Analysis of SLA Compliance in the Cloud: An Automated, Model-based Approach *

Frank S. de Boer[1], Elena Giachino[2], Stijn de Gouw[1,3], Reiner Hähnle[4,5], Einar Broch Johnsen[5], Cosimo Laneve[2], Ka I Pun[5], and Gianluigi Zavattaro[2]

[1] CWI Amsterdam, The Netherlands
[2] University of Bologna, Italy & INRIA, France
[3] SDL Fredhopper, Amsterdam, The Netherlands
[4] TU Darmstadt, Germany
[5] University of Oslo, Norway

August 2016

Summary. This white paper explains how formal models combined with static analysis tools and generated runtime monitors enable SLA-aware deployment of services on the cloud. The proposed approach fits well with a DevOps methodology.

1 Model-Centric Analysis of SLA Compliance

Every customer wants to be sure about the quality of their purchase. In the cloud world, this quality assurance includes guarantees on service *performance*[1].

Service Level Agreements (SLAs) are legal documents, signed and agreed upon by cloud service providers and their customers, where the specification of the agreed quality of service is written down. An SLA violation will result in penalties and possibly in a loss of money, clients, and credibility. Even though the stakes are high, there are only few tools with limited capabilities available to check the compliance of cloud services with SLAs. Why is it so difficult to provide tool support for SLA compliance checking?

*Partially funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services (`http://www.envisage-project.eu`).

[1] Other important concerns include security aspects, support, data and data protection [1].

For a start, a large number of complex and challenging questions arise: How to describe the performance of a service? How many resources should be assigned to a particular service? How to react optimally at runtime to take advantage of the elasticity of the cloud? How to estimate the future behavior of a service and adjust the resource configuration accordingly?

These are challenging issues! It is beyond current technology to address them in a general way for any given SLA and any given software. To develop tools for SLA compliance checking, we believe it is essential to work at the level of *models*: it is important to describe and analyze SLAs in a way which is independent of the concrete technology offered by the cloud service provider. Shifting to the modeling level increases the level of abstraction, reduces complexity, and removes dependency on a specific runtime environment.

The importance of models applies to SLAs as well as to software: a model-centric approach allows us to create a formal representation of the essential aspects of an SLA. At the same time, software deployed on the cloud can be represented as an executable model annotated with parametric expressions for the use of resources. Combining these models, we may now employ techniques with a formal basis, such as state-of-art static software analysis, to provide proven guarantees on service performance, thereby vastly raising the degree of automation.

Benefits of model-centric, tool-based SLA analysis

An effective solution to SLA design and compliance must coordinate all phases of service provisioning. It must encompass

- assistance in the configuration of SLA metric bounds and resource deployment to be compliant with a given service deployed on the cloud
- the automatic monitoring of the service at runtime
- prompt reaction to a SLA violation and assistance in its resolution
- deployment: significant simplification and increased automation

2 What to be Measured? What to be Verified?

Service performance metrics are used to quantitatively and periodically measure and assess the performance level of a service. Typical metrics fall into one of the following categories:

Availability is the property of a service to be accessible and usable on demand. It includes (*i*) the *level of uptime*, namely the percentage of time a service is up within

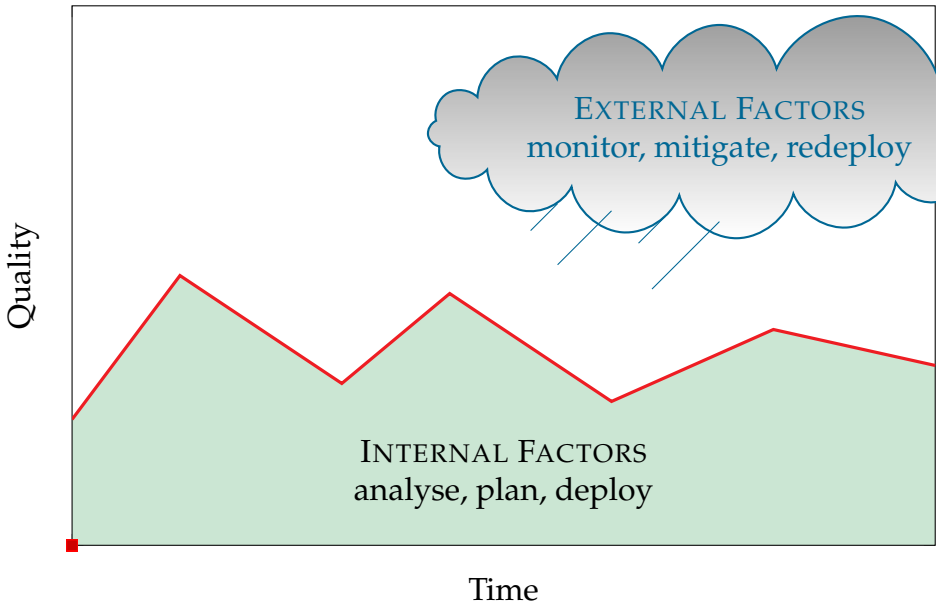

Figure 1: Service Quality Level

a defined period; (*ii*) the *percentage of successful requests*, namely the number of requests processed without an error over the total number of submitted requests; (*iii*) the *percentage of timely service provisioning requests*, that is the number of service provisioning requests completed within a defined time period over the total number of service provisioning requests.

Response time is the time period between a client request event and a service response event. The service metrics used to constrain the response time may return either an *average time* or a *maximum time*, given a particular form of request.

Capacity is the maximum amount of a resource used by a service. It includes the *service throughput metric*, namely the minimum number of requests that can be processed by a service in a stated time period.

Several factors contribute to the quality level of a service, as depicted in Figure 1. They can be classified as *internal*, such as the available resources or the computational complexity, and *external*. The latter are outside the direct control of the stakeholders and include, for example, network availability or the number of accesses/requests.

Internal factors can principally be controlled (and, if needed, modified) at deployment time with techniques that either directly verify the code (static analysis) or with the help of an underlying mathematical model (model checking, simulation, etc.). Whenever the service implementation does not comply with the metric, the designer makes code modifications that eventually lead to compliance. A typical example of this is the analysis of the *resource capacity* of a service, which measures how much a critical resource is used. For instance, a static analysis technique can determine an upper bound

of the number of virtual machines needed by a service; if a service is deployed on an insufficient number of machines, then its response time increases, or it even becomes unavailable.

External factors can not be controlled or analyzed in advance, but they can be supervised by monitoring code that runs independently of the service implementation. Monitoring is always needed, as there are (performance) metrics that are affected by external factors, that cannot be statically verified, for example, hardware failures. In this case, neither the service implementation nor the resource configuration is at fault. However, a runtime monitor can still be helpful, for example, when it triggers a dynamic resource re-allocation that compensates for a faulty component.

3 The Work Flow

In Section 1 we argued that a model-centric representation of SLAs and services constitutes the basis for advanced tool support for cloud service configuration and deployment. In Section 2 we explained how the service quality is influenced by internal factors, to be addressed by compliance checking of service implementations against SLAs; and by external factors, to be mitigated with the help of runtime monitors. In Figure 2 we illustrate a work flow that realizes such a scenario.

Static (deployment time) techniques play an important role in generating metrics and monitors that are used in run-time techniques (e.g., monitoring). *Feedback loops*, denoted with a dashed line, to a previous phase of the analysis, represent modifications to the system that ensure, for example, compliance.

Negotiation phase. This phase includes everything that might happen before signing the SLA. At this stage the SLA metrics are set, such that the *service model* can be verified against them. The SLA is written in a machine-readable standard format (ISO 19086-2) and mathematical *metric functions*, expressing upper bounds on the possible measurements, are extracted from it. The initial *resource configuration* is defined with the resource types that are allocated for the service. Such initial resource configurations can be specified manually, or they can be computed automatically by a solver that returns an optimal distribution of resources to objects, given the knowledge of the initial objects to be deployed, their required computing resources, and the resource costs. At the same time, an executable service model is extracted from the components of the actual service implementation. The configuration of the actual *service implementation* is generated from the service model and *deployed* according to the resource configuration.

A suite of deploy-time tools now takes these three inputs (SLA metric functions, Service model, and Resource configuration) and produces responses as output to form a feasibility assessment. The tools can verify properties such as: upper bounds of the resource consumption (bandwidth, virtual machines, memory allocation, CPU processing cycles), liveness (deadlock-freedom), safety (functionality). If the tools report that a service model violates an SLA constraint, then either the constraint can be relaxed or

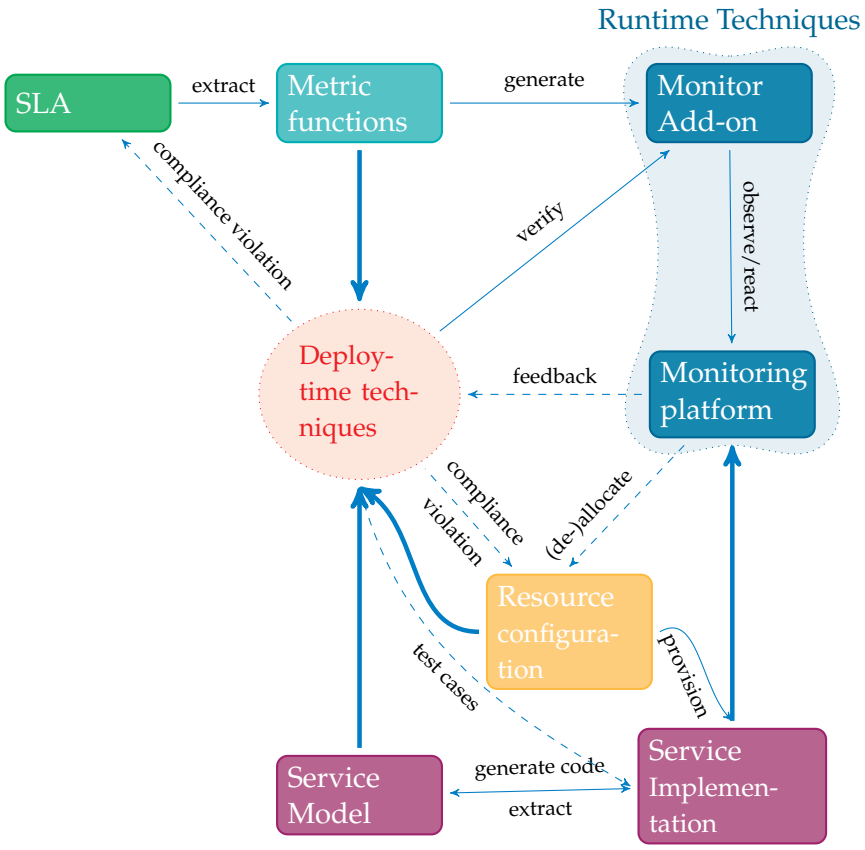

Figure 2: Work flow of service configuration and deployment

the resource allocation can be suitably enlarged during the negotiation phase (with a possible charge for the client). The tools can also produce test cases that can be used to validate that the service model captured the implementation.

Thus, the feedback provided by the deploy-time analysis guides the negotiation phase by discarding resource configurations, that hinder the ability of the service to meet the SLA. Feedback may also be used to select a better metric bound, given the available resources. A third feedback loop may connect back to the program and allow changes in the code to be applied, so as to better adapt to the given SLA and available resources.

Once the configuration is approved by the deploy-time tools—guaranteeing that, *in the absence of external factors*, the service implementation and the resource configuration comply to the SLA—the next phase can start.

Observation phase. The SLA is signed and the service is up and running. At this stage, factors under external control, such as the network infrastructure, may come into play and affect the behavior of the service in ways which could not have been predicted statically. To supervise the service metrics we use a monitoring system, namely code

external to the service that continuously monitors its execution.

The code of the monitors is automatically generated (or configured), starting from the specific metric functions they are intended to monitor. Static techniques may be used at this stage for proving the correctness of the generated code, i.e., that the monitors are observing the right property. Moreover, static techniques may be performed again at runtime, periodically, on the service model, to estimate the future behavior of the service in a next time window. Feedback from the monitoring system can significantly augment the precision of the analysis.

Reaction phase. Monitoring systems allow service providers to report violations of the agreed SLA. However, the ultimate goal for a provider is to *maintain* the resource configuration so that SLA violations remain under a given threshold while minimizing the cost of the running system. The first objective can be achieved by adding resources to the service (for instance, adding more CPUs).

The observation phase takes measurements on services. Subsequently, if an SLA mismatch is observed, in the reaction phase, the number of allocated resources is increased or decreased accordingly. As was done for the initial configuration, also in this phase the modification of the resources assigned to objects can be done either manually or automatically. A solver computes how new resources should be distributed when new objects are deployed, or how old objects and resources which are no longer necessary should be un-deployed, given the knowledge of the current resource configuration and the new requirements indicated by the monitoring framework. Fully automatic dynamic elasticity can be obtained thanks to the combined use of the monitoring framework and the external deployment solver.

4 ABS: A Modeling Language and Tool Suite for Systems Deployed on the Cloud

ABS is a modeling language which can be used to realize model-centric analysis of SLA compliance according to the workflow outlined in Section 3. The box on top of the next page gives a very concise overview of ABS. For more information, see `http://www.abs-models.org`.

ABS is a modeling language that was designed for analyzability. One of its strengths is the availability of a large portfolio of analysis and deployment tools, see the box at the bottom of the next page.

The ABS modeling language (☞ `http://www.abs-models.org`)

ABS is a language for Abstract Behavioral Specification, which combines implementation-level specifications with verifiability, high-level design with executability, and formal semantics with practical usability. ABS is a concurrent, object-oriented modeling language built around a simple functional language with user-defined algebraic datatypes. Models are easy to understand and written in a familiar, Java-like syntax. ABS supports the modeling and analysis of deployment decisions. Both the resource requirements and timing properties of models can be expressed and analyzed, which makes it easy to compare deployment decisions at the level of models.

The ABS tool suite (☞ `http://abs-models.org/abs-tools`)

Simulation tool allows rapid model exploration and visualization

Deadlock analysis tool automatically checks that the model is deadlock free, focusing on the communication protocols in the model

Systematic testing tool provides a technique to eliminate redundant test cases for the concurrent execution of ABS models.

Test case generation tool for the automatic generation of test cases for concurrent objects in ABS

Termination and resource consumption tool automatically infers cost bounds for selected parts of the model for, e.g., execution cost or transmission data size

ABS Smart Deployer finds the optimal deployment of components on virtual machines, given a user specification of how components should be connected and their resource requirements

Code generation tools enable rapid prototyping on real machines and integration with other programs, using Haskell or a Java library

Formal verification tool supports deductive analysis of behavioral properties, including communication traces

Monitoring framework for SLA metrics is used to automatically configure correct monitors for the deployed system and monitor the system at a high level, according to the SLA

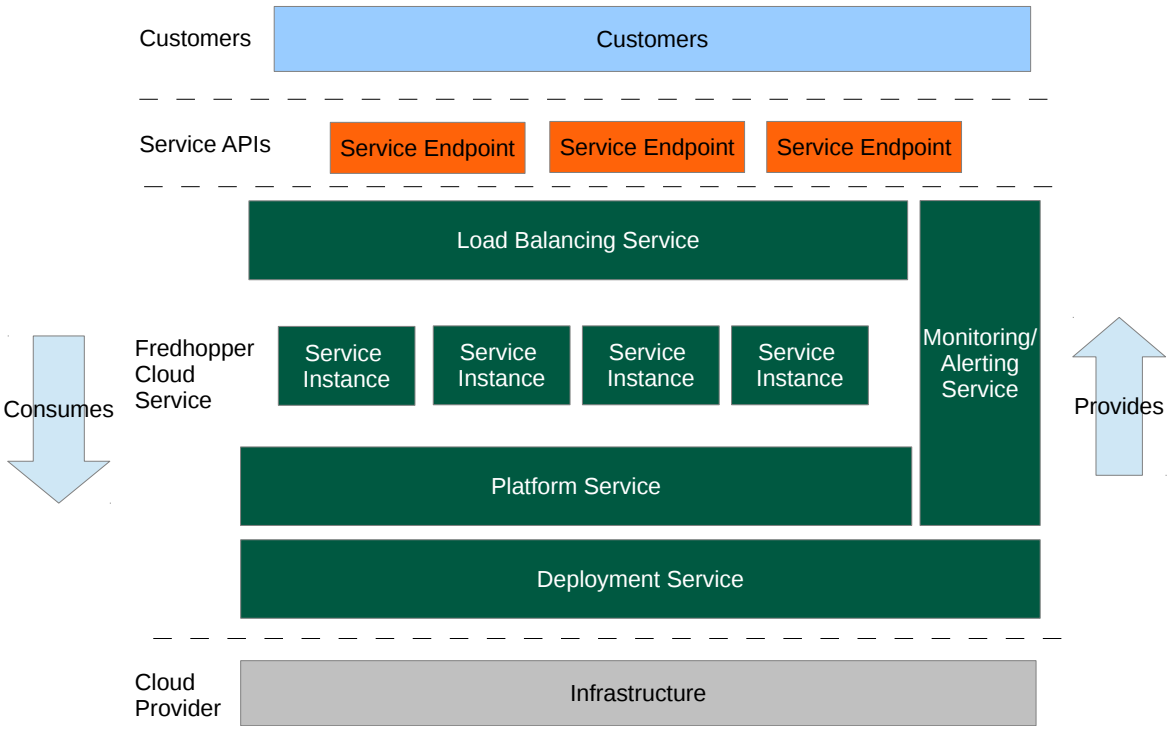

Figure 3: The architecture of the Fredhopper Cloud Services

5 Example

Fredhopper[2] provides 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). Fredhopper Cloud Services drives over 350 global retailers with more than 16 billion in online sales every year. A customer (service consumer) of Fredhopper is a web shop, and an end user is a visitor to the web shop.

The services offered by Fredhopper are exposed at endpoints. In practice, these services are implemented to be RESTful and accept connections over HTTP. Software services are deployed as *service instances*. Each instance offers the same service and is exposed via Load Balancer endpoints that distribute requests over the service instances. Figure 3 shows a block diagram of the Fredhopper Cloud Services.

The number of requests can vary greatly over time, and typically depends on several factors. For instance, the time of the day in the time zone where most of the end users are located, plays an important role. Typical lows in demand are observed between 2 am and 5 am. Figure 4 shows a visualization of monitored data in Grafana (the visualization framework used by ABS).

[2] `http://www.sdl.com/cxc/digital-experience/ecommerce-optimization/fredhopper.html`

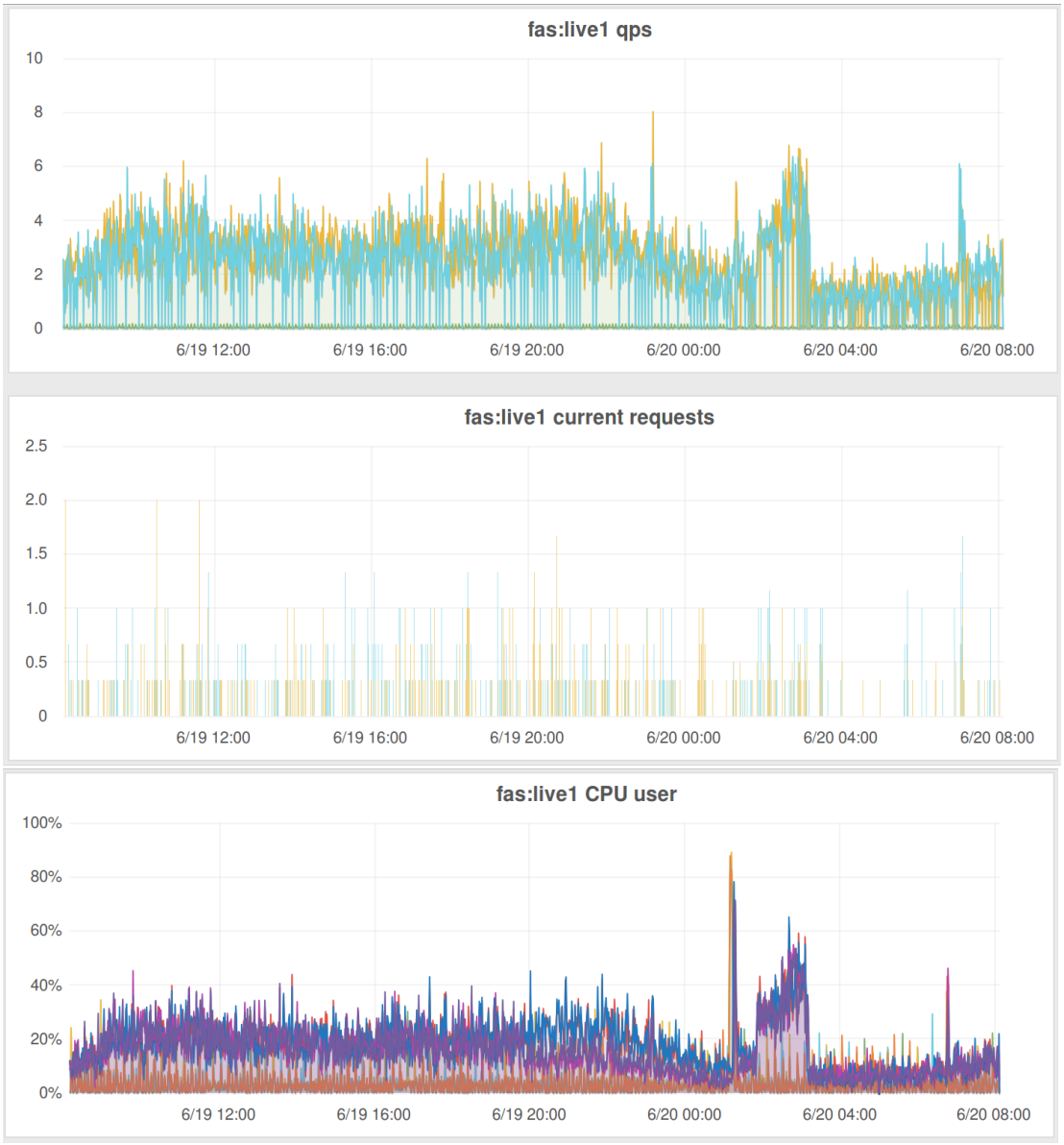

Figure 4: Visualization of metrics

SLA. Peaks in Fredhopper Cloud Services typically occur during promotions of the shop or around Christmas. To ensure a high quality of service, web shops negotiate an aggressive Service Level Agreement (SLA) with Fredhopper. QoS attributes of interest include query latency (*response time*) and throughput (*queries per second*). The SLA negotiated with a customer could express, e.g., *service degradation* requirements as follows:

> *"Services must maintain 100 queries per second with less than 200 milliseconds of response time over 99.5% of the service uptime, and 99.9% with less than 500 milliseconds."*

An SLA specifies properties of service metric functions. 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.

Formalizing the service metric function. In ABS we formalize a service metric function as a mapping of traces of events to values by a certain kind of grammar. These events indicate client interactions with an endpoint of an exposed service API. The values correspond to different levels of the provided quality of service. The grammars are a user-friendly formalism and are particularly well suited for the specification of both data- and protocol-oriented properties of event traces. It is not necessary to know technical details about them.

Suppose we wish to formalize our service degradation metric. We identify the processing of a client request that interacts with an endpoint of an exposed service API by an event

```
invoke(Time t, Rat procTime)
```

This event indicates that the request has been issued at time `t` and that the request has the rational `procTime` as its processing time. In our formalization, a service view identifies all the events which are relevant for a particular service metric and associates a name to each such event. A view which simply identifies the invoke event as the only relevant event and associates the name "query" with this event, is expressed as follows:

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

Figure 5 presents the grammar which computes as the main metric the percentage of slow queries "degradation". The string "fas.live.200ms" gives the name of the metric. The parameters of the invoke event, e.g., `procTime` , are directly referred to in the grammar by their name and are used to compute "degradation". The grammar further makes use of the auxiliary concepts "cnt" and "slowCnt".

The resource-aware service. We model in ABS the various services shown in Figure 3 at an *abstract level*. By way of example we show the model of a Service Instance (Figure 6) and the Load Balancing Service (Figure 7). The load balancer distributes requests by means of a round robin policy and forwards them to the service instances. The real service instances process the requests and return a response, e.g., a list of products in the case of Fredhopper Cloud Services. The ABS model abstracts from a detailed implementation and focuses on execution cost by means of the statement `[Cost: cost] log = log + 1`. The annotation `[Cost: cost]` is a measure of the estimated number of instructions. An initial value for it can be obtained by using the SACO

```
Pair<String, Rat> degradation = Pair("fas.live.200ms", 0);
Int cnt = 0;
Int slowCnt = 0;

S ::= query
     { cnt = cnt+1;
       slowCnt = slowCnt + case(procTime > 200) { True => 1;
                                                  False => 0;};
       degradation = Pair("fas.live.200ms", slowCnt/cnt);
     }
     S
```

Figure 5: Grammar for Service Degradation

tool for cost analysis of models in the ABS tool suite, or by averaging execution times for existing code.

```
class ServiceImpl (...) implements Service {
  ...
  Response invoke (Request request) {
    assert state == RUNNING;
    Int cost = cost(request);
    Int time = currentms();
    [Cost: cost] log = log + 1;
    time = currentms() - time;
    latency = max(latency, time);
    return success();
  }
}
```

Figure 6: Service Instance

Negotiation phase. Before we can accept a proposed SLA we need to determine whether we can meet it at with appropriate expense by deploying a number of `ServiceImpl` instances. We assume a setting where `ServiceImpl` instances run on machines with an allocated *capacity* of `K` execution resources (CPU execution capacity, also called ECU).

Static analysis with SACO yields `cost/K` as the total time required by the `invoke` method to reply to a single query, so we obtain `(cost/K)`$\leq$ `0.2` as a first bound from the SLA. In order to meet the *service degradation* requirement expressed in the SLA above, we need to determine the minimum number of resources in a configuration that complies with the SLA. For simplicity, we here assume a uniform arrival time for the requests, ignore the overhead of load balancing and distribution, and let `n` be the number of machines with `k` execution resources that we need. In this case, we know that

```
class LoadBalancerEndPointImpl implements LoadBalancerEndPoint {
  Int log = 0;
  State state = STOP;
  List<Service> services = Nil;
  List<Service> current = Nil;
  ...
  Response invoke (Request request) {
    log = log + 1;
    assert state == RUNNING;
    if (current=Nil) { current = services; }
    EndPoint p = head(current);
    current = tail(current);
    return await p!invoke(request);
  }
  ...
}
```

Figure 7: Load Balancing Service Endpoint

$(100 \times cost/n \times K) \leq 20$, and we obtain $(5 \times cost/K) \leq n$. For more complex scenarios (especially involving sub-services and synchronization), the ABS tool suite comes in handy to help calculating the required number of machines.

This ignores the actual arrival time of requests as well as any *external* factors (see Figure 1) which may come into play and disrupt service execution. To ensure compliance to the service metrics under non-ideal conditions, we use code, external to the service, that continuously monitors it.

The observation phase. The observation phase in our framework consists of computing the value of the service metric function as specified by the grammar in Figure 5 from a given event trace. This involves parsing the event trace according to the grammar. From the grammar we automatically synthesize an ABS implementation of the corresponding parser. The use of grammars allows to build on well-established and widely known parsing technology with optimal performance. Observations can also come from external systems which interact with the model using an API over HTTP.

Given an ABS model of the system, we can now replay a *real-world log* using this API, which generates corresponding invoke events for the model according to the specified timings in the logfile (see Figure 8). The resulting trace of invoke events is then parsed according to the grammar in order to compute the "degradation" service metrics.

Reaction phase. Figure 9 shows a monitor corresponding to the above grammar for service degradation. Here `metricHist` contains the time-stamped history of metric values which is provided by the general monitoring framework. The monitoring framework further integrates a powerful tool (the ABS Smart Deployer) for the automated

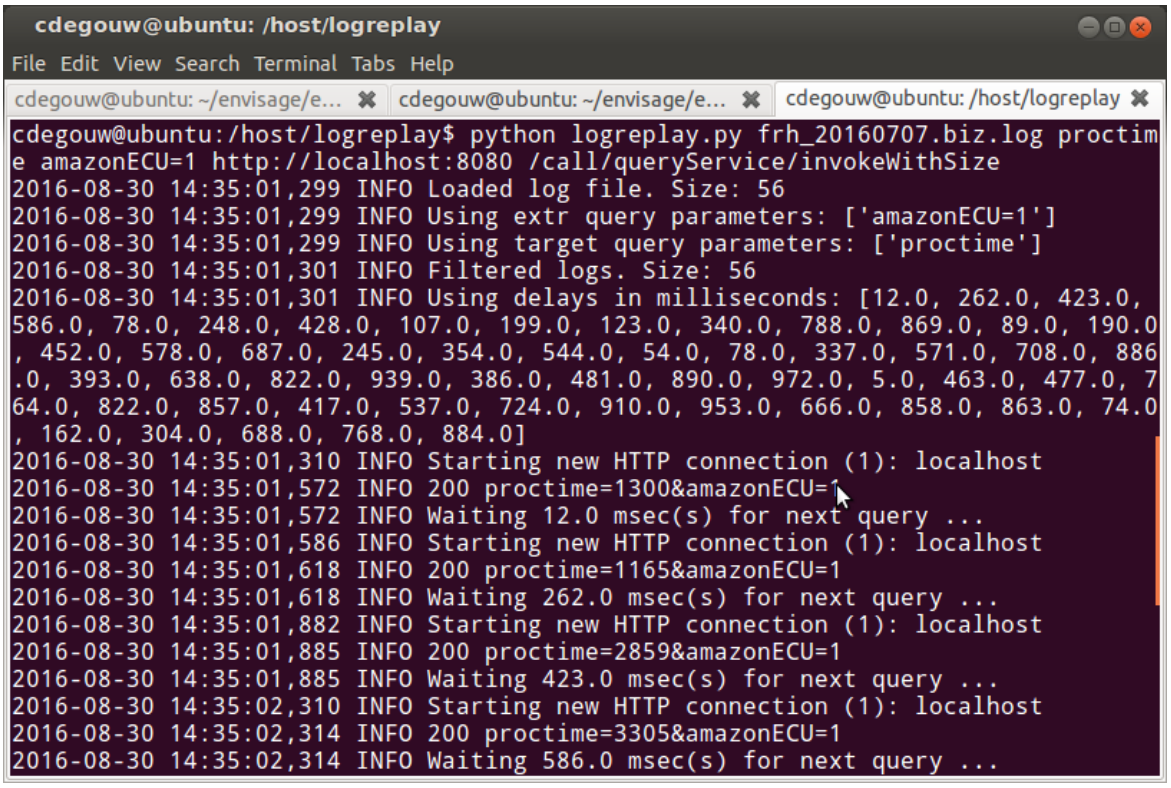

Figure 8: Log replay

deployment of new service instances which is based on high-level requirements of deployment configurations. A solver synthesizes an ABS class implementing `deployerIF` with appropriate scaling actions.

```
Unit monitor (DeployerIF deployer) {
  Rat degradation = head(metricHist);
  if (degradation > 5/1000) {
     deployer.scaleUp();
  } else if (degradation < 1/1000) {
     deployer.scaleDown();
  }
}
```

Figure 9: Monitor for Service Degradation

The above ABS monitor can react to these metrics by calling to the deployer to scale up or down the service instances. Running a monitor can be expensive and great care must be taken that it does not itself degrade performance below the level stipulated in the SLA. Static analysis and simulation of the ABS model *together* with the monitor allows to analyze the effect of the monitor on the SLA *before* the system is deployed. ABS allows monitors to be deployed asynchronously and decoupled.

Feedback

Would you spend

3 minutes

to answer

3 simple questions

about this white paper? Your feedback is essential, please reply at

`https://goo.gl/forms/vMNzDQX6cvLWPE292`

References

[1] Cloud Select Industry Group. Cloud service level agreement standardisation guidelines, June 2014. Developed as part of the Commissions European Cloud Strategy. Available at `http://ec.europa.eu/information_society/newsroom/cf/dae/document.cfm?action=display&doc_id=6138`.