Vienna University of Technology
Information Systems Institute
Distributed Systems Group

CloudScale: Efficiently Implementing Elastic Applications for Infrastructure-as-a-Service Clouds

Under Review for Publication in ICSE 2014

P. Leitner, Z. Rostyslav, W. Hummer, C. Inzinger, S. Dustdar
leitner@infosys.tuwien.ac.at

TUV-1841-2013-1 9/17/13

The Infrastructure-as-a-Service (IaaS) model of cloud computing is a promising approach towards building elastically scaling systems. Unfortunately, building such applications today is a complex, repetitive and error-prone endeavor, as IaaS does not provide any abstraction on top of naked virtual machines. Hence, all functionality related to elasticity needs to be implemented anew for each application. In this paper, we present CloudScale, a Java-based middleware that supports building elastic applications on top of a public or private IaaS cloud. CloudScale allows to easily bring applications to the cloud, with minimal changes to the application code. We discuss the general architecture of the middleware as well as its technical features, and evaluate our system with regard to both, user acceptance (based on a small-scale user study) and performance overhead. Our results indicate that CloudScale indeed exhibits many attractive advantages in comparison to standard IaaS tooling. CloudScale is available as open source software from Google Code.

Keywords: Cloud Computing, Middleware, Elastic Applications, Java, CloudScale

©2013, Distributed Systems Group, Vienna University of Technology

Argentinierstr. 8/184-1
A-1040 Vienna, Austria
phone: +43 1 58801-18402
fax: +43 1 58801-18491
http://www.infosys.tuwien.ac.at/

CloudScale: Efficiently Implementing Elastic Applications for Infrastructure-as-a-Service Clouds

Philipp Leitner
Distributed Systems Group
Vienna University of Technology
Argentinierstrasse 8/184-1,
1040 Vienna, Austria
leitner@infosys.tuwien.ac.at

Zabolotnyi Rostyslav
Distributed Systems Group
Vienna University of Technology
Argentinierstrasse 8/184-1,
1040 Vienna, Austria
rst@infosys.tuwien.ac.at

Waldemar Hummer
Distributed Systems Group
Vienna University of Technology
Argentinierstrasse 8/184-1,
1040 Vienna, Austria
hummer@infosys.tuwien.ac.at

Christian Inzinger
Distributed Systems Group
Vienna University of Technology
Argentinierstrasse 8/184-1,
1040 Vienna, Austria
inzinger@infosys.tuwien.ac.at

Schahram Dustdar
Distributed Systems Group
Vienna University of Technology
Argentinierstrasse 8/184-1,
1040 Vienna, Austria
sd@dsg.tuwien.ac.at

ABSTRACT

The Infrastructure-as-a-Service (IaaS) model of cloud computing is a promising approach towards building elastically scaling systems. Unfortunately, building such applications today is a complex, repetitive and error-prone endeavor, as IaaS does not provide any abstraction on top of naked virtual machines. Hence, all functionality related to elasticity needs to be implemented anew for each application. In this paper, we present CloudScale, a Java-based middleware that supports building elastic applications on top of a public or private IaaS cloud. CloudScale allows to easily bring applications to the cloud, with minimal changes to the application code. We discuss the general architecture of the middleware as well as its technical features, and evaluate our system with regard to both, user acceptance (based on a small-scale user study) and performance overhead. Our results indicate that CloudScale indeed exhibits many attractive advantages in comparison to standard IaaS tooling. CloudScale is available as open source software from Google Code.

Categories and Subject Descriptors

D.2.2.c [**Software Engineering**]: Distributed/Internet based software engineering tools and techniques; D.2.0.c [**Software Engineering**]: Software Engineering for Internet projects

General Terms

Languages, Experimentation, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

Keywords

Cloud Computing, Middleware, Programming, CloudScale

1. INTRODUCTION

In recent years, the cloud computing paradigm [8] has provoked a significant push towards more flexible provisioning of IT resources, including computing power, storage and networking capabilities. Besides economic factors (e.g., pay-as-you-go pricing), 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 (by virtue of operating close to optimal resource utilization levels), while still providing the expected level of application performance.

Elastic applications are typically built using either the IaaS (Infrastructure-as-a-Service) or the PaaS (Platform-as-a-Service) paradigm [5]. 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 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[1] or Google' AppEngine[2], GAE) come with significant disadvantages, which render this option infeasible for many users. These problems include: (1) strong vendor lock-in [13, 20], as one is typically required to program against a proprietary API; (2) little control over the elasticity behavior or the application (e.g., users have very little influence on when to scale up and down); (3) no root access to the virtual servers running the actual application

[1]http://www.windowsazure.com/
[2]https://developers.google.com/appengine/

code; and (4) little support for building applications that do not follow the basic architectural patterns assumed by the PaaS offering (e.g., Apache Tomcat based web applications in case of GAE). 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 paper, we introduce CloudScale, a Java-based middleware that eases the task of building elastic applications. Similar to PaaS, CloudScale takes over virtual machine management, application monitoring, load balancing, and code distribution. However, given that CloudScale is a client-side middleware instead of a complete hosting environment, users retain full control over the behavior of their application. Furthermore, CloudScale supports a wide range of different applications. CloudScale applications run on top of any IaaS cloud, making CloudScale a viable solution to implement applications for private or hybrid cloud settings [3,30]. In summary, we claim that the CloudScale model is a promising compromise between IaaS and PaaS, combining many advantages of both worlds.

The main contributions of this paper are two-fold. Firstly, we introduce the most relevant features of the CloudScale middleware, including the development process associated with the middleware. This contribution is in extension of our initial work in [23]. Secondly, we evaluate CloudScale with regard to both, runtime performance impact, as well as development productivity and user acceptance. The latter point is evaluated based on a small-scale user study, which compares applications built using CloudScale with applications built directly on top of an OpenStack[3] based private cloud.

The rest of this paper is structured as follows. Section 2 motivates the need for a system like CloudScale based on an illustrative example. In Section 3, we describe the basic CloudScale architecture, which we follow up with an in-depth discussion of specific elasticity-related features in Section 4. Section 5 gives an implementation overview of the middleware. This implementation forms the basis for the empirical evaluation in Section 6. Section 7 surveys related work, and, finally, Section 8 concludes the paper with an outlook on open issues.

2. ILLUSTRATIVE EXAMPLE

Consider JSTaaS ("JavaScript Testing-as-a-Service"), the product of an imaginary Web startup. JSTaaS provides testing of JavaScript applications as a cloud service. Clients register with the service, which triggers JSTaaS to periodically launch this client's registered test suites. Results of test runs are stored in a database, which can be queried by the client. Tests vary widely in the load that they generate on the servers, and clients are billed according to this load.

As the startup has limited initial funding, it has been decided that the initial version of JSTaaS should be deployed as a Java-based service on Amazon's EC2 public IaaS cloud[4], so that infrastructure costs will only grow in line with actual demand. Furthermore, to save costs, virtual machines, which will be used to launch actual tests, should be utilized to the highest degree possible. This means that tests should be co-located on the same virtual machines as far as possible, and idle machines should be released if they are not required any longer. To this end, the core of JSTaaS needs to continuously monitor the utilization of all hosts, as well as the execution time of tests, in order to decide which hosts to keep online and which to tear down.

[3]http://www.openstack.org/

[4]http://aws.amazon.com/

Using standard tools, this application is not trivial to implement. Developers need to split the application into task manager, load balancer and workers to execute the tests, setup virtual machines, install the respective application components on these virtual machines and, at runtime, monitor to make sure that the application is not over- or under-provisioned. Tools such as AWS Elastic Beanstalk (deployment) or CloudWatch (monitoring) can be used to ease these tasks to some extent. However, even using these advanced tools, cloud deployment quickly becomes a labor-intensive chore. Hence, the startup decides to use the CloudScale middleware to build JSTaaS. The startup hopes that CloudScale will help its few developers to efficiently write this elastic application, and devise good scaling and scheduling policies. Furthermore, the integrated event-based monitoring of CloudScale will also be useful for customer billing. Finally, building their application on top of the CloudScale abstraction allows the startup to easily migrate their application, e.g., to an OpenStack based private cloud, should they decide to move away from Amazon EC2 in the future.

3. THE CLOUDSCALE MIDDLEWARE

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

3.1 Basic Notions

CloudScale is a Java-based middleware for building elastic IaaS applications. The ultimate aim of CloudScale is to facilitate developers to implement cloud applications (in the following referred to as *target applications*) as local, multi-threaded 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, CloudScale appears as an additional library (e.g., a Maven dependency) plus a post-compilation build step of an otherwise regular Java application. This puts CloudScale 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 CloudScale are *cloud objects* (COs). COs are object instances which 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 CloudScale 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 post-compilation 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

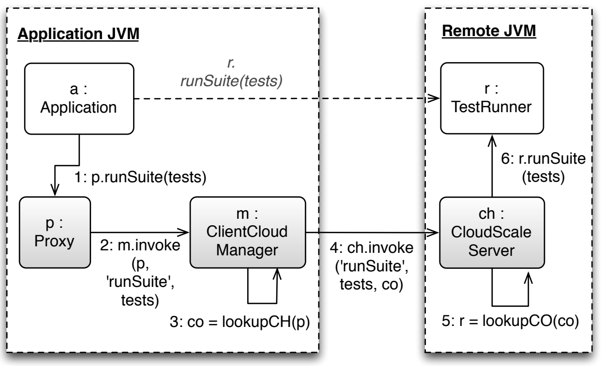

Figure 1: Basic Interaction with Cloud Objects

```
@CloudObject
public class TestRunner {

  @CloudGlobal
  private static String testRunnerName;

  @DataSource(name = "couchdb")
  private DataStore datastore;

  @EventSink
  private EventSink eventsink;

  public TestResult runSuite(
    @ByValueParameter TestSuite tests) {

      ...

  }

}
```

Listing 1: Declaring COs in Target Applications

invocation over the network. In the JSTaaS example, implementations of test runners are good candidates for COs. Test execution potentially produces high computational load, and little interaction between the test runner and the rest of the application is necessary during test execution. Fig. 1 illustrates the basic operation of CloudScale in an interaction diagram. The grey boxes indicate code that is injected. Hence, these steps are transparent to the application developer.

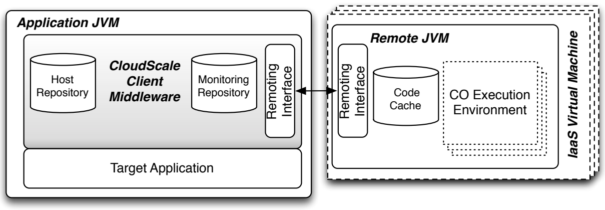

Figure 2: System Deployment View

Fig. 2 shows a high-level deployment view of a CloudScale application. The grey box in the target application JVM again indicates injected components. Note that CHs are conceptually "thin" components, i.e., most of the actual CloudScale business logics is running on client side. 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 CloudScale currently does not explicitly target multi-tenancy [6], these sand boxes are currently implemented in a light-weight way via custom Java classloaders. On client-side, the CloudScale 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 4.1).

3.2 Interacting with Cloud Objects

Application developers declare COs in their application code via simple Java annotations (see Listing 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, CloudScale 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). Additionally, classes defining COs may contain static fields and methods. Operations on those are not intercepted by CloudScale, as they potentially lead to a problem that we refer to as *JVM-local updates*: if code executing on a CH (for instance 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 CloudScale bytecode injection. To prevent this problem, static fields can be annotated with the `@CloudGlobal` annotation (see Listing 1). 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. Note that this behavior is not default for performance reasons, as synchronizing static field values is expensive, and only required if JVM-local updates are possible.

Evidently, most CloudScale applications require parameter objects to be passed from the target application to the COs, or between COs. As the purpose of these objects is typically to transport data, we refer to them as data objects. CloudScale supports data object passing via three common strategies, as summarized in Table 1. If small, primitive data objects (e.g., identifiers or numerical parameters) need to be passed, the common strategy is to pass them using `copy-by-value`. This strategy is simple and has a low overhead for small objects. However, it requires the objects to be marshallable. Technically, this means that they need to support standard Java serialization mechanisms. `By-reference` is more powerful, but should be used with care, as any interaction with the by-reference proxy results in additional remoting. CloudScale applications often use `by-reference` to implement callback mechanisms, allowing for flexible asynchronous programming models. Furthermore, the `by-reference` mechanism allows COs to get access to a proxy of a different CO instance, hence, enabling communication between different COs. Finally, CloudScale also supports the `shared` strategy, in which (serialized) data objects are stored in a persistent data store. This data store is shared among all CHs and the target application. This approach is commonly used if large chunks of data need to be passed around multiple times, as is the case for many scientific computing applications. The `shared` strategy is also helpful if the CloudScale application is expected to interface with external data producers or consumers.

Strategy	Description
Copy-by-value	Sends a deep copy of the object. Changes in the copy will not be reflected in the original object.
By-reference	Sends a proxy object (*by-reference proxy*) instead of a copy. Invocations of the proxy are redirected back to the original object.
Shared	Data objects are exchanged by storing them in a shared data store. All CHs and the target application operate on the same copy of the data (ensured by transactional mechanisms and concurrency control).

Table 1: Data Object Passing Strategies

`Copy-by-value` and `by-reference` are defined on Java method and field level, i.e., by annotating a parameter of a CO method, or a CO member field. In Listing 1, `TestResult` is a `by-reference` parameter, while the actual test suite is passed `copy-by-value`. The `shared` model needs to be triggered explicitly by requesting CloudScale to inject a connection handle to a shared database (*data store* in CloudScale terms) into the Java application via dependency injection. For example, in Listing 1, a CouchDB NoSQL data store [10] is injected. The application code then explicitly reads from or writes to this data store. CloudScale internally uses a custom data mapping framework, which allows to serialize arbitrary "Plain Old Java Objects" to a wide array of relational and non-relational data stores, including CouchDB, Riak, HBase and any SQL database compatible with the Java Persistence API (JPA). The `shared` approach also has the additional advantage that conflicts are detectable on database level via optimistic concurrency control [19]. Optimistic concurrency control essentially implies that each revision of a data object is associated with a numerical version flag. Whenever the data object is updated, the version flag is incremented. Whenever a data object should be updated, and the version flag in the data store is higher than the version of the object that should be written, a conflicting change is detected and reported back to the user via a "DataStoreException". For `by-reference` and `copy-by-value`, developers need to make sure that different COs do not override changes of other COs manually, just as it is the case for any other multi-threaded Java applications.

3.3 Remote Classloading

Whenever a CH has to execute a CO method, CloudScale has to ensure that all necessary resources (i.e., program code and other files, for instance 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 CloudScale to load missing or modified code efficiently and seamlessly for the application only whenever it is necessary, thus simplifying application development and maintenance. We discuss this process in more detail in [33].

4. SUPPORTING ELASTIC APPLICATIONS

After introducing the basic notions in Section 3, we now introduce how CloudScale enables elastic cloud application development.

4.1 Autonomic Elasticity via Complex Event Processing

One central advantage of CloudScale 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 CloudScale is to abstract from technicalities of these tasks, but still grant developers full control over the elasticity behavior of their application.

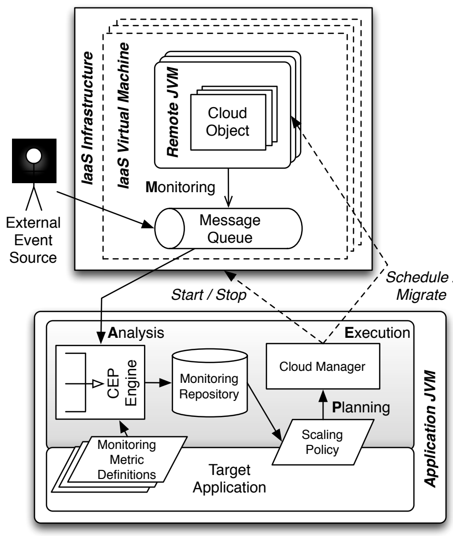

Figure 3: Autonomic Elasticity

An overview over the CloudScale components related to elasticity, and their interactions, is given in Fig. 3. Conceptually, our system implements the well-established autonomic computing control loop of monitoring-analysis-planning-execution [16] (MAPE). The base data of monitoring is provided using event messages. All components in a CloudScale system (COs, CHs, as well as the middleware itself) trigger a variety of predefined lifecycle and status events, indicating, for instance, that a new CO has been deployed or that the execution of a CO method has failed. Additionally, CloudScale 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* [25] for analysis. CEP is the process of merging a large number of low-level events into high-level knowledge, e.g., 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. Possible decisions of the *scaling policy* are the provisioning of new CHs, migrating existing COs between CHs, and scheduling the new CO to a (new or existing) CH. The policy is also responsible for deciding whether to deprovision an existing CH at the end of each billing time unit. Additionally, 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 `EngineSetupEvent`s received in a 10 second batch, is given in Listing 2.

```
MonitoringMetric metric =
  new MonitoringMetric();
metric.setName("AvgEngineSetupTime");
metric.setType(Double.class);
metric.setEpl(
  "select avg(duration)
   from EngineSetupEvent.win
      :time_batch(10 sec)"
);
EventCorrelationEngine.getInstance()
  .registerMetric(metric);
```

Listing 2: Defining Monitoring Metrics via CEP

Monitoring metrics range from very simple and domain-independent (e.g., calculating the average CPU utilization of all CHs) to rather application-specific ones, such as the example given in Listing 2. Whenever the CEP-statement is triggered, the CEP engine writes 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 work load. Hence, sophisticated scaling policies that minimize cloud infrastructure costs [21] or that maximize utilization [14] are easy to integrate. As part of the CloudScale release, we provide a small number of default policies that users can integrate out of the box, but expect users to write their own policy for non-trivial applications. This has proven necessary as, generally, no generic scaling policy is able to cover the needs of all applications.

Finally, the *cloud manager* component, which can be seen as the heart of the CloudScale 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, de-provisioning of existing ones, as well as the deployment or migration of COs).

Fig. 4 depicts the type hierarchy of all predefined events in CloudScale. 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

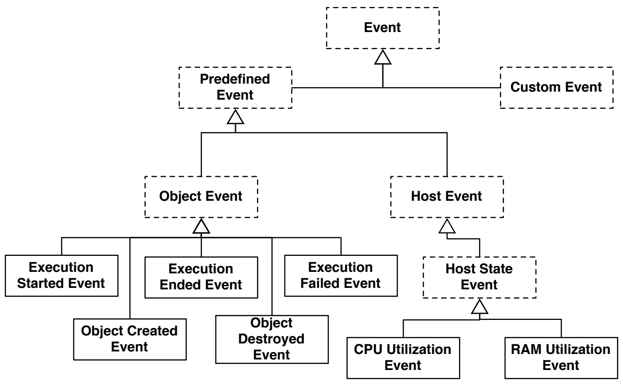

Figure 4: Monitoring Event Hierarchy

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 [22].

4.2 Deploying to the Cloud

As all code that interacts with the IaaS cloud is injected, the CloudScale programming model naturally decouples Java applications from the cloud environment that they are physically deployed to. This allows developers to re-deploy the same application to a different cloud simply by changing the respective parts of the CloudScale configuration. CloudScale 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.

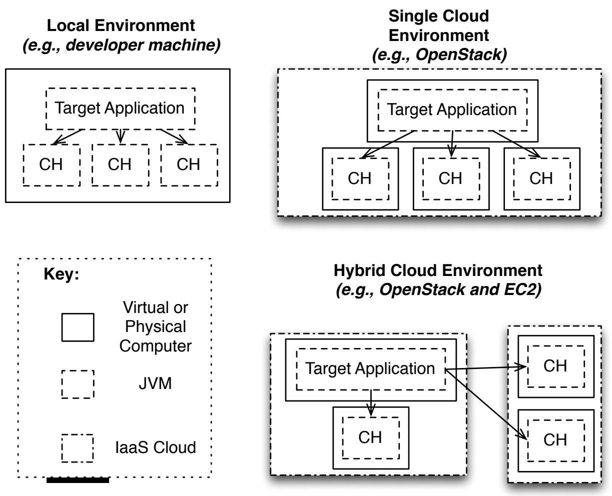

Figure 5: Different Types of Deployment Environments

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 on which cloud. Fig. 5 illustrates the different types of environments supported by CloudScale.

4.3 Development Process

As CloudScale makes it easy to switch between different cloud environments, the middleware supports a streamlined development process for elastic applications, as sketched in Fig. 6. The developer typically starts by building her application as a local, multi-threaded Java application using common software engineering tools and methodologies. Once the target application logic is implemented and tested, she adds the necessary CloudScale annotations, as well as scaling policies, monitoring metric definitions, and CloudScale configuration as required. Now she enables CloudScale code injection by adding the necessary post-compilation steps to the application build process. Via configuration, the developer specifies a deployment in the local environment first. This allows her to conveniently test and debug the application on her development machine, including tuning and customizing the scaling policy. Finally, once she is satisfied with how the application behaves, she changes the configuration to an actual cloud environment, and deploys and tests the application in a physically distributed fashion.

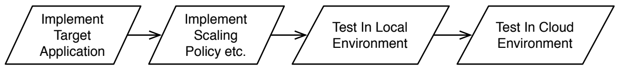
Implement Target Application
Implement Scaling Policy etc.
Test In Local Environment
Test In Cloud Environment

Figure 6: High-Level Development Process

We argue that this process significantly lessens the pain that developers experience when building applications for IaaS clouds, as it reduces the error-prone and time-consuming testing of applications on an actual cloud. However, of course this process is idealized. Practical usage shows that developers will have to go back to a previous step in the process on occasion. For instance, after testing the scaling behavior in the local environment, the developer may want to slightly adapt the target application to better support physical distribution. Still, we have found that the conceptual separation of target application development and implementation of the scaling behavior is well-received by developers in practice.

5. IMPLEMENTATION

We have implemented CloudScale as a Java-based middleware under an Apache Licence 2.0. The current stable version is available from Google Code[5]. This web site also contains documentation and sample applications to get users started with CloudScale.

Our implementation integrates a plethora of existing technologies, which is summarized in Fig. 7. CloudScale uses aspect-oriented programming [17] (via AspectJ) to inject remoting and cloud management code into target applications. Typically, this is done as a post-compilation step in the build process. Dynamic proxying is implemented by means of the CGLib code generation library. For event processing, the well-established Esper CEP engine is used. The client-side middleware interacts with CHs via a JMS-compatible message queue (currently Apache ActiveMQ). Furthermore, COs and the target application itself can read from and write to a shared data store (for example Apache CouchDB). CHs

[5]https://code.google.com/p/cloudscale/

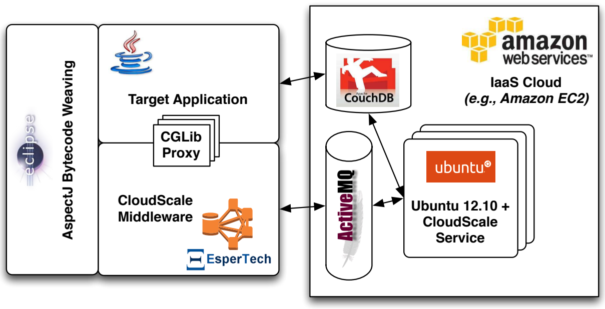

Figure 7: Implementation Overview

themselves are simple Ubuntu 12.10 Linux hosts running Java and a small CloudScale operating system service, which receives CO requests and executes them. Currently, we have built CH base images for OpenStack and Amazon EC2, which are linked from the Google Code web site, and which can be used out of the box with the stable version 0.2.0 of CloudScale (the current version at the time of writing). We will also provide images for future stable versions, once they become available.

6. VALIDATION

We validate CloudScale along two different dimensions. Firstly, we evaluate our claim that CloudScale indeed allows developers to build elastic applications more efficiently than by using solely regular IaaS tool set. To this end, we conducted an user study with 9 participants to assess the developers' experience with CloudScale as compared to using a plain IaaS platform. Secondly, we quantify the additional overhead introduced by CloudScale based on an implementation of the JSTaaS case study application deployed to an OpenStack-based private cloud.

6.1 User Study

One core claim of CloudScale is that it is easier and faster (in terms of development time) for developers to build elastic applications on top of CloudScale than by using IaaS facilities directly. Hence, we present the results of initial user testing we conducted with the goal of substantiating this claim. Note that we deliberately do not compare CloudScale to more domain-specific platforms (such as Apache Hadoop, which is a state-of-the-art implementation of the map/reduce idea [12]). CloudScale aims to be more general with regard to the use cases that it can support, hence, such a comparison would necessarily be unfair. Furthermore, we do not compare CloudScale to existing PaaS middleware. We argue that CloudScale is useful in many cases when adopting a PaaS middleware is infeasible, for instance, if a private IaaS cloud should be used, or if vendor lock-in needs to be avoided. Additionally, as current PaaS middleware typically focuses on scaling traditional 3-tier web applications, both applications that we used in the study would be difficult to implement on top of existing PaaS providers anyway.

6.1.1 Study Setup and Methodology

We conducted our study with 9 master students of computer science at TU Vienna. Of those, 6 are currently working or have previously worked as software engineers on industrial Web engineering projects (one participant has not worked in industry, but actively contributes to a number of open source projects). All but one participant reported significant prior experience in Java programming. Further, 6 of the 9 participants had previous knowledge of cloud computing. We used a private cloud system hosted at TU Vienna, which consists of 12 dedicated Dell blade servers with 2 Intel Xeon E5620 CPUs (2.4 GHz Quad Cores) each, and 32 GByte RAM. The private cloud is running OpenStack Folsom (release 2012.2.4). For the study, each participant was alloted a quota of up to 8 very small instances (1 virtual CPU, and 512 MByte of RAM), which they could use to implement and test their solutions.

We executed the study in two phases. For both phases, all participants were tasked with implementing a specific elastic application either directly on top of OpenStack or using CloudScale. Technologies (i.e., CloudScale or OpenStack directly) were assigned at random, but each participant was using each technology once. In the first phase, the task was to develop a parallel computing implementation of a genetic algorithm. In the second phase, a service that executes JUnit test cases (inspired by the JSTaaS example) had to be built. For both tasks, each participant had one month of time to implement the application using the assigned technology, and write a semi-structured report summarizing his experience. Each participant was working alone, but they were allowed to discuss their problems and solutions in a private Internet forum. We also provided technical support for both technologies.

6.1.2 Study Results and Interpretation

We now summarize our most interesting findings. Our first question was whether CloudScale makes it easier and faster to build elastic IaaS 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 2. $\tilde{X}$ and $\tilde{Y}$ represent the median size of solutions, while σ_X and σ_Y indicate standard deviations. It can be seen that using CloudScale indeed reduces the total source code size of applications. Going into the study, we expected CloudScale to mostly reduce the amount of code necessary for interacting with the cloud. However, our results indicate that using CloudScale indeed also 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 CloudScale takes over as "business logics" when building the elastic application directly on top of OpenStack. 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, the standard deviations are all rather large (i.e., solutions using both technologies varied widely in size). However, as will be discussed in Section 6.3, our results are still statistically significant with reasonable confidence.

	CloudScale		**OpenStack**		**Diff.**
	$\tilde{X}$	σ_X	$\tilde{Y}$	σ_Y	$\tilde{Y}-\tilde{X}$
Business Logics	389	273	504	375	-115
Cloud Management	150	71	200	218	-50
Other Code	100	207	300	221	-200
Entire Application	900	441	1100	589	-200

Table 2: Solutions Sizes (in Lines of Code)

Summarizing, the median CloudScale solution across both tasks is only a little over 80% of the size in lines of code as the median OpenStack based solution. Furthermore, 7 out of 9 participants reported that their CloudScale solution is smaller than their OpenStack solution, independent of which task they used which technology for. 1 participant reported that both solutions are about the same size, and for 1 participant the outcome of the CloudScale solution was significantly larger than the solution he built directly on OpenStack.

	CloudScale		**OpenStack**		**Diff.**
	$\tilde{X}$	σ_X	$\tilde{Y}$	σ_Y	$\tilde{Y}-\tilde{X}$
Tool Learning	8	5	10	6	-2
Coding	21	14	30	17	-9
Bug Fixing	10	10	10	11	0
Other Activities	7	10	11	12	-4
Entire Application	61	21	66	31	-5

Table 3: Time Spent (in Full Hours)

However, looking at lines of code alone is not sufficient to validate our initial claim, as it would be possible that the CloudScale solutions, while being more compact, are also more complicated (and, hence, 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 3. 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 CloudScale is lower than for working with OpenStack directly. For coding and other activities, CloudScale was also more efficient for our participants. However, participants spent about the same time on bug fixing with both approaches. In total, building CloudScale solutions took about 90% of the time of building comparable solutions directly on OpenStack. 6 out of 9 participants reported that they were significantly quicker with CloudScale (in a single case building the OpenStack solution amounted to almost twice the effort in hours), while 2 participants spent more time on the CloudScale solution. 1 participant spent about the same amount of time on both solutions.

It should be noted that both metrics reported so far (lines of code and work hours) do not consider whether CloudScale and OpenStack solutions differ in quality or features. While this is hard to judge objectively, most participants commented that they think their OpenStack solutions are more basic than their CloudScale submission. Hence, it is well possible that the metrics reported here are pessimistic, i.e., that actual savings in lines of code and development time are in fact larger than reported in our study.

	CloudScale		**OpenStack**		**Diff.**
	$\tilde{X}$	σ_X	$\tilde{Y}$	σ_Y	$\tilde{Y}-\tilde{X}$
Simplicity	2	0.60	4	1.13	-2
Debugging	2	1.13	3	1.07	-1
Development Process	2	1.12	4	0.74	-2
Stability	1	1.39	2	0.82	-1
Overall	2	0.64	3	0.71	-1

Table 4: Subjective Ratings

Secondly, we were interested in the participant's subjective evaluation of both technologies. Hence, we asked them to rate the technologies along a number of dimensions from 1 (very good) to 5 (insufficient). Here, we report on the

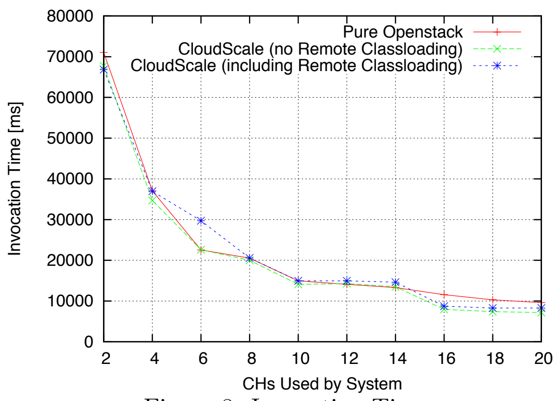

Figure 8: Invocation Time

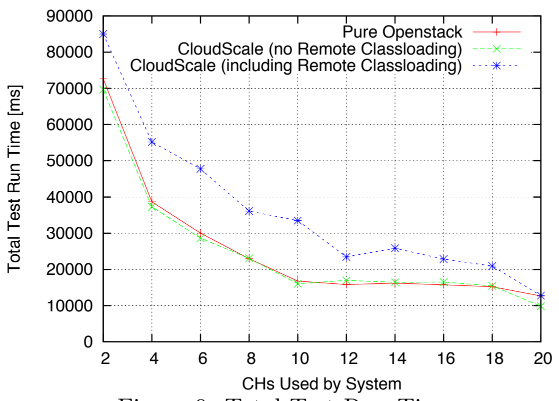

Figure 9: Total Test Run Time

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?). We have also collected data on other dimensions, but omit them here, as CloudScale and OpenStack have been rated identically in them. All in all, CloudScale was considered simpler than OpenStack, and participants valued the development process associated with CloudScale. Improved facilities for debugging as well as less unexpected problems were also judged positively. Overall, the participants graded CloudScale with a median rating of 2, and judged working directly on OpenStack a bit less desirable (rating 3). All participants were able to build (more or less sophisticated) solutions for both tasks using any technology, but 7 out of 9 participants reported that they would rather use CloudScale if tasked with a similar project again. However, most participants also found that CloudScale still has room for improvements. Most importantly, some participants struggled with writing complex scaling policies. All anonymized reports, task descriptions, solutions and more details are publicly available on Google Code[6].

6.2 Quantitative Experiments

Secondly, we investigated whether the improved convenience of CloudScale is paid for with significantly reduced application performance. Therefore, the main goal of our quantitative experiments was to compare the performance of the same application built on top of CloudScale and on OpenStack directly.

6.2.1 Experiment Setup

To achieve this, we built a simple implementation of the core JSTaaS functionality on top of the same OpenStack cloud that was introduced in Section 6.1.1. Secondly, we also implemented the same application using CloudScale, and ran it on the same OpenStack cloud. As the main goal was to calculate the overhead introduced by the CloudScale, we designed both implementations to have the same behavior and reuse as much business logics code as possible. In addition, to simplify our setup and to avoid major performance side effects, we limited ourselves to a scenario, where the number of CHs (or worker virtual machines in the OpenStack case) is static. The source code of both applications is available on the Google Code page referenced in Section 6.1.2.

Both solutions 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. Both solutions were tested with a test setup that consisted of 20 identical parallelizable long-running test suites scheduled evenly over the set of CHs or workers. Each test suite consisted of a set of JavaScript unit tests calculating Fibonacci numbers. During the evaluation we measured the execution time of each single test suite (denoted as "Invocation Time") as well as 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, etc., denoted as "Total Test Run Time"). Each experiment was repeated 10 times. Raw evaluation results are again available on Google Code.

6.2.2 Experiment Results

Fig. 8 shows the median time it took to execute a test suite. Clearly, there is no statistically significant difference in this metric between CloudScale and the native OpenStack solution. Hence, using CloudScale has not added a significant overhead to the actual execution of the target application business logics. However, as Fig. 9 indicates, this is not entirely true if considering the end-to-end invocation time, which also includes factors such as setup and code distribution. Here, CloudScale (blue dotted line) introduces a sizable overhead of relatively constant size (around 10 seconds in our experiments) in comparison to the native OpenStack application. To the largest part, this overhead is due to dynamic remote classloading. Remote classloading is a relatively expensive feature (as not only user code, but also external dependencies need to be loaded), which the native OpenStack solution simply does not have (in the OpenStack solution, all program code is statically pre-packaged in cloud images and does not need to be loaded over the network). Hence, in order to keep the comparison fair, we have additionally evaluated an alternative version of the CloudScale solution, which also did not require remote classloading (green dashed line). As we can see, this version again does not have a statistically significant performance overhead over the native OpenStack solution. This leads us to conclude that using CloudScale introduces a significant performance penalty, but this penalty is mostly due to

[6]https://code.google.com/p/cloudscale/wiki/ICSEValidation

the remote classloading feature, and not due to the CloudScale abstraction in general. That being said, we consider remote classloading a core feature of CloudScale, and conclude that our implementation could be further optimized for practical usage.

6.3 Threats to Validity

While we conducted our user study in all conscience, there are still a few threats to its validity. Firstly, the number of data points we collected (18 in total, 9 per evaluated technology) was rather low, which raises the question whether the results reported in Section 6.1 are statistically significant. Assuming normal distribution of data points, we have conducted one-tailed Student's t-tests [27] for the results we achieved. The resulting confidence values for the most relevant outcomes reported in Section 6 are given in Table 5. These tests indicate that our results regarding solution size and work hours are significant with a confidence in of more than 0.88. The confidence of the other reported results is higher (over 0.94 for all dimensions except Stability). The difference in the Stability ratings turned out to be statistically insignificant. Clearly, a higher confidence for the lines of code and work hour comparisons (i.e., more data points) would be desirable, but given the fact that participants already invested between 50 and 200 hours of work in this study as is, we are satisfied with the results. Secondly, the participants of the study are all students at TU Vienna, and, hence, a relatively homogeneous group. However, we have deliberately selected mostly students with prior work and project experience, and assume that our participants are representative of junior-level developers in our region. Thirdly, when doing a study with students, there is always the danger that students will feel pressured to report the results they assume their professors want to achieve. Analyzing the provided feedback, we have the impression that this factor was not significant in this study, as every participant also reported on clear negatives with CloudScale. However, this factor is hard to quantify objectively. Finally, a last threat is that the scenarios we asked participants to implement are particularly suitable to CloudScale. To minimize this factor, we have deliberately selected two relatively different scenarios, which also included different implementation challenges.

	Confidence $(1 - p)$
Total Lines of Code	0.914
Total Work Hours	0.886
Overall Evaluation	0.973
Simplicity	0.942
Development Process	0.981
Debugging	0.967
Stability	0.663

Table 5: Statistical Confidence Values

The main threat to the validity of our numerical overhead measurements are as follows. One the one hand, it would be possible that the reference OpenStack implementation of JSTaaS and the CloudScale implementation differ in factors unrelated to cloud deployment. We have minimized this factor by using identical code in both implementations as far as possible. In general, given that the performance is essentially the same for both solutions, we assume that this is not the case. On the other hand, and similar to the user study, it is not guaranteed that the JSTaaS case study is in fact representative of other applications. This concern cannot be resolved entirely, however, we have conducted similar experiments on entirely different case studies in the past (see for example [23], but consider that this paper refers to a much older version of CloudScale, hence the overhead of this early CloudScale version was larger). Both JSTaaS implementations used in our experiments are online and can be analyzed by the interested reader.

7. RELATED WORK

There are essentially two schools of thought of how one should build any distributed system. On the one hand, many approaches aim at hiding the complexity of distribution behind convenient abstractions, such as remote procedure call systems. On the other hand, some claim that such abstractions always have to be leaky, and, hence, should be avoided altogether (for just one example of this argument, see [32]). In our work, we follow the former school of thought. Essentially, CloudScale provides an abstraction that makes elastic applications running on top of an IaaS cloud seem like regular, non-distributed Java applications. Hence, CloudScale implements the ideal for building elastic applications mentioned in [29], a "single shared global memory space of mostly unbounded capacity". In doing so, our work is complementary to a number of related research works. In the following, we summarize the most significant related work and discuss what we consider the main contributions of our work over them.

7.1 Related Research

A comprehensive starting point for research work on elastic cloud application development is provided by the survey in [31]. According to the taxonomy used in this paper, our work clearly falls into the category of container-level scalability in the platform layer (the container being CloudScale in this case). Other research approaches that also fall into this group are Aneka [9] and AppScale [18]. Aneka is a .NET-based platform with a focus on enabling hybrid cloud applications. Unlike CloudScale, Aneka is more akin to traditional Grid computing middleware, providing a relatively low-level abstraction based on the message passing interface (MPI). In general, Aneka seems suitable for building scientific computing applications, but less so for enterprise applications. AppScale, on the other hand, specifically (and exclusively) targets Online Transaction Processing (OLTP) style enterprise applications. Essentially, AppScale is an open source implementation of GAE, and is interface-compatible to GAE. Further, there exist a number of research contributions proposing platforms for map/reduce data processing in the cloud, for instance [15]. Finally, another platform with a focus on data processing has been proposed by the BOOM research project [4]. BOOM builds on a custom, declarative programming model and is clearly geared towards data analytics. The main contribution of CloudScale over all of these models is that CloudScale has a more general claim, and is able to handle a wide variety of elastic application types, including data-intense, processing-intense and OLTP style web applications.

The European commission funded project Contrail [2] is currently building another PaaS for hosting relatively flexible elastic applications, called ConPaaS [28] (Contrail PaaS). ConPaaS supports web and high-performance applications built in Java or PHP, and can be used in combination with

Amazon EC2 or an OpenNebula private cloud. However, ConPaaS seems to operate on a lower level of abstraction than CloudScale. This PaaS environment consists of elastically hosted Web services, such as web or database servers, and mostly integrates existing stand-alone components.

Aside from elastic computing platforms, our work is also related to frameworks for cloud deployment. Cloud deployment is the process of provisioning the required resources from the cloud, installing and running the required software on each virtual machine, and starting the application in the required way. An early research framework that addressed this problem was Cafe [26]. Many ideas of Cafe have then cross-fed into the TOSCA OASIS specification [24]. A reference implementation is currently being worked on under the moniker OpenTOSCA [7]. Unlike CloudScale, these tools are not complete runtime platforms, but focus on deployment and provisioning only. Hence, they do not enable elasticity as such.

7.2 Related Tools and Commercial Approaches

CloudScale is related to a plethora of existing technologies and commercial PaaS offerings, which provide scalable platforms on top of the cloud. A service particularly related to our work is CloudBees RUN@Cloud [1], which provides continuous integration and an elastic platform for hosting Enterprise Java Beans (EJB) applications. Other PaaS offerings that support the Java programming language include the well-known GAE, Amazon's Elastic Beanstalk services, or Microsoft's Azure. Outside of the Java world, Heroku[7] is gaining traction as a provider of PaaS for dynamic scripting languages, such as Ruby or Python.

The main disadvantage of all of those systems is that they imply a significant loss of control for the developer. They typically require the usage of a given public cloud (typically provided by the same vendor), imply the usage of proprietary APIs, and restrict the types of applications that are supported (typically, these platforms support only Tomcat-based OLTP style systems). CloudScale, on the other hand, allows application developers to retain full control over their application. CloudScale applications are not bound to any specific cloud provider, are easy to migrate, work well in the context of private or hybrid clouds, and support a wider variety of applications, while still providing an abstraction comparable to commercial PaaS solutions.

A special case is the AppScale platform (already introduced in Section 7.1), which also forms the basis of a commercial product. AppScale allows users to build their own GAE compliant PaaS on top of any private or public IaaS service. We consider this system a significant step forward in comparison to other vendors. However, other disadvantages in comparison to CloudScale (e.g., being only really suitable for OLTP style applications) naturally also apply to AppScale.

Many core features and terms of CloudScale, as introduced in Section 3.1, are akin to well-known Java remote procedure call technology, e.g., Java RMI or EJB. However, note that the main contribution of CloudScale is not in enabling remoting (as RMI and EJB do), but rather in enabling elasticity. Hence, the actual conceptual overlap between CloudScale and these frameworks is not particularly large. It certainly would have been possible to build CloudScale as an extension of RMI, but this has been decided against for technical reasons.

[7]https://www.heroku.com/

8. CONCLUSIONS

CloudScale is a novel, Java-based middleware that eases the development of elastic cloud applications on top of an IaaS cloud. CloudScale 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, CloudScale 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. Scaling policies can use an integrated CEP engine, allowing developers to formulate more sophisticated rules for scaling up and down than related tools, which mostly base elasticity on simple thresholds for CPU or memory usage.

We have introduced the core parts and concepts of CloudScale, and presented an evaluation of the middleware based on an initial user study as well as using a simple case study application (JSTaaS). Our results indicate that CloudScale is indeed well received among developers, and that the core ideas of the middleware are perceived as valuable when building elastic applications. However, our numerical results also indicate that some mechanisms in the current implementation of CloudScale, most notably the integrated remote class loading facilities, still leave room for improvement to reduce the performance overhead of the middleware in comparison to manually built IaaS applications.

8.1 Ongoing and Future Work

At the time of this writing, we are still actively working on several improvements for CloudScale. From a technical point of view, we are continuously improving the code base and documentation of the framework. For instance, we are currently working on adding support for additional IaaS cloud providers, as well as improving the general performance and stability of the middleware. We also plan to provide a hosted demo version of CloudScale as a service in the future, which will allow potential users to easily evaluate whether the CloudScale model is for them. Furthermore, we are currently testing CloudScale as a tool to cloud-migrate a number of existing tools, including Apache JMeter or the service composition engine JOpera [11]. Finally, we plan to extend our initial user study using a more heterogeneous and larger group of developers, in order to resolve the threats to validity identified in Section 6.3.

Acknowledgements

Primarly, we thank Gregor Schauer and Rene Nowak for their technical help with CloudScale. Furthermore, we are indebted to all participants of our study. The research leading to these results has received funding from the Austrian Science Fund (FWF) under project reference P23313-N23 (Audit 4 SOAs), as well as from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 257483 (INDENICA). The second author of this paper is financially supported by the Vienna PhD School of Informatics[8].

[8]http://www.informatik.tuwien.ac.at/teaching/phdschool

9. REFERENCES

[1] "CloudBees Platform," http://www.cloudbees.com/, Last visited: September 4th, 2013.

[2] "Contrail Open Computing Infrastructure for Elastic Services," http://contrail-project.eu/, Last visited: September 4th, 2013.

[3] L. Abraham, M. A. Murphy, M. Fenn, and S. Goasguen, "Self-provisioned hybrid clouds," in *Proceedings of the 7th international conference on Autonomic computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 161–168. [Online]. Available: http://doi.acm.org/10.1145/1809049.1809075

[4] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears, "Boom Analytics: Exploring Data-Centric, Declarative Programming for the Cloud," in *5th European Conference on Computer Systems (EuroSys'10)*. ACM, 2010, pp. 223–236.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A View of Cloud Computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[6] C.-P. Bezemer, A. Zaidman, B. Platzbeecker, T. Hurkmans, and A. t. Hart, "Enabling multi-tenancy: An industrial experience report," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2010.5609735

[7] T. Binz, G. Breiter, F. Leyman, and T. Spatzier, "Portable cloud services using tosca," *IEEE Internet Computing*, vol. 16, no. 3, pp. 80–85, May 2012. [Online]. Available: http://dx.doi.org/10.1109/MIC.2012.43

[8] R. Buyya, C. 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, pp. 599–616, 2009.

[9] R. N. Calheiros, C. Vecchiola, D. Karunamoorthy, and R. Buyya, "The Aneka platform and QoS-driven resource provisioning for elastic applications on hybrid Clouds," *Future Gener. Comput. Syst.*, vol. 28, no. 6, pp. 861–870, Jun. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.future.2011.07.005

[10] R. Cattell, "Scalable SQL and NoSQL Data Stores," *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27, May 2011. [Online]. Available: http://doi.acm.org/10.1145/1978915.1978919

[11] Cesare Pautasso and Gustavo Alonso, "JOpera: A Toolkit for Efficient Visual Composition of Web Services," *International Journal of Electronic Commerce*, vol. 9, no. 2, pp. 107–141, Jan. 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1278095.1278101

[12] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[13] T. Dillon, C. Wu, and E. Chang, "Cloud computing: Issues and challenges," in *Proceedings of the 2010 24th IEEE International Conference on Advanced Information Networking and Applications*, ser. AINA '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 27–33. [Online]. Available: http://dx.doi.org/10.1109/AINA.2010.187

[14] S. Genaud and J. Gossa, "Cost-Wait Trade-Offs in Client-Side Resource Provisioning with Elastic Clouds," in *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing*, ser. CLOUD '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/CLOUD.2011.23

[15] T. Gunarathne, T.-L. Wu, J. Qiu, and G. Fox, "MapReduce in the Clouds for Science," in *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 565–572.

[16] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003. [Online]. Available: http://dx.doi.org/10.1109/MC.2003.1160055

[17] G. Kiczales and E. Hilsdale, "Aspect-Oriented Programming," *SIGSOFT Software Engineering Notes*, vol. 26, no. 5, pp. 313–, Sep. 2001. [Online]. Available: http://doi.acm.org/10.1145/503271.503260

[18] C. Krintz, "The AppScale Cloud Platform: Enabling Portable, Scalable Web Application Deployment," *IEEE Internet Computing*, vol. 17, no. 2, pp. 72–75, Mar. 2013. [Online]. Available: http://dx.doi.org/10.1109/MIC.2013.38

[19] H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Transactions Database Systems*, vol. 6, no. 2, pp. 213–226, Jun. 1981. [Online]. Available: http://doi.acm.org/10.1145/319566.319567

[20] G. Lawton, "Developing Software Online With Platform-as-a-Service Technology," *Computer*, vol. 41, no. 6, pp. 13–15, Jun. 2008. [Online]. Available: http://dx.doi.org/10.1109/MC.2008.185

[21] 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 *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing (CLOUD '12)*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 213–220.

[22] 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 *Proceedings of the 2012 5th IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–8.

[23] P. Leitner, B. Satzger, W. Hummer, C. Inzinger, and S. Dustdar, "CloudScale: a novel middleware for building transparently scaling cloud applications," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, ser. SAC '12. New York, NY, USA: ACM, 2012, pp. 434–440. [Online]. Available: http://doi.acm.org/10.1145/2245276.2245360

[24] P. Lipton and S. Moser, "OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC," https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca, Last visited: September 4th, 2013.

[25] D. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Professional, May 2002.

[26] R. Mietzner, T. Unger, and F. Leymann, "Cafe: A Generic Configurable Customizable Composite Cloud Application Framework," in *On the Move to Meaningful Internet Systems (OTM 2009)*, R. Meersman, T. Dillon, and P. Herrero, Eds. Springer Berlin / Heidelberg, 2009, vol. 5870, pp. 357–364. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-05148-7_24

[27] Paul Rice, *Mathematical Statistics and Data Analysis*. Brooks/Cole, International Version, 2006.

[28] G. Pierre and C. Stratan, "ConPaaS: A Platform for Hosting Elastic Cloud Applications," *IEEE Internet Computing*, vol. 16, no. 5, pp. 88–92, 2012.

[29] P. Sampaio, P. Ferreira, and L. Veiga, "Transparent Scalability with Clustering for Java e-Science Applications," in *11th International Conference on Distributed Applications and Interoperable Systems*. Springer, 2011, pp. 270–277.

[30] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster, "Virtual infrastructure management in private and hybrid clouds," *IEEE Internet Computing*, vol. 13, no. 5, pp. 14–22, Sep. 2009. [Online]. Available: http://dx.doi.org/10.1109/MIC.2009.119

[31] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 1, pp. 45–52, Jan. 2011. [Online]. Available: http://doi.acm.org/10.1145/1925861.1925869

[32] S. Vinoski, "Convenience over correctness," *IEEE Internet Computing*, vol. 12, no. 4, pp. 89–92, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1109/MIC.2008.75

[33] R. Zabolotnyi, P. Leitner, and S. Dustdar, "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 2013), co-located with ICSE 2013*, 2013.