SPEEDL – A Declarative Event-Based Language to Define the Scaling Behavior of Cloud Applications

Rostyslav Zabolotnyi*, Philipp Leitner†, Stefan Schulte*, Schahram Dustdar*
*Distributed Systems Group, Vienna University of Technology
{rstzab, s.schulte, sd}@infosys.tuwien.ac.at
†software evolution & architecture lab, University of Zurich
leitner@ifi.uzh.ch

Abstract—**Contemporary cloud providers offer out-of-the-box auto-scaling solutions. However, defining a non-trivial scaling behavior that goes beyond the feature set provided by existing solutions is still challenging. In this paper we present *SPEEDL*, a declarative and extensible domain-specific language that simplifies the creation of elastic scaling behavior on top of IaaS clouds. *SPEEDL* simplifies the creation of event-driven policies for resource management (*How many resources, and what resource types, are needed?*), as well as task mapping (*Which tasks should be handled by which resources?*). Based on a dataset of real-life scaling policies, we demonstrate that *SPEEDL* can cover most scaling behaviors real-life developers want to express, and that the resulting *SPEEDL* policies are at the same time substantially more compact, easier to read, and less error-prone than the same behavior expressed via a general-purpose programming language.**

I. Introduction

Nowadays, the benefits of cloud computing [1] are widely recognized. Software development in the cloud is more agile, delivers value to the customer faster [2], and the "pay as you go" pricing model reduces server under-utilization [3]. However, to make efficient use of "pay as you go" pricing, cloud-native applications need to be able to adjust their cloud resource usage *elastically*. That is, an application needs to continuously monitor its state, and acquire new or release existing cloud resources accordingly. Modern Platform-as-a-Service (PaaS) solutions (e.g., Google Appengine[1] or IBM Bluemix[2]) provide simple automated, rule-based solutions to this problem, which e.g., add and remove servers based on CPU utilization thresholds. Those simple solutions are a perfect fit for many three-tier web applications [4]. However, there are many real-life applications that do not fit this model. For some applications, incoming tasks differ substantially in resource usage per request, or the architectural design requires non-trivial mapping of tasks to resources [5]. Similarly, problems appear when legislative rules regarding data handling apply. For example, the European Union establishes specific rules for how medical data is to be handled by service providers.

In these situations, cloud developers generally fall back to Infrastructure-as-a-Service (IaaS) clouds, which allow more fine-grained elasticity control. However, choosing IaaS also implies that developers have to create their own cloud management solutions, which are both, cumbersome and error-prone. Further, manual development of elasticity behavior is repetitive, as conceptually the same kind of abstract behavior needs to be implemented in many different applications.

In this paper we present *SPEEDL*, a declarative and extensible domain-specific language [6] (DSL) that simplifies the creation of elastic, application-specific cloud scaling behavior on top of IaaS clouds. *SPEEDL* allows for the definition of *scaling policies*, i.e., a set of event-condition-action (ECA) rules managing the amount and types of resources (e.g., VM instances) acquired from the cloud, as well as the mapping of incoming tasks to these resources for processing. Unlike existing industrial solutions, *SPEEDL* is extensible and allows for application-specific rules. For demonstration purposes, we illustrate *SPEEDL* using a Java-based implementation on top of our existing JCloudScale framework [7]. Further, we show, based on a dataset of real-life scaling policies, that *SPEEDL* can cover most scaling behaviors real-life developers want to express, and that the resulting *SPEEDL* policies are at the same time substantially more compact, easier to read, and less error-prone than the same behavior expressed in pure Java. An initial version of *SPEEDL* will be made available as part of the JCloudScale open source project[3], but the DSL is not hard-wired to JCloudScale and can easily be realized stand-alone as well.

II. Motivating Scenario

To motivate the remainder of this paper, we now briefly introduce an illustrative scenario of an application requiring more expressive elasticity rules than what existing industrial solutions are able to deliver.

Our motivating scenario is in the medical domain. Whenever future parents need to perform a detailed fetal scan, they have to wait for multiple days in order to obtain results. The main reason for this delay is the image processing software that struggles to provide results in a reasonable time on the limited hardware available on hospital premises. Clearly, cloud computing is a natural way to resolve this problem. Indeed, there are ongoing efforts in this direction [8].

[1]https://appengine.google.com/
[2]http://www.ibm.com/software/bluemix
[3]https://github.com/xLeitix/jcloudscale

978-1-4673-7275-6/15 $31.00 © 2015 IEEE
DOI 10.1109/SERVICES.2015.19

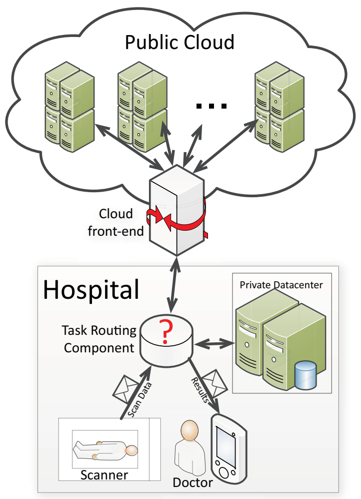

Figure 1. Overview of Motivating Scenario

A simplified illustration of this scenario is depicted in Figure 1. Each scan performed by a doctor results in a task that has to be processed using available computation resources, either within the medical institution, or in a public cloud. The clinic is saving money by processing data locally whenever sufficient resources are available or a case is marked by a medical practitioner as not urgent. However, when information is urgent, or local resources are insufficient, a task is processed in the cloud. When the task is completed, it is shown on the doctor's display.

Building the task routing component in Figure 1 results in a number of challenges. Modern cloud providers already offer load balancing solutions, but mainly within their own ecosystems and focusing on simple random or round-robin distribution [9]. In the described scenario, the system needs to schedule tasks across a hybrid cloud, depending on the current local situation and task severity. Additionally, due to legal reasons, unless a patient agrees, all private information has to stay within the hospital boundaries [10]. Another problem is that existing resource management solutions (e.g., auto-scaling) usually consist of external rule-based systems that treat the application as a black box [11]. Due to this, it is very difficult to provide any domain-specific knowledge to the external component or adjust resource usage accordingly to the doctors' schedule and already-known upcoming patient appointments. A sometimes used, but cumbersome, workaround is to use artificial metrics and events that trigger required infrastructure adaptations in advance [12].

In this paper, we present an approach that allows developers to more easily implement a custom resource management and task scheduling solution, based on doctors' timetable or legal agreements with each client, in the same way to an in-house solution. Our approach not only reduces vendor lock-in, but also allows the hospital to use their existing resources more efficiently, decide how many and which types of resources to acquire from the public cloud based on load predictions, and incorporate domain knowledge in a way that a solution that treats the application purely as a black box cannot.

III. Related Work

The problem of task scheduling did not originate in the cloud computing area. Clearly, the workload distribution challenge is present in any distributed or parallel system [13]. With the appearance and maturing of common scheduling algorithms [9], DSLs for scheduling and distributed systems started to appear. Nowadays task scheduling, often in a form of DSLs, is researched for instance within the fields of high performance computing [14] and embedded systems [15]. In the area of cloud computing, scheduling is usually performed in a form of balancing [9] or greedy [16] workload distribution in order to parallelize execution or optimize resource usage. Such approaches satisfy data processing or classic three-tier [4] cloud applications, and usually do not require complex DSLs or special scheduling frameworks. However, when task distribution needs to address such dynamic or domain-specific features as data locality [17] or system heterogeneity [18], the necessity of an additional layer of abstraction becomes more plausible. Our work does not focus on advances in cloud task scheduling. Instead, we provide a holistic approach that contains a significant amount of common algorithms and allows developers to address their workload management needs as easy and clear as possible.

Resource management in general, and the elasticity concept particularly play a vital role in cloud computing. Mainly, research is focusing on SLA-conformance [19], cost optimization [20] and "green" computing [16]. However, there are multiple DSLs and frameworks that are facilitating the problem of resource management by providing a user-friendly API and predefined set of behaviors [21]. However, these DSLs and frameworks are either completely outside of the developed application and provide some uniform means of resource management like TOSCA [22], or have a limited set of access APIs from within the developed application that allow passing information to some external decision module [21]. Instead, our paper focuses on providing a tightly-integrated, extensible, cloud management framework that is running within the developed application and does not require any standalone components.

The major difference between the presented solutions and our approach is that we are aiming at providing a cloud management component that (1) does not enforce any specific application design or architecture, (2) allows developer-friendly configuration and extension using the same language as the developed application, and (3) performs all actions within the developed application, allowing full usage of application-specific knowledge.

IV. The *SPEEDL* Language

We now introduce the *SPEEDL* design and implementation. We first discuss some relevant design considerations, which we follow-up by a detailed discussion of the language's central elements.

A. *Language Design Considerations*

While every cloud application has its own specifics and unique requirements, cloud applications typically all make use of a number of general constructs defining how cloud resources should be acquired and used. With *SPEEDL*, we structure and formalize these requirements and represent them in a declarative DSL. The design and architecture of *SPEEDL*, as well as the concrete out-of-the-box rules provided, are influenced by existing industrial cloud systems and platforms, ongoing parallel research activities in the field [21], [23] and our former experience with building and supporting elastic applications [7], [24], [25].

Existing cloud research typically models elasticity either in the form of a control loop (e.g., in the sense of autonomic computing [26]), or, more reactively, as a set of ECA rules [27]. While the former approach is often preferred in scientific work, those solutions often struggle with being narrow for a specific domain and challenging to reuse or adapt to fit different applications. The ECA-based approach avoids this problem [27]. Hence, we decided to base *SPEEDL* on the notion of complex event processing (CEP) [28], which provides reactiveness and responsiveness to complex scenarios and application behaviors. An additional advantage is that the basic declarative event-based model used by *SPEEDL* is conceptually close to how practitioners define elasticity behavior in common PaaS services [11]. Hence, we argue that the *SPEEDL* approach integrates better with current cloud developer's mindsets.

B. SPEEDL *Overview*

Scaling behavior in *SPEEDL* is provided by the developer as a scaling policy *SP*. Every application makes use of exactly one scaling policy, which can be understood as a 2-tuple $TP = < TM, RM >$, with TM being a set of task management rules, and RM a set of resource management rules. Both, TM and RM are allowed to be the empty set ($TM, RM = \{\}$). In this case, *SPEEDL* does not consider request scheduling, or does not actually scale up or down. Every concrete rule $r \in TM \cup RM$ is in turn a 3-tuple $r = < E, C, A >$, with E, C and A being sets of triggering events, guarding conditions, and resulting actions. Actions differ for task and resource management rules. For example, task management actions often entail scheduling a task to one specific resource. The notion of "task" in this scope represents any workload or application component that needs to be executed on a cloud resource. Resource management actions may, for instance, entail starting a new resource of a specific type. The ECA structure of *SPEEDL* defines a distinct responsibility of each part of the scaling policy and provides clear and effective ways to configure the behavior of each rule. Additionally, this allows applications to quickly react to changes in the system state, without requiring periodic background checks as it is common in other approaches [27]. *SPEEDL* is both, a general DSL and a prototypical implementation with Java as a host language. The implementation makes use of the existing JCloudScale framework [7] to interact with the cloud, and is technically realized as a fluent interface [29]. This makes the actual DSL feel concise, expressive, and easy to understand. Using method cascading, developers can simply invoke required rules separated by dots and produce compact and tidy code that can be read like a declarative sentence.

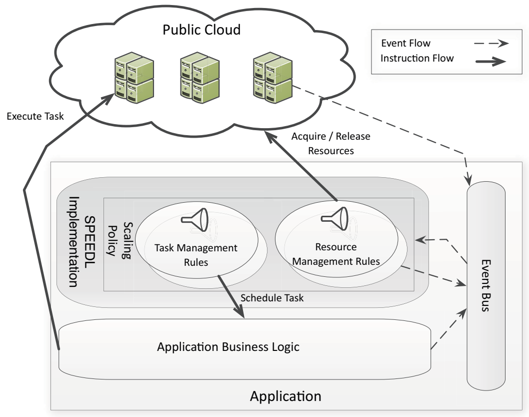

Figure 2. Using *SPEEDL* for Elasticity Control

In Figure 2, we give a high-level overview over the main components and interactions of a *SPEEDL*-based application. *SPEEDL* integrates with the actual application business logic as a third-party component (i.e., a library in the Java implementation). The *SPEEDL* implementation mainly executes a defined scaling policy, which consists of task and resource management rules. All rules are triggered via events from an event bus. This bus receives and correlates, in the sense of CEP, events from the cloud resources, the application, and *SPEEDL* itself. Task management rules instruct the application to execute specific tasks on specific hosts, while resource management rules interact with the cloud to acquire and release resources. For both, events and rules, *SPEEDL* contains a useful set of predefined constructs, which we have defined based on requirements and features of other literature and existing products. Additionally, developers are always free to extend these sets of predefined events and rules with application-specific ones.

C. *Top-Level Language Grammar*

We discuss the formal *SPEEDL* grammar using Backus Normal Form (BNF). Due to space restrictions, we focus only on the most important details, while the full grammar is available online on supplementary materials website[4]. The top level of a *SPEEDL* definition, shown in Grammar 1, consists of a rules sequence, followed by the optional `validation` section and the terminal statement (`build`). Rules are split into `Scale Up`, `Scale Down`, `Scheduling` and `Migration` sets.

[4]http://www.infosys.tuwien.ac.at/staff/phdschool/rstzab/papers/SERVICES15/

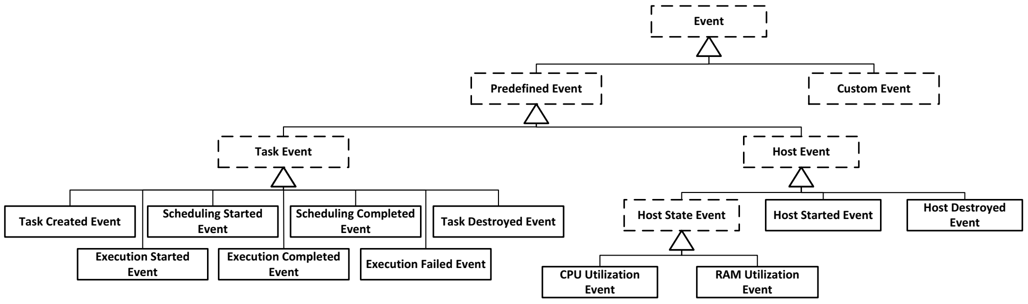

Figure 3. Simplified Hierarchy of Pre-Defined Events

$\langle$*ScalingPolicy*$\rangle$ ::= $\langle$*SPConfigElements*$\rangle$

$\langle$*SPConfigElements*$\rangle$ ::= $\langle$*Rule*$\rangle$ $\langle$*SPConfigElements*$\rangle$
| $\langle$*Validation*$\rangle$ $\langle$*SPTerminalStatement*$\rangle$
| $\langle$*SPTerminalStatement*$\rangle$

$\langle$*SPTerminalStatement*$\rangle$ ::= ‘`build`’

$\langle$*Rule*$\rangle$ ::= $\langle$*ScaleUpRule*$\rangle$
| $\langle$*ScaleDownRule*$\rangle$
| $\langle$*SchedulingRule*$\rangle$
| $\langle$*MigrationRule*$\rangle$

Grammar 1. Top-level Formal Language Specification of *SPEEDL*

The `validation` section allows to trigger an optional consistency validation of the scaling policy. *SPEEDL* distinguishes two types of validation: (1) internal rule validation warns about rules that are internally inconsistent (e.g., scaling up whether the number of hosts is larger than -1), while (2) external validation checks for inter-rule inconsistencies. Out-of-the-box, *SPEEDL* currently only supports internal rule validation. External validation logics need to be provided by the developer, if required.

In the following sections, we discuss each group of rules in more detail and introduce the available out-of-the-box constructs. Due to page limitations, we are unable to discuss every element and rule in *SPEEDL* in detail. However, we will discuss the most central and interesting features of the language, and provide code examples based on the JCloud-Scale Java implementation of *SPEEDL.*

D. Event-Driven Elasticity

Events form the basis of all rules in *SPEEDL*. Naturally, different systems and implementations make available different pre-defined events. In the Java implementation of *SPEEDL*, the events depicted in the event hierarchy in Figure 3 are available. Predefined events are mainly produced and consumed within the framework itself and cover a range of common elasticity-related situations, such as task scheduling or execution, host lifetime and resource usage. These predefined events are an extension of our earlier work in [30]. Additionally, application developers can implement custom, domain-specific events, which are typically triggered either in custom rules or directly in the application. In the hospital scenario from Section II, a potential domain-specific event may be the creation of a task that is not allowed to be scheduled to a public cloud due to privacy reasons.

E. Task Management

Task management rules focus on how to map tasks to resources. While these rules may take into account the state of the cloud infrastructure, the only actions that are initiated is that one or more tasks are assigned to exactly one host for execution. Task management rules come in two flavors, the *task scheduling* or the *task migration* rule sets. Task scheduling represents the initial mapping of a new task to a resource, while task migration re-maps an already-existing task. Task migration controls the dispersion and load of each resource by arranging and moving tasks in order to maintain overall system stability.

$\langle$*SchedulingRule*$\rangle$ ::= ‘`Schedule`’ $\langle$*ScheduledTaskType*$\rangle$ $\langle$*HostFilter*$\rangle$ $\langle$*SelectedSchedulingRule*$\rangle$
| $\langle$*customSchedulingRuleImplementation*$\rangle$

$\langle$*ScheduledTaskType*$\rangle$ ::= ‘`task`’ $\langle$*allowedTaskType*$\rangle$
| ‘’

$\langle$*HostFilter*$\rangle$ ::= ‘`allHosts`’
| ‘`onRandom`’ $\langle$*selectedHostsCount*$\rangle$
| ‘`onHosts`’ $\langle$*customHostFilter*$\rangle$
| ‘’

$\langle$*SelectedSchedulingRule*$\rangle$ ::= $\langle$*GreedyRule*$\rangle$
| $\langle$*BalancingRule*$\rangle$

$\langle$*GreedyRule*$\rangle$ ::= ‘`greedy`’ $\langle$*greedyRuleConfig*$\rangle$

$\langle$*BalancingRule*$\rangle$ ::= ‘`balance`’ $\langle$*balancingRuleConfig*$\rangle$

Grammar 2. Formal Specification of Task Scheduling Rules

1) Task Scheduling Rules: There are two prevailing approaches to distribute tasks in the cloud. (1) Balancing rules [9] aim to evenly distribute tasks over all available hosts, with the ultimate goal of achieving a close-to-uniform distribution of tasks over hosts, while (2) greedy rules [16] aim to saturate a single resource before using the next. Both of these behaviors have merits, and domain- and application knowledge

```
Schedule
  .tasks(FetalScanTask.class)
  .onHosts(
    (host, task)->
      host.getType() == (canRunInCloud(task) ?
          PUBLIC_CLOUD : PRIVATE_CLOUD))
  .greedy()
  .maxTasks(4);
```

Listing 1. Greedy Scheduling Rule

is required to select which of those fundamental strategies is more suitable.

Additionally, each balancing or greedy rule is further shaped by a set of restrictions. Developers can specify a criterion that selects the set of hosts that should be considered. Alternatively, developers can specify which type of tasks this scheduling rule applies to, as well as a maximal amount of concurrent tasks running per host. Finally, specific scoring criteria, comparable to a fitness function in optimization, can be specified for each host or scheduled task. This criterion allows developers to balance tasks depending on application-specific task properties, thus achieving better, domain-specific, scheduling results by exploiting data locality [17] or achieving cost-effectiveness.

We provide the formal definition of a scheduling rule in Grammar 2. An illustrative example of a rule that distributes fetal scan tasks between private and public cloud depending on a custom developer-defined predicate is shown in Listing 1 in the syntax of the *SPEEDL* Java implementation.

2) Task Migration Rules: In many applications, especially those with long-running tasks, e.g., scientific computing, it may often make sense to re-assign tasks that have already been started to execute on a resource. The technical process of task migration is out of scope in this paper. However, *SPEEDL* provides a set of rules that allow the definition of a migration strategy as part of the scaling policy, if the underlying application is able to suspend and move tasks, as it is for instance the case in the JCloudScale middleware used for the Java implementation of *SPEEDL*. The formal structure and main rule categories are again defined using BNF in Grammar 3.

$\langle$*MigrationRule*$\rangle$ ::= ‘`migration`’ $\langle$*MigrationType*$\rangle$
    | $\langle$*customMigrationRuleImplementation*$\rangle$

$\langle$*MigrationType*$\rangle$ ::= $\langle$*IntegrationRule*$\rangle$
    | $\langle$*OptimizationRule*$\rangle$

$\langle$*IntegrationRule*$\rangle$ ::= ‘`integrate`’ $\langle$*integrationConfig*$\rangle$

$\langle$*OptimizationRule*$\rangle$ ::= ‘`optimize`’ $\langle$*optimizationConfig*$\rangle$

Grammar 3. Formal Specification of Migration Rules

The process of task migration consists of four distinct phases. At first, situations that require migration need to be detected (*detection phase*). As *SPEEDL* is based on the notion of ECA rules, this phase is implemented via events. Next, tasks that should be migrated are selected (*task selection phase*). By default, *SPEEDL* prefers to migrate tasks that have been started last, but oftentimes an application developer will want to substitute this behavior with application-specific logic. After that, the destination host to which the task should be migrated, needs to be selected (*host selection phase*). By default this is controlled by the same metric as the migration condition (e.g., when high RAM usage is detected, objects are migrated to hosts with the least RAM usage). However, again developers are able to customize this selection strategy or provide their own implementation. Finally, the actual migration needs to be be performed (*migration phase*).

An example of an optimization migration rule that allows decreasing the load on the private hospital infrastructure of our motivating scenario by moving some tasks to the public cloud during working time is shown in Listing 2.

```
Migration.optimize()
  .hosts(host -> host.getType() == PRIVATE_CLOUD)
  .withMoreTasks(4)
  .migrateTo(host -> host.getType() == PUBLIC_CLOUD)
  .ifViolatedFor(ofMinutes(5))
  .minActionInterval(ofMinutes(10))
  .canMigrate(task -> canRunInCloud(task))
  .arrangeTasks(task -> task.getStartTime(),
      DESCENDING)
  .isEnabled(DoctorsSchedule
        .isWorkingTime(now()));
```

Listing 2. Optimizing Migration Rule

F. Resource Management

Resource management rules provide a mechanism to control and adapt the resources that the application requests from the cloud infrastructure. While rules may take into account host resource usage, task executions or the application state, the main outcome of all resource management rules is a change in the number and/or types of available resources. This happens primarily through the *scale-up* and *scale-down* rule sets. Industrial PaaS platforms usually take scale up and scale down decisions based on resource usage metrics, e.g., average CPU load. This generic approach is also supported by *SPEEDL*. However, resource-based scalability is reactive, cumbersome to write and hard to tweak [31], as all decisions have to be based on the current resource usage. Therefore, *SPEEDL* provides additionally an alternative approach that allows taking resource management decisions based on application-specific events and conditions [30]. This allows adapting cloud resource usage in advance, e.g., based on domain-specific predictions of future load. For example, in the motivating scenario in Section II, medical practitioners often know in advance when a large batch of new fetal scans is due, based on their appointment schedule.

1) Scale-Up Rules: Scaling up is usually controlled via one or more application-dependent metrics (e.g., CPU/RAM usage, task throughput, predictions of future load). The behavior of all those rules is similar – if a metric threshold is exceeded, a scale-up action is executed. Hence, we created a single configurable behavior policy that accepts a controlled metric and additional configuration that allows defining the actual action, e.g., how many and which resources to start. The event-based nature of *SPEEDL* gives us the ability to flexibly adjust thresholds and actions, depending on application needs. Further, by leveraging complex event processing, developers have access to powerful means of data aggregation and analysis when defining metrics. However, in addition to these metric-threshold based rules, *SPEEDL* also contains other rules for scale-up. For long-running applications, *SPEEDL* also provides time-based scale-up rules. These rules do not trigger based on changes in the actual or predicted load, but ensure that a proper amount of hosts is running at specified points in time. This model is suitable for applications with well-known periods of high usage. A formal definition of *SPEEDL* scale-up rules is given in Grammar 4.

$\langle \textit{ScaleUpRule} \rangle$ ::= ‘`scale up`’ $\langle \textit{HostFilter} \rangle$ $\langle \textit{ScaleUpRule} \rangle$
  | $\langle \textit{customScaleUpRuleImplementation} \rangle$

$\langle \textit{HostFilter} \rangle$ ::= ‘`allHosts`’
  | ‘`hosts`’ $\langle \textit{customHostFilter} \rangle$
  | ‘’

$\langle \textit{ScaleUpRule} \rangle$ ::= $\langle \textit{hostStateRule} \rangle$
  | $\langle \textit{timeBasedRule} \rangle$
  | $\langle \textit{taskQueueStateRule} \rangle$
  | $\langle \textit{customMetricRule} \rangle$

Grammar 4. *SPEEDL* scale up rules specification.

A sample scale-up rule that scales from 1 to 20 cloud hosts when we have more scheduled appointments over the next hour than we have processing resources, is shown in Listing 3.

```
ScaleUp
  .hosts(host ->
    host.getType() == PUBLIC_CLOUD)
  .when(hosts ->
    countTaskCapacity(hosts) < DoctorsSchedule
        .appointments(now(), ofHours(1)))
  .checkEvery(ofMinutes(5))
  .minHosts(1)
  .maxHosts(20)
  .scaleUpStep(1)
  .newHostType("PUBLIC_CLOUD", "m1.small")
  .minScaleUpInterval(ofMinutes(10));
```

Listing 3. A Scale-Up Rule Based on a Domain-Specific Metric

2) Scale-Down Rules: While scaling up is often based on current or predicted load, scaling down should in contemporary IaaS cloud systems be aligned with the billing time unit (BTU) of the cloud provider. In IaaS cloud systems, computing resources are typically billed periodically (e.g., hourly in Amazon EC2[5], per minute after the first 10 minutes in Google). Economically, it makes little sense to release a resource while it is still paid for. Hence, evaluation whether resources should be scaled down or not in *SPEEDL* is triggered briefly before the resource would enter the next billing period. A second peculiarity of scaling down is that it often needs to integrate with migration (see Section IV-E2) in order to move tasks still scheduled to a resource that is about to be scaled down. Aside from those aspects, scaling down is conceptually similar to scaling up. As always, a formal definition of scale-down rules in BNF is presented in Grammar 5.

[5]http://aws.amazon.com/ec2/

$\langle \textit{ScaleDownRule} \rangle$ ::= ‘`scale down`’ $\langle \textit{ScaleDownRule} \rangle$
  | $\langle \textit{customScaleDownRuleImplementation} \rangle$

$\langle \textit{ScaleDownRule} \rangle$ ::= $\langle \textit{resourceUsageRule} \rangle$
  | $\langle \textit{hostStateRule} \rangle$
  | $\langle \textit{taskQueueStateRule} \rangle$
  | $\langle \textit{timeBasedRule} \rangle$
  | $\langle \textit{customMetricRule} \rangle$

Grammar 5. *SPEEDL* scale down rules specification.

An example of a scale-down rule that releases cloud resources when no longer needed outside of hospital working hours is shown in Listing 4.

```
ScaleDown.runningTasks(0)
      .checkAdvance(ofMinutes(1))
      .minHosts(
        host -> host.getType() == PUBLIC_CLOUD, 1)
      .isEnabled(
        host -> host.getType() == PUBLIC_CLOUD &&
          !DoctorsSchedule.isWorkingTime(now()))
```

Listing 4. A Scale-Down Rule Based on Task Count

Hosts that are currently running tasks may also be scaled down. In some cases, it is safe to restart the aborted task on another host. This is a common assumption in many state-of-the-art PaaS platforms, which primarily deal with HTTP requests as tasks. However, this is not always the case. Sometimes, tasks cannot be aborted due to high startup costs, or the possibility of introducing state inconsistencies. In such cases, the host either has to be left running until the tasks are finished or, if this is possible, the tasks have to be migrated to another host. In *SPEEDL* this is controlled by the *ifWithTasks* condition. It defines whether tasks can be discarded, left running or migrated to another host. In more sophisticated cases, developers can perform any custom actions with a particular host (including task migration or abortion) within the custom predicate that allows determining if particular scale down rule is applicable to this host. As an example, such a custom predicate is used in Listing 4 to release only cloud hosts while the hospital is not operating.

V. Evaluation

In order to validate our work, we now evaluate *SPEEDL* based on a dataset of real-life scaling code collected during a previous user study.

A. Evaluation Setup

For evaluation, we use a dataset of IaaS scaling code in Java, which we harvested during a previous programming study. The setup and results of this study are described in detail in [25]. Summarizing, we had 14 male Master students of computer science at TU Vienna implement non-trivial elastic applications on top of EC2 and OpenStack[6]. One of the central outcomes of this earlier study was that implementing the scaling behavior was harder than actually getting the business logics of the application right. In fact, multiple participants reported that they spent significant amount of the alloted time trying to develop a robust scaling behavior [25].

```
synchronized (lock)
{
  try {
    // dirty hack to get correct
    // host.getCloudObjectsCount()
    Thread.sleep(1000);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
  while(selectedHost == null) { ... }
}
```

Listing 5. Snippet from Real-Life Scaling Code

From the resulting applications, we extracted the code that was responsible for scaling and task distribution. The scaling code we obtained in this way differs dramatically in size and complexity. The shortest was only 27 lines of Java code (LoC), while the longest one was 177 LoC (median length was 75 LoC). Informal inspection of these code snippets revealed that many study participants indeed struggled with getting the scaling behavior right, and ended up building rather fragile, “hacky” Java solutions (see Listing 5 for an example). After detailed analysis by the first author, we manually implemented equivalent code in the *SPEEDL* DSL. Both, extracted scaling code and the equivalent *SPEEDL* scaling policies are available online on supplementary materials website.

B. Results and Discussion

We now compare the original Java-based solutions to *SPEEDL*. We are particularly interested in the amount of code necessary to represent the same behavior, and to what extend the out-of-the-box rules of *SPEEDL* are useful for expressing the scaling behavior that the participants of our study wanted to implement.

In terms of LoC, the *SPEEDL* representations indeed turned out to be substantially shorter than the equivalent pure Java code (shortest was 8 LoC, longest was 21 LoC, with a median of 11.5 LoC). This is illustrated in Figure 4, which depicts the LoC for each scaling behavior next to the size of an equivalent *SPEEDL* policy. Additionally, we argue that the more compact *SPEEDL* equivalents are also easier to read and comprehend. For instance, we were able to replace a complex multi-method scaling behavior, which included “sleep” statements, nested iterations, and global locking for synchronization with the *SPEEDL* policy shown in Listing 6.

As a second step, we evaluated to what extend the existing rules in *SPEEDL* are able to cover the needs of real-life

[6]https://www.openstack.org

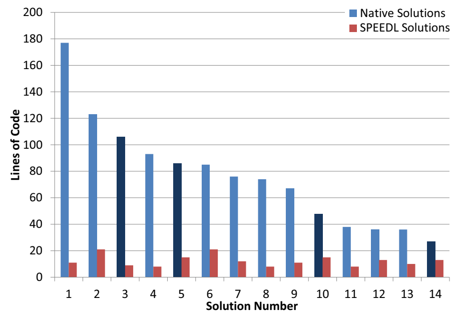

Figure 4. Length Comparison of Evaluated Scaling Policies

application developers, and how often developers need to build custom rules. Our analysis showed that we were able to provide equivalent versions of 71% of all scaling behaviors in the dataset using out-of-the-box rules alone. 100% of all behaviors could be represented with a small amount of custom rules. The scaling behaviors that required custom rules are plotted in darker color in Figure 4.

```
SmartPolicy
  .create(Schedule.greedy()
      .maxTasks(
        schedulerConfig.getMaxCloudObjects()))
  .add(ScaleUp
    .queueLength(
      schedulerConfig.getMaxCloudObjects())
      .maxHosts(schedulerConfig.getMaxNodes())
      .newHostsType(schedulerConfig.getFlavor()))
  .add(ScaleDown.runningTasks(0))
```

Listing 6. Complete Example of a *SPEEDL* Scaling Policy

Finally, another interesting observation was that 4 of the scaling behaviors in the dataset contained minor errors (29%). 3 scaling behaviors are not correctly synchronized and can potentially fail due to race conditions. Similarly, another code has (based on the intent shown in a code comment) incorrectly defined “if"-conditions, which would lead to unwanted scaling in edge cases. Hence, the usage of the out-of-the-box rules of *SPEEDL* does not only simplify the definition of scaling behavior, but also reduces the potential for developer errors.

We conclude that *SPEEDL* indeed provides a noteworthy improvement in scaling policy readability and length. While the existing out-of-the-box rules provided by *SPEEDL* cannot cover everything a developer would want to express, we were still able to re-implement 71% of all scaling behaviors using out-of-the-box rules alone. Using custom rules, we were able to implement equivalents to all scaling behaviors. Finally, we have seen that the complications of building real-life scaling behavior can easily lead to hard-to-detect bugs, such as race conditions. Using the out-of-the-box rules of *SPEEDL* greatly reduces the risk of such bugs.

VI. Conclusions

In this paper we presented *SPEEDL*, a domain-specific declarative language that simplifies defining advanced task and resource management policies for IaaS cloud applications. Contrary to existing approaches, *SPEEDL* is aiming to provide cloud management abilities as part of the cloud application rather than via an external system, thus allowing developers to incorporate domain-specific information and flexible application design. As we showed in our evaluation, using *SPEEDL*, developers can significantly decrease the code necessary for defining custom scaling solutions, simplify the creation of sophisticated scaling policies and decrease the amount of errors. Additionally, we have shown that the pre-defined *SPEEDL* rules alone already cover 71% of the functionality of our dataset of existing scaling policies.

While we believe that the current version of *SPEEDL* is already useful and improves developers' experience when interacting with IaaS clouds, there are still a number of improvements we plan to investigate in future. For example, currently *SPEEDL* is designed to run on a single machine and to solely control the execution of a single application. In the future, we plan to soften this constraint, to prevent *SPEEDL* becoming a single point of failure and hamper scalability. Additionally, we plan to continue enhancing the set of pre-defined rules provided with *SPEEDL*, as well as further improve the expressibility of the language itself.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreements 610802 (CloudWave) and 318201 (SIMPLI-CITY).

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 Computing Systems*, vol. 25, no. 6, pp. 599–616, 2009.

[2] J. Cito, P. Leitner, T. Fritz, and H. C. Gall, "The Making of Cloud Applications – An Empirical Study on Software Development for the Cloud," 2014. arXiv e-print. arXiv:1409.6502 [cs.SE].

[3] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 50–55, 2008.

[4] J. Bi, Z. Zhu, R. Tian, and Q. Wang, "Dynamic Provisioning Modeling for Virtualized Multi-tier Applications in Cloud Data Center," in *3rd International Conference on Cloud Computing (CLOUD 2010)*, pp. 370–377, IEEE, 2010.

[5] D. Warneke and O. Kao, "Exploiting Dynamic Resource Allocation for Efficient Parallel Data Processing in the Cloud," *IEEE Transactions Parallel and Distributed Systems*, vol. 22, no. 6, pp. 985–997, 2011.

[6] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, 2005.

[7] P. Leitner, B. Satzger, W. Hummer, C. Inzinger, and S. Dustdar, "CloudScale – A Novel Middleware for Building Transparently Scaling Cloud Applications," in *ACM Symposium on Applied Computing (SAC'12)*, pp. 434–440, ACM, 2012.

[8] R. Barlow, "Medicine in the cloud," 2013. "http://www.bu.edu/bostonia/winter-spring13/medicine-in-the-cloud/" , Last accessed: 2015-02-11.

[9] P. Naik, S. Agrawal, and S. Murthy, "A survey on various task scheduling algorithms toward load balancing in public cloud," *American Journal of Applied Mathematics*, vol. 3, no. 1-2, pp. 14–17, 2015.

[10] E. J. Schweitzer, "Reconciliation of the cloud computing model with US federal electronic health record regulations," *Journal of the American Medical Informatics Association*, vol. 19, no. 2, pp. 161–165, 2012.

[11] G. Galante and L. C. Bona, "A Survey on Cloud Computing Elasticity," in *2012 IEEE Fifth International Conference on Utility and Cloud Computing (UCC 2012)*, pp. 263–270, IEEE, 2012.

[12] M. Xiao and X. Song, "A Survey of Cloud Computing Dynamic Resource Management," *International Journal of Advancements in Computing Technology*, vol. 4, no. 14, pp. 498–506, 2012.

[13] H. El-Rewini and T. G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *Journal of Parallel and Distributed Computing*, vol. 9, no. 2, pp. 138–153, 1990.

[14] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for High Performance Computing," *Parallel Computing*, vol. 38, no. 1–2, pp. 37–51, 2012.

[15] F. Singhoff and A. Plantec, "AADL Modeling and Analysis of Hierarchical Schedulers," in *2007 Annual ACM SIGAda International Conference on Ada*, pp. 41–50, ACM, 2007.

[16] C.-C. Lin, P. Liu, and J.-J. Wu, "Energy-Aware Virtual Machine Dynamic Provision and Scheduling for Cloud Computing," in *4th International Conference on Cloud Computing (CLOUD 2011)*, pp. 736–737, IEEE, 2011.

[17] J. Jin, J. Luo, A. Song, F. Dong, and R. Xiong, "BAR: An Efficient Data Locality Driven Task Scheduling Algorithm for Cloud Computing," in *The 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID '11)*, pp. 295–304, IEEE, 2011.

[18] H. Choi, W. Choi, T. M. Quan, D. Hildebrand, H. Pfister, and W.-K. Jeong, "Vivaldi: A Domain-Specific Language for Volume Processing and Visualization on Distributed Heterogeneous Systems," *IEEE Transactions on Visualization and Computer Graphics,*, vol. 20, no. 12, pp. 2407–2416, 2014.

[19] P. Leitner, W. Hummer, B. Satzger, C. Inzinger, and S. Dustdar, "Cost-Efficient and Application SLA-Aware Client Side Request Scheduling in an Infrastructure-as-a-Service Cloud," in *Fifth International Conference on Cloud Computing (CLOUD 2012)*, pp. 213–220, IEEE, 2012.

[20] S. Chaisiri, B.-S. Lee, and D. Niyato, "Optimization of Resource Provisioning Cost in Cloud Computing," *IEEE Transactions on Services Computing*, vol. 5, no. 2, pp. 164–177, 2012.

[21] G. Copil, D. Moldovan, H.-L. Truong, and S. Dustdar, "SYBL: An Extensible Language for Controlling Elasticity in Cloud Applications," in *The 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID '13)*, pp. 112–119, IEEE, 2013.

[22] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, "TOSCA: Portable Automated Deployment and Management of Cloud Applications," in *Advanced Web Services*, pp. 527–549, Springer, 2014.

[23] S. Schulte, D. Schuller, P. Hoenisch, U. Lampe, S. Dustdar, and R. Steinmetz, "Cost-Driven Optimization of Cloud Resource Allocation for Elastic Processes," *International Journal of Cloud Computing*, vol. 1, no. 2, pp. 1–14, 2013.

[24] R. Zabolotnyi, P. Leitner, and S. Dustdar, "Profiling-Based Task Scheduling for Factory-Worker Applications in Infrastructure-as-a-Service Clouds," in *40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 119–126, IEEE, 2014.

[25] R. Zabolotnyi, P. Leitner, W. Hummer, and S. Dustdar, "JCloudScale: Closing the Gap Between IaaS and PaaS," 2014. arXiv e-print. arXiv:1411.2392 [cs.SE].

[26] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[27] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Exploring Alternative Approaches to Implement an Elasticity Policy," in *4th International Conference on Cloud Computing (CLOUD 2011)*, pp. 716–723, IEEE, 2011.

[28] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[29] M. Fowler, *Domain Specific Languages*. Pearson Education, 2010.

[30] P. Leitner, C. Inzinger, W. Hummer, B. Satzger, and S. Dustdar, "Application-Level Performance Monitoring of Cloud Services Based on the Complex Event Processing Paradigm," in *5th IEEE International Conference on Service-Oriented Computing and Applications (SOCA'12)*, 2012.

[31] J. Allspaw, *The Art of Capacity Planning: Scaling Web Resources*. O'Reilly Media, Inc., 2008.