intended purpose, as the Raspberry Pi testbed should be used initially for executing small-scale experiments, and afterwards a larger version of these experiments can be executed on a larger and more powerful cloud testbed.

Although we did not run into any issues when using a cluster with up to 15

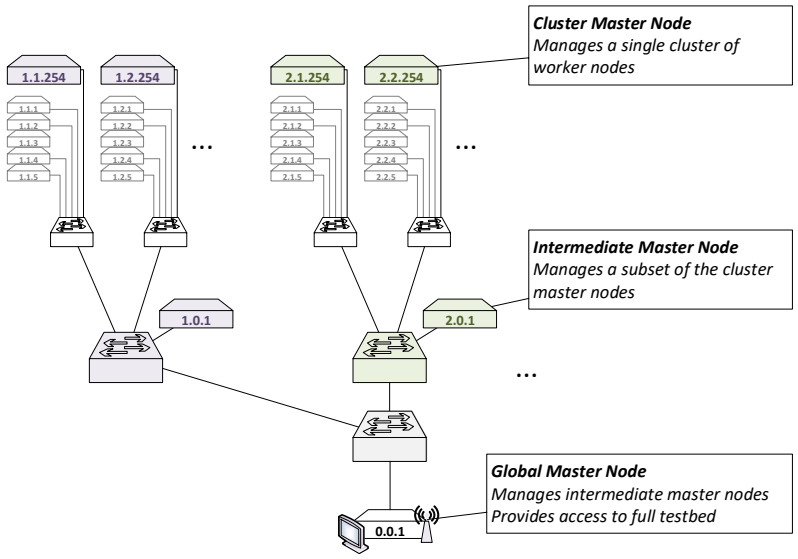


Figure 4.11: An illustrative hierarchically distributed architecture consisting of 3 levels for supporting large-scale clusters. This topology follows the same distributed management approach as many existing Internet services such as DHCP and DNS. If a single master node can manage a maximum of 15 worker nodes, then this topology can already support up to 3375 worker nodes.

Raspberry Pi worker nodes, even when using a Raspberry Pi instead of a Linux desktop as master node, the master node can become a bottleneck when using hundreds or thousands of worker nodes. First of all, the network bandwidth will be a limiting factor, as the worker nodes mount the root file system using an NFS share. The RPiaaS testbed however is designed to execute experiments in Docker containers, and these containers are stored on the local memory card of the Raspberry Pi nodes. As a result, the network bandwidth overhead introduced by NFS is minimal as we will illustrate in Section 4.6. Apart from this, the NFS server could have a limit on the number of concurrent NFS connections possible. This limit will mainly be defined by the number of available ports on the master node, as each NFS connection will create a new socket using a random port. Apart from the limitations introduced by the network, the master node also monitors the worker nodes, which consumes some computational resources. As the number of worker nodes increases, more resources will be consumed, and especially when the configured update interval is small, it is possible that the master node is no longer able to process all retrieved information between two consecutive updates.

When using a large number of worker nodes however, multiple master nodes

can be used, with each master node responsible for monitoring and providing the required services towards a subset of the worker nodes. The different master nodes can even be managed by another master node, creating a hierarchical management structure with a single master at the root. By doing so, there is no limit on the number of nodes, and the monitoring data from the different master nodes could be aggregated to have a full overview of the testbed. For large environments, even more levels can be added in between, as illustrated in Figure 4.11. If the maximum number of nodes that can be managed by a single master node is known (M), then the maximum number of worker nodes in the cluster equals M^L with L being the number of levels in the hierarchical tree. For example, if a single master node can manage a maximum of 15 worker nodes, then the maximum number of worker nodes is 15^3 or 3375 when using the 3-level topology of Figure 4.11.

Regarding the costs, it is clear that the Raspberry Pi testbed is both low-cost and has a very low energy footprint. Therefore, it should not be a problem to execute experiments that need to run for several days or even weeks to months. Running these experiments on a public cloud testbed could quickly become expensive, and building a private cloud requires a high initial investment. Furthermore, the Raspberry Pi cluster is very portable, and is an ideal tool for demo purposes, as a 15-node cluster easily fits into a single flight case. The whole testbed is also fan-less and therefore silent, in contrast to traditional servers which are typically located in a data center with limited access, and which are producing a high level of noise while requiring specialized cooling equipment.

4.6 Case Study

To illustrate the approach introduced in Section 4.3, a resource allocation strategy focusing on the management of hierarchically structured tenant data is evaluated in this section, using a custom simulation tool as well as a practical implementation using the RPiaaS testbed described in Section 4.4. A comparison is made between the results obtained through simulations and experiments on the RPiaaS testbed. The RPiaaS experiments not only allowed for measuring other useful metrics, but also lead to new insights regarding the migration strategy, illustrating the importance of experimental validation.

4.6.1 Context

The management of tenant data within cloud applications is subject to stringent constraints regarding the reallocation of existing tenant data. Migration of existing data is both costly, due to the required bandwidth for performing the migrations, as time consuming. As a result, the number of migrations and the migration sizes over time should be minimized. The allocation of tenant data over multiple storage

Table 4.8: Overview of the evaluated allocation strategies.

Abbreviation	Name	Description
FFD	First-Fit Decreasing	Heuristic for the bin packing problem - this algorithm serves as a baseline.
HFFD	Hierarchical First-Fit Decreasing	Adaption of the FFD algorithm for the hierarchical bin packing problem.
dHFFD	Dynamic Hierarchical First-Fit Decreasing	Dynamic version of the HFFD strategy.

instances can in fact be seen as a bin packing problem, and existing bin packing algorithms could be used to solve the data allocation problem.

Traditional solutions for the bin packing problem handle the items to be packed as individual items with no relations between them. The First-Fit Decreasing (FFD) strategy for example sorts all individual items in decreasing order, and allocates each item to the first available bin that fits the item. This will result in a high average bin usage, using a minimal number of bins, but it will not necessarily minimize the amount and size of migrations.

When the tenants however are hierarchically structured, the allocation problem can be seen as a hierarchical bin packing problem, which goal is to minimize the number of bins while keeping related items, e.g. siblings belonging to the same parent node, together as much as possible. There are multiple reasons to organize the tenants using a hierarchical structure [25]. Tenants for example can be organized based on their geographical location, in order to allocate each tenant in a region close to its physical location.

In our previous work [25] we introduced a dynamic storage system for the allocation of hierarchically structured tenant data, together with several novel strategies for the hierarchical bin packing problem. The system invokes the selected strategy for the initial allocation of the tenant data, and reinvokes the strategy whenever some of the data needs to be reallocated. It uses three parameters: the bin size (MAX), an allocation factor (AF) and reallocation delta (RD). The MAX parameter is defined by the type of node, and corresponds to the maximum amount of data that can be stored on a single node. The AF parameter ($0 < AF < 1$) is used during (re)allocations, as the selected strategy will try to fill each bin (node) up to $AF \times MAX$. The RD ($0 < RD < 0.5$) parameter is used to trigger reallocations, as the system will reallocate tenant data when one of the nodes holds either less than $(AF - RD) \times MAX$ (= underloaded node) or more than $(AF + RD) \times MAX$ data (= overloaded node).

Table 4.8 lists the selected data (re)allocation strategies which will be evaluated in this section. FFD is an implementation of the First-Fit Decreasing heuristic, and will serve as a baseline for our evaluations. HFFD is a newly developed heuristic for the hierarchical bin packing problem. When one of the nodes is under- or overloaded, either all items can be reallocated (by executing the algorithm on the full input set), or only the items located within the under- and overloaded node(s) can be reallocated. The HFFD strategy will reallocate all items, whereas dHHFD will only migrate items that are located on the underloaded and overloaded instances. Note that these items can be migrated to one of the existing worker nodes, the dynamic version of the algorithm only ensures that no other items are being migrated.

4.6.2 Simulations

After the design and implementation of the resource allocation strategies, we ran a wide range of experiments using a custom simulation tool [44]. The simulation tool virtualizes a set of nodes, and allocates the tenant data based on a given input data set using the selected allocation strategy. The simulation tool measures both the bin usage for each active node as well as the expected migration sizes. The migration sizes give a good indication about the time required for the migrations, as a bigger value for this metric will result in an increase of the migration time. It would be interesting to calculate the actual time required for each migration, but this is not straightforward when using simulations. Multiple factors need to be taken account, such as the maximum number of migrations that can be executed in parallel, the available network bandwidth and the disk I/O speed of the worker nodes. However, when using physical hardware instead of a simulation tool, the actual migration times can be easily measured.

Using the simulation tool, we ran several experiments using different values for the configurable parameters MAX, AF and RD. An in-depth explanation of these parameters and the full simulation results can be found in [25], but in Section 4.6.4 we will highlight some of the most interesting results and compare them to the results obtained through experimental evaluation.

4.6.3 Experimental Evaluation

In order to evaluate the strategies on physical hardware, we implemented a distributed data storage system which can be executed on the RPiaaS testbed. This storage system consists of two main components, which are designed as microservices running inside Docker containers:

- The **storage agent** runs on all worker nodes and is responsible for managing the tenant data located on the selected node.

Table 4.9: Overview of the main operations of the storage agent.

Method	Path	Data	Description
GET	/get/{name}	-	Download file with given {name}
POST	/update	{name} (string), {size} (int)	Update file with given {name} to the provided {size}
POST	/transfer	{name} (string), {source} (string)	Transfer the file with given {name} from the {source} node to this node
POST	/delete	{name} (string)	Delete file with given {name} from this node

- The **storage master** runs on the master node and manages tenant data using the storage agents on the worker nodes. This component invokes the selected allocation strategy whenever reallocation of data is required.

Both components were implemented using a combination of Node.js and Python code, and the allocation strategies were implemented as pluggable modules using C++. Although the hierarchical bin packing problem is NP-hard, the implemented strategies are heuristics that are able to find a feasible allocation scheme for the evaluated datasets within a few seconds. The **storage agent** provides a REST interface with four main operations to manage the local tenant data, as listed in Table 4.9. The `/update` method is used to update a file on the node. When the file does not exist, a new file is created on the node, otherwise the existing file is appended or truncated to the new size by adding or removing random bytes. The `/transfer` method transfers a file directly from another worker node using the `/get` method, and saves it to this node. When the transfer has completed, the `/delete` method is invoked on the source node to remove the original file. The **storage master** keeps track of all data located on the worker nodes and monitors these nodes. When one of the nodes triggers reallocation, the storage master invokes the selected strategy and migrates the tenant data using the REST interfaces of the storage agents.

For the actual reallocation of data, the system calculates the minimal migration path between the old allocation scheme and the new allocation scheme. After calculating the migration path, the system executes all required data migrations. We initially implemented the system to perform all data migrations in parallel, but this however lead to the crashing of some of the worker nodes as during the migrations these nodes ran out of free space. More concrete, this happens when an existing node contains items that will be migrated, while at the same time the node would receive items from other nodes. To avoid running out of space, the execution order for the migrations is important, as these nodes should first migrate

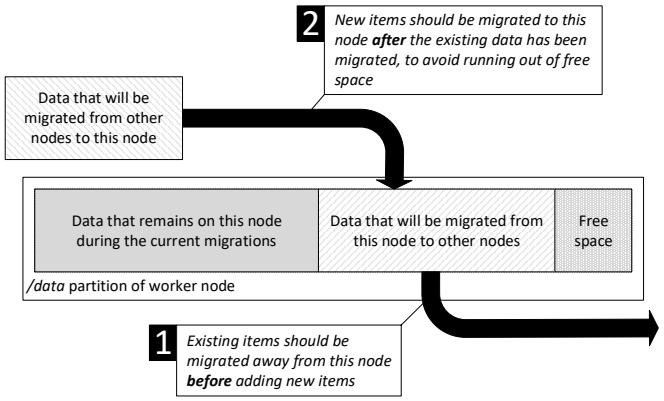


Figure 4.12: When executing the required migrations, it is important to first migrate the existing items away from the node before migrating new items to this node, to avoid that the node runs out of free space.

existing items to other nodes, before transferring items from other nodes, as illustrated in Figure 4.12. This issue did not occur during the simulations, as with the simulation tool all migrations are executed instantaneously. A possible solution could determine an optimal execution order for all migrations, but this could lead to a deadlock situation in which two or more existing nodes are waiting for each other to complete. An alternative solution is to partition the set of required migrations into two subsets: the first subset contains all migrations to new nodes (nodes that were not allocated before), and the second set contains all migrations to existing nodes. The first set of migrations is executed first, freeing up some space on the existing nodes, before the second set is executed. After implementing this behaviour, all experiments executed successfully, while avoiding the risk of a deadlock situation.

The worker nodes have a memory card of 64GB installed. The Docker containers however consume some space, resulting in about 60GB of free space on the local memory card which can be filled by tenant data. However, due to the limited network bandwidth and disk I/O speed of the raspberry Pi nodes, we reduced the size of the */data* partition to 10GB in order to speed up our experiments. By doing so, it takes less than one hour to fill up the */data* partition on a single node instead of several hours, and the execution time of a single experiment is reduced from several days to several hours. Furthermore, we created a separate */docker* partition on the memory card to store all Docker-related files (mainly images and containers) so that these files are not taken into account when calculating the disk usage.

During the execution of the experiments, both the bin usage for every active

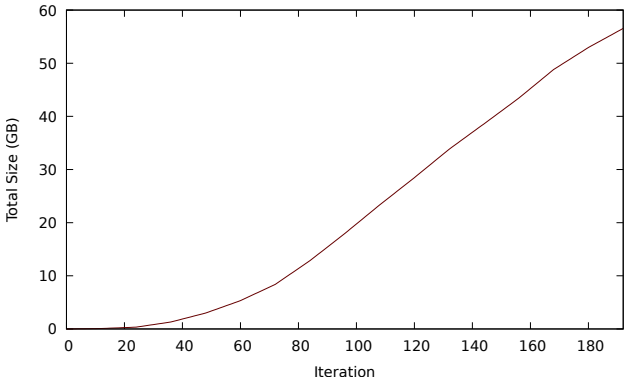


Figure 4.13: Total size (in GB) of the resized dataset over the different iterations.

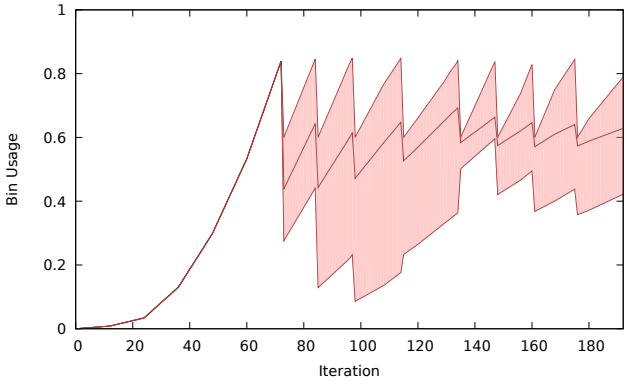
node is calculated after every iteration, as well as the time required for executing all required migrations.

4.6.4 Results Comparison

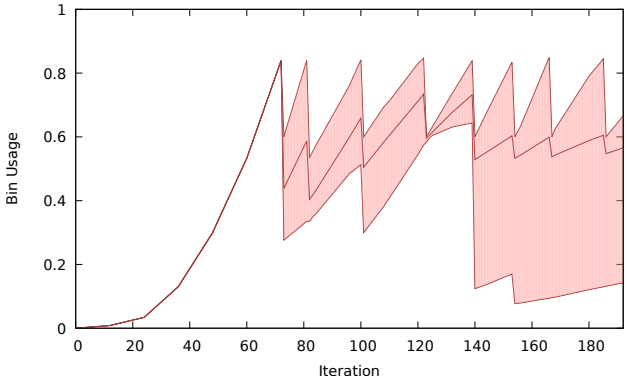
For both the simulations and the experiments on the RPiaaS testbed, a dataset was used consisting of 214 tenants which are hierarchically organised using their geographical location. The dataset consists of 193 iterations, and during each iteration the size of all tenants is updated. Figure 4.13 illustrates the total size of the dataset over the different iterations. This dataset is the Fast-Growing dataset that was used in [25], but it was resized to fit onto our RPiaaS testbed which consists of 15 nodes with a 10GB */data* partition. For this case study, the experiments were executed with a bin size of 10GB (MAX), and $AF \pm RD = 0.6 \pm 0.25$.

Figure 4.14 shows the average, minimum and maximum bin usage (disk utilization) over the different iterations as obtained by the simulation tool, whereas Figure 4.15 shows the measured results from the experimental evaluation on the RPiaaS testbed after partitioning the migrations as described in the previous subsection. The values obtained through RPiaaS are slightly higher due to the overhead introduced by the filesystem, but apart from this the results are near identical to the results obtained by the simulation tool. This overhead however is something that needs to be taken into account when selecting the value for the reallocation delta (RD).

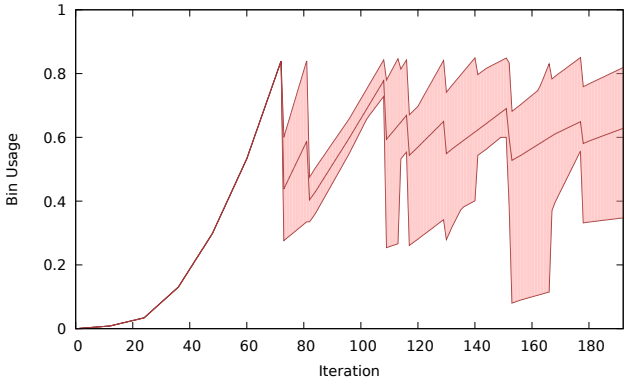
Figure 4.16a and Figure 4.16b illustrate the cumulative migration size (in GB) as obtained by the simulation tool and our initial (crashed) experiments respectively, whereas Figure 4.16c shows the cumulative migration time which was measured during the execution on the RPiaaS testbed after applying the solution described in the previous subsection. As can be seen from these results, the initial



(a) FFD

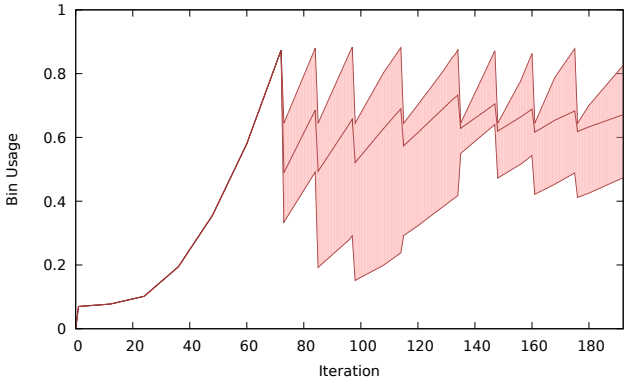


(b) HFFD

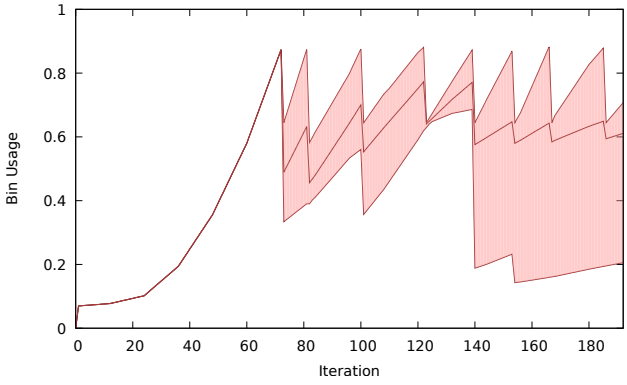


(c) dHFFD

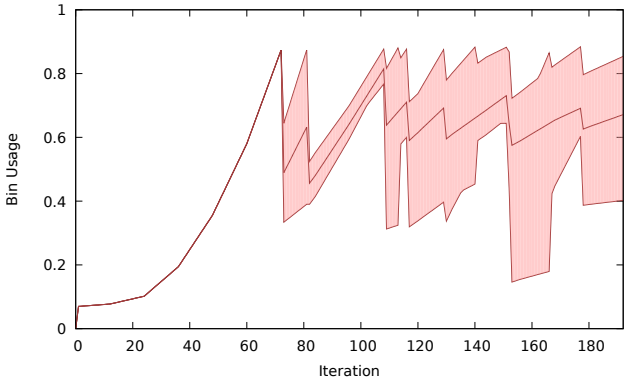
Figure 4.14: Minimum, average and maximum bin usage over the different iterations (simulations).



(a) FFD



(b) HFFD



(c) dHFFD

Figure 4.15: Minimum, average and maximum bin usage over the different iterations (RPiaaS testbed with partitioning).

experiments crashed during the execution of a large migration. After partitioning the migrations into two sets, this issue was resolved, and the obtained results (not included in Figure 4.16) were identical to the simulation results of Figure 4.16a. The HFFD strategy reduces the total migration size by $\pm 25\%$ compared to the baseline FFD strategy, whereas for the dHFFD strategy the migration sizes are reduced by $\pm 66\%$. The migration times however are only reduced by $\pm 10\%$ and $\pm 50\%$ respectively, which illustrates that the migration times do not necessarily have a linear relationship with the migration sizes. This is due to various factors such as the maximum I/O speed, the available network bandwidth, and the number of migrations that can be executed in parallel. Furthermore, the migrations are now divided into two sets based on their destination, and other partitioning methods might yield other results.

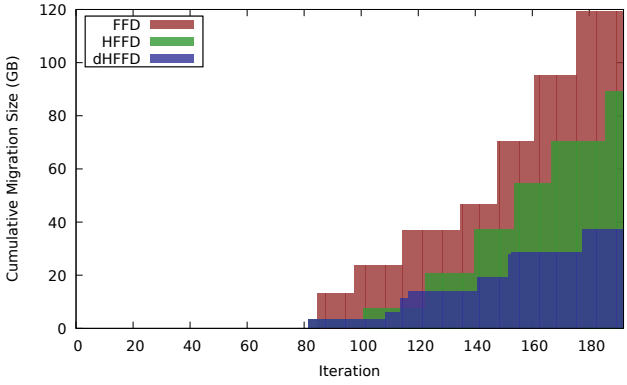
4.6.5 Discussion

In this section, we illustrated the approach introduced in Section 4.3 using a case study focusing on the allocation of hierarchically structured tenant data. After the design and implementation of the various allocation strategies, several simulations were executed using a custom simulation tool as well as an experimental evaluation using the RPiaaS testbed described in Section 4.4. For the executed experiments, we used a fast-growing dataset containing the size of 214 tenants over 193 iterations. During each iteration, the size of every tenant in the dataset is updated, and whenever one of the worker nodes is either under- or overloaded, the selected allocation strategy is invoked resulting in a migration of some of the data.

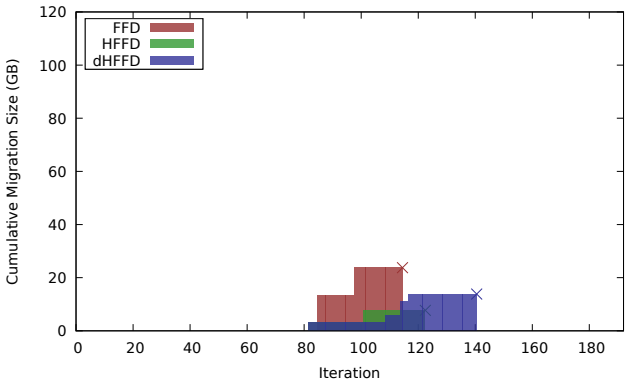
The simulation tool outputs (among other information) the bin usage for each active node over the different iterations, as well as the total migration size. Calculating the actual time required for these migrations however is not feasible with the simulation tool. In order to calculate the migration times, various factors such as the disk I/O speed, the available bandwidth and the number of migrations that can be executed in parallel should be taken into account. However, when executing the experiment on a physical testbed, the actual migration times can easily be measured.

Regarding the bin usage, the results obtained through simulations and the results measured during execution on the RPiaaS testbed are near identical. The experimental results however are slightly higher due to the overhead introduced by the file system. When comparing the migration sizes obtained through simulations to the actual migration times, there is a visible difference. Although the migration sizes are reduced by a factor of 3 when using the dynamic version of our custom algorithm, the actual migration times are only reduced by a factor 2, due to the limited network bandwidth and disk I/O speed of the nodes.

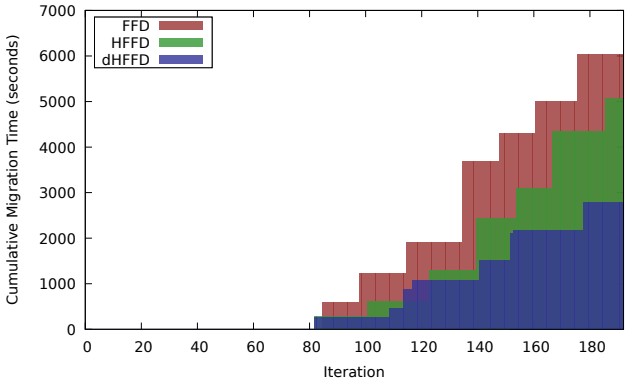
When implementing the experiments for the RPiaaS testbed, we initially ran



(a) Migration size (simulations)



(b) Migration size (RPiaaS testbed without partitioning)



(c) Migration time (RPiaaS testbed with partitioning)

Figure 4.16: Cumulative migration size and migration time over the different iterations.

into an issue which led to the crashing of some experiments. During the initial design of the allocation strategies, no assumptions were made regarding the order of migrations during reallocation, but during the experiments it became clear that this order is important to avoid that some nodes are running out of space. This issue was not visible during the simulations, therefore illustrating the importance of experimental validation. Furthermore, as some changes were required to the design and implementation, we had to run several iterations of the same experiments, which would have been quite costly if we immediately executed our experiments on a large-scale cloud testbed.

During the execution of the experiments, we also monitored the consumed network bandwidth on the master node. The consumed network bandwidth was minimal (less than 100 kbps on average), and mainly consisted of commands sent to the storage agents for managing the tenant data. This again illustrates that the overhead introduced by using NFS on the worker nodes is minimal, due to the fact that all containers and related files are stored on the local memory card.

Execution of the experiments on a large-scale cloud testbed is now straightforward, thanks to the container-based microservice architecture. For a large-scale evaluation, several cloud nodes are required with a basic Linux distribution and Docker installed. All developed microservice containers for this experiment were implemented for multiple architectures, and can therefore be executed on traditional Unix-based environments. On the worker nodes, only the RPiaaS CAS and the storage agent services need to be deployed. As all migrations are executed directly between nodes, the master services can be deployed locally, as long as this node can reach the worker nodes to send the appropriate commands for monitoring the nodes and allocating the tenant data.

4.7 Conclusions and Future Work

This chapter presents a general workflow for the design, implementation and validation of cloud resource allocation strategies. Furthermore, it introduces RPiaaS, a low-cost embedded cloud testbed which was built using Raspberry Pi nodes. Although at some point experimental validation of new resource allocation strategies should also be considered, many novel strategies are often only validated by means of simulations. This is mainly due to the high cost, complexity and limited access of traditional private and public cloud environments. When executing experiments on these cloud environments, failure of experiments should be avoided, and therefore these environments should only be used for mature, large-scale tests. The presented RPiaaS testbed is a good alternative for the development and initial evaluation of novel cloud resource allocation strategies.

The RPiaaS testbed consists of two main components, the CMS which is running on the master node and used for managing and monitoring the testbed, and

the CAS which is deployed on all worker nodes and is used to retrieve information about the current resource usage. Some other services are also required on the master node, for providing DHCP and DNS towards the worker nodes, and network booting the nodes. RPiaaS is designed using a microservice architecture, in which all services are running inside Docker containers. This not only makes it straightforward to deploy all required services at once, but it also allows to deploy the master services on a wide variety of systems and operating systems. For a small-scale setup, the master services could for example be deployed onto a Raspberry Pi node, but when working with larger testbeds consisting of a higher number of nodes, a Linux-based desktop could be used instead.

During startup, the worker nodes mount an NFS folder as root file system, which makes customizing the cluster easy and fast as there is no need to update individual worker nodes. A single base image containing the preferred operating system and all software required for the experiments needs to be prepared only once, after which all worker nodes in the cluster can use this base image. Furthermore, by using Docker on all worker nodes, experiments can be easily deployed onto a single or multiple worker nodes, and the cluster can even be managed using Docker Swarm or transformed into a Kubernetes cluster.

During the design of RPiaaS, one of the goals was to minimize the overhead introduced by the RPiaaS services on the worker nodes. Using containers and/or having to deploy a custom agent service on the worker nodes could introduce some overhead. Several benchmark experiments however illustrated that the overhead is negligible, as the obtained results were near identical when executed directly on the operating system or inside a Docker container, as well as when executed on a standalone Raspberry Pi node or a RPiaaS worker node that is running the CAS and managed through the CMS. Compared to a traditional VM, it is clear that a Raspberry Pi node has much lower performance, but this should not be an issue. When using the RPiaaS testbed, this means that the experiments should be downscaled by an appropriate factor. The obtained results presented in this chapter can help to define this factor, or when using other types of testbeds, a similar experiment can be executed to gain insights regarding the performance.

The described workflow is illustrated using a case study focusing on the allocation of hierarchically structured tenant data. For this case study, three different resource allocation strategies were implemented and evaluated using both a custom simulation tool and the RPiaaS testbed. The results from both evaluations are very similar, but the experimental evaluation on the RPiaaS testbed allowed for measurement of other metrics, while also offering some new insights regarding the evaluated strategy. The RPiaaS testbed proved to be a very useful tool for the evaluation. The initial execution of the experiments failed, requiring modifications to the design and implementation, and several iterations of the same experiments to solve all issues. If the experiments would have been executed directly on a

large-scale testbed, this process would have been very costly or might even have been impossible due to the limited availability of these environments.

Although the RPiaaS cluster was initially designed for the validation of resource allocation strategies, it can be used in a wide variety of cloud-based experiments due to the generic and modular design. The RPiaaS cluster can in fact be seen as an embedded edge or fog testbed. Recently we started adding low-cost SDN-based switches to the clusters, and in the near future we will also start using the RPiaaS testbed as an inexpensive tool for SDN-based experimental research.

References

[1] Z. Lu, S. Takashige, Y. Sugita, T. Morimura, and Y. Kudo. *An analysis and comparison of cloud data center energy-efficient resource management technology*. International Journal of Services Computing, 2(4):32–51, 2014.

[2] B. Jennings and R. Stadler. *Resource Management in Clouds: Survey and Research Challenges*. Journal of Network and Systems Management, 23(3):567–619, 2015. https://doi.org/10.1007/s10922-014-9307-7. doi:10.1007/s10922-014-9307-7.

[3] A. Yousafzai, A. Gani, R. Noor, M. Sookhak, H. Talebian, M. Shiraz, and M. Khan. *Cloud resource allocation schemes: review, taxonomy, and opportunities*. Knowledge and Information Systems, 50(2):347–381, 2017. https://doi.org/10.1007/s10115-016-0951-y. doi:10.1007/s10115-016-0951-y.

[4] Z. Mann. *Cloud simulators in the implementation and evaluation of virtual machine placement algorithms*. Software: Practice and Experience, 48(7):1368–1389, 2018. https://doi.org/10.1002/spe.2579. doi:110.1002/spe.2579.

[5] R. Calheiros, R. Ranjan, A. Beloglazov, C. De Rose, and R. Buyya. *CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms*. Software: Practice and Experience, 41(1):23–50, 2011. https://doi.org/10.1002/spe.995. doi:10.1002/spe.995.

[6] G. Kecskemeti. *DISSECT-CF: a simulator to foster energy-aware scheduling in infrastructure clouds*. Simulation Modelling Practice and Theory, 58(2):188–218, 2016. https://doi.org/10.1016/j.simpat.2015.05.009. doi:10.1016/j.simpat.2015.05.009.

[7] A. Ahmed and A. Sabyasachi. *Cloud computing simulators: A detailed survey and future direction*. 2014 IEEE International Advance Computing Conference (IACC), pages 866–872, Gurgaon, India, February 21–22, 2014. https://doi.org/10.1109/IAdCC.2014.6779436. doi:10.1109/IAdCC.2014.6779436.

[8] J. Son, A. Dastjerdi, R. Calheiros, X. Ji, Y. Yoon, and R. Buyya. *CloudSimSDN: Modeling and Simulation of Software-Defined Cloud Data Centers*. CCGrid 2015. 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pages 475–484, Shenzhen, China, May 4–7, 2015. https://doi.org/10.1109/CCGrid.2015.87. doi:10.1109/CCGrid.2015.87.

[9] S. Piraghaj, A. Dastjerdi, R. Calheiros, and R. Buyya. *ContainerCloudSim: An environment for modeling and simulation of containers in cloud data centers*. Software: Practice and Experience, 47(4):505–521, 2017. https://doi.org/10.1002/spe.2422. doi:10.1002/spe.2422.

[10] A. Barker, B. Varghese, J. Ward, and I. Sommerville. *Academic Cloud Computing Research: Five Pitfalls and Five Opportunities*. 6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14), pages 1–6, Philadelphia, PA, USA, June 17–18, 2014. https://www.usenix.org/conference/hotcloud14/workshop-program/presentation/barker.

[11] R. Ricci, E. Eide, and The CloudLab Team. *Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications*. USENIX ;login:, 39(6):36–38, 2014. https://www.usenix.org/publications/login/dec14/ricci.

[12] H. Mambretti, J. Chen, and F. Yeh. *Next Generation Clouds, the Chameleon Cloud Testbed, and Software Defined Networking (SDN)*. 2015 International Conference on Cloud Computing Research and Innovation (ICCCRI), pages 73–79, Singapore, Singapore, October 26–27, 2015. https://doi.org/10.1109/ICCCRI.2015.10. doi:10.1109/ICCCRI.2015.10.

[13] W. Chung, P. Shih, K. Lai, K. Li, C. Lee, J. Chou, C. Hsu, and Y. Chung. *Taiwan UniCloud: A Cloud Testbed with Collaborative Cloud Services*. 2014 IEEE International Conference on Cloud Engineering, pages 107–116, Boston, Massachusetts, USA, March 10–14, 2014. https://doi.org/10.1109/IC2E.2014.28. doi:10.1109/IC2E.2014.28.

[14] *Virtual Wall - iLab-t testbeds 1.0.0 documentation,*. https://doc.ilabt.imec.be/ilabt-documentation/virtualwallfacility.html.

[15] A. Sanatinia, S. Deshpande, A. Munshi, D. Kohlbrenner, M. Yessaillian, S. Symonds, A. Chan, and G. Noubir. *Hyperdrive: A flexible cloud testbed for research and education*. 2017 IEEE International Symposium on Technologies for Homeland Security (HST), pages 1–4, Waltham, Massachusetts, USA, April 25–26, 2017. https://doi.org/10.1109/THS.2017.7943500. doi:10.1109/THS.2017.7943500.

[16] P. Abrahamsson, S. Helmer, N. Phaphoom, L. Nicolodi, N. Preda, L. Miori, M. Angriman, J. Rikkilä, X. Wang, K. Hamily, and S. Bugoloni. *Affordable and Energy-Efficient Cloud Computing Clusters: The Bolzano Raspberry Pi Cloud Cluster Experiment*. IEEE CloudCom 2013. 5th International Conference on Cloud Computing Technology and Science, pages 170–175, Bristol, United Kingdom, December 2–5, 2013. https://doi.org/10.1109/CloudCom.2013.121. doi:10.1109/CloudCom.2013.121.

[17] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee. *A Container-Based Edge Cloud PaaS Architecture Based on Raspberry Pi Clusters*. IEEE FiCloudW 2016. 4th International Conference on Future Internet of Things and Cloud Workshops, pages 117–124, Vienna, Austria, August 22–24, 2016. https://doi.org/10.1109/W-FiCloud.2016.36. doi:10.1109/W-FiCloud.2016.36.

[18] F. Tso, D. White, S. Jouet, J. Singer, and D. Pezaros. *The Glasgow Raspberry Pi Cloud: A Scale Model for Cloud Computing Infrastructures*. IEEE ICDSW 2013. 33rd International Conference on Distributed Computing Systems Workshops, pages 108–112, Philadelphia, PA, USA, July 08–11, 2013. https://doi.org/10.1109/ICDCSW.2013.25. doi:10.1109/ICDCSW.2013.25.

[19] D. Alshammari, J. Singer, and T. Storer. *Does CloudSim Accurately Model Micro Datacenters?* IEEE CLOUD 2017. 10th International Conference on Cloud Computing, pages 705–709, Honolulu, Hawaii, USA, June 25–30, 2017. https://doi.org/10.1109/CLOUD.2017.97. doi:10.1109/CLOUD.2017.97.

[20] G. Kecskemeti, W. Hajji, and F. Tso. *Modelling Low Power Compute Clusters for Cloud Simulation*. 2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), pages 39–45, St. Petersburg, Russia, March 6–8, 2017. https://doi.org/10.1109/PDP.2017.33. doi:10.1109/PDP.2017.33.

[21] D. Alshammari, J. Singer, and T. Storer. *Performance evaluation of cloud computing simulation tools*. 2018 IEEE 3rd International Conference on Cloud Computing and Big Data Analysis (ICCCBDA), pages 522–526, Chengdu, China, April 20–22, 2018. https://doi.org/10.1109/ICCCBDA.2018.8386571. doi:10.1109/ICCCBDA.2018.8386571.

[22] *IEEE Xplore Digital Library,*. https://ieeexplore.ieee.org/.

[23] N. Liu, Z. Li, J. Xu, Z. Xu, S. Lin, Q. Qiu, J. Tang, and Y. Wang. *A Hierarchical Framework of Cloud Resource Allocation and Power Management Using Deep Reinforcement Learning*. IEEE ICDCS 2017. 37th International Conference on Distributed Computing Systems, pages 372–382, Atlanta, GA, USA, June 5–8, 2017. https://doi.org/10.1109/ICDCS.2017.123. doi:10.1109/10.1109/ICDCS.2017.123.

[24] G. Portaluri, D. Adami, A. Gabbrielli, S. Giordano, and M. Pagano. *Power Consumption-Aware Virtual Machine Placement in Cloud Data Center*. IEEE Transactions on Green Communications and Networking, 1(4):541–550, 2017. https://doi.org/10.1109/TGCN.2017.2725488. doi:10.1109/TGCN.2017.2725488.

[25] P. Maenhaut, H. Moens, B. Volckaert, V. Ongenae, and F. De Turck. *A dynamic Tenant-Defined Storage system for efficient resource management in cloud applications*. Journal of Network and Computer Applications, 93:182–196, 2017. https://doi.org/10.1016/j.jnca.2017.05.014. doi:10.1016/j.jnca.2017.05.014.

[26] M. Khoshkholghi, M. Derahman, A. Abdullah, S. Subramaniam, and M. Othman. *Energy-Efficient Algorithms for Dynamic Virtual Machine Consolidation in Cloud Data Centers*. IEEE Access, 5:10709–10722, 2017. https://doi.org/10.1109/ACCESS.2017.2711043. doi:10.1109/ACCESS.2017.2711043.

[27] J. Bi, H. Yuan, W. Tan, M. Zhou, Y. Fan, J. Zhang, and J. Li. *Application-Aware Dynamic Fine-Grained Resource Provisioning in a Virtualized Cloud Data Center*. IEEE Transactions on Automation Science and Engineering, 14(2):1172–1184, 2017. https://doi.org/10.1109/TASE.2015.2503325. doi:10.1109/TASE.2015.2503325.

[28] S. Rampersaud and D. Grosu. *Sharing-Aware Online Algorithms for Virtual Machine Packing in Cloud Environments*. IEEE CLOUD 2015. 8th International Conference on Cloud Computing, pages 718–725, New York, USA, June 27–July 2, 2015. https://doi.org/10.1109/CLOUD.2015.100. doi:10.1109/CLOUD.2015.100.

[29] S. Rampersaud and D. Grosu. *Sharing-Aware Online Virtual Machine Packing in Heterogeneous Resource Clouds*. IEEE Transactions on Parallel and Distributed Systems, 28(7):2046–2059, 2017. https://doi.org/10.1109/TPDS.2016.2641937. doi:10.1109/TPDS.2016.2641937.

[30] A. Atrey, H. Moens, G. Van Seghbroeck, B. Volckaert, and F. De Turck. *Design and Evaluation of Automatic Workflow Scaling Algorithms for Multi-tenant SaaS*. CLOSER 2016. 6th International Conference on Cloud Computing and Services Science, pages 221–229, Rome, Italy, April 23–25, 2016. https://doi.org/10.5220/0005812002210229. doi:10.5220/0005812002210229.

[31] K. Metwally, A. Jarray, and A. Karmouch. *MILP-Based Approach for Efficient Cloud IaaS Resource Allocation*. IEEE CLOUD 2015. 8th International Conference on Cloud Computing, pages 1058–1062, New York, USA, June 27–July 2, 2015. https://doi.org/10.1109/CLOUD.2015.152. doi:10.1109/CLOUD.2015.152.

[32] A. Aral and T. Ovatman. *Subgraph Matching for Resource Allocation in the Federated Cloud Environment*. IEEE CLOUD 2015. 8th International Conference on Cloud Computing, pages 1033–1036, New York,

USA, June 27–July 2, 2015. https://doi.org/10.1109/CLOUD.2015.145. doi:10.1109/CLOUD.2015.145.

[33] N. Hieu, M. Di Francesco, and A. Ylä-Jääski. *Virtual Machine Consolidation with Usage Prediction for Energy-Efficient Cloud Data Centers*. IEEE CLOUD 2015. 8th International Conference on Cloud Computing, pages 750–757, New York, USA, June 27–July 2, 2015. https://doi.org/10.1109/CLOUD.2015.104. doi:10.1109/CLOUD.2015.104.

[34] Z. Zhang, C. Hsu, and M. Chang. *CoolCloud: A Practical Dynamic Virtual Machine Placement Framework for Energy Aware Data Centers*. IEEE CLOUD 2015. 8th International Conference on Cloud Computing, pages 758–765, New York, USA, June 27–July 2, 2015. https://doi.org/10.1109/CLOUD.2015.105. doi:10.1109/CLOUD.2015.105.

[35] P. Maenhaut, B. Volckaert, V. Ongenae, and F. De Turck. *Demo abstract: RPiaaS: A raspberry pi testbed for validation of cloud resource management strategies*. IEEE INFOCOM WKSHPS 2017. 8th International Conference on Computer Communications Workshops, pages 946–947, Atlanta, GA, USA, May 1–4, 2017. https://doi.org/10.1109/INFCOMW.2017.8116503. doi:10.1109/INFCOMW.2017.8116503.

[36] *GitHub - RPiaaS,*. https://github.com/IBCNServices/RPiaaS.

[37] *Docker docs - Use the OverlayFS storage driver,*. https://docs.docker.com/engine/userguide/storagedriver/overlayfs-driver/.

[38] *Docker docs - Use the Device Mapper storage driver,*. https://docs.docker.com/engine/userguide/storagedriver/device-mapper-driver/.

[39] *Docker docs - Select a storage driver,*. https://docs.docker.com/engine/userguide/storagedriver/selectadriver/.

[40] *akopytov/sysbench: Scriptable database and system performance benchmark,*. https://github.com/akopytov/sysbench.

[41] *raas/mbw: Memory Bandwidth Benchmark,*. https://github.com/raas/mbw.

[42] *Amazon Elastic Block Store,*. https://aws.amazon.com/ebs/.

[43] *AWS Service Limits - Amazon Web Service,*. https://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html.

[44] P. Maenhaut, H. Moens, B. Volckaert, V. Ongenae, and F. De Turck. *A Simulation Tool for Evaluating the Constraint-Based Allocation of Storage Resources for Multi-Tenant Cloud Applications*. IEEE/IFIP NOMS 2016. Network Operations and Management Symposium, pages 1017–1018, Istanbul,

Turkey, April 25–29, 2016. https://doi.org/10.1109/NOMS.2016.7502951. doi:10.1109/NOMS.2016.7502951.

5

Resource Management in a Containerized Cloud: Status and Challenges

Recently, new deployment models such as fog and edge computing have gained maturity, bringing the cloud closer to the end user. The RPiaaS testbed presented in Chapter 4 can in fact be seen as an embedded edge or fog testbed. Furthermore, it was designed using a microservice architecture, where experiments and all required management services are running inside containers. Containers are gaining popularity as virtualization technology, due to the minimal overhead compared to traditional virtual machines and the offered portability. Traditional resource management strategies however are typically designed for the allocation and migration of virtual machines in a traditional cloud environment, so the question arises how these strategies can be adapted for the management of a containerized cloud. In this chapter, we provide an overview of the current state of the art regarding resource management within the broad sense of cloud computing, complementary to existing surveys in literature. We investigate how research is adapting to the recent evolutions within the cloud, being the adoption of container technology and the rise of fog and edge computing, and identify several challenges and possible opportunities for future research.

⋆ ⋆ ⋆

P.-J. Maenhaut, B. Volckaert, V. Ongenae and F. De Turck

Submitted to Springer Journal of Network and Systems Management.

5.1 Introduction

Over recent years, cloud computing has become an important aspect of our daily life. Nowadays, the cloud is often used for storing personal and/or professional data, such as documents, pictures and even backups. It also facilitates online collaboration, as the stored documents are accessible from any device and location, as long as an internet connection is available. But apart from online data storage, many novel applications have been developed on top of the cloud. These applications are often available as online applications, which can be accessed through a custom app or directly through the web browser. The term cloud computing has a broad meaning: it not only refers to the online applications and services hosted in the cloud, but also to the underlying frameworks and technologies that enable them.

One of the key enablers of cloud computing is the so-called elasticity, which allows cloud applications to dynamically adjust the amount of provisioned resources based on the current and/or expected future demand. Given the increasing popularity and amount of cloud applications, efficient resource management is of great importance, as it can not only result in higher scalability of the cloud environment, but also in lower operational costs. Efficient resource management can be beneficial for multiple actors. For the cloud infrastructure provider, it aids to minimize the power consumption, as unprovisioned hardware can be put in standby or even turned off. This also helps to reduce the energy footprint of the data center, which is one of the main goals of green cloud computing. For the consumer, efficient resource management helps to minimize the rental costs. When multiple consumers share the same physical hardware, the provider can offer its instances at a lower price.

As a result, resource management within cloud environments has been a major research topic since the introduction of cloud computing. A typical research objective is to minimize the amount of provisioned computational resources, in order to lower the operational costs, without violating the objectives described in so-called Service Level Agreements (SLAs). An example of this is Virtual Machine (VM) packing, which aims to consolidate virtual servers onto a minimal number of physical machines. Multiple resource allocation strategies have been developed by both academics and industry, often resulting in open source and/or commercial products. A popular example is Swift [1], a highly scalable cloud storage system, which is integrated into the OpenStack cloud stack, and OpenStack [2]

itself, an open-source framework for building a private cloud environment, which has multiple resource management functions built in.

A recent trend within cloud computing is the uprise of new types of clouds, such as edge and fog computing [3, 4]. The cloud is no longer limited to the centrally hosted data center, accessible from a laptop or desktop computer with a broadband internet connection, but lightweight devices such as mobile phones and Internet of Things (IoT) devices can also benefit from the near infinite amount of resources offered by the cloud. These devices can offload computational intensive tasks to a centrally hosted cloud, and by installing dedicated hardware at the edge of the network, close to the end user devices, the latency can be reduced, as well as the consumed network bandwidth towards the public cloud.

When it comes to virtualization, a key enabler for cloud computing, container technology has recently gained popularity, thanks to the minimal overhead compared to traditional VMs, and the great portability it offers [5, 6]. Not only can containers be easily migrated between different cloud environments, they can also be moved to the edge of the cloud, and for example be deployed onto less powerful ARM hardware located within IoT devices. This facilitates offloading within a cloud environment, as developers can now easily reconfigure which services are running locally or in the cloud, without paying the heavy penalty of traditional VM migrations.

In this chapter, we investigate how recent research related to cloud resource management is adapting to support these new technologies. This survey is complementary to existing surveys in literature, as most previously published surveys only handle resource management within traditional cloud environments [5–15] or only consider virtual machines as virtualization technology [3, 9–12, 14, 15]. Furthermore, as illustrated in Section 5.3.1, a majority of surveys focus on a specific aspect of resource management such as resource scheduling or dynamic spot pricing. This chapter covers the broad range of resource management, and is not limited to a single cloud type or virtualization technology. The remainder of this chapter is structured as follows. In the next section, we introduce all relevant concepts and technologies related to resource management in containerized cloud environments. In Section 5.3 we provide an overview of recent research related to resource management, and identify several challenges and opportunities in Section 5.4. We finish this chapter by presenting our conclusions in Section 5.5.

5.2 Overview

In this section, we provide an overview of all relevant concepts and technologies related to resource management in containerized cloud environments. First, we start with a brief summary of cloud, edge and fog computing. Next, we elaborate on virtualization, as this is one of the key enablers for cloud computing, and intro-

duce containerization (OS-level virtualization) as an alternative for VMs. Finally, we describe all main functions related to the management of cloud resources.

5.2.1 Cloud, Edge and Fog Computing

5.2.1.1 Traditional Cloud Computing

With cloud computing, different deployment models can be distinguished. The National Institute of Standards and Technology (NIST) defined four main deployment models [16]:

- In a **Private Cloud**, the cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers.
- A **Community Cloud** is similar to a private cloud, but the infrastructure is provisioned for exclusive use by a specific community of consumers.
- A **Public Cloud** is provisioned for open use by the general public, and is usually fully accessible over the public internet.
- A **Hybrid Cloud** is a composition of two or more distinct cloud infrastructures.

Applications can either be deployed within a single cloud, or using multiple clouds. To avoid vendor lock-in, one can for example choose to deploy its application using different public cloud platforms offered by different providers. Another example is a hybrid cloud which consists of a private cloud and a public cloud. In this model, the main application is typically deployed on the private cloud, and the public cloud is used for executing computational intensive tasks, or to support the private cloud when the demand for computing capacity spikes. The latter case is often referred to as **Cloud Bursting**.

Within the context of public cloud computing, three main service models can be distinguished, as defined by the NIST [16]:

- **Infrastructure as a Service (IaaS)**: in this model, the provider offers (typically virtual) computational resources to the consumer, for example as Virtual Machines. The consumer does not manage or control the underlying cloud infrastructure, but does have control over operating systems, storage, deployed applications and possibly limited control over the network (e.g. for defining firewall rules).
- **Platform as a Service (PaaS)**: in this model, the provider offers a set of languages, libraries, services, and tools to the consumer to deploy its applications. In contrast to IaaS, the consumer typically has no control over the operating system and storage, but can control the deployed applications and applicable configuration settings for the hosting environment.

- **Software as a Service (SaaS)**: in this model, applications running on a cloud infrastructure are offered to the consumer. These applications are typically deployed on top of an IaaS or PaaS environment. The consumer has no control over the underlying infrastructure and software, except for limited application specific customization.

In the above definitions, a provider offers services to a consumer. The term provider however has a broad sense, and Armbrust et al. defined three main actors within Cloud Computing [17]:

- The **Cloud Provider** or infrastructure provider manages a physical data center, and offers (typically virtualized) resources to the cloud users, either as IaaS or PaaS instances.

- The **Cloud User** rents virtual resources from the cloud provider to deploy its cloud applications, which he provides (typically as SaaS) to the end users.

- The **End User** uses the SaaS applications provided by the cloud user. The end user generates workloads that are processed using cloud resources.

5.2.1.2 Fog and Edge Computing

Access to the cloud is not limited to traditional devices such as servers, desktops and laptops. With **Mobile Cloud Computing** (also referred to as Mobile Edge Computing), mobile devices collaborate with a cloud environment for offloading of tasks. As these devices are usually connected using a less reliable connection with limited bandwidth, and are often battery powered, some tasks will be executed directly on the device, whereas other tasks will be transferred to the cloud. Executing tasks on the device can reduce the network congestion and lower the latency, but will increase the energy consumption of the device. Offloading tasks to the cloud on the contrary can decrease the energy consumption, and can also decrease the execution time for computational intensive tasks. Mobile edge computing can in fact be seen as a special case of **Edge Computing**. Edge Computing and Fog Computing are related terms, which aim to bring the cloud closer to the end user [18, 19].

- With **Fog Computing**, dedicated fog nodes (e.g. gateways, devices, computers or micro data centers) are deployed at the edge of the network, typically inside the LAN of the edge devices. These nodes gather data from the edge devices, and will often perform some (pre-)processing of the gathered data, instead of transferring all tasks to the central cloud environment. Doing so can help to reduce the network congestion towards the central cloud, and can also help to reduce the response time.

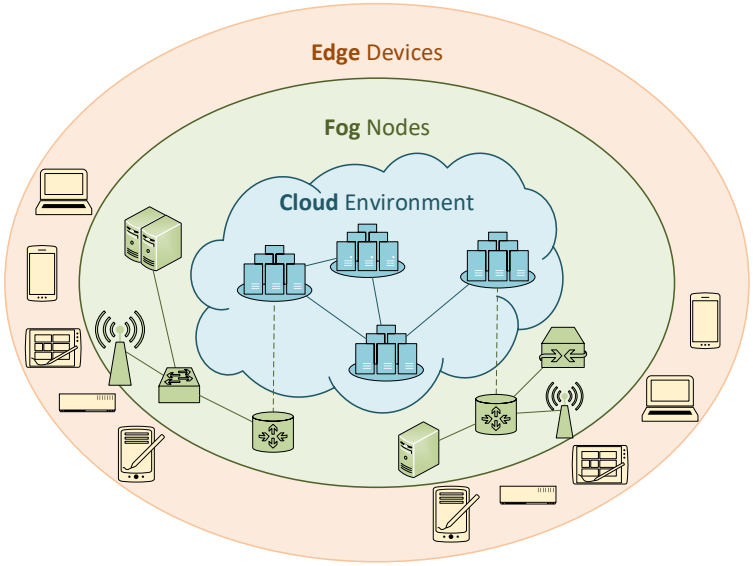


Figure 5.1: Relationship between traditional cloud computing, fog computing and edge computing. Fog nodes bring the cloud closer to the end user, and the edge devices can offload computational intensive tasks to the central cloud.

- With **Edge Computing**, the edge devices that are collecting the data do some processing and analysis themselves. This however will increase the energy consumption of the devices, but when the devices are connected using a limited connection, this can help to reduce the latency and network congestion inside the local network. With edge computing, a typical research question is to determine which tasks should be executed on the local device, and which tasks should be offloaded to the fog or cloud environment.

An example topology for fog and edge computing is illustrated in Figure 5.1, as well as the relationships between the different environment. Both edge and fog computing typically aim to reduce the latency and the load on the cloud, and are often used in the context of IoT, in which large amounts of data are collected for analysis and processing [19–23].

5.2.2 Virtualization

Cloud computing is mainly built on top of virtualization, as cloud users typically rent virtual resources from the cloud providers. A typical form of virtualization is the use of VMs, in which multiple VMs are emulated on top of a so-called hypervisor. A VM runs the full software stack, meaning that an Operating System (OS) is deployed on top of the VM, and the required software is installed on top of this OS. When deploying a VM, the cloud user can either start from scratch and create a new virtual machine, install the preferred OS and all required binaries and libraries, or a pre-configured template can be used for deploying a new VM which already contains the operating system and a typical software stack (e.g. a web server). In the latter case, the cloud user only needs to customize the packages, and deploy its application on top. Because the full OS is installed on the virtual disk of the VM, this virtual disk is easily a few gigabytes in size. This makes migration of VMs challenging, especially when moving the VM to a different physical location, as the whole virtual disk needs to transferred. When migrating a VM, the machine can first be turned off, which facilitates the migration process as there are almost no risks such as losing state or consistency, but there will be a noticeable downtime. Most hypervisors however also support the migration of running virtual machines between different physical machines, without disconnecting the client or application, referred to as live migration. Live migrations will also include some downtime of the VM, but when this is not noticeable by the end users, the migration is called a seamless live migration.

Recently, container technologies have emerged as a more lightweight alternative for VMs [24–27]. The major difference with VMs is that a container typically has no operating system installed, but instead all containers deployed on a single machine are running directly on the operating system kernel (OS-level virtualization). As a result, containers are much smaller in size, and a typical container file

is a few hundreds megabytes, whereas a similar VM with the same applications installed will typically be a few gigabytes. To launch a new container, the user can either start from a base image (e.g. an Ubuntu-flavored base image or an official NodeJS base image) and install and configure all required software packages, or he can create a new container based on a pre-configured image that is pulled from a central repository, with most of the required software already installed and configured.

OS-level virtualization (also referred to as containerization) has existed for some time, with LXC [28] being one of the first popular container engines. LXC was initially released in 2008, but in 2013 Docker [29] was released as a successor for LXC, and quickly became one of the most popular container engines. Initial releases of Docker were still using LXC as default execution environment, but in later releases Docker replaced LXC with its own library. To facilitate the deployment, Docker containers can be published to Docker Hub [30], a publicly available, centrally hosted repository for storing fully configured container images, or organizations can configure their own private Docker image repository. Docker however only offers tools for deploying and managing containers on top of a single physical machine. For the management and deployment of containerized applications over a cluster of Docker servers, a container orchestration system such as Kubernetes [31] is required. Docker initially offered its own orchestration tools, called Docker Swarm mode, which offered limited functionality for managing container clusters [32]. In 2017, the team behind Docker however announced native Kubernetes support, and recommended Kubernetes as orchestration tool for enterprise environments [33].

As containers are lightweight, they are often used for deployment of applications that are designed using a Service-Oriented Architecture (SOA). With SOA, an application is decomposed into several collaborating services, and every service can be deployed into a separate container. This allows for fine-grained scalability, as each service can be scaled up or down individually, instead of scaling the whole application as a whole. In multi-cloud environments, the use of lightweight containers also offers multiple opportunities for achieving high scalability and cost-efficient deployments, thanks to the offered portability [34].

A combination of both virtualization technologies is possible, for example when deploying a container engine on top of VMs hosted on a cloud environment [26, 35]. In this scenario, the application is deployed inside a container, and the container runtime is running on top of the guest OS of the VM. VMs typically offer a higher level of isolation and security, while containers have a lower virtualization overhead, and a hybrid model could combine the advantages of both technologies. Figure 5.2 provides an overview of the typical models for deployment of an application or service within a virtualized environment. For VM-based deployments, the hypervisor can either run directly on the hardware (bare-metal

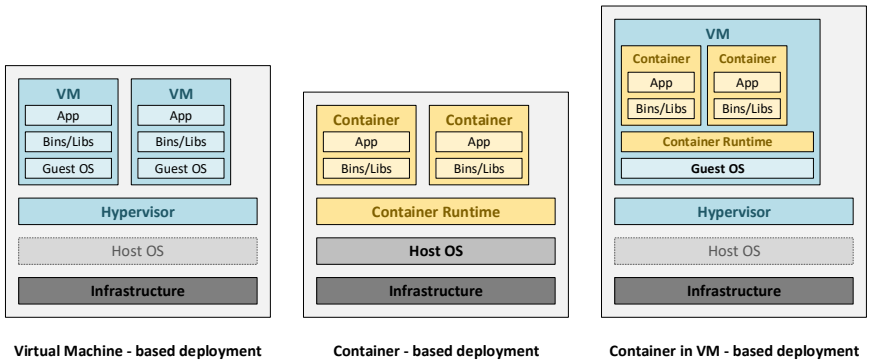


Figure 5.2: Comparison between the different models for deployment within a virtualized environment. The application or service can be either deployed inside a VM, a container, or a container hosted in a VM.

virtualization), or on top of the host OS.

5.2.3 Resource Management

Resource management is a broad term, which refers to all required functionalities related to the allocation, provisioning and pricing of (virtual) resources. For the deployment of cloud applications, the minimal required amount of resources needs to be determined, and in an elastic cloud environment the allocated amount of resources can change dynamically based on the current demand. Furthermore, by monitoring and profiling the applications or the resources, an estimate can be made regarding the future demand. In a public cloud environment, the cloud provider needs to determine the price billed to the cloud users based on the actual resource usage, and the cloud user can charge the end users for using the SaaS applications.

5.2.3.1 Management Objectives

With public cloud computing, cloud providers need to satisfy the SLAs agreed upon with the cloud users regarding the provisioning of virtual infrastructure. Such a SLA can consist of multiple constraints (which must always be satisfied) and objectives (which should be satisfied). A typical management objective is a specified monthly uptime percentage for the virtual instances, or a maximum allowed response time for the cloud environment. The provider can choose to offer its infrastructure to all cloud users using a single SLA, or can pursue service differentiation by offering different service levels to the customers. The provider could also choose to apply different objectives during different operational conditions, for example by guaranteeing different objectives during low load or overload.

The cloud user can also have a SLA with the end-users, consisting of objectives regarding the offered services (typically as SaaS). To comply with these objectives, the cloud user may seek to exploit the elasticity property of the cloud environment. The cloud user could for example over-provision resources in order to guarantee the objectives, or could try to minimize the operational costs but with the risk of violating the SLA with the end users.

5.2.3.2 Resource Allocation, Provisioning and Scheduling

In an optimal scenario, every cloud application would be deployed in a location close to the end users in order to minimize latency, on hardware that is powerful enough to guarantee compliance with the selected SLAs, and on a dedicated server to maximize performance isolation. This scenario however would lead to high operational costs and energy consumption, and a waste of resources as most of the time the provisioned server instances would be in an idle state. Resource allocation strategies aim to solve this issue, by packing multiple applications belonging to different customers onto the same physical hardware, while guaranteeing performance and data isolation and compliance with SLA requirements. Resource management consists of multiple tasks, with the main tasks being the allocation, provisioning and scheduling of (virtual) resources.

- **Resource Allocation** refers to the allocation (reservation) of a pool of resources (e.g. computational resources, network bandwidth and storage) for a given consumer.
- **Resource Provisioning** on the other hand is the effective provisioning of (a part of) the allocated resources in order to execute a given task. A typical example of resource provisioning is the deployment of a new virtual machine by the consumer, which uses a subset of the allocated CPU, network and storage resources.
- When executing a large batch of tasks in the cloud, **Resource Scheduling** aims to find a feasible execution order for these tasks, making optimal usage of the available resources while respecting the deadlines defined for each individual task.
- **Resource Orchestration** is a broad term, that includes both scheduling, management and provisioning of additional resources. Orchestrators typically manage complex cross-domain processes, and aim to meet the defined objectives, for example meeting the application performance goals while minimizing costs and maximizing performance.

Resource allocation, provisioning, scheduling and orchestration are closely related, and are the main building blocks for application elasticity within a cloud

environment. When allocating resources, a further distinction can be made between static and dynamic allocation.

- With a **Static Resource Allocation** strategy, the required amount of resources is determined during deployment, and the allocation of resources does not change during the lifetime of the deployed applications. Static resource allocation however can lead to under- or over-provisioning, when the amount of allocated resources is not in line with the current demand.
- With **Dynamic Resource Allocation**, the amount of allocated resources can change during execution, in order to meet the current demand. Dynamic resource allocation can lead to a higher utilization of the physical resources, and allows for server consolidation in order to reduce the operating costs.

Dynamic resource allocation is often seen as the most efficient means to allocate hardware resources in a data center [36]. However, dynamic resource allocation typically involves migration of running applications, which leads to an overhead and possible service disruptions.

5.2.3.3 Monitoring and Profiling

When allocating resources, a distinction can be made between reactivity and proactivity:

- With a **Reactive** control mechanism, the amount of allocated resources is adjusted over time in response to a detected change in demand.
- With a **Proactive** control mechanism, the amount of allocated resources is adjusted based on a predicted change in demand.

For proactive control mechanisms, a prediction of the demand is often made using historical measurements. This is typically done using **Demand Profiling**, and can happen either at the application level, when predicting the demand for individual applications, or at the infrastructure (data center) level, when predicting the global demand within the cloud environment. Apart from estimating the demand, an estimation can also be made regarding the state of the physical and virtualized resources, often referred to as **Resource Utilization Estimation**. An estimation can be made for the different types of resources, such as compute, network, storage and power resources, and these estimations serve as input for both the monitoring and scheduling processes.

By **Monitoring** the actual resource utilization, the provider can detect if the current allocation scheme fits the current demand. In an elastic cloud environment, additional resources can be provisioned on the fly if there is an overutilization of the provisioned resources (under-provisioning), and when more resources are allocated than required (over-provisioning), a certain amount of resources can be

deallocated to decrease the operational costs. Monitoring processes can also be used to determine failure of certain components. Furthermore, monitoring information can provide useful input for both demand profiling and resource utilization estimation.

5.2.3.4 Resource Pricing

Especially with public cloud computing, the cloud user or end user will be charged based on its usage of the cloud resources or cloud services. In this context, a distinction can be made between application pricing and infrastructure pricing [9].

- With **Application Pricing**, the cloud user determines the price for the services (typically offered as SaaS applications) provided to the end users.
- With **(Virtual) Infrastructure Pricing**, the cloud provider determines the price charged for the virtual resources rented to the cloud users.

For application pricing, the cloud user could either provide its application at a fixed price (e.g. a monthly incurring bill, with the price based on the number of active users) or he could charge the cloud user based on the actual usage (e.g. the total amount of bandwidth or data storage used by the consumer). With public cloud computing, cloud providers traditionally use a **Static Pricing** scheme to cover the infrastructure and operational costs of the data center. With static pricing, the cloud users are charged a fixed price based on either the number of virtual instances, the actual resource usage, or a combination of both. The cloud provider can for example charge a fixed price for each instantiated virtual machine, together with a variable price based on the amount of consumed network bandwidth and/or additional storage. Recently, **Dynamic Pricing** schemes are gaining popularity as an alternative to static pricing, mainly to increase the utilization of the data center [10, 37–40]. With dynamic pricing, the cloud provider can for example lease its resources at a lower price when the demand is low, and increase the prices as the demand increases. Another example of dynamic pricing is spot pricing, in which the cloud provider offers dynamically priced resources at a lower price, but with less guarantee of availability [10]. Dynamic pricing can also be based on an auction-based pricing model, in which multiple cloud users bid for a bundle of virtual cloud resources [38, 39]. The cloud provider will then select a set of cloud users, the winners, and needs to determine a feasible allocation over its physical hardware.

5.3 State of the Art

This section provides an overview of recent research (published between 2015 and 2018) focusing on resource management within cloud environments. We se-

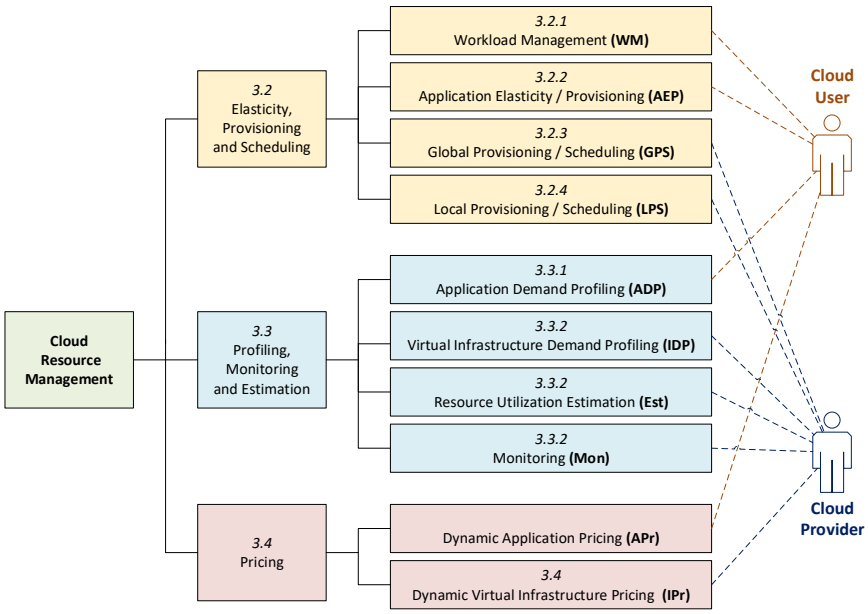


Figure 5.3: Cloud resource management taxonomy used in this chapter, based on the conceptual framework introduced by Jennings and Stadler [9]. For each functional element, the corresponding subsection is denoted in the figure.

lected this time period as this chapter extends the survey previously published by Jennings and Stadler [9], which already provides an extensive overview of research related to resource management published before 2015. We reviewed over 150 research papers from five main publishers, namely ACM, Elsevier, IEEE, Springer and Wiley. A majority of the reviewed articles were published in either ACM Transactions on Internet Technology (TOIT) [41], IEEE Transactions on Cloud Computing (TCC) [42], IEEE Transactions on Parallel and Distributed Systems (TPDS) [43], IEEE Transactions on Network and Service Management (TNSM) [44], Springer Journal of Network and Systems Management (JNSM) [45] or Wiley Journal of Software: Practice and Experience (SPE) [46].

In the remainder of this section, a brief summary of previous surveys focusing on resource management is first provided. We then categorize the research within three main areas, as illustrated in Figure 5.3. For each category, a summary of the most relevant research is provided for each resource management functional element. Table 5.1 provides a mapping from all research items (excluding surveys) to the covered resource management functional elements. As can be seen from this table, some publications can be attributed to multiple categories and/or functional elements. For these items, we selected the most relevant category and/or element, and in the remainder of this section these items are included in the corresponding subsection. For each research item, we also added attributes to denote the used cloud type, the scope and the virtual allocation entity. Table 5.2 provides an overview of all applicable attributes, and the abbreviations used in the remainder of this section.

5.3.1 Previous Surveys

Table 5.3 provides an overview of previous surveys related to resource management within cloud environments. In 2015, Jennings and Stadler published an extensive overview of resource management within the public cloud [9]. In their survey, the authors introduce a conceptual framework for cloud resource management consisting of multiple functional elements, and classify related research into eight functional areas. Furthermore, they characterize cloud provisioning schemes based on the placement approach (static, dynamic, network aware and/or energy aware) and the control architecture (centralized, hierarchical or distributed). The framework is illustrated in Figure 5.4. In this figure, we added a mapping from the different resource management functional elements to the categories used in this chapter, namely Elasticity, Profiling and Pricing.

Yousafzai et al. extended the research of Jennings and Stadler by introducing a taxonomy for categorizing cloud resource allocation schemes [14]. The introduced taxonomy is based on multiple attributes, being the optimization objective, the design approach, the target resource allocation type, the applied optimization

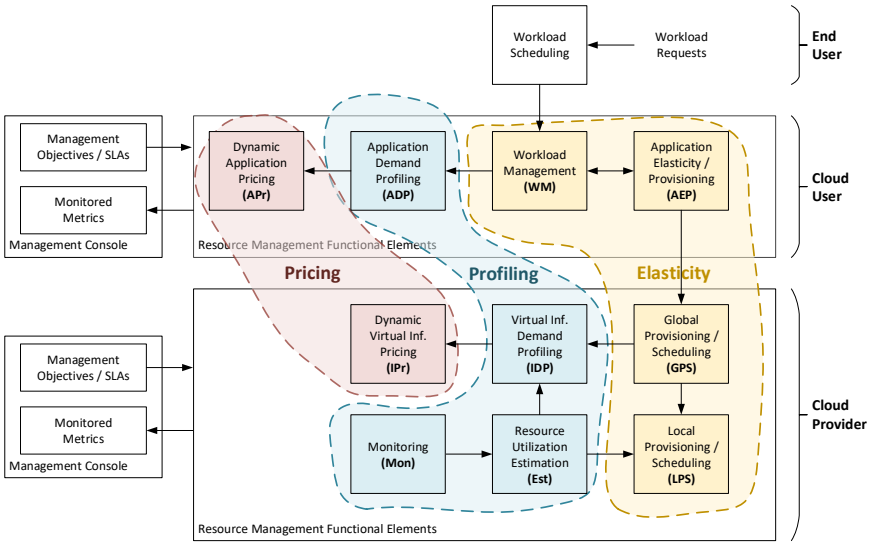


Figure 5.4: Conceptual framework for resource management in a cloud environment, as introduced by Jennings and Stadler [9]. In this figure, we added a mapping from the functional elements of the framework to the categories used in this chapter (Elasticity, Profiling and Pricing).

Table 5.1: Mapping from all research items (excluding surveys) to the resource management functional elements of Figure 5.3.
WM Workload Management, AEP Application Elasticity and Provisioning, GPS Global Provisioning and Scheduling, LPS Local Provisioning and Scheduling, ADP Application Demand Profiling, IDP (Virtual) Infrastructure Demand Profiling, Est Resource Utilization Estimation, Mon Monitoring, APr Dynamic Application Pricing, IPr Dynamic Virtual Infrastructure Pricing.

		—Elasticity—				—Profiling—				Pricing	
Publication	Year	WM	AEP	GPS	LPS	ADP	IDP	Est	Mon	APr	IPr
Aazam et al. [47]	2015			✓			✓	✓			✓
AbdelBaky et al. [48]	2015			✓							
Amannejad et al. [49]	2015		✓		✓						
Chiang et al. [50]	2015			✓							
Dabbagh et al. [51]	2015			✓			✓	✓	✓		
Dhakate et al. [52]	2015								✓		
Huang et al. [53]	2015			✓							✓
Jin et al. [54]	2015										✓
Katsalis et al. [55]	2015				✓						
Kumbhare et al. [56]	2015	✓	✓								

—Continued on next page—

Table 5.1: Mapping from all research items (excluding surveys) to the resource management functional elements of Figure 5.3 (continued).

Publication	Year	WM	AEP	GPS	LPS	ADP	IDP	Est	Mon	APr	IPr
Lee et al. [57]	2015			✓							✓
Liu et al. [58]	2015			✓					✓		
Mashayekhy et al. [38]	2015			✓							✓
Moens et al. [59]	2015		✓								
Mukherjee et al. [60]	2015				✓						
Petri et al. [61]	2015			✓							✓
Sharma et al. [62]	2015										✓
Stankovski et al. [63]	2015			✓					✓		
Wang et al. [64]	2015			✓							✓
Wuhib et al. [65]	2015			✓			✓	✓	✓		
Zhang et al. [66]	2015		✓								
Aazam et al. [67]	2016			✓							✓
Ayoubi et al. [68]	2016		✓	✓							
Choi et al. [69]	2016			✓					✓		
D.C. Rodrigues et al. [70]	2016								✓		
Dai et al. [71]	2016			✓							
Elgazzar et al. [72]	2016		✓								
Espling et al. [73]	2016			✓							
Goudarzi et al. [74]	2016			✓			✓	✓			
Huang et al. [75]	2016			✓							
Kang et al. [76]	2016	✓									
Khatua et al. [77]	2016		✓								
Mashayekhy et al. [39]	2016			✓							✓
Mishra et al. [78]	2016			✓							
Nakagawa et al. [79]	2016				✓			✓	✓		
Pantazoglou et al. [80]	2016			✓							
Righi et al. [81]	2016		✓								
Salah et al. [82]	2016		✓								
Sharma et al. [26]	2016				✓						
Wajid et al. [83]	2016			✓					✓		
Wan et al. [84]	2016			✓							✓
Wanis et al. [85]	2016									✓	✓
Wolke et al. [36]	2016			✓				✓	✓		
Wu et al. [86]	2016			✓							
Xu et al. [87]	2016	✓	✓								
Zhou et al. [88]	2016								✓		
Awada et al. [89]	2017			✓							
Awada et al. [90]	2017			✓							
Babaioff et al. [91]	2017			✓			✓				✓
Chard et al. [92]	2017		✓			✓					
Chi et al. [37]	2017			✓							✓
Dalmazo et al. [93]	2017							✓			

—Continued on next page—

Table 5.1: Mapping from all research items (excluding surveys) to the resource management functional elements of Figure 5.3 (continued).

Publication	Year	WM	AEP	GPS	LPS	ADP	IDP	Est	Mon	APr	IPr
Hai et al. [94]	2017			✓						✓	✓
Hoque et al. [95]	2017			✓							
Jin et al. [96]	2017			✓							
Khasnabish et al. [97]	2017			✓					✓		
Li et al. [98]	2017				✓						
Li et al. [99]	2017			✓							
Lloyd et al. [100]	2017					✓					
Maenhaut et al. [101]	2017		✓	✓							
Mebrek et al. [102]	2017		✓								
Mechtri et al. [103]	2017			✓							
Merzoug et al. [104]	2017			✓							
Mireslami et al. [105]	2017		✓								
Nardelli et al. [106]	2017			✓							
Nitu et al. [107]	2017			✓							
Paya et al. [108]	2017		✓								
Rankothge et al. [109]	2017		✓	✓							
Tang et al. [110]	2017			✓							✓
Xu et al. [111]	2017	✓	✓								
Yang et al. [112]	2017			✓							
Yi et al. [113]	2017			✓							✓
Yu et al. [114]	2017			✓							
Zhang et al. [115]	2017			✓	✓				✓		
Alam et al. [116]	2018		✓								
Aral et al. [117]	2018		✓	✓							
Atrey et al. [118]	2018	✓	✓			✓					
Barkat et al. [119]	2018			✓							
Balos et al. [120]	2018		✓			✓	✓	✓			
Barrameda et al. [121]	2018	✓	✓			✓					
Borjigin et al. [122]	2018			✓							✓
Bouet et al. [123]	2018	✓	✓								
Cheng et al. [124]	2018			✓							
Diaz-Montes et al. [125]	2018	✓	✓	✓							
Gill et al. [126]	2018	✓									
Guo et al. [127]	2018	✓	✓			✓					
Guo et al. [128]	2018	✓	✓								
Guo et al. [129]	2018			✓							
Hauser et al. [130]	2018			✓			✓	✓	✓		
Heidari et al. [131]	2018	✓									
Jia et al. [132]	2018			✓							
Jia et al. [133]	2018			✓			✓	✓			
Khabbaz et al. [134]	2018	✓		✓							
Lahmann et al. [135]	2018				✓						

—Continued on next page—

Table 5.1: Mapping from all research items (excluding surveys) to the resource management functional elements of Figure 5.3 (continued).

Publication	Year	WM	AEP	GPS	LPS	ADP	IDP	Est	Mon	APr	IPr
Lin et al. [136]	2018		✓	✓							
Mikavica et al. [40]	2018										✓
Nawrocki et al. [137]	2018	✓	✓								
Prakash et al. [35]	2018				✓			✓	✓		
Prats et al. [138]	2018	✓				✓	✓		✓		
Rahimi et al. [139]	2018	✓	✓			✓					
Sahni et al. [140]	2018	✓	✓								
Santos et al. [21]	2018		✓	✓							
Scheuner et al. [141]	2018		✓			✓					
Simonis [142]	2018	✓	✓								
S. Sofia et al. [143]	2018	✓	✓	✓							
Takahashi et al. [144]	2018	✓	✓								
Tesfatsion et al. [27]	2018				✓				✓		
Trihinas et al. [145]	2018								✓		
Wang et al. [146]	2018			✓							
Wei et al. [147]	2018			✓				✓			
Xie et al. [148]	2018	✓		✓							
Yao et al. [23]	2018	✓	✓								
Zhang et al. [149]	2018	✓	✓								
Zhang et al. [150]	2018			✓							✓

method, the utility function, the processing mode, and the target instances. Poullie et al. also focus on the allocation of resources, and present an overview of multi-resource allocation schemes for data centers [13].

Other surveys are mainly focusing on scheduling and orchestration [5–8, 12, 15]. Bittencourt et al. for example introduce a taxonomy for scheduling in traditional cloud environments [7]. Masdari et al. also investigate the topic of scheduling, but their main focus is on scheduling schemes based on particle swarm optimization [12]. Herrera and Botero focus on Network Functions Virtualization (NFV), and present an overview of allocation and scheduling schemes for virtual network functions [131]. Rodriguez et al. recently published an extensive overview of orchestration systems specific for container-based clusters [6]. Similarly, Pahl et al. provide an overview of recent research focusing on the orchestration of containers [5].

When it comes to resource pricing, Kumar et al. provide an overview of dynamic (spot) pricing within traditional clouds [10]. The authors categorize different spot pricing models in three main categories, namely economics based models (auction-based or game theory based), statistics based models and optimization

Table 5.2: Overview of the attributes used in this section.

Cloud Type	Traditional Cloud (TC)	The item applies to traditional cloud types. These can be either public, private, community clouds or a combination (hybrid clouds).
	Fog Computing (FC)	The item applies to Fog Computing.
	Edge Computing (EC)	The item applies to Edge Computing, which includes Mobile Edge Computing.
Scope	Single Cloud (SC)	The research focuses on resource management within a single cloud environment.
	Multi-Cloud (MC)	The research handles resource management in a multi-cloud environment.
Entity	Virtual Machine (VM)	Virtual machines are used as virtualization technology.
	Container (CT)	Containers are used as virtualization technology.

based models.

Recently, Mouradian et al. published an extensive survey on fog computing [4]. In their survey, the authors provide some comments regarding resource allocation, scheduling and pricing in the context of fog and edge computing. Their survey however is not limited to resource management, but instead aims to provide a general overview of all aspects of fog computing. The authors for example also discuss several possible architectures within fog computing.

As can be seen from this overview, previously published surveys either focus on a specific aspect of resource management, or a specific cloud type. Most surveys cover resource management within traditional cloud environments, and do not yet consider containers as an alternative for VMs. In this chapter however, our goal is to cover the broad range of resource management, and we also do not limit ourselves to a single cloud type or virtualization technology.

5.3.2 Resource Elasticity

Table 5.4 provides an overview of recent work with a main focus on the allocation, provisioning and scheduling of cloud resources and the scheduling and management of user workloads. In this table, we categorized the research items based on the attributes of Table 5.2.

Table 5.3: Overview of previous surveys focusing on resource management within cloud environments.

		Function			Cloud Type			Entity	
Publication	Year	Elasticity	Profiling	Pricing	Traditional	Fog	Edge	VM	Container
Jennings & Stadler [9]	2015	✓	✓	✓	✓			✓	
Mann [11]	2015	✓	✓		✓			✓	
Yi et al. [3]	2015	✓		✓		✓	✓	✓	
Zhan et al. [15]	2015	✓	✓		✓			✓	
Herrera & Botero [8]	2016	✓			✓			✓	✓
Masdari et al. [12]	2017	✓			✓			✓	
Yousafzai et al. [14]	2017	✓			✓			✓	
Bittencourt et al. [7]	2018	✓			✓			✓	✓
Kumar et al. [10]	2018	✓		✓	✓			✓	
Mouradian et al. [4]	2018	✓		✓		✓	✓	✓	✓
Pahl et al. [5]	2018	✓			✓				✓
Poullie et al. [13]	2018	✓			✓			✓	✓
Rodriguez & Buyya [6]	2018	✓	✓		✓				✓

Table 5.4: Overview of recent research with a main focus on resource elasticity (allocation and provisioning, scheduling and/or workload management).
WM Workload Management, AEP Application Elasticity and Provisioning, GPS Global Provisioning and Scheduling, LPS Local Provisioning and Scheduling, TC Traditional Cloud, FC Fog Computing, EC Edge Computing, SC Single Cloud, MC Multi-Cloud, VM Virtual Machine, CT Container.

		Function				Cloud Type			Scope		Entity	
Publication	Year	WM	AEP	GPS	LPS	TC	FC	EC	SC	MC	VM	CT
AbdelBaky et al. [48]	2015			✓		✓				✓		
Amannejad et al. [49]	2015		✓		✓	✓	✓		✓		✓	
Chiang et al. [50]	2015			✓		✓			✓		✓	
Katsalis et al. [55]	2015				✓	✓			✓		✓	
Kumbhare et al. [56]	2015	✓	✓			✓			✓		✓	
Liu et al. [58]	2015			✓		✓			✓		✓	
Moens et al. [59]	2015		✓			✓			✓		✓	
Mukherjee et al. [60]	2015				✓	✓			✓		✓	
Stankovski et al. [63]	2015			✓		✓				✓		✓
Wuhib et al. [65]	2015			✓		✓			✓			
Zhang et al. [66]	2015		✓			✓				✓		
Ayoubi et al. [68]	2016		✓	✓		✓			✓		✓	

—Continued on next page—

Table 5.4: Overview of recent research with a main focus on resource elasticity (continued).

Publication	Year	WM	AEP	GPS	LPS	TC	FC	EC	SC	MC	VM	CT
Choi et al. [69]	2016			✓		✓			✓			✓
Dai et al. [71]	2016			✓		✓			✓		✓	
Elgazzar et al. [72]	2016		✓			✓		✓	✓		✓	
Espling et al. [73]	2016			✓		✓			✓		✓	
Goudarzi et al. [74]	2016			✓		✓			✓		✓	
Huang et al. [75]	2016			✓		✓			✓		✓	
Kang et al. [76]	2016	✓				✓			✓			✓
Khatua et al. [77]	2016		✓			✓			✓		✓	
Mishra et al. [78]	2016			✓		✓			✓		✓	
Nakagawa et al. [79]	2016				✓	✓			✓			✓
Pantazoglou et al. [80]	2016			✓		✓			✓		✓	
Righi et al. [81]	2016		✓			✓			✓		✓	
Salah et al. [82]	2016		✓			✓			✓		✓	
Sharma et al. [26]	2016				✓	✓			✓		✓	✓
Wajid et al. [83]	2016			✓		✓				✓	✓	
Wolke et al. [36]	2016			✓		✓			✓		✓	
Wu et al. [86]	2016			✓		✓			✓		✓	
Xu et al. [87]	2016	✓	✓			✓				✓	✓	
Awada et al. [89]	2017			✓		✓				✓		✓
Awada et al. [90]	2017			✓		✓			✓			✓
Hoque et al. [95]	2017			✓			✓		✓			✓
Jin et al. [96]	2017			✓		✓			✓		✓	
Khasnabish et al. [97]	2017			✓		✓			✓			
Li et al. [98]	2017				✓	✓			✓		✓	
Li et al. [99]	2017			✓		✓			✓		✓	
Maenhaut et al. [101]	2017		✓	✓		✓			✓			
Mebrek et al. [102]	2017		✓			✓	✓	✓		✓		
Mechtri et al. [103]	2017			✓		✓				✓	✓	
Merzoug et al. [104]	2017			✓		✓			✓		✓	
Mireslami et al. [105]	2017		✓			✓			✓		✓	
Nardelli et al. [106]	2017			✓		✓			✓		✓	✓
Nitu et al. [107]	2017			✓		✓			✓		✓	✓
Paya et al. [108]	2017		✓			✓			✓		✓	
Rankothge et al. [109]	2017		✓	✓		✓			✓		✓	
Xu et al. [111]	2017	✓	✓			✓			✓			
Yang et al. [112]	2017			✓		✓			✓		✓	
Yu et al. [114]	2017			✓		✓			✓			
Zhang et al. [115]	2017			✓	✓	✓			✓		✓	
Alam et al. [116]	2018		✓			✓	✓	✓		✓		✓
Aral et al. [117]	2018		✓	✓		✓	✓	✓		✓		
Atrey et al. [118]	2018	✓	✓			✓			✓		✓	
Barkat et al. [119]	2018			✓		✓				✓	✓	

—Continued on next page—

Table 5.4: Overview of recent research with a main focus on resource elasticity (continued).

Publication	Year	WM	AEP	GPS	LPS	TC	FC	EC	SC	MC	VM	CT
Barrameda et al. [121]	2018	✓	✓			✓		✓	✓		✓	
Bouet et al. [123]	2018	✓	✓			✓	✓	✓		✓		
Cheng et al. [124]	2018			✓		✓			✓		✓	
Diaz-Montes et al. [125]	2018	✓	✓	✓		✓				✓	✓	
Gill et al. [126]	2018	✓				✓			✓		✓	
Guo et al. [127]	2018	✓	✓			✓				✓	✓	
Guo et al. [128]	2018	✓	✓			✓				✓	✓	
Guo et al. [129]	2018			✓		✓				✓	✓	
Heidari et al. [131]	2018	✓				✓			✓		✓	
Jia et al. [132]	2018			✓			✓			✓		
Jia et al. [133]	2018			✓		✓			✓		✓	
Khabbaz et al. [134]	2018	✓		✓		✓			✓		✓	
Lahmann et al. [135]	2018				✓	✓			✓		✓	✓
Lin et al. [136]	2018		✓	✓		✓	✓	✓	✓			
Nawrocki et al. [137]	2018	✓	✓			✓		✓	✓			
Prakash et al. [35]	2018				✓	✓			✓		✓	✓
Rahimi et al. [139]	2018	✓	✓			✓	✓	✓		✓		
Sahni et al. [140]	2018	✓	✓			✓			✓		✓	
Santos et al. [21]	2018		✓	✓		✓	✓	✓		✓		
Simonis [142]	2018	✓	✓			✓				✓		✓
S. Sofia et al. [143]	2018	✓	✓	✓		✓			✓		✓	
Takahashi et al. [144]	2018	✓	✓			✓				✓	✓	✓
Tesfatsion et al. [27]	2018				✓	✓			✓		✓	✓
Wang et al. [146]	2018			✓		✓			✓			
Wei et al. [147]	2018			✓		✓			✓		✓	
Xie et al. [148]	2018	✓		✓		✓			✓			
Yao et al. [23]	2018	✓	✓				✓	✓	✓		✓	
Zhang et al. [149]	2018	✓	✓			✓		✓	✓			

5.3.2.1 Workload Management

The scheduling of workloads within a cloud environment differs from scheduling on traditional distributed systems, due to the on-demand resource provisioning and the pay-as-you-go pricing model which is often used by infrastructure providers [140]. The workloads being scheduled are often bound by **multiple constraints**, such as strict deadlines for individual tasks [126, 128, 134, 140], while also considering task dependencies [124]. Sahni and Vidyarthi propose a dynamic cost-effective deadline-constrained heuristic algorithm for scheduling of scientific workflows in a public cloud environment [140]. The proposed algorithm aims to minimize the costs, while taking into account the VM performance

variability and instance acquisition delay to identify a just-in-time schedule for a deadline-constrained workflow. Guo et al. also introduce a strategy for scheduling of deadline-constrained scientific workflows, but within multi-cloud environments [128]. The presented strategy aims to minimize the execution cost of the workflow, while meeting the defined deadline. Khabbaz et al. propose a deadline-aware scheduling scheme [134], but their focus is on improving the data center's Quality of Service (QoS) performance, by considering the request blocking probability and the data center's response time. Sathya Sofia and GaneshKumar on the other hand introduce a multi-objective task scheduling strategy based on a non-dominated sorting genetic algorithm [143]. The authors use a neural network for predicting the required amount of VM resources, based on the characteristics of the tasks and the resource features. Cheng et al. present a system for resource provisioning and scheduling with task dependencies, based on deep reinforcement learning [124]. The proposed solution also invokes a deep Q-learning-based two-stage resource provisioning and task scheduling processor, for the automatic generation of long-term decisions. Gill et al. argue that few existing resource scheduling algorithms consider cost and execution time constraints [126]. As a result, the authors present a novel strategy for the scheduling of workloads on the available cloud resources, based on Particle Swarm Optimization. According to Kumbhare et al., traditional stream processing systems often use simple scaling techniques with elastic cloud resources to handle variable data rates, which can have a significant impact on the application QoS [56]. To tackle this issue, the authors introduce the concept of dynamic dataflows for the scheduling of high-velocity data streams with low latency in the cloud. These dataflows use alternate tasks as additional control over the dataflow's cost and QoS. Xu et al. note that inside data centers, there exist a vast amount of delay-tolerant jobs, such as background and maintenance jobs [111]. As a result, the authors propose a scheme for the provisioning of both delay sensitive and delay-tolerant jobs, that aims to minimize the total operational costs, while still guaranteeing the required QoS for the delay sensitive jobs, and achieving a desirable delay performance for the delay-tolerant jobs.

Big-data computing applications can also benefit from the elasticity of cloud environments [131, 142, 148]. Such applications typically demand concurrent data transfers among the computing nodes, and it is important to determine an optimal transfer schedule in order to achieve a maximum throughput. Xie and Jia however claim that some existing methods cannot achieve this, as they often ignore link bandwidths and the diversity of data replicas and paths [148]. As a result, the authors propose a max-throughput data transfer scheduling approach that aims to minimize the data retrieval time. Large amounts of data generated by internet and enterprise applications are often stored in the form of graphs. To process such data, graph processing systems are typically used. In this context, Heidari and Buyya propose two dynamic repartitioning-based algorithms for scheduling of large-scale

graphs in a cloud environment [131]. The proposed algorithms consider network factors in order to reduce the costs. The authors also introduce a novel classification for graph algorithms and graph processing systems, which can aid to select the best strategy for processing a given input graph.

In a **federated multi-cloud environment**, different types of resources that may be geographically distributed can be collectively exposed as a single elastic infrastructure. By doing so, the execution of application workflows with heterogeneous and dynamic requirements can be optimized, and the federated multi-cloud can tackle larger scale problems. Diaz-Montes et al. introduce a framework for managing the end-to-end execution of data-intensive application workflows within a federated cloud [125]. The proposed framework also supports dynamic federation, in which computational sites can join or leave on the fly, and the framework can recover from failures happening within a site.

For scheduling of workloads that are executed inside **containers**, Kang et al. propose a brokering system that aims to minimize the energy consumption, while guaranteeing an acceptable performance level [76]. The authors also propose a new metric, called Power consumption Per Application (ppA), and the proposed system applies workload clustering using the k-medoids algorithm. Simonis presents a container-based architecture for big-data applications, that allows for interoperability across data providers, integrators and users [142]. By using self-contained containers, the presented architecture allows for horizontal scale-out, high reliability and maintainability. Takahashi et al. introduce a portable load balancer for Kubernetes clusters, which is usable in any environment, and hence facilitates the integration of web services [144].

Highlights for workload management: Workloads that are being executed in a cloud environment are often bound by multiple constraints, which should be taken into account by the scheduling strategy to guarantee the required QoS. In recent years, several strategies have been proposed, but most of them focus on the execution of workflows inside VM instances, for example by predicting the minimal amount of VM resources for a given set of tasks. In a federated multi-cloud environment, geographically distributed resources can be exposed as a single elastic infrastructure, to optimize the execution of application workflows and to tackle large scale problems. An important challenge in this context is support for dynamic federation, meaning that computational sites should be able to join or leave on the fly, and the used framework should be able to cope with such changes. Executing workloads inside containers offers great portability and scalability as the virtualization overhead is much smaller compared to traditional VMs.

5.3.2.2 Application Elasticity and Provisioning

Applications deployed in a cloud environment can benefit from the offered elasticity by adjusting the provisioned amount of resources based on the current demand. Additional instances can be deployed on the fly, and a load balancer will typically be used to distribute the load over the available instances. Cloud applications however are often stringent to given Service-Level Objectives (SLOs), agreed upon between the cloud user and the application end user. In order to satisfy a given service level objective, the **minimal amount of cloud resources** required for the given task needs to be determined. Salah et al. present an analytical model, based on Markov chains, to predict the minimal number of VMs required for satisfying a given SLO performance requirement [82]. Their model takes the offered workload and number of VM instances as input, together with the capacity of each VM instance. The model not only returns the minimal number of VMs required for the workload, but also the required number of load balancers needed for achieving proper elasticity. Mireslami et al. present a multi-objective cost-effective algorithm for minimizing the deployment cost while meeting the QoS performance requirements [105]. The proposed algorithm offers the cloud user an optimal choice when deploying a web application in a traditional cloud environment. Righi et al. introduce a fully-organizing PaaS-level elasticity model, designed specifically for running High-Performance Computing (HPC) applications in the cloud [81]. Their model does not require any user intervention or modifications to the application's source code, but (de-)allocates VMs using an aging-based approach to avoid unnecessary VM re-configurations. The model also uses asynchronism for creating and terminating VMs in order to minimize the execution time of the HPC applications.

In a **mobile cloud environment**, mobile devices can transfer resource-intensive computations to a more resourceful computing infrastructure, such as a public cloud environment. Multiple offloading approaches exist, often focusing on different objectives or following a different approach [23, 72, 121, 137, 149]. Nawrocki and Sniezynsky for example propose an agent-based architecture with learning possibilities, based on supervised and reinforcement learning, to optimally schedule services and tasks between the mobile device and the cloud [137]. Elgazzar et al. propose a framework for cloud-assisted mobile service provisioning, which aims to assist mobile devices in delivering reliable services [72]. The presented framework supports dynamic offloading, based on the current resource utilization and network conditions, while satisfying the user-defined energy constraints. Barrameda and Samaan focus on the costs, and present a statistical cost model for offloading in a mobile cloud environment [121]. In this cost model, the application is modelled as a tree structure for representing dependencies and relations among the application modules. The cost for each module is then modelled as a cumulative distribution function that is statistically estimated through profiling. Zhang et

al. on the other hand investigate the topic of energy-efficient task offloading, and propose an algorithm that aims to minimize the energy consumption on the mobile devices while still guaranteeing deadlines [149]. Somehow related, Mebrek et al. also focus on the energy efficiency, but in the context of a multi-tier IoT-fog-cloud environment, and the authors present a model for the power consumption and delay for IoT applications within both fog and traditional cloud environments [102]. Similarly, Yao and Ansari present an approach for offloading and resource provisioning in an IoT-fog environment, but the authors aim to minimize the VM rental cost for the fog environment while still guaranteeing QoS requirements.

In **multi-cloud environments**, applications or individual components should be deployed in the environment that is best suited. Cloud providers may offer their services using different pricing models, and some models may be more suitable for either short term or long term tasks. For the storage of data in heterogeneous multi-cloud environments, Zhang et al. introduce a data hosting scheme which aims to help the cloud user by selecting the most suitable cloud environment, together with an appropriate redundancy strategy for achieving high availability [66]. The proposed solution considers the used pricing strategy, the availability requirements and the data access patterns. For deploying applications in a multi-cloud environment, Khatua et al. introduce several algorithms which aim to determine the optimal amount of resources to be reserved, while minimizing the total cost by selecting the most appropriate pricing model. Xu et al. focus on scientific workflows, and present an energy-aware resource allocation method for the dynamic deployment and scheduling of VMs across multiple cloud computing platforms [87].

For applications running in a **multi-tiered (layered) cloud environment**, which for example could consists of an edge, a fog and a central cloud layer, Alam et al. present a layered modular and scalable architecture that aims to increase the efficiency of the applications [116]. The proposed architecture collects and analyzes data at the most efficient and logical place, balances the load, and pushes computation and intelligence to the appropriate layers. Furthermore, the proposed architecture uses Docker containers, which simplifies the management and enables distributed deployments. Similarly, Santos et al. propose a framework for the autonomous management and orchestration of IoT applications in an edge-fog-cloud environment [21]. The authors introduce a Peer-to-Peer fog protocol for the exchange of application service provisioning information between fog nodes. Rahimi et al. focus on multi-tiered mobile cloud environments, and present a framework for modeling mobile applications as location-time workflows, in which user mobility patterns are translated to mobile service usage patterns [139]. These workflows are then mapped to the appropriate cloud resources using an efficient heuristic algorithm. Bouet and Conan also focus on multi-tiered mobile cloud environments, and propose a geo-clustering approach for optimizing the edge computing resources [123]. The authors introduce an algorithm that provides a partition of

mobile edge computing clusters, which consolidates as many communications as possible at the edge.

Highlights for application elasticity and provisioning: Applications deployed in a cloud environment can be stringent to given SLOs. To satisfy these objectives, the required amount of resources needs to be determined. Multiple prediction models have been presented, but most of them focus on the deployment of applications inside VMs, which differs from deployment inside containers as VM re-configurations are typically costly and should hence be avoided. With mobile cloud, edge and fog computing, less powerful devices can transfer computational intensive tasks to another environment. This requires an offloading approach, that could for example focus on energy efficiency or minimizing the operational costs. For these environments, containers offer clear benefits, as they facilitate the management and allow for distributed deployments. In multi-cloud environments, the application or individual components should be deployed in the optimal environment, for example to balance the load or to minimize the operational costs.

5.3.2.3 Local Provisioning and Scheduling

With traditional cloud computing, multiple VMs are typically deployed onto a single server, and a **hypervisor** is used for allocating the virtual resources on top of the physical hardware. Zhang et al. argue that when VMs deployed onto the same physical server compete for memory, the performance of the applications deteriorates, especially for memory-intensive applications [115]. To tackle this issue, the authors propose an approach for optimizing the memory control using a balloon driver for server consolidation. Li et al. on the other hand argue that in virtualized environments, the accuracy of CPU proportional sharing and the responsiveness of I/O processing are heavily dependent on the proportion of the allocated CPU resources [98]. The authors illustrate that an inaccurate CPU share ratio, together with CPU proportion dependent I/O responsiveness, can affect the performance of the hypervisor. This could lead to unstable performance and therefore could violate SLA requirements. As a result, the authors propose a novel scheduling scheme that achieves accurate CPU proportional sharing and predictable I/O responsiveness. Katsalis et al. also focus on CPU sharing, and present several CPU provisioning algorithms for service differentiation in cloud environments [55]. The algorithms are based on dynamic weighted round robin, and guarantee CPU service shares in clusters of servers. The authors illustrate that the presented solution provides service differentiation objectives, without requiring any knowledge about the arrival and service process statistics. Mukherjee et al. argue that, while resource management methods may manage application performance by controlling the sharing of processing time and input-output rates, there is generally no

management of contention for virtualization kernel resources or for the memory hierarchy and subsystems [60]. Such contention however can have a significant impact on the application performance. As a result, the authors present an approach for detecting contention for shared platform resources in virtualized environments. Amennejad et al. illustrate that when VMs compete for shared physical machine resources, the web services deployed on these VMs could suffer performance issues [49]. Cloud users however typically have only access to VM-level metrics and application-level metrics, but these metrics are often not useful for detecting inter-VM contention. As a result, the authors propose a machine-learning based interference detection technique to predict whether a given transaction being processed by a web service is suffering from interference. The proposed technique only relies on web transaction response times, and does not require any access to performance metrics of the physical resources.

For **container-based deployments**, Nakagawa and Oikawa argue that the deployed containers often consume much more memory than expected [79]. Although there are several methods to prevent such memory overuse, most existing methods have their shortcomings such as an increase in operational costs, or the detection of false-positives. In their paper, the authors propose a new memory management method for container-based virtualization environments. The proposed method detects containers that have a sign of memory overuse, and puts a limitation on the allowed memory consumption for these containers. Lahmann et al. investigate if VM resource allocation schemes are appropriate for container deployments [135]. Specifically, they focus on the gaps between memory allocation and memory utilization for application deployments in container clusters. Sharma et al. study the differences between hardware virtualization (VMs) and OS virtualization (containers) regarding performance, manageability and software development [26]. According to their findings, containers promise bare metal performance, but they may suffer from performance interference as they share the underlying OS kernel. Unlike VMs which typically have strict resource limits, containers also allow for soft limits, which can be helpful in over-commitment scenarios as they can make use of underutilized resources allocated to other containers. Tesfatsion et al. also study the differences between VMs and containers, but with a focus on the virtualization overhead [27]. According to the presented results, no single virtualization technology is a clear winner, but each platform has its advantages and shortcomings. Containers for example offer a lower virtualization overhead, but can raise security issues due to the lower level of isolation. Both Tesfatsion and Sharma however note that a hybrid form, in which containers are deployed on top of VMs, could offer promising solutions that combine the advantages of both virtualization technologies.

However, when containers are provisioned inside VMs, the guest OS manages virtual resources inside a VM, whereas the hypervisor manages the physical re-

sources distributed among the VMs. As a result, two control centers are managing the set of resources used by the containers. The hypervisor typically takes control actions such as memory ballooning, which allows a host system to artificially enlarge its memory pool by reclaiming unused memory allocated to other virtual machines, or withdrawal of a virtual CPU to manage over-provisioning, without being aware of the effects of those actions on individual containers deployed inside the VM. Prakash et al. illustrate that such actions can have unpredictable and non-deterministic effects on the nested containers [35]. To tackle this issue, the authors propose a policy driven controller that smooths over the effects of hypervisor actions on the nested containers.

Highlights for local provisioning and scheduling: With hardware virtualization, a hypervisor will strictly allocate resources to the deployed VMs. The deployed VMs however can compete for the shared physical resources, but the hypervisor should detect and prevent this to not violate SLA requirements. With OS level virtualization, the underlying OS kernel is shared, and containers can use unutilized resources allocated to other containers. These soft limits should be taken into account, as they can have unpredictable effects on other unrelated containers deployed on the same physical hardware. Each virtualization technology clearly has its advantages and limitations, and deploying containers inside VMs could combine the advantages of both technologies, but this introduces challenges for resource management as two control centers are managing the set of resources used by the containers.

5.3.2.4 Global Provisioning and Scheduling

As illustrated in Table 5.4, the majority of research is focusing on the global provisioning and scheduling of cloud resources. When it comes to resource allocation, the used scheme can be either static or dynamic, with the latter indicating that the amount of resources allocated for a specific task can change over time. In this context, Wolke et al. did an experimental study on the benefits of **dynamic resource allocation** for the allocation of VMs [36]. According to their findings, reactive or proactive control mechanisms do not always decrease the average server demand, but instead can lead to a high number of migrations, which negatively impacts the response times and could even lead to network congestion. The authors note that in general, live VM migrations should be exceptional, and capacity planning via optimization should be used instead of dynamic allocation, especially in environments with long-running and predictable application workloads. Somewhat related, Wu et al. study the overhead introduced by launching new Virtual Machines in the context of Cloud bursting [86]. According to their findings, this overhead is not constant, but instead depends on the physical resource utilization (e.g. CPU and

I/O device utilization) at the time when the VM is launched. This variation in overhead can have a significant impact on cloud bursting strategies. As a result, they introduce a VM launching overhead reference model based on operational data, which could help to decide when and where a new VM should be launched.

Global provisioning and scheduling often includes **VM consolidation** [73, 75, 78, 96, 107, 119, 129], which typically aims to pack the virtual machines onto few physical servers in order to reduce the operational costs. Huang et al. for example present a framework for VM consolidation that aims to achieve a balance among multiple objectives [75], which can also be used in a context that requires minimal system re-configurations. Similarly, Guo et al. present an approach for the real-time adaptive placement of VMs in large data centers. The authors use a shadow routing based approach, which allows for a large variety of objectives and constraints to be treated within a common framework. When consolidating VMs, both the relationships and possible interference between collocated VMs, as well as the tightness of packing should also be taken into account. Espling et al. for example introduce an approach for the placement of VMs with an internal service structure, component relationships and placement constraints between them [73]. Jin et al. present an approach that takes into account the possible interference between collocated VMs, as this interference can have a negative impact on the performance of the deployed applications. Mishra et al. on the other hand present a study on the tightness of VM packing [78]. A tight packing approach can lead to future issues as there is no room to expand, whereas provisioning VMs for their peak usage can result in wasted resources as peaks occur infrequently and typically for a short time. Liu et al. however prefer an aggressive resource provisioning approach [58], by initially over-provisioning resources and later reducing the amount of resources if needed. Doing so can increase the performance by reducing the adaption time, while limiting SLO violations when dealing with rapidly increasing workloads. On the physical servers hosting the VMs, some resources could be left unused and therefore wasted when they are insufficient for hosting a new VM. In this context, Nitu et al. propose a consolidation strategy that dynamically divides a VM into smaller 'pieces', so that each piece fits into the available 'holes' on the servers [107].

Some provisioning and scheduling schemes have been proposed that focus on the **deployment of containers** in a cloud environment [48, 69, 89, 90, 95, 106]. Awada and Barker for example present a cloud-based container management service framework, that offers the required functionalities for orchestrating containerized applications [90]. Their framework takes into account the heterogeneous requirements of the applications, and jointly optimizes sets of containerized applications and resource pools within a cloud environment. The authors also presented an extension of their framework for use in multi-region cloud container-instance clusters [89]. Abdelbaky et al. also focus on a multi-cloud environment, and in-

troduce a framework that enables the deployment and management of containers across multiple hybrid clouds and clusters [48]. Their framework takes into account the objectives and constraints of both the cloud provider and cloud user, and uses a constraint-programming model for selecting the required resources. For the deployment of containers within VMs, Nardelli et al. introduce a strategy for the elastic provisioning of VMs required for deploying the containers [106]. Hoque et al. analyzed different container orchestration tools, and present a framework for the orchestration of containers within a fog cloud environment [95].

Highlights for global provisioning and scheduling: The allocation of resources can be either static or dynamic. A dynamic allocation strategy could lead to a higher efficiency, but the introduced reconfiguration overhead should not be neglected. Therefore, using a dynamic allocation strategy will not always be beneficial, especially when provisioning VMs. The (re)allocation of VMs often includes VM consolidation, which aims to pack the VMs onto few physical servers. During the VM consolidation process, the tightness of packing plays an important role, and possible relationships between VMs should be taken into account. When deploying containers, an orchestrator is typically used to optimize the allocation scheme over the available resources.

5.3.3 Resource Profiling

Table 5.5 provides an overview of recent work related to application or infrastructure demand profiling, resource utilization estimation and/or monitoring. This table is quite limited in size, as it only includes items with a main focus on profiling. In this table, we categorized the research items based on the attributes of Table 5.2.

5.3.3.1 Application Demand Profiling

When deploying applications in an IaaS cloud environment, both the quantity and type of VM resources need to be determined. Application demand profiling can be used for assessing demand patterns for individual applications, which can be used as input for workload management and application pricing. In this context, Lloyd et al. introduce a workload cost prediction methodology which harnesses operating system time accounting principles to support equivalent workload performance using alternate virtual machine types [100]. By using resource utilization checkpoints, the total resource utilization profile is captured for service oriented application workloads executed across a pool of VM. Based on the obtained workload profiles, the estimated cost is calculated, which could help cloud users for finding alternate infrastructures that afford lower hosting costs while offering equal or

Table 5.5: Overview of recent research with a main focus on profiling (application and infrastructure demand profiling, resource utilization estimation and/or monitoring). ADP Application Demand Profiling, IDP (Virtual) Infrastructure Demand Profiling, Est Resource Utilization Estimation, Mon Monitoring, TC Traditional Cloud, FC Fog Computing, EC Edge Computing, SC Single Cloud, MC Multi-Cloud, VM Virtual Machine, CT Container.

		Function				Cloud Type			Scope		Entity	
Publication	Year	ADP	IDP	Est	Mon	TC	FC	EC	SC	MC	VM	CT
Dabbagh et al. [51]	2015		✓	✓	✓	✓			✓		✓	✓
Dhakate et al. [52]	2015				✓	✓				✓	✓	✓
Rodrigues et al. [70]	2016				✓	✓			✓		✓	
Zhou et al. [88]	2016				✓	✓			✓		✓	
Chard et al. [92]	2017	✓				✓				✓	✓	
Dalmazo et al. [93]	2017			✓		✓			✓		✓	
Lloyd et al. [100]	2017	✓				✓			✓		✓	
Balos et al. [120]	2018	✓	✓	✓		✓				✓	✓	
Hauser et al. [130]	2018		✓	✓	✓	✓			✓		✓	✓
Prats et al. [138]	2018	✓	✓		✓	✓			✓			
Scheuner et al. [141]	2018	✓				✓			✓		✓	
Trihinas et al. [145]	2018				✓	✓			✓	✓	✓	

better performance. Somewhat related, Prats et al. introduce an approach for the automatic generation of workload profiles [138]. The authors examine and model application behavior by finding phases of similar behavior in the workloads. In the presented approach, resource monitoring data is first passed through conditional restricted Boltzmann machines to generate a low-dimensional and time-aware vector. This vector is then passed through clustering methods such as k-means and hidden Markov models to detect the similar behavior phases.

Chard et al. introduce a middleware for the profiling, prediction and provisioning of applications in a cloud environment [92]. The authors have developed an automated profiling service that is able to derive approximate profiles for applications executed on different environments. Based on these profiles, the expected cost is calculated for executing a particular workload in a dynamic cloud market, with the aim of computing bids that are based on probabilistic-durability guarantees. Once the results from profiling and market prediction are obtained, the middleware provisions infrastructure and manages it throughout the course of the workload execution.

Due to the immense growth in the cloud computing market and the resulting wide diversity of cloud services, micro-benchmarks could be used for identifying

the best performing cloud services. As a result, Scheuner and Leitner have developed a cloud benchmarking methodology that uses micro-benchmarks to profile applications, in order to predict how an application performs on a wide range of cloud services [141]. The authors validated their approach using several metrics and micro-benchmarks with two applications from different domain. Although micro-benchmarking is a useful approach, the results illustrate that only few selected micro-benchmarks are relevant when estimating the performance of a particular application.

Within the context of scientific computing, Balos et al. present an analytical model that matches scientific applications to effective cloud instances for achieving high application performance [120]. The model constructs two vectors, an application vector consisting of application performance components and a cloud vector comprising cloud-instance performance components. By profiling both the application and cloud instances, an inner product of both vectors is calculated to produce an application-to-cloud score, which represents the application's execution time on the selected cloud instance.

Highlights for application demand profiling: Application demand profiling can be useful for estimating the required amount of resources, as well as the expected operational costs. In a public cloud market, profiling applications can also be used to determine the best suited environment. Applications can either be profiled as a whole, or micro-benchmarks can be used to predict how an application would perform.

5.3.3.2 Monitoring, Infrastructure Demand Profiling and Resource Utilization Estimation

Da Cunha Rodriguez et al. present a general overview of cloud monitoring [70]. According to the authors, cloud monitoring systems play a crucial role for supporting scalability, elasticity, and migrations within a cloud environment. They also provide a comparison among relevant cloud monitoring solutions, focusing on abilities such as the accuracy, autonomy and comprehensiveness.

For automatic resource provisioning, the deployed applications, services and the underlying platforms need to be continuously monitored at multiple levels and time intervals. Trihinas et al. however argue that current cloud monitoring tools are either bound to specific cloud platforms, or have limited portability to provide elasticity support [145]. The authors describe several challenges for monitoring elastically adaptive multi-cloud services, and introduce an automated, modular, multi-layer and portable cloud monitoring framework. The presented framework can automatically adapt when elasticity actions are enforced to either the cloud

service or to the monitoring topology, and can recover from faults introduced in the monitoring configuration.

Hauser and Wesner present an approach for monitoring resource statistics on the physical infrastructure level [130], to provide the required information for profiling of the physical resources. Based on the monitoring information, a resource utilization profile is provided to the cloud middleware and customer. Such a profile consists of both a static (e.g. number of CPU cores) and dynamic part (e.g. current utilization), and is generated using statistical computations like histograms and Markov chains.

Dabbagh et al. propose an energy-aware resource provisioning framework that predicts future workloads [51]. Based on monitoring information, the proposed framework predicts the number of future VM requests, along with the amount of CPU and memory resources associated with each of these requests, and provides accurate estimations of the number of physical machines required. By putting unused physical machines to sleep, the framework also reduces the energy consumption of the cloud data center. Although the proposed solution is based on the provisioning of VMs, the authors note that their framework could easily be adapted for estimating the number of physical machines required for the provisioning of containers.

Monitoring can also play an important role for achieving high availability and reliability. As the public cloud is a multi-tenant environment, failure of a single physical component can have a significant impact on a large number of tenants. To increase cloud reliability, Zhou et al. present a recovery approach based on checkpoint images, which consist of service checkpoint images and delta checkpoint images [88].

Dhakate and Godbole propose an architecture for monitoring, testing, reporting and alerting of an entire cloud environment [52]. The required monitoring software is packed inside Docker containers, which can be deployed directly from the Docker Hub repository. The authors also developed a dashboard that provides a general overview of the health status of the whole cloud environment.

Highlights for monitoring, infrastructure demand profiling and resource utilization estimation: Monitoring systems play a crucial role for supporting scalability, elasticity, and migrations within a cloud environment. Together with resource utilization estimation, a resource utilization profile can be generated. Monitoring can also aid in achieving high availability and reliability. When the monitoring system detects a failure, it can initiate a recovery approach, or alert the cloud provider.

Table 5.6: Overview of recent research with a main focus on resource pricing. APr Dynamic Application Pricing, IPr Dynamic Virtual Infrastructure Pricing. TC Traditional Cloud, FC Fog Computing, EC Edge Computing, SC Single Cloud, MC Multi-Cloud, VM Virtual Machine, CT Container.

		Function		Cloud Type			Scope		Entity	
Publication	Year	APr	IPr	TC	FC	EC	SC	MC	VM	CT
Aazam et al. [47]	2015		✓	✓	✓	✓		✓		
Huang et al. [53]	2015		✓	✓		✓		✓		
Jin et al. [54]	2015		✓	✓			✓		✓	
Lee et al. [57]	2015		✓	✓		✓	✓		✓	✓
Mashayekhy et al. [38]	2015		✓	✓			✓		✓	
Petri et al. [61]	2015		✓	✓				✓	✓	
Sharma et al. [62]	2015		✓	✓			✓			
Wang et al. [64]	2015		✓	✓				✓		
Aazam et al. [67]	2016		✓	✓				✓	✓	
Mashayekhy et al. [39]	2016		✓	✓			✓		✓	
Wan et al. [84]	2016		✓	✓			✓			
Wanis et al. [85]	2016	✓	✓	✓			✓		✓	
Babaioff et al. [91]	2017		✓	✓			✓		✓	✓
Chi et al. [37]	2017		✓	✓			✓		✓	
Hai et al. [94]	2017	✓	✓	✓		✓	✓			
Tang et al. [110]	2017		✓	✓			✓	✓	✓	
Yi et al. [113]	2017		✓	✓			✓		✓	✓
Borjigin et al. [122]	2018		✓	✓				✓		
Mikavica et al. [40]	2018		✓	✓			✓		✓	
Zhang et al. [150]	2018		✓	✓			✓			

5.3.4 Resource Pricing

Table 5.6 provides an overview of recent research focusing on resource pricing. In this table, we categorized the research items based on the attributes of Table 5.2. As most items focus on (virtual) infrastructure pricing, in the remainder of this section, we will only discuss this functional element. We will first provide a brief overview of research built on top of static pricing models, followed by research focusing on dynamic pricing models.

5.3.4.1 Static Pricing

In the IaaS market, virtual resources are typically priced using a pay-per-use pricing model, and the granularity of usage for such pricing is often at virtual machine level. However, a majority of applications running on top of VMs struggle to

fully utilize the allocated amount of resources, leading to a waste of unused resources [54, 57] and are therefore not cost-efficient due to these coarse-grained pricing schemes.

Jin et al. investigate an optimized fine-grained and fair pricing scheme [54]. The authors address two main issues: the profits of resource providers and customers often contradict mutually, and the VM maintenance overhead like startup costs are often too huge to be neglected. The presented solution not only derives an optimal price in the acceptable price range, that satisfies both customers and providers, but also finds a best-fill billing cycle to maximize social welfare. Lee et al. also propose a resource management mechanism for fine-grained resource sharing, which allows for real pay-per-use pricing [57]. Their mechanism consists of a container-based resource allocator, and a real-usage based pricing scheme. By using containers instead of virtual machines, a higher resource utilization can be achieved and the authors also illustrate that the proposed mechanism can achieve a near-optimal cost efficiency.

Tang et al. investigate the problem of joint pricing and capacity planning in the IaaS provider market [110]. The authors study two models, in the first model there is a single IaaS provider (monopoly market), whereas the second model considers multiple IaaS providers. For the monopoly market model, the authors propose a method for determining the optimal amount of end-user requests to admit and number of VMs to lease for SaaS providers, based on the current resource price charged by the IaaS provider. For the model with multiple IaaS providers, the authors propose an iterative game-theory based algorithm for finding the so-called Nash equilibrum. Borjigin et al. also present an approach for finding the Nash equilibrum, but within NFV markets [122]. The presented double-auction approach aims to maximize the profits for all participants, being the brokers, the cloud users and the cloud providers.

Yi et al. argue that cloud users with small and short demands, typically cannot find an instance type offered by a cloud provider that fits their needs or fully utilizes the purchased instance-hours [113]. On the other hand, cloud providers are faced with the challenge of consolidating small, short jobs, which exhibit strong dynamics, to effectively improve resource utilization. To address these issues, the authors propose a novel group buying mechanism that organizes jobs with complementary resource demands into groups, and allocates them to container group buying deals predefined by cloud providers. Each group buying deal offers a resource pool for all the jobs in the deal, which can be implemented as a virtual machine or a physical server. By running each job inside a container, the proposed solution allows for flexible resource sharing among the different users in the same group buying deal, while improving resource utilization for the cloud providers.

Highlights for static virtual infrastructure pricing: Static pricing models are often based on the number of provisioned VMs. A majority of applications however struggle to fully utilize the allocated amount of resources, leading to a waste of unused resources. A fine-grained pricing model could tackle this issue, presenting an interesting opportunity for the deployment of applications inside containers. Small and short tasks can be executed in containers, which can then be grouped and allocated to VMs. A group buying approach can be used for acquiring the required set of VMs.

5.3.4.2 Dynamic Pricing

When using a dynamic, **auction-based pricing model**, multiple cloud users bid for a bundle of typically heterogeneous cloud instances. The cloud provider will then select a set of cloud users, and needs to determine a feasible allocation over its set of physical machines. A major issue with dynamic auction-based pricing is that cloud users are typically self-interested, meaning that they want to maximize their own utility. The cloud users could untruthfully alter their requests, for example by requesting several sets of resources different from their actual need, in order to manipulate the outcomes of the bidding and to gain an unfair advantage [38, 64, 67]. To tackle this issue, Mashayekhy et al. [38] propose a resource management mechanism that consists of three phases: winner determination, provisioning and allocation, and pricing. In the winner determination phase, the cloud provider decides which users receive the requested bundles. In the provisioning and allocation phase, VM instances are provisioned to the winning users. In the pricing phase, the cloud provider dynamically determines the price that the winning users should pay for their requests. The authors claim that their solution is strategy-proof, meaning that cloud users have no incentives to lie about their requested bundles and their valuations. In [39], the authors propose an auction-based online mechanism for VM provisioning, allocation and pricing in clouds that considers several types of resources. The proposed mechanism allocates VM instances to selected users for the period they are requested for, and ensures that users will continue using their VM instances for the requested period. In addition, the mechanism determines the price users have to pay for using the allocated resources. The authors prove that the mechanism is incentive-compatible, meaning that it gives incentives to users to reveal their actual requests.

Cloud data centers often consist of heterogeneous infrastructure, and the cloud provider could adapt the offered prices based on the used hardware. Zhang et al. for example present an approach for the pricing of cloud storage for data centers consisting of multiple storage tiers that offer distinct characteristics [150]. The approach is based on a two-stage auction process for requesting storage capacity

and accesses with given latency requirements. The presented solution provides a hybrid storage and access optimization framework, which aims to maximize the cloud provider's net profit over multiple dimensions.

When the current demand is low, cloud providers can offer their services at a lower price, e.g. Amazon's **spot instances**. Recently, Amazon introduced a new variety of spot instances, namely spot block instances [151]. These instances run continuously for a finite duration (1 to 6 hours). Pricing is based on the requested duration and the available resources, and spot block prices are typically 30 to 45% less than on-demand prices. Mikavica et al. analyze two auction-based pricing mechanisms, namely uniform price auction and generalized second-price auction, for pricing the cloud provider's idle resources in the form of spot block instances [40]. Furthermore, the authors propose a model for spot block price determination under these pricing mechanisms. The presented results show that, regardless of the chosen auction mechanism and bidding strategy, spot block instances are a cost-effective solution that embodies advantages of both on-demand instances and spot instances. Wan et al. on the other hand present a reactive pricing algorithm, allowing the cloud provider to determine the server price based on the actual resource demand [84]. The presented approach takes into account the renewable energy, spot power price and the battery level, and dynamically tunes the server price in response to state changes. The authors focus on pricing of physical servers, but the presented approach can easily be extended for pricing of VMs.

In a **multi-cloud environment**, service and resource providers can co-exist in a market where the relationship between clients and services depends on the nature of the application and can be subject to a variety of different QoS constraints. Deciding whether a cloud provider should host a service in the long-term would be influenced by parameters such as the service price, the QoS guarantees required by the customers, the deployment costs and the constraints. In this context, Petri et al. introduce a market model to support federated clouds and investigate its efficiency using two real application scenarios [61]. The authors also identify a cost-decision based mechanism to determine when tasks should be outsourced to external sites in the federation. Wang et al. also focus on multi-cloud environments, by introducing an intelligent economic approach for dynamic resource allocation, which can be used for the trading of various kinds of resources among multiple consumers and providers [64]. The presented approach is based on intelligent combinatorial double auction, and includes a price formation mechanism, consisting of price prediction and matching. The authors also propose a reputation system to exclude dishonest participants, as well as a paddy field algorithm for selecting the winners.

In a **federated cloud environment**, services can be provided through two or more clouds, which is often done using a middleware entity, called a cloud broker. Such cloud broker is responsible for reserving and managing the resources, discovering services according to the customer's demands, SLA negotiation, and

match-making between the involved service provider and the customer. Aazam et al. present a holistic brokerage model to manage on-demand and advanced service reservation, pricing and reimbursement [67]. The authors consider dynamic management of customer's characteristics as well as taking into account historical records when evaluating the economics related factors. Futhermore, they provide a mechanism of incentives and penalties, which helps to establish trust between the cloud users and service providers.

Highlights for dynamic virtual infrastructure pricing: With a dynamic, auction-based pricing model, multiple cloud users bid for a bundle of cloud resources. A major issue with this is that the cloud users can alter their requests in order to manipulate the bidding outcomes. To tackle this issue, the cloud provider could give incentives to the users to reveal their actual requests. In federated and other multi-cloud environments, a broker is typically used for reserving and managing resources. When allocating resources, this broker could take into account the actual prices offered by the different environments, in order to minimize the costs.

5.4 Challenges and Opportunities

Virtualization is the fundamental technology that powers cloud computing, and the majority of cloud providers are still providing virtual resources in the form of VMs to the cloud users. As a result, the majority of research about resource management in cloud environments is focusing on the different aspects related to the provisioning, profiling and pricing of such VMs. Container technology however is gaining popularity, as it offers a more lightweight alternative to traditional VMs. Apart from this new virtualization technology, new cloud models are emerging, bringing the cloud closer to the end user, which is especially useful for devices with a limited network connection, or for low-latency applications. In this section, we identify several challenges and opportunities for resource management in cloud environments, mainly related to these recent trends.

5.4.1 Dynamic resource allocation for containerized applications

Dynamic resource allocation for VMs will not always be beneficial for the cloud environment, due to the costly nature of VM migrations [36]. Existing dynamic resource allocation approaches therefore often put a heavy penalty on such migrations to avoid unnecessary VM re-configurations. However, as containers are much more lightweight and portable, live migrations of containers will have a

much smaller overhead compared to VM migrations. Especially when the application is designed using a service oriented architecture with mainly stateless microservices running inside the containers, migration of running containers and scaling up or down individual services should be straightforward.

In this context, it would be interesting to see how existing dynamic allocation strategies designed for the allocation of VMs perform when handling containers. Lahmann et al. already did some initial research [135], but with a main focus on memory allocation and memory utilization. When containers are deployed inside VMs, a static resource allocation strategy could be used for the provisioning of the virtual machines, combined with a dynamic strategy for the deployment of containers inside VMs.

5.4.2 Cloud management systems for bare-metal containers

When containers are deployed inside VMs, actions taken by the hypervisor can have unpredictable and non-deterministic effects on the nested containers [35]. Virtual machines also introduce a noticeable overhead, as they typically run a full software stack. When running containers directly on the OS of the physical machine, this overhead could be eliminated, which could lead to a higher scalability, efficiency and a higher resource utilization. This however introduces the need for a cloud management system that manages the allocation of containers on the physical hardware.

A bare-metal cloud container management system should not only provide the required functionalities for allocation and provisioning of containers, but should also guarantee sufficient security and isolation between the different tenants.
Achieving a clear isolation however is challenging, as containers share the underlying OS kernel [26]. Furthermore, such a system should also monitor the actual amount of resources used over time by the deployed containers, in order to charge the customers based on the actual resource usage. Unlike VMs, containers often have soft limits, meaning that the actual usage can be different from the allocated amount of resources [26, 79]. This presents opportunities for achieving a higher overall resource utilization, but the management system should also have built-in functionalities for preventing starvation when highly demanding containers clog up all available resources.

5.4.3 Management of a hybrid edge/fog/cloud environment

In a hybrid edge-fog-cloud environment, resources that may be geographically distributed can be collectively exposed as a single elastic infrastructure. This however introduces the need for a framework that coordinates the management of resources among the different environments. While there is already some initial research available [152], many research challenges are still remaining.

To achieve an efficient deployment of applications in such an environment, a feasible location for each component needs to be determined [116], ideally in an autonomous way. The use of portable containers can facilitate the management and migration of the components. For components deployed in a public cloud environment, security challenges introduced by the multi-tenant cloud environment should also be addressed [153, 154]. The hybrid environment should also allow for auditing in order to create reliable and secure cloud services [155].

5.4.4 Experimental validation of resource management strategies

Resource management strategies are often only validated by means of simulations [156], for example by using CloudSim [157], in which the whole cloud computing environment is modeled and simulated in software. This is mainly because of the nature of the research, as resource allocation strategies for example are often designed for managing large sets of applications within large cloud environments. Experimental validation using real cloud hardware would not only be costly as it would require multiple cloud instances for a relative long time period, the validation process would also be time-consuming.

The rise of new cloud types such as fog cloud environments, as well as the adoption of container technology however can facilitate the validation of resource management strategies. Using low-cost hardware, a small-scale test bed could be built for the initial validation. A Raspberry Pi for example is already powerful enough to host several containers. By combining experiments on a small-scale test bed with simulations using large-scale scenarios, the research would not only gain credibility, but an implementation of the proposed solution on real hardware would also illustrate that the resource management strategy works in practice.

5.5 Conclusions

In this chapter, we presented an overview of recent research, published between 2015 and 2018, with a main focus on resource management within cloud environments. We especially investigated how cloud resource management is adapting to support newly introduced trends, such as containers as the virtualization technology and the rise of fog and edge computing. We categorized the research items based on the main resource management functional element, and provided a brief summary for each element. While the majority of recent research is still focusing on the management of virtual machines in a traditional single cloud environment, we identified several interesting opportunities for resource management in a future fully containerized multi-tiered edge-fog-cloud, which could overcome many shortcomings of today's cloud environments.

References

[1] *Swift - OpenStack*. https://wiki.openstack.org/wiki/Swift.

[2] *OpenStack - Build the future of Open Infrastructure*. http://openstack.org.

[3] S. Yi, C. Li, and Q. Li. *A Survey of Fog Computing: Concepts, Applications and Issues*. In Proceedings of the 2015 Workshop on Mobile Big Data, Mobidata '15, pages 37–42, New York, NY, USA, 2015. ACM. Available from: http://doi.acm.org/10.1145/2757384.2757397, doi:10.1145/2757384.2757397.

[4] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, and P. A. Polakos. *A Comprehensive Survey on Fog Computing: State-of-the-Art and Research Challenges*. IEEE Communications Surveys Tutorials, 20(1):416–464, Firstquarter 2018. doi:10.1109/COMST.2017.2771153.

[5] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi. *Cloud Container Technologies: a State-of-the-Art Review*. IEEE Transactions on Cloud Computing, pages 1–1, 2018. doi:10.1109/TCC.2017.2702586.

[6] M. A. Rodriguez and R. Buyya. *Container-based cluster orchestration systems: A taxonomy and future directions*. Software: Practice and Experience, 0(0), 2018. Available from: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2660, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2660, doi:10.1002/spe.2660.

[7] L. F. Bittencourt, A. Goldman, E. R. Madeira, N. L. da Fonseca, and R. Sakellariou. *Scheduling in distributed systems: A cloud computing perspective*. Computer Science Review, 30:31 – 54, 2018. Available from: http://www.sciencedirect.com/science/article/pii/S1574013718301187, doi:https://doi.org/10.1016/j.cosrev.2018.08.002.

[8] J. G. Herrera and J. F. Botero. *Resource Allocation in NFV: A Comprehensive Survey*. IEEE Transactions on Network and Service Management, 13(3):518–532, Sep. 2016. doi:10.1109/TNSM.2016.2598420.

[9] B. Jennings and R. Stadler. *Resource Management in Clouds: Survey and Research Challenges*. Journal of Network and Systems Management, 23(3):567–619, Jul 2015. Available from: https://doi.org/10.1007/s10922-014-9307-7, doi:10.1007/s10922-014-9307-7.

[10] D. Kumar, G. Baranwal, Z. Raza, and D. P. Vidyarthi. *A Survey on Spot Pricing in Cloud Computing*. Journal of Network and Systems Management, 26(4):809–856, Oct 2018. Available from: https://doi.org/10.1007/s10922-017-9444-x, doi:10.1007/s10922-017-9444-x.

[11] Z. A. Mann. *Allocation of Virtual Machines in Cloud Data Centers - A Survey of Problem Models and Optimization Algorithms*. ACM Comput. Surv., 48(1):11:1–11:34, August 2015. Available from: http://doi.acm.org/10.1145/2797211, doi:10.1145/2797211.

[12] M. Masdari, F. Salehi, M. Jalali, and M. Bidaki. *A Survey of PSO-Based Scheduling Algorithms in Cloud Computing*. Journal of Network and Systems Management, 25(1):122–158, Jan 2017. Available from: https://doi.org/10.1007/s10922-016-9385-9, doi:10.1007/s10922-016-9385-9.

[13] P. Poullie, T. Bocek, and B. Stiller. *A Survey of the State-of-the-Art in Fair Multi-Resource Allocations for Data Centers*. IEEE Transactions on Network and Service Management, 15(1):169–183, March 2018. doi:10.1109/TNSM.2017.2743066.

[14] A. Yousafzai, A. Gani, R. M. Noor, M. Sookhak, H. Talebian, M. Shiraz, and M. K. Khan. *Cloud resource allocation schemes: review, taxonomy, and opportunities*. Knowledge and Information Systems, 50(2):347–381, Feb 2017. Available from: https://doi.org/10.1007/s10115-016-0951-y, doi:10.1007/s10115-016-0951-y.

[15] Z.-H. Zhan, X.-F. Liu, Y.-J. Gong, J. Zhang, H. S.-H. Chung, and Y. Li. *Cloud Computing Resource Scheduling and a Survey of Its Evolutionary Approaches*. ACM Comput. Surv., 47(4):63:1–63:33, July 2015. Available from: http://doi.acm.org/10.1145/2788397, doi:10.1145/2788397.

[16] P. Mell and T. Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Technical report, Gaithersburg, MD, United States, 2011.

[17] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. *A View of Cloud Computing*. Commun. ACM, 53(4):50–58, April 2010. doi:10.1145/1721654.1721672.

[18] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere. *Edge-centric Computing: Vision and Challenges*. SIGCOMM Comput. Commun. Rev., 45(5):37–42, September 2015. Available from: http://doi.acm.org/10.1145/2831347.2831354, doi:10.1145/2831347.2831354.

[19] H. Hong. *From Cloud Computing to Fog Computing: Unleash the Power of Edge and End Devices*. In 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pages 331–334, Dec 2017. doi:10.1109/CloudCom.2017.53.

[20] F. Haouari, R. Faraj, and J. M. AlJa'am. *Fog Computing Potentials, Applications, and Challenges*. In 2018 International Conference on Computer and Applications (ICCA), pages 399–406, Aug 2018. doi:10.1109/COMAPP.2018.8460182.

[21] J. Santos, T. Wauters, B. Volckaert, and F. De Turck. *Fog Computing: Enabling the Management and Orchestration of Smart City Applications in 5G Networks*. Entropy, 20(1), 2018. Available from: http://www.mdpi.com/1099-4300/20/1/4, doi:10.3390/e20010004.

[22] S. Sarkar, S. Chatterjee, and S. Misra. *Assessment of the Suitability of Fog Computing in the Context of Internet of Things*. IEEE Transactions on Cloud Computing, 6(1):46–59, Jan 2018. doi:10.1109/TCC.2015.2485206.

[23] J. Yao and N. Ansari. *QoS-Aware Fog Resource Provisioning and Mobile Device Power Control in IoT Networks*. IEEE Transactions on Network and Service Management, pages 1–1, 2018. doi:10.1109/TNSM.2018.2888481.

[24] T. Adufu, J. Choi, and Y. Kim. *Is container-based technology a winner for high performance scientific applications?* In 2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS), pages 507–510, Aug 2015. doi:10.1109/APNOMS.2015.7275379.

[25] E. Eberbach and A. Reuter. *Toward El Dorado for Cloud Computing: Lightweight VMs, Containers, Meta-Containers and Oracles*. In Proceedings of the 2015 European Conference on Software Architecture Workshops, ECSAW '15, pages 13:1–13:7, New York, NY, USA, 2015. ACM. Available from: http://doi.acm.org/10.1145/2797433.2797446, doi:10.1145/2797433.2797446.

[26] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay. *Containers and Virtual Machines at Scale: A Comparative Study*. In Proceedings of the 17th International Middleware Conference, Middleware '16, pages 1:1–1:13, New York, NY, USA, 2016. ACM. Available from: http://doi.acm.org/10.1145/2988336.2988337, doi:10.1145/2988336.2988337.

[27] S. K. Tesfatsion, C. Klein, and J. Tordsson. *Virtualization Techniques Compared: Performance, Resource, and Power Usage Overheads in Clouds*. In Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE '18, pages 145–156, New York, NY,

USA, 2018. ACM. Available from: http://doi.acm.org/10.1145/3184407.3184414, doi:10.1145/3184407.3184414.

[28] *Linux Containers*. https://linuxcontainers.org.

[29] *Docker - Enterprise Container Platform*. https://www.docker.com.

[30] *Docker - Docker Hub*. https://www.docker.com/products/docker-hub.

[31] *Kubernetes - Production-Grade Container Orchestration*. https://kubernetes.io.

[32] *Docker - Swarm mode overview*. https://docs.docker.com/engine/swarm/.

[33] *Docker Blog - Extending Docker Enterprise Edition to Support Kubernetes*. https://blog.docker.com/2017/10/docker-enterprise-edition-kubernetes/.

[34] V. Reniers. *The Prospects for Multi-Cloud Deployment of SaaS Applications with Container Orchestration Platforms*. In Proceedings of the Doctoral Symposium of the 17th International Middleware Conference, Middleware Doctoral Symposium'16, pages 5:1–5:2, New York, NY, USA, 2016. ACM. Available from: http://doi.acm.org/10.1145/3009925.3009930, doi:10.1145/3009925.3009930.

[35] C. Prakash, P. Prashanth, U. Bellur, and P. Kulkarni. *Deterministic Container Resource Management in Derivative Clouds*. In 2018 IEEE International Conference on Cloud Engineering (IC2E), pages 79–89, April 2018. doi:10.1109/IC2E.2018.00030.

[36] A. Wolke, M. Bichler, and T. Setzer. *Planning vs. Dynamic Control: Resource Allocation in Corporate Clouds*. IEEE Transactions on Cloud Computing, 4(3):322–335, July 2016. doi:10.1109/TCC.2014.2360399.

[37] Y. Chi, X. Li, X. Wang, V. C. M. Leung, and A. Shami. *A Fairness-Aware Pricing Methodology for Revenue Enhancement in Service Cloud Infrastructure*. IEEE Systems Journal, 11(2):1006–1017, June 2017. doi:10.1109/JSYST.2015.2448719.

[38] L. Mashayekhy, M. M. Nejad, and D. Grosu. *Physical Machine Resource Management in Clouds: A Mechanism Design Approach*. IEEE Transactions on Cloud Computing, 3(3):247–260, July 2015. doi:10.1109/TCC.2014.2369419.

[39] L. Mashayekhy, M. M. Nejad, D. Grosu, and A. V. Vasilakos. *An Online Mechanism for Resource Allocation and Pricing in Clouds*. IEEE Transactions on Computers, 65(4):1172–1184, April 2016. doi:10.1109/TC.2015.2444843.

[40] B. Mikavica and A. Kostić-Ljubisavljević. *Pricing and bidding strategies for cloud spot block instances*. In 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pages 0384–0389, May 2018. doi:10.23919/MIPRO.2018.8400073.

[41] *ACM Transactions on Internet Technology (TOIT)*. https://dl.acm.org/citation.cfm?id=J780.

[42] *IEEE Transactions on Cloud Computing (TCC)*. https://www.computer.org/csdl/journal/cc.

[43] *IEEE Transactions on Parallel and Distributed Systems (TPDS)*. https://www.computer.org/csdl/journal/td.

[44] *IEEE Transactions on Network and Service Management (TNSM)*. https://www.comsoc.org/publications/journals/ieee-tnsm.

[45] *Springer Journal of Network and Systems Management (JNSM)*. https://www.springer.com/computer/communication+networks/journal/10922.

[46] *Wiley Journal of Software: Practice and Experience (SPE)*. https://onlinelibrary.wiley.com/journal/1097024x.

[47] M. Aazam and E. Huh. *Fog Computing Micro Datacenter Based Dynamic Resource Estimation and Pricing Model for IoT*. In 2015 IEEE 29th International Conference on Advanced Information Networking and Applications, pages 687–694, March 2015. doi:10.1109/AINA.2015.254.

[48] M. Abdelbaky, J. Diaz-Montes, M. Parashar, M. Unuvar, and M. Steinder. *Docker Containers across Multiple Clouds and Data Centers*. In 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC), pages 368–371, Dec 2015. doi:10.1109/UCC.2015.58.

[49] Y. Amannejad, D. Krishnamurthy, and B. Far. *Managing Performance Interference in Cloud-Based Web Services*. IEEE Transactions on Network and Service Management, 12(3):320–333, Sep. 2015. doi:10.1109/TNSM.2015.2456172.

[50] Y. Chiang, Y. Ouyang, and C. . Hsu. *An Efficient Green Control Algorithm in Cloud Computing for Cost Optimization*. IEEE Transactions on Cloud Computing, 3(2):145–155, April 2015. doi:10.1109/TCC.2014.2350492.

[51] M. Dabbagh, B. Hamdaoui, M. Guizani, and A. Rayes. *Energy-Efficient Resource Allocation and Provisioning Framework for Cloud Data Centers*. IEEE Transactions on Network and Service Management, 12(3):377–391, Sep. 2015. doi:10.1109/TNSM.2015.2436408.

[52] S. Dhakate and A. Godbole. *Distributed cloud monitoring using Docker as next generation container virtualization technology*. In 2015 Annual IEEE India Conference (INDICON), pages 1–5, Dec 2015. doi:10.1109/INDICON.2015.7443771.

[53] X. Huang, R. Yu, J. Kang, J. Ding, S. Maharjan, S. Gjessing, and Y. Zhang. *Dynamic Resource Pricing and Scalable Cooperation for Mobile Cloud Computing*. In 2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), pages 786–792, Aug 2015. doi:10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.155.

[54] H. Jin, X. Wang, S. Wu, S. Di, and X. Shi. *Towards Optimized Fine-Grained Pricing of IaaS Cloud Platform*. IEEE Transactions on Cloud Computing, 3(4):436–448, Oct 2015. doi:10.1109/TCC.2014.2344680.

[55] K. Katsalis, G. S. Paschos, Y. Viniotis, and L. Tassiulas. *CPU Provisioning Algorithms for Service Differentiation in Cloud-Based Environments*. IEEE Transactions on Network and Service Management, 12(1):61–74, March 2015. doi:10.1109/TNSM.2015.2397345.

[56] A. G. Kumbhare, Y. Simmhan, M. Frincu, and V. K. Prasanna. *Reactive Resource Provisioning Heuristics for Dynamic Dataflows on Cloud Infrastructure*. IEEE Transactions on Cloud Computing, 3(2):105–118, April 2015. doi:10.1109/TCC.2015.2394316.

[57] Y. C. Lee, Y. Kim, H. Han, and S. Kang. *Fine-Grained, Adaptive Resource Sharing for Real Pay-Per-Use Pricing in Clouds*. In 2015 International Conference on Cloud and Autonomic Computing, pages 236–243, Sep. 2015. doi:10.1109/ICCAC.2015.36.

[58] J. Liu, Y. Zhang, Y. Zhou, D. Zhang, and H. Liu. *Aggressive Resource Provisioning for Ensuring QoS in Virtualized Environments*. IEEE Transactions on Cloud Computing, 3(2):119–131, April 2015. doi:10.1109/TCC.2014.2353045.

[59] H. Moens, B. Dhoedt, and F. D. Turck. *Allocating resources for customizable multi-tenant applications in clouds using dynamic feature placement*. Future Generation Computer Systems, 53:63 – 76, 2015. Available from: http://www.sciencedirect.com/science/article/pii/S0167739X15002010, doi:https://doi.org/10.1016/j.future.2015.05.017.

[60] J. Mukherjee, D. Krishnamurthy, and J. Rolia. *Resource Contention Detection in Virtualized Environments*. IEEE Transactions

on Network and Service Management, 12(2):217–231, June 2015. doi:10.1109/TNSM.2015.2407273.

[61] I. Petri, J. Diaz-Montes, M. Zou, T. Beach, O. Rana, and M. Parashar. *Market Models for Federated Clouds*. IEEE Transactions on Cloud Computing, 3(3):398–410, July 2015. doi:10.1109/TCC.2015.2415792.

[62] B. Sharma, R. K. Thulasiram, P. Thulasiraman, and R. Buyya. *Clabacus: A Risk-Adjusted Cloud Resources Pricing Model Using Financial Option Theory*. IEEE Transactions on Cloud Computing, 3(3):332–344, July 2015. doi:10.1109/TCC.2014.2382099.

[63] V. Stankovski, S. Taherizadeh, I. Taylor, A. Jones, C. Mastroianni, B. Becker, and H. Suhartanto. *Towards an Environment Supporting Resilience, High-Availability, Reproducibility and Reliability for Cloud Applications*. In 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC), pages 383–386, Dec 2015. doi:10.1109/UCC.2015.61.

[64] X. Wang, X. Wang, H. Che, K. Li, M. Huang, and C. Gao. *An Intelligent Economic Approach for Dynamic Resource Allocation in Cloud Services*. IEEE Transactions on Cloud Computing, 3(3):275–289, July 2015. doi:10.1109/TCC.2015.2415776.

[65] F. Wuhib, R. Yanggratoke, and R. Stadler. *Allocating Compute and Network Resources Under Management Objectives in Large-Scale Clouds*. Journal of Network and Systems Management, 23(1):111–136, Jan 2015. Available from: https://doi.org/10.1007/s10922-013-9280-6, doi:10.1007/s10922-013-9280-6.

[66] Q. Zhang, S. Li, Z. Li, Y. Xing, Z. Yang, and Y. Dai. *CHARM: A Cost-Efficient Multi-Cloud Data Hosting Scheme with High Availability*. IEEE Transactions on Cloud Computing, 3(3):372–386, July 2015. doi:10.1109/TCC.2015.2417534.

[67] M. Aazam, E. Huh, M. St-Hilaire, C. Lung, and I. Lambadaris. *Cloud Customer's Historical Record Based Resource Pricing*. IEEE Transactions on Parallel and Distributed Systems, 27(7):1929–1940, July 2016. doi:10.1109/TPDS.2015.2473850.

[68] S. Ayoubi, Y. Zhang, and C. Assi. *A Reliable Embedding Framework for Elastic Virtualized Services in the Cloud*. IEEE Transactions on Network and Service Management, 13(3):489–503, Sep. 2016. doi:10.1109/TNSM.2016.2581484.

[69] S. Choi, R. Myung, H. Choi, K. Chung, J. Gil, and H. Yu. *GPSF: General-Purpose Scheduling Framework for Container Based on Cloud Environment*. In 2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), pages 769–772, Dec 2016. doi:10.1109/iThings-GreenCom-CPSCom-SmartData.2016.162.

[70] G. Da Cunha Rodrigues, R. N. Calheiros, V. T. Guimaraes, G. L. d. Santos, M. B. de Carvalho, L. Z. Granville, L. M. R. Tarouco, and R. Buyya. *Monitoring of Cloud Computing Environments: Concepts, Solutions, Trends, and Future Directions*. In Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC '16, pages 378–383, New York, NY, USA, 2016. ACM. Available from: http://doi.acm.org/10.1145/2851613.2851619, doi:10.1145/2851613.2851619.

[71] X. Dai, J. M. Wang, and B. Bensaou. *Energy-Efficient Virtual Machines Scheduling in Multi-Tenant Data Centers*. IEEE Transactions on Cloud Computing, 4(2):210–221, April 2016. doi:10.1109/TCC.2015.2481401.

[72] K. Elgazzar, P. Martin, and H. S. Hassanein. *Cloud-Assisted Computation Offloading to Support Mobile Services*. IEEE Transactions on Cloud Computing, 4(3):279–292, July 2016. doi:10.1109/TCC.2014.2350471.

[73] D. Espling, L. Larsson, W. Li, J. Tordsson, and E. Elmroth. *Modeling and Placement of Cloud Services with Internal Structure*. IEEE Transactions on Cloud Computing, 4(4):429–439, Oct 2016. doi:10.1109/TCC.2014.2362120.

[74] H. Goudarzi and M. Pedram. *Hierarchical SLA-Driven Resource Management for Peak Power-Aware and Energy-Efficient Operation of a Cloud Datacenter*. IEEE Transactions on Cloud Computing, 4(2):222–236, April 2016. doi:10.1109/TCC.2015.2474369.

[75] Z. Huang and D. H. K. Tsang. *M-Convex VM Consolidation: Towards a Better VM Workload Consolidation*. IEEE Transactions on Cloud Computing, 4(4):415–428, Oct 2016. doi:10.1109/TCC.2014.2369423.

[76] D. Kang, G. Choi, S. Kim, I. Hwang, and C. Youn. *Workload-aware resource management for energy efficient heterogeneous Docker containers*. In 2016 IEEE Region 10 Conference (TENCON), pages 2428–2431, Nov 2016. doi:10.1109/TENCON.2016.7848467.

[77] S. Khatua, P. K. Sur, R. K. Das, and N. Mukherjee. *Heuristic-Based Resource Reservation Strategies for Public Cloud*. IEEE

Transactions on Cloud Computing, 4(4):392–401, Oct 2016. doi:10.1109/TCC.2014.2369434.

[78] M. Mishra and U. Bellur. *Whither Tightness of Packing? The Case for Stable VM Placement*. IEEE Transactions on Cloud Computing, 4(4):481–494, Oct 2016. doi:10.1109/TCC.2014.2378756.

[79] G. Nakagawa and S. Oikawa. *Behavior-Based Memory Resource Management for Container-Based Virtualization*. In 2016 4th Intl Conf on Applied Computing and Information Technology/3rd Intl Conf on Computational Science/Intelligence and Applied Informatics/1st Intl Conf on Big Data, Cloud Computing, Data Science Engineering (ACIT-CSII-BCD), pages 213–217, Dec 2016. doi:10.1109/ACIT-CSII-BCD.2016.049.

[80] M. Pantazoglou, G. Tzortzakis, and A. Delis. *Decentralized and Energy-Efficient Workload Management in Enterprise Clouds*. IEEE Transactions on Cloud Computing, 4(2):196–209, April 2016. doi:10.1109/TCC.2015.2464817.

[81] R. d. R. Righi, V. F. Rodrigues, C. A. da Costa, G. Galante, L. C. E. de Bona, and T. Ferreto. *AutoElastic: Automatic Resource Elasticity for High Performance Applications in the Cloud*. IEEE Transactions on Cloud Computing, 4(1):6–19, Jan 2016. doi:10.1109/TCC.2015.2424876.

[82] K. Salah, K. Elbadawi, and R. Boutaba. *An Analytical Model for Estimating Cloud Resources of Elastic Services*. Journal of Network and Systems Management, 24(2):285–308, Apr 2016. Available from: https://doi.org/10.1007/s10922-015-9352-x, doi:10.1007/s10922-015-9352-x.

[83] U. Wajid, C. Cappiello, P. Plebani, B. Pernici, N. Mehandjiev, M. Vitali, M. Gienger, K. Kavoussanakis, D. Margery, D. G. Perez, and P. Sampaio. *On Achieving Energy Efficiency and Reducing CO lt;sub gt;2 lt;/sub gt; Footprint in Cloud Computing*. IEEE Transactions on Cloud Computing, 4(2):138–151, April 2016. doi:10.1109/TCC.2015.2453988.

[84] J. Wan, R. Zhang, X. Gui, and B. Xu. *Reactive Pricing: An Adaptive Pricing Policy for Cloud Providers to Maximize Profit*. IEEE Transactions on Network and Service Management, 13(4):941–953, Dec 2016. doi:10.1109/TNSM.2016.2618394.

[85] B. Wanis, N. Samaan, and A. Karmouch. *Efficient Modeling and Demand Allocation for Differentiated Cloud Virtual-Network as-a Service Offerings*. IEEE Transactions on Cloud Computing, 4(4):376–391, Oct 2016. doi:10.1109/TCC.2015.2389814.

[86] H. Wu, S. Ren, G. Garzoglio, S. Timm, G. Bernabeu, K. Chadwick, and S. Noh. *A Reference Model for Virtual Machine Launching Overhead*. IEEE Transactions on Cloud Computing, 4(3):250–264, July 2016. doi:10.1109/TCC.2014.2369439.

[87] X. Xu, W. Dou, X. Zhang, and J. Chen. *EnReal: An Energy-Aware Resource Allocation Method for Scientific Workflow Executions in Cloud Environment*. IEEE Transactions on Cloud Computing, 4(2):166–179, April 2016. doi:10.1109/TCC.2015.2453966.

[88] A. Zhou, S. Wang, Z. Zheng, C. Hsu, M. R. Lyu, and F. Yang. *On Cloud Service Reliability Enhancement with Optimal Resource Usage*. IEEE Transactions on Cloud Computing, 4(4):452–466, Oct 2016. doi:10.1109/TCC.2014.2369421.

[89] U. Awada and A. Barker. *Improving Resource Efficiency of Container-Instance Clusters on Clouds*. In 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pages 929–934, May 2017. doi:10.1109/CCGRID.2017.113.

[90] U. Awada and A. Barker. *Resource Efficiency in Container-instance Clusters*. In Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing, ICC '17, pages 181:1–181:5, New York, NY, USA, 2017. ACM. Available from: http://doi.acm.org/10.1145/3018896.3056798, doi:10.1145/3018896.3056798.

[91] M. Babaioff, Y. Mansour, N. Nisan, G. Noti, C. Curino, N. Ganapathy, I. Menache, O. Reingold, M. Tennenholtz, and E. Timnat. *ERA: A Framework for Economic Resource Allocation for the Cloud*. In Proceedings of the 26th International Conference on World Wide Web Companion, WWW '17 Companion, pages 635–642, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee. Available from: https://doi.org/10.1145/3041021.3054186, doi:10.1145/3041021.3054186.

[92] R. Chard, K. Chard, R. Wolski, R. Madduri, B. Ng, K. Bubendorfer, and I. Foster. *Cost-Aware Cloud Profiling, Prediction, and Provisioning as a Service*. IEEE Cloud Computing, 4(4):48–59, July 2017. doi:10.1109/MCC.2017.3791025.

[93] B. L. Dalmazo, J. P. Vilela, and M. Curado. *Performance Analysis of Network Traffic Predictors in the Cloud*. Journal of Network and Systems Management, 25(2):290–320, Apr 2017. Available from: https://doi.org/10.1007/s10922-016-9392-x, doi:10.1007/s10922-016-9392-x.

[94] T. H. Hai and P. Nguyen. *A Pricing Model for Sharing Cloudlets in Mobile Cloud Computing*. In 2017 International Conference on Advanced Computing and Applications (ACOMP), pages 149–153, Nov 2017. doi:10.1109/ACOMP.2017.13.

[95] S. Hoque, M. S. d. Brito, A. Willner, O. Keil, and T. Magedanz. *Towards Container Orchestration in Fog Computing Infrastructures*. In 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), volume 2, pages 294–299, July 2017. doi:10.1109/COMPSAC.2017.248.

[96] X. Jin, F. Zhang, L. Wang, S. Hu, B. Zhou, and Z. Liu. *Joint Optimization of Operational Cost and Performance Interference in Cloud Data Centers*. IEEE Transactions on Cloud Computing, 5(4):697–711, Oct 2017. doi:10.1109/TCC.2015.2449839.

[97] J. N. Khasnabish, M. F. Mithani, and S. Rao. *Tier-Centric Resource Allocation in Multi-Tier Cloud Systems*. IEEE Transactions on Cloud Computing, 5(3):576–589, July 2017. doi:10.1109/TCC.2015.2424888.

[98] J. Li, R. Ma, H. Guan, and D. S. L. Wei. *Accurate CPU Proportional Share and Predictable I/O Responsiveness for Virtual Machine Monitor: A Case Study in Xen*. IEEE Transactions on Cloud Computing, 5(4):604–616, Oct 2017. doi:10.1109/TCC.2015.2441705.

[99] J. Z. Li, M. Woodside, J. Chinneck, and M. Litiou. *Adaptive Cloud Deployment Using Persistence Strategies and Application Awareness*. IEEE Transactions on Cloud Computing, 5(2):277–290, April 2017. doi:10.1109/TCC.2015.2409873.

[100] W. J. Lloyd, S. Pallickara, O. David, M. Arabi, T. Wible, J. Ditty, and K. Rojas. *Demystifying the Clouds: Harnessing Resource Utilization Models for Cost Effective Infrastructure Alternatives*. IEEE Transactions on Cloud Computing, 5(4):667–680, Oct 2017. doi:10.1109/TCC.2015.2430339.

[101] P.-J. Maenhaut, H. Moens, B. Volckaert, V. Ongenae, and F. D. Turck. *A dynamic Tenant-Defined Storage system for efficient resource management in cloud applications*. Journal of Network and Computer Applications, 93:182 – 196, 2017. Available from: http://www.sciencedirect.com/science/article/pii/S1084804517302114, doi:https://doi.org/10.1016/j.jnca.2017.05.014.

[102] A. Mebrek, L. Merghem-Boulahia, and M. Esseghir. *Efficient green solution for a balanced energy consumption and delay in the IoT-Fog-Cloud computing*. In 2017 IEEE 16th International Symposium on

Network Computing and Applications (NCA), pages 1–4, Oct 2017. doi:10.1109/NCA.2017.8171359.

[103] M. Mechtri, M. Hadji, and D. Zeghlache. *Exact and Heuristic Resource Mapping Algorithms for Distributed and Hybrid Clouds*. IEEE Transactions on Cloud Computing, 5(4):681–696, Oct 2017. doi:10.1109/TCC.2015.2427192.

[104] S. Merzoug, O. Kazar, and M. Derdour. *Intelligent Strategy of Allocation Resource for Cloud Datacenter Based on MAS & CP Approach*. In Proceedings of the International Conference on Computing for Engineering and Sciences, ICCES '17, pages 50–55, New York, NY, USA, 2017. ACM. Available from: http://doi.acm.org/10.1145/3129186.3129197, doi:10.1145/3129186.3129197.

[105] S. Mireslami, L. Rakai, B. H. Far, and M. Wang. *Simultaneous Cost and QoS Optimization for Cloud Resource Allocation*. IEEE Transactions on Network and Service Management, 14(3):676–689, Sep. 2017. doi:10.1109/TNSM.2017.2738026.

[106] M. Nardelli, C. Hochreiner, and S. Schulte. *Elastic Provisioning of Virtual Machines for Container Deployment*. In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, ICPE '17 Companion, pages 5–10, New York, NY, USA, 2017. ACM. Available from: http://doi.acm.org/10.1145/3053600.3053602, doi:10.1145/3053600.3053602.

[107] V. Nitu, B. Teabe, L. Fopa, A. Tchana, and D. Hagimont. *StopGap: Elastic VMs to Enhance Server Consolidation*. In Proceedings of the Symposium on Applied Computing, SAC '17, pages 358–363, New York, NY, USA, 2017. ACM. Available from: http://doi.acm.org/10.1145/3019612.3019626, doi:10.1145/3019612.3019626.

[108] A. Paya and D. C. Marinescu. *Energy-Aware Load Balancing and Application Scaling for the Cloud Ecosystem*. IEEE Transactions on Cloud Computing, 5(1):15–27, Jan 2017. doi:10.1109/TCC.2015.2396059.

[109] W. Rankothge, F. Le, A. Russo, and J. Lobo. *Optimizing Resource Allocation for Virtualized Network Functions in a Cloud Center Using Genetic Algorithms*. IEEE Transactions on Network and Service Management, 14(2):343–356, June 2017. doi:10.1109/TNSM.2017.2686979.

[110] L. Tang and H. Chen. *Joint Pricing and Capacity Planning in the IaaS Cloud Market*. IEEE Transactions on Cloud Computing, 5(1):57–70, Jan 2017. doi:10.1109/TCC.2014.2372811.

[111] D. Xu, X. Liu, and Z. Niu. *Joint Resource Provisioning for Internet Datacenters with Diverse and Dynamic Traffic*. IEEE Transactions on Cloud Computing, 5(1):71–84, Jan 2017. doi:10.1109/TCC.2014.2382118.

[112] Y. Yang, X. Chang, J. Liu, and L. Li. *Towards Robust Green Virtual Cloud Data Center Provisioning*. IEEE Transactions on Cloud Computing, 5(2):168–181, April 2017. doi:10.1109/TCC.2015.2459704.

[113] X. Yi, F. Liu, D. Niu, H. Jin, and J. C. S. Lui. *Cocoa: Dynamic Container-Based Group Buying Strategies for Cloud Computing*. ACM Trans. Model. Perform. Eval. Comput. Syst., 2(2):8:1–8:31, February 2017. Available from: http://doi.acm.org/10.1145/3022876, doi:10.1145/3022876.

[114] B. Yu and J. Pan. *Optimize the Server Provisioning and Request Dispatching in Distributed Memory Cache Services*. IEEE Transactions on Cloud Computing, 5(2):193–207, April 2017. doi:10.1109/TCC.2015.2469663.

[115] W. Zhang, H. Xie, and C. Hsu. *Automatic Memory Control of Multiple Virtual Machines on a Consolidated Server*. IEEE Transactions on Cloud Computing, 5(1):2–14, Jan 2017. doi:10.1109/TCC.2014.2378794.

[116] M. Alam, J. Rufino, J. Ferreira, S. H. Ahmed, N. Shah, and Y. Chen. *Orchestration of Microservices for IoT Using Docker and Edge Computing*. IEEE Communications Magazine, 56(9):118–123, Sep. 2018. doi:10.1109/MCOM.2018.1701233.

[117] A. Aral and T. Ovatman. *A Decentralized Replica Placement Algorithm for Edge Computing*. IEEE Transactions on Network and Service Management, 15(2):516–529, June 2018. doi:10.1109/TNSM.2017.2788945.

[118] A. Atrey, G. V. Seghbroeck, B. Volckaert, and F. D. Turck. *BRAHMA+: A Framework for Resource Scaling of Streaming and ASAP Time-Varying Workflows*. IEEE Transactions on Network and Service Management, 15(3):894–908, Sep. 2018. doi:10.1109/TNSM.2018.2830311.

[119] A. Barkat, M.-T. Kechadi, G. Verticale, I. Filippini, and A. Capone. *Green Approach for Joint Management of Geo-Distributed Data Centers and Interconnection Networks*. Journal of Network and Systems Management, 26(3):723–754, Jul 2018. Available from: https://doi.org/10.1007/s10922-017-9441-0, doi:10.1007/s10922-017-9441-0.

[120] C. Balos, D. D. L. Vega, Z. Abuelhaj, C. Kari, D. Mueller, and V. K. Pallipuram. *A2Cloud: An Analytical Model for Application-to-Cloud Matching to Empower Scientific Computing*. In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pages 548–555, July 2018. doi:10.1109/CLOUD.2018.00076.

[121] J. Barrameda and N. Samaan. *A Novel Statistical Cost Model and an Algorithm for Efficient Application Offloading to Clouds*. IEEE Transactions on Cloud Computing, 6(3):598–611, July 2018. doi:10.1109/TCC.2015.2513404.

[122] W. Borjigin, K. Ota, and M. Dong. *In Broker We Trust: A Double-Auction Approach for Resource Allocation in NFV Markets*. IEEE Transactions on Network and Service Management, 15(4):1322–1333, Dec 2018. doi:10.1109/TNSM.2018.2882535.

[123] M. Bouet and V. Conan. *Mobile Edge Computing Resources Optimization: A Geo-Clustering Approach*. IEEE Transactions on Network and Service Management, 15(2):787–796, June 2018. doi:10.1109/TNSM.2018.2816263.

[124] M. Cheng, J. Li, and S. Nazarian. *DRL-cloud: Deep reinforcement learning-based resource provisioning and task scheduling for cloud service providers*. In 2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC), pages 129–134, Jan 2018. doi:10.1109/ASPDAC.2018.8297294.

[125] J. Diaz-Montes, M. Diaz-Granados, M. Zou, S. Tao, and M. Parashar. *Supporting Data-Intensive Workflows in Software-Defined Federated Multi-Clouds*. IEEE Transactions on Cloud Computing, 6(1):250–263, Jan 2018. doi:10.1109/TCC.2015.2481410.

[126] S. S. Gill, R. Buyya, I. Chana, M. Singh, and A. Abraham. *BULLET: Particle Swarm Optimization Based Scheduling Technique for Provisioned Cloud Resources*. Journal of Network and Systems Management, 26(2):361–400, Apr 2018. Available from: https://doi.org/10.1007/s10922-017-9419-y, doi:10.1007/s10922-017-9419-y.

[127] T. Guo and P. Shenoy. *Providing Geo-Elasticity in Geographically Distributed Clouds*. ACM Trans. Internet Technol., 18(3):38:1–38:27, April 2018. Available from: http://doi.acm.org/10.1145/3169794, doi:10.1145/3169794.

[128] W. Guo, B. Lin, G. Chen, Y. Chen, and F. Liang. *Cost-Driven Scheduling for Deadline-Based Workflow Across Multiple Clouds*. IEEE Transactions on Network and Service Management, 15(4):1571–1585, Dec 2018. doi:10.1109/TNSM.2018.2872066.

[129] Y. Guo, A. L. Stolyar, and A. Walid. *Shadow-Routing Based Dynamic Algorithms for Virtual Machine Placement in a Network Cloud*.

IEEE Transactions on Cloud Computing, 6(1):209–220, Jan 2018. doi:10.1109/TCC.2015.2464795.

[130] C. B. Hauser and S. Wesner. *Reviewing Cloud Monitoring: Towards Cloud Resource Profiling*. In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pages 678–685, July 2018. doi:10.1109/CLOUD.2018.00093.

[131] S. Heidari and R. Buyya. *Cost-efficient and network-aware dynamic repartitioning-based algorithms for scheduling large-scale graphs in cloud computing environments*. Software: Practice and Experience, 48(12):2174–2192, 2018. Available from: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2623, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2623, doi:10.1002/spe.2623.

[132] B. Jia, H. Hu, Y. Zeng, T. Xu, and Y. Yang. *Double-matching resource allocation strategy in fog computing networks based on cost efficiency*. Journal of Communications and Networks, 20(3):237–246, June 2018. doi:10.1109/JCN.2018.000036.

[133] G. Jia, G. Han, J. Jiang, S. Chan, and Y. Liu. *Dynamic cloud resource management for efficient media applications in mobile computing environments*. Personal and Ubiquitous Computing, 22(3):561–573, Jun 2018. Available from: https://doi.org/10.1007/s00779-018-1118-5, doi:10.1007/s00779-018-1118-5.

[134] M. Khabbaz and C. M. Assi. *Modelling and Analysis of A Novel Deadline-Aware Scheduling Scheme for Cloud Computing Data Centers*. IEEE Transactions on Cloud Computing, 6(1):141–155, Jan 2018. doi:10.1109/TCC.2015.2481429.

[135] G. Lahmann, T. McCann, and W. Lloyd. *Container Memory Allocation Discrepancies: An Investigation on Memory Utilization Gaps for Container-Based Application Deployments*. In 2018 IEEE International Conference on Cloud Engineering (IC2E), pages 404–405, April 2018. doi:10.1109/IC2E.2018.00076.

[136] Y. Lin, Y. Lai, J. Huang, and H. Chien. *Three-Tier Capacity and Traffic Allocation for Core, Edges, and Devices for Mobile Edge Computing*. IEEE Transactions on Network and Service Management, 15(3):923–933, Sep. 2018. doi:10.1109/TNSM.2018.2852643.

[137] P. Nawrocki and B. Sniezynski. *Adaptive Service Management in Mobile Cloud Computing by Means of Supervised and Reinforcement*

Learning. Journal of Network and Systems Management, 26(1):1–22, Jan 2018. Available from: https://doi.org/10.1007/s10922-017-9405-4, doi:10.1007/s10922-017-9405-4.

[138] D. B. Prats, J. L. Berral, and D. Carrera. *Automatic Generation of Workload Profiles Using Unsupervised Learning Pipelines*. IEEE Transactions on Network and Service Management, 15(1):142–155, March 2018. doi:10.1109/TNSM.2017.2786047.

[139] M. R. Rahimi, N. Venkatasubramanian, S. Mehrotra, and A. V. Vasilakos. *On Optimal and Fair Service Allocation in Mobile Cloud Computing*. IEEE Transactions on Cloud Computing, 6(3):815–828, July 2018. doi:10.1109/TCC.2015.2511729.

[140] J. Sahni and D. P. Vidyarthi. *A Cost-Effective Deadline-Constrained Dynamic Scheduling Algorithm for Scientific Workflows in a Cloud Environment*. IEEE Transactions on Cloud Computing, 6(1):2–18, Jan 2018. doi:10.1109/TCC.2015.2451649.

[141] J. Scheuner and P. Leitner. *Estimating Cloud Application Performance Based on Micro-Benchmark Profiling*. In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pages 90–97, July 2018. doi:10.1109/CLOUD.2018.00019.

[142] I. Simonis. *Container-based Architecture to Optimize the Integration of Microservices into Cloud-based Data-intensive Application Scenarios*. In Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings, ECSA '18, pages 34:1–34:3, New York, NY, USA, 2018. ACM. Available from: http://doi.acm.org/10.1145/3241403.3241439, doi:10.1145/3241403.3241439.

[143] A. Sathya Sofia and P. GaneshKumar. *Multi-objective Task Scheduling to Minimize Energy Consumption and Makespan of Cloud Computing Using NSGA-II*. Journal of Network and Systems Management, 26(2):463–485, Apr 2018. Available from: https://doi.org/10.1007/s10922-017-9425-0, doi:10.1007/s10922-017-9425-0.

[144] K. Takahashi, K. Aida, T. Tanjo, and J. Sun. *A Portable Load Balancer for Kubernetes Cluster*. In Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2018, pages 222–231, New York, NY, USA, 2018. ACM. Available from: http://doi.acm.org/10.1145/3149457.3149473, doi:10.1145/3149457.3149473.

[145] D. Trihinas, G. Pallis, and M. D. Dikaiakos. *Monitoring Elastically Adaptive Multi-Cloud Services*. IEEE Transactions on Cloud Computing, 6(3):800–814, July 2018. doi:10.1109/TCC.2015.2511760.

[146] L. Wang and E. Gelenbe. *Adaptive Dispatching of Tasks in the Cloud*. IEEE Transactions on Cloud Computing, 6(1):33–45, Jan 2018. doi:10.1109/TCC.2015.2474406.

[147] L. Wei, C. H. Foh, B. He, and J. Cai. *Towards Efficient Resource Allocation for Heterogeneous Workloads in IaaS Clouds*. IEEE Transactions on Cloud Computing, 6(1):264–275, Jan 2018. doi:10.1109/TCC.2015.2481400.

[148] R. Xie and X. Jia. *Data Transfer Scheduling for Maximizing Throughput of Big-Data Computing in Cloud Systems*. IEEE Transactions on Cloud Computing, 6(1):87–98, Jan 2018. doi:10.1109/TCC.2015.2464808.

[149] W. Zhang and Y. Wen. *Energy-Efficient Task Execution for Application as a General Topology in Mobile Cloud Computing*. IEEE Transactions on Cloud Computing, 6(3):708–719, July 2018. doi:10.1109/TCC.2015.2511727.

[150] Y. Zhang, A. Ghosh, V. Aggarwal, and T. Lan. *Tiered cloud storage via two-stage, latency-aware bidding*. IEEE Transactions on Network and Service Management, pages 1–1, 2018. doi:10.1109/TNSM.2018.2875475.

[151] *Introducing Amazon EC2 Spot Instances for Specific Duration Workloads*. https://aws.amazon.com/about-aws/whats-new/2015/10/introducing-amazon-ec2-spot-instances-for-specific-duration-workloads/.

[152] X. Masip-Bruin, E. Marín-Tordera, A. Juan-Ferrer, A. Queralt, A. Jukan, J. Garcia, D. Lezzi, J. Jensen, C. Cordeiro, A. Leckey, A. Salis, D. Guilhot, and M. Cankar. *mF2C: Towards a Coordinated Management of the IoT-fog-cloud Continuum*. In Proceedings of the 4th ACM MobiHoc Workshop on Experiences with the Design and Implementation of Smart Objects, SMARTOBJECTS '18, pages 8:1–8:8, New York, NY, USA, 2018. ACM. Available from: http://doi.acm.org/10.1145/3213299.3213307, doi:10.1145/3213299.3213307.

[153] A. Almutairi, M. I. Sarfraz, and A. Ghafoor. *Risk-Aware Management of Virtual Resources in Access Controlled Service-Oriented Cloud Datacenters*. IEEE Transactions on Cloud Computing, 6(1):168–181, Jan 2018. doi:10.1109/TCC.2015.2453981.

[154] Y. Zhai, L. Yin, J. Chase, T. Ristenpart, and M. Swift. *CQSTR: Securing Cross-Tenant Applications with Cloud Containers*. In Proceedings of the

Seventh ACM Symposium on Cloud Computing, SoCC '16, pages 223–236, New York, NY, USA, 2016. ACM. Available from: http://doi.acm.org/10.1145/2987550.2987558, doi:10.1145/2987550.2987558.

[155] S. Lins, S. Schneider, and A. Sunyaev. *Trust is Good, Control is Better: Creating Secure Clouds by Continuous Auditing*. IEEE Transactions on Cloud Computing, 6(3):890–903, July 2018. doi:10.1109/TCC.2016.2522411.

[156] P.-J. Maenhaut, B. Volckaert, V. Ongenae, and F. De Turck. *Efficient resource management in the cloud: From simulation to experimental validation using a low-cost Raspberry Pi testbed*. Software: Practice and Experience, 49(3):449–477, 2019. Available from: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2669, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2669, doi:10.1002/spe.2669.

[157] R. Calheiros, R. Ranjan, A. Beloglazov, C. De Rose, and R. Buyya. *CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms*. Software: Practice and Experience, 41(1):23–50, 2011. https://doi.org/10.1002/spe.995. doi:10.1002/spe.995.

6
Conclusions and Future Work

"There's no way that [1] *company exists in a year."*

–Tom Siebel, Founder of Siebel CRM Systems (2001)

In this dissertation, several approaches have been proposed for the efficient management of resources in a multi-tenant cloud environment, from both the perspective of the applications deployed on top of the cloud and from the infrastructure perspective. A cloud environment offers a near infinite amount of resources, and applications can benefit from this by scaling up or down to support the current demand. However, as cloud users are typically charged based on the amount of allocated resources, an efficient usage of the available cloud resources is highly recommended to avoid unnecessary rental costs. For the cloud provider, the main challenge is to determine a feasible allocation of the requested resources over the physical hardware located within the data centers, minimizing the amount of physical hardware required to reduce the operational costs while still guaranteeing the objectives described in so-called Service Level Agreements (SLAs).

This chapter reviews the challenges addressed in this dissertation and provides a brief summary of future work. To conclude, it outlines several research directions currently arising in the field of cloud computing, which we believe will be relevant in future years.

[1] That company is none other than SalesForce.com, which is today one of the biggest CRM platforms, and is one of the best known examples of Software as a Service (SaaS). Siebel itself however no longer exists as a company, as it was bought by Oracle Corporation in September 2005 and is now a brand name owned by Oracle Corporation.

6.1 Review of the Addressed Challenges

Challenge #1: Design and deployment of applications in a multi-tenant cloud environment. The deployment of applications inside cloud environments is slightly different from deployments within traditional data centers. The cloud environment could introduce several limitations, for example regarding the supported libraries and runtime systems. Deployment in the cloud however can help applications to achieve a higher scalability, while reducing the hosting costs. This introduces the need for an efficient management of the allocated resources by the deployed applications.

Chapter 2 introduced a generic approach for migrating legacy software to the public cloud, and for incorporating support for multi-tenancy in the application. Migrating an application to the public cloud only requires a limited number of changes, while the conversion from a single-tenant to a multi-tenant application typically requires more steps as the latter requires limited changes to the application architecture. The presented approach is proactive, as it includes identifying and eliminating possible future risks, for example by mitigating security risks and analyzing the architecture regarding its scalability. If the application is designed using a Service-Oriented Architecture (SOA), select application components can be individually scaled up or down, therefore allowing for a fine-grained scalability. Adding multi-tenancy to the application on the other hand enables the serving of multiple consumers by a single application instance. A combination of both will therefore maximize the utilization of the available cloud resources, while minimizing the costs. Migrating legacy software to the cloud comes at a cost, and some application components may need to be modified or rewritten. However, by following the multi-step migration approach presented in Chapter 2, the long-term benefits of a cloud migration can easily outweigh the costs of implementing the described changes.

Challenge #2: Achieving high scalability and a high level of resource utilization while guaranteeing tenant isolation. When the demand for a given cloud application changes, additional application instances might need to be provisioned. However, multiple applications belonging to different cloud users are typically deployed within a single cloud environment, and a single application should not impact the performance of other applications deployed on top of the same physical hardware. Apart from this, in a multi-tenant environment, tenants should not be able to access the data and applications belonging to other tenants. In summary, both the multi-tenant application and the multi-tenant cloud environment should guarantee data and performance isolation between tenants.

Chapter 3 presented the design and implementation of a scalable system for the allocation of storage resources in a multi-tenant environment. The focus is

on the allocation of storage, as this type of computational resources introduces additional challenges. The most important challenge is minimizing the migration of storage over time. As the number of applications and the amount of data for individual applications increases over time, some of the existing data might need to be reallocated, but this can quickly become both costly and time consuming. In the presented approach, tenants are hierarchically structured, and a dynamic resource allocation algorithm is used to determine a feasible allocation of the tenant data over a set of storage resources. The hierarchical structure enables the allocation of data from the tenant's perspective, guaranteeing a clear isolation of tenants, and taking custom tenant characteristics into account. Two algorithms are proposed for the static allocation, based on the fast and lightweight First-Fit Decreasing (FFD) heuristic, but adapted for managing the allocation of items with a hierarchical structure. Furthermore, a dynamic variant for both algorithms is introduced, which can be used for the dynamic (re-)allocation over time. Although both algorithms are very similar, they achieve very different results regarding the static and dynamic allocation of tenant data. The dHFFD strategy achieves a high average utilization of the provisioned storage instances, and the overhead in number of required instances is small compared to the optimal solution. Furthermore, migrations over time are nicely spread. The dHGD algorithm on the other hand reduces the number of migrations significantly, making it a valid alternative for real-time scenarios with stringent performance constraints, but this comes at the price of a lower average resource utilization and therefore higher operational costs as more storage resources need to be provisioned.

Challenge #3: Cost-effective and reproducible validation of resource management approaches. When designing a new strategy for resource management in a cloud environment, an important challenge is the validation of the approach in practice. Experimental evaluation on a physical cloud testbed is both costly and time-consuming, and many cloud computing environments also have some important limitations, as cloud users rarely have full control over the underlying hardware resources. Simulations on the other hand can be effectively used as a prototyping mechanism to provide a rough idea of how a particular algorithm may perform, but it is very difficult to verify if the simulation environment is an accurate representation of a real world data center environment.

Chapter 4 presented a general approach for the validation of cloud resource allocation strategies, and illustrated the importance of experimental validation on physical testbeds. While simulations can be used for the validation of large-scale scenarios, small-scale experiments on a physical testbed can be used to accurately fine-tune the configurable parameters and can often lead to new insights. Chapter 4 also introduced the design and implementation of Raspberry Pi as a Service (RPiaaS), a low-cost embedded cloud testbed which was built using Rasp-

berry Pi nodes. This testbed facilitates the step towards experimental validation on a large-scale testbed, and proved to be especially useful for the development and initial evaluation of novel cloud resource allocation strategies. As a proof of concept, the storage system of Chapter 3 was implemented on top of the RPiaaS testbed, and the obtained evaluation results were compared to the results obtained using a custom developed simulation tool. Although the results from both evaluations were very similar, the experiments executed on the RPiaaS testbed allowed for measurement of other metrics, such as the actual migration times, and also offered new insights regarding the proposed migration strategy.

Challenge #4: Adapt to recent evolutions within cloud computing. A recent trend within cloud computing is the uprise of new types of clouds, such as edge and fog computing. Apart from this, new virtualization technologies are gaining popularity, such as the use of containers as a lightweight alternative to Virtual Machines (VMs). Containers have a minimal overhead compared to traditional virtual machines and offer a great portability, making them a good alternative for dynamic scenarios with a high number of reconfigurations over time. The question arises how these new evolutions impact existing resource management solutions.

Chapter 5 provided an overview of the current state of the art regarding resource management within the broad sense of cloud computing, complementary to existing surveys in literature. This chapter especially investigated how recent research is adapting to the newly introduced evolutions, being the introduction of new deployment models and the use of containers. Although the majority of research is still focusing on the management of VMs within a traditional cloud environment, several interesting opportunities were identified for resource management in a future fully containerized multi-tiered edge-fog-cloud environment. One of the main issues with VM-based environments is the high cost of VM migrations, and therefore the amount of VM reconfigurations over time should be minimized. Containers however are more lightweight and offer great portability, so reconfigurations over time are less of an issue, but containers typically offer a lower level of isolation. A hybrid solution however, in which containers are deployed inside VMs, could combine the advantages of both technologies. Within the context of edge and fog computing, less powerful devices can transfer computational intensive tasks to another environment. This requires an offloading approach, that could for example focus on energy efficiency or minimizing the operational costs. Apart from this, in a multi-tiered environment, an application or individual application components should be deployed in the optimal environment, for example to balance the load or to minimize the operational costs.

6.2 Future Work

The approach presented in Chapter 2 for both migrating applications to the cloud, and for incorporating support for multi-tenancy is mainly focusing on traditional cloud environments, in which application components are deployed inside cloud VM instances. As container technology is gaining popularity, the presented approach could be adapted for the migration of applications to a containerized cloud environment. In this context, a generic strategy for converting legacy applications to a service-oriented architecture can be developed, which allows for the deployment of individual services inside containers, and can further maximize the utilization of the available cloud resources.

Chapter 3 introduced several heuristics for the hierarchical bin packing problem, mainly designed for the allocation and provisioning of storage resources. The presented algorithms use a homogeneous set of bins (resource pools), in which the size of each bin is identical. As future work, the algorithms could be extended to support a heterogeneous set of bins, allowing bins of different size. A straightforward strategy could be to sort the set of available (free) bins in increasing order, and to select the smallest possible bin when instantiating a new bin, or a more advanced strategy could be developed that either focuses on increasing the average bin utilization or maximizing the tenant isolation, or a combination of both. Furthermore, the developed algorithms are one-dimensional bin packing algorithms, as they take into account a single type of resources. As future work, the algorithms can be extended to multi-dimensional bin packing algorithms. There already are several approximation algorithms available for the multi-dimensional bin packing problem [1–3], but these might need to be adapted to allow for packing of items with a hierarchical structure.

Chapter 4 introduced the design and implementation of RPiaaS, a low-cost embedded cloud testbed for the validation of resource management approaches. RPiaaS provides a rich toolset for both the deployment of experiments and the monitoring of relevant metrics such as the CPU, memory and disk utilization. Recently, we started integrating low-cost SDN switches into the testbed, and as future work the available toolset can be extended to support the automatic configuration and monitoring of the network aspects. By defining data flows between the worker nodes, the bandwidth between clusters can for example be limited to allow for more realistic scenarios. In the near future, we also plan to use the testbed for SDN-based experimental research, such as the autonomous management of flows in a heterogeneous network environment, consisting of multiple distinct SDN controllers.

6.3 Future Perspectives for Cloud Resource Management

Efficient resource management in a cloud environment is essential, for both the applications deployed on top of this cloud environment as for the underlying physical infrastructure. From an infrastructure point of view, efficient resource management can help to achieve higher scalability, as more cloud users can be served by the same amount of hardware, and can also lead to lower operational costs, as unused hardware can be put in standby or even powered off. As cloud users are typically charged based on the actual resource usage, applications deployed on top of the cloud should efficiently use the allocated amount of resources. Chapter 5 provided an overview of the current state of the art regarding resource management in cloud environment, and already introduced several challenges and opportunities. In the remainder of this section, some final research directions are briefly discussed regarding resource management in a future cloud.

6.3.1 Lightweight VMs, containers, or a combination of both

Traditionally, cloud computing is built using hardware-level virtualization, meaning that VMs are provisioned and managed by a hypervisor. VMs introduce a noticeable overhead, as they emulate hardware and run a full stack operating system, and as a result they require a fair amount of resources. Recently, containers are gaining popularity as an alternative virtualization technology. Compared to VMs, containers have a much smaller overhead, and are typically much smaller in size as they are executed directly on the operating system kernel. Containers however offer a lower level of isolation and can impose some security risks, which could prevent the wide adoption of containers within cloud computing. Meta-containers are similar to containers, but are embedded with extra components that allows for reasoning and control of containers [4]. Meta-containers can already solve some of the issues containers are facing, but they will be as powerful as VMs in terms of isolation. However, a combination of both technologies is also possible, by deploying containers inside VMs. While this could combine the advantages of both technologies, a downside of this approach is that actions taken by the hypervisor can have unpredictable and non-deterministic effects on the nested containers [5]. Deploying containers inside VMs would facilitate dynamic reconfigurations over time, as there is no longer a need for costly VM migrations but instead the nested containers can be migrated to a different VM running the same container engine. Therefore, deploying containers inside VMs currently seems to be the best option, and it will be interesting to see how this evolves in the near future.

6.3.2 Cloud-centric or edge-centric

In a multi-tiered cloud environment, multiple fog and edge environments can collaborate with a centrally hosted traditional cloud environment for offloading of computational intensive tasks. Fog and edge environments offer low latency towards the end devices, whereas the central cloud has a significant larger computational capacity. Applications deployed in a multi-tiered cloud environment environment can be either cloud-centric or edge-centric. With cloud-centric computing, the main functionality is executed in the central cloud environment, and the edge or fog environment is mainly used to reduce the load towards the cloud and the latency towards the end devices, for example by implementing caching or by pre-processing and filtering the data obtained from the edge devices before sending it to the centrally hosted cloud. With edge-centric computing on the other hand, the applications, data and services are moved away from the central cloud to the periphery of the network [6]. There is still a central cloud environment, but this environment is mainly used to support the edge or fog environment, for example for the execution of computational intensive tasks, or to support the fog or edge environment when the demand for computing capacity spikes (also referred to as cloud bursting). An edge-centric approach could bring many advantages, as the data and application logic are located in a trusted environment close to the end users and devices, but also introduces several challenges as the data and application logic are now distributed, and different edge and fog environments might need to collaborate. However, when cloud computing was introduced, some main issues preventing many enterprises to migrate to the cloud were that they either did not trust this central cloud for storing their data, they wanted to avoid vendor lock-in, or a migration to the public cloud was just not possible because of legal implications (e.g. some countries explicitly state that some sensitive data needs to be stored within the country, which is not possible with public cloud computing). Edge-centric computing could solve these issues, by only storing data at the edge of the network, but this paradigm is not yet largely adopted. However, as edge devices are getting more powerful, edge-centric computing could definitely gain popularity over the next years.

6.3.3 Towards serverless cloud computing

Although containers are gaining popularity, cloud computing is still mainly built around the provisioning of VMs, and cloud users are typically charged based on the number of provisioned VMs. A majority of applications however struggle to fully utilize the allocated amount of resources, leading to a waste of unused resources [7]. A fine-grained pricing model could tackle this issue, presenting an interesting opportunity for the deployment of applications inside containers. This is also one of the main ideas behind serverless computing [8]. Serverless com-

puting is an event-driven cloud execution model, in which the cloud user provides the code and the cloud provider manages the life-cycle of the execution environment of that code. Cloud users are then charged based on the actual amount of resources consumed by an application, rather than on pre-purchased units of capacity. Serverless computing could facilitate cloud deployments, as the cloud user no longer needs to deploy and manage several cloud instances, and could also offer economic advantages especially for the execution of small, short jobs. Furthermore, containers could play an important role in the evolution of serverless computing, as they can be deployed easily and fast and introduce minimal overhead. Therefore, serverless computing could become more adopted in the near future, and it could also facilitate the step towards cloud computing for a broader audience.

References

[1] L. T. Kou and G. Markowsky. *Multidimensional Bin Packing Algorithms*. IBM Journal of Research and Development, 21(5):443–448, Sep. 1977. doi:10.1147/rd.215.0443.

[2] G. Perboli, T. G. Crainic, and R. Tadei. *An efficient metaheuristic for multi-dimensional multi-container packing*. In 2011 IEEE International Conference on Automation Science and Engineering, pages 563–568, Aug 2011. doi:10.1109/CASE.2011.6042476.

[3] N. Bansal, A. Caprara, and M. Sviridenko. *Improved approximation algorithms for multidimensional bin packing problems*. In 2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06), pages 697–708, Oct 2006. doi:10.1109/FOCS.2006.38.

[4] E. Eberbach and A. Reuter. *Toward El Dorado for Cloud Computing: Lightweight VMs, Containers, Meta-Containers and Oracles*. In Proceedings of the 2015 European Conference on Software Architecture Workshops, ECSAW '15, pages 13:1–13:7, New York, NY, USA, 2015. ACM. Available from: http://doi.acm.org/10.1145/2797433.2797446, doi:10.1145/2797433.2797446.

[5] C. Prakash, P. Prashanth, U. Bellur, and P. Kulkarni. *Deterministic Container Resource Management in Derivative Clouds*. In 2018 IEEE International Conference on Cloud Engineering (IC2E), pages 79–89, April 2018. doi:10.1109/IC2E.2018.00030.

[6] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere. *Edge-centric Computing: Vision and Challenges*. SIGCOMM Comput. Commun. Rev., 45(5):37–42, September 2015. Available from: http://doi.acm.org/10.1145/2831347.2831354, doi:10.1145/2831347.2831354.

[7] H. Jin, X. Wang, S. Wu, S. Di, and X. Shi. *Towards Optimized Fine-Grained Pricing of IaaS Cloud Platform*. IEEE Transactions on Cloud Computing, 3(4):436–448, Oct 2015. doi:10.1109/TCC.2014.2344680.

[8] A. Eivy. *Be Wary of the Economics of "Serverless" Cloud Computing*. IEEE Cloud Computing, 4(2):6–12, March 2017. doi:10.1109/MCC.2017.32.

A

Characterizing the Performance of Tenant Data Management in Multi-Tenant Cloud Authorization Systems

This appendix extends the work presented in Chapter 3, but focuses on the distribution of database records over multiple relational database instances. When the amount of tenant data is too large to fit a single database, the data records might need to be distributed over multiple instances. Tables inside relational databases however can have indexes defined to speed up search operations. In this appendix, we characterize the impact on the performance for such operations, both theoretically and experimentally, for scenarios in which data records are divided over multiple separated instances. Furthermore, we introduce three models for load-balancing and multiple search approaches to efficiently locate the required data.

⋆ ⋆ ⋆

P.-J. Maenhaut, H. Moens, M. Decat, J. Bogaerts, B. Lagaisse, W. Joosen, V. Ongenae and F. De Turck

Published in proceedings of the 2014 IEEE Network Operations and Management Symposium (NOMS 2014), pages 1–8, 2014.

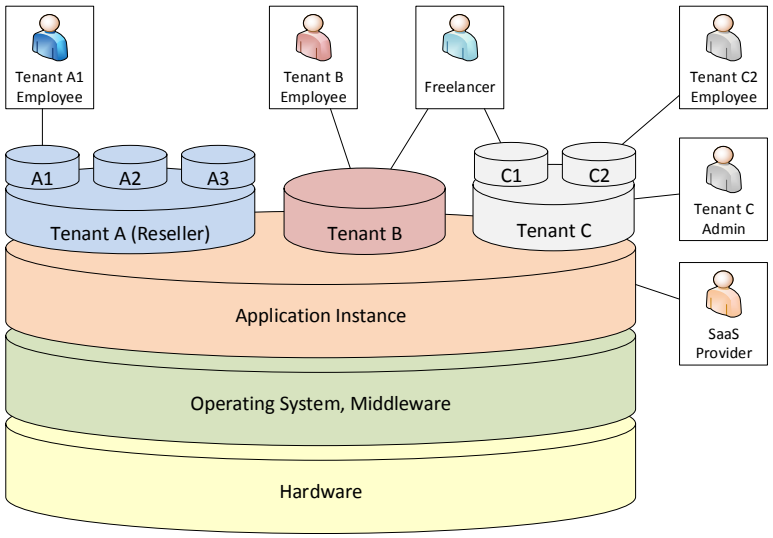


Figure A.1: In a multi-tenant application, most of the software stacks up until the application itself, which is shared by the different tenants.

A.1 Introduction

Multi-tenancy [1] enables the serving of multiple clients or tenants by a single application instance, with isolation of each tenant's data. The major benefits include increased utilisation of available hardware resources and improved ease of maintenance and deployment. These benefits can result in lower overall application costs. In a multi-tenant architecture, a software application is designed to virtually partition its data and configuration, as illustrated in Figure A.1, and each tenant works in a virtual application instance. Within the application, every tenant will typically have its own users and administrators. Some tenants may be divided into multiple subtenants, each one again having its own users. A reseller for example is a special tenant, serving multiple customers, its subtenants. The PUMA project [2] aims to develop a scalable security solution for the management and enforcement of user permissions for Software as a Service (SaaS) applications in a shared (multi-tenant) infrastructure. This solution offers support for essential security requirements, such as confidentiality, integrity and availability.

With the recent evolution of cloud computing [3], a technology that enables elastic, on-demand resource provisioning, and SaaS in particular, a multi-tenant architecture has gained popularity. With cloud computing, an optimal usage of available resources is recommended to reduce operating costs, as the infrastructure

provider usually charges for the number of instances used. As the number of tenants grows, a scalable architecture for authentication and authorization is needed. While most users belong to a single tenant (or subtenant), some users might belong to multiple tenants, which introduces extra challenges for a multi-tenant access control system. Examples include a custom tenant administrator, who is responsible for multiple (but not all) subtenants, and a freelancer who works for different tenants.

Performance is a key challenge in multi-tenant environments, because multiple tenants share the same resources and hardware utilisation is higher on average, and one tenant might clog up resources, compromising the performance of all other tenants. Scalability is another big challenge, especially when the number of tenants increases [4].

In this appendix we focus on the scalability and load-balancing of the storage component of multi-tenant applications, in particular the access control system. We present a hierarchical data management approach, taking performance metrics into account, for structuring the different tenants and subtenants. Different physical implementations are possible, and we will shortly describe the advantages and disadvantages of each alternative. We characterize the impact on the performance both theoretically and experimentally.

We will address these three research questions: (i) How to store, load balance and find tenant data in large multi-tenant environments with minimal overhead? (ii) How does the proposed model impact the performance of the application? (iii) How do tenants impact each other's performance?

In the next section we will discuss related work. Afterwards, in Section A.3, we will present a hierarchical model for managing and storing data, users and roles. We will discuss how specific data can be searched, and provide a theoretical analysis of the impact on the performance in Section A.4. In Section A.5, we will verify our theoretical analysis by different experiments. In Section A.6, we finish with our conclusions and future work.

A.2 Related Work

In previous work [5], we described the steps required to migrate an existing application to a public cloud environment, and proposed a solution to add multi-tenancy to the application. We focused on the use case of a medical communications application. In this appendix, we elaborate on the concept of management and storage of users and roles in a multi-tenant environment, and focus on the performance and load distribution of the access control system.

Related to the PUMA project, the work of Decat et al. [6] [7] is complementary to this appendix. They focus on scalable and confidentiality-aware access control management for SaaS applications from the point of view of the tenant. To

achieve this, they describe and evaluate the concept of *federated authorization* in which authorization is externalized from the SaaS application and centralized at the tenant [6]. To improve performance, they also describe a policy decomposition algorithm for more fine-grained policy deployment [7]. In this appendix, we focus on the performance of the storage part of the application.

Calero et al [8] describe an authorization model suitable for cloud computing in which hierarchical role-based access control, path-based object hierarchies and federation are supported. The model described can be used to implement authentication and roles in a multi-tenancy environment, but no details are added about where to store the users and roles, especially in large, scalable environments. By contrast, we focus on how to divide the users over different datasets, and how this will influence the performance of the authentication mechanism.

In [9], the design of the Force.com multi-tenant internet application development platform is described. The storage uses a set of metadata, data and pivot tables to store all tenant data generically. Typically, a single database is used for every tenant. The paper presents a very generic way for storing custom objects and custom data, which could be used to store the tenant's data, but doesn't really focus on the scalability. By contrast, we present a scalable model where multiple tenants can share a single database, and characterize the performance of the model.

In [10] a solution for access control in cloud environments is presented. Access policies based on data attributes are used to enforce authorization. Such could be used to encrypt the tenants data, combined with the hierarchical model presented in this appendix.

Walraven et al [11] described an architecture of a multi-tenancy enablement layer, which can be used for data isolation, feature management and tenant-specific customizations. This layer could be extended with the hierarchical model presented in this appendix to increase scalability and performance, for building a middleware for highly scalable multi-tenant applications.

A.3 Architecture Outline

Web applications are usually designed using a multitier architecture, where the application is separated into multiple layers, as illustrated in Figure A.2. Within the business logic layer, security is provided by the access control component. This component should be decoupled from the offered services as much as possible. The database layer holds the application data, the different users and (if applicable) roles. As an alternative for roles, Attribute-Based Access Control (ABAC) [12] could also be used. The reasoning behind ABAC is that every user, resource and action can have certain attributes related to them on which policies can define restrictions.

In this section, we focus on the scalability of the database layer in multi-tenant

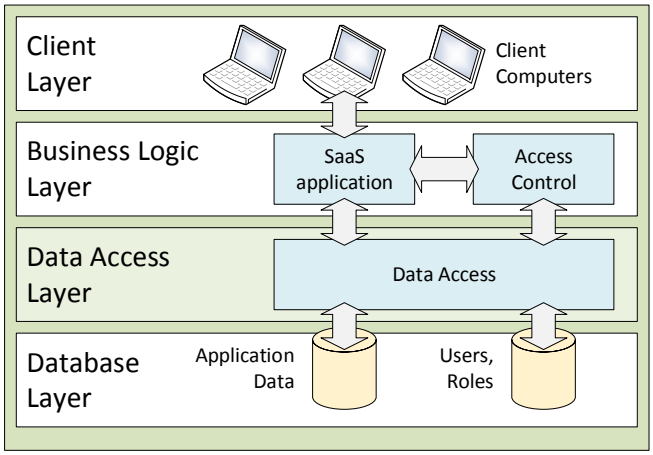


Figure A.2: Layered architecture with decoupled access control. The data access component is responsible for selecting the correct datastore.

applications. We will introduce a logical hierarchical representation of the different tenants and subtenants, and make a mapping to the physical storage. The data access layer is responsible for load balancing between the different datastores and holds the decision support for splitting or merging datastores when applicable.

A.3.1 Logical Representation

Tenants and subtenants can be structured hierarchically. In the rest of this appendix, we will refer to this representation as the tenant tree. At the top level is the SaaS provider, which can be seen as the root node of the tenant tree. The different tenants using the application are located on the next level, and can be seen as child nodes of SaaS provider. Therefore, all tenants share the same parent. Some tenants can even be divided into multiple subtenants, for example in the scenario of a reseller. In this case, the subtenants are child nodes of their respective parent tenant, making the tenants inner nodes (nodes with child nodes) of the tree and the subtenants leaf nodes (nodes without children).

Figure A.3 shows an example tenant tree where a multi-tenant software application, deployed on the public cloud by the SaaS provider, is used by three different tenants. All tenants have the same parent, the SaaS provider. Tenant A, a reseller, has three child nodes, its clients, while tenant C has two child nodes. Subtenant A1 and tenant B are examples of leaf nodes (colored in grey), whereas tenant A is an example of an inner node. Inner nodes also have some tenant data.

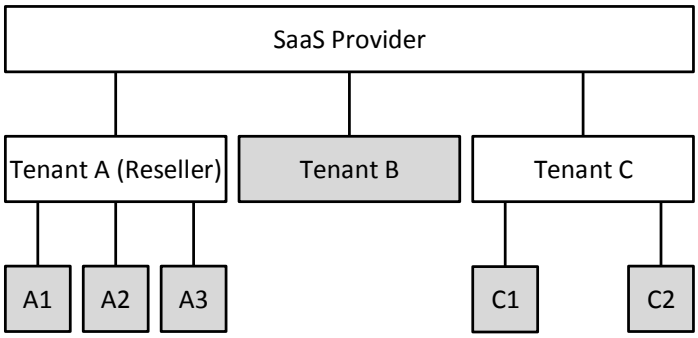


Figure A.3: The tenant tree, a logical representation of different tenants and subtenants in a multi-tenant application. Grey nodes are leaf nodes of the tree.

A.3.2 Physical Storage

By introducing the logical hierarchical representation, the question arises how and where to store the data for the different tenants. Data could be split in the same way as the logical representation, by creating a datastore for each (sub)tenant or merge multiple smaller databases. For small applications with a limited number of tenants, a single datastore can be used. We distinguish 3 different models for load balancing the data:

1. The **monolithic model**, where all data is stored in a single datastore.
2. The **fully distributed model** where every tenant and subtenant has its own datastore.
3. The **hybrid model** which is a mix of both previous models.

Table A.1 shows a comparison between the monolithic and fully distributed model. For a new SaaS application with a limited number of tenants, the provider could start using the monolithic model for storing all data, and move to the hybrid model or the fully distributed model as the amount of tenants and data grows. The monolithic model will be easier to implement with lower costs, as only a single database instance is needed. The fully distributed model on the other hand is more flexible and scalable, with guaranteed data isolation, but at a higher price. A good architecture should support both models, making it possible to select the optimal strategy for every tenant. The application provider could select the optimal strategy himself, or leave the choice to the tenant. In the latter case, the provider could offer the application in different versions with a different Service-Level Agreement (SLA), for example basic hosting using the monolithic model, silver hosting using the hybrid model with a single datastore for a tenant and its subtenants, and gold hosting using the fully distributed model.

Table A.1: Comparison between the monolithic and fully distributed model.

		Monolithic model		Fully distributed model
Cost	+	single instance, cheaper	-	multiple instances needed, higher cost
Implementation	+	easier to implement, search less complex	-	application needs to support multiple datastores, search more complex
Security	-	only on application level	+	data isolation, security both by application and database
Performance	-	shared resources, single tenant can clog-up application	+	dedicated instance for every tenant
Scalability	-	only usable for limited number of tenants	+	highly scalable

When storing data from multiple tenants into a single datastore (monolithic or hybrid model), each data row should be accompanied by a unique identifier for the tenant, the *tenantId*. In addition, add a table containing information about all different tenants and their corresponding *tenantId*. When speaking about tenant data, we can make a distinction between:

1. The tenant **users**, where all users belonging to a single tenant are stored.
2. The tenant **roles**, where the different roles for the tenant users are stored.
3. The tenant's **application data**, specific for the SaaS application.

Different combinations of the storage models for each data type are possible. Figure A.4 shows an example mapping between the tenant tree and the physical storage of the different types of data. In this example, all tenant users are stored using the monolithic model, the application data of tenant C using a hybrid model, and the roles using the fully distributed model. The application data of tenant C and subtenants C1 and C2 is stored in a single colocated datastore.

Apart from selecting the strategy for splitting the data, which data do we store in which datastore? The main goal is to store the data at the lowest possible node of the tree, starting at the root node. For example, when the roles are stored using the fully distributed model, as in Figure A.4, the subtenant-specific roles are stored at the subtenant datastores. A user who only belongs to subtenant B1 will have a corresponding role in the roles datastore of subtenant B1. A tenant administrator, who manages all subtenants of tenant B, will typically have a tenant administrator

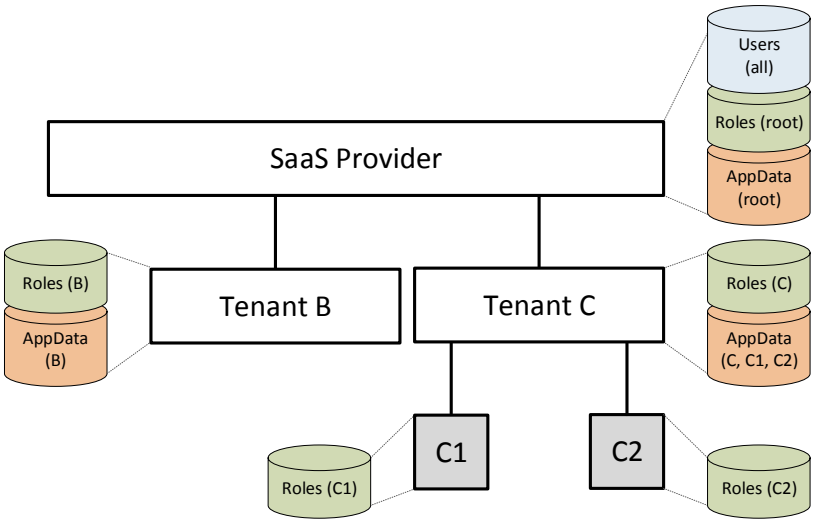


Figure A.4: Mapping between a part of the tenant tree and the physical storage of the data for the different tenants and subtenants.

role stored in the roles datastore of tenant B. A tenant administrator, who manages some (but not all) subtenants of tenant B will have a custom administrator role, also stored in the roles datastore of tenant B. Alternatively, we could give the last user separate roles in the datastores of the different subtenants, but this introduces a small overhead as multiple roles are needed for a single user.

A.4 Distributed Search

By introducing a hierarchical model for users, roles and tenant data, some data may be distributed over multiple database instances. This will have an impact on the performance and scalability of the application, as the number of users and amount of tenant data can now be much higher, but the system might have to search in multiple databases.

The question arises how to efficiently retrieve the needed data. In this section, we will propose multiple search methods, followed by a theoretical analysis of the impact on the performance.

Figure A.5 shows an illustrative scenario of a fully distributed model, with a tenant C and subtenant C2. An authenticated user “Bob” wants to access the data of subtenant C2. The authorization system needs to know whether “Bob” has

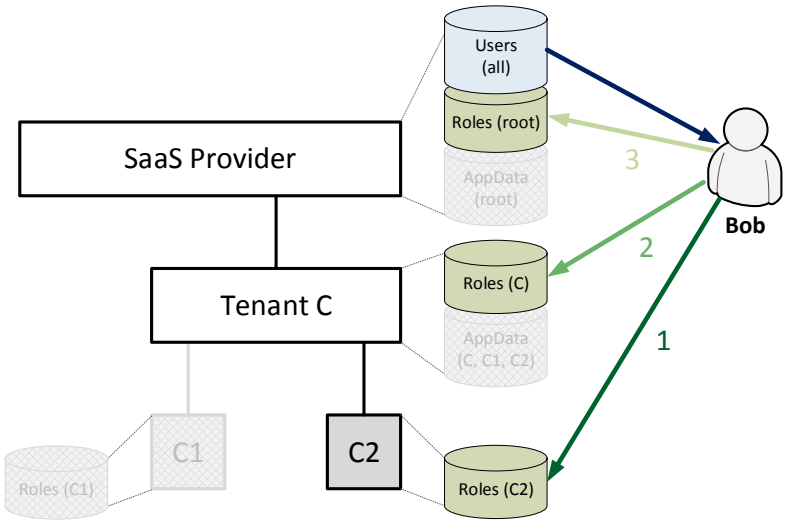


Figure A.5: Example scenario where user "Bob" wants to access the application data of subtenant C2. The required role can be stored at different locations in the tenant tree.

access to the data of subtenant C2, or more concrete, if this user has an applicable role for this subtenant. Because the model is fully distributed, meaning that every tenant and subtenant has its own roles datastore, the corresponding role for user "Bob" can be stored in three locations, as illustrated in Figure A.5:

1. The roles datastore of subtenant C2.
2. The roles datastore of tenant C.
3. The roles datastore of the SaaS Provider (the root).

Although this example scenario only handles the search for a role in the roles datastore, a similar scenario can be described for searching a user or finding some tenant data.

A.4.1 Search Methods

When searching for the data, in our example scenario a corresponding role for user "Bob", we need to search at different locations. In general, the data can be stored at any location along the path from the (leaf) node to the root. We can do a *serial search*, starting at a single datastore, e.g. the leaf datastore, and moving to the next datastore along the path in case the data is not yet found, or in case

Table A.2: Overview of used symbols.

Symbol	Description
t_{ds}	time to find a record in a single datastore, based on an indexed parameter
t^*_{ds}	time to find a record in a single datastore,based on a non-indexed parameter
n_{ds}	number of records in the datastore
n_{tot}	total number of records
p_{ds}	probability that user record is stored in the specific datastore

the datastores are stored on different (virtual) machines, we could also perform a *parallel search*), searching in all datastores at the same time and merging the results.

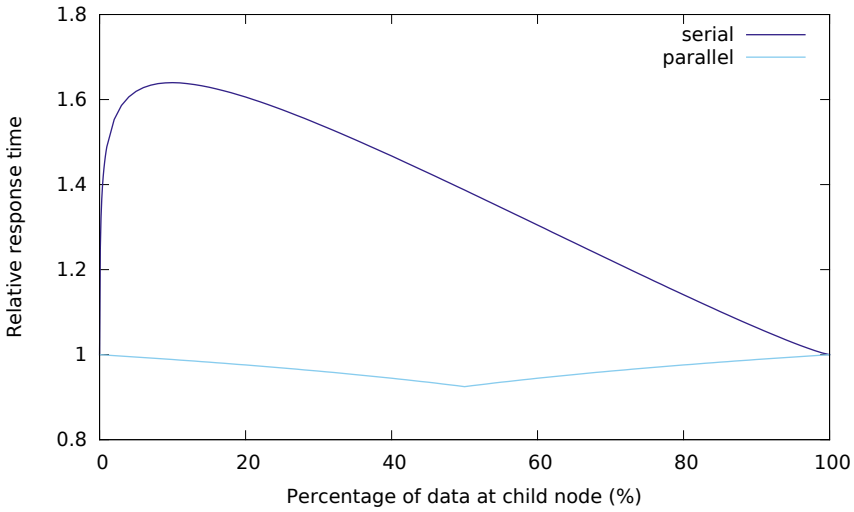
In case of the *serial search*, we can start at the leaf node, and move up the tree towards the root node, this is the *bottom-up* approach, or we could start at the root and continue down the tree towards the leaf node (*top-down* approach). However, we strongly prefer the bottom-up approach. The bottom-up approach is easier to implement, as in the top-down approach, the path from the root to the leaf node needs to be calculated in advance. Also, when using the top-down approach, all traffic needs to pass the root node, turning this datastore into a possible bottleneck.

In most cases, the search can stop when an applicable role is found. In some scenarios however, roles higher in the hierarchy could overrule lower roles. This will have a bad influence on the performance as the authorization system needs to search all datastores to get all roles. We strongly recommend to avoid such scenarios, as it will not only have a bad influence on the performance, but the authorization system will also become far more complex.

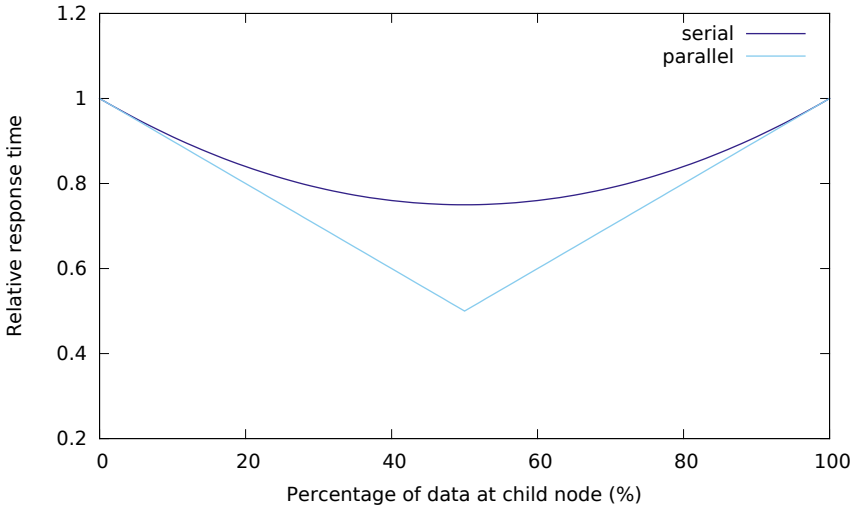
A.4.2 Theoretical Analysis

As the number of possibilities for dividing the data over the datastores is endless, we will breakdown the problem into two cases, which can be combined for the implemented model. The symbols used in this section are summarized in Table A.2. We will only focus on the distribution of users among different datastores, but the same approach can be followed for distribution of roles and/or tenant data (if applicable). When using the serial search, we will use the bottom-up search, for the above-mentioned reasons.

As described in the previous section, tenants can be logically organised in a hierarchical way, and a mapping can be made to the physical storage locations.



(a) Distributed Indexed Search



(b) Distributed Non-Indexed Search

Figure A.6: Theoretical analysis of the time needed for finding the tenant data in a 2-level hierarchical structure tenant-subtenant. The vertical axis denotes the relative response time, where 1 equals the time needed to find the data in a single datastore.

A.4.2.1 Time required to find a user in a datastore

We start our analysis with the time needed to find a single user in a large dataset. The value is dependent on both the number of users in the dataset, and the indexing method used.

When searching for an entry in a single table, based on an indexed parameter, the average time needed has a factor $O(\log(n))$, when searching on a non-indexed parameter, this factor is $O(n)$, where n equals the number of rows in the table.

Therefore, in theory, the time needed to find a user in a single datastore will be equal to

$$t_{ds} = C \times \log(n_{ds})$$

$$t^*_{ds} = C \times n_{ds}$$

where t_{ds} and t^*_{ds} denote the time needed to find a user (record) in a dataset, based on an indexed and non-indexed parameter, n_{ds} denotes the number of users in the datastore, and C is an unknown constant factor.

A.4.2.2 Probability

When we have multiple datastores, the probability that a random user u is stored in a specific datastore is equal to

$$p_{ds} = n_{ds}/n_{tot}$$

where n_{tot} equals the number of users in the datastore.

A.4.2.3 Vertical Search

When users are divided over two datastores, where one datastore ($parent$) is the parent of the other ($child$), the average time needed to find a random user, using a bottom-up serial search, can be calculated as

$$t_{avg} = p_{child} \times t^{(*)}_{child} + p_{parent} \times (t^{(*)}_{child} + t^{(*)}_{parent})$$

The average time to find a user, using the bottom up serial search, can be divided into 2 subcases: the case where the user is stored at the $child$ datastore (probability p_{child}), and the case where the user is stored at the $parent$ datastore (probability p_{parent}). In the first case, the bottom-up algorithm will only have to search the $child$ datastore. However, in the second case, the algorithm will start searching in the $child$ datastore, and continue the search in the $parent$ datastore.

In case of a parallel search, the average time corresponds to

$$t_{avg} = \max(t^{(*)}_{child}, t^{(*)}_{parent})$$

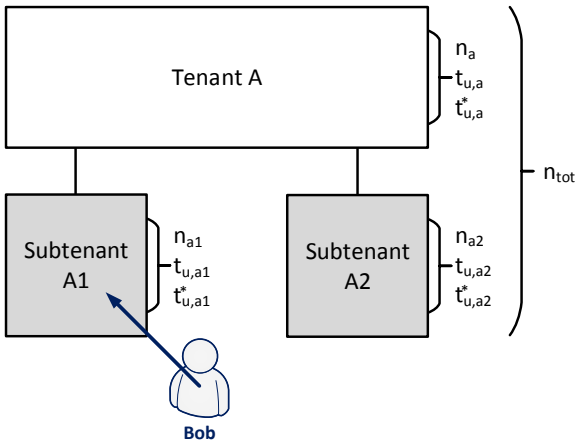


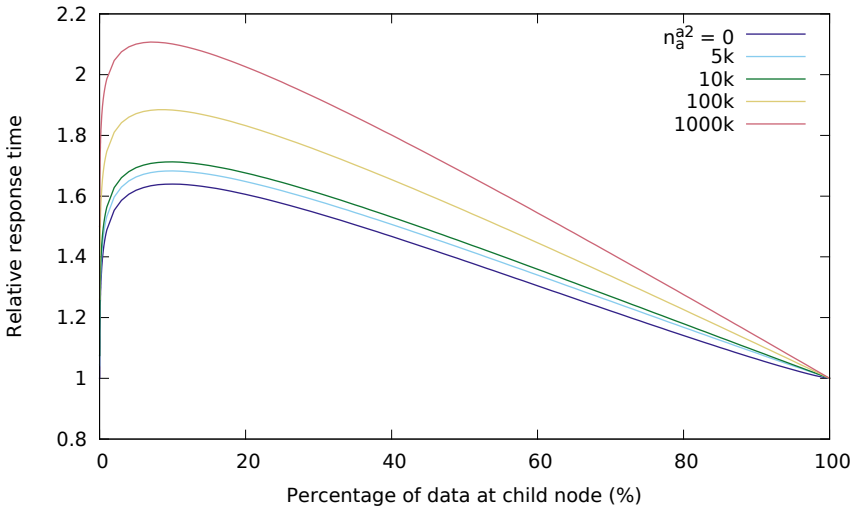
Figure A.7: Extension to the full model. User "Bob" wants to access the data of subtenant A1, and needs a role which can be stored at the subtenant or tenant node.

Figure A.6 illustrates the time needed for finding a corresponding data row in a 2-level hierarchical structure, existing of a tenant (parent) and a subtenant(child), using the *bottom-up* approach. The horizontal axis denotes the amount of data stored at the child node, while the vertical axis denotes the normalised response time, where 1 equals the time needed to find the data in a single datastore. As can be seen from this figure, the distributed serial search on an indexed parameter will have a bad influence on the performance when data is split over different datastores, especially if most of the data is located at the parent node. In all other models, splitting the data over multiple stores results in better performance, with an optimum if the amount of data is equally divided over both stores.

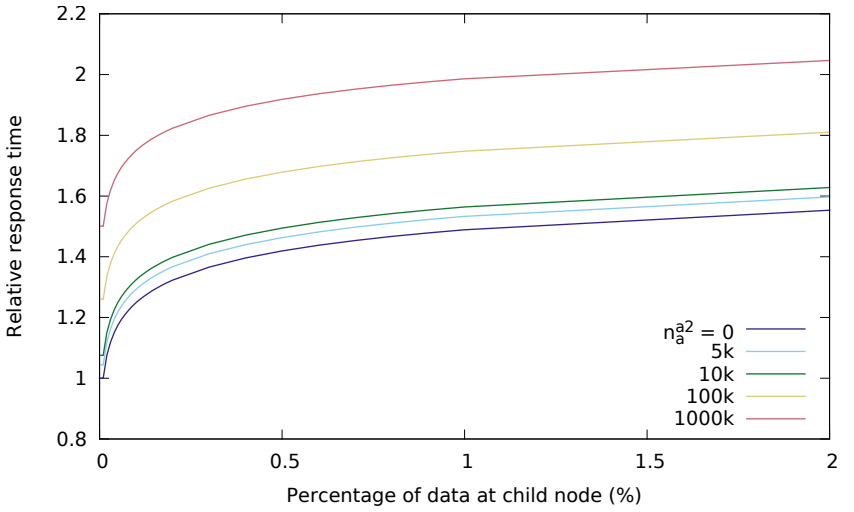
A.4.2.4 Horizontal Search

In case the data is divided over 2 datastores on the same level ($ds1$ and $ds2$), for example in the case of 2 subtenants, we won't have to search both datastores. For example, if we want to find out if user "Bob" has access to subtenant B1, we will only search the roles datastore of subtenant B1, and not the roles datastore of subtenant B2. So, in case there are no parent nodes, and data is divided on the same level, the average time to find a role in datastore $ds1$ can be given as

$$t_{avg} = t^{(*)}_{ds1}$$

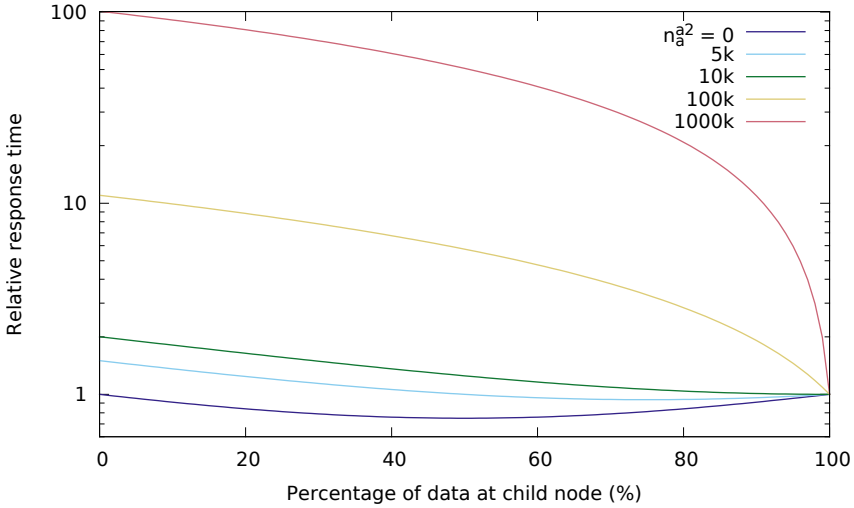


(a) Distributed Serial Indexed Search



(b) Distributed Serial Indexed Search - Detail

Figure A.8: Theoretical analysis of the impact of other subtenants on the time needed for finding tenant A1's data in a full model. Tenant A1 has a total of 10k records divided over the 2 datastores. The variable n_a^{a2} denotes the extra rows added to the shared (parent) datastore by the other subtenants.



(c) Distributed Serial Non-Indexed Search

Figure A.8: Theoretical analysis of the impact of other subtenants on the time needed for finding tenant A1's data in a full model. Tenant A1 has a total of 10k records divided over the 2 datastores. The variable n_a^{a2} denotes the extra rows added to the shared (parent) datastore by the other subtenants (continued).

A.4.2.5 Impact of other subtenants on the performance

We can easily extend the horizontal and vertical search to a full hierarchical model. Figure A.7 shows a simple tree with a single tenant and 2 subtenants. In our example scenario from Section A.4, user "Bob" want to access the data of subtenant A1. While the datastore of subtenant A1 is dedicated, the datastore of tenant A is shared between all subtenants. When a subtenant adds extra data to the shared datastore, this will have an impact on the performance. The total number of records in the shared datastore is given as

$$n_a = n_a^a + n_a^{a1} + n_a^{a2}$$

where n_a^a denotes the number of records added by tenant A and n_a^{a1} and n_a^{a2} the number of records added by subtenants A1 and A2. The time needed to find Bob's role equals

$$t_{avg} = p_{a1} \times t_{a1}^{(*)} + p_a \times (t_{a1}^{(*)} + t_a^{(*)})$$

with probabilities

$$p_{a1} = n_{a1}/(n_{a1} + n_a^{a1})$$

$$p_a = n_a^{a1}/(n_{a1} + n_a^{a1})$$

Figure A.8 illustrates what happens when the number of records added by the other subtenants (subtenant A2 represents all other subtenants) increases for both the distributed serial indexed and non-indexed search. Note that the different curves in Figure A.8a don't start at the same point, as can be seen in Figure A.8b. As can be seen from this figure, the influence of the other subtenants on the overall performance of subtenant A1 will decrease as more of the data of subtenant A1 is stored at the leaf node. Hence, when splitting a datastore vertical, it is never a good idea to equally divide the data over the tenant and subtenant.

A.4.3 Conclusions

The theoretical analysis shows that in most of the cases, splitting the data over multiple stores won't have a bad influence on the performance of the application. Only when using a distributed serial search in a 2-level hierarchical (vertical) structure, and when searching on an indexed value, the performance will be bad when most of the data is stored in the parent datastore. However, in most structures, the datastores will not only be split vertical, but also horizontal, and most of the data will be located at the leaf nodes, yielding much better performance.

A.5 Evaluation Results

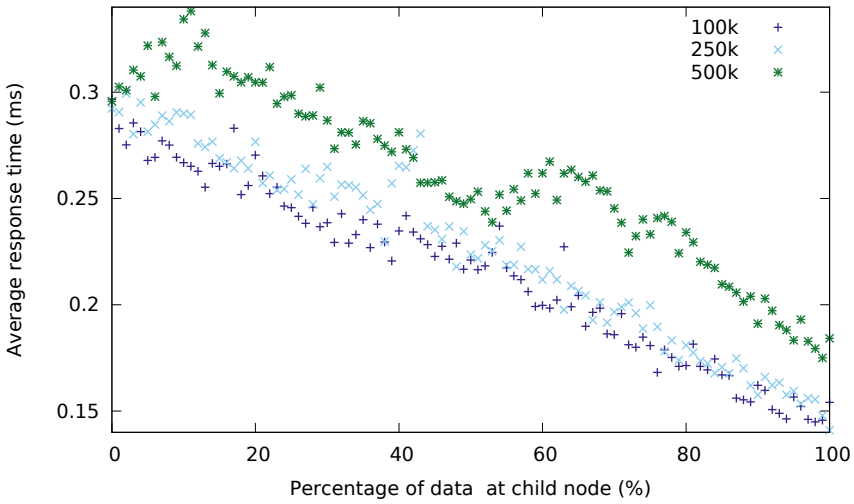
In this section, we will verify our theoretical analysis of the performance by different experiments.

A.5.1 Experimental Analysis

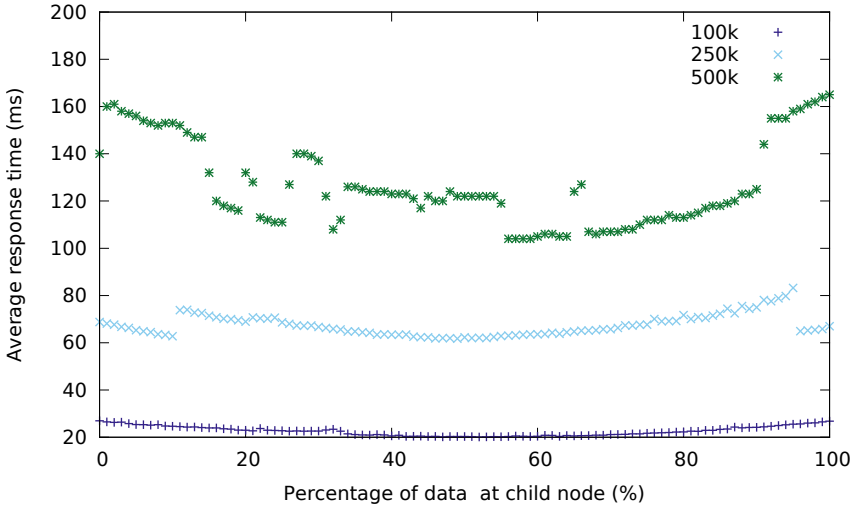
To verify our theoretical analysis, we ran some experiments on 2 different environments. For the first environment, we used a MySQL dbms on Ubuntu 13.04, running on a virtual machine with a single core vCPU and 2GB of memory installed. As a second environment, we configured Microsoft SQL Server on Windows 2008R2, running on a physical machine with a 2.8GHz Intel Core i5 (quad core) and 4GB of memory installed.

During the experiment, we calculated the average time to find a random user in a 2-level hierarchical structure (tenant-subtenant), using the distributed serial bottom-up search. Figure A.9 shows the results for both environments, for a total of 100k, 250k and 500k users. During all experiments, the average time was calculated by authenticating 10 percent of the total amount of users. The horizontal axis denotes the percentage of data stored at the child (subtenant) node, while the vertical axis denotes the average response time, expressed in milliseconds.

As can be seen from the figure, the experimental results resemble the calculated theoretical times, except for the indexed serial search on the SQL Server environment. This exception is due to the fact that response times are very low when

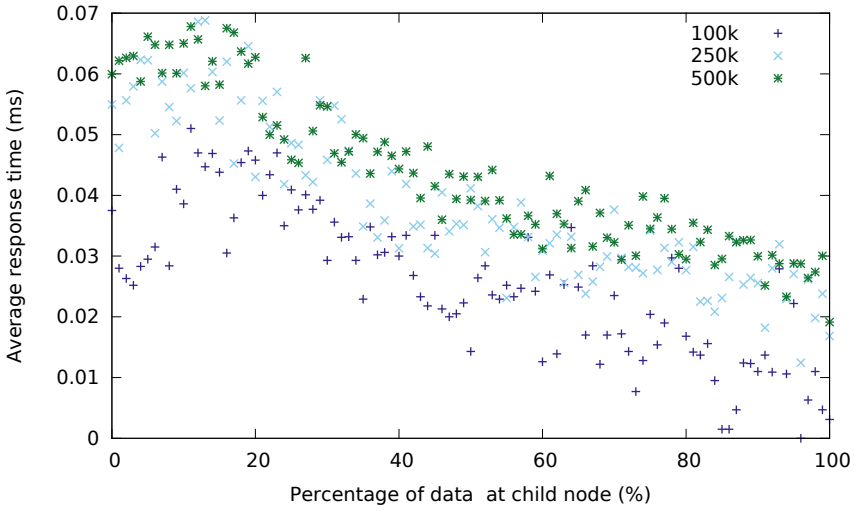


(a) MySQL - Distributed Serial Indexed Search

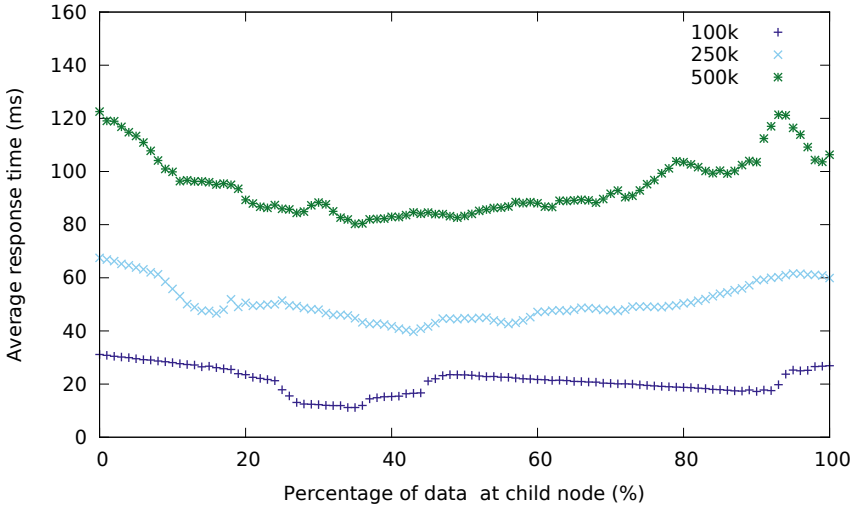


(b) MySQL - Distributed Serial Non-Indexed Search

Figure A.9: Experimental analysis of the time needed for finding the tenant data in a 2-level hierarchical structure tenant-subtenant.



(c) SQL Server- Distributed Serial Indexed Search



(d) SQL Server- Distributed Serial Non-Indexed Search

Figure A.9: Experimental analysis of the time needed for finding the tenant data in a 2-level hierarchical structure tenant-subtenant (continued).

using indexing, making the overhead of searching two databases bigger than the average time. As in the theoretical analysis, the distributed indexed search yields better results as more users are located at the child node. For the distributed non-indexed search, best results are achieved when data is equally divided over both datastores. As the experimental results are in line with the calculated theoretical results, the theoretical analysis can be used to optimize the hierarchical structure for big amounts of data.

A.5.2 Conclusions

Experiments confirmed that the results are in line with the theoretical analysis. Therefore, the theoretical analysis can be used to build the decision support part of the the data access layer, for autonomous splitting and merging tenant datastores. In case of a serial search, there is some small overhead due to the switching between datastores, but this overhead can be neglected as the size of the datastore grows. In our evaluation, we only focused on the two subcases (parent-child and 2 nodes at the same level), but this can be used for extension to a full hierarchical structure like the one proposed in Figure A.4.

A.6 Conclusions and Future Work

As the number of tenants and users in the multi-tenant application grows, users, roles and tenant data can be split over multiple datastores. In this appendix, we presented a hierarchical model for the logical representation of the tenant tree and a mapping to the physical storage. Users, roles and tenant data can be divided using the monolithic, the fully distributed or the hybrid model. When data is divided over multiple datastores, we can make use of a serial (bottom-up or top-down) search, or a parallel search when the datastores are located on different machines.

The theoretical and experimental analysis confirmed that the hierarchical model presented in this appendix can be used to build autonomous high scalable multi-tenant applications in the cloud. By using the hierarchical model for both the physical representation of tenants and subtenants, and choosing a strategy for the physical storage of the different datastores, it is straightforward to create a mapping between the two models, making the management of the application less complex.

In future work, we will focus on the elasticity of the SaaS application and decision support part of the data access layer, to build an autonomous system for automatic scaling of the application and data in the public cloud.

Acknowledgment

The research presented in this chapter is partly funded by the iMinds PUMA [2] project.

References

[1] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao. *A Framework for Native Multi-Tenancy Application Development and Management*. In 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., pages 551 – 558, 2007.

[2] *PUMA: Permissions, User Management and Availability for Multi-tenant SaaS Applications*, 2013. Available from: http://www.iminds.be/en/research/overview-projects/p/detail/puma-2.

[3] M. Armbrust, R. Fox, Armandoand Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. *Above the Clouds : A Berkeley View of Cloud Computing*. Technical report, University of California at Berkley, 2009.

[4] C.-P. Bezemer and A. Zaidman. *Multi-tenant SaaS applications: maintenance dream or nightmare?* In Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), IWPSE-EVOL '10, pages 88–92, New York, NY, USA, 2010. ACM. Available from: http://doi.acm.org/10.1145/1862372.1862393, doi:10.1145/1862372.1862393.

[5] P.-J. Maenhaut, H. Moens, M. Verheye, P. Verhoeve, S. Walraven, W. Joosen, and V. Ongenae. *Migrating Medical Communications Software to a Multi-Tenant Cloud Environment*. In IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), pages 900–903, May 2013. Available from: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6573107.

[6] M. Decat, B. Lagaisse, D. Van Landuyt, B. Crispo, and W. Joosen. *Federated Authorization for Software-as-a-Service Applications*. In To be published in the Proceedings of DOA-Trusted Cloud'13, 2013.

[7] M. Decat, B. Lagaisse, and W. Joosen. *Toward efficient and confidentiality-aware federation of access control policies*. In Workshop on Middleware for Next Generation Internet Computing. ACM, 2012. Available from: http://doi.acm.org/10.1145/2405178.2405182, doi:10.1145/2405178.2405182.

[8] J. M. A. Calero, N. Edwards, J. Kirschnick, L. Wilcock, and M. Wray. *Toward a Multi-Tenancy Authorization System for Cloud Services*. IEEE Security and Privacy, 8(6):48–55, November 2010. Available from: http://dx.doi.org/10.1109/MSP.2010.194, doi:10.1109/MSP.2010.194.

[9] C. D. Weissman and S. Bobrowski. *The design of the force.com multitenant internet application development platform*. In Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, SIGMOD '09, pages 889–896, New York, NY, USA, 2009. ACM. Available from: http://doi.acm.org/10.1145/1559845.1559942, doi:10.1145/1559845.1559942.

[10] S. Yu, C. Wang, K. Ren, and W. Lou. *Achieving secure, scalable, and fine-grained data access control in cloud computing*. In Proceedings of the 29th conference on Information communications, INFOCOM'10, pages 534–542, Piscataway, NJ, USA, 2010. IEEE Press. Available from: http://dl.acm.org/citation.cfm?id=1833515.1833621.

[11] S. Walraven, E. Truyen, and W. Joosen. *A middleware layer for flexible and cost-efficient multi-tenant applications*. In ACM/IFIP/USENIX 12th International Middleware Conference, volume 7049, pages 370–389. Springer Berlin / Heidelberg, December 2011. Available from: https://lirias.kuleuven.be/handle/123456789/313768, doi:10.1007/978-3-642-25821-3_19.

[12] E. Yuan and J. Tong. *Attributed based access control (ABAC) for Web services*. In Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on, pages –569, 2005. doi:10.1109/ICWS.2005.25.