



Project N°: **FP7-610582**
Project Acronym: **ENVISAGE**
Project Title: **Engineering Virtualized Services**
Instrument: **Collaborative Project**
Scheme: **Information & Communication Technologies**

Deliverable D4.3.3

Assurance of the FRH Case Study

Date of document: T36



Start date of the project: **1st October 2013**

Duration: **36 months**

Organisation name of lead contractor for this deliverable: **FRH**

Final version

STREP Project supported by the 7th Framework Programme of the EC		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary:

Assurance of the FRH Case Study

This document summarises deliverable D4.3.3 of project FP7-610582 (**Envisage**), a Collaborative Project supported by the 7th Framework Programme of the EC. within the Information & Communication Technologies scheme. Full information on this project is available online at <http://www.envisage-project.eu>.

This deliverable showcases the final version of the FRH case study. We show how the ABS language and its extensions (developed in WP1 and WP2) were able to successfully capture the Fredhopper Cloud Services. New material in D4.3.3 focuses on quality assurance for the FRH model through the application of the analysis techniques developed in WP3 as well as simulation, the modeling of FRH SLAs, dynamic deployment, monitoring and visualization.

List of Authors

Stijn de Gouw (FRH)

David Costa (FRH)

Guillermo Román-Díez (UCM)

Contents

1	Introduction	5
1.1	Realized Coverage of Project Objectives	6
2	SaaS Architecture	8
2.1	Service Endpoints	8
2.2	Service Instances	8
2.3	Load Balancing Service	9
2.4	Platform Service	9
2.5	Deployment Service	9
2.6	Infrastructure Service	9
2.7	Monitoring and Alerting Service	10
3	Object Oriented Design	11
3.1	Resources and Virtual machines	11
3.2	Service Configuration	11
3.3	Service Endpoints and Instances	12
3.3.1	Service Architecture	16
3.3.2	Monitoring	18
3.4	ABS Feature usage	21
4	Modeling Deployment Scenarios	27
4.1	Static Deployment	28
4.1.1	Modeling	28
4.1.2	Result	30
4.2	Dynamic Deployment	31
4.2.1	Modeling	32
4.2.2	Result	34
5	Application of Envisage Analyses	36
5.1	Simulator	36
5.2	Monitoring SLAs	36
5.2.1	Service View	37
5.2.2	Grammar	37
5.2.3	Monitor	38
5.3	Deployment	39
5.4	Deadlock Analysis	40
5.5	Resource Analysis	41
5.5.1	Termination	41
5.5.2	Analysis of Relevant Methods	42
5.5.3	Objects	43
5.5.4	Cost Annotations	44

5.5.5	Object Sensitive Analysis	45
5.5.6	SACO improvements	46
5.6	Verification	46
6	Conclusion	47
	Bibliography	48
	Glossary	50

Chapter 1

Introduction

FRH develops the Fredhopper Cloud Services to offer search and targeting facilities on a large product database to e-Commerce companies as services (SaaS) over the cloud computing infrastructure (IaaS). In Task 4.3 we conduct a case study on the Fredhopper Cloud Services, in which we aim to investigate the correspondence between user-level SLAs and lower level performance metrics. In particular, to fulfill user-level SLAs, our service deployment may offer SLA-aware services, evolve service implementation and configure cloud resource usage autonomously.

Fredhopper uses the DevOps methodology. Figure 1.1 shows a simple diagram for a corresponding workflow. DevOps emphasizes collaboration between software developers, operations personnel and quality



Figure 1.1: A DevOps work flow connects development and operations in a continuous iterative process (illustration source: Gene Kim, HP, and PwC, 2013).

assurance teams, recognizing interdependencies between software design, quality of service (QoS) and quality assurance. This is achieved by a recurring flow of rapid releases facilitated by automated configuration and continuous monitoring and testing (or more generally, formal analyses). Below we discuss how the Envisage approach provides a rigorous basis for and improves the existing DevOps work-flow as used in Fredhopper.

- **Automated Configuration.** Envisage supports **automated configuration** by allowing the modeling at the *development* level and the deployment at the *operations* level go hand in hand; in particular, the **Envisage** approach allows configuration and deployment choices already at the *modeling level* for even very abstract models. This allows early exploration and analysis of different alternative

deployments, thereby supporting the operations team to make informative choices, and supporting developers to quickly detect the possible need for further development iterations, in case the results are not satisfactory. The application to the case study of these techniques are discussed in Chapter 4 and Section 5.3.

- **Continuous monitoring** is lifted by Envisage from low-level metrics such as CPU usage, to high-level metrics directly related to (formalized) SLAs and KPIs (Section 5.2).
- **Collaborative development** is supported through the ABS collaboratory and integration into widely used existing IDEs (Eclipse and Emacs plug-ins).
- **Automated tests.** Envisage offers a wide range of powerful tool-supported (semi-) automated analyses, including automated tests: in addition to testing, there is support for functional verification, resource analysis and deadlock freedom. The application of these techniques is described in Chapter 5.

Relation to previous Deliverables Deliverable D4.3.1 provided an initial model in the Abstract Behavioral Specification language (ABS) [2, 6] of the structural and functional aspects of the Fredhopper Cloud Services. But support for deployment of services that allow autonomous management of cloud resource usage - based on formalizations of SLA's - requires a *resource-aware* model of the Fredhopper Cloud Services. The resource-aware version, which included the modeling on initial static deployment scenarios was described in deliverable D4.3.2. This deliverable contains the following new contributions:

- We discuss how the case study has realized coverage of the (and verification of the) overall project objectives (Section 1.1).
- We improved the structure of Chapter 2 and 3.
- We show the ABS language features in the case study that were needed to accurately model the Fredhopper Cloud Services (Section 3.4).
- We show how we managed to model real-world dynamic deployment scenarios, and extend the static initial deployment modeling to take different regions and availability zones into account (Chapter 4).
- We show how we formalized and monitor FRH SLAs (Section 5.2).
- We combine and apply the SAGA (monitoring) and SmartDeploy (deployment) tools to deploy services with autonomous management of cloud resource usage based on the SLA (Section 5.2 and 5.3).
- We discuss our experience with the ABS back-ends (Section 5.1) and successfully analyzed the resource analysis of the Fredhopper Cloud Services (Section 5.5), established its deadlock freedom (Section 5.4), and verified the correctness of monitors (Section 5.6).

1.1 Realized Coverage of Project Objectives

This section shows how the FRH case study has realized coverage of the overall project objectives.

Objective O1: Foundations of Computation with Virtualized Resources

The outcome of this objective is a semantic framework for scalable architectures, infrastructures, and virtualized resources. The framework provides the means to model and to specify resource-related non-functional requirements that arise in the context of virtualized resources. Specifically, using the ABS language, we were able to model the Fredhopper Cloud Services (Chapter 3).

Objective O2: Behavioral Specification Language for Virtualized Resources

The initial model of the Fredhopper Cloud Services (reported in D4.3.1) was successfully extended with *resource-awareness* based on the (tool supported) language extensions to ABS developed in T1.2 and T1.3. We modeled the different kinds of virtual machines used in the Fredhopper Cloud Services as deployment components (allocated through the infrastructure service as offered by the CloudProvider API), we added cost annotations based on measurements from real-world log files, and were able to capture real-world deployment scenarios. This work is reported in Chapter 3 and Chapter 4.

Objective O3: Design-by-Contract Methodology for Service Contracts

Using the technical contributions developed in T2.2 and T2.3, we were able to bridge the gap from the FRH SLAs formulated in natural-language, to formal SLAs and Service Contracts amenable to static and dynamic analysis.

Specifically, SLAs and Service Contracts are modeled as properties of service metric functions. A service metric function maps a trace of events from the Service APIs to a value that indicates a QoS property or the value of a Service Level Objective. The service metric functions themselves have been formalized in a declarative manner with attribute grammars (which are synthesized to ordinary ABS code, amenable to the *Envisage* analyses). The theory underling our approach is described in D2.2.2, D2.3.2 and D4.3.3. How the techniques are applied to capture and monitor FRH SLAs is described in Section 3.3.2.

Objective O4: Model Conformance Demonstrator

We took various steps to ensure a close relation between the in-production Fredhopper Cloud Services and its ABS model. The resources (i.e. kinds of virtual machines, and their properties) and resource consumption of Service instances modeled in ABS were based directly on data from the in-production Fredhopper Cloud Services. Furthermore, at the API level, the ABS model is almost identical to the in-production system. This allowed us to:

1. Faithfully formalize the FRH SLAs, as the SLAs between FRH and its customers concern QoS properties of the Service APIs exposed to the customers (Section 3.3.2 and see D2.2.2 and D2.3.2 for the theory).
2. Plug in data from the in-production system in real-time (such as externally defined monitors), through the ABS HTTP API (see D2.3.2).

Objective O5: Model Analysis Demonstrator

The outcome of this objective is the runtime support for the resource analysis and for the validation with the SLA. This was successfully accomplished through the application of a combination of *Envisage* analyses on the Fredhopper Cloud Services, as reported in Chapter 5.

Chapter 2

SaaS Architecture

Figure 2.1 shows a block diagram of the Fredhopper Cloud Services, where the arrows indicate service consumption and service provision.

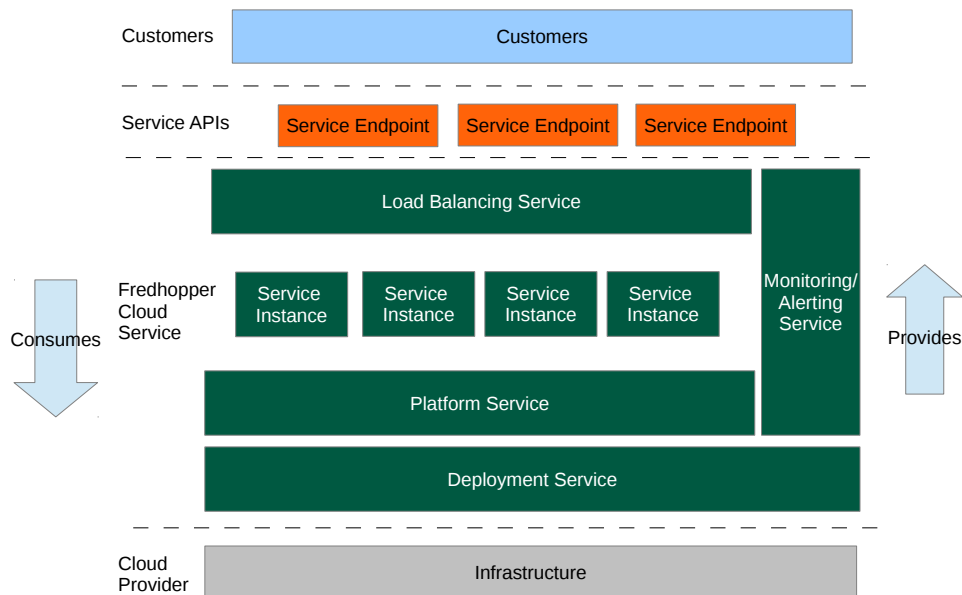


Figure 2.1: Block diagram of the Fredhopper Cloud Services

2.1 Service Endpoints

Fredhopper Cloud Services provides several SaaS offerings on the cloud. These services are exposed via endpoints. In practice these endpoints typically are implemented to be RESTful and accept communications over HTTP. For example, one of the services offered by these endpoints is the Fredhopper query service, which allows users to query over their product catalog via full text search¹ and faceted navigation². Service endpoints are exposed via the Load Balancing Service that distributes requests over multiple *service instances*.

2.2 Service Instances

The advantages of offering software as a service on the cloud over on-premise deployment include the following:

¹en.wikipedia.org/wiki/Full_text_search

²en.wikipedia.org/wiki/Faceted_navigation

- to increase fault tolerance;
- to handle dynamic throughputs;
- to provide seamless service update;
- to increase service testability; and
- to improve the management of infrastructure.

To fully utilize the cloud computing paradigm, software must be designed to be *horizontally* scalable³. Typically, software services are deployed as *service instances*. Each instance offers the same service and is exposed via the Load Balancing Service, which in turn offers a service endpoint (Figure 2.1). Requests through the endpoint are then distributed over the instances. In the event of varying throughput, a different number of instances may be deployed and be exposed through the same endpoint. Moreover, at any time, if an instance stops accepting requests, a new instance may be deployed in place.

2.3 Load Balancing Service

The Load Balancing Service is responsible for distributing requests from service endpoints to their corresponding instances. Currently at FRH, this service is implemented by HAProxy (www.haproxy.org), a TCP/HTTP load balancer.

2.4 Platform Service

The Platform Service provides an *interface* to the *Cloud Engineers* [5, Table 3.1] to deploy and manage service instances and to expose them through service endpoints. The Platform Service takes a service specification, which includes a *resource configuration* for the service [5, Section 3.1], and creates and deploys the specified service. A service specification from a customer determines which type of service is being offered, the number of service instances to be deployed initially and the amount of *virtualized resources* to be consumed by instance.

2.5 Deployment Service

The Deployment Service provides an API to the Platform Service to deploy service instances (using a dedicated Deployment Agent) onto specified virtualized resources provided by the *Infrastructure Service*. The API also offers operations to control the life-cycle of the deployed service instances. The Deployment Service allows the Fredhopper Cloud Services to be independent of the specific infrastructure that underlies the service instances.

2.6 Infrastructure Service

The Infrastructure Service offers an API to the Deployment Service to acquire and release virtualized resources. At the time of writing the Fredhopper Cloud Services utilizes virtualized resources from the Amazon Web Services (aws.amazon.com), where processing and memory resources are exposed through Elastic Compute Cloud instances (<https://aws.amazon.com/ec2/instance-types/>).

³en.wikipedia.org/wiki/Scalability#Horizontal_and_vertical_scaling

2.7 Monitoring and Alerting Service

The Monitoring and Alerting Service provides 24/7 monitoring services on the functional and non-functional properties of the services offered by the Fredhopper Cloud Services, the service instances deployed by the Platform Service, and the healthiness of the acquired virtualized resources.

If a monitored property is not satisfied, *Cloud Engineers* are alerted via emails and SMS messages and *Cloud Engineers* can react accordingly. For example, if the query throughput of a service instance is below a certain threshold, *Cloud Engineers* increase the amount of resources allocated to that service. For broken functional properties, such as a runtime error during service uptime, *Cloud Engineers* notify *Software Engineers* for further analysis.

Chapter 3

Object Oriented Design

In order to apply the **Envisage** framework, to provide feedback to its ongoing development and to evaluate its effectiveness, we develop an ABS model of the Fredhopper Cloud Services that faithfully captures the SaaS architecture described informally in Chapter 2. Support for deployment of services that allow autonomous management of cloud resource usage - based on formalizations of SLAs - requires a *resource-aware* model of the Fredhopper Cloud Services.

In this chapter, we apply the **Envisage** framework to obtain a resource-aware ABS model of the Fredhopper Cloud Services. We show how resources are integrated through virtual machines and cost annotations that specify resource consumption.

The chapter concludes with an overview of the ABS language features that proved to be needed (and in some cases were added specifically due to the continuous feedback from the case study to the technical tasks) to accurately model the resource-aware Fredhopper Cloud Services.

3.1 Resources and Virtual machines

To support a fine-grained management of resource usage, we first model the kinds of virtual machines that are available, together with their associated resource properties and cost. Capturing the detailed resource properties of a virtual machine is a prerequisite to be able to make an informed decision which kind of virtual machine is appropriate to use for each service instance, and when scaling. For instance, if the bandwidth is identified as a bottleneck using the monitoring framework, prefer virtual machines with enhanced networking.

As noted above, the Fredhopper Cloud Services currently utilizes Amazon AWS instances. Figure 3.1 shows a JSON file with several kinds of AWS instances, and their associated resources. “Cores” denotes the number of cores, “Memory” is the size of the memory (in MiB) and “IO” specifies the capacity of the storage device in GB.

The JSON file is processed and converted automatically into a representation of the virtual machines in the ABS. Figure 3.2 defines an ABS data type that enumerates the types of virtual machines.

The capacity of the resources associated to each type of virtual machine is stored in a map of type **Map<ResourceType, Rat>**. Figure 3.3 shows the map with the properties of each kind of virtual machine. The “CostPerInterval” property indicates the pricing of an instance per hour. The prices used are for *on-demand instances*: instances paid for by the hour, rather than “reserved” instances. Using a combination of on-demand instances *and* reserved instances for the same kind of virtual machine is possible by distinguishing two different virtual machine types (i.e., **C3_LARGE_RESERVED** and **C3_LARGE_ONDEMAND**) with the same resources but different cost per interval.

3.2 Service Configuration

Figure 3.4 shows the basic data types involved in a service configuration. A service configuration is modeled as a **Config** value that consists of the service type (**ServiceType**), the number of service instances and its re-

```

"DC_description":
  [ {
    "name" : "c3.large",
    "provide_resources" : {"IO" : 32, "Cores" : 2, "Memory" : 3750},
    "cost" : 105
  },
  {
    "name" : "c3.xlarge",
    "provide_resources" : {"IO" : 80, "Cores" : 4, "Memory" : 7500},
    "cost" : 210
  },
  {
    "name" : "c3.2xlarge",
    "provide_resources" : {"IO" : 160, "Cores" : 8, "Memory" : 15000},
    "cost" : 420
  },
  {
    "name" : "m3.medium",
    "provide_resources" : {"IO" : 4, "Cores" : 1, "Memory" : 3750},
    "cost" : 70
  },
  ...
]

```

Figure 3.1: JSON file with virtual machine types

```

data VMType =
    C3_LARGE
  | C3_XLARGE
  | C3_2XLARGE
  | M3_MEDIUM
  | ...
;

```

Figure 3.2: Enumeration of virtual machine types for Amazon infrastructure provider

source requirement ($\text{List} < \text{Map} < \text{ResourceType}, \text{Rat} > >$). Given a configuration c , the value `instances(c)` is a list of resource descriptions l such that the `length(l)` is the number of service instances to be deployed and the n -th map in the list denotes the minimal amount of resources required for the n -th instance. The resource requirements are used to identify a suitable virtual machine. For instance, if the map for the n -th instance contains the pair `Pair(Memory, 3750)`, then the virtual machine on which the n -th service will be deployed should have at least 3750 MiB of memory.

3.3 Service Endpoints and Instances

Figure 3.5 shows the static structure of an endpoint and its implementation and Figure 3.6 presents the corresponding ABS interface definition. The interface `EndPoint` models a service endpoint. It can be invoked (`invoke(Request)`) with a request of type `Request`. Currently `Request` is a type synonym to `Integer` to denote the *size* of the request. The method returns `Response` value to denote the corresponding response. Currently `Response` is a type synonym to `Boolean` to denote whether the request is successful. It is `True` if the invocation is successful, and `False` otherwise. In the production system of the Fredhopper Cloud

```

/*data Resourcetype = Cores
    | Memory
    | Bandwidth
    | CostPerInterval
    | ...; // Defined in ABS.DC
*/

def Map<Resourcetype, Rat> vmResources(VMType v) =
  case v {
    C3_LARGE => map[Pair(Memory,3750), Pair(Cores,2),
                      Pair(CostPerInterval, 105/1000)];
    C3_XLARGE => map[Pair(Memory,7500), Pair(Cores,4),
                    Pair(CostPerInterval, 210/1000)];
    C3_2XLARGE => map[Pair(Memory,15000), Pair(Cores,8),
                     Pair(CostPerInterval, 420/1000)];
    M3_MEDIUM => map[Pair(Memory,3750), Pair(Cores,1),
                    Pair(CostPerInterval, 67/1000)];
    ...
  };

```

Figure 3.3: Properties of virtual machine types

```

data ServiceType = ...;
data Config = Config(ServiceType serviceType, List< Map<Resourcetype, Rat> > instances);

```

Figure 3.4: Service specification

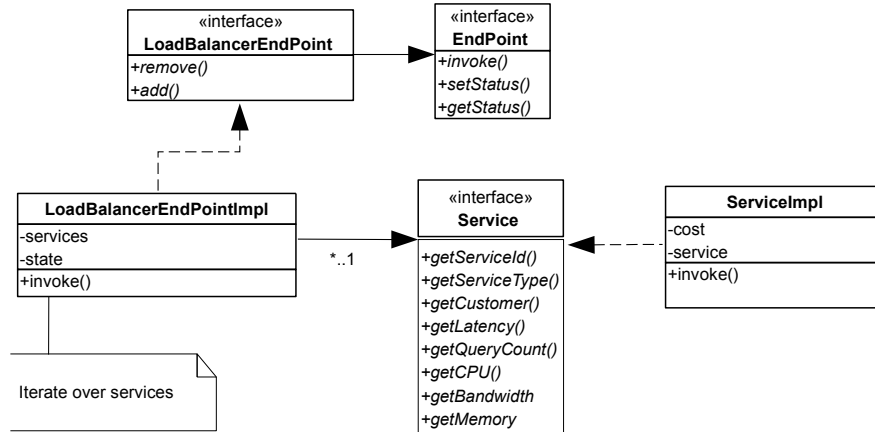


Figure 3.5: UML class diagram of the Fredhopper Cloud Services (1)

Services, the resource utilization, may be dependent on the following two factors:

- the amount of data over which a request queries¹; and
- the size of the corresponding (HTTP) response.

¹In the context of the Query API, this is the size of the underlying product catalog

```

type Id = Int; def Id init() = 1; def Id incr(Id id) = id + 1;
type Request = Int; def Int cost(Request r) = r;
type Response = Bool; def Response success() = True; def Bool isSuccess(Response r) = r;

interface EndPoint {
  Response invoke(Request req);
  Unit setStatus(State status);
  State getStatus();
}

interface LoadBalancerEndPoint extends EndPoint {
  Bool remove(Service service);
  Bool add(Service service);
}

interface Service extends EndPoint {
  Id getServiceId();
  ServiceType getServiceType();
  Customer getCustomer();
  Int getLatency();
  Int getRequestCount();
  Rat getCPU();
  Rat getBandwidth();
  Rat getMemory();
}

```

Figure 3.6: ABS interface of the Fredhopper Cloud Services (1)

We use a type synonym to Integer as an argument to `invoke(Request)` for modeling these factors. The interface `EndPoint` is extended by `Service` and `LoadBalancerEndPoint`.

The interface `Service` models a service instance that performs the actual computation for the request received by its service endpoint.

```

type Customer = String;
class ServiceImpl(Id id, ServiceType st, Customer c, Int cost) {
  Int latency = 0; Int log = 0;
  ..
  Int cost(Request request) {
    return max(1, cost(request)) * cost;
  }

  Response invoke(Request request) {
    Int cost = this.cost(request);
    Int time = currentms();
    [Cost: cost] this.log = this.log + 1;
    time = currentms() - time;
    this.latency = max(this.latency, time);
    return success();
  }
  ..
}

```

Figure 3.7: Implementation of `ServiceImpl.invoke(Int)`

Figure 3.7 shows the implementation of `ServiceImpl.invoke(Int)`. The class implementation takes as arguments at construction its Integer `id`, its service type `st`, the name of the customer to which the service is provided `c`, and the `cost` value, which denotes the amount of CPU resources consumed by the request when the size of the request is 1. Besides these two parameters, the exact latency of a request depends on the amount of resources available when performing the request: latency decreases if the virtual machine that processes the request has more CPU speed resources, and latency increases if `cost` and `Request` increase.

We use the type synonym `Customer` to model the customer's name. The function `currentms()` is a built-in ABS function that returns the current clock cycle, and function `max(a, b)` returns the larger value of `a` and `b`. The method `invoke` uses the cost annotation `[Cost: cost]` to denote the required number of CPU speed units to execute the annotated statement.

```
class LoadBalancerEndPointImpl(List<Service> services) implements LoadBalancerEndPoint {
  List<Service> current = services;
  { assert this.services != Nil; }
  Response invoke(Request request) {
    if (this.current == Nil) {
      this.current = this.services;
    }
    Service ser = head(this.current);
    this.current = tail(this.current);
    return await ser!invoke(request);
  }
  ..
}
```

Figure 3.8: Implementation of `LoadBalancerEndPointImpl.invoke(Request)`

```
class LoadBalanceCPU(List<Service> services) implements LoadBalancerEndPoint {
  { assert this.services != Nil; }
  Response invoke(Request request) {
    List<Service> remaining = services;

    Service best = head(remaining);
    Fut<Rat> fMostCPU = ser!getCPU();
    Rat mostCPU = fMostCPU.get;

    while(remaining != Nil) {
      remaining = tail(remaining);
      Service ser = head(remaining);
      Fut<Rat> fCPU = ser!getCPU();
      Rat cpu = fCPU.get;
      if(cpu < mostCPU) {
        best = ser;
        mostCPU = cpu;
      }
    }

    return await best!invoke(request);
  }
  ..
}
```

Figure 3.9: Implementation of `LoadBalancerEndPointImpl.invoke(Request)`

The interface `LoadBalancerEndPoint` extends interface `EndPoint` with the ability to dynamically as-

sociate service instances to a service endpoint, thereby allowing requests to the endpoint to be distributed. The class `LoadBalancerEndPointImpl` implements `LoadBalancerEndPoint` and its implementation of `invoke(Request)` is shown in Figure 3.9. This method implements a simple round-robin load balancing strategy to distribute requests. The class implementation `LoadBalancerEndPointImpl` is parametric to a non-empty list of unique `Service` references. The class `LoadBalancerOptimizeCPU` implements a resource-aware load balancer to distribute requests. Specifically, a request is directed to the service instance that currently has the most CPU speed resources available.

3.3.1 Service Architecture

Figure 3.10 shows the static structure of the Fredhopper Cloud Services, and Figure 3.11 shows the corresponding interfaces in ABS. We present the modeling of the Monitoring and Alerting Service in Section 3.3.2. The static structure shown in Figure 3.10 models dependencies between various services in the Fredhopper Cloud Services.

ABS models virtualized resources as a `DeploymentComponent`. A `DeploymentComponent` takes a value of the data type `Map<ResourceType, Rat>` as the resource specification. The abbreviation DC is a type synonym for `DeploymentComponent`.

The interface `InfrastructureService` is responsible for providing/managing virtual machines in the form of `DeploymentComponents`. This interface is implemented by the class `InfrastructureServiceImpl` shown in Figure 3.12. The figure shows the class's implementation of `acquire(Id, VMType)`. The method takes as input an `id` of the virtual machine to be acquired and its type, and returns an instance of the specified kind of machine. The method `acquire` either creates a new `DeploymentComponent` of the specified type, or reuses an existing `DeploymentComponent` if the `id` already exists. The capacity of the resources associated to the virtual machine are retrieved through the `Map<ResourceTypes, Rat> vmResources` given in Figure 3.3.

The interface `LoadBalancerService` in Figure 3.11 is responsible for binding requests to service endpoints to their constituent service instances. `DeploymentService` is responsible for allocating virtualized resources to service instances. This is implemented by the class `DeploymentServiceImpl`. Figure 3.13 shows the implementation of method `DeploymentServiceImpl.install(Customer, ServiceType, Id, VMType)`. This method instantiates a service of the specified type on a machine of the given type. The allocation of the virtualized resources to the new service instance is realized by the statement `[DC: vm] Service service = new ServiceImpl(serviceId, st, customer, 2);`, which indicates that the new service instance executes on the virtual machine `vm`.

`PlatformService` provides the interface to *Cloud Engineers* to add and to remove services. The interface also provides operations to the Monitoring and Alerting Service for adding and removing service instances to an endpoint and for adding and removing resources from a service instance. `PlatformService` is implemented by class `PlatformServiceImpl`, whose definition of method `Int createService(Config, Customer)` is shown in Figure 3.14. The method `createService` takes a service configuration and a customer identifier, deploys a corresponding service for that customer and returns the identifier of the service endpoint. Figure 3.16 shows how the services in the Fredhopper Cloud Services interact to deploy a service. Specifically, the method first iteratively creates the specified number of service instances, allocating each with a virtual machine that satisfies the resource requirements. Figure 3.16 shows the following:

1. the `PlatformService` interacts with the `DeploymentService` to install (`install(Customer c, ServiceType st, Id serviceId, VMType v)`) and start service instances (`start(Id)`);
2. the `DeploymentService` interacts with the `InfrastructureService` to allocate (`acquire(Id id, VMType vmType)`) the required resources (`DeploymentComponent`) to the service instances;
3. after all service instances are deployed, the `PlatformService` interacts with the `LoadBalancerService`

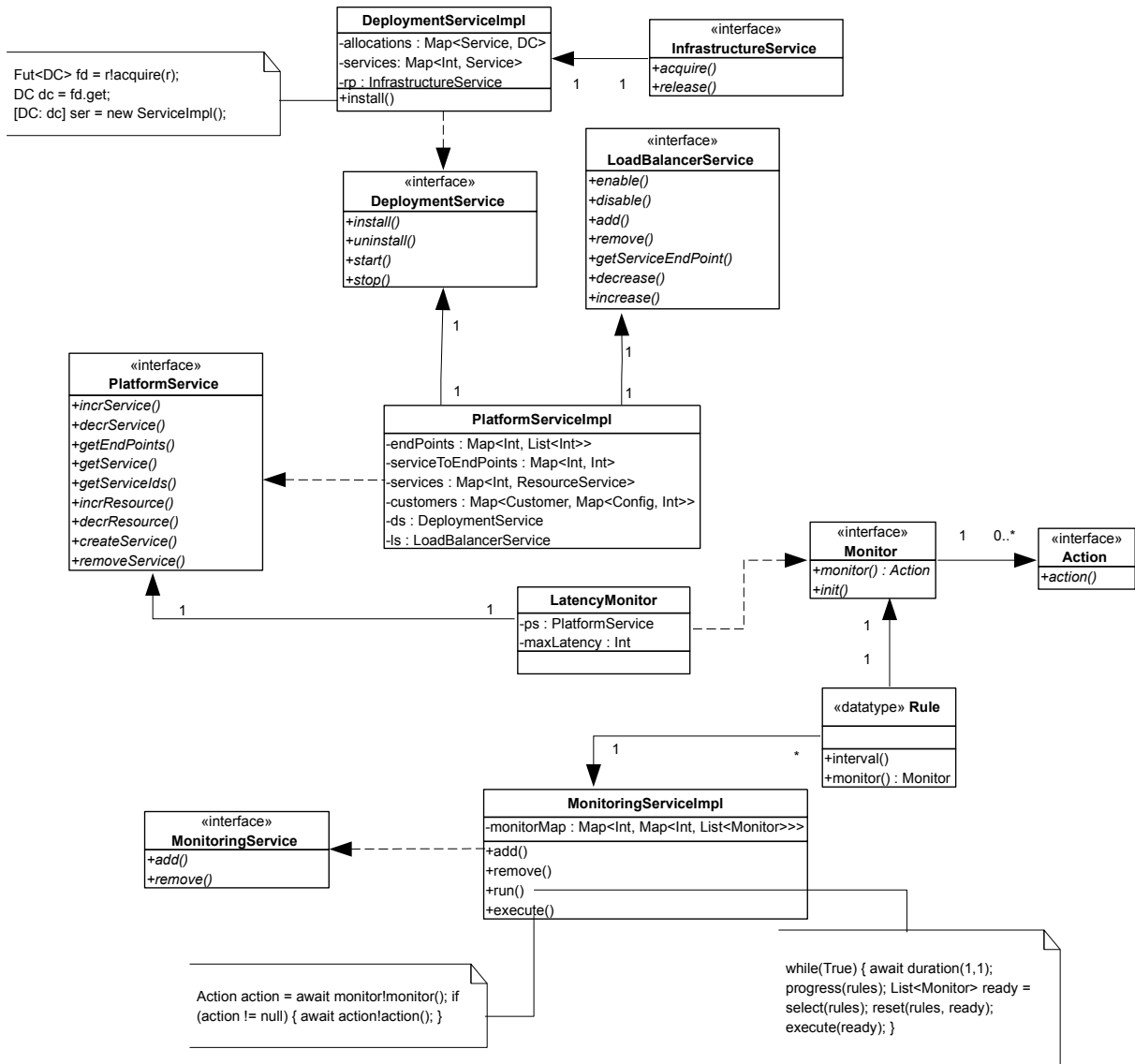


Figure 3.10: UML class diagram of the Fredhopper Cloud Services (2)

to bind (`add(List<Service>, Id)`) the instances to its service endpoint and to enable the endpoint (`enable(Id)`).

Figure 3.10 shows how the model of the Fredhopper Cloud Services respects the above dependencies. Specifically, `PlatformService` depends on `DeploymentService` and `LoadBalancerService` via the implementation `PlatformServiceImpl`, while `DeploymentService` depends on `InfrastructureService` via the implementation `DeploymentServiceImpl`. Dependencies are provided via dependency injection (i.e., passing the object that provides the service to the object that depends on it).

```

interface InfrastructureService {
    DeploymentComponent acquire(Id id, VMType vmType);
    Unit release(DeploymentComponent component);
}

interface DeploymentService {
    Unit install(Customer c, ServiceType st, Id serviceId, VMType v);
    Unit uninstall(Id serviceId);
    Unit start(Id serviceId);
    Unit stop(Id serviceId);
}

interface LoadBalancerService {
    Bool enable(Id endPointId);
    Bool disable(Id endPointId);
    Bool add(List<Service> services, Id endPointId);
    Bool remove(Id endPointId);
    Maybe<EndPoint> getServiceEndPoint(Id endPointId);
    Bool decrease(Id endPointId, List<Service> services);
    Bool increase(Id endPointId, List<Service> services);
}

interface PlatformService {
    Unit incrService(Id endPoint, List< Map<Resourcetype, Rat> > instances);
    Unit decrService(Id endPoint, List<Id> serviceIds);
    List<Id> getEndPoints();
    Maybe<Service> getService(Id serviceId);
    List<Id> getServiceIds(Id endPoint);
    Unit alterResource(Id serviceId, Map<Resourcetype, Rat> r);
    Id createService(Config config, Customer customer);
    Unit removeService(Id endPoint);
}

```

Figure 3.11: ABS interface of the Fredhopper Cloud Services (2)

3.3.2 Monitoring

The static structure diagram of the Fredhopper Cloud Services shown in Figure 3.10 includes the Monitoring and Alerting Service. This service is modeled by the interface **MonitoringService**, which is implemented by the class **MonitoringServiceImpl** in ABS. The class **MonitoringServiceImpl**, shown in Figure 3.15, has a **run** method that iteratively checks which monitors in the list of *scheduled* monitors (**monitorMap**) are ready in every clock cycle (**await duration(1, 1)**).

The list of scheduled monitors are recorded as a two level map, where the first level key records the number of clock cycles between each execution of the lists of **Monitors** in the second level map, and the second level key records the number of remaining clock cycles until the next execution.

Given a **Monitor** *m*, the method invocation **m!monitor()** executes a corrective action to improve the quality of service.

Figures 3.17 – 3.19 depict the interactions between services to scale up the underlying resources of service instances suffering from high latency.

Figure 3.17 shows a sequence diagram of the **MonitoringServiceImpl** invoking a **Monitor** object to check the average latency of requests being served by all service instances (**monitor()**). The diagram shows that the **Monitor** collects latency reading (**Service.getLatency()**) from all service instances (**PlatformService.getServiceIds(Int)**) of all end points (**PlatformService.getEndPoints()**). If the latency of one or more service instances is too high, the **Monitor** executes an action for scaling up the

```

class InfrastructureServiceImpl implements InfrastructureService {
    ...
    Map<Id, DeploymentComponent> inUse = EmptyMap;
    DC acquire(Id id, VMType vmType) {
        DC vm = null;
        Map<ResourceType, Rat> resourceConfig = vmResources(vmType);
        Maybe<DC> md = lookup(inUse, id);
        case md {
            Nothing => {
                //Allocate new instance of type vmType
                vm = new DeploymentComponent(intToString(id), resourceConfig);
                inUse = InsertAssoc(Pair(id, vm), inUse);
            }
            Just(d) => {
                //Use existing instance with the specified id
                vm = d;
            }
        }
        return vm;
    }
}

```

Figure 3.12: Definition of InfrastructureServiceImpl.acquire(Id, VMType)

```

class DeploymentServiceImpl(InfrastructureService rp) implements DeploymentService {
    Map<Service, DC> allocations = EmptyMap;
    Map<Id, Service> services = EmptyMap;

    Service install(Customer customer, ServiceType st, Id serviceId, VMType v) {
        assert lookup(services, serviceId) == Nothing;

        //acquire resource
        DC vm = await rp!acquire(serviceId, v);

        //instantiate service on vm
        [DC: vm] Service service = new ServiceImpl(serviceId, st, customer, 2);

        //update maps with resources (allocations) and service instances (services)
        allocations = InsertAssoc(Pair(service, dc), allocations);
        services = InsertAssoc(Pair(serviceId, service), services);
        return service;
    }
}

```

Figure 3.13: Definition of DeploymentServiceImpl.install

virtualized resources underlying the service instances.

Figure 3.18 shows a sequence diagram of scaling up the virtualized resources (CPU unit per clock cycle) of a service instance. The diagram shows that said instance must be first removed from the load balancer (`LoadBalancerService.decrease(Int, List<Service>)`), and uninstalled (`uninstall(Int)`) from the existing resource via the `DeploymentService`. Note that the said service instance must no longer be serving requests before its removal from the load balancer. The instance is then installed onto a new virtual machine with the specified resource requirements and then added back to the load balancer (`LoadBalancerService.increase(Int, List<Service>)`).

Figure 3.19 shows a sequence diagram of scaling up the virtualized resources of a service instance when

```

class PlatformServiceImpl(DeploymentService ds, LoadBalancerService ls) .. {
  Map<Id, ResourceService> services = EmptyMap;
  Map<Id, Id> serToEndPoint = EmptyMap; Map<Id, List<Id>> endPoints = EmptyMap;
  Map<Customer, Map<Config, Id>> customers = EmptyMap; Id serviceId = init();

  Id createService(Config config, Customer customer) {
    //this customer cannot already have the same service deployed
    ServiceType st = serviceType(config);
    assert lookupCustomerService(customers, customer, st) == Nothing;

    List<Int> instances = instances(config);
    //number of instances must be positive
    assert instances != Nil;

    //endpoint id
    Int endPoint = serviceId + 1;

    //create service instances
    List<Service> currentServices = Nil;
    List<Id> ids = Nil;

    while (instances != Nil) {
      Int res = head(instances);
      Service service = this.createServiceInstance(customer, st, res);
      Fut<Id> idf = service!getServiceId();
      Id id = idf.get;
      ids = Cons(id, ids);
      serviceToEndPoints = InsertAssoc(Pair(id, endPoint), serviceToEndPoints);
      currentServices = Cons(service, currentServices);
      instances = tail(instances);
    }

    //associate endpoint with service instances
    endPoints = InsertAssoc(Pair(endPoint, ids), endPoints);

    //update customer record
    customers = put(customers, customer,
      put(lookupDefault(customers, customer, EmptyMap), config, endPoint));

    //add services to load balancer
    await ls!add(currentServices, endPoint);

    //enable service
    await ls!enable(endPoint);

    return endPoint;
  }
}

```

Figure 3.14: Definition of PlatformServiceImpl.createService()

```

class MonitoringServiceImpl implements MonitoringService {
  Map<Int, Map<Int, List<Monitor>>> monitorMap = EmptyMap;
  ..
  Unit run() {
    while (True) {
      await duration(1, 1); // advance the clock
      this.monitorMap = decr(this.monitorMap); //decrement
      List<Monitor> toBeRun = lookupAllSecond(this.monitorMap, 0); //find all to be run
      this.monitorMap = reset(this.monitorMap); //reset
      //execute monitors
      while (toBeRun != Nil) {
        this!execute(head(toBeRun));
        toBeRun = tail(toBeRun);
      }
    }
  }

  Unit execute(Monitor m) {
    await m!monitor();
  }
  ..
}

```

Figure 3.15: Definition of MonitoringServiceImpl.run()

it is the only instance to its service endpoint. In order to keep the service running, before removing the instance from the load balancer, a new service instance must first be installed onto the required resource and added to the load balancer. The old instance may then be removed and uninstalled.

3.4 ABS Feature usage

The matrix below shows the ABS language features that were needed to accurately model the resource-aware Fredhopper Cloud Services. All major categories (i.e., functional layer, deployment modeling, etc) were used.

Feature	Used?
<i>Functional layer</i>	✓
User-defined data-types	✓
Rational Numbers	✓
<i>Imperative Layer</i>	✓
Local objects in COGs	✓
<i>Delta layer</i>	✓
Deltas	✓
More than one product in product line	
Non-trivial feature model	
<i>Deployment modeling</i>	✓
Creating deployment components in main block using “new DeploymentComponent”	
Creating deployment components in main block using the CloudProvider class	✓
Creating deployment components dynamically using “new DeploymentComponent”	
Creating deployment components dynamically using the CloudProvider class	✓
<i>Resource modeling</i>	✓
Using static resources (one or more of Cores, Memory)	✓
Using dynamic resources (one or more of Bandwidth, Speed)	✓
Using cost accounting (PaymentInterval, CostPerInterval; or StartupDuration, ShutdownDuration)	✓
“Cost: ” (CPU) annotations using constants	✓
“Cost: ” (CPU) annotations using non-constant expressions	✓
“DataSize: ” (Bandwidth) annotations using constants	
“DataSize: ” (Bandwidth) annotations using non-constant expressions	
Using load monitoring (e.g., using the result of “dc.load(...)”)	✓

Table 3.1: Usage of ABS language features in the Fredhopper Cloud Services

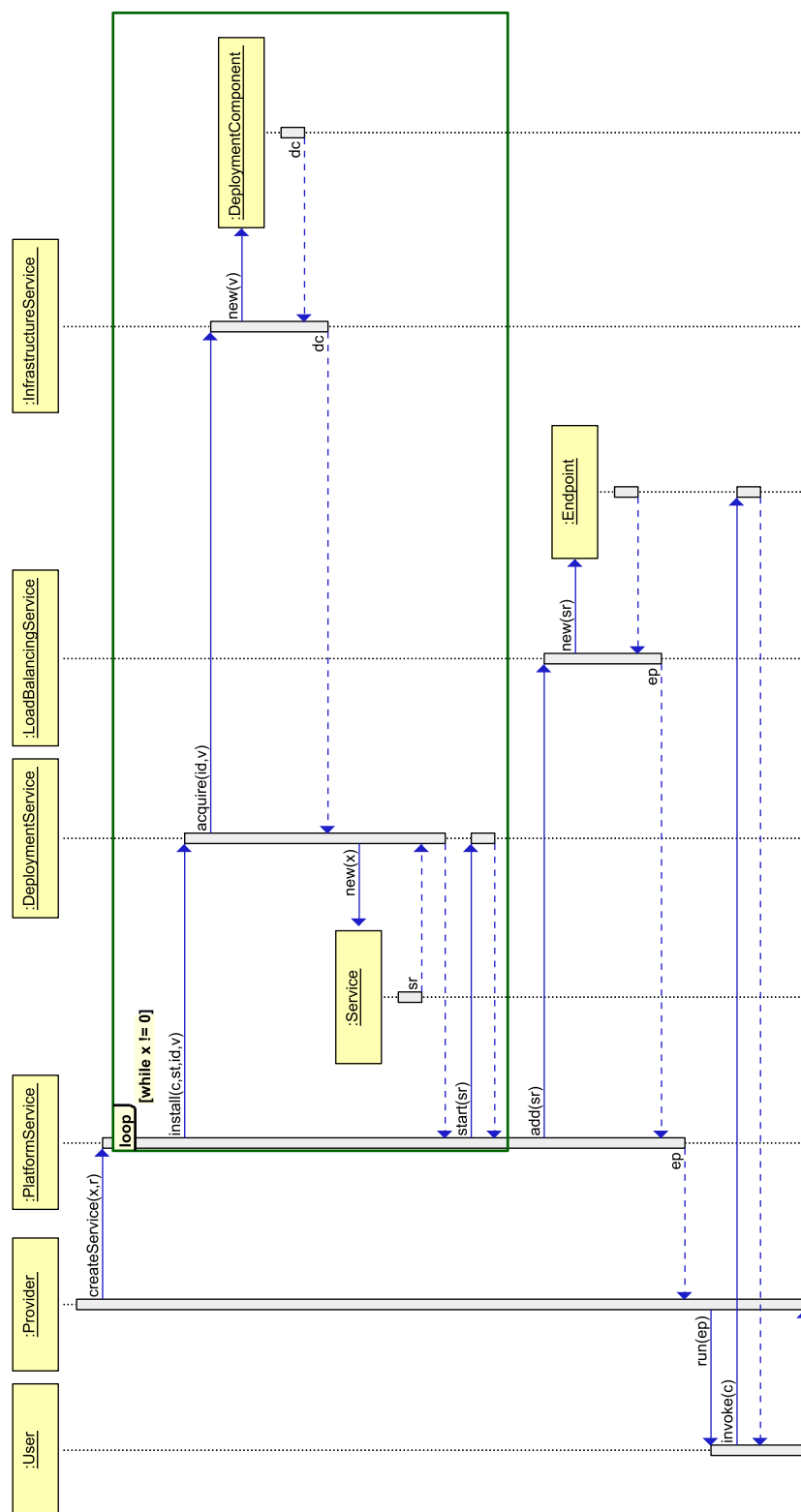


Figure 3.16: UML sequence diagram of creating a service using the Fredhopper Cloud Services

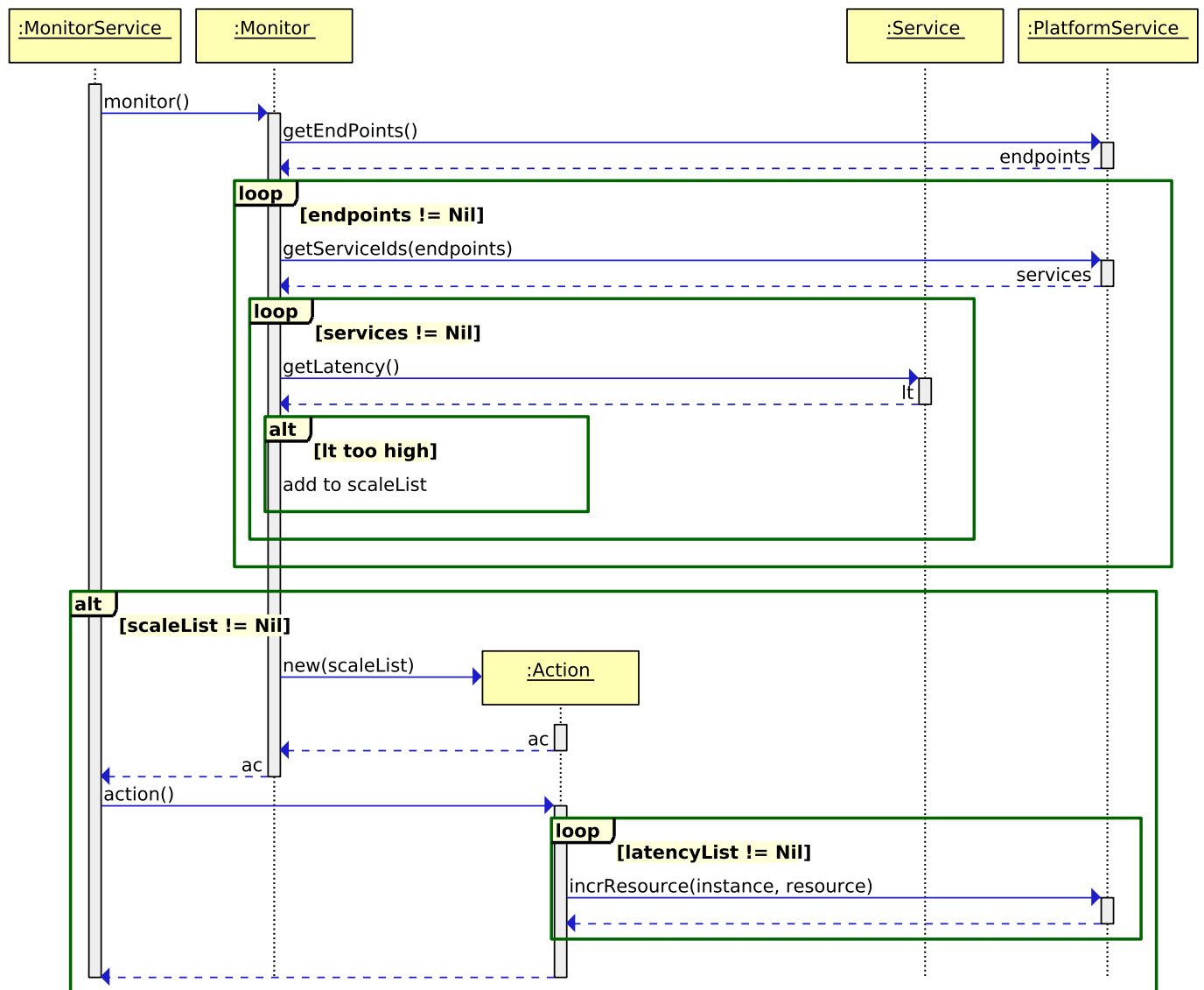


Figure 3.17: UML sequence diagram of monitoring service latency

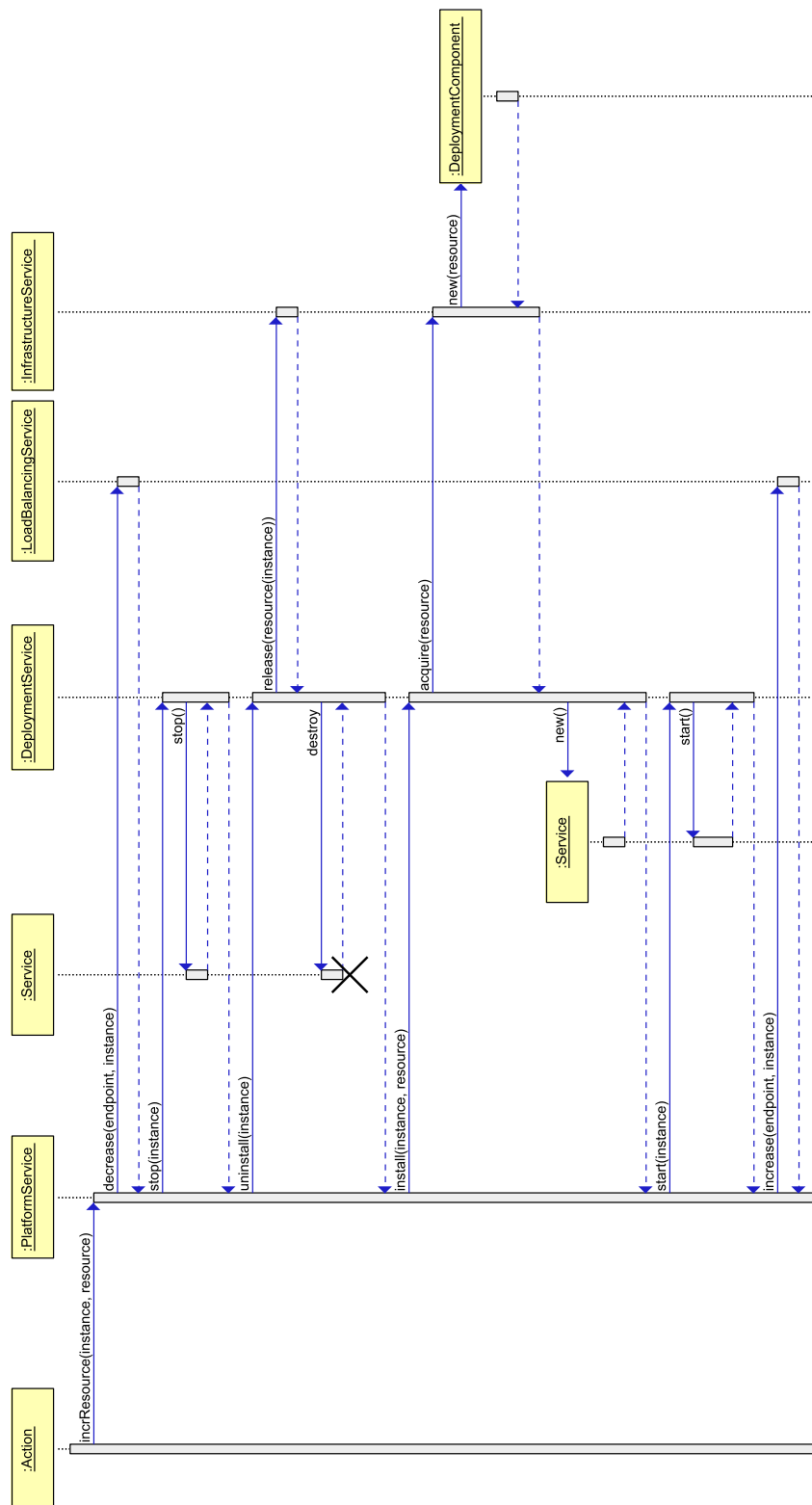


Figure 3.18: UML sequence diagram of scaling a service with more than one service instance



Chapter 4

Modeling Deployment Scenarios

To offer services with a high QoS level, it is crucial to find an optimal deployment configuration: the number and kind of virtual machines used in a deployment must be sufficiently powerful and the cost of the virtual machines must be maintained at an acceptable level. The deployment configuration must also take into account several requirements. For example, some services should be co-located with other services: deploying an instance of the Query Service to a machine requires the presence of the Deployment Service on that same machine. Other service instances should be deployed on different machines, for example to increase fault tolerance (a resource failure will then not affect both service instances) or for security or business reasons, such as allocating a dedicated (per-customer) machine to services that manipulate sensitive private customer data (Figure 4.1).

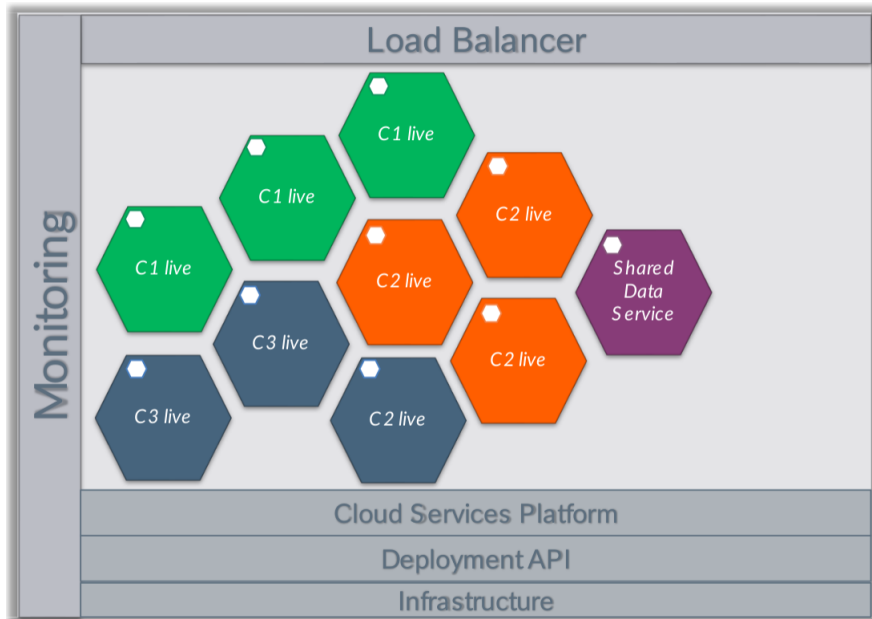


Figure 4.1: High-level view of a deployment

Finding an optimal deployment configuration that satisfies all requirements is a complex task that is currently done manually by an operations team. This requires domain-specific knowledge and is prone to human-error. Furthermore, the operations team takes conservative precautions to ensure customer quality, by overspending on the deployment configuration.

In collaboration with T1.3 (Deployment Modeling) and T2.3 (monitoring), and in publications [3, 4], a tool-supported rigorous formal approach was developed that helps evaluating and automatically synthesizes better *initial* deployment configurations in an early phase (statically, in the design phase). Since the synthe-

sized deployment configuration are known at an early stage, reserved instances of the virtual machines can be used, which are typically cheaper than on-demand instances. In Section 4.1 we show here how we have used these techniques to declaratively model the initial deployment in the Fredhopper Cloud Services, and in Section 4.2 we model dynamic deployment scenarios using the following ingredients.

- Annotations on ABS classes that specify:
 - (a) the amount of resource consumed by instances of the class, and
 - (b) dependencies to other classes.
- A high-level declarative specification that captures requirements that should be satisfied of the desired deployment.
- The number of available virtual machines of each kind. The types of virtual machines are provided in a JSON file along the lines of Figure 3.1. An example JSON file showing the number of available instances of each type is given in Figure 4.2.

```
{
  "m1.large": 3,
  "m1.xlarge": 3,
  "c3.medium": 10,
  "c3.large": 5,
  "c3.2xlarge": 3
}
```

Figure 4.2: JSON file with the number of available instances of each type

Cost annotations specify the amount of resources that the annotated identity consumes, such as memory consumption, bandwidth, or CPU cycles. We have annotated the ABS model of the Fredhopper Cloud Services with cost annotations. The costs used in the annotations were extracted based on real-world log files of the in-production system. The resource consumption varies over time, and scenarios can be used to capture peaks and lows in the corresponding resource consumption. Figure 4.3 is a graph representation of one of these log files showing the number of Query requests per second and the round trip time (the total time of the HTTP request performing a Query Request as reported by the HTTP client handler). Figure 4.4 shows the size of the Query response over time (for the same customer, on the same date).

4.1 Static Deployment

In Section 4.1.1 we show the modeling of the initial deployment configuration and Section 4.1.2 discusses the outcome: the deployment configuration automatically synthesized.

4.1.1 Modeling

Figures 4.5 and 4.6 show annotations for the Query and Deployment services. The ABS annotation `Param("c", User)` indicates that the automated deployer leaves the parameter unspecified, and the user is expected to manually enter the right parameter instantiation. The annotation `Param("ds", Req)` means that the parameter is required to be defined by the automated deployer (by first creating an appropriate object of the desired type, and passing that as a parameter). Different scenarios (such as “DefaultUsage” and “HeavyUsage”) can be defined to capture variations in deployment requirements (for example, for different customers), and variations in resource consumption.

When a system deployment is automatically computed, a user expects to reach specific goals and could have some desiderata. For instance, in the considered Fredhopper Cloud Services use case, the goal is to

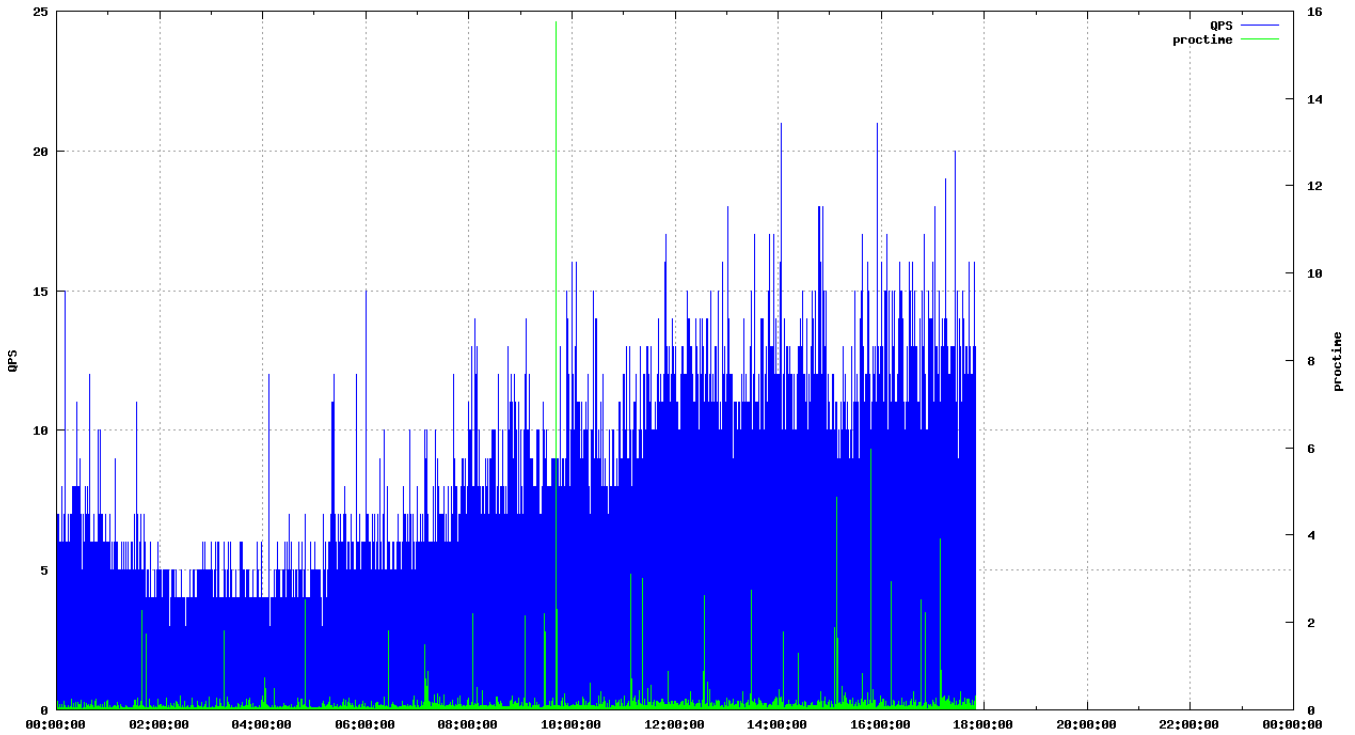


Figure 4.3: The number of queries per second (QPS) and roundtrip time (proctime) in seconds over time

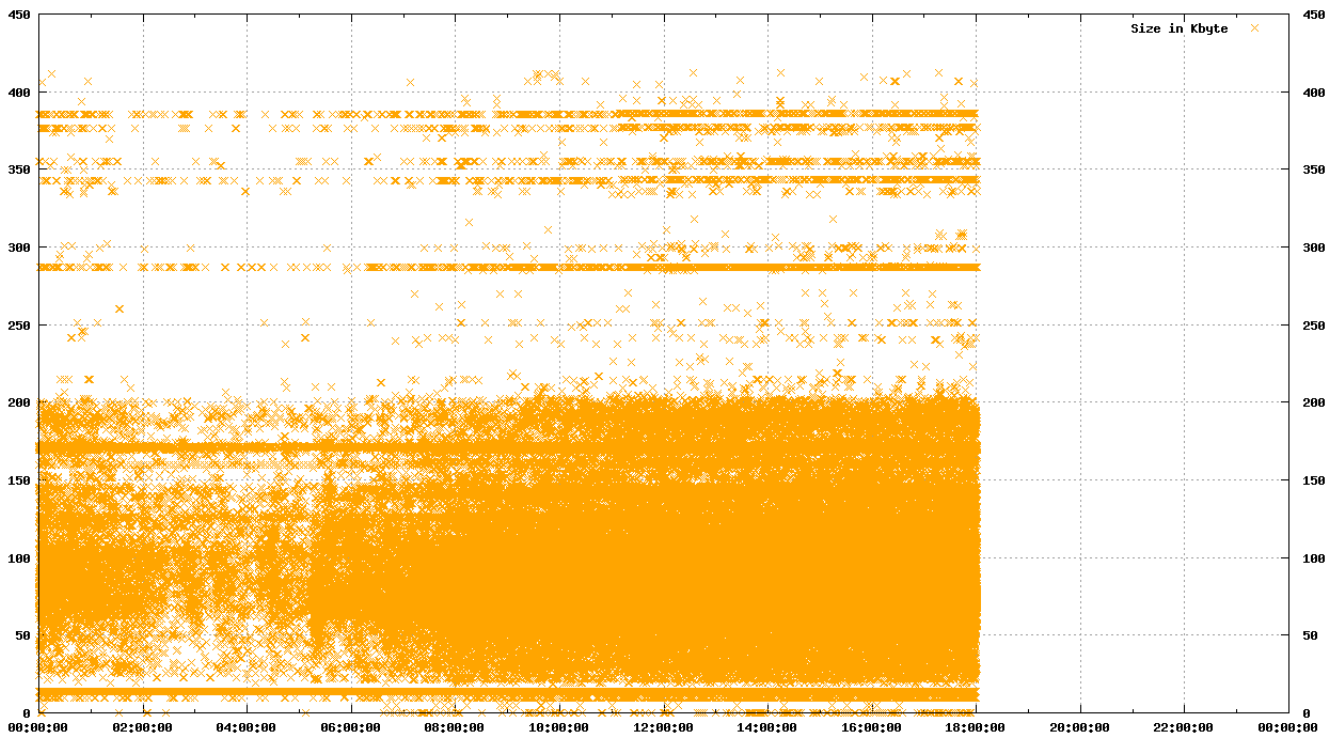


Figure 4.4: The response size (in kiB) of the Query Response over time

deploy a given number of Query Services and a Platform Service, possibly located on different machines (e.g., to improve fault tolerance).

```
[Deploy: scenario[Name("DefaultUsage"), MaxUse(1), Cost("CPU", 1), Cost("Memory", 3000),
    Param("c", User), Param("ds", Req)] ]
[Deploy: scenario[Name("HeavyUsage"), MaxUse(1), Cost("CPU", 2), Cost("Memory", 4500),
    Param("c", User), Param("ds", Req)] ]
class QueryServiceImpl(DeploymentService ds, Customer c) implements Service
```

Figure 4.5: Cost annotation of Query service

```
[Deploy: scenario[MaxUse(2), Cost("CPU", 1), Cost("Memory", 800),
    Param("rp", Req)]]
class DeploymentServiceImpl(InfrastructureService rp) implements DeploymentService
```

Figure 4.6: Cost annotation of the Deployment service

These goals and requirements are expressed in a *Declarative Deployment Language*: a language for stating the constraints that the final configuration should satisfy. For instance, the following constraint states that at least two `QueryService` instances and exactly one object of class `PlatformServiceImpl` should be deployed.

```
INTERFACE[IQueryService] >= 2 and CLASS[PlatformServiceImpl] = 1
```

More complex quantities involve constraints on the virtual machines used in deploying. For example, we can specify that no virtual machine with less than two CPUs should contain more than one object of class `QueryServiceImpl` as follows.

```
DC[ CPU <= 2 | CLASS[QueryServiceImpl] >= 2 ] = 0
```

Using such constraints it is also possible to express co-location or distribution requirements. For instance, for efficiency reasons it could be convenient to co-locate highly interacting objects or, for security or fault tolerance reasons, two objects should be required to be deployed separately. We can require that an object of class `QueryServiceImpl` must be always co-installed together with an object of class `DeploymentServiceImpl` with the following constraint.

```
DC[CLASS[QueryServiceImpl] > 0 and CLASS[DeploymentServiceImpl] = 0 ] = 0
```

4.1.2 Result

Based on the above deployment requirements, the ABS class annotations specifying resource consumption and inter-class dependencies, and the available virtual machines in the JSON file, the SmartDeploy tool automatically synthesizes a deployment configuration with the least cost. See Section 5.3 for a brief tool description of SmartDeploy. Figure 4.7 shows part of the generated code by SmartDeploy.

A graphical representation of this deployment configuration is shown in Figure 4.8. The external black and white boxes represent the deployment components used while the small boxes represent the object deployed. Every object has a name represented by its class prefix by its deployed scenario ¹. The red boxes belonging to an object *o* represent the objects that are required to invoke the creation of *o* or the addition of object reference to *o* such that later it can use them. The red boxes are connected to green boxes by arrows. The green boxes represent the object that are passed as arguments of the constructor of *o* or as argument of the methods to add their references to *o*.

The deployment configuration synthesized fully automatically by SmartDeploy is close to the production deployment manually configured by the Cloud operations team.

¹“Def” is used to represent the default deployment scenario when only one deployment scenario is available for that object.

```

Unit deploy() {
  DeploymentComponent c4_xlarge_us_1 =
    cloudProvider.prelaunchInstanceNamed("c4_xlarge_us");
  ls_DeploymentComponent = Cons(c4_xlarge_us_1,ls_DeploymentComponent);
  DeploymentComponent c4_xlarge_us_0 =
    cloudProvider.prelaunchInstanceNamed("c4_xlarge_us");
  ls_DeploymentComponent = Cons(c4_xlarge_us_0,ls_DeploymentComponent);
  DeploymentComponent c4_2xlarge_eu_0 =
    cloudProvider.prelaunchInstanceNamed("c4_2xlarge_eu");
  ls_DeploymentComponent = Cons(c4_2xlarge_eu_0,ls_DeploymentComponent);
  DeploymentComponent m4_large_eu_0 =
    cloudProvider.prelaunchInstanceNamed("m4_large_eu");
  ...
  [DC: c4_xlarge_us_0] LoadBalancerEndPoint
    oDef___LoadBalancerEndPointImpl_0_c4_xlarge_us_0 = new LoadBalancerEndPointImpl();
  ls_LoadBalancerEndPoint =
    Cons(Pair(oDef___LoadBalancerEndPointImpl_0_c4_xlarge_us_0,c4_xlarge_us_0),
        ls_LoadBalancerEndPoint);
  ls_EndPoint =
    Cons(Pair(oDef___LoadBalancerEndPointImpl_0_c4_xlarge_us_0,c4_xlarge_us_0),
        ls_EndPoint);
  [DC: c4_2xlarge_us_0] DeploymentService oDef___DeploymentServiceImpl_0_c4_2xlarge_us_0 =
    new DeploymentServiceImpl();
  ls_DeploymentService =
    Cons(Pair(oDef___DeploymentServiceImpl_0_c4_2xlarge_us_0,c4_2xlarge_us_0),
        ls_DeploymentService);
  [DC: c4_2xlarge_eu_0] DeploymentService oDef___DeploymentServiceImpl_0_c4_2xlarge_eu_0 =
    new DeploymentServiceImpl();
  ls_DeploymentService =
    Cons(Pair(oDef___DeploymentServiceImpl_0_c4_2xlarge_eu_0,c4_2xlarge_eu_0),
        ls_DeploymentService);
  [DC: m4_xlarge_us_0] IQueryService ostaging___QueryServiceImpl_0_m4_xlarge_us_0 =
    new QueryServiceImpl(oDef___DeploymentAgentImpl_0_m4_xlarge_us_0, "CustX", True);
  ...
}

```

Figure 4.7: Initial optimal deployment configuration by SmartDeploy

4.2 Dynamic Deployment

The previous section showed how we can synthesize an *initial* deployment configuration with optimal cost. The question arises how to support elasticity to scale the deployment configuration *dynamically* (at run-time). This required an extension of the deployment synthesis reported in D1.3.1 and applied in D4.3.2, to produce fully executable ABS code. In contrast to the static initial deployment scenarios described in the previous section, where deployment actions are applied at the very beginning of the execution, dynamic deployment actions are invoked *at user-specified points* during execution. The extension to support dynamic deployment actions was developed and implemented in a new tool SmartDeploy, reported in D1.3.2.

With the extension at hand, in collaboration with T1.3 and T2.3, in Deliverable D2.3.2 (Section Auto-Scaling) we developed a technique that fully supports elastic application, by integrating the SmartDeploy into the monitoring framework. The basic idea is to invoke deployment actions inside the monitors. Since the monitors formalize high-level metrics used in the SLAs (rather than lower-level metrics such as CPU usage), this approach provides a rigorous basis for auto-scaling based directly on the SLA.

In this section we report on the application of the dynamic deployment techniques described above to

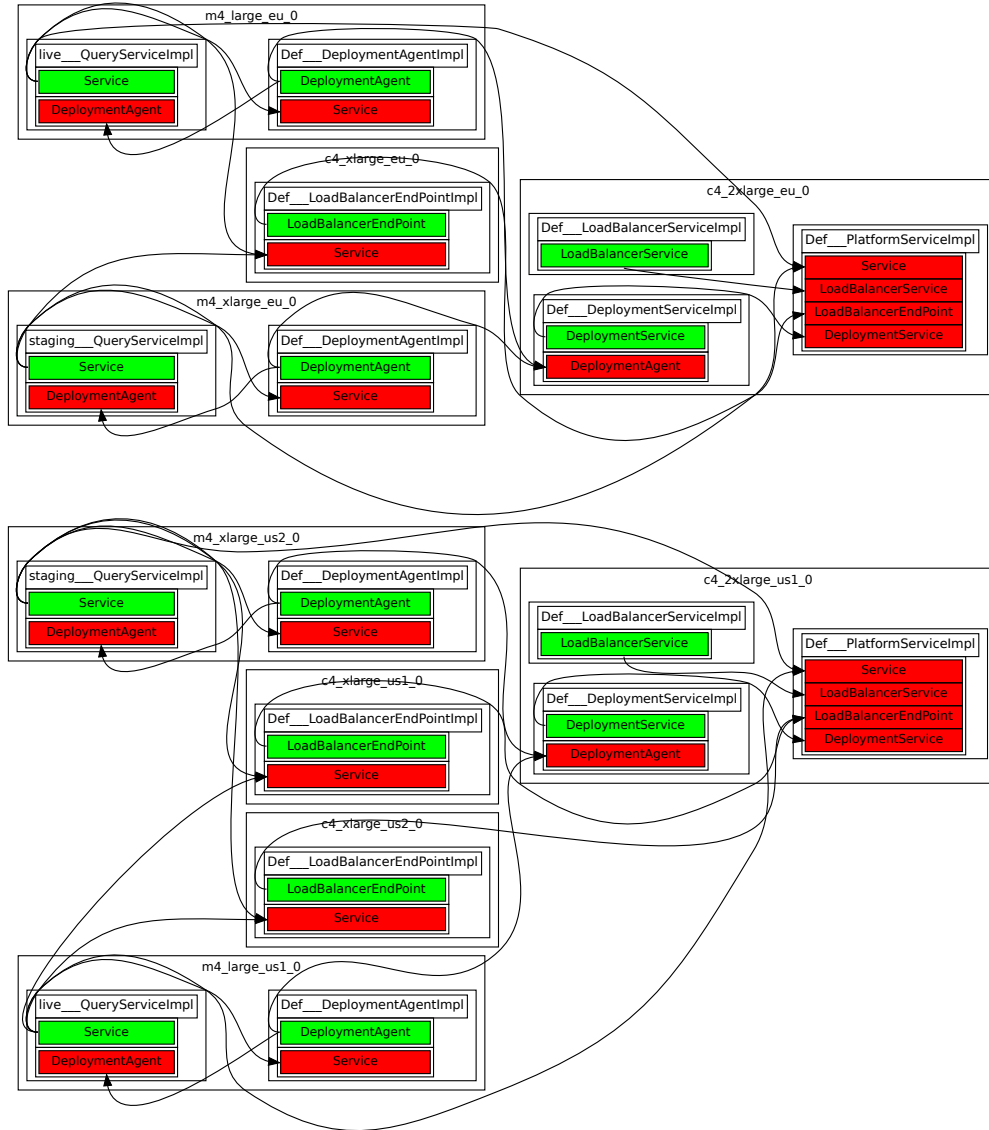


Figure 4.8: Initial Static Deployment Configuration Synthesized by SmartDeploy

the Fredhopper case study. The cost annotations for the dynamic deployment and JSON specification for the machine types remain unchanged from Section 4.1. We focus here on the aspects that differ.

4.2.1 Modeling

With domain-specific knowledge provided by the Cloud operation team, we first identified the deployment requirements that should be satisfied at run-time. We focus on the most interesting ones below. The main kind of service instances that should be scaled up or down at run-time is the **QueryService**. A **QueryService** can run in one of two modes:

- **Live:** a **QueryService** instance running in this mode, the instance is responsible for evaluating and responding to queries.
- **Staging:** in this mode, the instance additionally offers the capability to customers (webshops) to update the product catalog and update the product indices accordingly.

Dynamic Deployment Requirements

We require the presence of at least one instance in staging mode. This is formalized as follows:

```
QueryServiceImpl[staging] > 0
```

Furthermore, whenever a new **QueryService** should be deployed, we require a **DeploymentServiceImpl** present on the same virtual machine. This requirement was also needed for the static initial deployment (Section 4.1) and should be preserved at run-time. It can be expressed more concisely through the use of quantified expressions and matching based on regular expressions, which were newly introduced in D1.3.2 based on the experiences with the case study:

```
forall ?x in DC: (
  ?x.QueryServiceImpl['.*'] > 0 impl
  ?x.DeploymentServiceImpl > 0 )
```

Here **impl** stands for logical implication. The regular expression **'.*'** allows us to match with both deployment modalities for the Query Service (**staging** and **live**).

An important requirement for performance reasons and robustness is that load balancers must be installed on a dedicated virtual machine (without other Service instances). It is formalized as follows.

```
forall ?x in DC: (
  ?x.LoadBalancerServiceImpl > 0 impl
  (sum ?y in obj: ?x.?y) = ?x.LoadBalancerServiceImpl )
```

To increase fault tolerance, the Infrastructure provider currently used in the Fredhopper Cloud Services, Amazon, introduced the concept of *regions* and *Availability Zones*. Each region is a separate geographic area (such as US). Each region has multiple isolated locations known as Availability Zones. Amazon provides the ability to place resources, such as instances, and data in multiple locations. In everyday practice, this means that the Cloud operators scale the number of **QueryService** in two availability zones simultaneously, so that a failure in one zone does not affect the availability. This policy is captured with quantified expressions as follows:

```
(sum ?x in '.*_us1': ?x.QueryServiceImpl['.*']) =
(sum ?x in '.*_us2': ?x.QueryServiceImpl['.*'])
```

The left-hand side denotes the number of query service instances in the **us1** availability zone, and the right-hand side denotes the number of instances in the **us2** zone.

Binding preferences

In order to allow the connection of components with components located in the same availability zone or region, SmartDeploy introduced the notion of binding preferences. SmartDeploy first computes the components for the optimal configuration and then it uses these preferences to compute the final configuration establish the component connections that maximize the binding preferences.

As an example, the **QueryService** in the US region must be connected to the **LoadBalancerEndPoint** deployed in the same region. SmartDeploy enforces this with the following preference.

```
"sum ?x of type QueryServiceImpl['.*'] in '.*_us.' :
  forall ?y of type LoadBalancerEndPointImpl in '.*_us.' :
    ?x used by ?y",
```

This preference requires to maximize the number of connections between every **QueryService** deployed in US region and every **LoadBalancerEndPoint** in that same region. As a consequence all the **QueryService** in the US region will be connected to all the **LoadBalancerEndPoint** in the same region.

Binding preferences also make it possible to avoid establishing connections between certain objects, or add connections to existing objects instead of newly created instances.

Installation actions

To successfully deploy new service instances, certain installation actions should be executed. For example, whenever a new **QueryService** is added, it should be added to the appropriate load balancers (all load balancers in the same region as the new query service) through an **add(Service)** method. It should then be announced to the platform through an **addServiceInstance** method. The last step is to finalize the deployment of the new query service, by calling the **install** method on the deployment service.

The order of these actions are important and can be specified as follows:

```
"add_method_priorities":[
  {
    "class":"LoadBalancerEndPointImpl",
    "method":"add"
  },
  {
    "class":"PlatformServiceImpl",
    "method":"addServiceInstance"
  },
  {
    "class":"DeploymentAgentImpl",
    "method":"install"
  }
]
```

4.2.2 Result

SmartDeploy was able to automatically synthesize a dynamic deployment configuration an appropriate deployment configuration with optimal cost. It satisfies all the dynamic requirements, respects the specified binding preferences and generates fully executable ABS code that execute the installation actions arising from dynamic scaling actions in the desired order.

Figure 4.9 shows the resulting dynamic deployment configuration (modifying the current deployment configuration at run-time) synthesized automatically by the SmartDeploy. This picture depicts the addition of 2 **QueryService** in the US region with the requirement that their addition should be balanced between the two availability zones. The external black and white boxes represent the used deployment components and the small boxes represent the object deployed. Every object has a name represented by its class prefixed by its deployed scenario ². The red boxes belonging to an object *o* represent the objects that were required to create *o*, or are needed by *o* as references. The red boxes are connected to green boxed by arrows. The green boxes represent the objects that are passed as arguments of the constructor of *o*, or as argument of the installation methods of *o*. The deployment component on the right, denoted as “**___initial_DC**”, is a fictional location used to contain all the already available objects that can provide resources for the objects that need to be created. In this case the initial objects represent two Platform Services (one deployed in the US and one in the EU region), two LoadBalancerService, two DeploymentService, and three LoadBalancerEndPoint (one in Europe and two in US, one for every availability zone). As expected, SmartDeploy decided to require two new machines of type m4-large, one in the **us1** availability zone and one in the **us2** availability zone. In each machine, SmartDeploy installs both a **QueryService** and a **DeploymentAgent**.

²“Def” is used to represent the default deployment scenario when only one deployment scenario is available for that object.

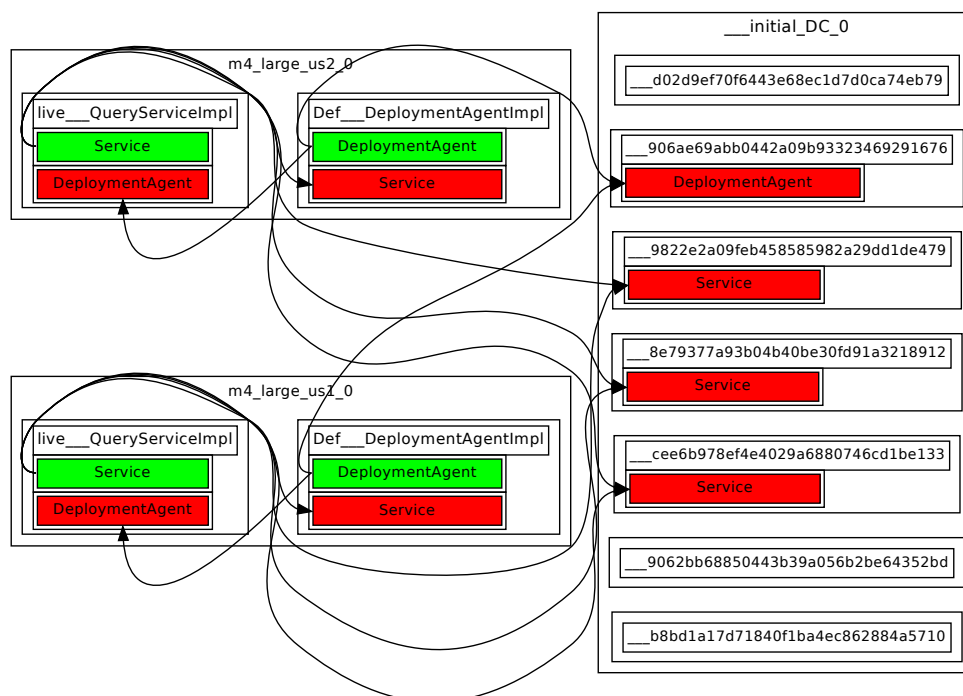


Figure 4.9: Dynamically scaling the deployment with new Query Service using SmartDeploy

Chapter 5

Application of Envisage Analyses

Analysis Name
Simulator
Deadlock Analysis
Resource Analysis
Monitoring
Verification
Deployment

Table 5.1: Overview of analyses used on the case study

5.1 Simulator

Throughout the project, the case study has used multiple back-ends. Initial focus was on the Maude back-end, subsequently (and currently) the Erlang back-end, and recently work on the Haskell back-end has been initiated. The Fredhopper Cloud Services exploits a wide-range of ABS features (see Section 3.4) and libraries (including the recently delivered HTTP API from D2.3.2), making it a challenging case for the back-ends.

The case study has been used to uncover many subtle problems (which subsequently led to fixes, see the bug tracker¹) and triggered improvements to i.e., Deployment Modeling (the `CloudProvider` as well as `DeploymentComponent`), Timed-ABS and the ABS libraries (some of the feedback has been reported formally in Chapter 3 of D4.3.2, D4.5 details how it was addressed). The improvements led to corresponding extensions in the back-ends.

5.2 Monitoring SLAs

To ensure a high quality of service, web shops negotiate an aggressive Service Level Agreement (SLA) with FRH. QoS attributes of interest include query latency (*response time*) and throughput (*queries per second*). At FRH, the following SLA negotiated with a customer expresses *service degradation* requirements (the exact percentages are negotiable):

“Services must maintain 100 queries per second with less than 200 milliseconds respectively 500 ms according to the target percentages in Table 5.2, ignoring the 2% slowest queries.”

¹<https://github.com/abstools>

Customer	$\leq 200\text{ms}$	$\leq 500\text{ms}$
cust1	99.5%	99.9%
cust2	95%	99%

Table 5.2: Target query proctimes for Service Degradation SLA per Customer

An SLA specifies properties of service metric functions. In general, a service metric function maps an event trace of interactions of customers with the Service APIs to a value, which indicates the corresponding QoS level of the metric. In this case, the service metric function is defined in terms of the percentage of client requests which are processed in a “slow” manner. For the example SLA, the service degradation is concerned with the percentage of queries slower than 200 (500) milliseconds. The two percent slowest queries are subtracted for the following reason. When a new query service is instantiated, certain initialization steps must be executed. During initialization, the processing time is much higher and would immediately result in violations of the SLA.

In this section we formalize the above service metric function using a service view and an attribute grammar. From these two ingredients, an executable ABS monitor with optimal performance is synthesized fully automatically. The theory underlying the formalization of service metrics with grammars is described in D2.3.2, Chapter 3.

5.2.1 Service View

A service view names the events in the Service APIs relevant for the service metric. The service view for the degradation metric is shown in Figure 5.1.

```
view Degradation {
  invoke(Time t, Customer customer, Rat procTime) query
}
```

Figure 5.1: Service View for Degradation

It identifies the invoke event as the only relevant event for Service Degradation and associates the name query. The attribute Time t indicates the time the event was issued, Customer customer identifies the customer, i.e. web-shop that issued it (the type Customer is an alias for String), and Rat procTime indicates the processing time of the event.

5.2.2 Grammar

Grammars map event traces to QoS values. In our running example of Service Degradation, we map a trace of events named query from the service view to the degradation percentages per customer for queries slower than 200ms respectively 500ms. The Service Degradation grammar declares 5 attributes (line 3 - 17):

- `slowQpct`: this is the main attribute formalizing the Service Degradation percentage per customer. As an example, if the percentage of queries slower than 200ms for a customer named “cust1” is 1.3%, `slowQpct` maps the string “cust1.fas.live.200ms” to the value 1.3.
- `degradationSLA`: this mapping formalizes the SLA requirements per customer: for each metric it yields the agreed (with the customer) target value of that metric.
- `cnt` is a helper attribute indicating the total number of events per customer.
- `slow200Cnt` is a helper attribute storing the total number of queries slower than 200ms per customer.
- `slow500cnt` is a helper attribute storing the total number of queries slower than 500ms per customer.

Figure 5.2 shows how these attributes are updated with actions in the grammar. The local variables `new200` and `new500` store the new value of the number of queries slower than 200ms respectively 500ms, and they are updated (line 22 and 23) based on the `procTime` (see the service view in Figure 5.1) of the query.

```

1  type Customer = String;
2
3  // main metric attribute
4  Map<String, Rat> slowQpct = EmptyMap;
5
6  // SLA requirements
7  Map<String, Rat> degradationSLA =
8    map[Pair("cust1.fas.live.200ms", 5/1000),
9          Pair("cust1.fas.live.500ms", 1/1000),
10         Pair("cust2.fas.live.200ms", 5/100),
11         Pair("cust2.fas.live.500ms", 1/100)
12    ];
13
14  // auxiliary attributes
15  Map<Customer, Int> cnt = EmptyMap;
16  Map<Customer, Int> slow200Cnt = EmptyMap;
17  Map<Customer, Int> slow500Cnt = EmptyMap;
18
19  S ::= query
20    { Int old200 = lookup(slow200Cnt, customer);
21      Int old500 = lookup(slow500Cnt, customer);
22      Int new200 = old200 + case(procTime>200) { True => 1; False => 0;};
23      Int new500 = old500 + case(procTime>500) { True => 1; False => 0;};
24      slow200Cnt = put(slow200Cnt, Pair(customer, new200));
25      slow500Cnt = put(slow500Cnt, Pair(customer, new500));
26
27      cnt=cnt+1;
28      Int twoPct = 2*cnt/100;
29      slowQpct = insert(slowQpct, Pair(customer + ".fas.live.200ms",
30                                100*max(0,new200-twoPct)/cnt),
31                      Pair(customer + ".fas.live.500ms",
32                                100*max(0,new500-twoPct)/cnt));
33    }
34  S

```

Figure 5.2: Grammar for Service Degradation

5.2.3 Monitor

From the grammar, the monitoring framework automatically synthesizes executable ABS code. The generated monitor is resource efficient: CPU and memory consumption is constant per event. The values of the metrics over time are visualized with Grafana. Figure 5.3 and Figure 5.4 show the percentage of queries *slower* than 200ms, respectively 500ms (minus the 2% lowest queries) based on feeding the Fredhopper Cloud Services in ABS with input data from a real-world log file, by invoking a `LoadBalancerEndPointImpl` endpoint. The figures are screenshots of the Service Degradation monitor defined in the previous subsections, visualized using the ABS monitoring visualization framework (see D2.3.2).

After a short initialization time (beginning of Figure 5.3), the target values for the Service Degradation in Table 5.2 are met (Figure 5.4).

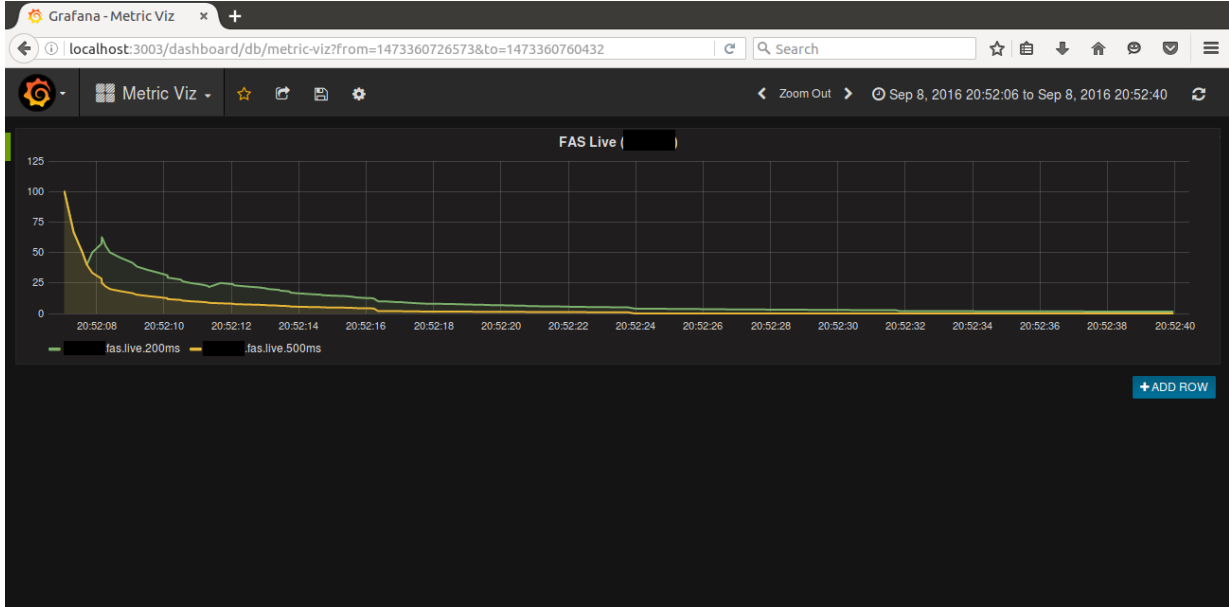


Figure 5.3: Degradation Visualization Overview

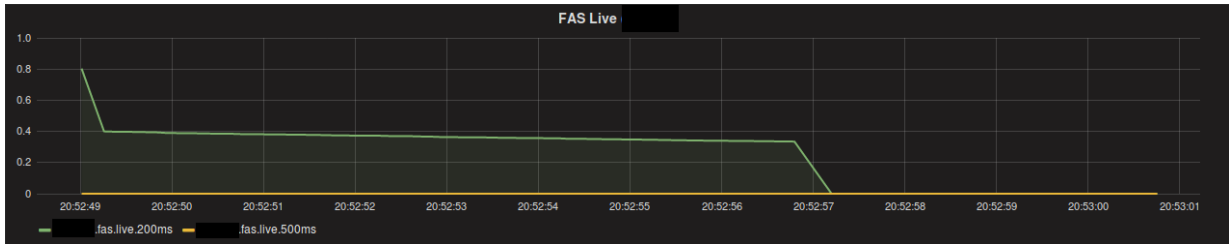


Figure 5.4: Degradation Visualization Zoomed

5.3 Deployment

SmartDeploy is a tool that automatically synthesizes an appropriate deployment configuration with minimal cost. It takes into account the declarative deployment requirements, an ABS model with cost annotations for the services to deploy, the kinds and number of available virtual machines, and optionally binding preferences and installation action priorities (Chapter 4).

SmartDeploy relies on Zephyrus2 [1], i.e., a configuration optimizer that allows to compute the optimal configuration to deploy on different computational resources. Zephyrus2 uses constraint programming techniques to solve in sequence several NP-hard minimization problems. Thus, these problems being NP-hard, there are no polynomial time guarantees on the performances of SmartDeploy. However, in practice SmartDeploy can address and compute the optimal allocation of a hundred of ABS objects in dozens of different deployment component types in matter of seconds. A detailed evaluation of the scalability of SmartDeploy is left as a future work.

SmartDeploy generates as output a delta module (≈ 300 lines of code). This allows the definition of classes needed to trigger the creation of the deployment components and the object creation. The added classes may be freely used inside the ABS program.

In the end, after a close collaboration between the case study owners and the SmartDeploy tool developers, SmartDeploy was able to faithfully capture all the different aspects of the case study. The case study has driven support for many of the new features reported in D1.3.2, such as the ability to execute dynamic deployment actions, use quantified expressions and regular expression matching. Some of the extensions go

beyond even what was reported in D1.3.2, specifically, the binding preferences and the ordering of installation actions. (see Section 4.2).

In the existing system, the Cloud operations team manually decides to scale, and since FRH has very aggressive SLAs, the team is typically conservative with downscaling, leading to potential over-spending. The ability of SmartDeploy to deploy in the programming language (ABS) itself allows to leverage the extensive tool-supported analyses available for ABS. This enabled us to integrate deployment/undeployment actions inside monitors that track the quality of services, thereby providing a rigorous basis (the formalized SLA agreed with the customer) for auto-scaling. For example, the monitor in Figure 5.5 for Service Degradation (see Section 5.2.3) shows a simple example how we can scale up or down based on the current QoS vs. the SLA.

```

1  Unit monitor() {
2      Set<String> slaMetrics = keys(degradationSLA);
3
4      while(!emptySet(slaMetrics)) {
5          String metricName = take(slaMetrics);
6          Rat latestMeasurement = snd(head(lookup(metricHist, metricName)));
7          if(latestMeasurement > lookup(degradationSLA, metricName)) {
8              deployer.scaleUp();
9          } else if(2*latestMeasurement < lookup(degradationSLA, metricName)) {
10             deployer.scaleDown();
11         }
12     }
13 }

```

Figure 5.5: Corrective Actions for Service Degradation

Furthermore, while the operations team currently uses ad-hoc scripts to configure newly added or removed service instances, and these scripts are specific to the infrastructure provider, SmartDeploy automatically generates code that accomplishes this. SmartDeploy is flexible in the sense that it is infrastructure independent, allowing to seamlessly switch between different infrastructure providers: virtual machines are launched and terminated through a generic Cloud API offered by ABS for managing virtual resources. Executable code is automatically generated from ABS for any of the infrastructures for which an implementation of the Cloud API exists (e.g., Amazon, Docker, OpenStack).

5.4 Deadlock Analysis

We used the SACO deadlock analysis to check whether the model is deadlock free. The application of the deadlock analysis took less than one second and returns that the program is deadlock free. In particular, this ensures that the asynchronous communication between different services (see Chapter 2 and chapter 3) do not introduce deadlocks. The output of the tool shows:

```

Abs program loaded in 25 ms.
Rule based representation generated in 73 ms.
Points-to analysis: Fixpoint reached in 281 iterations
Points-to analysis performed in 71 ms.
LMhp analysis performed in 84 ms.
Mhp graph created in 12 ms.
Closure time 186 ms.
Deadlock analysis 9 ms.
Discarded 0 cycles with freshness analysis.

The program is deadlock free

```


Complete analysis performed in 408 ms.

5.5 Resource Analysis

The resource analysis of the model has been done by using SACO, which is integrated in the collaborative tool suite published in <http://abs-models.org/>. We used SACO to identify potential bottlenecks in the Fredhopper Cloud Services and analyze the resource consumption of various services. To perform the analysis, some modifications on the original model were required to properly handle the model:

- Some rational numbers were removed to properly maximize expressions.
- We replaced sets by lists, because it is needed by SACO to guarantee that the loops that traverse the data structure terminate.
- Some annotations were added to the model to incorporate to the source code some information about the “size relations” of certain variables of the model.
- Due to a “possible” inheritance, the analysis identifies a possible recursion which could imply the non-termination of the method. Thus, we have changed the type of one variable to allow SACO to prove termination of the method.
- In the model, there are two intentional **while(True)** loops: these concern methods in Services that should keep running indefinitely, such as **MonitoringService**. The termination analysis has identified them and indicates that these methods will not terminate. However, to obtain the cost associated to the cost «annotations», we have changed those loops (introducing a fresh variable that bounds the number of iterations) in order to obtain the total cost of the program starting from the **main** method.

5.5.1 Termination

We used SACO’s Resource Analysis to evaluate the termination of the methods contained in the model. We applied the *termination* cost model to all methods in the model and obtained the following results:

```
Method InfrastructureServiceImpl.cpu terminates?: YES
Method InfrastructureServiceImpl.cpu terminates?: YES
Method InfrastructureServiceImpl.acquireInstance terminates?: YES
Method InfrastructureServiceImpl.release terminates?: YES
Method LoadBalancerEndPointImpl.remove terminates?: YES
Method LoadBalancerEndPointImpl.add terminates?: YES
Method LoadBalancerEndPointImpl.invoke terminates?: YES
Method LoadBalancerEndPointImpl.setStatus terminates?: YES
Method LoadBalancerEndPointImpl.getStatus terminates?: YES
Method ServiceImpl.getCPU terminates?: YES
Method ServiceImpl.getResource terminates?: YES
Method ServiceImpl.cost terminates?: YES
Method ServiceImpl.getLatency terminates?: YES
Method ServiceImpl.getStatus terminates?: YES
Method ServiceImpl.setStatus terminates?: YES
Method ServiceImpl.getServiceId terminates?: YES
Method ServiceImpl.getServiceType terminates?: YES
Method ServiceImpl.getCustomer terminates?: YES
Method DeploymentServiceImpl.install terminates?: YES
Method DeploymentServiceImpl.lookup terminates?: YES
Method DeploymentServiceImpl.uninstall terminates?: YES
Method DeploymentServiceImpl.invoke terminates?: YES
```

```

Method DeploymentServiceImpl.start terminates?: YES
Method DeploymentServiceImpl.stop terminates?: YES
Method LoadBalancerServiceImpl.change terminates?: YES
Method LoadBalancerServiceImpl.decrease terminates?: YES
Method LoadBalancerServiceImpl.increase terminates?: YES
Method LoadBalancerServiceImpl.status terminates?: YES
Method LoadBalancerServiceImpl.enable terminates?: YES
Method LoadBalancerServiceImpl.add terminates?: YES
Method LoadBalancerServiceImpl.getServiceEndPoint terminates?: YES
Method PlatformServiceImpl.createService terminates?: YES
Method PlatformServiceImpl.createServiceInstance terminates?: YES
Method PlatformServiceImpl.uninstallInstance terminates?: YES
Method PlatformServiceImpl.alterResource terminates?: YES
Method PlatformServiceImpl.getEndpoints terminates?: YES
Method PlatformServiceImpl.getServiceIds terminates?: YES
Method PlatformServiceImpl.getService terminates?: YES
Method PlatformServiceImpl.findVM terminates?: YES
Method PlatformServiceImpl.matchResources terminates?: YES
Method RepeatUserImpl.use terminates?: UNKOWN
Method RepeatUserImpl.invoke terminates?: YES
Method RepeatUserImpl.invokeWithSize terminates?: YES
Method RepeatUserImpl.getState terminates?: YES
Method MonitoringServiceImpl.add terminates?: YES
Method MonitoringServiceImpl.run terminates?: UNKOWN
Method MonitoringServiceImpl.execute terminates?: YES
Method LatencyMonitor.scaling terminates?: YES
Method LatencyMonitor.getLatencies terminates?: YES
Method LatencyMonitor.monitor terminates?: YES
Method ScaleResourceAction.action terminates?: YES
Method ServiceProviderImpl.addCustomer terminates?: YES
Method QueryServiceImpl.invoke terminates?: YES
Method QueryServiceImpl.setStatus terminates?: YES
Method QueryServiceImpl.getStatus terminates?: YES
Method QueryServiceImpl.getServiceId terminates?: YES
Method QueryServiceImpl.getServiceType terminates?: YES
Method QueryServiceImpl.getCustomer terminates?: YES
Method QueryServiceImpl.getLatency terminates?: YES
Method QueryServiceImpl.cost terminates?: YES
Method QueryServiceImpl.getCPU terminates?: YES
Method QueryServiceImpl.getResource terminates?: YES
Method ExampleSetup.getConfigs terminates?: YES
Method ExampleSetup.main terminates?: UNKOWN
Method DeploymentComponent.total terminates?: YES

```

SACO guarantees the termination of all methods except those methods with the **while(True)** loops. These methods (also emphasized in the listing) are: (1) `MonitoringServiceImpl.run`, which is responsible for monitoring the behavior of the model, and (2) `RepeatUserImpl.use`, which simulates a simple example user that repeatedly invokes requests to `ServiceImpl`. Since the `main` invokes these methods, SACO also (correctly) does not prove that `main` terminates.

5.5.2 Analysis of Relevant Methods

The next step of the work aims to infer the complexity of some particular methods of interest. This exposes potential bottlenecks. The resource analysis has been performed on the number of steps and the results are summarized in the following table:

Method	Complexity
LoadBalancerServiceImpl::change	n^2
QueryServiceImpl::invoke	1
LoadBalancerEndPointImpl::invoke	1
ServiceImpl::invoke	1
DeploymentServiceImpl::invoke	n
PlatformServiceImpl::createService	n^2
PlatformServiceImpl::incrService	n^3

5.5.3 Objects

The number of object creations are an important measure for memory consumption. SACO was able to successfully obtain the following upper-bound:

```

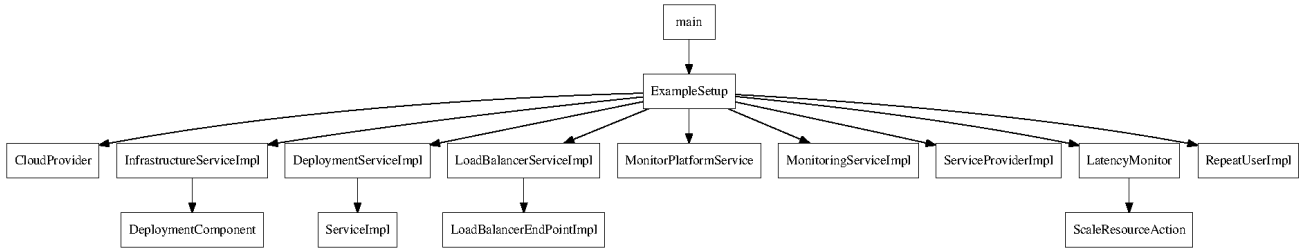
c(object('ExampleSetup'))
+ c(object('CloudProvider'))
+ c(object('InfrastructureServiceImpl'))
+ c(object('DeploymentServiceImpl'))
+ c(object('LoadBalancerServiceImpl'))
+ c(object('PlatformServiceImpl'))
+ c(object('MonitoringServiceImpl'))
+ c(object('ServiceProviderImpl'))
+ c(object('LatencyMonitor'))
+ 2*c(object('DeploymentComponent'))
+ nat(max(monitorIterations))*nat(max(monitorsToBeRun)) * (
  c(object('ScaleResourceAction'))
+nat(max(latencyMonitor_scalings))*(c(object('ServiceImpl'))+c(object('DeploymentComponent'))))
+ nat(max(exampleSetup_configs))*nat(max(exampleSetup_instancesConfigs)) * (
  nat(max(num_instancesInConfig))*(c(object('ServiceImpl'))+c(object('DeploymentComponent'))))
+ c(object('LoadBalancerEndPointImpl'))
+ nat(max(exampleSetup_usersInConfig))*c(object('RepeatUserImpl'))

```

From the upper-bound expression we can compute the information of the following table, which shows the number of objects of each type created by the model:

Class	Upper-bound on the number of instances
ExampleSetup	1
CloudProvider	1
InfrastructureServiceImpl	1
DeploymentServiceImpl	1
LoadBalancerServiceImpl	1
PlatformServiceImpl	1
MonitoringServiceImpl	1
ServiceProviderImpl	1
LatencyMonitor	1
DeploymentComponent	$2 * \text{nat}(\text{max}(\text{monitorIterations})) * \text{nat}(\text{max}(\text{monitorsToBeRun})) + \text{nat}(\text{max}(\text{setup_configs})) * \text{nat}(\text{max}(\text{setup_instancesConfigs}))$
ScaleResourceAction	$\text{nat}(\text{max}(\text{monitorIterations})) * \text{nat}(\text{max}(\text{monitorsToBeRun}))$
ServiceImpl	$\text{nat}(\text{max}(\text{monitorIterations})) * \text{nat}(\text{max}(\text{monitorsToBeRun})) + \text{nat}(\text{max}(\text{latencyMonitor_scalings})) + \text{nat}(\text{max}(\text{setup_configs})) * \text{nat}(\text{max}(\text{setup_instancesConfigs})) + \text{nat}(\text{max}(\text{num_instancesInConfig}))$
LoadBalancerEndPointImpl	$\text{nat}(\text{max}(\text{setup_configs})) * \text{nat}(\text{max}(\text{setup_instancesConfigs}))$
RepeatUserImpl	$\text{nat}(\text{max}(\text{setup_configs})) * \text{nat}(\text{max}(\text{setup_instancesConfigs})) + \text{nat}(\text{max}(\text{setup_usersInConfig}))$

By means of a *points-to* analysis, SACO deduces that the object creations are captured by the following graph, where edges indicates the object responsible of creating each instance:



5.5.4 Cost Annotations

In addition to the complexity analysis discussed above, the model also contains some program points of interest which contain annotations of the form `[Cost: x]`, where `x` represents the cost of executing this program point. The use of cost annotations allowed to abstract away from detailed low-level implementations. By using the *user* cost model SACO was able to infer the cost of all annotated program points.

- Method `ServiceImpl::invoke`: `[Cost: 1]`

This annotation is used to represent the cost of executing one service invocation and this cost is accumulated in the field `cost`. By means of SACO we can determine that the maximum cost accumulated in `cost` depends on the expression:

$$\text{nat}(\text{max}(\text{exSetup_configs})) * \text{nat}(\text{max}(\text{exSetup_configs})) * \text{nat}(\text{max}(\text{exSetup_usersInConfig})) * \text{nat}(\text{max}(\text{nUses}))$$

where `exSetup_configs` represents of configurations to test in the object `ExampleSetup`, `exSetup_usersInConfig` the number of users which will try each configuration and the number of requests performed by each user (`nUses`).

- Method `LoadBalancerServiceImpl::add`: `[Cost: length(services)]`

This annotation captures the cost of creating a new `LoadBalancerEndPointImpl`, which depends on the number of `services` used by the balancer. The following expression bounds this cost:

$$\text{nat}(\max(\text{exSetup_configs})) * \text{nat}(\max(\text{exSetup_configs})) * \text{nat}(\max(\text{exSetup_configs}))$$

As before, `exSetup_configs` represents the configurations created in `ExampleSetup`.

- Method `PlatformServiceImpl::createServiceInstance`: [Cost: 20]

This annotation models the cost of creating a new service. With this annotation we can bound the cost of creating service instances within the model, with respect to the initial configuration of `ExampleSetup`

$$\begin{aligned} &20 * \text{nat}(\max(\text{monitorIterations})) * \text{nat}(\max(\text{monitServiceImpl_monitorMap})) \\ &* \text{nat}(\max(\text{latencyMonitor_scalings})) + \\ &20 * \text{nat}(\max(\text{exSetup_configs})) * \text{nat}(\max(\text{exSetup_configs})) \\ &* \text{nat}(\max(\text{exSetup_configs})) \end{aligned}$$

In this expression we see that there are two ways of creating new service instances, one which only depends on `exSetup_configs` (the initial configuration), and another way that depends on the monitoring service and the amount of scales needed during execution of the model.

5.5.5 Object Sensitive Analysis

We used the object sensitive analysis to find out which objects are responsible for executing the cost-annotated statements. We obtained the following results:

```
UB for ([14,main],ExampleSetup): 0
UB for ([6,14],CloudProvider): 0
UB for ([7,14],InfrastructureServiceImpl): 0
UB for ([8,14],DeploymentServiceImpl): 0
UB for ([9,14],LoadBalancerServiceImpl):
    nat(max(exSetup_configs))*nat(max(exSetup_configs)) * nat(max(exSetup_configs))
UB for ([10,14],PlatformServiceImpl):
    20*nat(max(exSetup_configs))*nat(max(exSetup_configs))*nat(max(exSetup_configs))+
    20*nat(max(monitorIterations))*nat(max(monitoringServiceImpl_monitorMap))*
    nat(max(latencyMonitor_scalings))
UB for ([11,14],MonitoringServiceImpl): 0
UB for ([12,14],ServiceProviderImpl): 0
UB for ([13,14],LatencyMonitor): 0
UB for ([1,7],DeploymentComponent): 0
UB for ([5,14],RepeatUserImpl): 0
UB for ([4,13],ScaleResourceAction): 0
UB for ([3,9],LoadBalancerEndPointImpl):
    nat(max(exSetup_configs))*nat(max(exSetup_configs))*nat(max(exSetup_usersInConfig))*
    nat(max(nUses))
UB for object ([2,8],ServiceImpl):
    nat(max(exSetup_configs))*nat(max(exSetup_configs))*
    nat(max(exSetup_usersInConfig))*nat(max(nUses))
UB for object ([main],main): 0
```

These results confirm that the objects responsible of executing the cost annotations are the instances of the classes `LoadBalancerServiceImpl`, `PlatformServiceImpl`, `LoadBalancerEndPointImpl` and `ServiceImpl`. Due to a loss of precision in the analysis, the cost that should be attributed to `ServiceImpl` is also attributed to the instance of `LoadBalancerEndPointImpl`. This loss of precision happens because the method `invoke` is implemented by both classes and the dynamic dispatching is over-approximated by the analysis.

5.5.6 SACO improvements

The use of SACO to analyze the resource consumption of the industrial case study has lead to various improvements in SACO, needed to obtain relevant information of the model.

- The possibility to add program variables into the cost annotations was added to SACO. In the beginning of the project, cost annotations could only handle constant values. The cost expressions are maximized with respect to the context of the program point where the annotation is placed, giving the worst case cost of such cost expression.
- To upper-bound expressions, the possibility of adding cost centers with program points was added, allowing SACO users the possibility to obtain the cost of all program points, or restrict upper bounds to the program points of interest. This option was crucial in the analysis of monitors.
- New kinds of annotations to include «size relations» between variables which cannot be inferred with enough precision with the automatic size analysis.
- The case study uncovered a bug in SACO's points-to analysis, which is now fixed.

In conclusion, the application of SACO to the Fredhopper Cloud Services successfully allowed to infer the resource consumption of services and exposed potential bottlenecks. In the other direction, the SACO developers reported that the application helped them to find the requirements of a real world case study and the kind of resource information that could be useful for business purposes. As future work for SACO, together with SACO developers we are exploring the possibility of adding “user-defined” cost centers to the cost annotations, allowing the programmers to “group” the cost annotations and the possibility of identifying the program point where a «size» annotation would be needed to get a properly maximized upper-bound. Currently, the expertise of the programmer is needed to add them at concrete program points where maximizing the expressions is problematic for SACO.

5.6 Verification

We used KeY ABS to verify correctness of generated monitors. In particular, we verified that a monitor for the Service Degradation metric takes appropriate corrective scaling actions whenever the SLA for the Fredhopper Query Service is threatened. This work is discussed in detail in Deliverable D2.3.2.

Chapter 6

Conclusion

This deliverable reported on the final version of the FRH case study. We discussed how Envisage improves the existing work-flow in Fredhopper, and, vice versa, how the case study covers the project objectives. Before Envisage, the status at Fredhopper was as follows:

- SLAs were a natural language document.
- There is no formalized model to derive suitable deployment configurations from the SLA automatically.
- SLAs were not amenable to formal, automated analyses. For example, there was no means to automatically generate monitors for SLAs.
- Low-level implementation details present in the in-production version of the Fredhopper Cloud Services complicated the application of formal analyses.

Our approach was model-based. We developed a resource-aware model of the Fredhopper Cloud Services in ABS, which proved to be a powerful language able to capture all relevant aspects of the case study (see Chapter 3, the main Code statistics are summarized in Table 6.1), we modeled real-world deployment scenarios (Chapter 4) and applied the Envisage Analyses (Chapter 5).

Code Metric	Year 1	Year 2	Final
Lines of code	1230	1410	2666
Functions	30	30	41
Classes	13	16	21
Interfaces	15	15	21
Data types and type synonyms	8	8	9

Table 6.1: Statistics

This allowed us to achieve the following improvements:

- SLAs are formalized in a declarative formalism.
- Optimal (in cost) deployment configurations that take the SLA into account are synthesized automatically.
- SLAs and the Fredhopper Cloud Services in general are amenable to formal, (semi-) automated analyses. For example, verifiable correct executable monitors with optimal resource consumption are generated fully automatically from SLAs.
- The model-based approach allowed us to abstract away irrelevant implementation details enabling us to successfully analyze various formal properties of the Fredhopper Cloud Services.

Summarizing, this provides a promising basis for industrial take-up of systematic development, configuration and deployment, and analysis of SLA-aware services. Beyond Envisage, we intend to explore a further integration of the ABS monitors for SLAs into the in-production Fredhopper Cloud Services.

Bibliography

- [1] Erika Abraham, Florian Corzilius, Einar Broch Johnsen, Gereon Kremer, and Jacopo Mauro. Zephyrus2: On the Fly Deployment Optimization using SMT and CP Technologies. In *Symposium on Dependable Software Engineering (SETTA 2016)*, 2016. To appear.
- [2] Elvira Albert, Frank S. de Boer, Reiner Hähnle, Einar Broch Johnsen, Rudolf Schlatter, Silvia Lizeth Tapia Tarifa, and Peter Y. H. Wong. Formal modeling of resource management for cloud architectures: An industrial case study using Real-Time ABS. *Journal of Service-Oriented Computing and Applications*, 8(4):323–339, 2014.
- [3] Stijn de Gouw, Michael Lienhardt, Jacopo Mauro, Behrooz Nobakht, and Gianluigi Zavattaro. On the integration of automatic deployment into the ABS modeling language. In Schahram Dustdar, Frank Leymann, and Massimo Villari, editors, *Service Oriented and Cloud Computing - 4th European Conference, ESOC 2015, Taormina, Italy, September 15-17, 2015. Proceedings*, volume 9306 of *Lecture Notes in Computer Science*, pages 49–64. Springer-Verlag, 2015.
- [4] Stijn de Gouw, Jacopo Mauro, Behrooz Nobakht, and Gianluigi Zavattaro. Declarative elasticity in ABS. In *Service-Oriented and Cloud Computing - 5th IFIP WG 2.14 European Conference, ESOC 2016, Vienna, Austria, September 5-7, 2016, Proceedings*, pages 118–134, 2016.
- [5] Initial User Requirements, January 2014. Deliverable D4.1 of project FP7-610582 (ENVISAGE), available at <http://www.envisage-project.eu>.
- [6] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatter, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer-Verlag, 2011.

Glossary

Terms and Abbreviations

Cloud Engineer A *Cloud Engineer* handles the day-to-day operation of the Fredhopper Cloud Services. She deploys/updates services through PaaS and IaaS according to incomplete *service requirements* from *Consultants*, diagnoses issues at service-level and either resolves them at real time or informs the *Support Engineers* and/or the *Software Engineers*. She manages the up and down scaling of service resources according to alerts and metric visualizations provided by the monitoring system. She also performs any necessary infrastructural changes to the Fredhopper Cloud Services

Consultant A *Consultant* manages the technical setting that enables *Customer* to use the APIs offered by the Fredhopper Cloud Services. She provides *service requirements* to *Cloud Engineers*

Customer A *Customer* is a business entity that powers her on-line shop using the APIs provided by the Fredhopper Cloud Services

Faceted Navigation Faceted navigation is a technique for accessing information organized according to a faceted classification system, allowing users to explore a collection of information by applying multiple filters. Facets correspond to properties of the information elements

Fredhopper Cloud Services A set of services managed by FRH through cloud computing that allows the offering of search and targeting facilities on a large product database to e-Commerce companies

Full Text Search In text retrieval, full-text search refers to techniques for searching a single computer-stored document or a collection in a full text database

IaaS Infrastructure as a Service

Infrastructure as a Service A provision model in which an organization outsources the equipment used to support IT operations, including storage, hardware, servers and networking components. The service provider owns the equipment and is responsible for housing, running and maintaining it. The client typically pays on a per-use basis

JSON JavaScript Object Notation. A data format that uses human-readable text to transmit data objects consisting of attribute-value pairs.

PaaS Platform as a Service

Platform as a Service A category of cloud service offerings that facilitates the deployment of applications without the cost and complexity of buying and managing the underlying hardware and software and provisioning hosting capabilities

QoS Quality of Service

Quality of Service Generic term encapsulating all the non-functional aspects of a service delivery

Resource Configuration A description of the number of service instances initially required for a service offered to a *Customer* and the virtualized resource to be allocated initially to those service instances

SaaS Software as a Service

Service Level Agreement A legal contract between a service provider and his customer. It records a common understanding about services, priorities, responsibilities, guarantees, and warranties

Service Requirement A service requirement consists of the agreed SLA and the *Customer's* specific configuration such as expected query throughput based on historical data in terms of monthly and peak page views

SLA Service Level Agreement

Software as a Service A software delivery model in which software and associated data are centrally hosted on the cloud. SaaS is typically accessed by users using a thin client via a web browser

Software Engineer A *Software Engineer* develops and maintains the Fredhopper Cloud Services. She provides technical support to *Cloud Engineers* and *Support Engineers*. She fixes bugs on the Fredhopper Cloud Services and continuously improves the Fredhopper Cloud Services by either adding new features or improving existing ones

Support Engineer A *Support Engineer* receives and coordinates issues identified either by *Customers* or *Cloud Engineers*. She receives questions from *Customer*. She interacts with *Customer*, and either addresses them directly or informs the *Software Engineers*