Department of Computing
Cork, Ireland

Cloud Service Brokering for enterprise-ready, container-based Cloud Services

Constraint Programming based Docker container placement

by

Ruediger Schulze
e-mail: ruediger.schulze@me.com

Module: M.Sc. in Cloud Computing

March 28, 2016

Supervisor:

Dr Donagh Horgan

Abstract

Title: Cloud Service Brokering for enterprise-ready, container-based Cloud Services

Subtitle: Constraint Programming based Docker container placement

Author: Ruediger Schulze

By exploiting the portability of application containers, Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) providers receive the flexibility to deploy and move their cloud services across infrastructure services delivered by different providers. The aim of this research project is to apply the concepts of cloud service brokering to container-based cloud services and to define a method for the optimised placement of the containers in an environment with multiple Infrastructure-as-a-Service (IaaS) providers. In this report, the use cases for cloud service brokering in the context of enterprise-ready, container-based cloud services are described and a new placement method based on Constraint Programming (CP) is introduced for the deployment of containers across multiple IaaS providers, the benefits and limitations of the method are discussed, and a framework for demonstrating the feasibility of the proposed method is presented.

Keywords: Cloud Service Broker, Constraint Programming, Docker Container

Acknowledgements

The author wishes to express sincere appreciation to Dr Donagh Horgan for his assistance and feedback during the supervision phase of this research project. Furthermore, the author would like to thank the staff members of the Cork Institute of Technology for offering the opportunity to study Cloud Computing in an online programme and for running the classes in the evening which allowed the author to attend in the courses and to study alongside to his work as an employee.

Declaration of Originality

I hereby declare that this thesis and the work reported herein was composed by and originated entirely from me. Information derived from the published and unpublished work of others has been acknowledged in the text and references are given in the list of sources.

March 28, 2016

Rüdiger Schulze

Ruediger Schulze

Contents

Chapter 1

Introduction

Container-based applications are on the rise. Docker [1] as an open platform for application containers has become very popular since its first release in June 2014. With Docker, developers receive the option to package applications and their dependencies into self-container images that can run as containers on any Linux server. Multiple large Cloud Service Providers (CSPs) have embraced Docker as container technology and announced alliances with the Docker company. Container-based applications are highly portable and independent of the hosting provider and hardware.

Cloud services such as PaaS and SaaS have usually a topology that aligns with the components of a multi-tier architecture, including tiers for presentation, business logic and data access. The topology of the cloud service describes the structure of the IT service delivered by the CSP, the components of the service and the relationships between them. Cloud services designed for the use in enterprises have multiple Quality of Service (QoS) requirements such as High Availability (HA) and Disaster Recovery (DR) targets, performance and scalability requirements, and require compliance to security standards and data location regulations. The requirements for HA, DR and horizontal scaling are the drivers to design and deploy the cloud services in multi-node topologies which may span multiple availability zones and geographical sites.

PaaS and SaaS providers in public or private cloud environments receive with Docker the option to deliver their cloud services using container-based applications or microservices. The portability of the application containers enables them to easily move cloud services between IaaS providers and to locations where, for instance, the customer's data resides, the best Service Level Agreement (SLA) is achieved or where the hosting is most cost-effective. PaaS and SaaS providers will benefit from employing brokering services that allow them to choose from a variety of IaaS providers and to optimise the deployment of their cloud services with respect to the QoS requirements, distribution and cost. By using a Cloud Service Broker (CSB), PaaS and SaaS providers will be enabled to structure their offering portfolio with additional flexibility and to quickly respond to new or changing customer requirements. New options arise from the easy deployment of the containers across the multiple IaaS providers. A PaaS or SaaS provider may deploy a container to a new geographic location by selecting a different IaaS provider when the primary one is not available there. A globally delivered cloud service may be provided by using resources from multiple IaaS providers. DR use cases may be realized by selecting a different provider for the backup site. Load-balancing and scalability built into the cloud service will allow to gradually migrating the service from one provider to another one with no downtime by simply moving the containers to the new provider.

Cloud service brokering for application containers requires to define a new method for the optimised placement of the containers on IaaS resources of multiple providers. The method has to take the attributes of the containers and the IaaS resources into account, and honour the QoS requirements of enterprise-ready cloud services. The initial placement of the containers has to follow the specification of the topology of the cloud service. Aside of the attributes used for rating IaaS providers, the CSB has to consider container related attributes such as the built-in container support of the IaaS providers, the packaging of containers in virtual machines and the clustering of containers. The optimisation of the infrastructure resources of the container-based cloud services has to be without impact and visibility to the cloud service consumers. Access to user data and network connectivity to the cloud services have to be handled and delivered uninterrupted, and without the need to reconfigure clients.

A CSB for application containers may use a CP-based engine to make the placement decision about the optimal IaaS provider. The engine will use as input the requirements of the container and information about each of the available IaaS providers. The objective of this research project is to define a method that allows for the optimised placement of containers in a cloud environment with multiple IaaS providers and to demonstrate the feasibility of the proposed method with a framework for use by CSBs.

The research report is organised into the following chapters:

1. In the chapter *Background Research*, the concepts of cloud service brokering are investigated, and CP as method for solving placement problems in cloud computing environments is introduced.
2. The requirements and use cases of a CSB framework for application containers are described in chapter *Analysis and Design*. The major architecture decisions and the component model of the CSB framework are presented. A container-specific CP model is derived and the data model of the CSB framework is described.
3. The development details of the CSB framework are explained in the chapter *Implementation*. A description of the data access layer is given, the algorithms for finding solutions of the CP model are documented, and the use of Docker Cloud API [2] for the deployment of Docker containers on different IaaS providers is illustrated.
4. The test scenarios for verifying the CSB framework are summarised in the chapter *Verification and Analysis*, the population of the framework database with test data is explained, and the results of the verification are presented.
5. A summary of the project and conclusions are given in the last chapter of the report.

Chapter 2

Background Research

2.1 Cloud Service Broker

2.1.1 Definitions

The definition of a cloud broker as introduced by the National Institute for Standards and Technology (NIST) is widely referenced by other authors, e.g., in [3], [4], [5] and [6]. In the NIST Cloud Computing Reference Architecture (CCRA) [7], a cloud broker is described as follows:

> A cloud broker is an entity that manages the use, performance and delivery of cloud services and negotiates relationships between cloud providers and cloud consumers.

The cloud broker is an organisation that serves as a third-party entity as a centralised coordinator of cloud services for other organisations and enables users to interact through a single interface with multiple service providers [3] [8]. The role of the provider of a broker is emphasised in the definition of cloud service brokerage by Gartner [9]:

> Cloud Service Brokerage (CSBg) is an IT role and business model in which a company or other entity adds value to one or more (public or private) cloud services on behalf of one or more consumers of that service via three primary roles including aggregation, integration and customisation brokerage. A CSBg enabler provides technology to implement CSBg, and a CSBg provider offers combined technology, people and methodologies to implement and manage CSBg-relates projects.

2.1.2 Categorisation

NIST [7] and Gartner [10] define three categories of services that can be provided by a CSB:

Service Intermediation: Service intermediation enables a CSB to enhance a service by improving some specific capability and providing value-added services to cloud consumers. Service intermediation is responsible for service access and identity management, performance reporting, enhanced security, service pricing and billing.

Service Aggregation: A CSB uses service aggregation to combine and integrate multiple services into one or more new services while ensuring interoperability. The CSB is responsible for the integration and consolidation of data across multiple service providers, and ensures the secure movement of the data between the cloud consumer and the cloud providers. The key aspect of the service aggregation is to ease service selection and present services from separate providers as a unique set of services to the cloud service consumer.

Service Arbitrage: Service arbitrage is the process of determining the best CSP. The CSB has the flexibility to choose a service from multiple providers and can, for example, use a credit-scoring service to measure and select a provider with the best score. Service arbitrage adds flexibility and choice to service aggregation as the aggregated services are not fixed.

Six key attributes of CSBs are derived in [8] mainly based on the categories defined by NIST [7] and Gartner [9] [10], and used for evaluation of existing CSBs: *intermediation*, *aggregation*, *arbitrage*, *customisation*, *integration* and *standardisation*. According to [11], integration is focused on creating an unified, common system of services by integrating private and public clouds or bridging between CSPs. Customisation refers to the aggregation and integration with other value-added services, including the creation of new original services [11]. Both integration and customisation are closely interlinked with service aggregation and intermediation, and in-fact it is difficult to find distinct definitions of these attributes in the literature. Standardisation among the CSB mechanisms and across the cloud services of different providers enables interoperability, and supports the process of service selection by a CSB.

2.1.3 Architecture

2.1.3.1 Functional Architecture

A comprehensive model of a CSB architecture is described in [3]. The CSB environment is built of the same components as the management platform for a single cloud but additional complexity is introduced into the system by the requirement to support multiple CSPs. Bond [3] distinguishes between the functions of the Cloud Broker Portal, the Cloud Broker Operations and Management and the multiple CSPs, and aligns them in a layered model of a vendor-agnostic CSB architecture. Governance is introduced in addition as a set of functions which are orthogonal to the layers of the model and have to be realised for all functional components of the architecture. A CSB has to establish a vital API economy in order to promote its easy adoption into cloud applications.

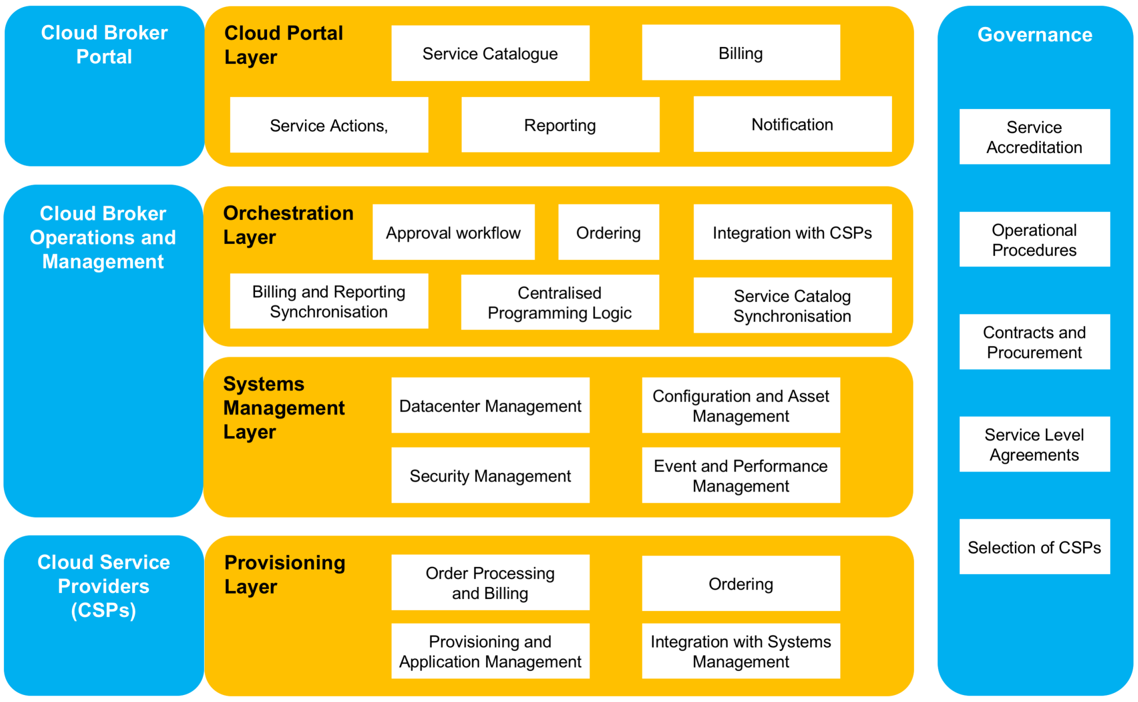

Cloud Broker Portal
Cloud Portal Layer
Service Catalogue
Billing
Service Actions,
Reporting
Notification
Cloud Broker Operations and Management
Orchestration Layer
Approval workflow
Ordering
Integration with CSPs
Billing and Reporting Synchronisation
Centralised Programming Logic
Service Catalog Synchronisation
Systems Management Layer
Datacenter Management
Configuration and Asset Management
Security Management
Event and Performance Management
Cloud Service Providers (CSPs)
Provisioning Layer
Order Processing and Billing
Ordering
Provisioning and Application Management
Integration with Systems Management
Governance
Service Accreditation
Operational Procedures
Contracts and Procurement
Service Level Agreements
Selection of CSPs

Figure 2.1: CSB architecture according to [3]

The CSB architecture according to [3] is shown in a simplified form in figure 2.1, and its components can be summarized as follows:

2.1.3.1.1 Cloud Broker Portal The Cloud Broker Portal is a multi-tenant web user interface and provides the capabilities for ordering services from the service catalogue, approval of requests and subscription management. Multiple dashboards allow to obtain information about SLA fulfilment, performance and status. Billing and financial reports support the management of expenses and enable budget tracking.

Cloud Portal Layer The Cloud Portal Layer provides the primary user interface to the cloud consumer and delivers the following features:

Service Catalogue The Service Catalogue contains a list and descriptions of all available cloud services and options, and displays pricing information to the consumers. The available cloud services can be aggregations from other CSPs and be built on function for service arbitration in order to determine the best service.

Billing Billing allows the consumers to track invoices, chargeback and budget balances using reports and dashboards.

Service Actions Services Actions allow a consumer to modify or change the state of existing cloud service instances which have been ordered from the service catalogue. Service Actions are a wide range of activities that can be applied to a service instance, e.g., the modification of service attributes such as the SLA, updates of the infrastructure resources such as storage mirroring attributes or the horizontal scale-out of a service, and as well backup and snapshot operations.

Reporting Reporting provides the consumers with various reports, e.g., about service consumption, service performance and SLA fulfilment.

Notification News and announcements are distributed by the CSB provider to the consumers via the notification features of the cloud broker portal. Notifications may include updates about the service portfolio of the CSB, information about the service availability and options for on-line chat.

2.1.3.1.2 Cloud Broker Operations and Management The CSB initiates service provisioning and collects data from the CSPs. The cloud services and their related assets and configuration are aggregated from all providers and tracked by the CSB. In order to ensure secure operations, the CSB has to implement functions for security monitoring, event logging, alerting and remediation. Compliance with the IT Infrastructure Library (ITIL) processes [12], including change management, supports to operate the CSB environment with control. Functions for communication with users and effective methods for responding to issues and incidents enable customer satisfaction. The use of an integrated operations console and continuous monitoring of the infrastructure and application resources introduce visibility and transparency into the management of the CSB environment.

Orchestration Layer The orchestration layer contains the workflows, logic and integration to interact with the CSPs of the provisioning layer and the components of the systems management layer. The orchestration layer provides abstraction across the functions of multiple, different CSPs and hides their intrinsic complexity from the cloud broker portal and the system management components.

Approval Workflow Depending of the customer's preferences, the ordering process for new services may require one or multiple approvals, e.g., from additional business user roles. The approval workflow handles the approval process. Once a new order is submitted, an e-mail is sent to the predefined approver(s). The e-mail contains a link to the approval form of the cloud broker portal which allows the user to approve or deny the order. Upon approval, the order is processed further.

Ordering Requests from the portal for new cloud services and for modifying or terminating existing instances are handled through ordering. Standardised Application Programming Interface (API) calls are used to transmit new orders and requests to the CSPs and to interact with the billing system. The ordering functions track the execution of the provisioning tasks, detect their completion and update the configuration, asset and management systems, and provide status and instance information to the portal.

Service Catalogue Synchronisation The CSB's service catalogue is regularly updated with the current service offering descriptions and prices from the cloud services providers. The synchronisation uses the APIs of the CSPs to pull the information, and can either occur on a scheduled basis or whenever a change is detected.

Billing and Reporting Synchronisation The orchestration system queries billing and service information, and the resource utilisation from the provisioning layer and provides the information to the portal, so that the customers can view their account balances and utilisation. Depending of the contractual agreements, the CSB may bill the customer for all consumed services, or billing may be handled directly between the customer and each of the CSPs. The billing information is provided to the costumers for all consumed service, independent of how billing occurs.

Centralised Programming Logic A centralised engine determines the CSP to be used for a given order from the service catalogue. The service selection can either occur dynamically through service arbitration, based on customer preferences or based on an intentionally hard-coded entry in the service definition that prescribes the use of a specific CSP.

Integration with CSPs The orchestration layer uses standardised APIs to integrate with the CSPs. As most of the CSPs use different APIs (either proprietary or open-standard), the orchestration layer has libraries that implement the standardised APIs for each of the CSPs on the base of the provider-specific APIs.

Systems Management Layer The system management layer contains the tools for managing the CSB environment and aggregates the functions and data related to the management of servers, network, security, events and performance from the CSPs:

Datacenter Management CSBs are not involved into the physical management of the data centres, as the related functions are in the responsibility of the CSPs. The CSB has to have an awareness of the physical setup of the CSPs in order to provide redundant services for HA, e.g., using availability zones. A CSB may offer services to manage virtual private datacenters and dedicated or virtual private network connections on behalf of the customer.

Configuration and Asset Management The configuration and asset information for every service, Virtual Machine (VM), bare metal server, network, disk and software ordered from the CSPs is stored by the CSB and updated automatically when there is a change performed. The data contains information about the relationships of the configuration items and assets to each other and how they are linked into the services delivered by the CSB. The data can be queried from the portal for reporting to give customers visibility into their cloud environment.

Security Management New VMs and bare metal servers must automatically be added to the security management systems in order to ensure compliance with the security standards, regulations and legislations required by the type of workload for which the CSB offers the related service. Security related events such as disabling a user or the termination of a service must be synchronised within the CSB environment instantly in order to avoid any vulnerabilities and potential threads.

Event and Performance Management The CSB monitors the network, compute and application resources for status, troubleshooting and performance management. Monitoring data such as the measured network throughput for a specific CSP may be used as input to the service arbitration performed by the CSB. For services which fully utilise the compute resources assigned to them, the CSB may initiate scale-out actions either automatically or after approval by the customer and add additional VMs to the service.

2.1.3.1.3 Cloud Service Providers (CSPs) The CSPs provision services on request of the CSB, allocate resources to the services, and manage the lifecycle and the QoS. The CSP gives administrative control to the CSB for the ordered services and delivers data about the SLA, status and performance to the CSB. Metering data related to the resource utilisation of the requested services is made available for consumption by the CSB's billing system. The CSB utilises APIs to integrate with the CSPs. The publication of well-defined and easy-to-use APIs by the CSPs is essential in order to foster the integration into various brokering solutions.

Provisioning Layer The provisioning layer is specific to each individual CSP. Each of the CSPs is built on its own unique cloud management platform and provides the following features via APIs to the CSB:

Order Processing and Billing The CSB places an order to the CSP and the CSP processes the order, provisions the requested infrastructure and application resources, and starts billing. Depending of the service ordered and the contract terms, the billing may be a fixed price per billing period or variably, usage-based priced. In the later case, metering data about the resource utilisation is used to calculate the bill and generate the invoice at the billing period. The billing activities, invoice history and account balances of each CSP are synchronised by the orchestration layer into the CSB, so that an user of the broker portal can view them using the billing and financial reports.

Control Panels The control panels represent the self-service web portals that are specific to each CSP. The portal of a CSP provides role-based access to the CSP for administrators and users. The CSB's administrator assigned to manage the CSP can perform administrative tasks using this portal. Depending of the services offered by the CSB, a cloud service user may have direct access to this portal as well in order to work with the services of the specific CSP.

Provisioning and Application Management These functions of a CSP are responsible for creating and managing the resources of the cloud services offered by the CSP. Depending of the type of CSP, e.g., IaaS or SaaS, these functions create new infrastructure resources such as VMs or install applications, add new application accounts, and allow to manage them.

Integration with Systems Management The CSP keeps track of resources by updating its own configuration, asset and managements systems whenever there is a new service ordered or a service instance modified, and allows the CSB to query the same information via APIs.

2.1.3.1.4 Governance As the ownership of traditional IT department tasks shifts when organisations start to use CSBs and move workloads to cloud services, governance becomes a significant functional area in order to ensure compliance with an organisation's standards and processes. The organisations have to negotiate with the CSB provider and potentially with the CSPs which of tasks are performed by the CSB or CSPs, and which of them remain within the organisation in order to achieve compliance. The essential governance functions include the following:

Security Accreditation The responsibility to ensure the security of their services resides with the CSB and the CSPs. They must establish the related security measures, monitoring and operations to prevent threads and remediate any potential vulnerabilities. The security attestation of the CSB and the related CSPs can be either performed by the organisation using the CSB or by a third-party independent organisation. For the attestation, it is important that the standards to be met are prescribed by the organisation using the CSB environment.

Operational Procedures As the normal operation of the infrastructure resources and applications is transferred to the CSB and the CSPs, the organisation has to determine what level of involvement into the operational monitoring and associated events is now required, so that the CSB can provide the related interfaces.

Contracts and Procurement The organisation must determine how cloud services are contractually acquired through the procurement process. The CSB provides a service catalogue with several offerings, but the organisation must approve them for use, and the content of catalogue must potentially be customised in order to meet the organisation's regularity and internal standards. With respect to contracting and who is legally liable, the organisation must understand the role of the CSB. In case the CSB only acts as a coordinator, the organisation has to contract with each of the CSPs themselves. If the CSB takes full responsibility for all subordinate CSPs, the contract will only be between the organisation and the CSB.

SLAs The organisation has to specify the SLA to be met by the CSB and the CSPs. It is recommended to standardise the SLA requirements across the organisation in order to avoid that SLAs have to be individually managed and controlled.

Ideally the cloud broker portal supports to specify the SLA at the time when a new services is ordered or a service instance is updated. By using the SLA as input and matching it to the ones of the CSP, the organisation can select a preferred CSP or leave the decision to the CSB. The monitoring and enforcement of the SLA fulfilment depends on the contractual agreements that the organisation has made. If the CSB is contractually obligated to deliver services with a specific SLA, it is in responsibility of the CSB to track and enforce the fulfilment of the SLA. If the organisation has established contracts with the CSPs directly, the CSB has no legal authority to enforce the SLA and the organisation has work with the CSP directly.

Selection of CSPs The organisation has to determine which CSPs (or which sites of a CSP) are selected for use by the CSB. Before using a particular CSP, the organisation may want to evaluate the CSP whether or not it meets the organisation's requirements. Legislation and regulations can require to include or exclude a specific CSPs from the selection by the CSB, e.g., in order to meet data location requirements. The selection of the CSPs to be used by an organisation, can either be done by the organisation itself, and require that the organisation makes its own business relationships and contracts with the CSP, or by the CSB in agreement with the organisation.

2.1.3.2 Taxonomy

A taxonomy of brokering mechanisms is given in [13]. *Externally managed* brokers are provided off-premise by a centralized entity, for instance, by a third-party cloud provider or SaaS offering. *Externally managed* brokers are transparent for applications using the services provided by the broker. *Directly managed* brokers are incorporated into an application, have to keep track of the application's performance and be built to meet the availability and dependability requirements of the applications. *Externally managed* brokers can be classified as follows:

SLA-based broker: A cloud user specifies the brokering requirements of a SLA in form of constraints and objectives, and the CSB finds the most suitable cloud service by taking into account the user requirements specified by the SLA.

Trigger-action broker: A cloud user specifies a set of triggers and associates actions with them. A trigger becomes active when a predefined condition is meet, e.g., the threshold of a specific metric is exceeded, and the associated action is executed. Actions can be, for instance, scale-out and provisioning activities, and may include bursting into a public cloud when there is a spike in the demand for computing capacity.

2.1.3.3 SLA-based Broker

The SLA is a contract between a CSP and the cloud service consumer that defines the parameters of the service to in order to ensure QoS. Five SLA-based brokers are surveyed and evaluated in [8], and a high-level architecture of a SLA-based CSB is derived from these projects. The following main components were identified:

Management: Management is the central component of the CSB and provides parameters and functions for managing resources, QoS and SLA, registry, identity and authentication, adoption, deployment and composition.

Discovery: The discovery component supports the selecting of services from multiple providers according to requirements of the cloud service consumer. The discovery component runs the processes for service discovery, service selection, match making and service ranking. Service discovery may also be used by the CSB to find SLA templates that meet the consumer's requirements.

Monitoring: The monitoring component observes the performance of the services, the QoS and the fulfilment of SLAs negotiated with the service providers and consumers. The component may include functions for system optimisation based on events generated by the monitoring functions.

Scheduler: The scheduler component supports the planning and automated provisioning of cloud services.

Repository: The repository stores the information related to service providers and services, cloud provider drivers, possible resources and catalogue entries. The repository stores also the SLA templates for use by the service selection and match making.

In the context of the interoperability of clouds, the following challenges are described in [14] and applied here to CSBs:

Federated SLA Management: In an environment with multiple cloud service providers, each provider is expected to have its own SLA management mechanism. A CSB has to establish federation of the SLAs from each CSP in order to set up and enforce a global SLA. Methods and protocols for the negotiation of dynamic and flexible SLAs between the CSB and the multiple CSPs are required. Another important issue is the enforcement of the SLAs in environments with conflicting policies and goals, e.g., a CSB may offer a service with a SLA for HA, while none of the providers are willing to offer such a service.

Federation-Level Agreement: In addition to the SLA, there can be a Federation-Level Agreement (FLA) that defines rules and conditions between the CSB and the CSPs, e.g., about pools of resources and the QoS such as the minimum expected availability.

SLA Monitoring and SLA Dependency: The CSB has to establish functions for matching the guaranteed QoS of cloud services offered by the CSPs with the QoS requirements of the end-user, and for monitoring that the promised QoS and SLA is provided to the cloud service consumer. The dependencies of a CSP to other providers have to be considered by the CSB as well. The QoS of a higher-layered service can be affected in cases when the CSP of the service itself uses external services. If one of the providers of the lower-layered services is not functioning properly, the performance of the higher-layered service may be affected and impact finally the SLA agreed by the CSB.

Legal Issues: The CSB has to guarantee the security, confidentiality and privacy of the data processed by the services provided. Within the country where the services are delivered, the CSB must comply with the legislation and laws concerning the privacy and security of the data. Therefore, the CSB has to implement geo-location and legislation awareness policies and enforce compliance with those policies. As part of the SLA management, services of specific providers can be avoided or agreement can be made that placing data outside of a given country is prohibited.

2.1.4 Projects

The results of a survey of CSB projects are described in [15]. The authors consider four categories of CSB technologies: *CSBs for performance* to address issues of cloud performance comparison and prediction, *CSBs for application migration* which provide decision support when moving applications to the cloud, *theoretical models for CSBs* which describe purely theoretical and mathematical techniques and *data for CSBs* that summarises providers of data and metrics available for use by CSBs. A comprehensive list of commercial CSB projects is given in [11]. Recent research about CSBs has a significant focus on service arbitration across numerous CSPs, in particular on optimising the allocation of resources from different IaaS providers. The use of arbitration engines enables CSBs to automatically determine the best CSP and service for any given customer order. The attributes considered in the optimisation process vary depending on the proposed method. Typically attributes for rating IaaS providers are: the supported operating systems and configurations, geographical location, costs and rates, bandwidth and performance, SLA terms, legalisation and security, compliance and audit [16] [17].

The placement of VMs in cloud and virtual environments is a critical operation as it has an direct impact on the performance, resource utilisation, power-consumption and cost. The subject of VM placement is widely discussed in the research literature. Detailed reviews of the current VM placement algorithms can be found in [18] and [19]. According to [19], the following approaches can be distinguished with respective optimisation objectives:

Mono-Objective Approach: Mono-objective methods are designed for the optimisation of a single objective or the individual optimisation of more objective functions, but one at a time.

Multi-Objective solved as Mono-Objective Approach: Multiple objective functions are combined into a single objective function. The weighted sum method is used most often by this approach. This method defines one objective function as the linear combination of multiple objectives. A disadvantage of this approach is that it requires knowledge about the correct combination of the objective functions – which is not always available.

Pure Multi-Objective Approach: A vector of multiple objective functions is optimised by this approach. Only a small number of methods are described in the literature which use a pure multi-objective approach for VM placement.

A broad range of different VM placement schemes (18 different in total) are analysed in [18]. Constraint programming based VM placement (see section 2.2.1) is described as one of the considered placement schemes. The following classification of the placement schemes is proposed:

Resource-aware VM placement schemes: The infrastructure resource requirements of the VMs are considered in the placement decisions by these schemes. Efficient resource-aware placement tries to optimally place VMs on the hosts, so that the overall resource utilisation is maximised. Most of the schemes consider CPU and memory resources, some network resources, and a minor number includes the device or disk I/O, or try to minimise the number of active hosts.

Power-aware VM placement schemes: Designed for the use by the CSPs, these schemes try to make cloud data centres more efficient and to reduce the power consumption in order to enable green cloud computing. The objective of these schemes is to reduce the number of active host, networking and other data center components. The methods include VM consolidation and packaging of VMs on the same host or in the same rack, and powering off not needed VMs, hosts and network components. The attributes considered by the power-aware schemes include the CPU utilisation (e.g., based on the states: idle, average, active and over utilised), the server power usage and the host status (running, ready, sleep, off), costs for network routing and data center power usage, and the distance between VMs.

Network-aware VM placement schemes: These schemes try to reduce the network traffic or try to distribute network traffic evenly in order avoid congestion. The placement schemes allocate the VMs with more or extensive communication on the same host, to the same switch and rack, or within the same data center in order to reduce the network traffic within the data center and across data centres. Most common is the consideration of the traffic between VMs by the network-aware VM placement schemes, some evaluate the traffic between the hosts and selected schemes try to minimise the transfer time between the VMs and data, and the distance between the VMs.

Cost-aware VM placement schemes: These schemes try to reduce the costs for the CSPs while considering the QoS of the cloud services and honouring the SLAs. The schemes use different types of costs as attributes, such as the VM, physical machine, cooling and data center costs, and the distance between the VMs and the clients.

Conceptually a similar approach is taken in [19] with the classification of the objective functions. Based on the study of 56 different objective functions, the classification into five groups of objective functions is described: *Energy Consumption Minimisation*, *Network Traffic Minimisation*, *Economical Costs Optimisation*, *Performance Maximisation* and *Resource Utilisation Maximisation*. Most of the publications are focused on single-cloud environments, i.e. for use by CSPs. Seven of the methods are suitable for multi-cloud environments, i.e. use multiple cloud computing data centres from one or more CSP. Only two articles take a broker-oriented approach.

Different methods are employed by CSBs for rating IaaS providers, e.g., genetics algorithms [20] [21] and rough sets [17] [22]. Multiple projects propose CSBs which take advantage of the different price options such as for on-demand, reservation and spot instances [23], examples can be found in [24], [25], [26] and [27]. There are a couple of academic and open-source implementations of CSBs, e.g., STRATOS [28], QBROKAGE [20] and CompatibleOne [29]. A summary of selected CSB projects and their underlying concepts is described in the following.

2.1.4.1 STRATOS

Using an application's topology, resource requirements and Key Performance Indicators (KPIs) as input, the STRATOS broker introduced in [28] supports the dynamic selection of CSPs during the application's runtime and enables to allocate resources from multiple CSPs for this application. The service arbitration to be performed by the CSB is described as Resource Acquisition Decision (RAD) problem. The RAD problem involves the selection of n resources from m CSPs, so that a set of constraints, requirements and preferences are met. The topology of the application is described in a Topology Descriptor File (TDF) which is input to a cloud manager that passes resource requests to the CSB. In order to compare services from different CSPs, KPIs as introduced by the Service Measurement Index (SMI) framework [30][31] are used. The SMI framework is hierarchical structured into seven major categories: accountability, agility, assurance, financial, performance, security and privacy, and usability. Each category is further refined with a set of attributes which are expressed as KPIs. The RAD problem is solved using a multi-objective approach with objective functions for cost and vendor lock-in. Vendor lock-in is measured here in form of the balancing of the application topology across the CSPs, i.e. the avoidance of lock-in where an entire topology runs on a single CSP. The proposed method uses as input the desired configuration as specified in the TDF, a set of objectives which are expressed through KPIs, and data from CloudHarmony [32] which provides benchmark, network and availability metrics about a large set of CSPs. The selection process of the CSP identifies in a first step the feasible configurations, and optimises the result in a second step with respect to the objectives. In a set of experiments, it was shown that the proposed broker distributes the workload across the CSPs while minimising cost, and makes appropriate decisions in response to varying workloads.

2.1.4.2 Adaptive Cloud Provider Selection

A method for adaptive cloud provider selection (ACPS) using linear programming models is proposed in [33]. The article emphasises that the CSP selection is dependent on the provider performance and the end-user perception. The performance is mainly influenced by the provider location, the time and the VM performance. The end-user perception, also described as Quality of Experience (QoE), is a subjective measure of the service compared to the user expectation and depends primarily from the KPIs chosen by the user. Possible KPIs are security, reliability, availability, cost, network performance and scalability of the service. The proposed CSP selection uses two algorithms. A simple cloud provider selection (SCPS) algorithm is used initially to determine the optimal CSP on the base of location, bandwidth, latency and cost, and by taking user specified weights for these attributes into account. A second algorithm is employed for the adaptive cloud provider selection. The algorithm detects if there is a decrease in the quality of the currently offered service and if another service gives more benefits to the user. The algorithm continually measures the current service for the KPIs, e.g. the network performance, and invokes the SCSP algorithm again if the benchmark score of the service falls short of a given break point. Using the measurement from multiple CSPs around the globe, it was possible to show that no conclusions can be drawn about the specific provider bandwidth and latency because of the dynamic aspects of the network, and that the proposed method provides a practical solution for real-time handling of dynamic changes in the provider performance.

2.1.4.3 MOGA-CB

A multi-objective genetic algorithm for a three tier cloud model is presented in [21] which considers two optimisation objectives. The proposed brokering method allows a client to submit requests with QoS requirements and assigns these requests to VMs of different sizes. In order to compare the different flavours of VMs, a performance index is introduced as normalisation value. The method uses a concept of client satisfaction and profit in order to measure the impact of the optimisation. Two objective functions are defined for minimising the VM cost and the response time. It was shown that the proposed method allows to find an optimal solution while trading off between the response time of the VMs and their cost, and that the client satisfaction in this model is more related to the response time rather than the VM cost. It was proven that an orientation on the broker profit decreases significantly the client satisfaction.

2.2 Constraint Programming

2.2.1 Definitions

Constraint programming is a form of declarative programming which uses variables and their domains, constraints and objective functions in order to solve a given problem. The purpose of constraint programming is to solve constraint satisfaction problems as defined in [34] and [35]:

Definition 2.2.1 (Constraint Satisfaction Problem) *A Constraint Satisfaction Problem $\mathcal{P}$ is a triple $\mathcal{P} = (X, D, C)$ where X is an n-tuple of variables $X = (x_1, x_2, ..., x_n)$, D is a corresponding n-tuple of domains $D = (D_1, D_2, ..., D_n)$ such that $x_i \in D_i$, C is a t-tuple of constraints $C = (C_1, C_2, ..., C_t)$.*

The domain D_i of a variable x_i is a finite set of numbers, and can be continuous or of a discrete set of values. In order to describe a constraint satisfaction problem $\mathcal{P}$, a finite sequence of variables with their respective domains is used together with a finite set of constraints. A constraint over a sequence of variables is a subset of the Cartesian product of the variables' domains in the scope of the constraint.

Definition 2.2.2 (Constraints) *$C = (C_1, C_2, ..., C_t)$ is the set of constraints. A constraint C_j is a pair $\left(R_{S_j}, S_j\right)$ where R_{S_j} is a relation on the variables in $S_j = scope(C_j)$, i.e. the relation R_{S_j} is a subset of the Cartesian product $D_1 \times ... \times D_m$ of the domains $D_1, D_2, ..., D_m$ for the variables in S_j.*

A solution of the Constraint Satisfaction Problem $\mathcal{P}$ is defined as follows:

Definition 2.2.3 (Solution of $\mathcal{P}$) *A solution to the Constraint Satisfaction Problem $\mathcal{P}$ is a n-tuple $A = (a_1, a_2, ..., a_n)$ where $a_i \in D_i$ and each C_j is satisfied in that R_{S_j} holds the projection of A onto the scope of S_j.*

The definition of multiple global constraints such as the `alldifferent` constraint is described in the literature. The constraint `alldifferent` requires that the variables $x_{1,2}, ..., x_n$ take all different values. An overview of the most popular global constraints is given in [36].

2.2.2 Projects

Several publications focus on the use of CP-based cloud selection and VM placement methods. A method for cloud service match making based on QoS demand is introduced in [37]. CP is a convenient method for optimising the placement of VMs, as placement constraints can be directly expressed by variables representing the assignment of the VMs to the hosts and the allocation of resources for the VMs placed on each host. Resource-aware VM placement schemes are presented in [38], [39] and [40]. A combined CP and heuristic algorithm is utilised in [39]. Special focus is put on fault tolerance and HA in [40]. In [41], the CP-based, open source VM scheduler BtrPlace [42] is used to exhibit SLA violations for discrete placement constraints, as these do not consider interims states of a reconfiguration process. As consequence, BtrPlace is extended with a preliminary version of continuous constraints and it is proved that these remove the temporary violation and improve the reliability. Power-aware methods are discussed in [43] and [44]. In the remainder of this section, selected CP-based methods are reviewed with respect to the definition of the variables and constraints, and the implementation of the CP model.

2.2.2.1 Cloud Service Match Making

The method for cloud service selection described in [37] supports the specification of list-typed QoS properties and preferences for deviating property values, and uses a CP solver which also allows for fuzzy service requests. The service descriptions are modelled as vectors of QoS parameters, i.e. a QoS parameter is represented as a variable of the CP model. The description of the constraints is based on the definition of the `element` constraint as given by [36]:

Definition 2.2.4 (Element constraint) *The constraint element*(i, l, v) *expresses that the i-th variable in a list of variables* $l = [x_1, ..., x_n]$ *takes the value of* v*, i.e.* $x_i = v$*.*

The proposed CP models finally use the more generic implementation of the `element` constraint as given by the Java Constraint Programming API Specification (JSR-331) [45] which allows the use of operators, e.g., to express that all variables $l = [x_1, ..., x_n]$ are smaller than the value v. The implementation of different CP models is described in [37]:

1. *Discrete value matching with hard constraints* which requires an exact match

between the QoS parameters of the service specification and the QoS demand. (The possibility of interval matching is discussed but not evaluated further.)

2. *Discrete value matching with soft boolean constraints* which determines for each service specification a violation sum. The violation sum is the sum of QoS parameter weights for those QoS parameters that don't match the respective constraint. The service with the minimum violation is returned by this CP model.

3. *Discrete value matching with soft constraints using distance* which allows to specify approximated values for the QoSs requirements, and takes the distance between the QoS parameters of the service specification and the QoS demand into account.

4. *Feature list matching* which determines a matching degree for mapping a set of provided features given by a QoS parameter of the service specification against the set of requested features of a QoS demand, and uses ranking rules to assign a score to the service definition based on the matching degree. The objective is to maximise the score of the feature list matching.

In order to combine the four CP models, the calculation of a final score for each service is proposed. The final score of a service can be determined as sum of the scores for each QoS parameter minus the violation sum of the same QoS parameter.

2.2.2.2 Virtual Cloud Resource Allocation

A CP-based method for virtual cloud resource allocation (VCRA-CP) is described in [38]. The model considers user specified QoS requirements in form of the average application response time and the cost of the virtual cloud resources. The method assumes an environment with heterogeneous applications $A = (a_1, a_i, ..., a_m)$ for which VMs have to be provisioned and packaged on a set of hosts $P = (p_1, p_j, ..., p_q)$. In order to run the applications, c different types of VMs $S = (s_1, s_k, ..., s_c)$ can be allocated. Each VM type s_k is specified with a tuple $s_k = (s_{k,cpu}, s_{k,mem}, s_{k,bandwidth})$ in order to describe the VM resources. The method uses a two phase approach. In the first phase, the VM provisioning phase, a virtual machine allocation vector N_i is determined for each application a_i, wherein the vector $N_i = (n_{i,1}, n_{i,k}, ..., n_{i,c})$ denotes the number of VMs of each type s_k to be provisioned for the application a_i. In the provisioning phase, the objective function U_{cost} is used to minimise the cost for all resource while honouring the performance satisfaction level u_i of each application a_i, and a set of constraints is defined to limit the number of VMs per type and application, and to honour the physical resource limits of the hosts. The performance satisfaction level $u_i = 1$ if the performance goal is met, otherwise $u_i < 1$. The objective function U_{cost} is given as

$$U_{cost} = minimise \sum_{i=1}^{m} \big(\alpha_i \cdot (u_i - 1)^2 + \epsilon \cdot cost(N_i)\big) \tag{2.1}$$

where the weight α_i is the importance of resource allocation for the application a_i and the coefficient ϵ allows to make a trade off between the performance and the resource cost. The resource cost function $cost(N_i)$ is given with

$$cost(N_i) = \frac{\sum_{k=1}^{c} n_{i,k} \cdot \big(0.7 \cdot s_{k,cpu} + 0.2 \cdot s_{k,mem} + 0.1 \cdot s_{k,bandwidth}\big)}{\sum_{j=1}^{a} \big(0.7 \cdot c_{j,cpu} + 0.2 \cdot c_{j,mem} + 0.1 \cdot c_{j,bandwidth}\big)} \tag{2.2}$$

where $c_{j,cpu}$, $c_{j,mem}$ and $c_{j,bandwidth}$ are the CPU, memory and bandwidth capacity of the host p_j and the coefficients 0.7, 0.2, 0.1 are the cost-weights of CPU, memory and bandwidth. In the second phase, the VM packaging phase, for each virtual machine allocation vector N_i are the packaging vectors H_j assigned that model the placement of the VMs on the hosts. The objective of the VM placement phase is to minimise the number of physical hosts. The vector $N_i = (n_{i,1}, n_{i,k}, ..., n_{i,c})$ with the number of VMs of each VMs type s_k is converted into a VM vector $V = (vm_1, vm_l, ..., vm_v)$ which has a distinct representation of each VM, and the vector $R = (r_1, r_l, ..., r_v)$ is created with the resources $r_l = (r_{l,CPU}, r_{l,memory}, r_{l,bandwitdh})$ of each VM vm_l. The packaging vector $H_j = (h_{j,1}, h_{j,l}, ..., h_{j,v})$ is a bit vector which denotes for a host $p_j \in P$ the VMs assigned to it (if $h_{j,l} = 1$, VM vm_l runs on the host p_j). The constraints defined for the packaging of the VMs are focus on ensuring the resource limits of the hosts are honoured and that each VM is assigned to a host. The objective function for minimising the number of hosts M counts the hosts which have at least one VMs assigned to them, and is defined as follows:

$$M = minimise \sum_{j=1}^{q} m_j,\ where\ m_j = \begin{cases} 1 & \exists\, vm_l \in V\ and\ h_{j,l} = 1 \\ 0 & otherwise \end{cases} \tag{2.3}$$

The method is designed to run repeatedly. By comparing the vectors N_i of the VM provisioning phase with the one of the previous cycle, the VMs to be created or removed can be determined. The method is validated through simulation experiments and it is shown that it is able to efficiently allocate and manage the virtual resources.

2.2.2.3 Guaranteeing High Availability Goals

HA specific properties are considered by the VM placement approach described in [40]. The authors introduce the concept of k-resiliency. As long as there are only up to k host failures within a cluster of hosts, it should be guaranteed that a VM marked as k-resilient is relocated to a non-failed host without relocating other VMs. Other publications such as [46] use the concept of k-fault tolerance which is focused on service availability and defined as:

> A k-fault tolerant service must be configured with k additional servers such that the minimum server configuration for the service can still be satisfied when k hosting servers fail simultaneously.

The approach of k-resiliency allows to specify a HA property indicating the level of k-resiliency at the VM level and is a cold stand-by technique, i.e. only one instance of a VM is running at a time and in case of a host failure, the VM has to be restarted on a stand-by host. The natural expression of k-resiliency constraints is based on second order logic statements and uses relations between subsets. The direct implementation of the k-resiliency problem as CP constraints is not feasible as it requires solving an exponential number of sub-problems and can not be done by the CP engine in a timely manner. The authors of [40] propose to transforms the k-resiliency constraints into a simpler constraint satisfaction problem by utilising *shadow VMs*. Shadow VMs are placeholders, i.e. a VM can be relocated to its shadow in the event of a host failure. In order to establish k-resiliency for a VM, k shadows of the VM have to placed on the available hosts, and all shadows of the VM and the VM itself must be anti-collocated. Unlike regular VMs, shadow VM can *overlap*, i.e. share the same resources if it can be guaranteed that, regardless of which hosts fails, their original VMs will never be evacuated together to those shadows. In order to enable shadow overlaps, each failing host and each shadow of a VM is assigned an unique index. In case the i-th host encounters a failure, the VMs on that host are evacuated to the location of their i-th shadow. As a consequence, the shadows with the same index can overlap for VMs which are originally placed on different hosts because the VMs' host receive a different index on failure and hence, the VMs are relocated to shadows with a different index. In order to solve the placement of the VMs and the shadow VMs, the following input is assumed:

1. A set $V^* = \{v_i\}_{v \in V, 0 \leq i \leq l_v}$ that includes all VMs $v \in V$ and l_v shadow VMs $v_1, ..., v_{l_v}$, and a set of resource requirements $\{r_{v,q}\}_{v \in V, q \in Q}$ of the VMs, wherein VM v_0 denotes the original VM and Q is the set of resource types.
2. A set of hosts H and resource capacities $\{c_{h,q}\}_{h \in H, q \in Q}$.
3. Resource constraints that require $c_{h,q}$ of each host h to be greater than the sum of the resources of VMs placed on the host and the VMs that might be placed on the host h.
4. Anti-location constraints $AL' \subseteq H \times V^*$ which require that the VMs v are never located on the hosts h.
5. Anti-collocation constraints $ACL' \subseteq V^* \times V^*$ which require that two VMs v and u are never located on the same host h.

The objective of solving the constraint satisfaction problem is to find a feasible, initial placement $\mathbb{P}^0(V^*, H)$ for all VMs v and their shadows which satisfies the resource constraints and locations constraints AL' and ACL'. The authors of [40] formally proved their approach and showed the advantages of the proposed method compared to simple standby host solutions with simulations.

Chapter 3

Analysis and Design

In this chapter, the requirements of a CSB for application containers are analysed and the design of a framework for the optimised placement of containers across different IaaS providers is described.

3.1 Requirements Analysis

PaaS and SaaS providers want to deploy container-based cloud services in a cost-effective manner in multiple geographies and with a guaranteed QoS. A CSB framework for Docker containers has to be provided that uses a service arbitration engine to determine the most cost-effective placement of the containers while honouring the QoS requirements for HA and DR.

3.1.1 System Context

The system context diagram of a CSB for Docker containers is shown in figure 3.1. Two user roles interacts with the CSB. The *CSB admin* performs administrative and management tasks of the CSB. The *CSB admin* registers the CSPs for use by the CSB and obtains reports about the usage of the CSPs from the CSB. As a result of the CSP registration, the CSB receives the attributes and prices of the services delivered by the CSP. The CSB downloads information about available Docker images from the Image Repositories. The *CSB user* requests the deployment of selected Docker images with specific QoS requirements from the CSB, and the CSB orchestrates the deployment of the containers. In order to run the containers, the CSB requests infrastructure services such as VMs from the CSPs. The CSPs provision those services and return them back to the CSB. The CSB completes the deployment of the containers and delivers the related endpoint information to the *CSB user.*

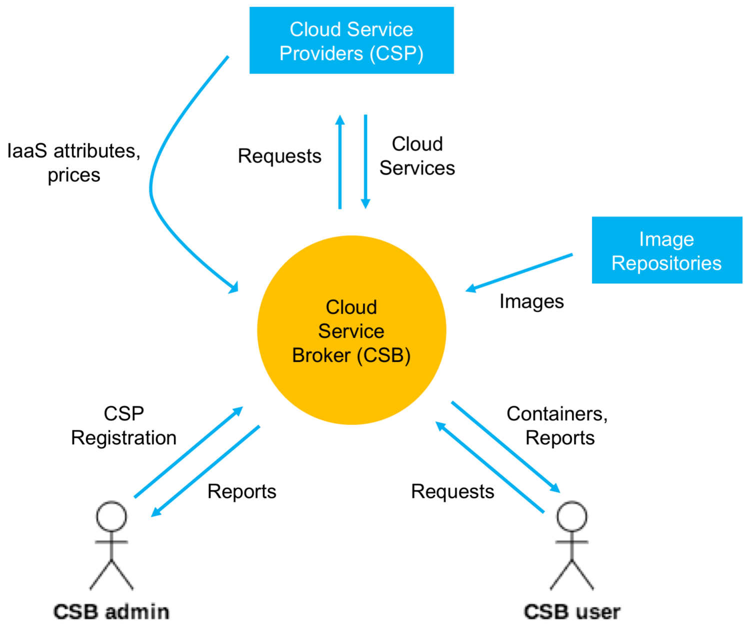


Figure 3.1: System Context Diagram of a CSB for Docker containers

3.1.2 Use Cases

The use cases to be supported by the CSB for application containers are shown in figure 3.2

3.1.2.1 Administrative Use Cases

1. *Register CSP:* As a CSB admin, I want to register a new CSP in the CSB, so that containers can be deployed to this CSP.
2. *Disable CSP:* As a CSB admin, I want to disable a CSP in the CSB, so that no new containers can be deployed to this CSP but any runtime information about containers hosted by this CSP is retained by the CSB.
3. *Enable CSP:* As a CSB admin, I want to enable an CSP in the CSB, so that new containers can be deployed to this CSP again.
4. *Deregister CSP:* As a CSB admin, I want to de-register a CSP from the CSB after all containers hosted by this CSP were removed, so that any information

about the CSP is removed from CSB and no new containers can be deployed to this CSP.

5. *Update CSP:* As a CSB admin, I want to update the information about a CSP stored by the CSB, so that the new information will be used from now on.

3.1.2.2 Deployment Use Cases

1. *Deploy Container:* As a CSB user, I want to deploy a container using the CSB, so that the CSB selects the CSP and host at the lowest cost, and places the container on the host.
2. *Deploy Containers with HA:* As a CSB user, I want to deploy n containers of the same image using the CSB, so that the CSB selects n hosts in different data centres of the same region or in different availability zones, the hosts are made available at the lowest cost, and each container is placed on a different host.
3. *Deploy Containers with DR:* As a CSB user, I want to deploy n containers of the same image using the CSB, so that the CSB selects n hosts in n different regions of the same CSP or different CSPs, the hosts are made available at the lowest cost, and each container is placed on a different host.
4. *Delete Containers:* As a CSB user, I want to delete one or multiple containers using the CSB, so that the resources used by the container are released and the host of the container is removed if no other containers run on the host.

3.1.2.3 Reporting Use Cases

1. *Display CSP report:* As a CSB admin or CSB user, I want to request a report about the CSPs used by the CSB, so that the primary attributes of the providers and prices are shown to me.
2. *Display container report:* As a CSB admin or CSB user, I want to request a report about the containers hosted by the CSB, so that the CSPs hosting the containers, the primary attributes of the containers, runtime information and the associated cost are shown to me.
3. *Display packaging report:* As a CSB admin or CSB user, I want to request a report about the packaging of the containers on the VMs acquired by CSB, so that the distribution of the containers across the VMs and the associated costs are shown to me.

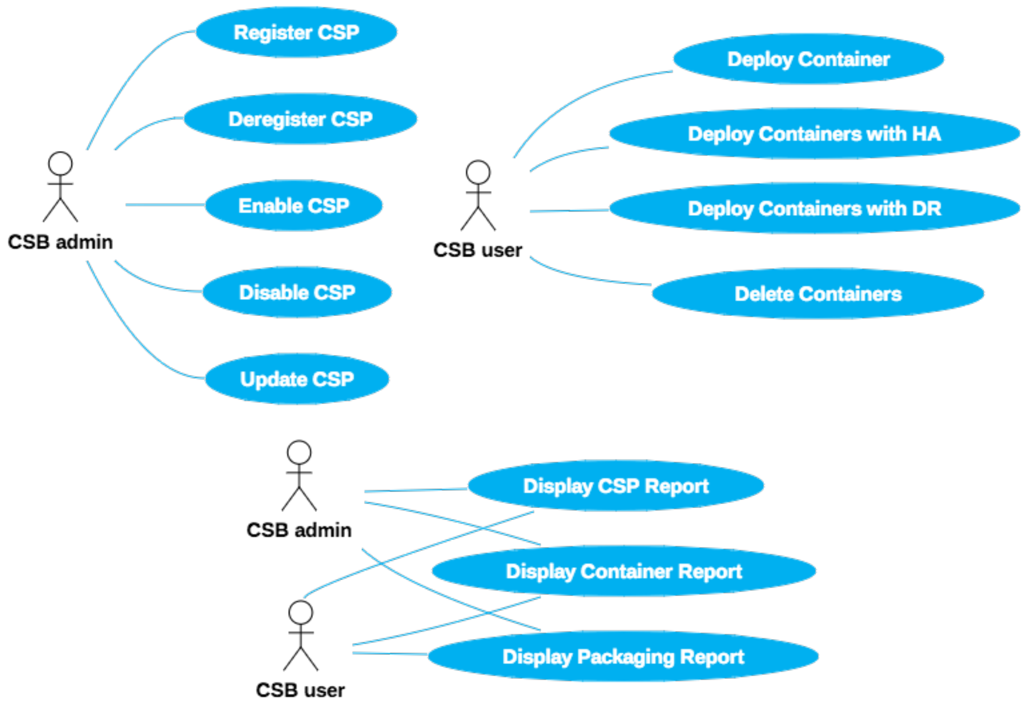


Figure 3.2: Use Cases to be supported by the CSB

3.2 Architecture Decisions

3.2.1 Use of Container Hosting Offerings

Context

Aside of IaaS providers which allow to order virtual or physical machines, there are a number of newly founded companies which offer Docker hosting services on top of their own infrastructure. An example of such an offering is sloppy.io [47]. The larger CSPs step into this market as well, as offerings such as Google Container Engine [48] and IBM Containers for Bluemix [49] show. Not all companies succeed in this new market of hosting services. Orchard turned down business and joined Docker [50]. The website of StackDock [51] is no longer reachable.

Options

1. Build the CSB framework on top of Docker hosting services offered by newly founded companies which use their own infrastructure.
2. Build the CSB framework on top of Docker hosting services provided by the

larger companies like Google [48] and IBM [49].

3. Build the CSB framework on top of established IaaS providers like Amazon Web Services (AWS) [52] or IBM SoftLayer [53], with the option to use Docker hosting offerings which directly integrate with those IaaS providers.

Decision

Build the CSB framework on top of established IaaS providers in order to lower the risk for the project. The container hosting services are relatively new in the market and do not offer yet the variety of deployment options like traditional IaaS offerings, e.g., in terms of geographical distribution. Once the container hosting services are further developed, they become a reasonable option for future projects.

Consequence

1. The deployment of a container will be performed in a two-step process. In the first step, the infrastructure resources have to be requested from one of the IaaS providers. In the second step, the container is deployed on those resources which includes the installation of the Docker engine on the related virtual or physical machine.

3.2.2 Use of Docker Cloud

Context

CSPs have usually different, often proprietary APIs. As discussed in [54], an abstraction layer is required in order to interface from the CSB framework with multiple, different CSPs.

Options

1. Design and develop interfaces to each CSPs as part of the CSB framework.
2. Use Docker Cloud [55] as common layer for deploying containers on IaaS resources of different CSPs.
3. Use Rancher [56] as platform for deploying containers on any infrastructure.

Decision

Use of Docker Cloud because it is available as a ready-to-use cloud service today, provides integration with multiple IaaS providers via a common API and handles all aspects of the container deployment. Rancher is currently in Beta status. It provides a valid alternative for future use, especially as it supports advanced features for modelling HA requirements using an annotated Docker Compose format [57].

Consequences

1. The CSPs for use by the CSB framework are limited to the IaaS providers supported by Docker Cloud.
2. The configurations delivered by the IaaS providers are limited to those which are supported by Docker Cloud.
3. Additional charges may occur through the use of Docker Cloud.

3.2.3 Use of SLAs

Context

Different IaaS providers support different SLAs. As pointed out in section 2.1.3.3, a CSB has to establish federation of the SLAs from each CSP in order to set up and enforce a global SLA. On the other hand, the CSB has no legal authority to enforce the SLA if organisations establish contracts with the CSPs directly (see section 2.1.3.1.4).

Options

1. CSB framework to provide support for a global SLAs based on federation of SLAs from different CSPs.
2. CSB framework to not support any specific SLAs and leave the contracting of SLAs to the exploiters of the framework.

Decision

CSB framework to not support any specific SLAs because most of the IaaSs providers do not give a SLA for single virtual machines and due to lack of standards which allow for SLAs negotiation.

Consequences

1. Exploiters of the CSB framework may contract own SLAs that are specific to their services and guarantee the SLA through the use of scalable, fault-tolerant application topologies, and use the framework's capability to distribute containers according to their QoS requirements.
2. An exploiter of the CSB framework may choose to disable specific providers in case these don't fulfil the requirements for the SLA that the exploiter would like to give.

3.2.4 Modelling of QoS requirements

Context

The QoS requirements for DR and HA are often expressed in terms of anti-collocation or k-fault tolerance (see section 2.2.2.3). A method for expressing the QoS requirements is required.

Options

1. Add attributes to each container deployment request to express location affinity to a specific host and anti-collocation to other containers.
2. Add a k-fault tolerance attribute to the container deployment request and let the CSB framework built k redundant instances of the container in different data centres or available zones in one region for HA and across multiple regions for DR.
3. Add the attributes *ha_scale* and *dr_scale* to the container deployment request. The attribute *ha_scale* describes how many container instances have to be deployed for HA across different data centres or availability zones within one region, and *dr_scale* is the number of instances that have to be deployed for DR across multiple regions of the same or multiple CSP.

Decision

Add the attributes ha_scale and dr_scale to the container deployment request because k-fault tolerance can be expressed using the attributes *ha_scale* and *dr_scale* as the sum of k and the minimum configuration required by the service. The specification of explicit location affinity and anti-collocation attributes for each container becomes an option, once an integration of the CSBframework with Rancher [56] is planned.

Consequences

All container instances are deployed with the same application and resource characteristics.

3.3 Component Model

The primary focus of the CSB framework is on the functional components *Centralised Programming Logic* and *Integration with CSPs*, as they are proposed by the CSB architecture in section 2.1.3.1. The component model of the CSB framework is shown in figure 3.3. The *CSB user* specifies the requirements for the deployment of a Docker image with a request that is input to the *Deployment Planner*. The planner uses a CP solver to derive placement decisions based on the requirements specified in the request and data available from two registries. The *IaaS Provider Registry* provides information about the different IaaS providers, regions, server sizes and prices. The *Runtime Registry* holds information about existing hosts. The *Deployment Engine* takes the placement decisions of the planner as input, and determines and executes the deployment steps. The deployment of the containers is performed using Docker Cloud [55]. Once the deployment is complete, the *Runtime Registry* will be updated with information about the used hosts and the new containers.

3.4 Data Model

The data model of the CSB framework is shown in figure 3.4 and a description of the database tables is given in the tables 3.1 and 3.2.
The following assumptions are made in design of the data model:

1. The CSPs are registered by the *CSB admin* in the CSB and stored in the table `Provider`. Each CSP supports the selection of predefined configurations which determine defaults for the number of CPUs, memory and the disk size. In addition, attributes like the region, availability zone and disk type, either HDD or SSD, can be selected in one or the other form. The various types of servers which can be selected by combining those different attributes are stored for each of the CSPs in the table `HostTemplate`. The CSPs use different schemes

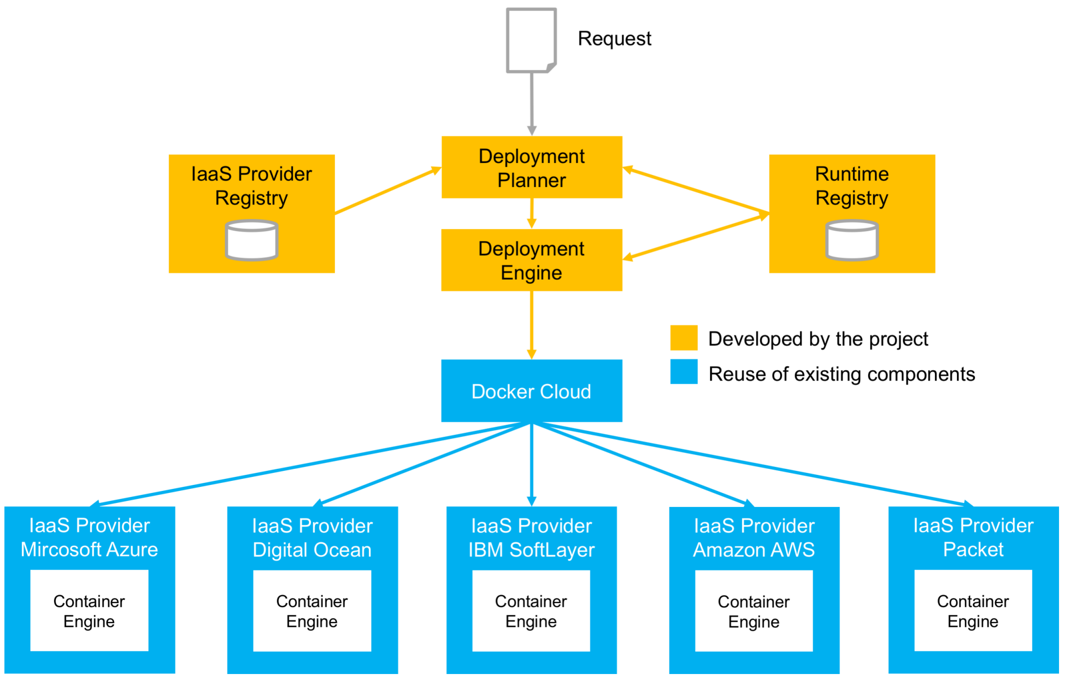


Figure 3.3: Components of the CSB framework

for representing the values of the host attributes, e.g., regions are named for one CSP as the cities where the data centres are located, and for others as the country or the geographical areas. In order to match the attribute values, they are stored in the database in a generalised form, i.e. all regions are referred to as the country of data centre, or in case of the U.S., as there are most of the data centres, as the name of the U.S. state. The attribute values in the `HostTemplate` table are numeric unique identifiers in order to allow processing by the CP engine. Descriptive names of the attribute values are available from the related tables `Provider`, `Region`, `DataCenter`, `AvailabilityZone`, `DiskType`, `HostType` and `OptimizationType`.

2. The requests made by the *CSB user* are stored in the table `Request`. As result of the request processing, one or multiple containers are placed on a set of hosts. The hosts acquired from the CSPs by the CSB are stored in the table `Host`. The `Host` table is linked to `HostTemplate` table, so that it can be determined which template was used for the request of a host. A host can run multiple containers. The containers are stored with the information about their resource consumption in the table `Container`. The free capacity of a host can be calculated later by calculating the total sum of the resources consumed by the containers running on that host.

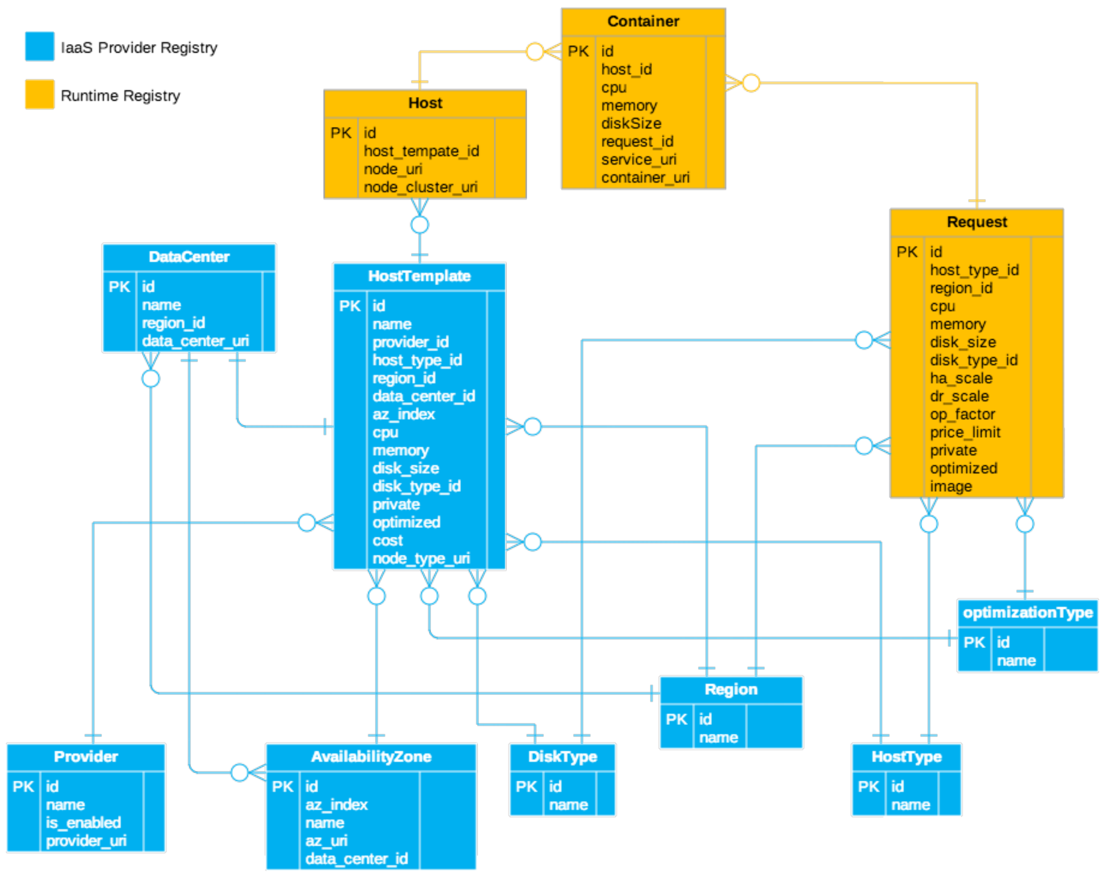

IaaS Provider Registry
Runtime Registry
Container
PK id host_id cpu memory diskSize request_id service_uri container_uri
Host
PK id host_tempate_id node_uri node_cluster_uri
Request
PK id host_type_id region_id cpu memory disk_size disk_type_id ha_scale dr_scale op_factor price_limit private optimized image
DataCenter
PK id name region_id data_center_uri
HostTemplate
PK id name provider_id host_type_id region_id data_center_id az_index cpu memory disk_size disk_type_id private optimized cost node_type_uri
optimizationType
PK id name
Region
PK id name
Provider
PK id name is_enabled provider_uri
AvailabilityZone
PK id az_index name az_uri data_center_id
DiskType
PK id name
HostType
PK id name

Figure 3.4: Data Model of the CSB framework

3. Most of the entities of the data model have a related representation as resources of the Docker Cloud API [2]. The tables contain therefore the Uniform Resource Identifiers (URIs) of those resources.

3.5 CP Model

One of the aims of the project is to define a method for optimising the deployment of container-based applications across different CSPs. The proposed CP model was initially prototyped using the MiniZinc IDE [58]. The objective is to find for a given Docker container c the optimal host h. A host h is a virtual or physical machine that meets the requirements of the container c and is delivered by an IaaS provider. The optimisation problem to be solved can be described as transformation of a container

Table	**Description**
`Provider`	Table for storing the registered CSPs and indication if a CSP is enabled
`Region`	Countries or states of the United States (U.S.) where at least one CSPs has a data centre
`AvailabilityZone`	Table of the availability zones, currently only supported by AWS [52]
`DataCenter`	Table of the data centres within a region
`DiskType`	Table with the available disk types, currently *Hard Disk Drive (HDD)* and *Solid State Disk (SSD)*
`HostType`	Tables with the available host types, currently *virtual* and *physical*
`HostTemplate`	Table with the host templates which can be provisioned by the different CSPs, includes information about costs
`OptimizationType`	Table with the types of infrastructure resources for which the CSPs offer performance optimisation

Table 3.1: Tables of the IaaS Provider Registry

Table	**Description**
`Request`	Table with the requests submitted by the *CSB user*
`Host`	Table with the hosts acquired by the CSB framework
`Container`	Table with data about the containers deployed by the CSB framework

Table 3.2: Tables of the Runtime Registry

domain $\mathbb{C}$ into a host domain $\mathbb{H}$:

$$f : \mathbb{C} \rightarrow \mathbb{H},\ c \mapsto h,\ where\ c \in \mathbb{C}\ and\ h \in \mathbb{H}. \tag{3.1}$$

The requirements of a container c are expressed as vector $r_c = (r_{c,1}, r_{c,i}..., r_{c,n})$. Each attribute $r_{c,i}$ is an element or subset of a domain R_i with a finite number of elements. Likewise, the attributes of a host h are described by the a vector $a_h = (a_{h,1}, a_{h,j}, ..., a_{h,m})$ for which each attribute $a_{h,j}$ belongs to a domain A_j and $A = (A_1, A_j, ..., A_m)$. In order to solve the optimisation problem, the requirements $r_{c,i}$ of the containers and the attributes $a_{h,j}$ of the hosts have to be considered. As method for finding the optimal host h for a given container c, a CP model is used. As per definition 2.2.1, a constraint satisfaction problem $\mathcal{P}$ is defined as the triple $\mathcal{P} = (X, D, C)$. The objective of the CP model is to provide solutions for the container placement problem $\mathcal{P}_{placement}$. The variables X and the corresponding domains D of the problem $\mathcal{P}_{placement}$ are defined by the attribute domains A of the hosts $\mathbb{H}$, i.e. $D = A$ and $X = (a_1, a_j, ..., a_m)$ where $a_j \in A_j$. Provided that the function *index* returns the index set I of any given set S, so that

$$index : S \to I,\ s_i \mapsto i = index(s), \tag{3.2}$$

then can be defined for the variables and domains of the host attributes the following:

$$\begin{aligned}
provider: &\quad a_1 \in A_1 \text{ and } A_1 = index(\mathbb{P}) \text{ where} && (3.3)\\
&\quad \mathbb{P} = \{\text{AWS}, \text{DigitalOcean}, \text{Azure}, \text{SoftLayer}, \text{Packet}\}\\
host\ type: &\quad a_2 \in A_2 \text{ and } A_2 = index(\mathbb{T}) \text{ where} && (3.4)\\
&\quad \mathbb{T} = \{\text{physical}, \text{virtual}\}\\
region: &\quad a_3 \in A_3 \text{ and } A_3 = index(\mathbb{R}) \text{ where} && (3.5)\\
&\quad \mathbb{R} = \{\text{Australia}, \text{Brazil}, \text{Canada}, \text{France}, \text{Germany}\}\\
&\quad \mathbb{R} = \mathbb{R} \cup \{\text{Great Britain}, \text{Hongkong}, \text{India}, \text{Ireland}\}\\
&\quad \mathbb{R} = \mathbb{R} \cup \{\text{Italy}, \text{Japan}, \text{Mexico}, \text{Netherlands}, \}\\
&\quad \mathbb{R} = \mathbb{R} \cup \{\text{Singapore}, \text{California}, \text{Iowa}, \text{New Jersey}\}\\
&\quad \mathbb{R} = \mathbb{R} \cup \{\text{New York}, \text{Oregon}, \text{Texas}, \text{Washington}\}\\
&\quad \mathbb{R} = \mathbb{R} \cup \{\text{Virginia}\}\\
data\ centres: &\quad a_4 \in A_4 \text{ and } A_4 = \{1, .., 52\} && (3.6)\\
availability\ zone: &\quad a_5 \in A_5 \text{ and } A_5 = \{0, .., 5\} && (3.7)\\
cpu: &\quad a_6 \in A_6 \text{ and } A_6 = \{1, .., 40\} && (3.8)\\
memory\ (GB): &\quad a_7 \in A_7 \text{ and } A_7 = \{1, .., 488\} && (3.9)\\
disk\ size\ (GB): &\quad a_8 \in A_8 \text{ and } A_8 = \{1, .., 48000\} && (3.10)\\
disk\ type: &\quad a_9 \in A_9 \text{ and } A_9 = index(\mathbb{D}) \text{ where} && (3.11)\\
&\quad \mathbb{D} = \{\text{HDD}, \text{SSD}\}\\
private: &\quad a_{10} \in A_{10} \text{ and } A_{10} = \{0, 1\} && (3.12)\\
optimised: &\quad a_{11} \in A_{11} \text{ and } A_{11} = index(\mathbb{O}) \text{ where} && (3.13)\\
&\quad \mathbb{O} = \{\text{compute}, \text{memory}, \text{gpu}, \text{storage}\}\\
&\quad \mathbb{O} = \mathbb{O} \cup \{\text{network}, \text{none}\}\\
cost: &\quad a_{12} \in A_{12} \text{ and } A_{12} = \{1, .., 99999\} && (3.14)
\end{aligned}$$

In order to describe the constraints C, the requirements $r_c = (r_{c,1}, r_{c,i}..., r_{c,n})$ of a container c have to be detailed first. The properties *ha_scale* and *dr_scale* are introduced to address the requirements for HA and DR. The requirement *ha_scale* describes how many containers have to be deployed across the data centres or available zones within one region and *dr_scale* is the number of containers that have to be deployed across multiple regions of the same or multiple providers in order to achieve protection against a disaster. The *op_factor* $r_{c,9}$ allows to request a larger host which can run $r_{c,9}$ instances of the container c. The *price limit* ($ 0.0001 per hour) specifies the maximum cost of host per hour which must not be exceeded. The attribute *private* $r_{c,11}$ allows to request to place the container c on a dedicated host.

$$host\ type: \quad r_{c,1} \in R_1 \text{ and } R_1 = A_2 \tag{3.15}$$

$$region: \quad r_{c,2} \in R_2 \text{ and } R_2 = A_3 \tag{3.16}$$

$$cpu: \quad r_{c,3} \in R_3 \text{ and } R_3 = A_6 \tag{3.17}$$

$$memory\ (GB): \quad r_{c,4} \in R_4 \text{ and } R_4 = A_7 \tag{3.18}$$

$$disk\ size\ (GB): \quad r_{c,5} \in R_5 \text{ and } R_5 = A_8 \tag{3.19}$$

$$disk\ type: \quad r_{c,6} \in R_6 \text{ and } R_6 = A_9 \tag{3.20}$$

$$ha\_scale: \quad r_{c,7} \in R_7 \text{ and } R_7 = \{1, .., 5\} \tag{3.21}$$

$$dr\_scale: \quad r_{c,8} \in R_8 \text{ and } R_8 = \{1, .., 5\} \tag{3.22}$$

$$op\_factor: \quad r_{c,9} \in R_9 \text{ and } R_9 = \{1, .., 10\} \tag{3.23}$$

$$price\ limit: \quad r_{c,10} \in R_{10} \text{ and } R_{10} = A_{12} \tag{3.24}$$

$$private: \quad r_{c,11} \in R_{11} \text{ and } R_{11} = A_{10} \tag{3.25}$$

$$optimized: \quad r_{c,12} \in R_{12} \text{ and } R_{12} = A_{11} \tag{3.26}$$

$$image: \quad r_{c,13} \in R_{13} \text{ and } R_{13} = \mathbb{I} \text{ where } \mathbb{I} \text{ is the set of Docker images} \tag{3.27}$$

In order to allow the placement of a container c either on an existing host are a new host, the domain $\mathbb{H}$ to be searched is defined as the union of the host templates T, i.e. the set of hosts which can be provisioned, and the already provisioned hosts H:

$$\mathbb{H} = T \cup H \tag{3.28}$$

wherein T is the superset of the templates t^p from all providers $\mathbb{P}$:

$$T = \bigcup_{p \in \mathbb{P}} T^p \text{ and } t^p \in T^p \tag{3.29}$$

The template t^p is described by the attribute vector a^p_t and domains A^p:

$$a^p_t = (a^p_{t,1}, a^p_{t,j}, ..., a^p_{t,m}) \text{ and } a^p_{t,j} \in A^p_j \tag{3.30}$$

$$A^p = A^p_1, A^p_j, ..., A^p_m \text{ and } p \in \mathbb{P} \tag{3.31}$$

The set of already provisioned hosts H is the union of the deployed hosts h^p at all providers $\mathbb{P}$:

$$H = \bigcup_{p \in \mathbb{P}} H^p \text{ and } h^p \in H^p \tag{3.32}$$

Provided that the host h^p is provisioned using the template h^p and that $\mathbb{C}_h$ denotes the set of containers deployed on the host h, the available resources on the host h^p can the be determined as follows:

$$cpu: \quad a_{h,6} = a^p_{t,6} - \sum_{c \in \mathbb{C}_h} r_{c,3} \tag{3.33}$$

$$memory(GB): \quad a_{h,7} = a^p_{t,7} - \sum_{c \in \mathbb{C}_h} r_{c,4} \tag{3.34}$$

$$disksize(GB): \quad a_{h,8} = a^p_{t,8} - \sum_{c \in \mathbb{C}_h} r_{c,5} \tag{3.35}$$

$$cost: \quad a_{h,12} = \frac{a^p_{t,12}}{|\mathbb{C}_h|} \tag{3.36}$$

The variables X represent the attributes of a single host. In order to respond to the requirements for HA and DR, multiple containers have to be placed on k anti-collocated hosts. $X_{e,f}$ denotes the variables and $a^{e,f}_j$ the attributes required to describe the k hosts $h_{e,f}$:

$$k = dr\_scale \cdot ha\_scale \tag{3.37}$$

$$X_{e,f} = (a^{e,f}_1, a^{e,f}_j, ..., a^{e,f}_m) \text{ where } a^{e,f}_j \in A_j \tag{3.38}$$

$$1 \leq e \leq dr\_scale \tag{3.39}$$

$$1 \leq f \leq ha\_scale \tag{3.40}$$

The constraints C can then be defined as follows:

$$\begin{aligned}
\textit{hosts}: &\quad C_1: \vee\Big(\wedge\big(a_j^{e,f} = a_{h,j}\big)\Big),\ \forall h \in \mathbb{H} \text{ and } 1 \leq j \leq 12 && (3.41)\\
\textit{host type}: &\quad C_2: \begin{cases} a_2^{e,f} = r_{c,1}, & \text{if } r_{c,1} \geq 0 \\ \textit{true}, & \text{otherwise} \end{cases} && (3.42)\\
\textit{region}: &\quad C_3: \begin{cases} a_3^{e,f} = r_{c,2}, & \text{if } r_{c,2} \geq 0 \\ \textit{true}, & \text{otherwise} \end{cases} && (3.43)\\
\textit{cpu}: &\quad C_4: a_6^{e,f} \geq \big(r_{c,3} \cdot r_{c,9}\big) && (3.44)\\
\textit{memory}: &\quad C_5: a_7^{e,f} \geq \big(r_{c,4} \cdot r_{c,9}\big) && (3.45)\\
\textit{disk size}: &\quad C_6: a_8^{e,f}\big(\geq r_{c,5} \cdot r_{c,9}\big) && (3.46)\\
\textit{disk type}: &\quad C_7: \begin{cases} a_9^{e,f} = r_{c,6}, & \text{if } r_{c,6} \geq 0 \\ \textit{true}, & \text{otherwise} \end{cases} && (3.47)\\
\textit{price limit}: &\quad C_8: \begin{cases} a_{12}^{e,f} \leq r_{c,10}, & \text{if } r_{c,10} \geq 0 \\ \textit{true}, & \text{otherwise} \end{cases} && (3.48)\\
\textit{private}: &\quad C_9: \begin{cases} a_{10}^{e,f} = r_{c,11}, & \text{if } r_{c,11} \geq 0 \\ \textit{true}, & \text{otherwise} \end{cases} && (3.49)\\
\textit{optimised}: &\quad C_{10}: \begin{cases} a_{11}^{e,f} = r_{c,12}, & \text{if } r_{c,12} \geq 0 \\ \textit{true}, & \text{otherwise} \end{cases} && (3.50)\\
\textit{HA}: &\quad C_{11}: \Big(\big(a_5^{e,f} \neq a_5^{e,g}\big) \wedge \big(a_4^{e,f} = a_4^{e,g}\big) \wedge \big(az_{\textit{enabled}} = 1\big)\Big) \vee && (3.51)\\
&\quad \Big(\big(a_4^{e,f} \neq a_4^{e,g}\big) \wedge \big(a_3^{e,f} = a_3^{e,g}\big)\Big) \\
&\quad \text{where } 1 \leq g \leq \textit{ha\_scale} \text{ and } f \neq g \\
\textit{DR}: &\quad C_{12}: \big(a_3^{d,f} \neq a_3^{e,g}\big) && (3.52)\\
&\quad \text{where } 1 \leq d \leq \textit{dr\_scale} \text{ and } d \neq e
\end{aligned}$$

The property $az_{\textit{enabled}}$ indicates that the used technology generally supports the deployment of hosts into availability zones. The objective function $f_{\textit{cost}}$ for minimising the cost across the k hosts is defined as follows:

$$f_{\textit{cost}} = \textit{minimise} \sum a_{12}^{d,f}. \tag{3.53}$$

Chapter 4

Implementation

The implementation of the CSB framework is described in this chapter. The framework is implemented in Python and uses NumberJack [59] as CP platform, the Python SDK of the Docker Cloud API [60] for deploying containers and SQLAlchemy [61] for database access.

4.1 IaaS Provider and Runtime Registry

4.1.1 Database Schema

The database schema `cbroker` of the CSB framework defines both the tables of the IaaS Provider and the Runtime Registry. The tables are created following the description of the data model in section 3.4 and are shown in figure 4.1. In addition to the tables, the following views are created:

`host_domain`: View containing all hosts h of the domain $\mathbb{H}$.

`host_used_cap`: View providing information about the consumed CPU, memory and storage capacity of the hosts $h \in H$, calculated as the sum of the resource consumption of the containers running on the hosts.

`host_free_cap`: View providing the information about the free CPU, memory and storage capacity of the hosts $h \in H$, i.e. the attributes $a_{h,6}$, $a_{h,7}$ and $a_{h,8}$.

The database schema of the CSB framework is implemented for MySQL [62]. For the development of the schema, a MySQL database instance running as Docker container [63] was used.

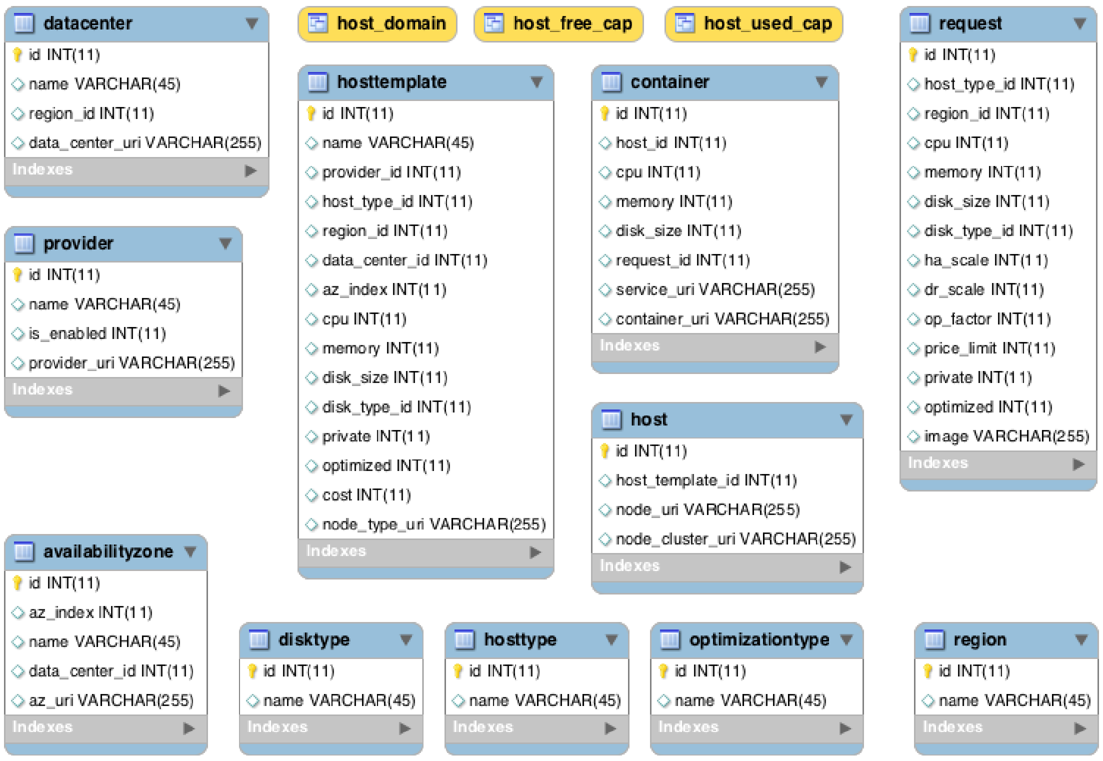


Figure 4.1: Database Schema of the CSB framework

4.1.2 Data Access Layer

The access to the CSB database is built on the functionality of the SQLAlchemy toolkit [61]. The class diagram of the data access layer is shown in figure 4.2. The CSB framework provides access to the databases tables by the class `CBrockerDB`. This class has methods for each of the database tables to return a SQLAlchemy `Table` object. The SQLAlchemy `Table` class allows to run for the associated database table `select`, `insert` and `delete` statements, and returns the related result set. In order to obtain an object of the class `CBrockerDB`, the class `DBConfig` has to be used. The class `DBConfig` manages the connection parameters of the database, and reads the database Uniform Resource Locator (URL) and the credentials from a configuration file. A data access object class `*Dao` and a data object class is defined for each entity of the data model. The `*Dao` classes provide methods for loading database records into related data objects and for saving objects into the database. The host attribute related classes such as `Region` are derived from the superclass `BaseAttribute` which defines the basic attributes `id` and `name`. The generalised classes `Resource` and `HostSpecification` inherit the common attributes of the classes `Request`, `HostTemplate` and `Container`. As an example, the use of the class

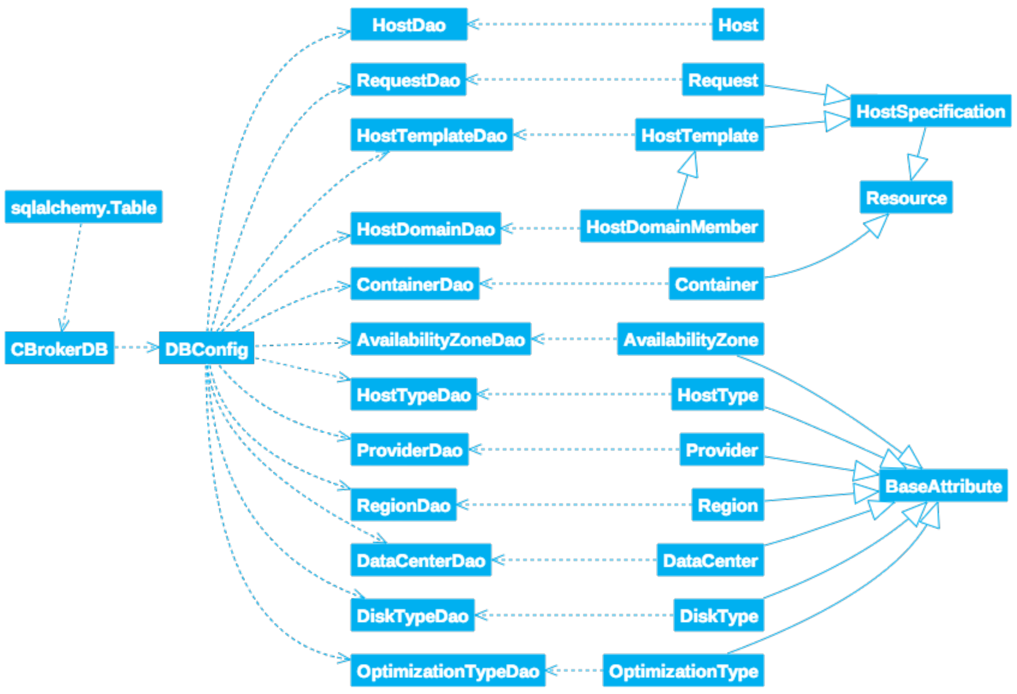


Figure 4.2: Class Diagram of the Data Access Layer

`HostDao` for saving a host into the database and for loading it from the database is illustrated in the following:

```
from de.sonnenfeldt.cbroker.model.host import Host
from de.sonnenfeldt.cbroker.db.hostdao import HostDao
# obtain Host object and assign values to attributes
host = Host()
host.host_template_id = host_template_id
host.node_uri = node_uri
host.node_cluster_uri = node_cluster_uri
# obtain HostDao object for the host object and save it into the database
dao = HostDao(host)
host_id = dao.save()
# read the Host object for a given host_id from the database
same_host = dao.load(host_id)
```

Listing 4.1: Use of the class `HostDao`

4.2 Deployment Planner

The classes involved in the deployment planner are shown in figure 4.3. The class `CBPlanner` is the central one of the deployment planner. The method `process()` of this class runs the algorithm for initialising the CP model with the variables and the constraints, and invokes the CP engine to solve the placement problem. The variables of the CP model are combined into the `CBVariables` class, so that multiple instances of the variables can be allocated when multiple hosts have to be placed. The variables are defined as instances of the `NumberJack.Variable` class. The algorithm performed by `CBPlanner.process()` is given on page 50. The algorithm 1 invokes the methods of the class `CBModel` to add the specific constraints to the CP model. Algorithm 1 returns a set of hosts on which the requested container has to be deployed. The algorithm 2 for adding the constraint C_1 is performed by `CBModel.add_host()` and can be found on page 51. The constraints C_2 to C_{10} are added by `CBModel.add_request()`, and C_{11} and C_{12} by `CBModel.add_anti_collocation()`. For the two algorithms 3 and 4, see the pages 52 and 53. The solution of the CP model is determined using the `CBSolver` class which invokes the related methods of the NumberJack library.

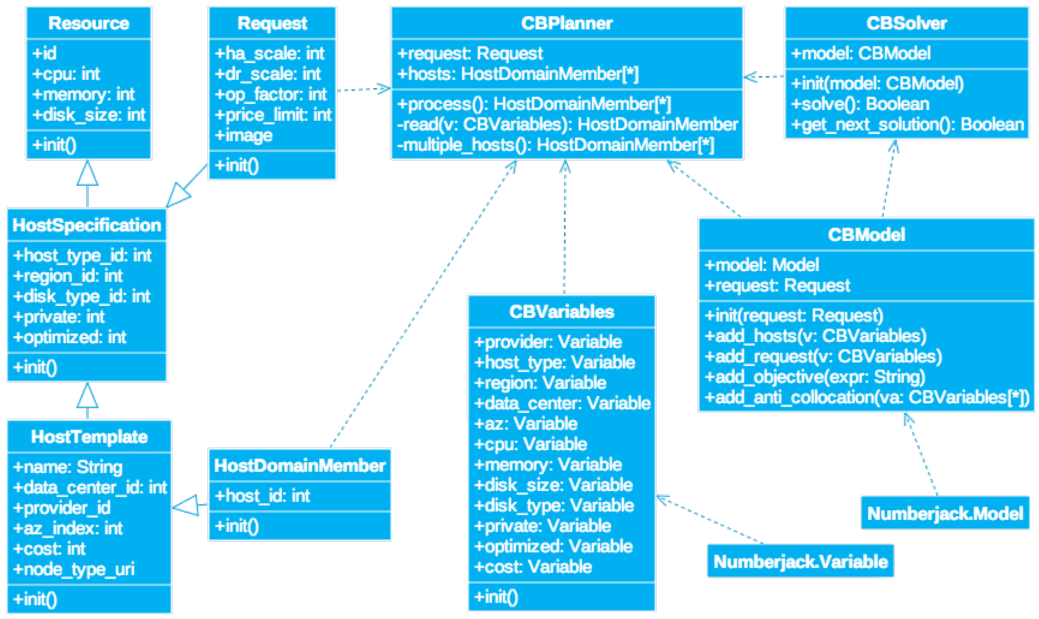


Figure 4.3: Class Diagram of the Deployment Planner

Algorithm 1: Algorithm of the Deployment Planner

```
Result: Return a set $hs$ of $k$ hosts which meet the requirements of the request $r$.
1  Obtain a CP model by creating a CBModel object $m$ and initialise it with $r$;
2  Initialise a two-dimensional array $va[dr\_scale][ha\_scale]$;
3  Initialise the expression $obj=None$ for use by the objective function;
4  foreach $e < dr\_scale$ do
5      foreach $f < ha\_scale$ do
6          Obtain a CBVariables object $v$;
7          Apply the constraint $C_1$ derived from the attributes of the hosts $h \in \mathbb{H}$ to
           the variables in $v$ and add them to the CP model $m$;
8          Apply the constraints $C_{2,..,10}$ derived from the request $r$ to the variables
           in $v$ and add them to the CP model $m$;
9          if $obj$ is None then
10             Assign the cost variable of $v$ to $obj$: $obj = v.cost$;
11         else
12             Concatenate the cost variable of $v$ to $obj$ using addition:
13             $obj = obj + v.cost$;
14         end
15         Assign $v$ to $va[e][f]$;
16     end
17 end
18 Apply the anti-collocation constraints $C_{11}$ and $C_{12}$ for HA and DR to the
   variables in $va$ and add them to the CP model $m$;
19 Add the expression $obj$ to the objective function in order to minimise the cost
   across the allocation of the $k$ hosts;
20 Obtain a CBSolver object $s$;
21 if solver $s$ found a satisfiable solution then
22     foreach $e < dr\_scale$ do
23         foreach $f < ha\_scale$ do
24             Read the current attribute values of the variables in $va[e][f]$ and
               create a HostDomainMember object $h$ with those values;
25             Load the remaining attributes of the host $h$ from the database;
26             Append the host h to the result set $hs$;
27         end
28     end
29 end
```

Algorithm 2: Algorithm to apply constraint C_1

```
Result: Constraint $C_1$ added to the CP model $m$
Input: CBVariables object $v$ and CP model $m$
1  Obtain a HostDomainDao data access object $dao$;
2  Obtain from $dao$ the list $result$ of HostDomainMember objects which meet the
   requirements of the request $r$;
3  Initialise the expression $expr=False$;
4  foreach $res$ in $result$ do
5      $expr = expr$ or (
6      $v.provider == res.provider\_id$ and
7      $v.host\_type == res.host\_type\_id$ and
8      $v.region == res.region\_id$ and
9      $v.data\_center == res.data\_center\_id$ and
10     $v.az == res.az\_index$ and
11     $v.cpu == res.cpu$ and
12     $v.memory == res.memory$ and
13     $v.disk\_size == res.disk\_size$ and
14     $v.disk\_type == res.disk\_type\_id$ and
15     $v.private == res.private$ and
16     $v.optimized == res.optimized$ and
17     $v.cost == res.cost$ )
18 end
19 Add the expression $expr$ to the CP model $m$;
```

Some of the request attributes such as the region allow to specify the value „any“ as the host selection criteria, i.e. the container may be placed in any region. If „any“ is selected by the user, the value of zero is assigned to the related request attribute (`region_id=0`). In this case, no constraint is added for the specific request attribute to the CP model and all values of the attribute domain are considered when solving the placement problem. In the algorithm 3, the request attributes are therefore tested if they are larger than zero.

Algorithm 3: Algorithm to apply constraints $C_{2,..,10}$

```
Result: Constraints $C_{2,..,10}$ added to the CP model $m$
Input: CBVariables object $v$, Request object $r$ and CP model $m$
1  if $r.host\_type\_id > 0$ then
2      Add ($v.host\_type == r.host\_type\_id$) to $m$;
3  end
4  if $r.region\_id > 0$ then
5      Add ($v.region == r.region\_id$) to $m$;
6  end
7  Add ($v.cpu \geq r.cpu \cdot r.op\_factor$) to $m$;
8  Add ($v.memory \geq r.memory \cdot r.op\_factor$) to $m$;
9  Add ($v.disk\_size \geq r.disk\_size \cdot r.op\_factor$) to $m$;
10 if $r.disk\_type\_id > 0$ then
11     Add ($v.disk\_type == r.disk\_type\_id$) to $m$;
12 end
13 if $r.price\_limit > 0$ then
14     Add ($v.cost \leq r.price\_limit$) to $m$;
15 end
16 if $r.private > 0$ then
17     Add ($v.private == r.private$) to $m$;
18 end
19 if $r.optimized > 0$ then
20     Add ($v.optimized == r.optimized$) to $m$;
21 end
```

Although the Docker Cloud API [2] allows to retrieve information about availability zones, it does not support yet to specify an available zone on the request to build a node cluster. The algorithm 4 checks therefore a related property to prevent the addition of related constraints into the CP model.

Algorithm 4: Algorithm to apply constraints C_{11} and C_{12}

```
Result: Constraints $C_{11}$ and $C_{12}$ added to the CP model $m$
Input: CBVariables array $va$ and Request object $r$
foreach $e < dr\_scale$ do
    foreach $f < ha\_scale$ do
        foreach $g < ha\_scale$ do
            if $f \neq g$ then
                $expr = va[e][f].data\_center \neq va[e][g].data\_center$ and
                $va[e][f].region == va[e][g].region$;
                if availability zones are supported then
                    expr = expr or ($va[e][f].az \neq va[e][g].az$ and
                    $va[e][f].data\_center == va[e][g].data\_center$);
                end
                Add the expression $expr$ to the CP model $m$;
            end
        end
    end
end
foreach $d < dr\_scale$ do
    foreach $e < dr\_scale$ do
        if $d \neq e$ then
            foreach $f < ha\_scale$ do
                foreach $g < ha\_scale$ do
                    $va[d][f].region \neq va[e][g].region$;
                end
                Add the expression $expr$ to the CP model $m$;
            end
        end
    end
end
```

4.3 Deployment Engine

The deployment engine class `CBEngine` is initialised with the result set of hosts returned by the deployment planner, and uses the `CBDockerCloud` class to deploy the requested containers on those hosts. The deployment process of the Docker Cloud API [2] requires to first create a node cluster with a given number of hosts. The CSB framework currently assumes that a node cluster is always created with a single host. Once a node cluster and a host are available, a service can be created and the container be deployed. Here a similar constraint applies as for node clusters, the Docker Cloud allows to create multiple containers under one service, while the CSB framework at this stage of the development only allows for one container per service. The class `CBDockerCloud` uses the classes of the Python SDK of the Docker Cloud API [60] in order to interact with the Docker Cloud. The use of the Docker Cloud API is illustrated in the listing B.1 on page 72. In case, the CSB framework selected to deploy a container on an already existing host, only the APIs for deploying a new service and container are invoked. Once the deployment is complete, the related `*Dao` classes are used to update the database.

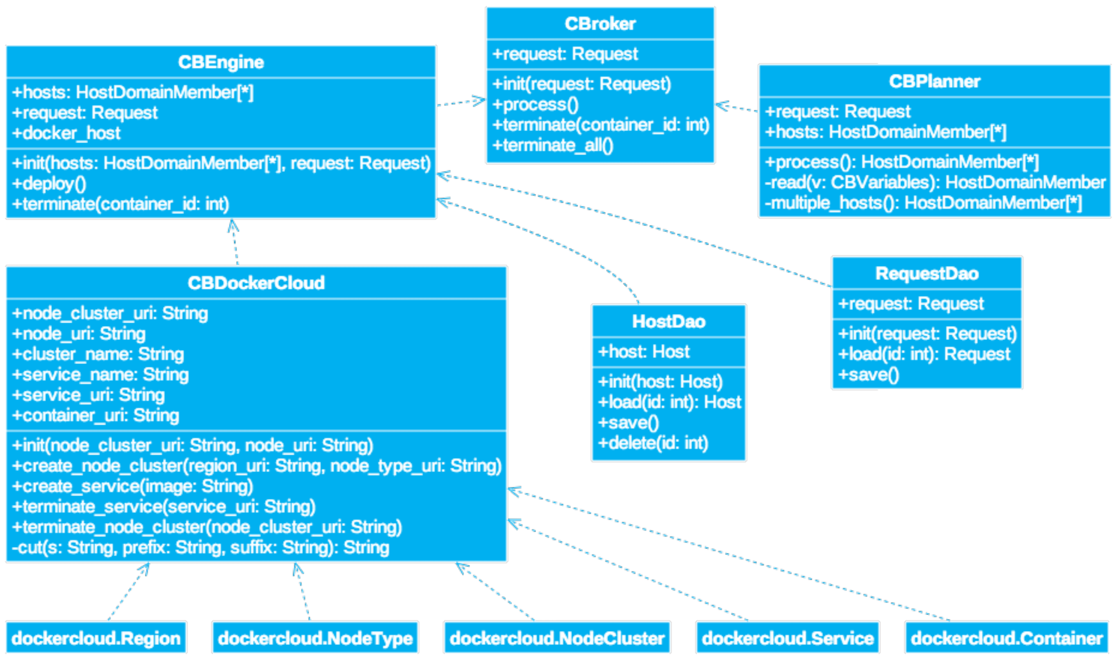

CBroker
+request: Request
+init(request: Request)
+process()
+terminate(container_id: int)
+terminate_all()
CBEngine
+hosts: HostDomainMember[*]
+request: Request
+docker_host
+init(hosts: HostDomainMember[*], request: Request)
+deploy()
+terminate(container_id: int)
CBPlanner
+request: Request
+hosts: HostDomainMember[*]
+process(): HostDomainMember[*]
-read(v: CBVariables): HostDomainMember
-multiple_hosts(): HostDomainMember[*]
CBDockerCloud
+node_cluster_uri: String
+node_uri: String
+cluster_name: String
+service_name: String
+service_uri: String
+container_uri: String
+init(node_cluster_uri: String, node_uri: String)
+create_node_cluster(region_uri: String, node_type_uri: String)
+create_service(image: String)
+terminate_service(service_uri: String)
+terminate_node_cluster(node_cluster_uri: String)
-cut(s: String, prefix: String, suffix: String): String
HostDao
+host: Host
+init(host: Host)
+load(id: int): Host
+save()
+delete(id: int)
RequestDao
+request: Request
+init(request: Request)
+load(id: int): Request
+save()
dockercloud.Region
dockercloud.NodeType
dockercloud.NodeCluster
dockercloud.Service
dockercloud.Container

Figure 4.4: Class Diagram of the Deployment Engine

4.4 Use of the CSB Framework

In order to use the CSB Framework the following steps have to be performed:

1. Download the source code of the CSB framework from the following repository: `https://github.com/sonnenfeldt/peyresourde/tree/master/cbroker`.
2. Installation of Python:
   (a) Install Python 2.7.x
   (b) Install the SQLAlchemy [61], MySQL-python [64], NumberJack [59] and python-dockercloud [60] libraries:
   ```
   pip install sqlalchemy
   pip install mysql-python
   pip install numberjack
   pip install python-dockercloud
   ```
3. Installation and configuration of MySQL:
   (a) Install MySQL [62] or deploy a MySQL container [63]
   (b) Install the MySQL Workbench [65] and connect to the MySQL installation
   (c) Create a schema `cbroker`
   (d) Use the Data Import wizard to import the self-contained file `cbrocker.sql` from package `cbroker.db.sql` which contains the database structure and pre-populated tables.
   (e) Configure the file `cbrokerdb.properties` in package `cbroker.cb`:
   ```
   [MySQL]
   database.user=<user name>
   database.password=<password>
   database.hostname=<hostname>
   database.port=<port>
   ```
4. Set up of the Docker Cloud API access:
   (a) Please note that with the use of Docker Cloud and the associated cloud providers costs occur which the user will have to pay on its own expense.
   (b) Login to Docker Cloud and configure the cloud provider access, please note that this step requires to obtain and configure own accounts for AWS [52], DigitalOcean [66], IBM SoftLayer [53], Microsoft Azure [67] and Packet [68].

   (c) Create an Docker Cloud API key and use it in the next step.
   (d) Configure the file `docker.properties` in package `cbroker.cb`:

```
[Credentials]
dockercloud.user=<user name>
dockercloud.apikey=<api key>
```

5. In order to deploy a container using the CSB framework, the following code fragment has to be executed:

```
p = CBPlanner(request)
hosts = p.process()
e = CBEngine(hosts,request)
e.deploy()
```

   The class `CBroker` defines the basic methods for deploying containers and terminating them, and can be used as entry point for working with the framework.

Chapter 5

Verification and Analysis

5.1 Test Scenarios

In order to verify the functionality of the CSB framework, the following test scenarios are executed:

TC1. Deploy a Wordpress container [69] in one of the most common configurations of the host templates (1 CPU, 2 GB memory) and verify that the most cost-effective host for this configuration is chosen for deployment.

TC2. Select the region with the most data centres (”California”) as input, deploy the Wordpress container and verify that the container is deployed on the most cost-effective host within the specified region.

TC3. Select region ”California” and *ha_scale* = 5 as input, deploy the Wordpress image and verify that containers are deployed on hosts in five different data centres of the region.

TC4. Select *dr_scale* = 3 and *ha_scale* = 2 as input, deploy the Wordpress image and verify that containers are deployed on hosts in three different regions and in two different data centres of each region.

TC5. Run the test case TC1 with an over-provisioning factor $op\_factor = 10$, verify the host, rerun the test case again, and verify the second container is deployed on the previously provisioned host.

TC6. Terminate all containers and hosts created during the test execution and verify that all resources are released.

5.2 Test Environment

The verification of the CSB framework is performed by using a Docker Cloud account which is configured with the five available IaaS providers AWS [52], DigitalOcean [66], Microsoft Azure [67], IBM SoftLayer [53] and Packet [68]. The database of the CSB framework is populated with test data generated using the node types supported by Docker Cloud, and the resource and price information of the IaaS providers. In total, the database holds 3587 host templates of different server configurations in 22 regions and 52 data centres. The distribution of the test data is shown in figure 5.1, and the locations of the data centres are given in figure 5.2. The prices are examples which correlate to the actual prices of the CSPs in the Ireland region, and in some cases to the prices in the region where a CSP is primarily present. The CSB framework is run on a Mac Mini [70] with a MySQL instance created using Docker Cloud and hosted on an AWS VM.

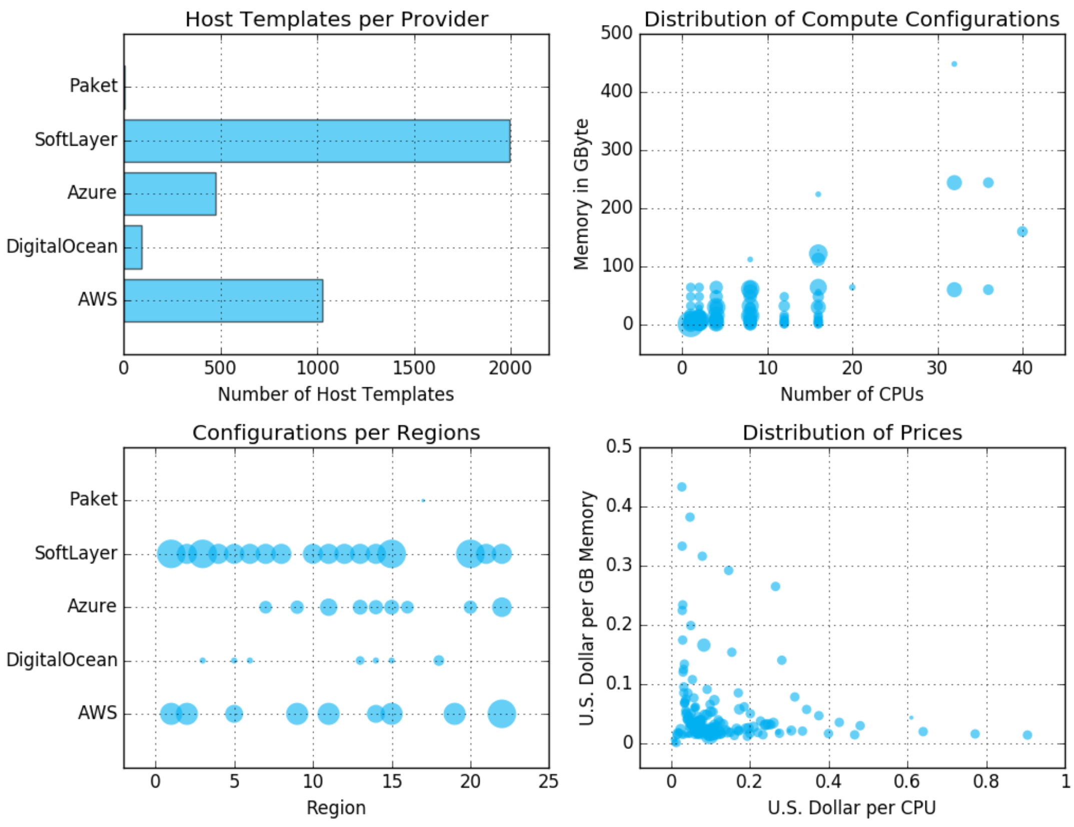


Figure 5.1: Test data of the CSB framework

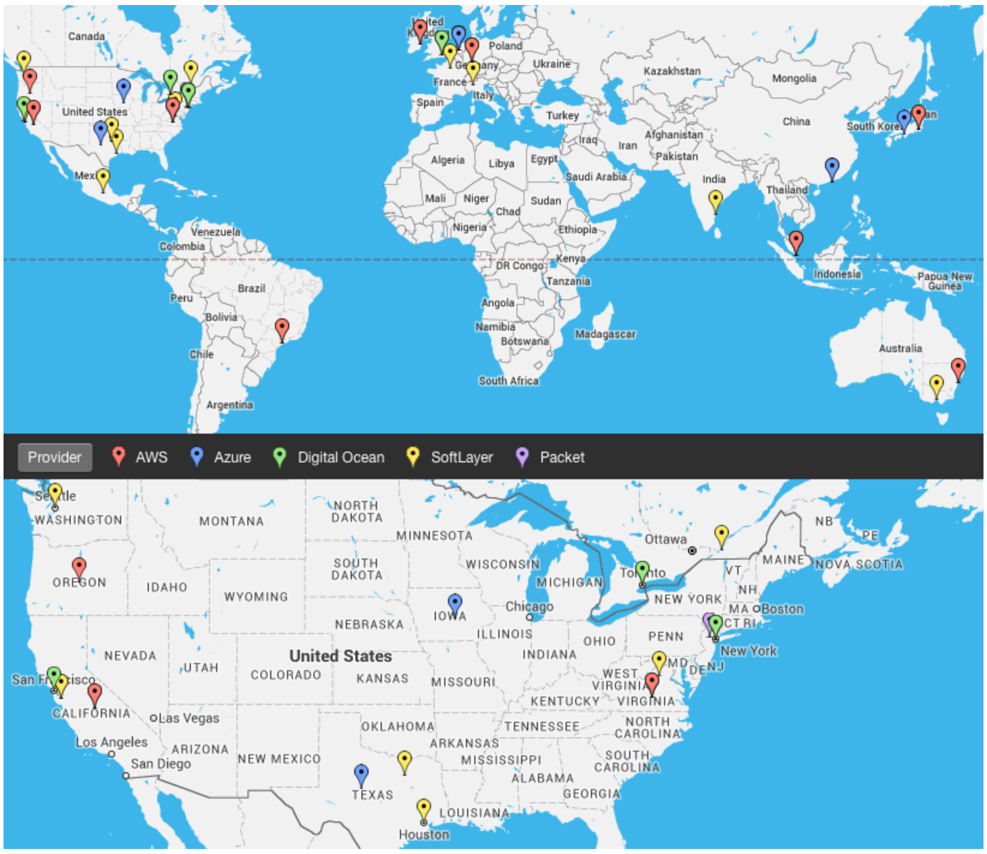


Figure 5.2: Map of the IaaS provider data centres (created with [71])

5.3 Evaluation of the Test Results

The request attributes shown in table 5.1 are used for the execution of the test scenarios described in section 5.1. The test results are given in table 5.2. The CSB frameworks implements CP solutions using the NumberJack platform and the CP solver "Mistral" [72]. The following observations were made:

1. The test results show that the CP solver used by the CSB framework is able to find solutions for the CP models derived for the given test scenarios. The CP solver returns solutions in a reasonable time when only a single container has to be placed.
2. The runtime of the CP solver increases significantly when multiple containers have to placed across different locations for HA and DR, as the number of variables and constraints increases, and the objective function becomes more complex. In order to validate if better performance results can be achieved with another CP solver supported by NumberJack, the "MiniSat" solver [73]

was used for solving the CP model of test case TC4. The test execution had to be cancelled due to extensive CPU utilisation and a reboot of the hosting system was required. All further tests were executed using the "Mistral" solver afterwards.

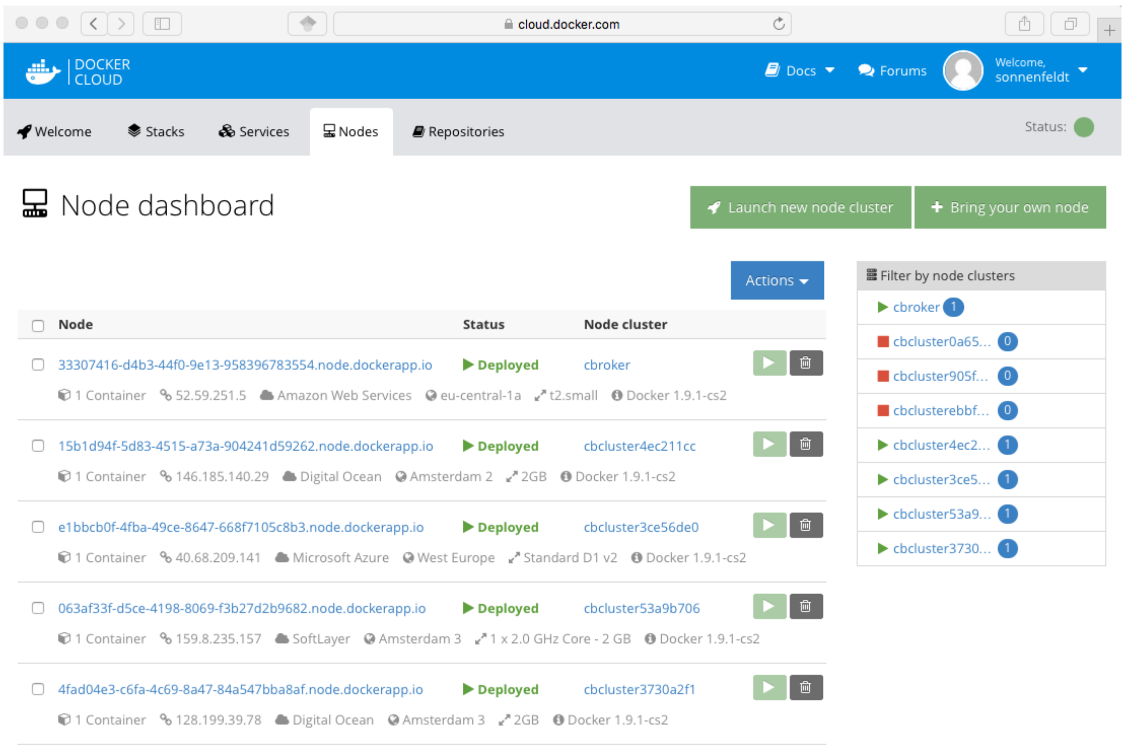
3. By applying a lower price limit to the request of TC4, it was possible to obtain a solution, although it still took a significant amount of time. With the price limit applied, the initial number of host templates is already reduced by the CSB framework before constraints are added to the CP model.
4. The solution of TC4 shows that regions on different continents are selected by the CP Solver, Europe and North America. This solution may not be suitable for business use in all cases, e.g., when legal restrictions apply. Additional locality constraints or an objective function for minimising the distance between regions may be added to the CP model in future.
5. The deployment of the container in the Microsoft Azure site failed for TC3 due to an issue of the Docker Cloud interacting with the Azure API (Bad Request). It was possible to reproduce the issue from the Docker Cloud portal and to confirm that the issue is not related to the CSB framework. The test case TC3 was therefore repeated with modified attributes (TC3.1), and the results of the deployment are shown in figure 5.3.
6. The CSB framework has to be extended with additional features to pass mandatory parameters to the Docker containers and to publish end points.

Parameter	**TC1**	**TC2**	**TC3**	**TC3.1**	**TC4**	**TC5.1/5.2**
host_type	any	any	any	any	any	any
region	any	California	California	Netherlands	any	any
cpu	1	1	1	1	1	1
memory	2	2	2	2	2	2
disk_size	5	5	5	5	5	5
disk_type	any	any	any	any	any	any
ha_scale	1	1	5	4	2	1
dr_scale	1	1	1	1	3	1
op_factor	1	1	1	1	1	10
price_limit	0	0	0	0.10	0.10	0
private	any	any	any	any	any	any
optimized	any	any	any	any	any	any

Table 5.1: Request attributes of the test execution

TC	Provider	Region	Data Center	CPU	Memory (GB)	Disk Size (GB)	Price ($)	Host Domain (# templates)	Time (sec)
1	DigitalOcean	Netherlands	Amsterdam 2	2	2	40	0.030	3232	9.04
2	DigitalOcean	California	San Francisco 1	2	2	40	0.030	330	0.22
3	AWS	California	us-west-1	1	2	5	0.058	330	808.77
	DigitalOcean		San Francisco 1	2	2	40	0.030		
	MS Azure		West US	1	3	50	0.069		
	SoftLayer		San Jose 1	1	2	100	0.076		
	SoftLayer		San Jose 3	1	2	100	0.076		
3.1	DigitalOcean	Netherlands	Amsterdam 2	2	2	40	0.030	9	0.02
	DigitalOcean		Amsterdam 3	2	2	40	0.030		
	MS Azure		West Europe	1	2	50	0.069		
	SoftLayer		Amsterdam 3	1	2	100	0.076		
4	AWS	Germany	eu-central-1	1	2	5	0.058	113	661.12
	DigitalOcean	Germany	Frankfurt 1	2	2	40	0.030		
	DigitalOcean	Netherlands	Amsterdam 2	2	2	40	0.030		
	DigitalOcean	Netherlands	Amsterdam 3	2	2	40	0.030		
	DigitalOcean	New York	New York 1	2	2	40	0.030		
	DigitalOcean	New York	New York 2	2	2	40	0.030		
5.1	MS Azure	Texas	South Central US	16	112	224	0.1542	470	0.47
5.2	Same host as in TC5.1						0.077	471	0.36
6	All node clusters and services successfully terminated								

Table 5.2: Hosts selected and provisioned during test case execution



Node dashboard

Node	Status	Node cluster
33307416-d4b3-44f0-9e13-958396783554.node.dockerapp.io 1 Container 52.59.251.5 Amazon Web Services eu-central-1a t2.small Docker 1.9.1-cs2	Deployed	cbroker
15b1d94f-5d83-4515-a73a-904241d59262.node.dockerapp.io 1 Container 146.185.140.29 Digital Ocean Amsterdam 2 2GB Docker 1.9.1-cs2	Deployed	cbcluster4ec211cc
e1bbcb0f-4fba-49ce-8647-668f7105c8b3.node.dockerapp.io 1 Container 40.68.209.141 Microsoft Azure West Europe Standard D1 v2 Docker 1.9.1-cs2	Deployed	cbcluster3ce56de0
063af33f-d5ce-4198-8069-f3b27d2b9682.node.dockerapp.io 1 Container 159.8.235.157 SoftLayer Amsterdam 3 1 x 2.0 GHz Core - 2 GB Docker 1.9.1-cs2	Deployed	cbcluster53a9b706
4fad04e3-c6fa-4c69-8a47-84a547bba8af.node.dockerapp.io 1 Container 128.199.39.78 Digital Ocean Amsterdam 3 2GB Docker 1.9.1-cs2	Deployed	cbcluster3730a2f1

Filter by node clusters	
cbroker	1
cbcluster0a65…	0
cbcluster905f…	0
cbclusterebbf…	0
cbcluster4ec2…	1
cbcluster3ce5…	1
cbcluster53a9…	1
cbcluster3730…	1

Figure 5.3: Deployment of the containers of TC3.1 using Docker Cloud

Chapter 6

Summary and Conclusions

A first CSB framework using service arbitration for Docker containers is presented in this research report. The arbitration engine of the CSB framework is build on the concepts of constraint programming (CP), and a CP model for the cost-effective placement of containers on hosts of multiple CSPs is proposed. The CP model is designed to handle placement requests with QoS requirements related to HA and DR. The CSB framework is implemented with a deployment planner component build on the CP platform NumberJack [59] and a deployment engine that uses the Docker Cloud [55] as abstraction layer for managing all aspects of the container deployment and to interact with the IaaS providers for resource provisioning.

An important aspects of the proposed CP model is that the emphasis is not on rating the CSPs but the particular hosts for their capability to run a specific container, so that the CSP becomes only as an attribute of the host. The proposed CP model is validated based on test data with host templates from five IaaS providers, 22 regions and 52 data centres. In this experimental research it is shown that the proposed CP model is capable to find the optimal placement for containers also in complex environments, and that HA and DR topologies of applications can be realised. It is further shown that the CP solver "Mistral" [72] used by NumberJack runs stable for large CP models. The duration of the process for finding the optimal solution increases significantly when multiple containers have to placed across different locations for HA and DR. Hence, the practical use of the proposed CP model in a production environment is not possible. Further research is required to reduce the complexity inherited from the input attributes before the actual CP model is constructed. Other CP solvers may be evaluated and the integration of rule-based algorithms such as Rete [74] into the CSB framework can be investigated. The objective of the integration with a rule-based approach will be to limit the number of CP variables and constraints to only the ones which are valid for a given request, so that the CP solver can find a solution to the objective function within a few seconds. The advantage of the combination of the two approaches will be that the rule-based algorithm can provide a reduction of the solution space, while the CP solver is still used to find the best solution for the given objective function.

Aside of the runtime aspect, the CSB framework can be extended in future in various ways. The CSB framework implements today a mono-objective function to minimise the cost. A multi-objective approach may take additional performance attributes into consideration and allow to maximise the service performance while providing the most cost-effective price. Such performance attributes could be gathered from monitoring data of the containers during runtime or be collected from IaaS benchmarking services like CloudHarmony [32]. The data model of the CSB framework is centred around host templates which represent virtual or physical servers. CSPs offer compute, storage and network resources today as independent services with various, flexible options for selection. The data model of the CSB framework may be extended to allow for better consumption and distinction of those services in the placement decisions. In order to include accurate price information, the CSB framework has to be able to query the price information from CSP APIs in future, honour region specific prices, and optionally take the price differences for reservation, on-demand and spot instances into account.

The integration of the CSB framework with the Docker Cloud [55] was demonstrated, and it was shown that the placement decisions of the CP-based planner are successfully implemented using the Docker Cloud API [2]. Further development of the CSB framework is required in order to pass mandatory parameters to the Docker containers, publish end points, and to support node clusters with multiple nodes and services with multiple containers. The integration with other Docker hosting platforms like Rancher [56], Google Container Engine [48], IBM Containers for Bluemix [49] and OpenStack Magnum [75] may be added to the CSB framework in future as well.

With the CSB framework, a first brokering solution using service arbitration for Docker containers is provided. The underlying concepts were successful verified in this research project and allow now for future research and development in this area.

Appendix A

Bibliography

[1] "What is Docker?."
[Online]. Available: https://www.docker.com/what-docker.

[2] "Docker Cloud – API documentation."
[Online]. Available: https://docs.docker.com/apidocs/docker-cloud/.

[3] J. Bond, *The Enterprise Cloud: Best Practices for Transforming Legacy IT.* O'Reilly Media, Inc., 2015.

[4] A. Amato, B. Di Martino, and S. Venticinque, "Cloud Brokering as a Service," in *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2013 Eighth International Conference on*, pp. 9–16, IEEE, 2013.

[5] J. Park, Y. An, and K. Yeom, "Virtual Cloud Bank: An Architectural Approach for Intermediating Cloud Services," in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2015 16th IEEE/ACIS International Conference on*, pp. 1–6, IEEE, 2015.

[6] P. Khanna, S. Jain, and B. Babu, "BroCUR: Distributed cloud broker in a cloud federation: Brokerage peculiarities in a hybrid cloud," in *Computing, Communication and Automation (ICCCA), 2015 International Conference on*, pp. 729–734, IEEE, 2015.

[7] F. Liu, J. Tong, J. Mao, R. Bohn, J. Messina, L. Badger, and D. Leaf, "NIST Cloud Computing Reference Architecture," *NIST special publication*, vol. 500, no. 2011, p. 292, 2011.

[8] E. Mostajeran, B. I. Ismail, M. F. Khalid, and H. Ong, "A Survey on SLA-based Brokering for Inter-cloud Computing," in *Computing Technology and Information Management (ICCTIM), 2015 Second International Conference on*, pp. 25–31, IEEE, 2015.

[9] Gartner, “Cloud Service Brokerage.” Gartner Research, 2016. [Online]. Available: http://www.gartner.com/it-glossary/cloud-services-brokerage-csb.

[10] Gartner, “Gartner Says Cloud Consumers Need Brokerages to Unlock the Potential of Cloud Services.” Gartner Research, July 2009. [Online]. Available: http://www.gartner.com/newsroom/id/1064712.

[11] M. Guzek, A. Gniewek, P. Bouvry, J. Musial, and J. Blazewicz, “Cloud brokering: Current practices and upcoming challenges,” *Cloud Computing, IEEE*, vol. 2, pp. 40–47, 2015.

[12] “What is ITIL Best Practice?.” [Online]. Available: https://www.axelos.com/best-practice-solutions/itil/what-is-itil.

[13] N. Grozev and R. Buyya, “Inter-cloud architectures and application brokering: taxonomy and survey,” *Software: Practice and Experience*, vol. 44, no. 3, pp. 369–390, 2014.

[14] A. N. Toosi, R. N. Calheiros, and R. Buyya, “Interconnected Cloud Computing Environments: Challenges, Taxonomy, and Survey,” *ACM Computing Surveys (CSUR)*, vol. 47, no. 1, p. 7, 2014.

[15] A. Barker, B. Varghese, and L. Thai, “Cloud services brokerage: A survey and research roadmap,” in *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pp. 1029–1032, IEEE, 2015.

[16] L. D. Ngan, S. Tsai Flora, C. C. Keong, and R. Kanagasabai, “Towards A Common Benchmark Framework for Cloud Brokers,” in *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pp. 750 – 754, IEEE, 2012.

[17] A. Tiwari, A. Nagaraju, and M. Mahrishi, “An optimized scheduling algorithm for cloud broker using adaptive cost model,” in *Advance Computing Conference (IACC), 2013 IEEE 3rd International*, pp. 28 – 33, IEEE, 2013.

[18] M. Masdari, S. S. Nabavi, and V. Ahmadi, “An Overview of Virtual Machine Placement Schemes in Cloud Computing,” *Journal of Network and Computer Applications*, 2016.

[19] F. Lopez-Pires and B. Baran, “Virtual Machine Placement Literature Review,” *arXiv preprint arXiv:1506.01509*, 2015.

[20] G. F. Anastasi, E. Carlini, M. Coppola, and P. Dazzi, "QBROKAGE: A Genetic Approach for QoS Cloud Brokering ," in *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pp. 304 – 311, IEEE, 2014.

[21] Y. Kessaci, N. Melab, and E.-G. Talbi, "A pareto-based genetic algorithm for optimized assignment of vm requests on a cloud brokering environment," in *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pp. 2496–2503, IEEE, 2013.

[22] N. K. Srivastava, P. Singh, and S. Singh, "Optimal adaptive CSP scheduling on basis of priority of specific service using cloud broker," in *Industrial and Information Systems (ICIIS), 2014 9th International Conference on*, pp. 1–6, IEEE, 2014.

[23] "Amazon EC2 Pricing." Amazon Web Services.
[Online]. Available: https://aws.amazon.com/ec2/pricing/.

[24] W. Wang, D. Niu, B. Liang, and B. Li, "Dynamic cloud instance acquisition via iaas cloud brokerage," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, no. 6, pp. 1580–1593, 2015.

[25] K. Liu, J. Peng, W. Liu, P. Yao, and Z. Huang, "Dynamic resource reservation via broker federation in cloud service: a fine-grained heuristic-based approach," in *Global Communications Conference (GLOBECOM), 2014 IEEE*, pp. 2338–2343, IEEE, 2014.

[26] S. Nesmachnow, S. Iturriaga, and B. Dorronsoro, "Efficient heuristics for profit optimization of virtual cloud brokers," *Computational Intelligence Magazine, IEEE*, vol. 10, no. 1, pp. 33–43, 2015.

[27] C. Vieira, L. Bittencourt, and E. Madeira, "A scheduling strategy based on redundancy of service requests on iaas providers," in *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, pp. 497–504, IEEE, 2015.

[28] P. Pawluk, B. Simmons, M. Smit, M. Litoiu, and S. Mankovski, "Introducing STRATOS: A cloud broker service," in *2012 IEEE Fifth International Conference on Cloud Computing*, pp. 891–898, IEEE, 2012.

[29] S. Yangui, I.-J. Marshall, J.-P. Laisne, and S. Tata, "Compatibleone: The open source cloud broker," *Journal of Grid Computing*, vol. 12, no. 1, pp. 93–109, 2014.

[30] "Service Measurement Index Framework Version 2.1." Cloud Services Measurement Initiative Consortium (CSMIC), July 2014.

[Online].
Available: http://csmic.org/downloads/SMI_Overview_TwoPointOne.pdf.

[31] J. Siegel and J. Perdue, “Cloud services measures for global use: the Service Measurement Index (SMI),” in *SRII Global Conference (SRII), 2012 Annual*, pp. 411–415, IEEE, 2012.

[32] “Cloud Harmony - Transparency for the Cloud.” Gartner, Inc.
[Online]. Available: https://cloudharmony.com.

[33] M. Souidi, S. Souihi, S. Hoceini, and A. Mellouk, “An adaptive real time mechanism for iaas cloud provider selection based on qoe aspects,” in *Communications (ICC), 2015 IEEE International Conference on*, pp. 6809–6814, IEEE, 2015.

[34] E. C. Freuder and A. K. Mackworth, “Constraint satisfaction: An emerging paradigm,” *Handbook of Constraint Programming*, vol. 2, pp. 13–27, 2006.

[35] K. Apt, *Principles of constraint programming.*
Cambridge University Press, 2003.

[36] A. Bockmayr and J. N. Hooker, “Constraint programming,” *Handbooks in Operations Research and Management Science*, vol. 12, pp. 559–600, 2005.

[37] B. Zilci, M. Slawik, and A. Kupper, “Cloud Service Matchmaking Using Constraint Programming,” in *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2015 IEEE 24th International Conference on*, pp. 63–68, IEEE, 2015.

[38] L. Zhang, Y. Zhuang, and W. Zhu, “Constraint programming based virtual cloud resources allocation model,” *International Journal of Hybrid Information Technology*, vol. 6, no. 6, pp. 333–334, 2013.

[39] Y. Zhang, X. Fu, and K. Ramakrishnan, “Fine-grained multi-resource scheduling in cloud datacenters,” in *Local & Metropolitan Area Networks (LANMAN), 2014 IEEE 20th International Workshop on*, pp. 1–6, IEEE, 2014.

[40] E. Bin, O. Biran, O. Boni, E. Hadad, E. K. Kolodner, Y. Moatti, and D. H. Lorenz, “Guaranteeing high availability goals for virtual machine placement,” in *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pp. 700–709, IEEE, 2011.

[41] H. T. Dang and F. Hermenier, “Higher SLA satisfaction in datacenters with continuous VM placement constraints,” in *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems*, p. 1, ACM, 2013.

[42] "BtrPlace – An Open-Source flexible virtual machine scheduler." [Online]. Available: http://www.btrplace.org.

[43] H. N. Van, F. D. Tran, and J.-M. Menaud, "Performance and power management for cloud infrastructures," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pp. 329–336, IEEE, 2010.

[44] C. Dupont, G. Giuliani, F. Hermenier, T. Schulze, and A. Somov, "An energy aware framework for virtual machine placement in cloud federated data centres," in *Future Energy Systems: Where Energy, Computing and Communication Meet (e-Energy), 2012 Third International Conference on*, pp. 1–10, IEEE, 2012.

[45] "JSR-331 Java Constraint Programming API Specification." Oracle. [Online]. Available: http://download.oracle.com/otn-pub/jcp/constraint_programming-1_0_0-final-spec/constraint_programming-1_0_0-final-spec.pdf.

[46] A. Zhou, S. Wang, B. Cheng, Z. Zheng, F. Yang, R. Chang, M. Lyu, and R. Buyya, "Cloud service reliability enhancement via virtual machine placement optimization," 2016.

[47] "sloppy.io." [Online]. Available: http://sloppy.io.

[48] "Google Container Engine." [Online]. Available: https://cloud.google.com/container-engine/.

[49] "IBM Containers for Bluemix." [Online]. Available: https://console.ng.bluemix.net/docs/containers/container_index.html#container_index.

[50] "Orchard is joining Docker." [Online]. Available: https://www.orchardup.com/blog/orchard-is-joining-docker.

[51] "StackDock." [Online]. Available: http://archive.is/stackdock.com.

[52] "Amazon Web Services." [Online]. Available: https://aws.amazon.com/.

[53] "IBM SoftLayer." [Online]. Available: http://www.softlayer.com.

[54] J. Tordsson, R. S. Montero, R. Moreno-Vozmediano, and I. M. Llorente, "Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers," *Future Generation Computer Systems*, vol. 28, no. 2, pp. 358–367, 2012.

[55] "Docker Cloud – Build Ship & Run, Any App, Anywhere."
[Online]. Available: https://cloud.docker.com.

[56] "Rancher – Introducing the First Platform for Building a Private Container Service."
[Online]. Available: http://rancher.com/rancher/.

[57] "Rancher – Scheduling with Rancher-Compose."
[Online].
Available: http://docs.rancher.com/rancher/rancher-compose/scheduling/.

[58] "MiniZinc."
[Online]. Available: http://www.minizinc.org.

[59] "NumberJack – A Python Contraint Programming platform."
[Online]. Available: http://numberjack.ucc.ie.

[60] "python-dockercloud – Python library for Docker Cloud."
[Online]. Available: https://github.com/docker/python-dockercloud.

[61] "SQLAlchemy - The Database Toolkit for Python."
[Online]. Available: http://www.sqlalchemy.org.

[62] "MySQL – The world's most popular open source database."
[Online]. Available: http://www.mysql.com.

[63] "MySQL Docker file."
[Online]. Available: https://hub.docker.com/_/mysql/.

[64] "MySQL-python."
[Online]. Available: https://pypi.python.org/pypi/MySQL-python.

[65] "MySQL Workbench."
[Online]. Available: https://www.mysql.de/products/workbench/.

[66] "DigitalOcean."
[Online]. Available: https://www.digitalocean.com.

[67] "Mircosoft Azure."
[Online]. Available: https://azure.microsoft.com/.

[68] "packet."
[Online]. Available: https://www.packet.net/.

[69] "appcontainers/wordpress."
[Online]. Available: https://hub.docker.com/r/appcontainers/wordpress/.

[70] "Apple Mac mini Core i5 2.6 (Late 2014) Specs."
[Online].
Available: http://www.everymac.com/systems/apple/mac_mini/specs/mac-mini-core-i5-2.6-late-2014-specs.html.

[71] "batchgeo."
[Online]. Available: http://www.batchgeo.com.

[72] "Mistral."
[Online]. Available: http://homepages.laas.fr/ehebrard/mistral.html.

[73] "The MiniSat page."
[Online]. Available: http://minisat.se.

[74] C. L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artificial intelligence*, vol. 19, no. 1, pp. 17–37, 1982.

[75] "OpenStack Magnum."
[Online]. Available: https://wiki.openstack.org/wiki/Magnum.

[76] M. Maurer, V. C. Emeakaroha, I. Brandic, and J. Altmann, "Cost–benefit analysis of an SLA mapping approach for defining standardized Cloud computing goods," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 39–47, 2012.

Appendix B

Use of the Docker API

```python
import dockercloud, time, ConfigParser
config = ConfigParser.RawConfigParser()
config.read('dockercloud.properties')
dockercloud.user = config.get('Credentials', 'dockercloud.user')
dockercloud.apikey = config.get('Credentials','dockercloud.apikey')

class CBDockerCloud():
    node_cluster_uri=node_uri=service_uri=container_uri=None
    # Constructor
    def __init__(self,cl=None,n=None):
        self.node_cluster_uri = cl; self.node_uri = n
    # Remove the prefix and suffix from string s
    def __cut(self,s,prefix,suffix):
        if s.startswith(prefix): s = s[len(prefix):]
        if s.endswith(suffix): s = s[:-len(suffix)]
        return s
    # Create a node cluster with a single node to run a the service
    def create_node_cluster(self,region_uri,node_type_uri):
        region = dockercloud.Region.fetch(self.__cut(region_uri,
            '/api/infra/v1/region/','/'))
        node_type = dockercloud.NodeType.fetch(self.__cut(node_type_uri,
            '/api/infra/v1/nodetype/','/'))
        nodecluster = dockercloud.NodeCluster.create(name='cbcluster',
        node_type=node_type, region=region, target_num_nodes = 1)
        nodecluster.save(); nodecluster.deploy()
        # Wait for the node cluster to come online
        self.node_cluster_uri = nodecluster.resource_uri;t = 0
        while (nodecluster.state != 'Deployed') & (t < 300):
            time.sleep(10); t+=10
            nodecluster = dockercloud.NodeCluster.fetch(
            self.__cut(self.node_cluster_uri,
                '/api/infra/v1/nodecluster/','/'))
        # Get the resource_uri of the single node
        node = None
        for n in nodecluster.nodes:
            node = dockercloud.Node.fetch(
                self.__cut(n,'/api/infra/v1/node/','/'))
        if node is not None:
            self.node_uri = node.resource_uri; t = 0
            # Wait for the single node to come online
            while (node.state != 'Deployed') & (t < 300):
                time.sleep(10); t+=10
                node = dockercloud.Node.fetch(
                    self.__cut(self.node_uri,'/api/infra/v1/node/','/'))
    # Create a service with a single container on the node cluster
    def create_service(self,image):
        nodecluster = dockercloud.NodeCluster.fetch(
            self.__cut(self.node_cluster_uri,
                '/api/infra/v1/nodecluster/','/'))
        # Assign the tag to deploy the service on a desired node cluster
        tag = 'nodecluster-name=' + nodecluster.name
        service = dockercloud.Service.create(image=image,
            name='service', target_num_containers=1, tags=[tag])
        service.save(); service.start()
        # Wait for the service to come online
        self.service_uri = service.resource_uri; t = 0
        while (service.state != 'Running') & (t < 300):
            time.sleep(10); t+=10
            service = dockercloud.Service.fetch(
                self.__cut(self.service_uri,'/api/app/v1/service/','/'))
        # Get the resource_uri of the single container
        container = None
        for c in service.containers:
            container = dockercloud.Container.fetch(
                self.__cut(c,'/api/app/v1/container/','/'))
        if container is not None:
            # Wait for the single container to come online
            self.container_uri = container.resource_uri; t = 0
            while (container.state != 'Running') & (t < 300):
                time.sleep(10); t+=10
                container = dockercloud.Container.fetch(
                    self.__cut(c,'/api/app/v1/container/','/'))
```

Listing B.1: `CBDockerCloud` class demonstrating the use of the Docker Cloud API

Appendix C

Glossary

anti-collocation describes the constraint that two or multiple computing resources are not hosted in the same location, e.g., on the same server.

API economy describes the positive effect of APIs on an organisation's profitability.

availability zone provides computing resources which are isolated from the resources in other zones in order to prevent outages from spreading across the zones.

Docker Cloud is a Docker container hosting platform. Originally built by Tutum which was acquired by Docker company in 2015.

genetic algorithm is an evolutionary algorithm based on the idea to evolve a population of candidate solutions towards a better solution.

microservices allow complex applications to be composed of independent services with language-agnostic API.

rough sets is a concept to represent uncertainty using boundary regions for sets, as opposed to particular memberships in the sets, often applied in systems for data and knowledge discovery.

second order logic is an extension of the first-order logic and allows to quantify variables over relations.

Service Level Agreement is a contract between the CSP and cloud service consumer that defines the parameters of the service to ensure QoS.

SLA template contains a description of a particular QoS level at which a service can be provided. The SLA template defines the structure of a SLA, the names of the service attributes and the values of the service attributes [76] .

Appendix D

Abbreviations

API	Application Programming Interface	14
AWS	Amazon Web Services	35
CCRA	Cloud Computing Reference Architecture	10
CP	Constraint Programming	2
CSB	Cloud Service Broker	8
CSBg	Cloud Service Brokerage	10
CSP	Cloud Service Provider	7
DR	Disaster Recovery	7
FLA	Federation-Level Agreement	20
HA	High Availability	7
HDD	Hard Disk Drive	41
IaaS	Infrastructure-as-a-Service	2
ITIL	IT Infrastructure Library	13
KPI	Key Performance Indicator	23
NIST	National Institute for Standards and Technology	10
PaaS	Platform-as-a-Service	2
QoE	Quality of Experience	24
QoS	Quality of Service	7
RAD	Resource Acquisition Decision	23
SaaS	Software-as-a-Service	2
SSD	Solid State Disk	41
SMI	Service Measurement Index	23

SLA	Service Level Agreement	7
TDF	Topology Descriptor File	23
URI	Uniform Resource Identifier	40
URL	Uniform Resource Locator	47
U.S.	United States	41
VM	Virtual Machine	15