PLACEMENT AND MONITORING OF ORCHESTRATED CLOUD SERVICES

Lars Larsson

Licentiate Thesis, January 2015
Department of Computing Science
Umeå University
Sweden
Abstract

Cloud computing offers pay-per-use on-demand access to computer resources for hosting program execution environments for software service deployment. Management of cloud resources includes determining, based on current monitored resource availability, which part(s) of a computational infrastructure should host such program execution environments in a process called placement. Our work defines directives that lets consumers of cloud resources influence placement to express relationships between cloud services (orchestration) and deployment constraints to uphold for related service components, without surrendering the ultimate control over placement from the infrastructure owner. The infrastructure owner remains free to define their policies and placement optimization criteria, e.g., to consolidate work that needs to be done to as few physical host machines as possible for power savings reasons. We show how the placement process can be adjusted to take such influence into account and validate through simulations that the adjustments produce the correct result without too large computational impact on the placement process itself. Further, we present a technique for transferring large data files between cloud data centers that operate in (separate) cloud federations that avoids repeated transfers in a delegation chain between members of (different) cloud federations. Finally, we present a non-invasive method of extracting monitoring data from a service deployed in a cloud federation, and a framework for making monitoring information available and understandable in spite of technical differences between monitoring systems used in cloud federations.
Preface

This thesis contains an introduction to the field and the papers listed below (reprinted with permission from the individual publishers).

Paper I  Erik Elmroth, Lars Larsson  

Paper II  Lars Larsson, Daniel Henriksson, Erik Elmroth  

Paper III  Daniel Espling, Lars Larsson, Wubin Li, Johan Tordsson, Erik Elmroth  

This work has been supported financially by the Swedish Research Council (VR) under contract 621-2005-3667 and for the Cloud Control project, the Swedish Government’s strategic research project eSENCE, and by the European Community’s Seventh Framework Programme ([FP7/2001-2013]) under grant agreements no. 215605 (RESERVOIR) and no. 257115 (OPTIMIS).

Publications by the author not included in this thesis:

Acknowledgements

This thesis has been a long time coming. When I joined the research group in 2008 as a systems developer, cloud computing was still in its relative infancy. Since starting as a PhD student in 2009, I have both witnessed and been part of a lot of important development that has taken place in the cloud computing field, some of which has made it into papers and ultimately, this thesis. I am grateful to my supervisor Erik Elmroth who took me on and has let me walk this path, and to him and Johan Tordsson for the side path that we took in 2011 when they started Elastisys and I got to dive head-first into pure(-ish!) software development again in that context. I am in constant awe of how much you two manage to get done and still bring your full focus and attention to each individual meeting. Resource contention issues solved at its finest!

At Elastisys, Peter Gardfjäll always teaches me something new. You are by far the best software developer I know, and I feel like I grow every day I work with you. I am incredibly happy that I get to share an office with you, and that you are also laying down some serious roots in Umeå with your family. I hope that I get to be around you as a friend and colleague for a long time to come!

My dear former office-mate, colleague, friend, and fellow globetrotter Daniel Espling: you are amazing, and my life is richer for having you in it. Thank you for all the good times we have had, in- and outside of the office. I feel like we went through all of this together, and I thank you for being there with me for every step of the way, even when it got rough.

If it wasn’t for P-O Östberg, I probably wouldn’t even be a PhD student to begin with. I would likely have taken a job at Ericsson in southern Sweden in 2008 and only looked back when that whole division was laid off only two years later. Thank you for believing in me, encouraging me, and for your friendship: your love for research and enthusiasm is infectious (and also saved me from making a huge mistake)!

It is a pleasure to be part of our ever-expanding and always amazing group, that is full of so many wonderful people. Both the present members (Ahmed, Amardeep, Cristian, Ewnetu, Francisco, Gonzalo, Jakub, Lennart, Luis, Mina, Olumuyiwa (Muyi), Selome, and Tomas apart from those already mentioned) and the ones that are gone but not forgotten (Lei, Petter, and Wubin Li (Viali)) fill my days with interesting discussions literally about anything and everything between heaven and earth. Some of it even work-related! Thank you for enriching my life with new perspectives, insights, and philosophies. You all rock!

On a personal note, I wish to thank my friends and family. You keep me grounded, sane, and happy. To my mother Ursula and sister Lisa, there are not words that can
sufficiently thank you for all the countless ways you have enriched and helped shape my life. Johannes and Henrik, my brothers from other mothers, you have been with me for most of my life, offering your support and love no matter the circumstances. And my more recent dear friends I have got to know since coming to Umeå: Helena, Sofia, Peter, Mahsa, and Robin — you fill this cold high north with warmth and love.

And finally, my most immediate family, where even more important developments have taken place during these years with the birth of my three children (Tina in 2009, and the twins David and Samuel in 2013). I love you, and I am forever grateful for being able to spend time with you. Last, but in no way least, thank you to my beloved wife Anna, for being my love, support, my everything. I think I dreamed you into life.

Lars Larsson, Umeå, January 2015
Contents

1 Introduction 1

2 Cloud Computing 3
   2.1 Deployment Models 5
   2.2 Cloud Service Deployment 6
   2.3 A Look Ahead: Container Clouds 8

3 Cloud Resource Management 11
   3.1 Placement Optimization 13
   3.2 Monitoring 14
   3.3 Orchestration 15
   3.4 Storage 15

4 Service-oriented Architectures in the Cloud 17
   4.1 Services Developed for Cloud Environments 18

5 Thesis Contributions 21
   5.1 Paper I 22
   5.2 Paper II 22
   5.3 Paper III 23

6 Future Work 25

Paper I 39

Paper II 51

Paper III 61
Computers and their associated IT equipment such as networks and storage units are not only expensive to manufacture, but operating them requires expensive power and cooling, and when they break, they require maintenance by experts. While consumers are more than happy to periodically buy and briefly own computers used sparingly and mostly for leisure, businesses and researchers need to balance an ever-growing dependence on computational capabilities with keeping within a constrained budget. Hence, much effort has been devoted to letting some (third) party take over the costs of owning and operating large (distributed) computer infrastructures, while allowing the users of the infrastructures to pay in accordance with their usage. Seeing this need early on, John McCarthy in the 1960’s envisioned delivery of computational power as a public utility [1, 2].

Several implementations of the idea of providing (remote) access to (distributed) computational infrastructures for payment based on usage of said infrastructure have been created over the years, each of which designed for a particular niche task and with the limitations of that time-period in mind. The first implementation is that of mainframe computing (and clusters of mainframes followed as soon as networks had increased sufficiently in performance), where users run processes on a particular machine, running a particular operating system. The first commercially successful such system was the IBM-360 in 1964 [3]. Each program typically had to be tailored for the particular execution environment, thus making mainframe systems rather difficult to work with. Time-sharing systems such as UNIX came from the 1960–1970’s [4], and each user could be billed according to the amount of computer time they had consumed.

Fast network connectivity allowing transfers of large amounts of data made the grid emerge in the 1990’s and early 2000’s [5, 6, 7]. The main focus of grids is to allow for batch processing of typically large data sets, where access to results is not immediate, since submission of a time-limited grid job only specifies what work is to be done (possibly, however, with a deadline), and it is up to the grid system to find the best possible time slot to actually execute the job. The grid software (middleware) provides access to the computer, offering an execution environment for the grid job to be executed in. What libraries are available in this environment is dictated by the grid site’s operators, which implies that the job process must be tailored to the particular environment in which it will run. In contrast, cloud computing offers virtual machines as the runtime environment, and the cloud services deployed are not
time-limited processes. Cloud services are expected to start within minutes of the user having requested access to the computational infrastructure, and run consistently and constantly until the service owner at their own discretion terminates the process and its execution environment. Chapter 4 takes a deeper look at services and how they are deployed in cloud computing, but for now, the simple definition of a service as a piece of software that offers its functionality over a network, e.g. streaming music files from a library of files, will suffice. Clouds and virtual machines are presented in more depth in Chapter 2, and management of cloud resources in Chapter 3.

Each utility computing implementation is niched toward a particular task or set of tasks, and designed accordingly. They are all still relevant today as the niches still remain, but with regard to current research effort, cloud computing is by far the largest today. Its niche, long-running general services, is also considerably broader than e.g. pure batch processing, which attracts researchers from many fields.

Autonomous computational infrastructures, whether they are grids or clouds, are often joined together in an attempt to pool resources and provide even more utility to their users. Such collaborations are known as federations, and their formation typically requires significant engineering effort and service contract formulation, as the infrastructures may not have been designed with collaboration as a goal in addition to being fully autonomous. There is therefore typically no central management functionality for the federation as a whole. The Atlas project at CERN [8], for instance, consumes what at the time of writing is vast amounts of computational resources, including processing power, network bandwidth, and storage, pooled from several universities. It produces about 1 Petabyte of data per second, and requires processing equivalent to about 50,000 modern PCs\(^1\). Since this amount of data processing greatly exceeds what one data center can handle, federations of computational infrastructures are required to deal with this type of Big Data [8]. Federated computational infrastructures offer great processing power, but also come with a set of unique challenges and trade-offs, including ones related to security, performance, trust, accountability, and a lack of control over remote sites, as each site is autonomic.

The work in this thesis addresses some of the problems faced in cloud federations. In particular, we: propose to deal with the lack of control by mechanisms for users to influence decisions that ultimately affect how their computational needs are handled, ensure accountability toward resource consumers that they get what they pay for through monitoring systems, and enable infrastructure owners to optimize their own internal processes according to their own criteria. The work is motivated by the dissatisfaction felt as a long-time cloud user/customer with how services are deployed in current clouds, and how little influence one is given in that situation today.

\(^1\)According to: http://atlas.ch/computing.html
Following its inception in the early 2000’s, cloud computing has become an umbrella term coming to mean many different things that are not always even related to each other. In the broadest sense, cloud computing can mean anything where resources or software services are running at a remote site. This overly broad definition aids commercial marketing efforts, but hinders meaningful scientific discourse. We will only consider definitions that are in line with the utility computing vision, which is to allow for users to make use of computational resources (possibly owned by a third party), meter their usage, and (typically) then have them pay in accordance with their usage in some way that makes sense in the service context (e.g. either abstract credits from an allotment or actual currency). This is realized by virtual computing infrastructures, such as grids and clouds, that exist to provide an execution environment for computer programs. Execution environments are isolated from each other so that the resource demands and usage of each does not negatively affect the others [4, 9]. The type of program deployed dictates how the execution environments are designed and implemented, and how much the capacity consumer needs to worry about this detail depends on the service model, i.e. how the computational resources are made available to its consumers. For cloud computing, these service models are [10, 11]:

- **Software-as-a-Service.** Scalable installations of a given software are offered to the consumer’s users, e.g. Microsoft Outlook or SharePoint, and the consumer’s users pay per use in some fashion that is natural for the software, typically per-licence.

- **Platform-as-a-Service.** A platform of useful surrounding services, e.g. application servers, preconfigured databases, message queues, mailing systems, etc., is offered to software developers, making it easy to focus only on developing the business logic of scalable applications. Use of surrounding services and of the business logic itself is metered and billed, according to some plan.

- **Infrastructure-as-a-Service.** Consumers provision custom execution environments known as virtual machines (VMs) directly from the cloud resource owner, and pay per use for the resources they consume. The VMs can be configured with any software of the consumer’s choosing, and the consumer is responsible for setting up any required services and software stacks themselves.
We will henceforth use the terminology Infrastructure Provider (IP) for cloud capacity producers, Service Providers (SP) for cloud capacity consumers, as this is both in line with the papers contained in this thesis, and with the well-cited definition by Vaquero et al. [10].

In this thesis, we focus exclusively on the Infrastructure-as-a-Service (IaaS) service model. The central unit in contemporary IaaS is the highly customizable execution environment implemented as VMs (but this is about to change, see Section 2.3 for a look toward the future with containers). A VM is an abstraction layer that acts as a partition of a large physical *host* machine, where each virtual machine can run its custom operating systems independently and in isolation from each other [12, 13, 14]. Access to hardware, such as network interface cards, is emulated or provided shared access to in a bridged fashion within the host operating system [14]. The software that coordinates virtual machines on the physical actual machine is called a hypervisor. Emulating an entire computer in this fashion in software alone is possible, but prohibitively slow. It was not until hardware-assisted virtualization became available and efficient [15, 16, 17, 18, 19, 20, 21] that virtualization and cloud computing was reasonably possible and desirable for paying customers. Figure 1 shows an overview of the concept of partitioning physical machines in differently sized VMs.

VMs are assumed to start provisioning the service within minutes after having been requested, and do not (typically) terminate until the paying user dictates it. A single physical host in a cloud site typically runs a (large) number of VMs, meaning that they are not isolated from each other temporally. Instead, *service-level agreements* (SLAs) [22, 23] govern the quality of service experienced by the VMs, by defining a set of measurable *service-level objectives* (SLOs). Violations against SLOs are typically cause for financial compensation to the user, as outlined in the SLA.

Because VMs can be fully customized, they can support a large variety of use cases. One can deploy software in them to support long-running services (such as web services) or short-running tasks (as in the grid case). We focus solely on the long-running service aspect, as it opens up avenues for research and introduces addi-
tional complexity in e.g. the process of choosing which host should be used for VM provisioning. For short-running tasks, data locality is likely the primary driver in such processes, whereas longer-running services experience usage variations that can have significant implications on management processes (more about these in Chapter 3).

In closing of this brief introduction to cloud computing, we consider the definition of the U.S. National Institute of Standards and Technology (NIST) that lists the essential characteristics of cloud computing as follows [11] (with reworded explanations):

- **On-demand self-service.** As consumers of cloud resources (including virtual machines, networks, and storage), SPs should be able to provision resources with minimal IP effort, as demand dictates. Self-service implies that human interaction requirements should be minimized.

- **Broad network access.** Ubiquitous and standard network and transport protocols should be used to allow cloud resource consumers access from a (large) number of network locations.

- **Resource pooling.** SPs are presented with dynamically allocated resources from large pools of (heterogeneous) resources, but are typically kept unaware of precisely where these resources come from and with whom they are sharing these resources. This unawareness is of great importance for the contributions of this thesis, as we offer optimization possibilities from both the IP and SP aspects, hinged on the idea of allowing the IP to offer resources without surrendering neither information on nor control over management of these resources to SPs.

- **Rapid elasticity.** SPs should be able to rapidly provision more resources, or shed resources that are no longer needed, from a seemingly unlimited pool of resources. This is the topic of future work, see Chapter 6.

- **Measured service.** Use of various resources should be metered and billed according to usage. SPs typically pay per computational capacity and sizes of internal memory for virtual machines, network transfers, and storage size and use in terms of input/output operations separately, but on a single bill from the cloud provider.

### 2.1 Deployment Models

Cloud computing can be deployed according to a variety of models. The fundamental three models are, from most restrictive prospective users to least: **private clouds**, where a single organization owns and operates its own physical hardware and makes it available to their own users as cloud resources; **community clouds**, where collaborating organizations pool their resources together in a cloud spanning over the entire pool, but the resources are only offered to members of these partnering organizations; and **public clouds**, where a single or partnering organizations offer access to cloud resources to the public. Hybrid versions of these deployment models exist as well,
where the goal is to allow for cloud bursting, i.e. seamlessly making use of cloud resources from another cloud while still enforcing barriers between cloud entities, as appropriate. This typically requires compatible software to work [11].

The papers in this thesis all focus on federated clouds [24, 25], i.e. ones where some kind of collaboration takes place between cloud sites. We define a cloud site as an organizational entity within which the infrastructure is exposed to SPs as a unit, typically a data center. A single cloud infrastructure provider can own and maintain a number of such sites, in effect offering a federation of cloud sites. By this definition, Amazon Elastic Compute Cloud is a federation, as the various regions are completely separate, while availability zones within a single region are not to be regarded as units (hence, a region is not itself a federation). For our research, whether some of these are private, community, or public clouds does not matter, as long as there are no technical incompatibilities make a collaboration impossible.

For a toolkit aimed at making various cloud constellations easy to work with, see the OPTIMIS toolkit [26].

2.2 Cloud Service Deployment

Figure 2 shows the phases of service deployment onto a cloud and the tasks included in each phase, carried out by the SP and IP, respectively. We divide the provisioning process in three phases: staging, deployment, and operation. In the staging phase, the SP prepares a VM image and uploads it (Task 1) and any data required for processing to the IP for storage (Task 2). Note that clouds typically offer a catalog of pre-defined VM images. If this is the case, the SP may simply choose one from this catalog, and possibly pay licensing costs for doing so. A VM image is a file containing the contents of what is to become the (virtual) hard-drive contents of the VM. The response back from the IP is some kind of identifier to the stored image and data, which can later be referred to in the service manifest [25].

In the deployment phase, the SP prepares a service manifest for the service that is to be deployed (Task 3). This is a document containing orchestration information for one or more service components, and any additional information and rules concerning the placement of these service components. The cloud infrastructure performs admission control, i.e. determines if the service described in the service manifest can be accepted for deployment or not (Task 4). If the service is deemed admissible in accordance with the cloud’s rules and resource availability, a suitable placement for the VMs each service component is to be deployed in is found (Task 5).

The operation phase starts as soon as a VM is started, and it contains the concurrent tasks of controlling (Task 7a) and monitoring (Task 7b), performed by the SP and IP, respectively. The SP can modify the state of the VM by issuing commands to pause/resume, stop/start, resize/duplicate, or terminate the VM. The IP continuously optimizes VM placement and monitors the resource usage by the VM to present the data to the SP, and for billing purposes.
Figure 2: Service deployment phases and tasks for SP and IP.

Staging

1. Prepare image and data

2. Store

Deployment

3. Service manifest

4. Admission control

5. Placement

Operation

6. VM started

7a. Control VM state

7b. Monitor VM. Optimize placement

Figure 2: Service deployment phases and tasks for SP and IP.
2.3 A Look Ahead: Container Clouds

Providing access to fully customizable isolated program execution environments is one of the cornerstones of cloud computing. Currently, this isolation is provided by means of virtualization, for which a performance penalty must be paid as emulating an entire computer and its associated hardware is obviously more wasteful performance-wise than bare-metal access as offered by grids, even in spite of attempts to close the gap by e.g. hardware-assisted virtualization support in modern processors [15]. Virtualization is also wasteful in terms of storage space: most SPs base their VMs on some standard base VM (e.g. some well-supported version of Linux, such as Ubuntu or CentOS), but each individual VM still gets a full individual copy of the entire file system, although the difference between what is stored in each VM is typically orders of magnitude smaller than the overall allocated storage for each VM.

In recent years, containers rather than VMs have started to gain momentum as the customizable program execution environment of choice (in particular after recent crucial additions were made to the Linux kernel in version 3.8 [27]). Containers are a type of virtualization that does not rely on virtualizing an entire machine, as VMs do, but rather on isolating user space system instances within a single (shared) operating system kernel. In Linux, support for containers is granted by cgroups. Cgroups provides resource (CPU, memory, various I/O) and process namespace isolation, isolating applications from another with regard to process trees, network connections, user ids, and (notably) mounted file systems. Essentially, this means that containerized processes are free to define their own file systems, init systems, and background services but must share the host operating system’s kernel. This is a limit to some applications (ones that rely on a custom kernel), but on the other hand, containerized processes are thereby provided with bare-metal access to the computer’s hardware. This significantly reduces both performance and storage overheads [28]. Because containers offer merely a type of additional operating-system level isolation layers between processes, they start within seconds, rather than minutes, since they do not require lengthy provisioning and boot-up processes — essentially, running a program in a container is just like running a local program [29].

Containers are nothing new, as they have existed in various operating systems before (e.g. Solaris Zones [30], and BSD Jails [31]). Useful support for them in Linux, however, is. After the 3.8 Linux kernel release, and with the introduction of Docker [32], Linux containers and clouds based on them have gained significant momentum. This is in part because Docker makes it easy to package applications and all their dependencies together in an image that can then run unmodified on any other system that supports Docker — without paying the performance penalty that is associated with VMs. Additionally, the way that Docker images are constructed, and because they are mounted on a union file system, only unique data requires additional storage: data that can be shared between images is shared.

As a previously mentioned potential drawback, by construction, containerized processes must share kernel with the host machine. This may be a limit for certain services operating on a low enough level to warrant customized kernels, but the vast majority of cloud services do not require customized kernels. However, as containers
can define their own storage trees up to and including even entire Linux distributions, it is possible to run e.g. a mix of CentOS and Ubuntu containers on a single host, albeit with a kernel that may not be tuned for either distribution.
CHAPTER 3
Cloud Resource Management

Cloud infrastructures typically comprise several physical hosts, each capable of hosting a number of VMs. The SP should not (have to) be aware of how large the pool of resources actually is, but regard it as seemingly infinite. IPs can collaborate by forming cloud federations to make larger (possibly hybrid) cloud platforms that make use of capacity at remote sites, should the local resources be exhausted. The rapid, on-demand self-service that cloud infrastructures are supposed to offer requires capable management software, with core features including the following:

- **Admission control.** By examining the service manifest and current (and predicted) resource availability, the cloud management software must determine whether a given service can be accepted or not. Service deployment is a long-term commitment, and since terminating a service once it has been accepted typically violates the SLA, admission control must be done in a risk-conscious way [33].

- **Placement optimization.** Continuously finding the best possible physical host machine for the set of running VMs is termed placement optimization. Depending on optimization criteria, this process may lead to VMs being migrated from one host machine to another, to achieve e.g. consolidation to power off an unused host machine to save energy [34], or to minimize load differences among physical hosts for fault-tolerance reasons.

- **Monitoring.** The usage and condition of VMs must be monitored to ensure that no SLOs fail to be met, and to ensure that usage is correctly accounted and billed for. SLOs may state that no VM should suffer more than X% downtime over some time period, or that a certain performance characteristic is always guaranteed. Monitoring should transparently demonstrate any failures to meet these agreed-upon objectives.

- **Orchestration.** Services contained in VMs should be deployable in a deterministic, well-defined, and repeatable way. This process is known as orchestration, and while it has been the topic of much research [35,36,37,38,39,40] (including...
our own), sophisticated multi-VM orchestration functionality has only recently been adopted and offered by commercial cloud vendors. Standardization efforts are also underway [41].

- **Storage.** Storage location and availability are important factors for VM performance and hence placement optimization, since VMs should ideally run close to where their backing storage (i.e. virtualized hard drives) are physically located, as this helps them perform better and decreases the internal network load [42]. In addition to the storage used directly by VMs, clouds typically need to offer a catalog of VM images that act as templates or starting points for making custom VMs, to avoid having to install every operating system from scratch, and some sort of object storage, that allows SPs to store various static data in a network-accessible way. Given the importance of the stored data, and the difficulty or cost associated with moving it to another cloud provider, it is often a cause of vendor lock-in [43]. Additionally, due to lack of insight and control over what entities have access to data stored in the cloud, potential cloud users are hesitant to fully use the cloud unless additional security measures are added [44, 45].

- **Networking.** VMs need to be network-accessible — not necessarily publicly, but without a network connection of some sort, they have no way of communicating outside of their hypervisor. Should the VM migrate, networks typically need to be reconfigured to avoid VM network connectivity loss (and the negative effects this may have on the software running inside). *Software Defined Networking* acts as a layer of additional abstraction that separates the software that decides where traffic is sent (control plane) from the underlying system that forwards the traffic to selected destinations (data plane) [46, 47]. In doing so, network connectivity and logical topology is made more dynamic in nature, allowing networks that best suit the services deployed in the cloud to be defined.

- **Accounting.** Since cloud resource consumption has to be paid for, cloud management services need to provide accounting features tied to the monitoring service. Accounting systems need to charge users in accordance with their usage, either in a pre- or post-paid fashion [48].

- **User management.** Cloud users must be authenticated and their actions authorized to ensure that they only perform actions they are allowed to. Some kind of user or identity management is therefore needed.

Large public cloud providers provide all of these services, in addition to various provider-specific ones. The research and open source communities have access to cloud platforms that also provide these services, such as OpenNebula [49], CloudStack, and OpenStack [50].

Of particular interest to this thesis are the following activities: placement, monitoring, orchestration, and, to a certain degree, storage. Paper I deals with placement and storage concerns as VMs are migrated to other clouds. Paper II presents solutions for problems in placement, orchestration, and monitoring. Finally, Paper III is focused on placement and orchestration, as an extension of the concepts introduced in
3.1 Placement Optimization

Placement is the process of determining which VMs should be provisioned on which hosts or partner clouds in a cloud federation [51]. We refer to the outcome of the process as a mapping between VMs and hosts. The algorithm driving the placement process typically optimizes some set of criteria. Typical such criteria include, e.g., using as few hosts as possible to make others available for other tasks or to power off unused ones [52, 53, 54] to reduce energy and operational costs, or distributing VMs evenly to ensure good performance [51, 55]. While VMs are isolated in theory, they suffer from the noisy neighbors problem, where large consumption of some resource (e.g. CPU processing power) in one VM negatively affects the amount of resources available to other VMs deployed on the same host, breaking isolation and possibly failing to meet SLOs. Industry experience and research has found that VM performance differs greatly depending on factors beyond the SP’s control [56], something which is attributed to poor isolation between neighboring resource-intensive VMs (i.e. other VMs deployed on the same physical host or on the same network subnet). Placement, and the choice of optimization criteria, is therefore crucial to providing a reasonable cloud service.

The placement process is executed not just when the set of VMs change due to allocation or termination of VMs, but as a continuous process. This is necessary because the total execution time of VMs is not known up front, unlike in the grid job case. To optimize placement, and perform (re-)consolidation, should host machines have become unevenly loaded, VMs can be migrated (while running) from one host to another to improve an already established mapping if it is determined that the current mapping is suboptimal. This process is known as (live) migration, and its use to facilitate placement is a hot research topic [57, 58, 59, 60, 62, 63, 64, 65, 66]. Live migration moves a VM from one host to another, and the capacity requirements for the VM is kept the same (i.e. it demands the same amount of RAM, CPU, storage, and network capacity from the new host as it did from the old).

Placement optimization (as opposed to greedily just finding any placement) is at its core an instance of the Generalized Assignment Problem, and is therefore NP-HARD to solve [67]. To further complicate matters, and increase potential income, cloud IPs may apply over-booking of the resources they actually have, while increasing the risk of failing to meet some performance SLOs, should simultaneous resource requirements actually exceed available capacity [68, 69, 70].

Most approaches so far have considered placement based on explicit resource requirements made by the SP up front in a service manifest, dealing with VMs as computational black boxes. However, looking in to these black boxes, and possibly modifying them slightly, has been shown to enable even further optimization methods. If the workload can be determined [71, 72, 73, 74, 75], so can the capacity requirements of the VMs, allowing for ahead of time adjustments to avoid the service being de-
ployed with too low capacity [76, 77, 78]. If a capacity demand profile for a VM can be determined, modifying the profile can be done in an effort to re-pack VMs to better provide them with the resources they actually require, so that they require more or less resources as the workload fluctuates [79]. Coupled with over-booking, this can offer IPs great possibilities to optimize without the SP suffering from poor performance.

Finally, in some cases, it is impossible to correctly predict or react to a change in workload. In an approach similar to supporting application checkpointing [80], wherein the program or service of interest is modified to allow for compensating for inadequate resources to carry out the task at hand, SPs can make certain computationally intensive sections of the service optional [81]. The reasoning is that it is better for end-users to get some kind of response, than none at all. Such a modified service may perhaps appear to be less dynamic due to failing over to only serve cached static responses than when resources are abundant.

### 3.2 Monitoring

Since cloud computing is defined as a pay-as-you-go service, monitoring resource usage is a crucial task. Monitoring occurs at three conceptual levels:

- **Infrastructure monitoring.** Core cloud management functionality such as the placement algorithm needs up to date information regarding the state of the infrastructure, including network utilization and the state of the physical hosts and the storage units attached to them. This data is typically not made public, apart from vague service health indicators. Problems that are detected need to be mitigated, possibly by migrating or restarting VMs away from an errant host machine.

- **Resource consumption monitoring.** SLAs stipulate the terms under which SPs obtain resources from IPs, and should the IP fail to deliver the agreed-upon capacity, some compensation is typically in order (although the potential loss in reputation and credibility may be higher and hard to quantify, hence the focus on avoiding failure to meet SLOs in the Placement section). VMs and their resource consumption are continually monitored, and the data should be transparently presented to the user. Transparency in this issue is a cornerstone of the trust between SPs and IPs.

- **Service monitoring.** A higher-level view on capacity requirements and resource consumption is to consider how well an service can handle its current workload. Rather than, as in resource consumption monitoring, which focuses on e.g. CPU instructions per second, service monitoring focuses on key performance indicators (KPIs), such as the number of currently logged in users or successfully served web requests. These measurements can typically be fed into the management functionality of the cloud, and be used as a basis for automatically adjusting the capacity allocation to better fit the current workload [78].
3.3 Orchestration

SPs submit a service manifest that describes the capacity they require from the IP. The IP then accepts or rejects the service after determining that there is sufficient capacity to host the service by invoking its placement algorithms. In the most trivial case, the SP requests each VM individually, in effect creating a service manifest for each. In non-trivial cases, these service manifests are documents that cover a large number of service components (VMs) and can include scaling directives, service-level objectives [26,82], and explicit relationships between service components [83,84].

In the research community, in particular the European line of research initiated by the RESERVOIR project [82] in 2008, rather complex and fully-featured service manifests are assumed to be available, since they allow both the SP and IP to reason about the service that is to be deployed as a whole, rather than in individual parts. Industry support for non-trivial service manifests and orchestration declarations have, however, been slow to emerge, but Amazon Web Services offers Cloud Formation [85] and recent OpenStack versions offer Heat [86], its Cloud Formation-compatible orchestration features. Orchestration standardization efforts are also underway in the shape of the OASIS Cloud Application Management for Platforms (CAMP) [87] and OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) [41,88,89].

3.4 Storage

While VMs can either be close-to-optimally placed from the start, or be reasonably easily migrated from one physical host to another, the data stores that VMs use for computation input or output may be orders of magnitude larger than the working memories of the VMs. While storage units to VMs typically reside on network-accessible nodes, network limitations and bandwidth considerations limit how VMs can be migrated to hosts typically on the same subnet or network switch as the storage node, as migration over the wide-area network (WAN) is technically possible but inefficient and warrants ongoing research [58,90,91]. Thus, storage management is key to ensuring good performance and ensuring data safety is of utmost importance. For fault-tolerance and performance reasons, storage is typically distributed [92,93]. For cloud bursting or intra-federation migration, the data needs to be exported to the target cloud to ensure reasonable performance (WAN links are too slow to perform disk input and output). Doing so, without duplicating the transfers needlessly, is discussed in Paper I.
CHAPTER 4
Service-oriented Architectures in the Cloud

Service-oriented architectures (SOAs) are part of a design philosophy that helps shape and motivate cloud deployment of services, although the concepts have evolved since the 1990’s and 2000’s when SOAs were first introduced. Papers II and III in this thesis deal with orchestration of services in cloud environments, motivated by the particularities of services designed for cloud deployment. This chapter serves as a brief introduction to what type of software services are deployed in clouds, and how cloud computing itself is the evolution of software services.

Services are, in the context of cloud computing, a term for a composition of a number of service components, each of which contributes a well-defined functionality to the functionality of the service as a whole\(^1\). Not all components have to be deployed in the same cloud, or even in any cloud (i.e. a publicly facing web site could be deployed in a public cloud, but access information through a secured connection to an on-premise database with sensitive data). In SOA terms, services are network-accessible software programs that have a well-defined functionality, offer access over various transports and through various serialization formats, and they are regarded as always being available [94]. They should be technology neutral for interoperability, loosely coupled, and location transparent [94]. Their utility is the defining characteristic, and this utility is made available via some kind of Application Program Interface (API). Services (and the components thereof) and their APIs were in the SOA vision supposed to be automatically discoverable via cataloging services/registries, and the interaction between services facilitated almost to the point of automation by complex but well-defined document formats fully describing the operation signatures of the entire API of each service. For a number of reasons, most of which hardly technical, this is not how modern services are presented and used. Instead, Representational State Transfer (REST) [95, 96], which is focused on the data that a service represents and offers methods to query and modify the data using the standard HTTP methods (GET, PUT, POST, and DELETE), has emerged as the most common way of publishing an API that allows others to make use of a service.

\(^1\)In a 2007 interview at the High Scalability Blog, the then CTO at Amazon stated that generating a single page involves about 100–150 backend service components.
4.1 Services Developed for Cloud Environments

Cloud computing is well-suited for hosting services, since each VM can be customized to suit a particular service component in terms of what software is installed and how much computational power the component needs. A large relational database may require a large amount of RAM, whereas a web frontend that coordinates several backend services and presents a document to a web site visitor, might not. But clouds also has certain specific traits that shape how services are developed for being deployed in cloud environments.

Cloud computing offers rapid, on-demand elasticity by allowing SPs to commission and decommission VMs. This is called horizontal scaling, as it increases the number of VMs. Modifying the capacity of individual VMs is referred to as vertical scaling. The multi-core evolution [97] and how easy cloud computing makes it to obtain more VMs has had a profound effect on how software is designed: software services need to support running in a distributed manner, which adds significant complexity in terms of fault-tolerance, communication between components, and data consistency [98]. It is also desirable that the performance of the service increases (at least) linearly with each additional added instance.

A component that often becomes a performance bottleneck and has undergone such a change is the database. With the new scalability opportunities offered by cloud computing, we have over the last few years seen the emergence of new data storage options including NoSQL [99,100] and NewSQL [101] data stores, to replace traditional relational SQL data stores. Their utility is sometimes questioned [102, 103], since they typically surrender some traits of SQL data stores (e.g. database-wide transaction consistency) to achieve their higher scalability and availability properties [104].

From experience, the ideal service component that benefits most from horizontal scalability is one designed to:

- perform a low number of quickly executed concise tasks, preferably just one;
- maintain no state information; and
- allow repeated idempotent calls.

If a service component is highly specialized and performs just a single, or a few related, tasks, increasing the number of deployed component instances gives a targeted boost to that particular function. Tasks should be concise, in that they do not solve several problems at once (this is a tenet of good object-oriented design, anyway). State information should be avoided, since it limits how well load can be distributed among instances: if only the instance \( I \) can serve requests related to a session it has already initiated, adding more instances cannot offload \( I \). It is to prefer that new instances can connect to any instance, new or old, and get the same results. Similarly, if a service component instance is shut down in the middle of processing a request, it should be able to issue the same command to a different instance and get the sought response. This can only be guaranteed if idempotent calls are supported, meaning that several repetitions of a particular request does affect the system beyond what a single request would. Obviously, state has to be maintained somewhere (e.g. in a data store designed
for horizontal scalability, or a distributed caching system), but keeping components as free from it as possible is the goal of new frameworks that are emerging based on these ideas, e.g. the *Play! framework* [105] for web services.

Managing the deployment of all these instances of horizontally scalable service components is precisely what we set out to do in this thesis. Service components are obviously related to one another, whether they carry out the same or simply related tasks, and how they are deployed affects their performance and fault-tolerance. Without a language for the SP to express these relationships and how service components should be deployed, a suitable deployment cannot be guaranteed by the cloud IP.
Chapter 5

Thesis Contributions

This thesis contains contributions to cloud management and service deployment in cloud federations in three main areas: service orchestration, placement optimization, and monitoring. Challenges in these areas for federated cloud environments include dealing with a lack of control and trust across organizational domains. The work presented in the papers in this thesis highlight and present possible solutions to such problems, both from the point of view of the infrastructure provider and the service provider.

The main lessons learned during the work with this thesis are:

- **Influence rather than control**
  The IP and SP have conflicting goals and optimization criteria: the SP would optimally have the entire cloud site to itself, no network congestion, and each VM deployed on an individual dedicated physical host. The IP wants to make optimal use of the infrastructure, which means deploying as many services by different SPs on the infrastructure as possible. For this very simple reason, the SP cannot be granted control over how the infrastructure is used. And in contemporary (public) clouds, they are not. However, for some services, complete lack of any way to express placement restrictions is unacceptable. Offering influence to SPs, rather than control, is a reasonable trade-off.

- **Migration costs differ**
  The cost of migration, in terms of VM performance drop and network utilization (increases in congestion), needs to be taken into account when VMs are to be migrated. All VMs are not created equal in terms of migration cost: determining which VM is most suitable to move must take a large number of factors into account if we are to avoid picking the wrong VM. In particular, if we have granted the SP increased influence over placement, we might end up having to migrate a set of related VMs, should we choose one of them. Due to the impact on performance VM migration demonstrably has, heuristically determining which VM to migrate leads to more optimal use of resources and less performance degradation in the cloud site as a whole.

- **Monitoring systems can improve substantially**
  Monitoring, both the infrastructure itself on behalf of the IP and the deployed services on behalf of the SP, currently leaves much to be desired in terms of
compatibility, security, and ease of use. More work is required, and the field is rife with competing approaches, making one of the most obvious corner stones of any management process an exciting field for future developments.

5.1 Paper I

Paper I [106] deals with problems related to offloading (migrating) virtual machines from one cloud to another in an effort to provide the best possible placement, and monitoring of virtual machines, to ensure that they are given the agreed upon resources in the service-level agreement.

While migration is typically performed between physical hosts within a single cloud site (data center), it can be performed across wide area networks [58, 107], and thus within cloud federations, as well. The principle of location unawareness, as described in works by Hadas et al. [108], and a motivating design feature in the RESERVOIR project [82], suggests that the actual placement of a VM should be as transparent to the VM as possible.

Should a cloud A need to migrate away a VM to a partner cloud, that partner may in turn contact its partners (unknownst to A) to check if they can host the VM. If so, A’s partner can accept responsibility for hosting the VM, but delegates it to one of its partners. And so forth. While operative commands to modify the VM’s state need to be forwarded along such a chain, actually transferring the entire storage along the chain would be inefficient. In Paper I, we propose a more efficient alternative based on transfer proxies, which allows the originating cloud to transfer large data files to the final destination cloud directly, although it is unaware of which particular cloud that may be, due to being obscured by the chain of delegation.

A novel way of extracting monitoring data from VMs with the express goal of hiding complexity, loosening the coupling between the monitored service and the monitoring system, and requiring only minor modification to the service deployed in the VM is also presented.

5.2 Paper II

Paper II [83] expands on finding optimal placement and on monitoring of virtual machines, but does so also from a user perspective: as a user of a cloud infrastructure, one should be granted some level of influence (as opposed to outright control) over how virtual machines are placed. We propose adding service structure to service manifests, where SPs can define relationships between VMs and specify rules stipulating which components may or may not be deployed on the same host, cloud site, or geographical region as some other related VM. However, the SP is not granted any control over the placement, and cannot force the IP to use a certain set of hosts for the service deployment.

Service structure is expressed in terms of affinity and anti-affinity constraints on the host-, cloud-, and (geographical) region-level to types or instances of a given type.
Affinity requires co-placement on the defined level, whereas anti-affinity forbids it. The type and instance division makes it possible to express, for instance, that a single VM containing a database replica has a cloud-level affinity to all other database replicas (type), but a host-level anti-affinity to other database replicas (instance). We modeled relationships using directed acyclic graphs, an approach formalized and evaluated in Paper III.

A heuristic for determining the *migratability* of a (set of) VMs in a given placement mapping was also introduced, as part of determining which VMs would be easier to move, given that service structural constraints can imply that migration of one VM may require others need to be migrated, as well.

As a way to bridge the gap between incompatible monitoring systems in cloud federations, we suggested a novel monitoring system based on monitoring values with attached semantic meta-data.

### 5.3 Paper III

Paper III [84] expands further on the concept of service structure defined in Paper II. We provided a formal definition of the directed acyclic graphs that express the service structure and, as an illustration, how they can be translated into placement constraints matrices for input to an Integer Linear Programming (ILP) solver; presented a mathematical model for incorporating these constraint matrices in a placement engine based on ILP; and we also demonstrated through simulations that the proposed level of influence offers reasonable additional orchestration capabilities to users, while not making the management of the cloud infrastructure overly complex.

We simulated a cloud site of 80 hosts, and 100 VMs belonging to a single service to be placed upon these. Since service structure according to our definition does not apply to components in other services, i.e. components from a Service A cannot have any relationship to components from Service B (where A \( \neq \) B, of course). Therefore, we simulated the presence of other services simply as a certain level of background load already deployed on each host. 15300 input permutations were performed, and the results show that there is a relationship between number of affinity constraints, background load, and placement algorithm execution time that matches intuition:

- a low amount of both constraints and background load makes placement quick and easy;
- more work is required if there are more complex affinity constraints at low background loads since the number of possible solutions increases; and
- placement of services with complex constraints on already loaded systems is also quick to compute, due to the low number of possible solutions.

Regardless, the computational impact on the placement engine is very low, and we argue that support for service structure should be added in future cloud orchestration tools.
CHAPTER 6

Future Work

The work presented in this thesis leaves a number of interesting topics to pursue, including (in addition to future work mentioned in the papers):

- **Modifying placement based on monitored application behavior**
  Monitoring of applications can provide hints towards the fitness (i.e. how good a placement mapping is) of the current placement mapping [109]. Large amounts of measured network traffic between two VMs can indicate that these should be placed close together, for instance. Large CPU consumption by a VM indicates that it should not be placed with other VMs that also consume large amounts of CPU. Coupled with predictive techniques, placement based on application insight can be made more intelligent and offer increased cloud infrastructure utilization. If the placement engine is already set up for dealing with service structures, modifications to placement can be done by inferring service structure, and taking this inferred service structure into account for upcoming placement optimization iterations.

- **Formal definition of migratability heuristic**
  In Paper II, we informally discuss the migratability heuristic. This should be formally defined, and continually improved upon by the research community as new possibilities for VM or container migration emerge.

- **Monitoring causally related events**
  As one of many improvements to contemporary monitoring systems, we would like to propose taking a lesson from distributed systems research in taking causal relationships between events into account for service monitoring. Isolating the cause of errors in services based on timestamped data is brittle due to the non-existence of perfectly synchronized clocks, and offers poor support for developers in determining the actual cause of an error condition. If causal relationships between events are part of the monitoring data stream, however, events in one causing errors in another can easily be identified. The main challenge is determining and distributing the information required for establishing causal relationships in an efficient manner.

- **Implementing service structure and evaluating experimentally**
  With the recent interest in orchestration in the OpenStack community, imple-
menting service structure in the OpenStack placement engine (the Nova Scheduler) would be a worthwhile endeavor. Doing so would allow one to perform true experiments, rather than simulations, and get data and further insight on what utility service structure offers, and at what cost.

- **Investigate orchestration of services running in containers**
  Containers are very well suited for deploying services and constructing service-oriented architectures. Due to the cheap isolation that practically comes for free, and how platform-related software such as application servers are to be delivered as containers, dividing a service into components that each run in separate containers will be the norm in the containerized cloud. But this also creates a demand for automatically provisioning, orchestrating, and managing these containerized services in an automated fashion. The requirements and future challenges that this gives rise to need to be studied further.
Bibliography


overbooking,” in Proceedings of the ACM Cloud and Autonomic Computing

elastic services in compute clouds,” in IFIP/IEEE International Symposium on

[71] R. Ghosh and V. Naik, “Biting off safely more than you can chew: Predictive
analytics for resource over-commit in iaas cloud,” in IEEE 5th International

computing applications,” Journal of Software Engineering and Applications,
vol. 4, no. 07, p. 396, 2011.

[73] A. Khan, X. Yan, S. Tao, and N. Anerousis, “Workload characterization and
prediction in the cloud: A multiple time series approach,” in IEEE Network
Operations and Management Symposium (NOMS), 2012, pp. 1287–1294, April
2012.

bursty scientific workloads on the cloud using adaptive elasticity control,” in
Proceedings of the 3rd workshop on Scientific Cloud Computing, pp. 31–40,

[75] A. Eldin, A. Rezaie, A. Mehta, S. Razroev, S. Sjöstedt-de Luna, O. Seleznjev,
J. Tordsson, and E. Elmroth, “How will your workload look like in 6 years?
Analyzing wikimedia’s workload,” in IEEE International Conference on Cloud

[76] T. Lorido-Botrán, J. Miguel-Alonso, and J. A. Lozano, “Auto-scaling tech-
niques for elastic applications in cloud environments,” Research EHU-KAT-IK,

[77] E. Caron, F. Desprez, and A. Muresan, “Forecasting for grid and cloud comput-
ing on-demand resources based on pattern matching,” in IEEE Second Interna-
tional Conference on Cloud Computing Technology and Science (CloudCom),

troller for cloud infrastructures,” in Network Operations and Management Sym-
posium (NOMS), 2012 IEEE, pp. 204–212, April 2012.

packing approach to the horizontal vs. vertical elasticity trade-off for cloud auto-
scaling,” in Proceedings of the 2013 ACM Cloud and Autonomic Computing


