Handbook of Research on Architectural Trends in Service-Driven Computing

Raja Ramanathan
Independent Researcher, USA

Kirtana Raja
IBM, USA

A volume in the Advances in Systems Analysis, Software Engineering, and High Performance Computing (ASASEHPC) Book Series

Information Science REFERENCE
An Imprint of IGI Global

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA, USA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Handbook of research on architectural trends in service-driven computing / Raja Ramanathan and Kirtana Raja, editors.
pages cm
Includes bibliographical references and index.
ISBN 978-1-4666-6178-3 (hardcover) -- ISBN 978-1-4666-6179-0 (ebook) -- ISBN 978-1-4666-6181-3 (print & perpetual access) 1. Service-oriented architecture (Computer science) 2. Computer network architectures. 3. Enterprise application integration (Computer systems) I. Ramanathan, Raja, 1958- editor. II. Raja, Kirtana, 1987- editor.
TK5105.5828.H356 2014
004.6'54--dc23
2014013829

This book is published in the IGI Global book series Advances in Systems Analysis, Software Engineering, and High Performance Computing (ASASEHPC) (ISSN: 2327-3453; eISSN: 2327-3461)

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 26
Building Elastic Java Application Services Seamlessly in the Cloud: A Middleware Framework

Rostyslav Zabolotnyi
Vienna University of Technology, Austria

Philipp Leitner
University of Zurich, Switzerland

Schahram Dustdar
Vienna University of Technology, Austria

ABSTRACT

Cloud computing is gaining increasing attention from the industry and research; however, there is a lack of advanced Cloud software development tools. While Platform as a Service (PaaS) brings convenient software development platform for application development, it often comes with limitations in terms of application architecture functionality and requires provider lock-in. The Infrastructure as a Service (IaaS) model may sound like a solution to these problems by enabling application development freedom; however, it necessitates operation at the lower level of virtual machines and snapshots. In this chapter, the authors present CloudScale: a low-overhead middleware framework that migrates Java applications seamlessly to the Cloud with minimal changes in the application code. They focus on the main ideas behind CloudScale and its influence on solving Cloud software development and deployment problems with minimal overhead and Cloud-awareness required from developers.

DOI: 10.4018/978-1-4666-6178-3.ch026

INTRODUCTION

In the past few years, the advancement of Cloud computing (Mell & Grance, 2011) has transformed the IT industry, and has given new opportunities and abilities to developers and users. Moreover, Cloud computing simplifies the implementation of innovative ideas for small companies or individuals, and lowers production and maintenance costs for industrial applications (Armbrust et al., 2010). Applications designed with the Cloud in mind (so-called Cloud-native applications) allow developers to elastically adapt to market changes and optimize resource consumption.

Developers can adapt to the Cloud computing model at different levels. The most basic Cloud service model is the *IaaS* (Infrastructure as a Service) (Mell & Grance, 2011) approach. At this level, Cloud service providers offer virtualized resources with requested configuration and operating system (usually in the form of hard drive images and virtual machines) to satisfy application computation requirements (Bhardwaj, Jain, & Jain, 2010). For many use cases, this layer is preferable to *PaaS* (Platform as a Service), as it gives more freedom to the developers, requires less migration effort, and is better standardized than PaaS.

However, building IaaS-based *elastic Cloud applications* is not an easy task, and requires developers to face an entirely new range of challenges. For instance, developers have to introduce a significant amount of *platform-dependent boilerplate code* that allows them to control virtual machines rented from the Cloud, monitor the state of virtual machines, and elastically scale applications up and down according to the current or future load. These tasks are orthogonal to the mission of the applications, introduce significant complications, and bury the application's actual business logic deep under a mountain of platform-dependent code. This boilerplate code has to be developed over and over again for each new application or platform version. In addition, this platform-dependent integration code not only slows down application development, but also causes vendor lock-in, as each Cloud service provider has its own API that developers have to use to be able to interact with the Cloud.

In this chapter, we describe how the *CloudScale*[1] research prototype (Leitner, Satzger, Hummer, Inzinger, & Dustdar, 2012) solves the challenges described above. The CloudScale framework allows developers to declare application Cloud scaling and interaction rules declaratively with the help of Java annotations, thus allowing developers to focus on the business logic of the application. The CloudScale framework injects the IaaS platform-dependent code necessary to scale the application over the Cloud via bytecode manipulation. This approach significantly simplifies the development of Java-based Cloud-native applications and avoids vendor lock-in, as the Cloud-specific code is separated from the application business logic and can be easily changed.

The content of this chapter is structured as follows. In the next section, we provide some background information on the current state-of-the-art research and development attempts to simplify Cloud software development without losing control over the available equipment and infrastructure resources. Next, we introduce our illustrative example application and describe the CloudScale architecture, as well as main design decisions and limitations by presenting the research idea and application development process that enables the development of Cloud applications transparently and seamlessly, without thinking about platform-dependent code and Cloud performance monitoring.

Diving deeper into the scope of possibilities provided by CloudScale, we describe how CloudScale implements accurate and effective class loading on demand (Zabolotnyi, Leitner, & Dustdar, 2013), software state monitoring (Leitner, Inzinger, Hummer, Satzger, & Dustdar, 2012), and how multiple Cloud providers can be targeted in parallel by using the Cloud bursting model (Mell & Grance, 2011). Further, we illustrate how Java applications are built on top of CloudScale in an evaluation section and compare CloudScale with available modern Cloud platforms. Next, we discuss the amount of computational overhead introduced by CloudScale, and present numerical evaluation results of the platform (Leitner, Zabolotnyi, Hummer, Inzinger, & Dustdar, 2013). Finally, we conclude our chapter with an overview of future work and open research issues.

BACKGROUND

Elastic Cloud application development is a broad topic, with a significant amount of research related to it. According to the taxonomy represented by the survey (Vaquero, Rodero-Merino, & Buyya, 2011), CloudScale can be classified as *container-level scalability system* in the platform layer (where CloudScale plays the role of a container for user's application). Similar solutions from this category are AppScale (Krintz, 2013) and Aneka (Calheiros, Vecchiola, Karunamoorthy, & Buyya, 2012).

AppScale targets *Online Transaction Processing* (OLTP) style enterprise applications. AppScale is an open source implementation of GAE (Google App Engine), and is interface-compatible to GAE. Aneka, on the other hand, is a .NET-based platform with a focus on enabling hybrid Cloud applications. Contrasting with CloudScale, Aneka is more similar to traditional Grid computing middleware, providing a relatively low-level abstraction based on the message passing interface (MPI). In general, Aneka appears to be suitable for building scientific computing applications, but does not fit well for enterprise applications.

Another significant segment of distribution platforms is represented by solutions for *MapReduce* data processing (Gunarathne, Wu, Qiu, & Fox, 2010). BOOM (Alvaro, et al., 2010) also belongs to the data processing platforms and builds on a custom, declarative programming model that targets data analytics. CloudScale, in comparison to the platforms presented above, has a more general claim, and is able to handle an extensive variety of elastic application types, including processing-intensive, data-intensive, and OLTP style Web applications.

The project Contrail (Contrail Open Computing Infrastructure for Elastic Services), funded by the European commission (Contrail, 2013), represents another PaaS (called ConPaaS) for hosting flexible elastic applications (Pierre & Stratan, 2012). ConPaaS is supposed to support Web and high-performance applications built in Java or PHP, and may be used in combination with Cloud platforms such as Amazon EC2 or OpenNebula. However, ConPaaS operates on a lower level of abstraction than CloudScale. Its environment is focused on integration of existing stand-alone components such as database services or Web services.

As our work is also related to a set of frameworks for Cloud deployment, we have to mention some of the most prominent examples from this category. Cloud deployment is the process of provisioning the required resources from the Cloud, installing and running required software on each virtual machine, and starting the application. One of the first research frameworks that addressed this problem was Cafe (Mietzner, Unger, & Leymann, 2009). Many ideas of Cafe have migrated into the TOSCA OASIS specification (Paul & Simon, 2013). A reference implementation of TOSCA is currently being implemented under the name "OpenTOSCA" (Binz, Breiter, Leyman, & Spatzier, 2012). These tools focus on resource provisioning and application deployment only, while CloudScale provides a complete runtime platform. Hence, these tools do not enable elasticity as such.

Related Tools and Commercial Approaches

Many core features of CloudScale introduced above are similar to well-known *Java remote procedure call technology*, e.g., Java RMI or Enterprise Java Beans (EJB). However, note that the main contribution of CloudScale is not assisting with remoting (as RMI and EJB do), but rather enabling elasticity. Therefore, the actual intersection between CloudScale and these technologies is not particularly large.

The market of commercial PaaS solutions which provide scalable platforms on top of the Cloud is quite significant. A service named CloudBeesRUN@Cloud (CloudBees, 2013) has a strong similarity to our work. It provides continuous integration and an elastic platform for hosting EJB applications. There are some other PaaS platforms with Java language support, such as the already mentioned GAE, Amazon's Elastic Beanstalk Services, and Microsoft's Azure.

Outside of the Java biosphere, Heroku[2] is gaining traction as a provider of PaaS for dynamic scripting languages, such as Ruby or Python. All these systems imply significant limitations and loss of control for the developer. They typically lock users into a given public Cloud (typically provided by the same vendor), provide proprietary APIs, and restrict the types of applications that are supported (typically, only Tomcat-based OLTP style systems).

Contrarily, CloudScale provides developers full control over the application and does not tie to any specific Cloud provider, allowing easy migration and hybrid Cloud support, while still providing an abstraction comparable to commercial PaaS solutions.

CLOUDSCALE MIDDLEWARE PLATFORM

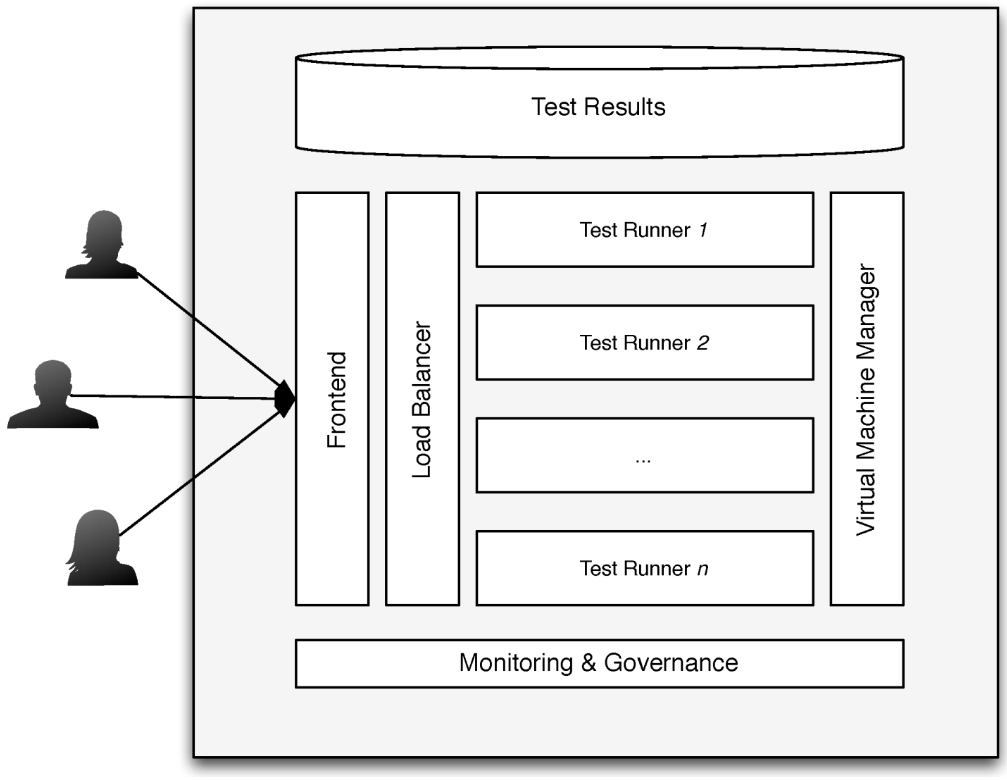
CloudScale is a middleware that takes Java-based applications to the Cloud. The main goal of CloudScale is to allow developers to build local, multithreaded applications (in this chapter referenced to as "target applications") that can be distributed over the available Cloud resources with minimum application distribution knowledge required by the developers. This means that developers design their applications while focusing on the business logic and paying minimal attention to the distribution that will happen at runtime. CloudScale seamlessly takes care of the technicalities of elasticity, such as virtual machine management, performance monitoring, load balancing, and program code distribution.

Example Case

Imagine a Web startup with the JSTaaS ("JavaScript Testing as a Service") business idea. JSTaaS represents a Cloud service for JavaScript application testing. JSTaaS allows clients to register test suites, which will be executed periodically. Service load produced by the different tests varies widely, and clients are billed accordingly. Execution results are stored in a database, which can be accessed by the clients (see Figure 1).

To reduce needed initial funding, hosts used for test execution should be utilized as highly as possible. This means that tests should be co-located on the same machines as much as possible. Therefore, the core of JSTaaS needs to continuously monitor the utilization of all hosts, as well as the execution time of tests. To reduce infrastructure costs, the

Figure 1. JavaScript Testing as a Service application overview



company decides to deploy the initial version of JSTaaS as a Java-based service on Amazon's EC2 public IaaS cloud[3].

This application is not trivial to implement with standard tools. Developers have to setup virtual machines, install the necessary application components on these virtual machines and use monitoring tools to ensure appropriate resource provisioning. Application must be split into task manager, load balancer, and workers to execute the tests. To some extent, AWS Elastic Beanstalk can be used to simplify deployment, and CloudWatch can handle monitoring, but these tools do not solve the core obstacles in the way of running the application.

The popular way to build a distributed system is to hide the complexity of the distribution behind convenient abstractions, such as remote procedure call systems. Some claim that such abstractions always will be leaky, and hence, should be avoided altogether (Vinoski, 2008). In our work, we stick to this abstraction approach, and claim that CloudScale represents an effective and convenient abstraction on top of IaaS, that simplifies Cloud application development. In the following sections, we discuss the concepts of CloudScale to establish the place of CloudScale within the domain of Cloud computing, and show how it is different from other approaches available on the market.

Basic Notions in CloudScale

CloudScale is weaved into the target application based on a set of Java annotations combined with compile-time bytecode modification. From the developer's point of view, CloudScale usage requires adding a new library dependency, some

Java annotations to mark distribution points, and a post-compilation step to weave the necessary distribution code. At this point, the contrast between CloudScale and industrial PaaS solutions is clearly visible. Usually, modern PaaS platforms require developers to design the application according to their architecture and proprietary API, along with a set of limitations on the possible functionality of the application. Such PaaS applications are usually not executable or debuggable outside of the targeted PaaS environment, while solutions based on CloudScale can be executed and tested without access to the IaaS platform, or entirely even without CloudScale.

The core concept of CloudScale is a notion of *Cloud Objects*. Cloud Objects represent instances of classes to be executed on Cloud hosts. Cloud hosts are represented by virtual machines from the IaaS Cloud with CloudScale server component running on them. During the application execution, Cloud Objects are deployed on the appropriate Cloud hosts and all further interactions are treated as remote invocations. CloudScale intercepts each interaction with Cloud Object, routes request to the appropriate Cloud host that executes the operation and returns the result. Because of this, it makes sense to keep Cloud Objects as loosely coupled to the rest of the application as possible, as otherwise the application's performance may suffer.

In the JSTaaS example introduced above, the wrapping objects that execute user tests are good candidates for Cloud Objects. Test execution usually produces a long-running and computation-intensive load, while there's little or no interaction between the separate tests. Figure 2 illustrates how tests are being executed by the application. White boxes represent the code written by developer, while gray boxes stand for code injected by CloudScale after application compilation. The application developer does not see this code and can solely focus on business logic implementation, while application distribution and interaction is handled by CloudScale in the background.

Figure 3 shows a high-level overview of an application that uses CloudScale. Components injected by CloudScale are within the gray box

Figure 2. Basic interaction with Cloud Objects

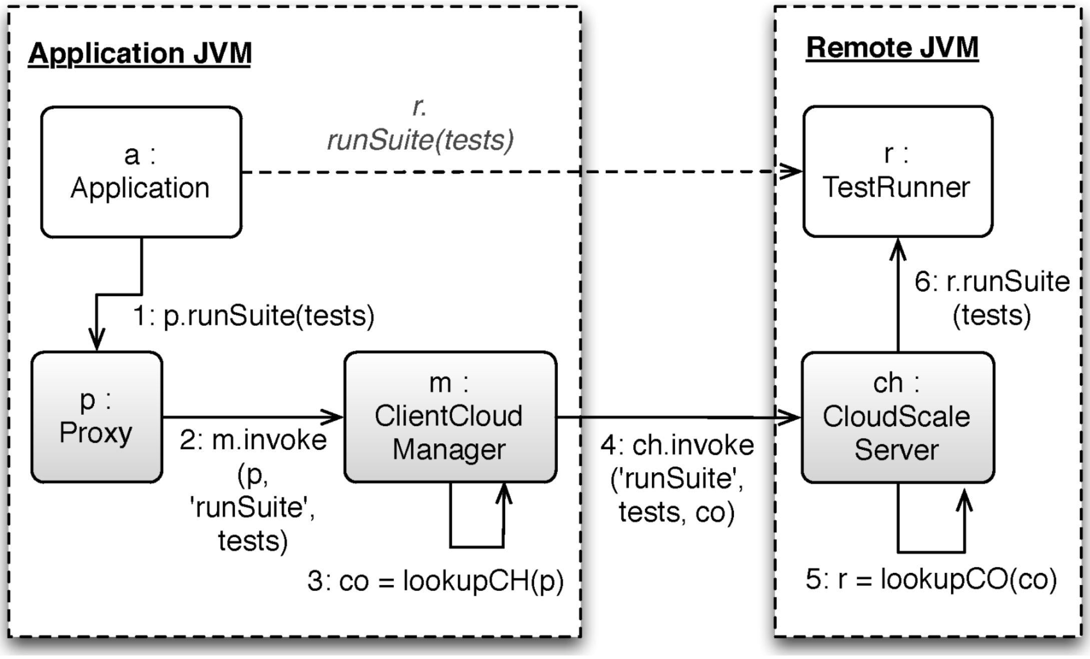
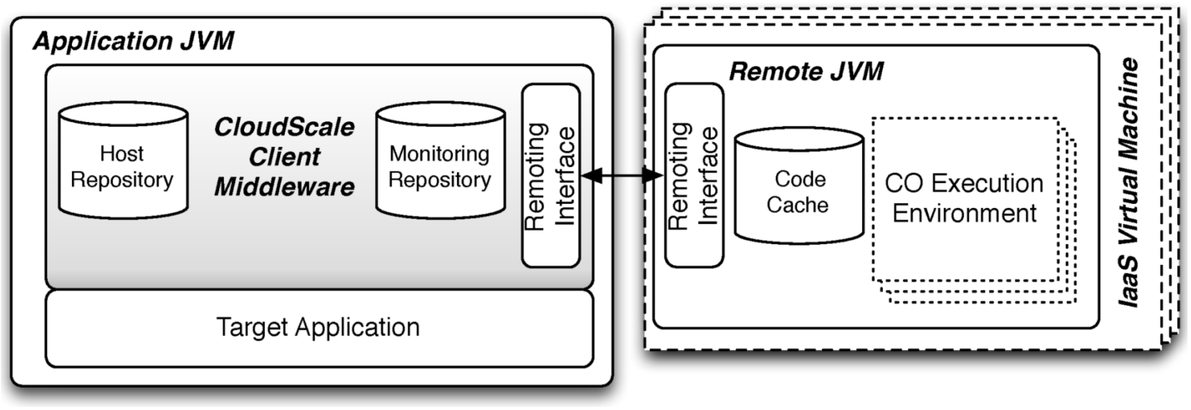


Figure 3. System deployment view



of application's JVM, while Cloud hosts with CloudScale code are indicated by the dashed boxes named "IaaS Virtual Machine". It is clearly visible that Cloud hosts are reasonably lightweight, minimizing performance impact and memory footprint on remote machines. Cloud hosts consist of a simple communication interface and code cache, described in more detail below. While CloudScale currently does not explicitly target multi-tenancy (Bezemer, Zaidman, Platzbeecker, Hurkmans, & Hart, 2010), each Cloud Object's execution is sandboxed and handled by different custom class loaders. The client-side of CloudScale is responsible for maintaining a set of available and used Cloud hosts and deployed Cloud Objects. In addition, it also collects the monitoring data and controls further scaling of the system.

Interaction Patterns

Cloud Objects are declared by developers with the help of simple Java annotations (see Listing 1). Applications can interact with Cloud Objects in the same way as with any other object in Java: invoking methods and getting or setting member fields. As it was described above, CloudScale

Listing 1. Declaring Cloud Objects in target applications

```
@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) {
             ...
       }
 }
```

intercepts each interaction with Cloud Object, routes request to the appropriate Cloud host that executes the operation and returns the result. During this process, the target application is blocked to simulate standard Java behavior for such interactions.

Classes with *@CloudObject* annotation may contain static fields and methods. Operations on them are not intercepted by CloudScale by default, as it is not clear on which host these operations should be executed. If an interaction with static members is stateless (e.g., it does not change the static state of the application), this default CloudScale behavior is the best solution as it introduces no overhead. However, if the application state is changed, this can cause a problem that we call a JVM-local update: if the Cloud Object changes the value of the static field, this update will be available only on the host where this change occurred, while other Cloud hosts and target application's JVM will not be aware of this change. To prevent this problem, static fields can be annotated with the *@CloudGlobal* annotation (see Listing 1). Interactions with such fields are intercepted by CloudScale and forwarded to the target application JVM, causing some overhead, but leading to all hosts operating on the same actual field value. This behavior is not default because of performance reasons, and should be used only if JVM-local updates are not possible.

Some method invocations require other objects or values to be passed to or from the invoked method. Such methods are legal in Cloud Objects, but as passed objects have to be transferred between the JVMs, some specific rules apply. As the common purpose of such passed objects is usually to transport data, we refer to them as *Data Objects*. CloudScale distinguishes three different Data Object passing strategies, summarized in Table 1.

Small, primitive Data Objects are usually passed by-value. Objects passed by-value have to support the Java serialization mechanisms. User-defined classes and all other complex types by default are passed by-reference. This approach is more powerful, as it simulates the same shared object modification pattern as default object passing behavior in Java. It behaves similarly to the Cloud object interception mechanism described above and allows Cloud Objects to interact with each other or pass dynamic information between them. However, it introduces a significant overhead and if it is not required, developers can declare passed object as passed by-value. Finally, the shared strategy allows accessing Data Objects stored in a persistent data store, shared between Cloud hosts and the target application. This approach is beneficial if large chunks of data must be passed within invocations multiple times, or updates to these data structures must be synchronized. This approach is also beneficial if data must be available to external applications.

Data Object's passing strategy is declared via annotations applied to field, method parameter, or method's return type. In Listing 1, the result of runSuite method call is passed by-reference, while the method's parameter "tests" is passed by-value because of the *@ByValueParameter* annotation. Shared data passing strategy has to be triggered explicitly on the connection handle injected by CloudScale. In the discussed example (see Listing

Table 1. Data Object passing strategies

Strategy	Description
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. Application and all Cloud hosts operate on the same copy of the data (ensured by transactional mechanisms and concurrency control of the database).

1), a CouchDB NoSQL data store (Cattell, 2011) is injected into the private field *datastore*. During execution, the application has to explicitly read and write data into the data store.

Internally, CloudScale uses special data mapping framework that allows serialization and deserialization of Plain Old Java Objects (POJO), and supports a range of relational and non-relational data stores such as CouchDB, Riak, HBase and any SQL database compatible with the Java Persistence API (JPA). The shared data passing strategy allows benefiting from database-specific features, such as concurrent access control (Kung & Robinson, 1981), data backup, or conflict resolution. For other Data Object passing strategies (passing by-value and by-reference), developers have to handle concurrent data access themselves.

Class Loading on Demand

Whenever a new host has to be used for target application execution, there is the problem of code availability on the new machine. To be able to execute Cloud Objects on the new host, CloudScale has to ensure that the same code version is available in both JVMs. The trivial approach is to either send the correct code version to each machine on every request or expect Cloud hosts to start with application code already preloaded. The first approach introduces significant overhead for application performance, while the second one causes difficulties for the developers and maintainers. In that case, whenever any change occurs to the codebase, new version of Cloud host images has to be built, significantly slowing down development and deployment process. It becomes even more complicated if we consider a situation when multiple different code versions can be operating on the same machine over time or even in parallel. The only possible alternative to handle all these issues seamlessly for the developer and ensure codebase integrity is to implement a dynamic code search and distribution functionality at the middleware level.

Program Code Distribution Challenges

Dynamic code search and distribution middleware faces a number of challenges that must be solved in order to be useful and effective. Firstly, the framework needs to detect when the code is missing or available version is inconsistent with the one required. Secondly, the appropriate version of the code has to be found and deployed to the target host prior to execution. To simplify suitable code search, target application itself can act as a code server, but a separate low-latency access server may also be used to improve performance. Thirdly, efficient and secure means of communication need to be used to transfer the code to the Cloud machine.

Fourthly, for performance reasons, only the necessary code should be transmitted using an effective batching strategy. For example, the performance impact will increase dramatically if each class will be transmitted separately, while a similarly negative impact on performance can be experienced if Cloud hosts have to wait for all code to be transmitted before starting any execution. Fifthly, to minimize memory and storage usage, the middleware has to ensure that received code is stored not longer than it is actually needed to avoid storage bloating and recurrent code transmissions. Particularly, whenever the code is changed in the target application, the old version should be discarded as outdated, while allowing multiple code versions to exist in parallel as long as they are used. These challenges are summarized in Table 2.

CloudScale Code Distribution Framework

In this section, we describe how CloudScale achieves efficient and seamless code distribution, solving the challenges described above.

On every interaction with a Cloud Object, CloudScale forwards the request to the appropriate Cloud host and starts awaiting execution comple-

Table 2. Summary of code distribution challenges

Challenge Name	Challenge Synopsis
Missing Code Detection	Cloud hosts need to be able to dynamically detect if program code is missing or the version is incorrect.
Trusted Code Storage	Cloud hosts need to be able to locate the code storage service and query appropriate code.
Communication Middleware	Cloud hosts need to have access to a suitable communication middleware that allows them to dynamically exchange code.
Batching Strategy	Cloud hosts need to be able to load only the necessary code using an efficient batching strategy.
Outdated Code Clearance	Cloud hosts need to be aware that program code can change or become obsolete, thus that loaded program code is not valid indefinitely.

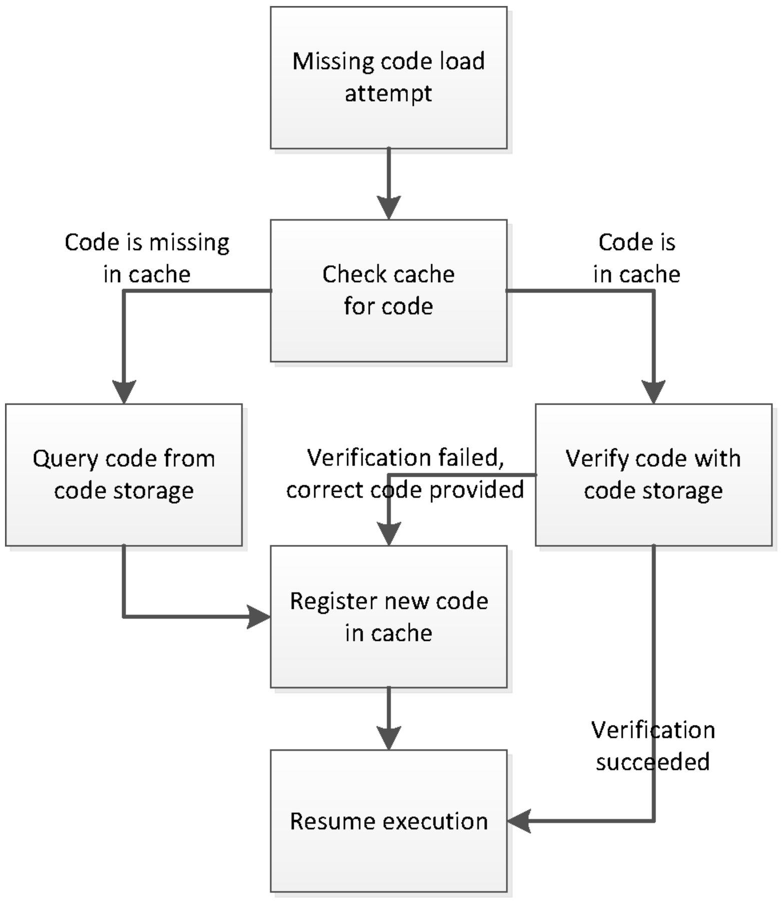
tion. On the Cloud host, the invocation is fired while a special Java classloader maintains and fetches all required code on the fly. Due to this approach, the application that is being executed does not have to care about code availability and versioning, as the underlying infrastructure handles these problems seamlessly and transparently. To provide an appropriate level of code encapsulation and allow multiple parallel executions with a different code base, CloudScale creates a separate classloader for each client request and treats them independently. This allows encapsulation of separate requests and provides the ability to load multiple different versions of the code at the same time.

On each class resolution request, CloudScale determines if the class has been already loaded by the corresponding classloader. In this case, the available code can be used without any additional checks, as code is not allowed to change during the same request execution. Otherwise, the classloader gets information about all available code versions for required set of classes from the *code cache* and asks the *class provider* in the target application JVM to detect the appropriate one. If none of the known code versions are correct, the class provider transfers the correct code, which can be used for application execution. This process is illustrated in Figure 4, where all possible classloader behaviors are shown.

Within the code distribution process described above, the code versioning problem requires some more explanation. Whenever suitable code has to be selected, target application has to select the appropriate code within available offers from cache, or determine that none of them matches. The simplest solution would be to send all available versions of the code itself to the target application; however, this introduces significant communication overhead and increases the load on the application's host. The best solution would be to use some notion of code version, but even if there was such a feature provided by Java, it would not protect against parallel changes made by different developers.

In the current implementation of CloudScale classloading framework, class binary size, and last modification date are used as version identifiers. This allows minimizing communication overhead and still lets to determine if the available code is the same or different from the one needed. This approach is not unique and is used in a set of other state-of-the-art applications, where files or documents have to be in sync (e.g., RSync[4], Apache Ant[5], GNU Make[6] and others). Evidently, this is not the only possible approach available. A set of alternatives was considered (e.g., using hash-codes, explicit versioning via version numbers, or partial code transfer), but we determined the selected heuristic approach is the simplest and

Figure 4. CloudScale code loading strategy


Missing code load attempt
Check cache for code
Code is missing in cache
Code is in cache
Query code from code storage
Verify code with code storage
Verification failed, correct code provided
Register new code in cache
Verification succeeded
Resume execution

fastest, while still reliable enough for practical applications. In addition, this approach allows for easier cache maintenance, as it enables detection of code updates (if the last modified time is newer), detection of older version usages (if the last modified time needed is older than the one available), and dropping of the old code that was not used for a long time.

Whenever a Cloud host faces the situation that a class is missing, it queries the client for the correct version of the class. While caching eliminates the need to transfer class code if it is already known to the Cloud host, the sequence of serial class loading request-responses still introduces a significant overhead. To reduce this, CloudScale classloading infrastructure has to deduce a set of possible following requests and batch the responses along with the requested class. For example, when a class is requested by the Cloud host, it is clear that all parent classes and interfaces that this class implements will be requested after it as well. In addition, when the requested class is available within the jar file, other classes from the same jar file are likely to be needed in the future. These assumptions significantly improve class loading performance for some cases, while introducing unnecessary transfer overhead for others. Currently, CloudScale uses classloader that provides the jar files whenever any class from it was requested, but we are considering other heuristics to improve class loading performance further.

The class loading feature of CloudScale is presented in more detail in (Zabolotnyi, Leitner, & Dustdar, 2013), which also contains numerical evaluation and overhead estimation of different class loading strategies and heuristics.

APPLICATION STATE MONITORING

The elastic Cloud computing model requires developers to carefully monitor the application's performance and resource demand and acquire only the appropriate amount of resources.

While monitoring is always a cornerstone of complex enterprise-scale applications, it is clear that elastic Cloud computing is effective only with smart and powerful monitoring facilities. Without appropriate monitoring tools, developers are not able to scale distributed applications properly to satisfy customers and effectively utilize Cloud resources. However, currently, Cloud monitoring solutions are not as fine-grained and effective as one would expect. Modern industrial solutions focus on low-level metrics such as RAM usage or CPU utilization, while we claim that applications should be monitored according to high-level metrics that determine how well the stated task is accomplished.

For example, for a Web service this is the request processing time, while for a Web site this is the number of users currently handled or the average page generation time. Additionally, these high-level metrics can simplify expressing and controlling service level agreements (SLA) for customers.

The monitoring approach that is used by CloudScale is based on the ideas of complex event processing (CEP) and event-based monitoring. The CloudScale monitoring framework relies on application components named *event emitters* that indicate their current status via events. Cloud hosts and Cloud Objects are obvious event emitters, but any Java code running within the application can act as an event emitter by producing events to the appropriate event stream. Produced events can be aggregated and processed within a CEP engine, allowing the definition of higher-level complex events.

The overall view of the monitoring framework is depicted in Figure 5. The application is running over a set of Cloud hosts, each hosting a set of Cloud Objects. Each of the shown components can act as event emitter. Locally, events are transmitted via API calls, but remote communication is handled by the same messaging infrastructure as used by the rest of CloudScale components.

Monitoring Event Hierarchy

As the main building blocks of the CloudScale monitoring framework are events produced by event emitters, let us describe the structure and hierarchy of existing events depicted in Figure 6. On the highest level, events split into two categories: predefined and custom. CloudScale monitoring framework predefines 15 runtime events (e.g., hosts and objects lifecycle events or resource usage events). Lifecycle events are triggered by CloudScale framework itself whenever the condition is met (e.g., when a Cloud Object starts to execute or host is shut down). This set of predefined events is comparable to available monitoring events in related systems (Michlmayr, Rosenberg, Leitner, & Dustdar, 2008; Karastoyanova, Leymann, Nitzsche, Wetzstein, & Wutke, 2006).

All predefined events contain event-specific information, which is omitted from Figure 6. For example, *Execution Started Event* consists of the

Figure 5. CloudScale monitoring framework overview

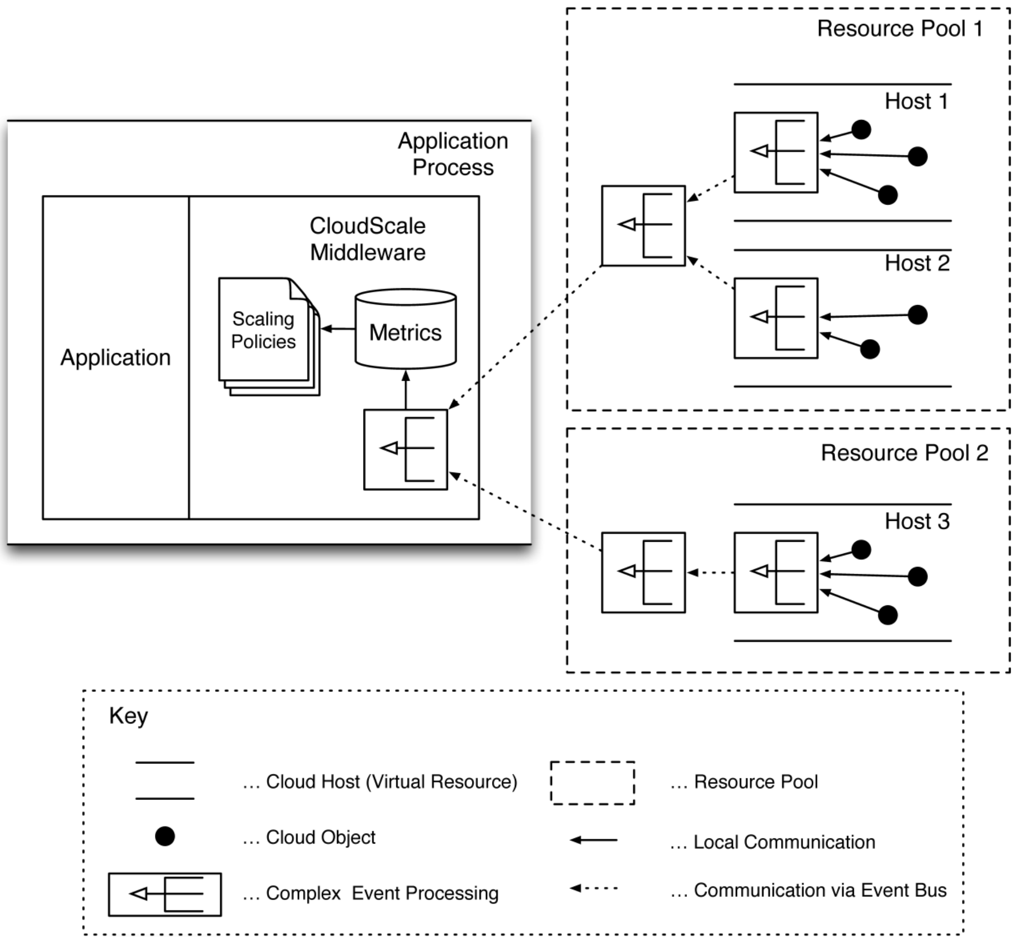
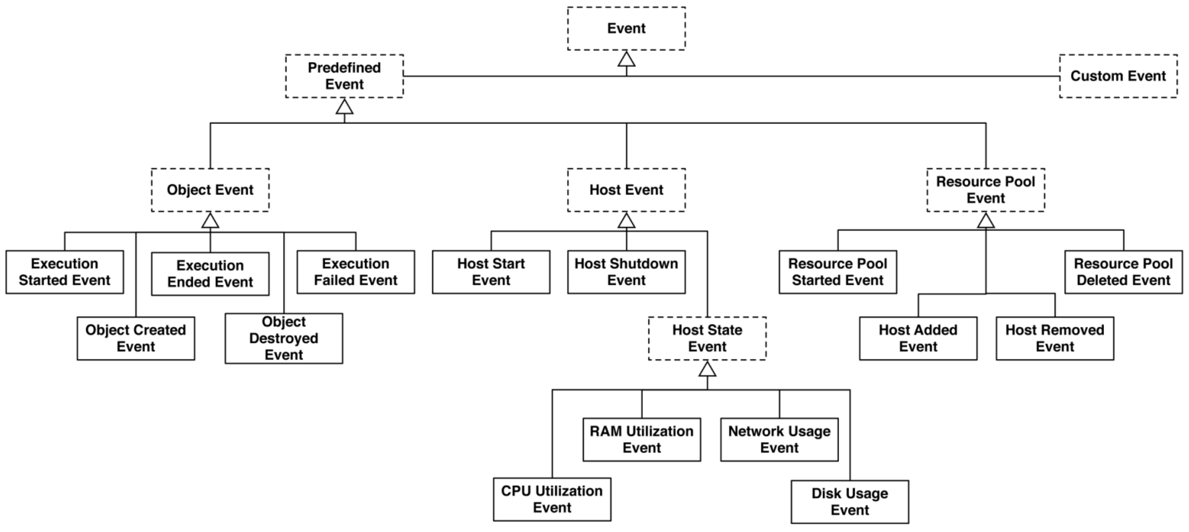


Figure 6. Monitoring event hierarchy in CloudScale



identifier of the Cloud Object, the name and parameters of the method, and generated execution identifiers as additional parameters.

With the help of custom events, any application-specific event can be emitted, which enables reacting to any high-level behavioral change of the application. For example, developers can control processing speed of the Web application by emitting custom events (see Table 3) and scale the application up or down, whenever processing speed changes. In the context of the example case application, developers may emit MyTestStateEvent (see Listing 2) whenever any test starts or completes.

CloudScale enables developers to trigger custom events by publishing events into an instance of MonitoringEventSink, injected by CloudScale via the @*EventSink* annotation (see Listing 2).

Scheduling Based on Monitored Values

The main use of the monitored values is to allow simple, while still expressive, elastic application behavior setup. In the case of CloudScale-based application, this is done within the scaling policy (defined in the Basic Notions section). Scaling policies are usually represented by a set of simple event-condition-action rules that define whether target application should scale up or down depending on the current or predicted application state.

While CloudScale offers a set of predefined scaling policies, currently developers are encouraged to write their own scaling policies that would fit the target application's needs and allow the best resource usage. At this time, this is a rather complicated task that can cause such problems

Table 3. Example of CloudScale custom events that allow fine-grained application elasticity

Metric Name	Description
RequestProcessingTime	Amount of time Cloud Object was processing user request.
RequestQueueLength	Current number of queued tasks that are waiting for processing.
AvgRequestsPerSecond	Average number of user requests arriving each second.
AvgDatabaseRequestTime	Average amount of time taken by the database query.
AvgTaskProcessingTime	Average time required to perform a particular task (e.g., logging).
WriteConflictsCount	Number of conflicts in database writes.
AbortedTransactionsCount	Number of aborted database transactions.

Listing 2. Triggering custom events

```
public class MyTestEventEmitter {
     @EventSink
     MonitoringEventSink eventSink;
     ...
     private void triggerEventOnTestStatus (TestState state) {
            CustomEvent myEvent = new MyTestStateEvent(...);
            eventSink.emitEvent(myEvent);
    }
 ...
 }
```

as synchronization issues, conflicting rules or oscillation (i.e., continuously triggering up- and down-scaling within short periods of time), but we are working on further ideas on how this can be simplified or even automated, while still achieving current levels of flexibility and control.

HYBRID CLOUD SUPPORT

Currently, two Cloud deployment models are popular: private and public Cloud deployment. Private Cloud deployment represents the usage of the private virtualized data center that is owned by the same company that owns the underlying infrastructure or provisioned by a vendor, specifically for the company. Public Clouds stand for the services that are available for the general public and offered by a separate provider (Mell & Grance, 2011). While private Clouds are usually of a limited size, public Cloud solutions are usually represented on such a huge scale that customers can assume it offers an infinite amount of resources.

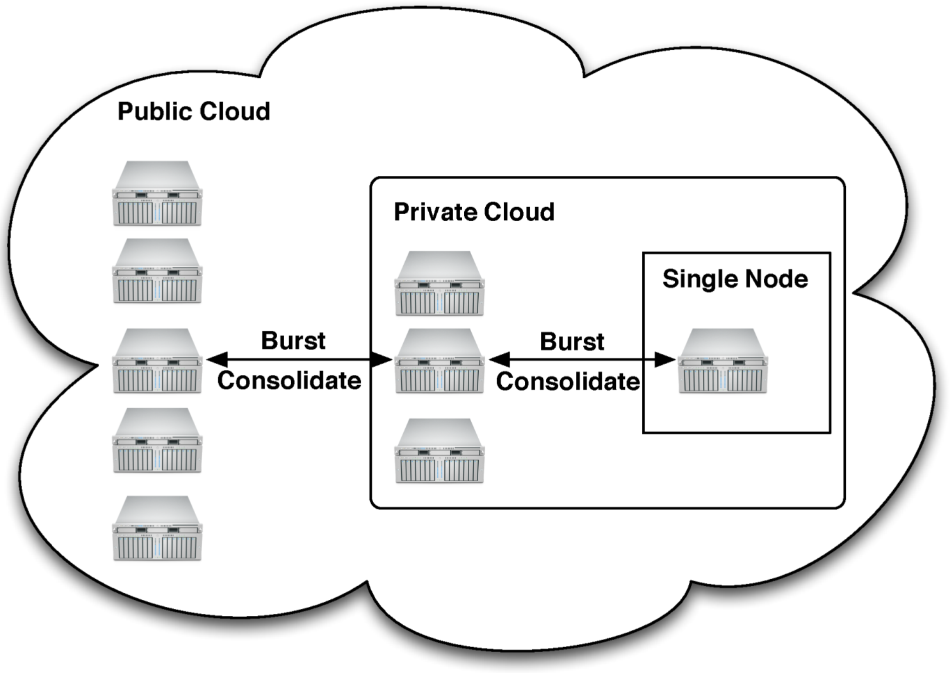
As the core idea of CloudScale is to conceal the Cloud management code from the target application and allow developers to focus on the business logic, CloudScale provides an efficient, fast and convenient way to switch between Cloud environments. While the Cloud management code is injected into the application after the compilation, CloudScale allows selecting the Cloud platform during application runtime. This allows avoiding Cloud provider API lock-in and being able to switch between different providers for testing and deployment purposes. As discussed earlier, this prevents from cluttering the business logic of the application and avoids writing the same Cloud-management boilerplate code in each target application.

Considering modern Cloud environments, due to API variance and resource management differences, applications have to be modified prior to running within the different environments. Because of this, the concept of the Cloud bursting is mainly a research idea that is quite hard to implement in practical applications. Cloud bursting represents the capability of an application to scale according to the demand not only within the available single Cloud resources, but over multiple Clouds as well. During low load conditions, such application can scale down to a single host, while as the load increases, additional resources are used from the set of private or public Clouds (see Figure 7).

Any application running over multiple Cloud platforms is not technically different from the one operating only in a single Cloud. However, there are a set of practical problems that have to be solved in order to operate in such heterogeneous environments. For example, firewalls are usually neglected within the homogenous Cloud environment, while it becomes quite problematic to configure communication between Cloud hosts deployed in a different environment, introducing the need of some tunneling to be able to communicate between hosts. Secondly, communication delay also has to be addressed. While within the Cloud, communication overhead is mostly homogenous and can be considered as constant; however, this overhead has to be considered if multiple Clouds are operated at the same time and resource access time is different depending on the Cloud they are deployed into. In addition, host performance is also different within the different Cloud environments (Li, Yang, Kandula, & Zhang, 2010).

With the architecture presented in CloudScale, it can be easily extended to operate over multiple Clouds. All CloudScale communication goes through the message server that can be instantiated within each Cloud platform. This allows running any CloudScale-based application in the Cloud

Figure 7. Basic three-phase Cloud Bursting model



given that target application can connect to the message server. To seamlessly transfer information between the Clouds, CloudScale uses the target application's host as a communication node between the hosts located in different environments.

While the complexity of operating with the different Cloud environments is hidden from the target application developer within CloudScale core code, there is one thing that developers have to consider while developing and running such applications. To successfully and efficiently run target applications in the Cloud bursting mode, scaling policy has to be written in awareness of the fact of multiple Cloud platform usage. Within the scaling policy, the scaling rules (discussed in more detail in section "Application State Monitoring") have to specify which actions have to be performed on each of the Cloud environments separately. However, the complexity of this task grows linearly with the number of environments that are being used as the same monitoring tools and metrics can be used to make a decision. More details regarding the implementation, usage and evaluation of Cloud bursting with CloudScale framework is described in (Leitner, Rostyslav, Gambi, & Dustdar, 2013).

APPLICATION DEVELOPMENT PROCESS

To illustrate the development process of CloudScale-based applications, we go through the set of steps necessary to bring Maven[7]-based applications to the Amazon EC2 Cloud. In more detail, this process is described in the CloudScale online documentation[8]. This process consists of three fundamental steps: at first we have to change the project setup to include CloudScale, then we have to select Cloud Objects and apply any other necessary annotations, and lastly, we have to configure CloudScale to scale the application according to our needs. The first step does not present any difficulties: pom.xml file has to be modified in order to specify dependencies to the CloudScale project and apply *AspectJ* annotations to the user code.

Applying CloudScale Annotations

The idea of the CloudScale is based on the notion of Cloud Objects. Cloud Objects are instances of resource-demanding classes that have to be distributed over the network. As the main task of JSTaaS application is to execute customer tests, the class that wraps each test suite execution is a good candidate for such a resource-intensive class. Furthermore, this class is strongly decoupled and requires minimal interaction with other components of the application.

Listing 3 shows the test execution class that is being distributed by CloudScale. It consists of mainly application-specific business logic with a number of CloudScale annotations added. As the *TestSuiteCloudObject* class is annotated with @ *CloudObject*, all interactions with the instances of this class are intercepted by CloudScale and scheduled to the appropriate Cloud hosts. In addition, to optimize performance, some method's parameters and return values are annotated with appropriate parameter passing annotations (see Interaction Patterns section) that allow treating parameters either as by-value or as by-reference. For example, as the statuses parameter of the *runCloudObject* method is not annotated by any specific annotation, it is treated as by-reference and all changes applied to this object are retransmitted to the target application.

Another important annotation is @*DestructCloudObject* on the *runCloudObject* method. This annotation specifies that this is the last invocation on this Cloud Object and after invocation of

Listing 3. The skeleton of the Test Execution Class

```
@CloudObject
 public class TestSuiteCloudObject {
       @CloudObjectId
       private UUID coId;

       @DataSource(name = "testresults")
       private Datastore datastore;

       public @ByValueParameter UUID getId() {
         return coId;
      }
      public void setSuite (@ByValueParameter TestSuite suite, int testId) {
                ...
      }
      @DestructCloudObject
       public void runCloudObject (TestSuiteExecution statuses, int suiteNr){
                ...
       }
 }
```

this method is finished, this Cloud Object can be destroyed. This allows optimizing resource usage and cleaning unnecessary objects from the Cloud hosts.

Separately, we would like to note the dependency injection feature of CloudScale. Two fields of this class (*coId and datastore*) are annotated with appropriate annotations to allow additional interaction with CloudScale framework. For example, the field annotated with *@DataSource* annotation enables the use of shared data passing mechanisms through the code of the class.

Configuring CloudScale

At this point, the JSTaaS application is already distributed by CloudScale. However, the distribution is happening in the so-called *debug mode*: instead of using separate Cloud hosts, CloudScale spawns new JVMs on the same host that the application is started. This mode is perfect for debugging and ensuring that everything works as expected prior to deploying the application to the Cloud. In order to deploy the application on the real Cloud, an appropriate configuration has to be provided.

There are a number of ways to configure CloudScale, described in more detail in the online documentation. Here, we will configure the framework through system properties. In order to configure CloudScale, the system property *Cloudscale.configuration* has to specify either the path to an XML file containing a serialized CloudScale configuration, or the name of a class that has a static method with the *@CloudScale-ConfigurationProvider* annotation. This method should return an instance of *CloudScaleConfiguration*. During application runtime, on the first interaction, CloudScale will load its configuration from the specified place.

If the appropriate Amazon EC2 configuration is specified, the application can already be distributed in Amazon EC2 Cloud. This EC2 configuration specifies platform-dependent parameters important for CloudScale. For example, developers can specify required size of the instance to start or the virtual host image id with preconfigured CloudScale service that will be used for spawning new cloud hosts. CloudScale online documentation describes how such image can be built on any platform and provides a reference to the public image in Amazon EC2 cloud.

However, the default host managing policy will not be optimal for this application. In order to optimize it, we have to create our own scaling policy (see Listing 4) based on the monitoring information described in the section on Appli-

Listing 4. A scaling policy example

```
public class ScalingPolicy implements IScalingPolicy {
      @Override
      public boolean scaleDown (IHost host, IHostPool hostPool) {
         ...// here we define if the specified host should be shut down
      }
      @Override
      public IHost selectHost (ClientCloudObject co, IHostPool host-
Pool)        {
         ... // here we define where to deploy the new Cloud object
      }
 }
```

cation State Monitoring, and specify it in the configuration. After these changes, our JSTaaS application is fully capable of running over the Amazon EC2 Cloud, where we can further adapt it to suit our needs.

CLOUDSCALE EVALUATION

In this section, we will briefly contrast application development using CloudScale with building an IaaS application directly on top of Amazon EC2 (without specific tooling outside of the EC2 API) and using a PaaS service, such as Google AppEngine. Our goal here is to show what advantages a "middle-of-the-road" solution such as CloudScale provides.

Starting with API complexity, CloudScale requires knowledge of a reasonably small amount of API functions, while offering large capabilities for application development. This is mainly due to the way applications are built on top of CloudScale and the amount of necessary changes to the target application. While both EC2 and AppEngine assume that developers will create applications specifically for their platforms using the provided API, CloudScale aims at seamless development and ease of taking existing Java based distributed applications to the Cloud.

In addition to that, Cloud application debugging is somewhat simpler with CloudScale than with EC2 or AppEngine. This is mostly due to the special scaling debug mode of CloudScale, which scales applications in the sandbox on the local machine, while developers of applications for EC2 or AppEngine can only debug application while the target platform is available and only through the limited set of tools available for the selected platform. It should be noted that an emulator for AppEngine is available, but practical experience has shown that most testing still needs to be carried out in the real PaaS environment.

A core advantage of the IaaS approach is that it always provides complete freedom as to which frameworks and application architecture designs are supported. CloudScale, on the other hand, is by its nature restricted to the Java programming language. Other than that, the restrictions imposed by CloudScale are minimal. AppEngine, on the other hand, induces quite significant limitations on application design, and restricts application developers, both with regard to what API functions they can use and what architecture the application needs to follow. Another thing that IaaS approach is good is for providing full access to the backend servers, thus enabling developers with complete flexibility and control over the resource usage and operating system configuration.

CloudScale aims to hide the complexity of virtual machines so that the developers can build the Cloud application without even controlling virtual machines; however, it does not forbid developers to modify the virtual machine as long as the core components of the CloudScale are still running. The online documentation also has instructions available on how to build custom CloudScale server images which can be modified in any way.

However, clearly the AppEngine model has significant advantages as well. One example of such a benefit of AppEngine is code distribution and application scalability. While AppEngine scales applications mostly automatically, for EC2-based applications, developers have to create their own rules and approaches to achieve elastic application scaling. From this point of view, CloudScale provides a reasonable alternative. Scalability is achieved by injected code and appears to be seamless to the developer, while the scaling rules can be provided separately within the convenient and powerful instrument based on the flexible monitoring framework that allows controlling not only basic parameters such as CPU load and memory usage, but also high-level application-specific metrics.

Finally, CloudScale offers support for applications that are scaling over multiple Clouds (forming so-called hybrid Clouds), which allows minimizing application operating costs and extends application flexibility beyond the limits of one Cloud provider. This model is not supported by AppEngine at all. Using an IaaS service such as EC2, it is possible to implement hybrid Clouds, but this requires a significant amount of development and configuration work. In contrast, setting up a hybrid Cloud with CloudScale comes at almost no effort to the developer.

In Table 4, we present a qualitative summary of the features that we consider significant for application development. We claim that CloudScale significantly simplifies the application development process, hides complexity of code distribution and Cloud management, while providing convenient and configurable debugging and development experience. Therefore, we think that developers (especially the ones new to Cloud computing) will benefit from using CloudScale and will be able to develop applications and take them to the Cloud faster than with existing tools.

Every platform provides some benefits to developers, while introducing measurable overhead to the developed system. To investigate the extent to which the descriptive approach offered by CloudScale influences application performance, we performed a numerical evaluation, which was originally presented in a separate paper, currently available as a technical report (Leitner, Zabolotnyi, Hummer, Inzinger, & Dustdar, 2013).

The main goal of our numerical evaluation is to compare the performance of the same application built on top of CloudScale and on an IaaS platform (e.g., OpenStack) directly. To investigate this, we developed the core functionality of the JSTaaS application introduced in Example Case section above. As the main goal was to evaluate overhead introduced by CloudScale, we designed both applications to have same behavior and reuse as much business logic code as possible. Additionally, we focused on scenarios where the number of Cloud hosts (CH) is fixed.

Each solution was tested in the same environment with the equivalent test setup consisting of 20 parallelizable long-running test suites, sched-

Table 4. Feature comparison of CloudScale and alternative solutions

Feature	Amazon EC2	CloudScale	Google AppEngine
Complexity of API	Small	Small	Significant
Amount of Platform Interaction Code	Significant	Very small	Small
Application Debugging Simplicity	Complicated	Simple	Reasonable
Architecture Limitations	None	Small	Significant
Scaling Configuration Convenience	Manual/None	Good	Basic
Code Distribution and Update	Manual/None	Semi-Automatic	Automatic
Monitoring Features	Manual/None	Advanced	Basic
Backend Server Access	Full	Restricted	None
Hybrid Cloud Support	Manual	Built-in	None
Programming Language Support	Any	Java	Java, Python, PHP
Provider Lock-in	Small	Very small	Significant

uled evenly over the available set of Cloud hosts. Figure 8 illustrates mean results of running test setup described above with the different numbers of Cloud hosts available to the application. From this figure, one notices that CloudScale (blue dotted line) indeed introduces a significant overhead in comparison to pure OpenStack implementation (red line). However, after some investigation, we discovered that this overhead is mainly caused by remote classloading feature in CloudScale.

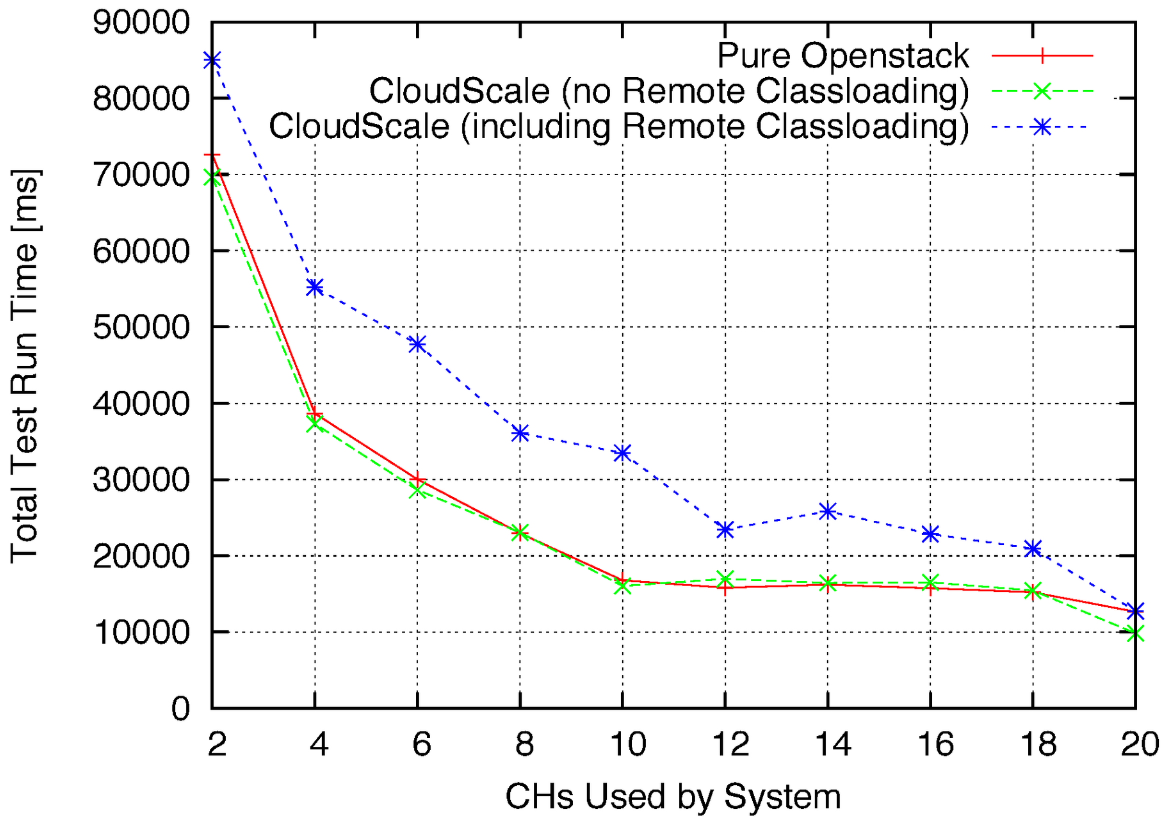
However, in OpenStack implementation, classloading is unnecessary, as all code is already available in Cloud hosts. Hence, to make our comparison fairer, we decided to run CloudScale version of the JSTaaS with all code pre-cached on the Cloud hosts. This evaluation run (green dotted line) appeared to be as fast as the OpenStack-based implementation of the application.

This evaluation allowed us to conclude that CloudScale does introduce some overhead, which is mainly due to remote classloading feature. To further improve user experience, we decided to explore the ways to optimize this in our future work.

FUTURE RESEARCH DIRECTIONS

While the current version of CloudScale is already stable and has a significant number of features, we are still actively working on further improve-

Figure 8. Total test runtime dependency with number of Cloud hosts used



ments. From a technical point of view, we still need to improve the code base, fix known issues and improve the documentation. To extend functionality of CloudScale, we are trying to add support of additional IaaS Cloud providers and improve general performance and stability of CloudScale.

From the research point of view, we continue to investigate the ways to automate the steps that currently have to be done manually and find new possible applications for the CloudScale framework. For example, we are working on the smart profiling system that may allow us to profile a Java application to detect classes that can be used as Cloud Objects. Another direction is to improve monitoring behavior to simplify creation of scaling policies and allow automatic application scaling according to historical load and predicted behavior. In addition, we are working on ways to further integrate shared data passing model and make it as seamless as the other two. Finally, we are developing a useful and convenient Cloud Object migration mechanism that will provide developers a way to balance current load on the Cloud hosts and migrate Cloud Objects between hosts.

To receive more usage feedback, we are currently popularizing CloudScale among researchers, students, and developers. This will allow us to verify our claims, collect feedback and improve the overall CloudScale usage experience. In the future, we also plan to release a hosted demo version of CloudScale, which will allow potential users to verify if CloudScale fits their needs. Additionally, to distinguish CloudScale from other tools and platforms with the same name, we plan to rebrand CloudScale to JCloudScale and move our public repository to GitHub.

Finally, we are applying CloudScale as a Cloud migration tool for a number of existing popular applications and frameworks, including service composition engine JOpera (Pautasso & Alonso, 2005) and Apache JMeter (Halili, 2008).

CONCLUSION

CloudScale facilitates the simplification of Cloud IaaS-based application development by handling most of the infrastructure and distribution-related issues under the hood, allowing developers to focus on the application's business logic implementation. CloudScale is injected into application code after compilation based on the Java annotations, that enables isolating application's business logic from the boilerplate of Cloud interaction and communication code. Therefore, CloudScale code injection can be simply enabled or disabled depending on the situation.

While CloudScale tries to be as seamless for developers as possible, it can be flexibly configured to fit developer needs. Following this ideology, CloudScale is configured by Java annotations or system properties, which eliminates any influence on the application execution logic and behavior. In this chapter, we introduced the main ideas and concepts behind CloudScale, illustrated how important Cloud distribution problems are solved seamlessly for target applications, and demonstrated the main steps of distributing applications with CloudScale. While CloudScale is reasonably stable and easy to use, it is still under development and we are continuing to work on further improvements to be able to take user applications into the Cloud as seamlessly and flexibly as possible.

REFERENCES

Alvaro, P., Condie, T., Conway, N., Elmeleegy, K., Hellerstein, J. M., & Sears, R. (2010). Boom Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. In *Proceedings of the 5th European Conference on Computer Systems,* (pp. 223-236). ACM. doi:10.1145/1755913.1755937

Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., & Konwinski, A. et al. (2010). A View of Cloud Computing. *Communications of the ACM*, *53*(4), 50–58. doi:10.1145/1721654.1721672

Bezemer, C.-P., Zaidman, A., Platzbeecker, B., Hurkmans, T., & Hart, A. (2010). Enabling multi tenancy: An industrial experience report. In *Proceedings of the IEEE International Conference on Software Maintenance*. Washington, DC: IEEE Computer Society. doi:10.1109/ICSM.2010.5609735

Bhardwaj, S., Jain, L., & Jain, S. (2010). Cloud Computing: A Study of Infrastructure as a Service (IaaS). *International Journal of Engineering and Information Technology*, *2*(1), 60–63.

Binz, T., Breiter, G., Leyman, F., & Spatzier, T. (2012). Portable Cloud Services Using TOSCA. *IEEE Internet Computing*, *16*(3), 80–85. doi:10.1109/MIC.2012.43

Calheiros, R. N., Vecchiola, C., Karunamoorthy, D., & Buyya, R. (2012). The Aneka platform and QoS-driven resource provisioning for elastic applications on hybrid Clouds. *Future Generation Computer Systems*, *28*(6), 861–870. doi:10.1016/j.future.2011.07.005

Cattell, R. (2011). Scalable SQL and NoSQL Data Stores. *SIGMOD Record*, *39*(4), 12–27. doi:10.1145/1978915.1978919

CloudBees. (2013). *CloudBees: The Java PaaS Company*. Retrieved from http://www.cloudbees.com/

Contrail. (2013). *Contrail Open Computing Infrastructure for Elastic Services*. Retrieved from http://contrail-project.eu/

Gunarathne, T., Wu, T.-L., Qiu, J., & Fox, G. (2010). MapReduce in the Clouds for Science. In *Proceedings of the 4th IEEE International Conference on Cloud Computing Technology and Science*. Los Alamitos, CA: IEEE Computer Society.

Halili, E. (2008). *Apache JMeter: A Practical Beginner's Guide to Automated Testing and performance measurement for your websites*. Packt Publishing.

Karastoyanova, D., Leymann, F., Nitzsche, J., Wetzstein, B., & Wutke, D. (2006). Parameterized BPEL Processes: Concepts and Implementation. In *Proceedings of the International Conference Business Process Management.* Academic Press.

Krintz, C. (2013). The AppScale Cloud Platform: Enabling Portable, Scalable Web Application Deployment. *IEEE Internet Computing*, *17*(2), 72–75. doi:10.1109/MIC.2013.38 PMID:23828721

Kung, H. T., & Robinson, J. T. (1981). On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, *6*(2), 213–226. doi:10.1145/319566.319567

Leitner, P., Inzinger, C., Hummer, W., Satzger, B., & Dustdar, S. (2012). Application-Level Performance Monitoring of Cloud Services Based on the Complex Event Processing Paradigm. In *Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications.* IEEE. doi:10.1109/SOCA.2012.6449437

Leitner, P., Rostyslav, Z., Gambi, A., & Dustdar, S. (2013). A Framework and Middleware for Application-Level Cloud Bursting on Top of Infrastructure-as-a-Service Clouds. In *Proceedings of the 6th IEEE/ACM Utility and Cloud Computing Conference.* IEEE/ACM. doi:10.1109/UCC.2013.39

Leitner, P., Rostyslav, Z., Waldemar, H., Inzinger, C., & Dustdar, S. (2013). *CloudScale: Efficiently Implementing Elastic Applications for Infrastructure-as-a-Service Clouds*. Vienna University of Technology. Retrieved from http://stockholm.vitalab.tuwien.ac.at:8090/TechReportGenerator/reports/TUV-1841-2013-1.pdf

Leitner, P., Satzger, B., Hummer, W., Inzinger, C., & Dustdar, S. (2012). CloudScale - A Novel Middleware for Building Transparently Scaling. In *Proceedings of the ACM Symposium on Applied Computing.* ACM. doi:10.1145/2245276.2245360

Li, A., Yang, X., Kandula, S., & Zhang, M. (2010). CloudCmp: Comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, (pp. 1-14). ACM.

Mell, P., & Grance, T. (2011). The NIST Definition of Cloud Computing (draft). *NIST Special Publication*, *800*, 145.

Michlmayr, A., Rosenberg, F., Leitner, P., & Dustdar, S. (2008). Advanced Event Processing and Notifications in Service Runtime Environments. In *Proceedings of the 2nd International Conference on Distributed Event-Based Systems.* Academic Press. doi:10.1145/1385989.1386004

Mietzner, R., Unger, T., & Leymann, F. (2009). Cafe: A Generic Configurable Customizable Composite Cloud Application Framework. *Lecture Notes in Computer Science*, *5870*, 357–364. doi:10.1007/978-3-642-05148-7_24

Paul, L., & Simon, M. (2013). *OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC*. Retrieved from https://www.oasisopen.org/committees/tc_home.php?wg_abbrev=tosca

Pautasso, C., & Alonso, G. (2005). JOpera: A Toolkit for Efficient Visual Composition of Web Services. *International Journal of Electronic Commerce*, *9*(2), 107–141.

Pierre, G., & Stratan, C. (2012). ConPaaS: A Platform for Hosting Elastic Cloud Applications. *IEEE Internet Computing*, *16*(5), 88–92. doi:10.1109/MIC.2012.105

Vaquero, L. M., Rodero-Merino, L., & Buyya, R. (2011). Dynamically scaling applications in the cloud. *SIGCOMM Computer Communicaitons Review, 41*(1), 45-52. http://doi.acm.org/10.1145/1925861.1925869

Vinoski, S. (2008). Convenience Over Correctness. *IEEE Internet Computing*, *12*(4), 89–92. doi:10.1109/MIC.2008.75

Zabolotnyi, R., Leitner, P., & Dustdar, S. (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.* San Francisco, CA: Academic Press. doi:10.1109/PESOS.2013.6635974

KEY TERMS AND DEFINITIONS

Cloud Bursting: The application behavior that provides scalability according to demand, not just within the available single Cloud resources, but over multiple Clouds as well.

Cloud Computing: The elastic computing over virtualized resources, provided via pay-as-you-go model.

Cloud Host: The virtual machine in the Cloud operated by CloudScale.

Cloud Object: An instance of a resource-demanding class that is distributed by CloudScale over the Cloud hosts.

Data Object: The object that is used to transfer data between classes or application components.

IaaS: Infrastructure as a service level of Cloud computing.

PaaS: Platform as a service level of Cloud computing.

Target Application: The application distributed by CloudScale.

ENDNOTES

1 http://code.google.com/p/cloudscale/
2 https://www.heroku.com/
3 http://aws.amazon.com/
4 http://rsync.samba.org/
5 http://ant.apache.org/
6 http://www.gnu.org/software/make/
7 http://maven.apache.org/
8 http://code.google.com/p/cloudscale/wiki/FirstSteps