A Model-Centric Approach to the Design of Resource-Aware Cloud Applications

Reiner Hähnle [1] and Einar Broch Johnsen [2]

[1] Technical University of Darmstadt, Germany haehnle@cs.tu-darmstadt.de

[2] University of Oslo, Norway einarj@ifi.uio.no

Capitalizing on the Cloud

The planet's data storage and processing are moving into the clouds. This has the potential to revolutionize how we interact with computers in the future. A cloud consists of virtual computers that can only be accessed remotely. It is not a physical computer, you do not necessarily know where it is, but you can use it to store and to process your data and you can access it at any time from your regular computer. If you still have an old-fashioned computer, that is. You might as well access your data or applications through your mobile device, for example while sitting on the bus.

Cloud-based data processing, or cloud computing, is more than just a convenient solution for individuals on the move. Even though privacy of data is still an issue, the cloud is already an economically attractive model for a start-up, an SME, or simply for a student who develops an app as a side project, due to an undeniable added value and compelling business drivers [1]. One

such driver is *elasticity*: businesses pay for computing resources when these are needed, and avoid major upfront investments for resource provisioning. Additional resources such as processing power or memory can be added to a virtual computer on the fly, or an additional virtual computer can be provided to the client application. Going beyond shared storage, the main potential in cloud computing lies in its scalable virtualized framework for data processing, which becomes a shared computing facility for multiple devices. If a service uses cloud-based processing, its capacity can be adjusted automatically when new users arrive. Another driver is *agility*: new services can be quickly and flexibly deployed on the market at limited cost, without initial investments in hardware.

Currently, there are more than 3.9 million jobs associated with cloud computing in the US, and more than 18 million jobs worldwide[1]. The EU believes[2] that cloud-based data processing will create another 2.5 million jobs and an annual value of 160 billion € in Europe by 2020. However, reliability and control of resources constitute significant barriers to the industrial adoption of cloud computing. To overcome these barriers and to execute control over virtualized resources on the cloud, client services need to become resource-aware.

---

1 Forbes, December 2014, http://www.forbes.com/sites/louiscolumbus/2014/12/12/where-cloud-computing-jobs-will-be-in-2015

2 Digital Agenda for Europe, http://ec.europa.eu/digital-agenda/en/european-cloud-computing-strategy

Challenges

Cloud computing is not merely a new technology for convenient and flexible data storage and implementation of services. Making full usage of the potential of virtualized computation requires nothing less than to rethink the way in which we design and develop software:

Empowering the Designer. The elasticity of software executed in the cloud means that designers have far reaching control over the resource parameters of the execution environment: the number and kind of processors, and the amount of memory, storage capacity, and bandwidth. These parameters can even be changed dynamically, at runtime. This means that the client of a cloud service can not merely deploy and run software, but can also fully control the trade-offs between the incurred cost of running the software and the delivered quality-of-service.

To realize these new possibilities, software in the cloud must be *designed for scalability*. Nowadays, software is often designed based on specific assumptions about deployment, including the size of data structures, the amount of RAM, and the number of processors. Rescaling may require extensive design changes, if scalability has not been taken into account from the start.

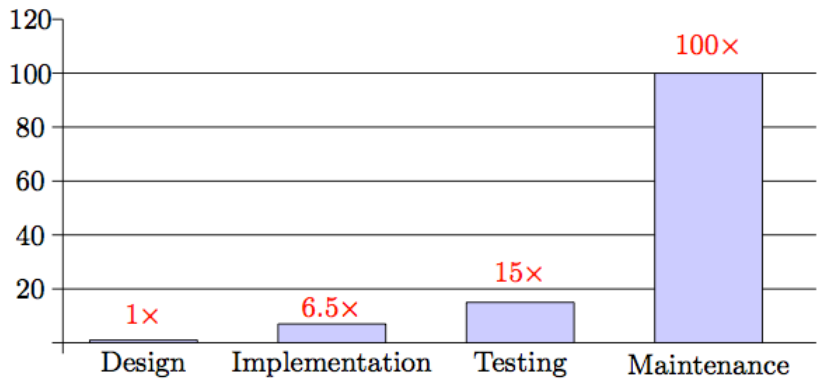

Fig. 1: Relative costs to fix software defects for static infrastructure (source: IBM Systems Sciences Institute). The columns indicate the phase of the software development at which the defect is found and fixed.

Deployment Aspects at Design Time. The impact of cloud computing on software design goes beyond scalability issues: traditionally, deployment is considered a late activity during software development. In the realm of cloud computing, this can be fatal. Consider the well-known cost increase for fixing defects during successive development phases [2]. IBM Systems Sciences Institute estimated that a defect which costs one unit to fix in design, costs 15 units to fix in testing (system/acceptance) and 100 units or more to fix in production (see Fig. 1). This estimation does not even consider the *impact cost* due to, for example, delayed time to market, lost revenue, lost customers, and bad public relations.

Now, these ratios are for *static* deployment. Considering the high additional complexity of resource management in virtualized environments, it is reasonable to expect even more significant differences; Fig. 2 conservatively suggests ratios for virtualized software in an *elastic* environment. This consideration makes it clear that it is essential to detect and fix deployment errors, for example, failure to meet a service level agreement (SLA), already *in the design phase*.

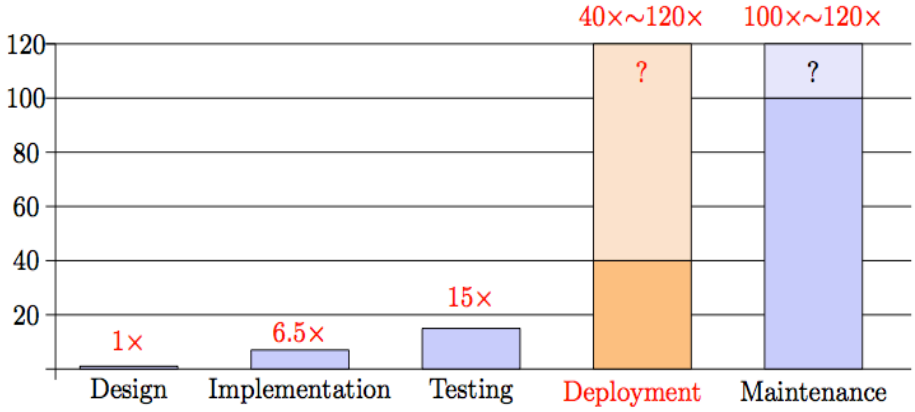

Fig. 2: Estimate of relative costs to fix software defects for virtualized systems with elasticity, from [3].

To make full usage of the opportunities of cloud computing, software development for the cloud demands a design methodology that (a) takes into account deployment modeling at *early* design stages and (b) permits the detection of *deployment errors* early and efficiently, helped by software tools such as simulators, test generators, and static analyzers.

Controlling Deployment In The Design Phase

Our analysis exhibits a *software engineering challenge*: how can the validation of deployment decisions be pushed up to the modeling phase of the software development chain without convoluting the design with deployment details?

When a service is developed today, the developers first design its functionality, then they determine which resources are needed for the service, and ultimately the provisioning of these resources is controlled through an SLA, see Fig. 3. The functionality is represented in the *client*

layer. The *provisioning layer* makes resources available to the client layer and determines available memory, processing power, and bandwidth. The SLA is a legal document that clarifies

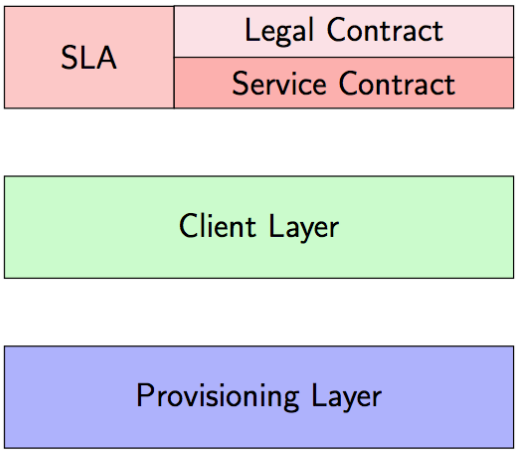

Fig. 3: Conceptual parts of a deployed cloud service.

what resources the provisioning layer should make available to the client service, what they cost, and states penalties for breach of agreement. A typical SLA covers two aspects: (i) a *legal contract* stating the mutual obligations and the consequences in case of a breach and (ii) the technical parameters and cost figures of the offered services, which we call the *service contract*.

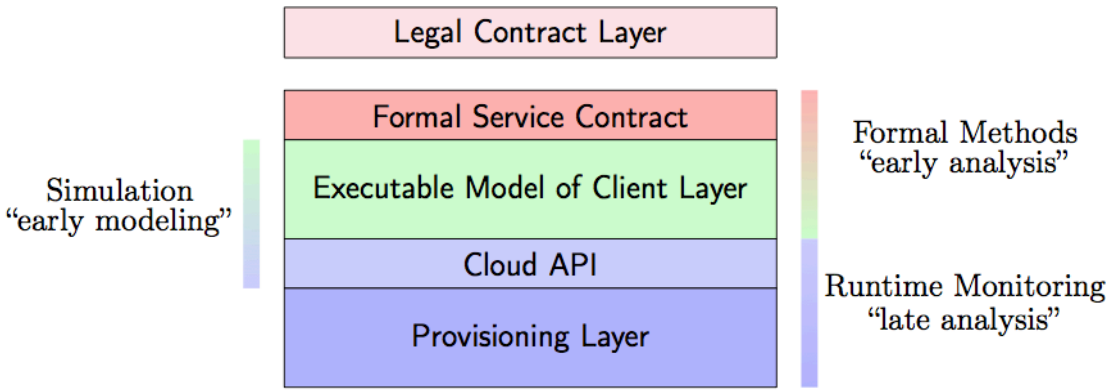

Fig. 4: Making services resource-aware.

Today the different parts of a deployed cloud service live in separate worlds, but we need to connect them. In a first step the provisioning layer is made available to the client, so that the client can observe and modify resource parameters. We call this the *Cloud API*. This is *not* the same as the APIs that cloud environments currently provide to their clients: our ultimate goal is to move deployment decisions into the *design phase*. We advocate that client behavior is represented by an *abstract behavioral model* of the client application.

How can the vision just sketched possibly be realized? In the remaining paper we will try to give one answer. We report on an effort towards a model-centric approach to the design of resource-aware cloud applications that we undertook as part of a European project (www.envisage-project.eu). It is based on a modeling language called ABS.

ABS: Modeling Support for Designing Resource-Aware Applications

The modeling language ABS (for: Abstract Behavioral Specification) is a concurrent, executable modeling language [4] with a formal semantics. It has been designed for being easy to learn and to use: ABS comprises data type, functional, imperative, and OO abstractions in an orthogonal, modular manner and with popular notational conventions (Java-ish syntax).

Before we sketch the main characteristics of ABS, let us explain why it is particularly well-suited as a basis for model-centric design of resource-aware cloud applications. From Fig. 4 three main requirements emerge:

1. the need for an abstract interface (the "Cloud API") to the provisioning layer;
2. the need to connect SLAs to the client layer, where the key point is to formulate service contract aspects of SLAs relative to a semantics expressed as a formal specification;
3. finally, the modeling framework must admit various static and dynamic analyses.

ABS fits this bill perfectly: it features a reflection layer that allows to describe timed resources of various kinds in an abstract manner; ABS has a formal semantics and comes with a correctness notion relative to formal behavioral specifications; most importantly, ABS was designed for being formally analyzable and comes with a wide range of scalable, inter-procedural "early" as well as "late" analyses [6], including resource consumption, deadlock detection, test case generation, runtime monitoring, and even functional verification.

The concurrency model of ABS is asynchronous and based on cooperative scheduling with futures: to call a method `m` with parameter `p` on an object `o` asynchronously we use the syntax "`o!m(p)`". This creates a new task in `o`'s processor to be scheduled at some future time point while the caller simply continues to execute and no preemption takes place. This approach to concurrency excludes one of the most problematic aspects of parallel code: data races. But one must address two issues: first, how to synchronize with the caller upon `m`'s termination and return its result; second, how to enable multitasking. In ABS it is possible to assign the result of asynchronous method calls to variables that have a future type. These provide a handle to retrieve the result once it is ready. Futures are in ABS are closely related to scheduling: `await` is a statement with one argument which releases control over the processor and enables rescheduling as soon as its argument evaluates to true. If the argument has the form `f?`, where `f`

has a future type, this amounts to giving up control until the task associated with `f` is ready. Afterwards its result can be safely retrieved with the expression `f.get`.

Resource Modeling with ABS

ABS supports the modeling of deployment decisions and resource awareness [7]. Let us illustrate the modeling of resource-aware systems in ABS by considering a virtualized service in which user requests trigger computational tasks whose *resource cost* varies with the size of input data. The system consists of a service endpoint, worker objects, a database, a load balancer and a resource manager which controls scaling of the pool of worker objects (see Fig. 5). The service endpoint can be modeled in ABS by a class which implements a service endpoint interface `SE` with the method `invokeService(Rat cost)`, where `cost` is an abstraction over the input data.

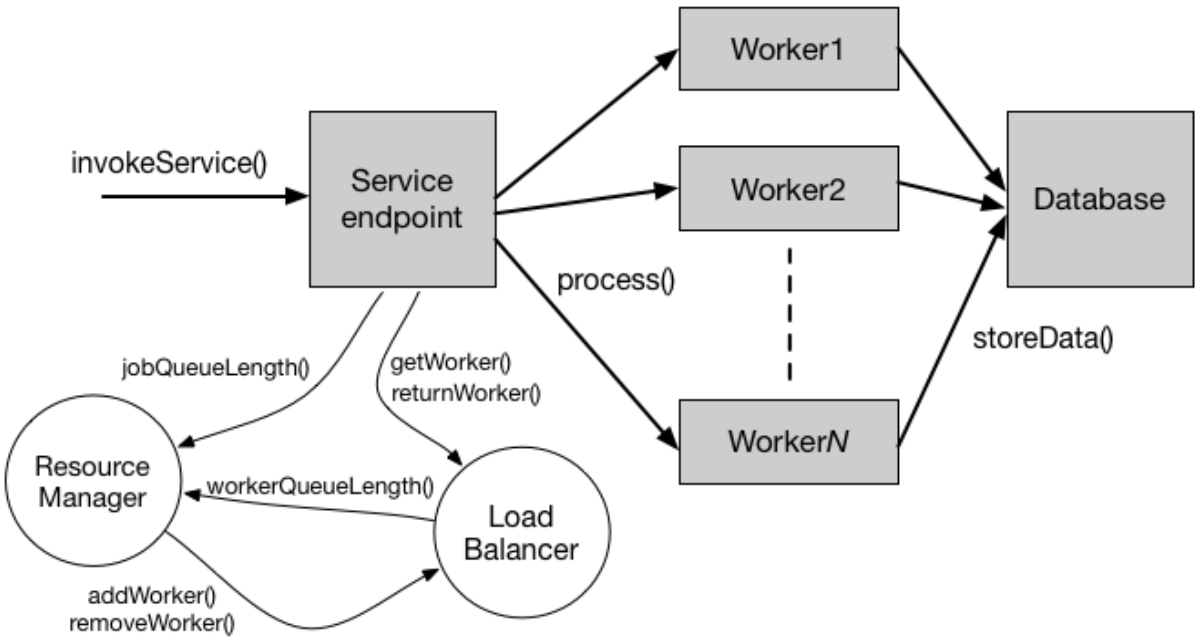

Fig. 5: A resource-aware service modeled in ABS.

When the service endpoint receives a request to invoke the service, it obtains a `Worker` object from a `LoadBalancer` and asks that `Worker` to process the request.

```
class ServiceEndpoint(Duration responseTime, ResourceManager rm,
                Database db, LoadBalancer lb) implements SE {
    Int lengthOfQueue = 0;

    Fut<Bool> invokeService(Rat cost){ Fut<Bool> success;
     lengthOfQueue = lengthOfQueue + 1;
     Worker w = await lb!getWorker();
     [Deadline: responseTime] success = w!process(cost);
     lengthOfQueue = lengthOfQueue - 1;
     await success?;
     Bool s = success.get;
     return success;
    }
    Unit run(){
     await duration(1,1);
     rm!jobQueueLength(lengthOfQueue);
     this!run();
    }
}
```

Here, the return type of `invokeService` is a future which will eventually contain the value `True` if the service invocation was processed within the specified response time and `False` if that deadline was violated. The annotation `[Deadline: responseTime]` is used to specify a local deadline to the method activation of `process` in class `ServiceEndpoint`. That class has `responseTime` as a parameter for processing service requests. The method `run` makes objects of an ABS class active; i.e., this method is automatically activated when the objects are created. Here, the `run` method suspends execution for a given time interval, reports on the size of its queue length (stored in the variable `lengthOfQueue`) to a `ResourceManager` and then calls recursively itself.

Let the class `WorkerObject` implement the `Worker` interface with a method `process.`

```
class WorkerObject(Database db) implements Worker {

    Bool process(Rat taskCost) {
     [Cost: taskCost] skip;
     [Deadline: deadline()] Fut<Bool> f = db!accessData();
     f.get;
     return (deadline() > 0);
    }

    DC getDC(){ return thisDC();}
}
```

We see how the non-functional specification of the computation is integrated into the ABS model: The resource cost of the local computation is modeled by a **`skip`** with an annotation `[Cost: taskCost]`, where `taskCost` represents the cost of processing the input data. The executing method passes on the remainder of its own deadline to the method call to the database `db` with the annotation `[Deadline: deadline()]`.

The `Worker` objects are deployed on *deployment components*. This is a modeling concept in ABS for locations where execution takes place (e.g., virtual machines or containers). An object in ABS can find its own location by the expression `thisDC()`.

The service endpoint interacts with a `LoadBalancer` to acquire a `Worker` by the method `getWorker` and to return a `Worker` by the corresponding method `releaseWorker`. A `LoadBalancer` can, for example, be implemented by a round-robin load balancer, as shown below. Similar to the `run` method of `ServiceEndpoint`, the `run` method here reports on the total number of deployed `Worker` objects to the `ResourceManager` befor calling itself.

```
class RoundRobinLoadBalancer(ResourceManager rm, CloudProvider
cloud, Int nResources, Database db) implements LoadBalancer {
    List<Worker> available = Nil;
    Int numberOfMachines = 0;

    Unit run(){ ... }
    Worker getWorker(){ ... }

    Unit releaseWorker(Worker w){
     available = appendright(available,w);
    }

    Unit addWorker(){
     DC machine = await cloud!launchInstance(
       {CPU:nResources, PaymentInterval:1, CostPerInterval:1});
     [DC: machine] Worker w = new WorkerObject(db);
     available = appendright(available,w);
     numberOfMachines = numberOfMachines + 1;
    }

    Unit removeWorker(){
     if (available != Nil) {
          Worker w;
          w = head(available);
          available = tail(available);
          DC machine = await w!getDC();
          cloud!releaseInstance(machine); }
    }
}
```

Two methods control the usage of resources by the service. The method `addWorker` interacts with the `CloudProvider` to deploy a new `Worker` on a deployment component with CPU processing capacity `nResources` and with billing for each time interval at cost 1. Similarly, `removeWorker` removes a `Worker` and releases the deployment component of that `Worker`.

Observe that the processing time of a task depends on its cost and on the processing capacity of the deployment component where it is deployed. The overall cost of running an ABS model of a service depends on the cost of the billing period and on the number of deployment components

which have been actively deployed per billing period. The `CloudProvider` interface is part of the ABS library and provides a highly configurable API for launching, acquiring, and releasing deployment components modeling virtual resources. It also provides cost accounting, for example, method `getAccumulatedCost` returns the accumulated cost at any time during execution. Hence it realizes the *Cloud API* stipulated above.

The class `RM` implements the `ResourceManager` interface, which defines a policy for scaling the service. In our model, this is done at regular intervals by comparing the ratio between the lengths of the job and worker queues to the thresholds `upper` and `lower`. More advanced policies could for example consider the number of deadline violations in a sliding window (such as the last 24 hours) or other aspects of an SLA.

```
class RM(Rat lower, Rat upper) implements ResourceManager {
    Int jobQueue = 0; Int workerQueue = 0; LBManager lb;

    Unit run(){
     await duration(1,1);
     // check conditions for scaling up
     if ((jobQueue / workerQueue) > upper) { lb!addWorker(); }
     // check conditions for scaling down
     if ((jobQueue / workerQueue) < lower) { lb!removeWorker();
}
     this!run();
    }

    Unit register(LBManager r){lb = r;}
    Unit jobQueueLength(Int n){jobQueue=n;}
    Unit workerQueueLength(Int n){workerQueue=workerQueue+1;}
}
```

In this example, we have specified the cost directly as an argument to the `process` method. For a detailed ABS model which models the actual execution of the method, automated tools help the modeler to analyze the model and to extract cost annotations.

Opportunities

Making deployment decisions at design-time moves control from the provisioning layer to the client layer. The client service becomes resource-aware. This provides a number of attractive opportunities:

Fine-grained provisioning. Business models for resource provisioning on the cloud are becoming more and more fine-grained, similar to those we know from other metered industry sectors such as telephony or electricity. It is an increasingly complex decision to select the best model for your software. So far this complexity is to the advantage of *resource providers*. Design-time analysis and comparison of deployment decisions allow an application to be deployed according to the optimal payment model for the *end-users*. Cloud customers can take advantage of fine-grained provisioning schemas such as spot price.

Tighter provisioning. Better profiles of the resource needs of the client layer help cloud providers to avoid over-provisioning to meet their SLAs. Better usage of the resources means that more clients can be served with the same amount of hardware in the data center, without violating SLAs and incurring penalties.

Application-specific resource control. Design-time analysis of scalability enables the client layer to make better use of the elasticity offered by the cloud, to know beforehand at which load thresholds it is necessary to scale up the deployment to avoid breaking SLAs and disappointing the expectations of the end-users.

Application-controlled elasticity. We envisage autonomous, resource-aware services that run their own resource management strategy. Such a service will monitor the load on its virtual machine instances as well as the end-user traffic, and make its own decisions about the trade-offs between the delivered quality of service and the incurred cost. The service interacts with the provisioning layer through an API to dynamically scale up or down. The service may even request or bid for virtual machine instances with given profiles on the virtual resource market place of the future!

Summary

We argued that the efficiency and performance of cloud-based services can be boosted by moving deployment decisions up the development chain. Resource-aware services give the client better control of resource usage, to meet SLAs at lower cost. We identify formal methods, executable models, and deployment modeling as the ingredients that can make this vision happen. A realization of our ideas has been implemented on the basis of the modeling language ABS. We illustrated our ideas with a concrete case study around a representative service provider scenario that consists of fully executable ABS code.

Acknowledgments. This work has been partially supported by EU project FP7-610582 Envisage: Engineering Virtualized Services.

References

[1] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Comp. Sys.*, 25(6):599–616, 2009.

[2] B. W. Boehm and P. N. Papaccio. Understanding and controlling software costs. *IEEE Trans. SW Eng.*, 14(10):1462–1477, 1988.

[3] E. Albert, F. de Boer, R. Hähnle, E. B. Johnsen, and C. Laneve. Engineering virtualized services. In M. A. Babar and M. Dumas, editors, *2nd Nordic Symp. Cloud Computing & Internet Technologies*, pages 59–63. ACM, 2013.

[4] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. de Boer, and M. M. Bonsangue, editors, *Proc. 9th Intl. Symp. on Formal Methods for Components and Objects*, volume 6957 of LNCS, pages 142–164. Springer, 2011.

[5] E. Albert, F. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, and P. Y. H. Wong. Formal modeling of resource management for cloud architectures: An industrial case study using Real-Time ABS. *J. of Service-Oriented Computing and Applications*, 8(4):323-339, 2014

[6] R. Bubel, A. Flores-Montoya, R. Hähnle. Analysis of Executable Software Models. Formal Methods for Executable Software Models. 14th Intl. School on Formal Methods for the Design

of Computer, Communication, and Software Systems, SFM, Advanced Lectures. LNCS 8483, pages 1-25, Springer 2014.

[7] E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Integrating Deployment Architecture and Resource Consumption in Timed Object-Oriented Models. *Journal of Logical and Algebraic Methods in Programming,* 84(1): 67-91, 2015.