JCloudScale: Closing the Gap Between IaaS and PaaS

ROSTYSLAV ZABOLOTNYI, Distributed Systems Group, Vienna University of Technology
PHILIPP LEITNER, Software Evolution & Architecture Lab, University of Zurich
WALDEMAR HUMMER and SCHAHRAM DUSTDAR, Distributed Systems Group,
Vienna University of Technology

Building Infrastructure-as-a-Service (IaaS) applications today is a complex, repetitive, and error-prone endeavor, as IaaS does not provide abstractions on top of virtual machines. This article presents JCloudScale, a Java-based middleware for moving elastic applications to IaaS clouds, with minimal adjustments to the application code. We discuss the architecture and technical features, as well as evaluate our system with regard to user acceptance and performance overhead. Our user study reveals that JCloudScale allows many participants to build IaaS applications more efficiently, compared to industrial Platform-as-a-Service (PaaS) solutions. Additionally, unlike PaaS, JCloudScale does not lead to a control loss and vendor lock-in.

CCS Concepts: • **Networks** → **Cloud computing**; • **Software and its engineering** → **Object oriented frameworks**; • **Computing methodologies** → *Distributed computing methodologies*; • **Computer systems organization** → *Client-server architectures;*

Additional Key Words and Phrases: Cloud computing, middleware, programming, JCloudScale

ACM Reference Format:
Rostyslav Zabolotnyi, Philipp Leitner, Waldemar Hummer, and Schahram Dustdar. 2015. JCloudScale: Closing the gap between IaaS and PaaS. ACM Trans. Internet Technol. 15, 3, Article 10 (July 2015), 20 pages.
DOI: http://dx.doi.org/10.1145/2792980

1. INTRODUCTION

In recent years, the cloud computing paradigm [Buyya et al. 2009] has provoked a significant push towards more flexible provisioning of IT resources, including computing power, storage, and networking capabilities. Besides economic factors, the core driver behind this cloud-computing hype is the idea of elastic computing. Elastic applications are able to increase and decrease their resource usage based on current application load, for instance, by adding and removing computing nodes. Optimally, elastic applications are cost and energy efficient, while still providing the expected level of application performance.

Elastic applications are typically built using either the Infrastructure-as-a-Service (IaaS) or Platform-as-a-Service (PaaS) paradigm [Armbrust et al. 2010]. In IaaS, users rent virtual machines from the cloud provider and retain full control (e.g., administrator rights). In PaaS, the level of abstraction is higher, as the cloud provider is responsible for managing virtual resources. In theory, this allows for more efficient

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 610802 (CloudWave) and no. 318201 (SIMPLICITY).
Authors' addresses: R. Zabolotnyi, W. Hummer, and S. Dustdar, TU Wien, Institute of Information Systems 184/1, Distributed Systems Group, Argentinierstrasse, 8 A-1040 Vienna, Austria; emails: {rstzab, hummer, sd}@dsg.tuwien.ac.at; P. Leitner, University of Zurich, Department of Informatics, Binzmühlestrasse 14, CH-8050 Zurich, Switzerland; email: leitner@ifi.uzh.ch.

DOI: http://dx.doi.org/10.1145/2792980

cloud application development, as less boilerplate code (e.g., for creating and destroying virtual machines, monitoring and load balancing, or application code distribution) is required. However, practice has shown that today's PaaS offerings (e.g., Windows Azure, Google's AppEngine, or Amazon's Elastic Beanstalk) come with significant disadvantages, which render this option infeasible for many developers. These problems include: (1) strong vendor lock-in [Dillon et al. 2010], as one is typically required to program against a proprietary API; (2) limited control over the elasticity behavior of the application (e.g., developers have very little influence on when to scale up and down); (3) no root access to the virtual servers running the actual application code; and (4) little support for building applications that do not follow the basic architectural patterns assumed by the PaaS offering [Jayaram 2013] (e.g., Apache Tomcat–based Web applications). All in all, developers are often forced to fall back to IaaS for many use cases, despite the significant advantages that the PaaS model would promise.

In this article, we discuss an alternative model for building elastic cloud applications, based on our initial work in Leitner et al. [2012]. We introduce JCloudScale, a Java-based middleware that eases the task of building elastic applications. Similar to PaaS, JCloudScale takes over virtual machine management, application monitoring, load balancing, and code distribution. However, given that JCloudScale is a client-side middleware instead of a complete hosting environment, developers retain full control. Furthermore, JCloudScale supports a wider range of applications. JCloudScale applications run on top of any IaaS cloud, making JCloudScale a viable solution to implement applications for private or hybrid cloud settings [Sotomayor et al. 2009]. In summary, we claim that the JCloudScale model is a promising compromise between IaaS and PaaS, combining many advantages of both worlds.

We validate JCloudScale via a user study, comparing our model to both existing IaaS (OpenStack and Amazon EC2) and PaaS (Amazon Elastic Beanstalk) systems. We address the runtime performance impact of JCloudScale, as well as development productivity and user acceptance. Our study results suggest that JCloudScale increases developer productivity in comparison to pure IaaS solutions and, to a lesser extent, to Elastic Beanstalk. Unlike Elastic Beanstalk, JCloudScale is more flexible, does not lead to vendor lock-in, and can also be used in a private or hybrid cloud environment. However, our results also show that there still are technical issues in the current prototype that need to be addressed. Further, our results show that, in its current version, JCloudScale impacts performance in a noticeable manner. JCloudScale is readily available as an open-source project on GitHub.

2. THE JCLOUDSCALE MIDDLEWARE

In the following, we introduce the main notions and features of JCloudScale.

2.1. Basic Notions

JCloudScale is a Java-based middleware for building elastic IaaS applications. The ultimate aim of JCloudScale is to help developers facilitate implementation of cloud applications (in the following referred to as *target applications*) as local, multithreaded applications, without even being aware of the cloud deployment. That is, the target application is not aware of the underlying physical distribution, and does not need to care about technicalities of elasticity, such as program code distribution, virtual machine instantiation and destruction, performance monitoring, and load balancing. This is achieved with a declarative programming model (implemented via Java annotations) combined with bytecode modification. To the developer, JCloudScale appears as an additional library (e.g., a Maven dependency) plus a postcompilation build step. This

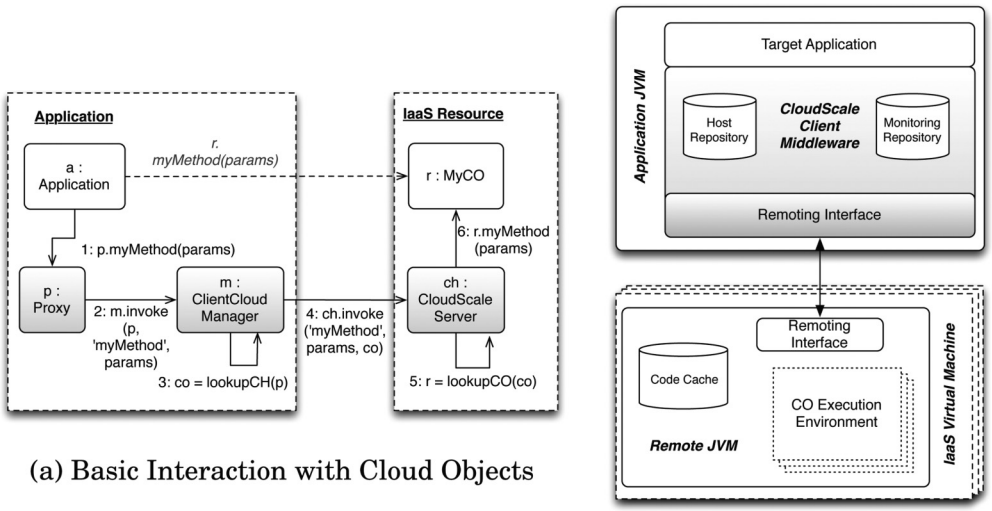

Fig. 1. JCloudScale Overview.

puts JCloudScale in stark contrast to most industrial PaaS solutions, which require applications to be built specifically for these platforms. Such PaaS applications are usually not executable outside of the targeted PaaS environment.

The primary entities of JCloudScale are *cloud objects* (COs). COs are object instances that execute in the cloud. COs are deployed to, and executed by, so-called *cloud hosts* (CHs). CHs are virtual machines acquired from the IaaS cloud, which run a JCloudScale server component. They accept COs to host and execute on client request. The program code responsible for managing virtual machines, dispatching requests to virtual machines, class loading, and monitoring is injected into the target application as a postcompilation build step via bytecode modification. Optimally, COs are highly cohesive and loosely coupled to the rest of the target application as, after cloud deployment, every further interaction with the CO constitutes a remote invocation over the network. Figure 1(a) illustrates the basic operation of JCloudScale in an interaction diagram. The gray boxes indicate code that is injected. Hence, these steps are transparent to the application developer.

Figure 1(b) shows a high-level deployment view of a JCloudScale application. The gray box in the target application JVM again indicates injected components. Note that CHs are conceptually "thin" components, that is, most of the actual JCloudScale business logics is running on the client side in the target application JVM. In its current version, JCloudScale does not support target applications that are themselves distributed. CHs consist mainly of a small server component that accepts requests from clients, a code cache used for classloading, and sand boxes for executing COs. As JCloudScale currently does not explicitly target multitenancy [Bezemer et al. 2010], these sand boxes are currently implemented in a lightweight way via custom Java classloaders. On the client side, the JCloudScale middleware collects and aggregates monitoring data, and maintains a list of CHs and COs. Further, the client-side middleware is responsible for scaling up and down based on user-defined policies (see Section 3.1).

Listing 1: Declaring COs in Target Applications

```
1  @CloudObject
2  public class MyCO {
3
4    @CloudGlobal
5    private static String myCoName;
6
7    @EventSink
8    private EventSink eventsink;
9
10   public MyResult myMethod(@ByValueParameter MyParameters params) {
11     ...
12   }
13 }
```

2.2. Interacting with Cloud Objects

Application developers declare COs in their application code via simple Java annotations. In the following, we refer to the minimal example given in Listing 1. A more comprehensive example, which also includes a step-by-step tutorial, is available online[1].

As is the case for any object in Java, the target application can fundamentally interact with COs in two different ways: invoking CO methods and getting or setting CO member fields. In both cases, JCloudScale intercepts the operation, executes the requested operation on the CH, and returns the result (if any) back to the target application. In the meantime, the target application is blocked (more concretely, the target application remains in an "idle wait" state while it is waiting for the CH response). Fundamentally, JCloudScale aims to preserve the functional semantics of the target application after bytecode modification. That is, every method call or field operation behaves functionally identical to a regular Java program.

One exception to this rule is CO-defining classes that contain static fields and methods. Operations on those are by default not intercepted by JCloudScale for performance reasons, as this would introduce a significant overhead even if the target application reads only from such static fields. However, this potentially leads to a problem that we refer to as *JVM-local updates*: if code executing on a CH (e.g., a CO instance method) changes the value of a static field, only the copy in this CH's JVM will be changed. Other COs, or the target application JVM, are not aware of the change. Hence, in this case, the value of the static field is tainted, and the execution semantics of the application changes after JCloudScale bytecode injection. To prevent this problem and preserve standard Java language semantics, static fields can be annotated with the `@CloudGlobal` annotation (see Listing 1, Lines 4 and 5). Changes to such static fields are maintained in the target application JVM, and all CH JVMs are operating on the target application JVM copy via callback.

2.3. Handling JCloudScale Faults

Distributing applications with JCloudScale can potentially introduce faults that are not apparent as long as the target application is executed locally. For instance, transient network outages can mean that a subset of COs is temporarily not available, or a terminated CH can lead to a permanent loss of COs. At this stage, JCloudScale does not provide sophisticated features to deal with these situations. However, JCloudScale notifies the target application via a custom exception type (`JCloudScaleException`) and a more detailed exception message about such problems, allowing developers to deal with these issues as required in the target application.

[1]https://github.com/xLeitix/jcloudscale/blob/master/docs/FirstSteps.md.

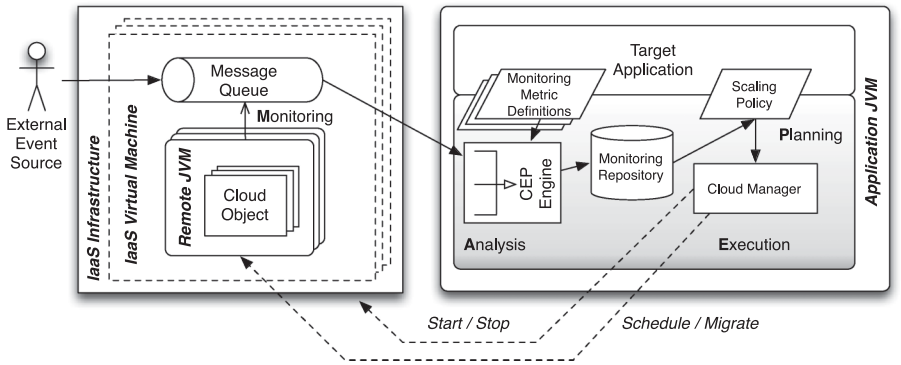

Fig. 2. Autonomic elasticity.

Fundamentally, JCloudScale is most suitable for applications in which the loss of individual COs or CHs is noncritical. This is in line with standard cloud architectures, which typically promote designing for failure, for example, by adopting statelessness and redundancy [Cito et al. 2015]. Built-in support for redundancy is not part of the current JCloudScale release, but is part of our future research.

2.4. Remote Classloading

Whenever a CH has to execute a CO method, JCloudScale has to ensure that all necessary resources (i.e., program code and other files, e.g., configuration files) are available on that CH. In order to ensure freshness of the available code and to retrieve missing files, we intercept the default class loading mechanism of Java and verify that the code available to the CH is the same as the one referenced by the client. If this is not the case, the correct version of the code is fetched dynamically from the target application. In order to improve performance, CHs additionally maintain a code cache, which is a high-speed storage of recently used code. This mechanism allows JCloudScale to load missing or modified code efficiently and seamlessly for the application only when necessary, thus simplifying application development and maintenance. We discuss this process in more detail in Zabolotnyi et al. [2013].

3. SUPPORTING CLOUD ELASTIC APPLICATIONS

So far, we have discussed how JCloudScale transparently enables remoting in cloud applications. We now explain how JCloudScale enables elastic applications.

3.1. Autonomic Elasticity via Complex Event Processing

One central advantage of JCloudScale is that it allows for building elastic applications by mapping requests to a dynamic pool of CHs. This encompasses three related tasks: (1) performance monitoring, (2) CH provisioning and deprovisioning, and (3) CO-to-CH scheduling and CO migration. One design goal of JCloudScale is to abstract from technicalities of these tasks, but still grant developers low-level control over elasticity behavior.

An overview of the JCloudScale components related to elasticity and their interactions is given in Figure 2. Conceptually, our system implements the well-established autonomic computing control loop of monitoring-analysis-planning-execution [Kephart and Chess 2003] (MAPE). The base data of monitoring is provided using event messages. All components in a JCloudScale system (COs, CHs, as well as the middleware itself) trigger a variety of predefined lifecycle and status events, indicating, for instance,

Listing 2: Example Round-Robin Scaling Policy

```
1  public class RoundRobin extends AbstractScalingPolicy {
2
3    int index = 0;
4
5    public IHost selectHost(ClientCloudObject newCloudObject, IHostPool hostPool) {
6      return hostPool.getHosts().get((index++) % hostPool.getHostsCount());
7    }
8
9    public boolean scaleDown(IHost host, IHostPool hostPool) {
10     return host.getCloudObjects().size() == 0;
11   }
12 }
```

Listing 3: Defining Monitoring Metrics via CEP

```
1  MonitoringMetric metric =
2    new MonitoringMetric();
3  metric.setName("AvgEngineSetupTime");
4  metric.setType(Double.class);
5  metric.setEpl(
6    "select avg(duration)
7     from EngineSetupEvent.win
8       :time_batch(10 sec)"
9  );
10 EventCorrelationEngine.getInstance()
11   .registerMetric(metric);
```

that a new CO has been deployed or that the execution of a CO method has failed. Additionally, JCloudScale makes it easy for applications to trigger custom (application-specific) events. Finally, events may also be produced by *external event sources*, such as an external monitoring framework. All these events form a consolidated stream of monitoring events in a *message queue*, by which they are forwarded into a *complex event processing (CEP) engine* [Luckham 2002] for analysis. CEP is the process of merging a large number of low-level events into high-level knowledge, for example, many atomic execution time events can be merged into meaningful performance indicators for the system in total.

Developers steer the scaling behavior by defining a *scaling policy*, which implements the planning part of this MAPE loop. This policy is invoked whenever a new CO needs to be scheduled, and is also responsible for deciding whether to deprovision an existing CH at the end of each billing time unit. We provide a simplistic example in Listing 2. This policy schedules COs in a round-robin fashion among existing CHs, and never scales up. The policy terminates a host if it is unused (i.e., there are no COs deployed at it) at the end of its billing time unit.

Clearly, most real scaling policies are more complex than the one in Listing 2. Using the `ClientCloudObject`, `IHostPool`, and `IHost` APIs, developers are able to schedule the provisioning of new CHs (optionally asynchronously), migrate existing COs between CHs, and schedule COs to a CH. Often, these decisions will be based on monitoring data. Hence, developers can define any number of *monitoring metrics*. Metrics are simple 3-tuples *<name, type, cep-statement>*. CEP-statements are defined over the stream of monitoring events. An example, which defines a metric `AvgEngineSetupTime` of type `java.lang.Double` as the average `duration` value of all `EngineSetupEvents` received in a 10-second batch, is given in Listing 3.

Monitoring metrics range from very simple and domain-independent (e.g., calculating the average CPU utilization of all CHs) to application-specific ones, such as the example given in Listing 3. Whenever the CEP-statement is triggered, the CEP engine writes

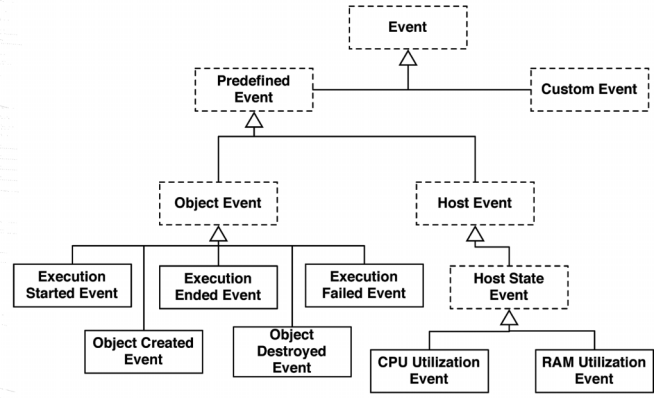

Fig. 3. Monitoring event hierarchy.

a new value to an in-memory *monitoring repository*. *Scaling policies* have access to this repository, and make use of its content in their decisions. In combination with monitoring metrics, scaling policies are a well-suited tool for developers to specify how the application should react to changes in its workload. Hence, sophisticated scaling policies that minimize cloud infrastructure costs or that maximize utilization [Genaud and Gossa 2011] are easy to integrate.

Finally, the *cloud manager* component, which can be seen as the heart of the JCloudScale client-side middleware and the executor of the MAPE loop, enacts the decisions of the policy by invoking the respective functions of the IaaS API and the CH remote interfaces (e.g., provisioning of new CHs, deprovisioning of existing ones, as well as the deployment or migration of COs).

Figure 3 depicts the type hierarchy of all predefined events in JCloudScale. Dashed classes denote abstract events, which are not triggered directly, but serve as classifications for groups of related events. All events further contain a varying number of event properties, which form the core information of the event. For instance, for `ExecutionFailedEvent`, the properties contain the CO, the invoked method, and the actual error. Developers and *external event sources* can extend this event hierarchy by inheriting from `CustomEvent`, and writing these custom events into a special event sink (injected by the middleware, see Listing 1). This process is described in more detail in Leitner et al. [2012].

3.2. Deploying to the Cloud

As all code that interacts with the IaaS cloud is injected, the JCloudScale programming model naturally decouples Java applications from the cloud environment to which they are physically deployed. This allows developers to redeploy the same application to a different cloud simply by changing the respective parts of the JCloudScale configuration. JCloudScale currently contains three separate cloud backends, supporting OpenStack-based private clouds, the Amazon EC2 public cloud, and a special *local environment*. The local environment does not use an actual cloud at all, but simulates CHs by starting new JVMs on the same physical machine as the target application. Support for more IaaS clouds, for instance, Microsoft Azure's virtual machine cloud, is an ongoing activity. Moreover, we aim to introduce systematic testing to ensure reliable deployment of CHs, which is a key requirement for elasticity [Hummer et al. 2013].

It is also possible to combine different environments, enabling hybrid cloud applications. In this case, the scaling policy is responsible for deciding which CO to execute

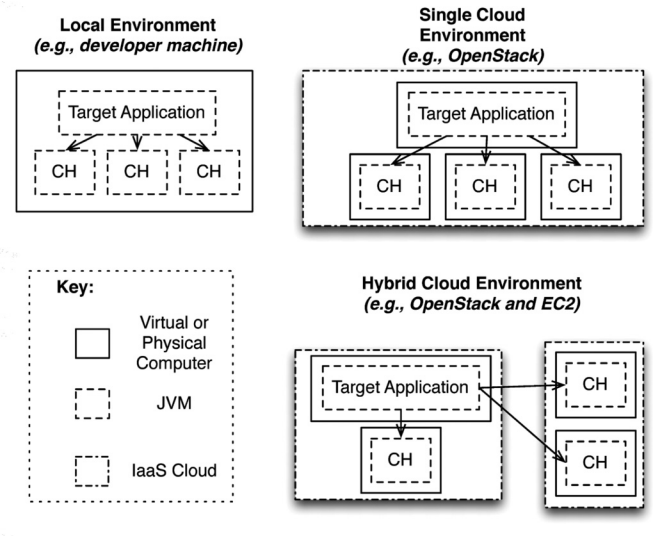

Fig. 4. Supported deployment environments.

on which cloud. Figure 4 illustrates the different types of environments supported by JCloudScale.

4. VALIDATION

As part of our validation of the JCloudScale framework, we aim at answering the following three research questions:

—RQ1: Does using JCloudScale instead of established tooling lead to more efficient development of cloud solutions, for example, in terms of solution size or development time?
—RQ2: How does JCloudScale compare with established tooling in terms of ease of use, debugging, and other more "soft" quality dimensions?
—RQ3: What runtime overhead does JCloudScale impose at execution time?

In order to answer RQ1 and RQ2, we conducted a multimonth user study. RQ3 is addressed via numerical overhead measurements on a case study application, with and without JCloudScale.

4.1. User Study

In order to evaluate RQ1 and RQ2, we conducted a user study with 14 participants to assess the developers' experience with JCloudScale as compared to using standard tools. Following the general ideas of action research [Avison et al. 1999], we aimed at a study methodology that focused on how real developers would actually use our middleware to build two separate, nontrivial cloud applications.

4.1.1. Study Setup and Methodology. We conducted our study with 14 male master students of computer science at TU Vienna (participants P01 to P14), based on two different nontrivial implementation tasks. The first task was to develop a parallel computing implementation of a genetic algorithm (T1). The second task required the participants to implement a service that executes JUnit test cases on demand (T2). Both tasks required solutions that were elastic, that is, participants needed to demonstrate that their solutions were able to react to changes in load dynamically and automatically by

Table I. Relevant Background for Each Participant of the Study

ID	Phase	Java Exp.	Cloud Exp.	JCS/OS	OS	JCS/EC2	Beanstalk
P01	Phase 1	+	+	T1	T2	–	–
P02	Phase 1	+	+	T1	T2	–	–
P03	Phase 1	∼	∼	T2	T1	–	–
P04	Phase 1	-	-	T1	T2	–	–
P05	Phase 1	∼	-	T2	T1	–	–
P06	Phase 1	+	-	T2	T1	–	–
P07	Phase 1	+	+	T2	T1	–	–
P08	Phase 1	+	∼	T2	T1	–	–
P09	Phase 1	+	∼	T2	T1	–	–
P10	Phase 2	+	+	–	–	T2	T1
P11	Phase 2	+	∼	–	–	T1	T2
P12	Phase 2	+	-	–	–	T1	T2
P13	Phase 2	+	∼	–	–	T2	T1
P14	Phase 2	+	+	–	–	T2	–

To preserve anonymity, we classify the self-reported background of participants related to their Java or cloud experience into three groups: relevant work experience (+), some experience (∼), or close to no experience (-). The last four columns indicate whether the participant submitted solutions for JCloudScale running on top of OpenStack, OpenStack directly, JCloudScale running on top of EC2, or AWS Elastic Beanstalk, as well as which tasks the participant solved.

scaling up and down in the cloud. Both T1 and T2 required roughly one to two developer weeks of effort (assuming that the respective participant did not have any particular prior experience with the used technologies).

The study ran in two phases. In Phase 1, we compared using JCloudScale on top of OpenStack with programming directly via the OpenStack API, without any specific middleware support. This phase reflected a typical private cloud [Dillon et al. 2010] use case of JCloudScale. In Phase 2, we compare JCloudScale on top of Amazon EC2 using Amazon Elastic Beanstalk. This reflects a common public cloud usage of the framework. In both study phases, we asked participating developers to build solutions for both tasks using JCloudScale and the respective comparison technology, and compare the developer experience based on quantitative and qualitative factors. Our choice of OpenStack and Amazon EC2 was motivated by the fact that those two platforms currently form the most well-known, as well as most widely used, private and public IaaS systems. Especially EC2 has established a quasi-standard in terms of API support, which many other IaaS systems also adhere to. Consequently, we chose Elastic Beanstalk as the PaaS system as we wanted to stay within the same cloud ecosystem to keep results as comparable as possible.

Phase 1 of the study lasted 2mo. We initially presented JCloudScale and the comparison technologies to the participants, and randomly assigned which of the tools each participant should be using for T1. Participants then had 1mo of time to submit a working solution to the task along with a short report, after which they could start working on T2 with the remaining technology. Similar to T1, participants were given 1mo to submit a solution and a short report. Based on the lessons learned from Phase 1, we slightly clarified and improved the task descriptions and gave participants more time (1.5mo per task) for Phase 2. Other than that, Phase 2 was executed identically to Phase 1. Table I summarizes the relevant background for each participant of the study.

For the OpenStack-related implementations, we used a private cloud system hosted at TU Vienna. This OpenStack instance consists of 12 dedicated Dell blade servers with 2 Intel Xeon E5620 CPUs (2.4GHz Quad Cores) each, and 32GB RAM, running on OpenStack Folsom (release 2012.2.4). All servers are redundantly connected through 3Gb switches. For the study, each participant was alloted a quota of up to 8 small cloud

Table II. Solutions Sizes in Lines of Code

	Phase 1					Phase 2				
	JCS/OS		OS			JCS/EC2		Beanstalk		
	$\tilde{A}$	σ_A	$\tilde{B}$	σ_B	$\tilde{A}-\tilde{B}$	$\tilde{C}$	σ_C	$\tilde{D}$	σ_D	$\tilde{C}-\tilde{D}$
T1										
Business logics	200	176	552	215	−352	388	152	825	947	−437
Cloud management	100	112	180	86	−80	163	24	676	742	−513
Other code	170	157	286	226	−116	1590	1203	897	1127	693
Entire application	400	416	1050	434	−650	2141	1331	2790	2660	−649
T2										
Business logics	450	292	375	669	75	800	434	208	280	592
Cloud management	100	48	250	364	−150	118	745	223	38	−105
Other code	325	213	300	297	25	140	2240	1290	972	−1150
Entire application	1025	461	1500	901	−475	1000	3328	2184	968	−1184

$\tilde{A}$ to $\tilde{D}$ represent the median size of solutions, while σ_A to σ_D indicate standard deviations.

instances (1 virtual CPU, and 512MB of RAM), which they could use to implement and test their solutions. For the AWS-related implementations, participants were assigned an AWS account with sufficient credit to cover their implementation and testing with no particular limitations. Our companion Web site contains the task descriptions that we used for our study, as well as the questionnaires and anonymized participant reports.

4.1.2. Comparison of Development Efforts (RQ1). RQ1 asked whether JCloudScale makes it easier and faster to build elastic cloud applications. To this end, we asked participants to report on the size of their solutions (in lines of code, without comments and blank lines). The results are summarized in Table II. It can be seen that using JCloudScale generally reduces the total source code size of applications. Most important, the size of the entire application was substantially smaller when using JCloudScale than in the comparison cases. Going into the study, we expected JCloudScale to mostly reduce the amount of code necessary for interacting with the cloud. However, our results indicate that using JCloudScale also often reduced the amount of code of the application business logics, as well as assorted other code (e.g., data structures). When investigating these results, we found that participants considered many of the tasks that JCloudScale takes over as "business logics" when building the elastic application on top of OpenStack or Elastic Beanstalk. To give one example, many participants counted code related to performance monitoring towards "business logics." Note that, due to the open nature of our study tasks, the standard deviations are all rather large (i.e., solutions using all technologies varied widely in size). Further, the large difference in T1 sizes (for JCloudScale on top of OpenStack and EC2) between Phase 1 and Phase 2 solutions can be explained by clarifications in the task descriptions. In Phase 1, some formulations in the tasks led to much simpler implementations, while our requirements were formulated much more unambiguously in Phase 2, leading to more complex (and larger) submissions. Hence, we caution the reader not to compare results from Phase 1 with those from Phase 2.

However, looking at lines of code alone is not sufficient to validate our hypothesis, as it would be possible that the JCloudScale solutions, while being more compact, are also more complicated (thus take longer to implement). That is why we also asked participants to report on the time they spent working on their solutions. The results are compiled in Table III. We have classified work hours into a number of different activities: initially learning the technology, coding, testing and bug fixing, and other activities (e.g., building OpenStack cloud images). Our results indicate that the initial learning curve for JCloudScale is lower than for working with OpenStack directly.

Table III. Time Spent in Full Hours

	Phase 1					Phase 2				
	JCS/OS		OS			JCS/EC2		Beanstalk		
	$\tilde{A}$	σ_A	$\tilde{B}$	σ_B	$\tilde{A}-\tilde{B}$	$\tilde{C}$	σ_C	$\tilde{D}$	σ_D	$\tilde{C}-\tilde{D}$
T1										
Tool learning	7	2	12	5	−5	28	18	16	1	12
coding	4	10	30	17	−26	42	25	54	23	−12
Bug fixing	7	7	18	12	−11	14	8	20	14	−6
Other activities	13	9	14	14	−1	5	0	6	6	−1
Entire application	31	25	76	33	−45	127	25	121	36	6
T2										
Tool Learning	8	6	2	1	6	15	10	23	18	−8
Coding	30	11	25	17	5	36	5	30	14	6
Bug fixing	10	11	10	7	0	16	16	5	0	11
Other activities	7	4	11	10	−4	5	0	9	9	−4
Entire application	62	13	46	17	16	125	40	102	13	23

$\tilde{A}$ to $\tilde{D}$ represent the median time spent, while σ_A to σ_D indicate standard deviations.

However, in comparison with Elastic Beanstalk, some participants reported equal or even more complexity of JCloudScale, mainly because less information about JCloudScale is available on the Internet. For coding, JCloudScale appeared to be a much faster tool for participants who had at least some prior experience with cloud computing. Generally, for task T2, JCloudScale proved troublesome for some participants. In this task, JCloudScale generally did not improve productivity over either OpenStack or Elastic Beanstalk. Further research will be required to analyze why the results between tasks T1 and T2 vary in this regard.

We also analyzed qualitative feedback by the participants in their reports. Multiple developers have reported that they felt more productive when using JCloudScale. For instance, P01 has stated that *"the coolest thing about* JCloudScale *is the reduction of development effort necessary, to host applications in the cloud (...) [there] are a lot of things you do not have to care about in detail."* P03 also concluded that using JCloudScale *"went a lot smoother than [using OpenStack directly]."* P07 also seemed to share this sentiment and stated that *"[After resolving initial problems] the rest of the project was without big problems and I was able to be very productive in coding the solution."* In comparison to Elastic Beanstalk, participants indicated that the core idea behind JCloudScale is easier to grasp for starting cloud developers than the one behind modern PaaS systems. For example, P13 indicated that *"the API is easier to understand and more intuitive to use. Also, it fits more into a Java-like programming model, instead of the weird request-based approach of the Amazon API."* However, some participants noted that the fact that Elastic Beanstalk is based on common technology also appeals to them. For instance, P10 specified that *"[In the case of Elastic Beanstalk,] Well-known technology is the basis for everything (Tomcat/Servlet)."* Hence, the participant argued that this allows developers who are already familiar with these platforms to be productive sooner.

Summarizing our study results regarding RQ1, our data suggests that JCloudScale allows for higher developer productivity for task T1. For T2, JCloudScale solutions are more compact, but it took participants longer to implement them. More research is required to substantiate the underlying reasons for this discrepancy.

4.1.3. Comparison of Developer-Perceived Quality (RQ2). In order to answer RQ2, we were interested in the participant's subjective evaluation of the used technologies. We asked them to rate the technologies along a number of dimensions from 1 (very good) to

Table IV. Subjective Participant Ratings from 1 (Very Good) to 5 (Insufficient)

	Phase 1					Phase 2				
	JCS/OS		OS			JCS/EC2		Beanstalk		
	$\tilde{A}$	σ_A	$\tilde{B}$	σ_B	$\tilde{A}-\tilde{B}$	$\tilde{C}$	σ_C	$\tilde{D}$	σ_D	$\tilde{C}-\tilde{D}$
T1										
Simplicity	3	0.6	3	1.2	0	2	0	2	1.4	0
Debugging	3	1.5	3	1	0	4	0	3.5	0.7	0.5
Development process	4	1.7	3.5	0.5	0.5	2	1.4	3	0	−1
Stability	2	1.4	2	0.8	0	2	1.4	1.5	0.7	0.5
Overall	3	0.6	3	0.8	0	2	0	2	1.4	0
T2										
Simplicity	2	0.4	3	1.4	−1	2	1.5	3	1.4	−1
Debugging	2	0.7	4	1.4	−2	4	0	4	0	0
Development process	2	0.6	3	1.4	−1	2	0	2.5	0.7	−0.5
Stability	2	1.5	1	0	1	3	0.5	2.5	0.7	0.5
Overall	2	0.4	3	0	−1	3	0.5	3	1.4	0

$\tilde{A}$ to $\tilde{D}$ represent the median ratings, while σ_A to σ_D indicate standard deviations.

5 (insufficient). We report on the dimensions "simplicity" (how easy is it to use the tool?), "debugging" (how easy is testing and debugging the application?), "development process" (does the technology imply an awkward development process?), and "stability" (how often do unexpected errors occur?). A summary of our results is shown in Table IV.

For T1, participants rated all used technologies similarly, while JCloudScale was appreciated more for T2. However, JCloudScale was rated worse than the comparison technologies mainly in terms of "stability." This is not a surprise, as JCloudScale still is a research prototype in a relatively early development stage. Participants mentioned multiple stability-related issues in their reports (e.g., P10 mentioned that *"When deploying many cloud objects to one host, there were behaviors which were hard to reason about"*). Further, some technical implementation decisions in JCloudScale were not appreciated by our study participants. To give an example, P11 noted that *"It is confusing in the configuration that the field AMI-ID actually expects the AMI-Name, not the ID."* In contrast, JCloudScale has been rated slightly better in terms of simplicity and ease of use, especially for T2. For example, participant P09 claimed that "JCloudScale *is the clear winner in ease of use. If you quickly want to just throw some objects in the cloud, it's the clear choice."* Similarly, P12 reported *"[*JCloudScale *is] programmer friendly. All procedure is more low level, and as a programmer there are more things to tune and adjust."* In terms of debugging features, all used technologies were not rated overly well. JCloudScale was generally perceived slightly better (arguably due to its local development environment), but realistically all compared systems are currently deemed too hard to debug if something goes wrong. Finally, in terms of the associated development process, JCloudScale is generally valued highly, with the exception of T1 and JCloudScale on OpenStack. We assume that this is a statistical artifact, as the development process of JCloudScale is judged well in all other cases. Concretely, P01 stated that with JCloudScale, *"You are able to get application into the cloud really fast. You are not forced to take care about a lot of cloud-specific issues."*

Independently of the subjective ratings, multiple participants stated that they valued the flexibility that the JCloudScale concept brought over Elastic Beanstalk. Particularly, P11 noted that *"[*JCloudScale *provides] more flexibility. The developer can decide when to deploy hosts, on which host an object gets deployed, when to destroy a host, etc."* Additionally, participants favored the monitoring event engine of JCloudScale for

performance tracking over the respective features of the PaaS system. For example, P12 specified as a JCloudScale advantage that *"programmatic usage of different events with a powerful event correlation framework [is] in combination with listeners extremely powerful."*

Concluding our discussion regarding RQ2, we note that JCloudScale has some way to go before it is ready for industrial usage. The general concepts of the tool are valued by developers but, currently, technical issues and lack of documentation and technical support make it hard for developers to fully appreciate the power of the JCloudScale model. One aspect that needs more work is how developers define the scaling behavior of their application. Both tasks in our study required the participants to define nontrivial scaling policies, for example, in order to optimally schedule genetic algorithm executions to cloud resources, which most participants felt unable to do with the current API provided by JCloudScale. Overall, in comparison to working directly on OpenStack, many participants preferred JCloudScale, but compared to a mature PaaS platform, AWS Elastic Beanstalk still seems slightly preferable to many. However, it should be noted that JCloudScale still opens up use cases for which using Beanstalk is not an option, for instance, for deploying applications in a private or hybrid cloud [Leitner et al. 2013].

4.2. Runtime Overhead Measurements (RQ3)

Finally, we investigated whether the improved convenience of JCloudScale is paid for with significantly reduced application performance. Therefore, the main goal of these experiments was to compare the performance of the same application built on top of JCloudScale and using an IaaS platform (OpenStack or EC2) directly.

4.2.1. Experiment Setup. To achieve this, we built a simple sample application (inspired by T2 from the user study) on top of Amazon EC2 and our private OpenStack cloud. For OpenStack, we used the same configuration already explained for RQ1 and RQ2. For EC2, we deployed our application in the `eu-west-1` region and used instances of size `t1.micro`. Initial experiments with Elastic Beanstalk have shown that there is no substantial performance difference between EC2 and Elastic Beanstalk. Hence, we omit Beanstalk in our discussions here. The application JavaScript Testing-as-a-Service (JSTaaS) provides testing of JavaScript applications as a cloud service (inspired by the real-life service provided by the New York–based startup Codeship[2]). Clients register with the service, which triggers JSTaaS to periodically launch the client's registered test suites. Results of the test runs are obtained by the client when they are available. Tests vary widely in the load that they generate on the servers; clients are billed according to this load.

Second, we also implemented the same application using JCloudScale. As the main goal was to calculate the overhead introduced by the JCloudScale, we designed both implementations to have the same behavior and reuse as much business logics code as possible. In addition, to simplify our setup, focus on execution performance evaluation, and to avoid major platform-dependent side effects, we limited ourselves to a scenario in which the number of available cloud hosts is static. The source code of both applications is available online as part of our companion material.

All four solutions (directly on OpenStack, directly on EC2, and using JCloudScale on both OpenStack and EC2) follow a simple master–slave pattern: a single node (the master) receives tests through a SOAP-based Web service and schedules them over the set of available worker nodes. All solutions were tested with a test setup that consisted of 40 identical parallelizable long-running test suites (each suite execution

[2]https://codeship.com.

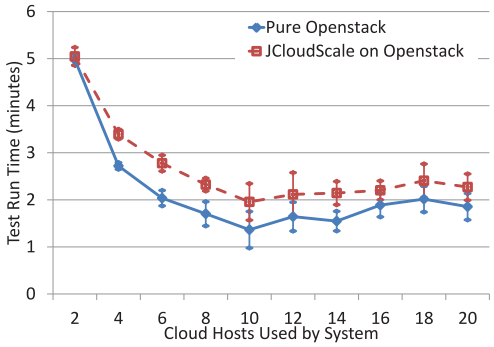

Fig. 5. Execution on OpenStack platform.

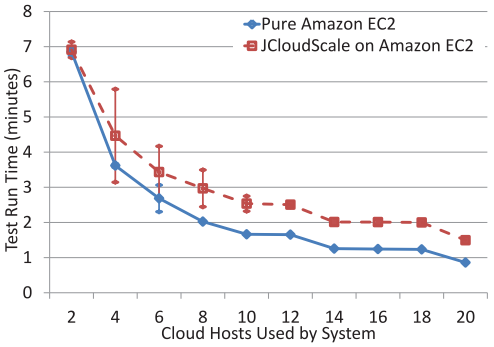

Fig. 6. Execution on EC2 platform.

takes around 30s in our OpenStack cloud environment), scheduled evenly over the set of available cloud machines. Each test suite consisted of a set of dummy JavaScript tests calculating Fibonacci numbers. During the evaluation, we measured the total time of an entire invocation of the service (i.e., how long a test request takes end to end, including scheduling, data transmission, result collection, and so on). A single experiment run consisted of 10 identical invocations of the testing Web service, each time with a different number of CHs (ranging from 2 to 20 CHs). In all experiment setups, our evaluation shared a physical cloud environment with other tenants, as would be the case in real-life usage. To reduce the performance impact of other tenants' activities, we repeated each experiment 10 times over the course of a day.

4.2.2. Experiment Results. Figure 5 and Figure 6 show the median total execution time for different numbers of hosts, including error bars demarkating standard deviations. In general, both applications show similar behavior in each environment, meaning that both approaches are feasible and have similar parallelizing capabilities with minor differences in overhead. In both environments, there is an overhead of JCloudScale that is proportional to the amount of used CHs and approximately equal to 2s to 3s per introduced host for multiple minutes evaluation application. This overhead may be significant for performance-critical production applications, but we believe that it is a reasonable price to pay in the current development stage of the JCloudScale middleware. Furthermore, how large this overhead is depends largely on how much the target application is required to communicate with the CHs. The main reason for this is that messaging in JCloudScale is more expensive in comparison to a pure OpenStack or EC2 solution, as JCloudScale appends some platform-specific metadata to remote invocations (e.g., which CH or CO to address, which code to run, and so forth). The evaluation application required a substantial amount of coordination between target application and CHs, hence we have reason to believe that these overhead measurements are relatively conservative. However, detailed investigation (and, subsequently, reduction) of the overhead introduced by JCloudScale is planned for future releases of the middleware.

Another important issue that is visible from Figure 5 and Figure 6 is cloud performance stability and predictability. With an increasing number of hosts, the total execution time is expected to monotonously decrease, up to a limit when the overhead of parallelization is larger than the gain of having more processors available. This happens in case of Amazon EC2. However, starting with 10 used hosts in OpenStack, the overall application execution time remains almost constant or even increases. In our case, this is mainly caused by the limited size of our private cloud. In our system,

starting with 10 hosts, physical machines start to get overutilized, and virtual machines start to compete for resources (e.g., CPU or disk IO).

4.3. Threats to Validity

The major threat to (internal) validity, which the results relating to RQ1 and RQ2 indicate, are that the small sample size of 14 study participants, along with relatively open problem statements, does not allow us to establish statistical significance. However, due to the reports we received from participants, as well as due to comparing the solutions themselves, we are convinced that our results showing that JCloudScale lets developers build cloud applications more efficiently was not a coincidence. Further, our participants were aware that JCloudScale is our own system. Hence, there is a chance that our participants gave their reports a more positive spin. However, given that all reports contained negative aspects for all evaluated frameworks, we are confident that most participants reported truthfully. There is also the possibility that our study design, which required all participants to work on two projects, skewed the results, as it can be expected that participants learned from the first project for the second one. This has been mitigated by letting a subset of the participants work with JCloudScale first and the remainder start with one of the comparison frameworks. Finally, there is the threat that JCloudScale solutions, while being implemented more efficiently, are also of lower quality. We have mitigated this threat by (partial) automatic testing of solutions against defined requirements, as well as by manual inspection and comparison of solutions. In terms of external validity, it is possible that the two example projects we chose for our study are not representative of real-world applications. However, we argue that this is unlikely, as the projects have specifically been chosen based on real-life examples that the authors of this article are aware of or had to build themselves in the past. Another threat to external validity is that the participants of our study are all students at TU Vienna. While most have some practical real-life experience in application development, none can be considered senior developers.

In terms of RQ3, the major threat to external validity is that the application we used to measure overhead is necessarily simplified and not guaranteed to be representative. Real applications are hard to replicate in exactly the same way on different systems, hence comparative measurements among such systems are always unfair. To minimize this risk, we have taken care to preserve what we consider to be core features of cloud applications even in the simplified measurement application. Similarly, there is a threat to the generalizability of our study related to RQ1 and RQ2. It is possible that the results of our study would have been substantially different if we had chosen different comparison systems, for example, Microsoft Azure or Google Appengine. However, we consider this threat small, as at the time we executed the study, the core features provided by most IaaS and PaaS providers were comparable to the extent required by our study.

5. RELATED WORK

We now put the JCloudScale framework into context of the larger distributed and cloud-computing ecosystem. As the scope of JCloudScale is rather wide, there is a large number of systems that are related to parts of the JCloudScale scope. We consider the main dimensions to compare frameworks across to be the following: (1) to what extent they transparently handle remoting and elasticity, (2) whether they handle scaling up and down, (3) how easy it is to locally test and debug applications, (4) whether the system restricts what kinds of applications can be built, (5) whether the system handles cloud virtual machines, and (6) whether the system is bound to one specific

cloud provider. We provide a high-level comparison of various systems along these dimensions in Table V.

First, JCloudScale can be compared to traditional distributed object middleware [Emmerich 2000], such as Java RMI or EJB. These systems provide transparent remoting features, but clearly do not provide any support for cloud specifics, such as VM management. It can be argued that EJB provides some amount of transparent elasticity, as EJB containers can be clustered. However, it is not easy to scale such clusters up and down. A recent work [Jayaram 2013] has introduced the idea of Elastic Remote Methods, which extends Java RMI with cloud-specific features. This is comparable in goals to our contribution. However, the technical approach is quite different. Aneka [Calheiros et al. 2012; Vecchiola et al. 2008], a well-known .NET-based cloud framework, is a special case of a cloud-computing middleware that also exhibits a number of characteristics of a PaaS system. We argue that Aneka's abstraction of remoting is not perfect, as developers still need to be intimately aware of the underlying distributed processing. To the best of our knowledge, Aneka does not automatically scale systems, and provides no local testing environment. Simao et al. [2011] also presented an approach that can be considered related to JCloudScale. Their A^2-VM framework schedules Java threads over a compute cluster.

Second, as already argued in Section 4, many of JCloudScale's features are comparable to common PaaS systems (e.g., Google Appengine, Amazon Elastic Beanstalk, or Heroku). All of these provide transparent remoting and elasticity, and take over virtual machine management for the user. However, they usually tie the user tightly to one specific cloud provider. Support for local testing is limited, although most providers currently have at least some tooling or emulators available for download.

In addition to these commercial PaaS systems, there are also multiple platforms coming out of a research setting. For instance, AppScale [Chohan et al. 2010; Krintz 2013] is an open-source implementation of the Google Appengine model. AppScale can also be deployed on any IaaS system, making it much more vendor-independent than other PaaS platforms. This is similar to the ConPaaS open-source platform [Pierre et al. 2011; Pierre and Stratan 2012], which originates from a European research project of the same name. ConPaaS follows a more service-oriented style, treating applications as collections of loosely coupled services.

In scientific literature, there are also a number of PaaS systems that are more geared towards data processing, for example, BOOM [Alvaro et al. 2010], Esc [Satzger et al. 2011], or Granules [Pallickara et al. 2009]. These systems are hard to compare with our work, as they generally operate in an entirely different fashion as compared to JCloudScale or the commercial PaaS operators. However, they typically support only a very restricted type of (data-driven) application model, and often do not actually interact with the cloud by themselves. This makes them cloud-provider independent, but also means that developers need to implement the actual elasticity-related features themselves.

Third, we need to compare JCloudScale to a number of cloud-computing related frameworks, which cover a part of the functionality provided by our middleware. JClouds is a Java library that abstracts from the heterogeneous APIs of different IaaS providers, and allows decoupling of Java applications from the IaaS system in which they operate. JCloudScale internally uses JClouds to interact with providers. However, by itself, JClouds does not provide any actual elasticity. Docker is a container framework geared towards bringing testability to cloud computing. Essentially, Docker has similar goals to the local test environment of JCloudScale.

JCloudScale also has some relation to the various cloud deployment models and systems that have recently been proposed in literature, for example, Cafe [Mietzner et al. 2009], MADCAT [Inzinger et al. 2014], or OpenTOSCA [Binz et al. 2013], which

Table V. High-Level Comparison of Distributed and Cloud Computing Systems

		Transparent remoting	Transparent elasticity	Local testing	Unrestricted architecture	Transparent VM management	Cloud provider independence
Remoting Frameworks							
Java RMI		yes	no	yes	yes	no	no
Enterprise Java Beans (EJB)		yes	partial	yes	yes	no	no
Elastic Remote Methods	[Jayaram 2013]	yes	yes	no	yes	yes	yes
Aneka	[Vecchiola et al. 2008] [Calheiros et al. 2012]	partial	no	no	yes	yes	yes
A^2-VM	[Simao et al. 2011]	partial	yes	no	yes	yes	yes
PaaS Systems							
Appengine		yes	yes	partial	no	yes	no
Amazon Elastic Beanstalk		yes	yes	no	no	yes	no
Heroku		yes	yes	partial	no	yes	no
AppScale	[Chohan et al. 2010] [Krintz 2013]	yes	yes	no	no	yes	yes
ConPaaS	[Pierre et al. 2011] [Pierre and Stratan 2012]	yes	yes	no	partial	yes	yes
BOOM	[Alvaro et al. 2010]	yes	yes	no	no	no	yes
Esc	[Satzger et al. 2011]	yes	yes	no	no	no	yes
Granules	[Pallickara et al. 2009]	yes	yes	yes	no	no	yes
Cloud Deployment & Test Frameworks							
JClouds		no	no	no	yes	no	yes
Docker		no	no	yes	yes	no	yes
Cafe	[Mietzner et al. 2009]	no	no	yes	no	no	yes
MADCAT	[Inzinger et al. 2014]	no	no	yes	no	no	yes
OpenTOSCA	[Binz et al. 2013]	no	partial	no	yes	yes	yes
JCloudScale		yes	partial	yes	yes	yes	yes

All systems are evaluated along different dimensions, and assigned "yes" (strong support), "no" (no real support), or 'partial" (some support). All evaluations are, to the best of the knowledge of the authors, based on tool documentations or publications.

is an open-source implementation of an upcoming OASIS standard. These systems do not typically cover elasticity by themselves (although TOSCA has partial support for auto-scaling groups), but they are usually independent of any concrete cloud provider.

By design, JCloudScale supports most of the characteristics that we discuss here. However, especially in comparison to PaaS systems, developers of JCloudScale applications are not entirely shielded from issues of scalability. Further, as the user study discussed in Section 4 has shown, the system still needs to improve how scaling policies are written to make building elastic systems easier for developers.

Item 4 in this list (Unrestricted Architecture) requires more discussion. Industrial PaaS systems (e.g., Appengine, Beanstalk, Heroku) are generally geared towards a very specific type of application (transaction-based Web applications). These systems assume that requests are (to a large extent) independent, and take very little time to process (Appengine, for instance, has a hard upper limit of 30s of processing time per request). This model is useful for many typical use cases in a Web context, for example, blogs or Web shops. However, developers aiming to build other kinds of applications (e.g., the JSTaaS example discussed in Section 4.2, banking solutions, video streaming platforms) have to switch for IaaS or struggle with the architectural and technical restrictions imposed by those PaaS systems. Other remoting and cloud frameworks (e.g., Java RMI or Docker) do not have such restrictions (e.g., Docker is useful for more or less arbitrary applications); however, these systems are also not concerned about providing automated scaling and elasticity. JCloudScale, as well as the related Aneka framework [Vecchiola et al. 2008] and the Elastic Remote Methods proposed by Jayaram [2013], strive for a middle ground. They do not inherently assume a specific, narrow type of application and can, in principle, be used to implement a wide range of elastic applications. However, they are most suitable for applications with durable request- or task-processing activities, such as video-audio encoding, Web-crawling, sentiment analysis, or image rendering. Additionally, JCloudScale provides significant benefits for applications that use cloud resources only to cover activity bursts [Leitner et al. 2013]. JCloudScale is less suitable for connection-oriented and latency-sensitive applications, such as streaming services or online games. Further, for big data–centric applications, JCloudScale is arguably less intuitive to use than state-of-the-art models (e.g., Hadoop or Spark SQL).

6. CONCLUSIONS

JCloudScale is a Java-based middleware that eases the development of elastic cloud applications on top of an IaaS cloud. JCloudScale follows a declarative approach based on Java annotations, which removes the need to actually adapt the business logics of the target application to use the middleware. Hence, JCloudScale support can easily be turned on and off for an application, leading to a flexible development process that clearly separates the implementation of target application business logics from implementing and tuning the scaling behavior.

We have introduced the core concepts behind JCloudScale, and presented an evaluation of the middleware based on a user study as well as using a case study application. Our results indicate that JCloudScale is well received among initial developers. Our results support our claim that the general JCloudScale model has advantages to both, working directly on top of an IaaS API or on an industrial PaaS system. However, further study is required to strengthen these claims, as the limited scale of our initial study was not sufficient to clear all doubts about the viability of the system. Further, there are also technical and conceptual issues that require further investigation. Most important, we have learned that implementing actually elastic applications is still cumbersome for developers, as getting the scaling policy right is still difficult for some developers. Additionally, we are currently testing JCloudScale as a tool to

cloud-migrate a number of existing standard systems, including Apache JMeter or the service composition engine JOpera [Pautasso and Alonso 2005]. Finally, we are investigating more powerful ways to handle faults in JCloudScale applications, in addition to the mechanisms introduced in Section 2.3. Concretely, we plan to investigate means to allow for replication of COs, which is currently not supported out of the box.

COMPANION MATERIAL

Additional material, including the tasks used in our study, as well as the anonymized participant feedback, is available on our companion website[3]. JCloudScale is available as an open-source project on GitHub[4].

REFERENCES

Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. 2010. Boom analytics: Exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*. ACM, New York, NY, 223–236.

Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A view of cloud computing. *Communications of the ACM* 53, 4, 50–58.

David E. Avison, Francis Lau, Michael D. Myers, and Peter Axel Nielsen. 1999. Action research. *Communications of the ACM* 42, 1, 94–97. DOI:http://dx.doi.org/10.1145/291469.291479

Cor-Paul Bezemer, Andy Zaidman, Bart Platzbeecker, Toine Hurkmans, and Aad 't Hart. 2010. Enabling multi-tenancy: An industrial experience report. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM'10)*. IEEE Computer Society, Washington, DC, 1–8. DOI:http://dx.doi.org/10.1109/ICSM.2010.5609735

Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak, and Sebastian Wagner. 2013. OpenTOSCA—A runtime for TOSCA-based cloud applications. In *Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC)*. 692–695.

Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. 2009. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computing Systems* 25, 6, 599–616.

Rodrigo N. Calheiros, Christian Vecchiola, Dileban Karunamoorthy, and Rajkumar Buyya. 2012. The Aneka platform and QoS-driven resource provisioning for elastic applications on hybrid Clouds. *Future Generation Computer Systems* 28, 6, 861–870. DOI:http://dx.doi.org/10.1016/j.future.2011.07.005

Navraj Chohan, Chris Bunch, Sydney Pang, Chandra Krintz, Nagy Mostafa, Sunil Soman, and Rich Wolski. 2010. AppScale: Scalable and open appengine application development and deployment. In *Cloud Computing*, Dimiter Avresky, Michel Diaz, Arndt Bode, Bruno Ciciani, and Eliezer Dekel (Eds.). *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering*, Vol. 34. Springer, Berlin, 57–70. DOI:http://dx.doi.org/10.1007/978-3-642-12636-9_4

J. Cito, P. Leitner, T. Fritz, and H. C. Gall. 2015. The making of cloud applications – An empirical study on software development for the cloud. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. To appear.

Tharam Dillon, Chen Wu, and Elizabeth Chang. 2010. Cloud computing: Issues and challenges. In *24th IEEE International Conference on Advanced Information Networking and Applications (AINA'10)*. IEEE Computer Society, Washington, DC, 27–33. DOI:http://dx.doi.org/10.1109/AINA.2010.187

Wolfgang Emmerich. 2000. *Engineering Distributed Objects*. John Wiley & Sons, New York, NY.

Stephane Genaud and Julien Gossa. 2011. Cost-wait trade-offs in client-side resource provisioning with elastic clouds. In *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing (CLOUD'11)*. IEEE Computer Society, Washington, DC, 1–8. DOI:http://dx.doi.org/10.1109/CLOUD.2011.23

Waldemar Hummer, Florian Rosenberg, Fabio Oliveira, and Tamar Eilam. 2013. Testing idempotence for infrastructure as code. In *ACM/IFIP/USENIX Middleware Conference*. 368–388.

Christian Inzinger, Stefan Nastic, Sanjin Sehic, Michael Vögler, Fei Li, and Schahram Dustdar. 2014. MADCAT: A methodology for architecture and deployment of cloud application topologies. In

[3]http://www.infosys.tuwien.ac.at/staff/phdschool/rstzab/papers/TOIT14/.

[4]https://github.com/xLeitix/jcloudscale.

Proceedings of the 8th International Symposium on Service Oriented System Engineering (SOSE). 13–22. DOI:http://dx.doi.org/10.1109/SOSE.2014.9

K. R. Jayaram. 2013. Elastic remote methods. In *Proceedings of Middleware 2013*, *Lecture Notes in Computer Science*, David Eyers and Karsten Schwan (Eds.), Vol. 8275. Springer, Berlin, 143–162. DOI:http://dx.doi.org/10.1007/978-3-642-45065-5_8

Jeffrey O. Kephart and David M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1, 41–50. DOI:http://dx.doi.org/10.1109/MC.2003.1160055

Chandra Krintz. 2013. The Appscale cloud platform: Enabling portable, scalable web application deployment. *IEEE Internet Computing* 17, 2, 72–75. DOI:http://dx.doi.org/10.1109/MIC.2013.38

Philipp Leitner, Christian Inzinger, Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. 2012. Application-level performance monitoring of cloud services based on the complex event processing paradigm. In *Proceedings of the 2012 5th IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. 1–8. DOI:http://dx.doi.org/10.1109/SOCA.2012.6449437

Philipp Leitner, Zabolotnyi Rostyslav, Alessio Gambi, and Schahram Dustdar. 2013. A framework and middleware for application-level cloud bursting on top of infrastructure-as-a-service clouds. In *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing (UCC'13)*. IEEE Computer Society, Washington, DC, 163–170. DOI:http://dx.doi.org/10.1109/UCC.2013.39

Philipp Leitner, Benjamin Satzger, Waldemar Hummer, Christian Inzinger, and Schahram Dustdar. 2012. CloudScale: A novel middleware for building transparently scaling cloud applications. In *27th ACM Symposium on Applied Computing (SAC'12)*. 434–440. DOI:http://dx.doi.org/10.1145/2245276.2245360

David Luckham. 2002. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Professional.

Ralph Mietzner, Tobias Unger, and Frank Leymann. 2009. Cafe: A generic configurable customizable composite cloud application framework. In *On the Move to Meaningful Internet Systems (OTM'09)*, Robert Meersman, Tharam Dillon, and Pilar Herrero (Eds.). Vol. 5870. Springer, Berlin, 357–364. http://dx.doi.org/10.1007/978-3-642-05148-7_24

Shrideep Pallickara, Jaliya Ekanayake, and Geoffrey Fox. 2009. Granules: A lightweight streaming runtime for cloud computing with support for map-reduce. In *Proceedings of the IEEE International Conference on Cluster Computing and Workshops (CLUSTER'09)*. IEEE, 1–10.

Cesare Pautasso and Gustavo Alonso. 2005. JOpera: A toolkit for efficient visual composition of web services. *International Journal of Electronic Commerce* 9, 2, 107–141. http://dl.acm.org/citation.cfm?id=1278095.1278101

Guillaume Pierre, Ismail El Helw, Corina Stratan, Ana Oprescu, Thilo Kielmann, Thorsten Schütt, Jan Stender, Matej Artač, and Aleš Černivec. 2011. ConPaaS: An integrated runtime environment for elastic cloud applications. In *Proceedings of the Workshop, Posters and Demos Track (Middleware'11)*. ACM, New York, NY, Article 5, 2 pages. DOI:http://dx.doi.org/10.1145/2088960.2088965

Guillaume Pierre and Corina Stratan. 2012. ConPaaS: A platform for hosting elastic cloud applications. *IEEE Internet Computing* 16, 5, 88–92. DOI:http://dx.doi.org/10.1109/MIC.2012.105

Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. 2011. Esc: Towards an elastic stream computing platform for the cloud. In *IEEE 5th International Conference on Cloud Computing (CLOUD'11)*. 348–355. DOI:http://dx.doi.org/10.1109/CLOUD.2011.27

Jose Simao, Joao Lemos, and Luis Veiga. 2011. A2-VM: A cooperative java VM with support for resource-awareness and cluster-wide thread scheduling. In *Proceedings of the 19th International Conference on Cooperative Information Systems (CoopIS'11)*.

Borja Sotomayor, Rubén S. Montero, Ignacio M. Llorente, and Ian Foster. 2009. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing* 13, 5, 14–22. DOI:http://dx.doi.org/10.1109/MIC.2009.119

Christian Vecchiola, Xingchen Chu, and Rajkumar Buyya. 2008. Aneka: A software platform for .NET based cloud computing. In *Proceedings of the High Performance Computing Workshop*. 267–295.

Rostyslav Zabolotnyi, Philipp Leitner, and Schahram Dustdar. 2013. Dynamic program code distribution in infrastructure-as-a-service clouds. In *Proceedings of the 5th International Workshop on Principles of Engineering Service-Oriented Systems (PESOS'13), co-located with ICSE 2013*.

Received September 2014; revised April 2015; accepted June 2015