PaaS Performance Evaluation Methodology

MARCIN JARZĄB
Department of Computer Science, AGH University of Science and Technology
Samsung Research and Development Poland
KRZYSZTOF ZIELIŃSKI, SŁAWOMIR ZIELIŃSKI and KAROL GRZEGORCZYK
Department of Computer Science, AGH University of Science and Technology
and
MAREK PIASCIK
Samsung Research and Development Poland

This paper proposes a PaaS performance assessment methodology which is oriented towards cloud computing attributes such as resource pooling, rapid elasticity and measured service. PaaS evaluation procedure requirements and taxonomy have been specified in this context. The innovative concept exploited by the proposed methodology is its completeness, evidenced by stress, dynamicity, and stability testing scenarios. A theoretical analysis of PaaS under test load conditions has been provided for the each proposed testing scenario. This allows interpretation of the experimental study performed using private and public cloud environments. Results validate the proposed methodology and establish it as a useful tool for benchmarking PaaS deployments.

Keywords: PaaS, QoS, performance, evaluation, qualitative, quantitative

1. INTRODUCTION

Platform as a Service (PaaS) [Cohen 2013] is a cloud computing service model that bridges the gap between business requirements, policies, available runtime resources and development of end-user solutions. Instead of relying on standardized software capabilities, PaaS users have more control over the architecture, quality of service, user experience, data models, identity, integration, and business logic. As a result, PaaS can be used to build and deliver customized applications or information services and support DevOps [Hosono et al. 2011] practices which include self-service, automated provisioning, continuous integration, and continuous delivery. On the other hand, PaaS must exploit the underlying infrastructure in order to satisfy IT objectives such as optimizing resource utilization, administration of resources across multiple tenants, workload processing scalability and performance optimization. Validation of these requirements should acknowledge attributes of the cloud computing model as defined by NIST [Mell and Grance 2009], including: (i) on-demand self-service, (ii) broad network access, (iii) resource pooling, (iv) rapid elasticity, and (v) measured service.

PaaS evaluation procedures should determine to what extent a particular solution is optimized not only for traditional static single-tenant deployment but also for multi-tenancy in the context of the above mentioned cloud environments. Multi-tenancy enables per-consumer customization of applications and services by changing runtime configuration settings instead of provisioning new instances. As user demand grows or shrinks, an elastic and distributed platform should re-size the application resources based on utilization, quality of service and cost optimization policies. When resources are added, subtracted, or moved, the automated service management components should dynamically re-wire resource connections.

The goal of this paper is to propose a PaaS performance evaluation methodology oriented towards cloud computing attributes. The evaluation procedure built in accordance with this methodology might be applied to existing or planned installations of cloud platforms, and should result in selection of the most suitable solution for a given application, help identify existing bottlenecks

and support assessment of the efficiency of key PaaS characteristics.
The proposed approach is the outcome of theoretical analysis of the platforms (also referred to as Systems Under Test - SUTs), validated by experimental studies of real cloud installations, both public and private. The presented evaluation methodology concerns quantitative and qualitative metrics and is based on a black-box approach in order to more accurately reflect practical use cases which represent various common application categories.
The paper is organized as follows: Section 2 provides a short review on current research in related topics. Section 3 discusses requirements which apply to the evaluation procedure. Section 4 presents a generic PaaS architecture model and singles out the elements which are of key relevance to cloud user experience. In Section 5, a procedure for testing PaaS performance is introduced. The procedure has been evaluated in a series of test scenarios, which are presented in Section 6, executed on public and private cloud resources. A discussion of the results is presented in Section 7. The paper ends with Section 8, which lists key conclusions drawn from the described research.

2. STATE OF THE ART

The most recent available research study regarding selection of PaaS platforms for end users focuses on business aspects and qualitative evolution. In [Geene 2012], a five-steps PaaS selection procedure is proposed. It begins with a business strategy for application definition. Subsequently, the availability of essential services for application development, such as databases, integration and security services is verified. Later steps address application portability, language support and availability of experienced human resources. Very similar concepts can be found in [Geene 2013]. The authors of [Ramchandani 2012] describe the process of choosing and implementing a suitable enterprise PaaS. Their study extends the evaluation criteria with aspects of architectural flexibility which indicate that PaaS solutions are built to decouple hardware from virtualization, allowing enterprises to pick the virtualization technology of choice and adopt a VM orchestration solution which decouples infrastructure services and applications entirely from the underlying orchestration layer. The referred papers provide very important selection criteria but they almost entirely neglect performance aspects. In [Orlando 2011] the selection criteria are extended with elastic scalability requirements. The notion of PaaS performance testing is addressed in [Blokland 2013] but this proposal is limited to scalability testing only. In order to define more generic performance test procedures for clouds, [Avram 2010] proposes four types of tests and five native application benchmarks running on five different cloud platforms. To perform measurements the authors used WebMetrics' services, sending requests at various intervals from various locations worldwide over the period of one month. Two of the cloud platforms, namely Salesforce.com and Google App Engine, provide PaaS services, while the other three, i.e. Amazon, Rackspace and Terremark, provide IaaS services. The only metric evaluated by this study was the average latency of retrieving data objects of different sizes. Recent papers [Agnihotri and Sharma 2014] [Mohammad and Mcheick 2012] [D.Jayasinghe et al. 2013] addressing PaaS improvements, scalability testing and performance also do not propose a generic PaaS evaluation methodology. The authors of [Agnihotri and Sharma 2014] focus on evaluating PaaS scalability and introduce a graphical model named Scalability Improvement System (SIS). The proposed scalability improvement metrics concern scaling down delays, scaling down the usage cost, scaling up connectivity and scaling down the implementation cost. [Mohammad and Mcheick 2012] presents extensive experimental measurements which reveal variance in the performance and scalability of clouds in two nontrivial scenarios. In the first scenario public Infrastructure as a Service clouds were targeted – the case concerns a multi-tier application migrated from a traditional datacenter to one of the three IaaS clouds. In the second scenario a similar study was performed upon three private clouds built using three mainstream hypervisors. These studies used the RUBBoS benchmark and compared its performance and scalability when hosted in Amazon EC2, Open Cirrus, and Emulab. A very important conclusion from this work is that a best-performing configuration in one cloud may become the worst-performing configuration in another cloud. This emphasizes the

importance of a proper PaaS performance evaluation methodology, much like the one presented in our paper. In [Reitbauer et al. 2011] interesting aspects related to the impact of virtualization on performance management in the context of running middleware within guest OS instances are evaluated. The paper precisely describes the relation between performance metrics related to virtualized infrastructures and hosted applications, including a testing methodology for such deployment architectures. The VMware vFabricPaaS Planning Service [VMware 2011] consulting program provides in-depth understanding of successful PaaS deployment through specification of related processes and their implementation according to production scenarios. It also defines blueprints showing how to monitor performance and application management, including metering and chargeback.

Interesting work has been performed by the [Spec 2012] committee which elaborates generalized cloud testing systems. Their specification recommends a common framework for cloud benchmarks, application workloads and performance metrics and shows how they can be adapted to the cloud context.

There are also research activities in the area of PaaS evaluation methodology which concentrate on adaptive management to automatically satisfy predefined SLA contracts. In [Zhang et al. 2012] the approach bases on the so-called LQN (Layer Queue Network) model as a performance model for PaaS. [Gunther 2007] introduces such concepts as iterative cycle of improvement called "The Wheel of Capacity Planning" and Virtual Load Testing, which provides a highly cost-effective method for assessing application scalability. The book facilitates rapid forecasting of capacity requirements based on opportunistic use of available performance data and tools so that management insight is expanded.

Unfortunately, none of these research and standardization activities provide a comprehensive methodology for evaluation of PaaS, including qualitative and quantitative elements. The presented testing environments instead focus on selected aspects of either PaaS functionality or performance.

In this paper we propose a different approach, combining various metrics and strategies targeted for PaaS environments. The innovative concept presented in the paper is a PaaS evaluation methodology utilizing qualitative and quantitative metrics. Based on the generic architecture of a PaaS we define specific use cases which reflect the organization of testing scenarios and affect selected aspects, related to stress, dynamicity and stability testing. The scenarios help us observe system behaviour and identify the maximum operating capacity, which is a crucial requirement for cloud providers wishing to define SLAs for potential customers. The presented methodology is a detailed procedure and consists of a series of steps which specify specific activities to be performed in order to evaluate any PaaS implementation. In our opinion the presented work is a solid foundation for definition of industry benchmarks for PaaS-based clouds.

3. PAAS EVALUATION PROCEDURE REQUIREMENTS

Evaluation of PaaS platforms is a very complex process, mostly because the platforms themselves are sophisticated distributed systems deployed over virtualized infrastructures. Evaluation is a multi-aspect procedure which may be constructed in accordance with the taxonomy presented in Fig. 1. A single path from the root of the taxonomy tree to its leaves categorizes an evaluation scenario. For example, an evaluation scenario might be categorized using the following path: **Evaluation** → **Quantitative** → **Performance** → **Local** → **White box** → **Functional** → **Dynamicity**.

Note, however, that a single path in the taxonomy tree does not define a single scenario; rather it denotes a class of test procedures to be performed. A complete multi-aspect evaluation of PaaS should cover all possible paths with a large number of possible scenarios.

The meaning of the proposed taxonomy in the context of evaluation scenarios can be explained by specifying taxonomy tree nodes as follows.

Quantitative vs. Qualitative evaluation. Quantitative evaluation describes system perfor-

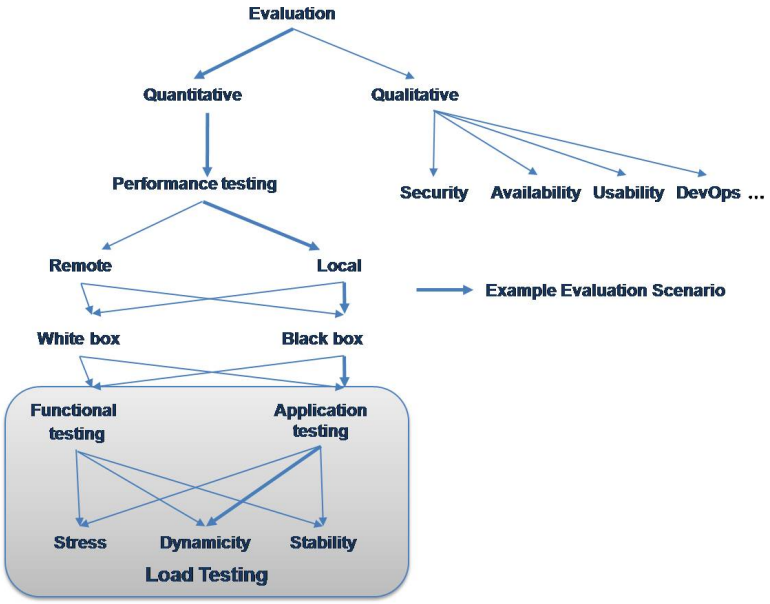

Figure 1: Taxonomy of evaluation scenarios

mance. Therefore this aspect refers to performance testing. System performance can be measured using synthetic load, generated in accordance with a particular test design. System operation can be evaluated under artificial load generated by testing tools or by a real application serving requests coming from a simulated population of users. On the other hand the goal of qualitative analysis is complete, detailed description and comparison of functional characteristics of the PaaS. Those features can be divided into the following groups: (i) security, (ii) high availability, (iii) usability, (iv) DevOps, and many others.

Performance testing. This is an essential element of quantitative analysis. It can serve various purposes, e.g., (i) demonstrate that the system meets performance criteria, (ii) compare two systems to find out which performs better or (iii) identify parts of the system or workload patterns that cause the system to perform poorly. The specification of a performance testing evaluation procedure should take into account the attributes described below.

Remote vs. Local. Regardless of the deployment mode (private, public or hybrid), the geographical distribution of request processing by geo-location services has to be considered as even private clouds may span multiple physical locations. The deployment topology of the benchmark workload generators must reproduce real user locations in respective availability zones of the PaaS. When a public PaaS cloud is considered, testing can be performed remotely from the respective users' locations. On the other hand, when the PaaS is private or hybrid, the assessment should be conducted locally, i.e. on the provider side.

White box vs. Black box. In this case it is necessary to choose either a white-box or a black-box approach to the evaluation procedure. Black-box testing is often preferred because the latter requires instrumentation of PaaS building blocks. In the case of PaaS testing, black-box evaluation relies on knowledge about the PaaS architectural components and their services, which may be defined using a generic, partially abstract architecture. Black-box testing is also simpler to implement and easier to repeat using different PaaS platforms. On the other hand, a white-box approach requires specific code instrumentation and detailed knowledge of the internal design of PaaS components, but enables more detailed monitoring features, and, as a consequence, provides more insight into potential performance bottlenecks.

Functional vs. Application. Functional evaluation concerns testing a system under artificial load conditions custom-tailored to test selected PaaS operations. In contrast, application

evaluation concerns the application characteristics that need to be addressed while developing enterprise benchmarks for PaaS, which are as follows:

—**Multitiered** nature of applications, consisting of the presentation, application and persistence tiers, deployed on distributed instances of virtual machines.

—**User scaling**, i.e., the capability of a system to scale up to satisfy increased user load. It is crucial to understand how the system behaves under increasing load and also to identify the limits beyond which performance may degrade substantially.

—**Vertical** (scaling up) and **horizontal** (scaling out) scaling which means that scalable systems are designed and constructed in such a way that each tier can scale independently to meet the growing user demand.

—**Secure interactions** where both secure and non-secure access is provided. It is important to understand the impact of enabling security to the overall performance of the application running over PaaS.

—**Session maintenance** is an important factor for ensuring the availability of applications deployed in a PaaS environment. Session persistence is an expensive concern and an important performance characteristic that should be examined.

—**Service and data availability** is also critical. Due to the large number of components distributed over multiple tiers, obtaining an accurate estimate of the availability of PaaS is a complex and difficult task.

Load testing is the process of generating demand for a system or device and measuring its response. Its goal is to observe a system's behaviour under both normal and anticipated peak load conditions. It helps identify the maximum operating capacity of the PaaS, detect bottlenecks and determine which elements are responsible for performance degradation. Load testing is performed to observe how a system operates in terms of responsiveness and stability under a particular workload. Because load testing procedures (and, consequently, test cases and scenarios) are conducted in the context of performance-related metrics based on assumed load characteristics, they might be categorized as one of the following three classes:

—**Stress testing**. In these tests the load is increased to the maximum level which still results in stable operation of the SUT.

—**Dynamicity testing**. In dynamicity tests the load is increased in steps, e.g. according to Heaviside's unit step function [Abramowitz 1964]. The time it takes the SUT to reach a new steady state and its performance level are duly observed. This test category evaluates the response of architectural elements to rapid load changes.

—**Stability testing**. In stability tests the load function resembles a stepped wave, parameterized by frequency and amplitude. The critical frequency and amplitude which necessitate operational changes (e.g. launching a new artefact) are observed. This test category evaluates sensitivity to changing conditions for a specific architectural element.

The presented evaluation taxonomy raises question about the most appropriate selection of testing procedures. Taking into account the PaaS evaluation aspects, the following requirements should be satisfied: (i) exploit as little knowledge about the PaaS internal construction as possible, i.e. rely only on a very generic PaaS architecture model, (ii) allow for remote and local testing scenarios, (iii) allow for both functional and application-oriented load testing.

4. GENERIC PAAS ARCHITECTURE MODEL

As discussed in the previous section, black-box PaaS testing should assume a generic PaaS model. This means that only selected services should be considered by evaluation scenarios when functional testing is performed. Relying on generic knowledge about the PaaS structure does not contradict white-box testing as long as it does not require any additional information collected

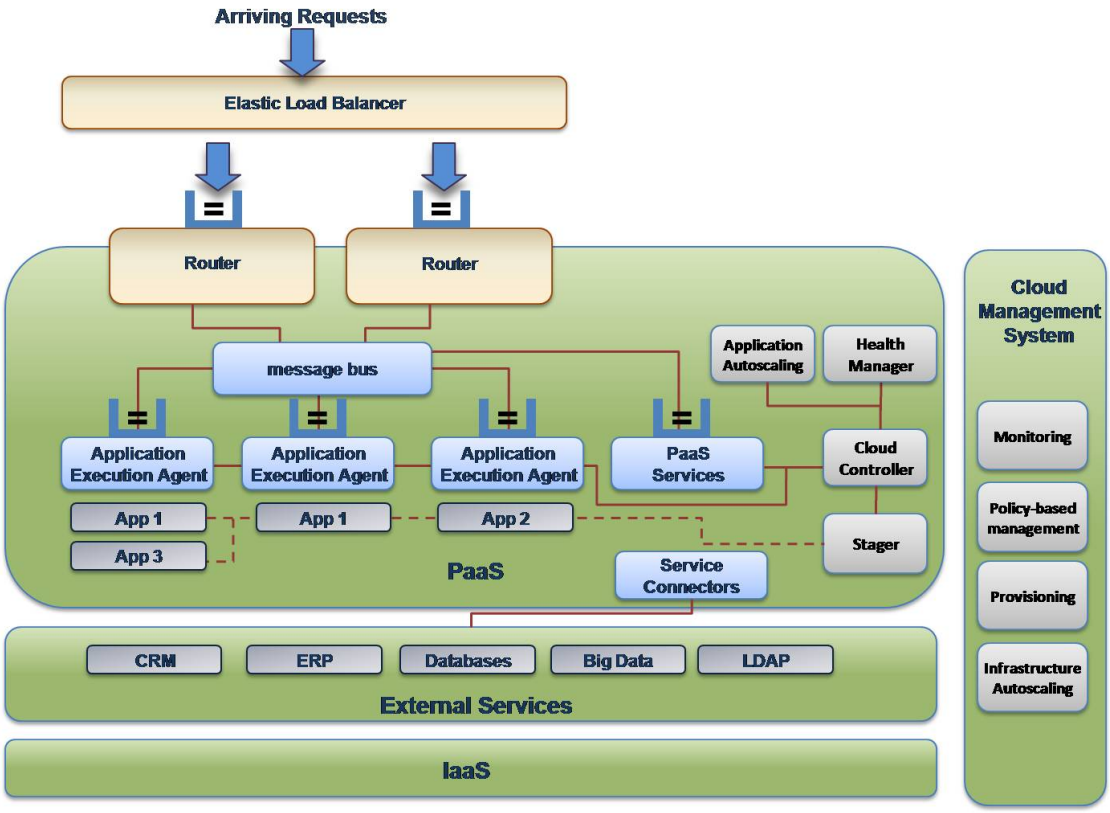

Figure 2: PaaS architectural components

via internal instrumentation of services.
This section introduces an unified terminology for common PaaS components as well as a generic, partially abstract, PaaS model. The model is used in later sections as a basis for tests related to PaaS configuration and application deployment.

4.1 PaaS components

The most important logical components of PaaS are presented in Fig. 2. An entity that serves as an entry point to the PaaS is called a **Router**. It is responsible for receiving requests from users (both direct PaaS users and PaaS applications' users) and forwarding them to an appropriate component – usually a **Cloud Controller** (CC) or an application running on an **Application Execution Agent** (AEA).

The Cloud Controller is a management entity that maintains metadata of all applications hosted in the PaaS and of services provided by the cloud for such applications. All management requests, such as starting an application, changing the number of its instances or restarting a service, are passed to the CC. When a user requests deployment of a new application or its newer version, the CC passes an application bundle to the **Stager** component. The Stager then updates the application's data sources and other configuration files. Subsequently, a re-packaged application is returned to the CC and then deployed by CC to the AEA. The AEA is a runtime environment (written e.g. in Java, Scala or Ruby) with a set of popular frameworks and libraries (e.g. Spring, Play, Lift). Multiple AEAs can reside in a single PaaS installation.
In addition to applications, PaaS hosts services such as databases, messaging queues or e-mail hosts that can be accessed by applications. The services are managed by CC, running on **External Services Nodes** and accessed by applications via unified interfaces called **Service Con-**

nectors.
One of the requirements of the PaaS is to be fault-tolerant. To achieve this, a **Health Manager** (HM) component constantly checks whether running services match predefined cloud configuration parameters. If some component is missing then, most probably, it has gone down due to some failure and therefore the HM will try to restart it. Finally, all management communication between PaaS components is performed asynchronously via a publish-subscribe distributed queuing messaging service depicted as the message bus in Fig. 2. The reliability of the messaging system is crucial for proper operation of the whole platform. When the number of requests is high, queues will begin to fill up with requests waiting to be processed.
Each PaaS component can be replicated to provide scalability. Multiple Routers, CCs and AEAs can coexist in the same environment. In most cases a load balancer is needed to distribute requests between Router instances. If possible, applications running on the PaaS should rely on built-in services; however when integration is inevitable, External Services can be accessed via Service Connectors.

As stated in the introduction, our methodology aims at testing elastic and dynamic clouds. However, most existing PaaS platforms do not provide auto-scaling features by themselves. To address this limitation, we assume that the PaaS is augmented by an additional external component called **Cloud Management System** (like Scalr, RightScale or Kaavo) depicted on the right-hand side in Fig. 2. The goal of this component is to add some advanced management features to the PaaS (such as auto-scaling, intelligent provisioning, policy-based management and automated monitoring).

4.2 IaaS influence on PaaS platform

In order for a particular PaaS to provide a multi-tenant, elastic and scalable environment, its services can utilize infrastructures exposed in accordance with the Infrastructure as a Service (IaaS) paradigm; e.g. Heroku PaaS is built on top of the Amazon Web Service technology. Such a deployment model supports adaptation to changing demands by adding or removing service instances. Note, however, that, in general, PaaS can run on top of a physical infrastructures as well.
Owing to the Cloud Management System described in the previous section, the scaling process can be automated and moreover the configuration of hybrid clouds becomes possible. Such deployment provides the means for evaluation of PaaS platforms deployed over public and private resources.
In order to perform reliable tests of the PaaS, the underlying infrastructure has to be stable and efficient. It would be relatively easy to ensure high performance parameters if PaaS services were deployed at the PaaS provider's data center. However, as mentioned above, we assume that PaaS runs on some third-party IaaS machines, so it is much harder to ensure stability of the testing environment. In particular, if it is desirable to compare two (or more) PaaS solutions, it would be important to provide the underlying IaaS cloud with the same configuration and allocation of resources.

4.3 Application testing aspects

In order to prepare appropriate test cases and execution scenarios we should first determine how the applications of interest use the underlying PaaS. Just as in the case of PaaS, the applications can be treated either as white- or black-box components. In the black-box approach tests focus mainly on Quality of Experience (QoE) measurements.
When no applications of interest can be readily identified, application-based assessment can rely on representatives of the most popular Internet application categories: social media (e.g. Facebook, LinkedIn), multimedia processing (e.g. Shazam, Siri, Iris), multimedia streaming (e.g. Grooveshark mobile), data storage (e.g. Dropbox) and computer games hosted in cloud environments.

Figure 3: Steps of the load testing procedure

5. PAAS PERFORMANCE TESTING PROCEDURE

The proposed generic load testing procedure consists of 7 steps, depicted in Fig. 3. These steps need to be executed for any test case. First, the scope and objective is defined. Next, suitable metrics and aspects of evaluation are selected. These two initial steps allow for appropriate selection of benchmarks and tools, and have substantial influence on the definition of use cases and usage patterns (steps 3 and 4 respectively). Usage patterns determine the types of load applied to the applications and services deployed in the PaaS. Key parameters (e.g. request volume) are specified at this point. Note that application parameters constitute only a subset of the test environment configuration, which also includes the configurations of the required services, underlying infrastructure, geographical locations of users and end device types, numbers of users simultaneously using a given service or application, time of day, etc. All this data is used in step 5, which concerns preparation of the test environment and test implementation. Following application tests (step 6), a throughout analysis can be performed (step 7) to determine the influence of various factors upon end-user experience.

As a very useful effect of the simple application testing procedure, key metrics that affect user experience can be identified. In the following subsections the presented procedure will be elaborated further, with particular focus on the specification of metrics, benchmarks and test cases. The test management system will also be briefly described.

5.1 Definition of aspects and metrics

As already elaborated in Section 2, the testing and evaluation methodology has to cover two fundamental classes of testing, i.e., quantitative and qualitative testing. As this paper focuses on performance testing, only qualitative testing will be discussed here.
Performance testing should be investigated in the context of performance-related metrics, test cases and scenarios, and may be categorized into three classes introduced in Section 3 as summarized in Table I. Each category of performance testing presented in Table I requires the load to be specified in more detail, including:

—**Application Workload Mix** of each system component - for example: 20% log-in, 40% database-intensive, 30% CPU-intensive, 10% I/O-intensive.

SUT components	Test category	Metrics
Load balancer, Router, Applic. Exec. Agent, Service Node, External Service, Application Instance	Stress testing	Latency, Maximum throughput
	Dynamicity testing	Time to stabilize: Response Time, Latency, Throughput; Change in: Response Time, Latency, Throughput; Failover time
	Stability testing	Critical frequency, Critical amplitude

Table I: The quantitative metrics related to SUT components

—**System Workload Mix** [Multiple workloads may be simulated in a single performance test] for example: 30% Workload A, 20% Workload B, 50% Workload C.

The metrics introduced in Table I are defined as follows: **Latency** - The time which elapses between initiation of request processing and receipt of the final result. This metric could be measured from the client's point of view, when application operations are performed, or from the PaaS point of view, when the time between a system request and its fulfillment is of particular interest. Latency depends on many factors, such as the complexity of task dispatching, the amount of time needed to start an execution engine, execution time, communication delays, etc.
Response time - When latency is measured from the client's point of view, it is often referred to as response time.
Throughput - The number of executed tasks performed per unit of time. In the context of PaaS evaluation, system tasks such as dispatching of requests to the execution engine or starting a new cloud node should be taken into account.
Maximum throughput - The maximum number of executed tasks that can be performed per unit of time. This metric is directly related to latency if a single architectural component of PaaS is considered. In such a case the maximum throughput may be calculated as follows: Maximum throughput = 1/Latency; In the generalized case, when N architectural element instances perform operations in parallel, the theoretical maximum throughput should be: Maximum throughput = N/Latency. Actual (measured) maximum throughput may not follow the formulas presented above due to scalability issues. Therefore, it is strongly recommended to measure the value of this metric experimentally.
Time to stabilize - This metric describes the dynamicity of PaaS by defining the time it takes the platform to transition from one steady state to another.
Response time change - The difference between response time in two different steady states of the PaaS.
Failover time - The time required to restore proper functionality and performance following a failure of an architectural element.
Latency change - The difference between latency time in two different steady states.
Throughput change - The difference between throughput value in two different steady states.
Critical frequency - the frequency of the stepped wave function which leads to instability (periodical changes of stable states) of the SUT.
Critical amplitude - the amplitude of the stepped wave function which leads to instability (periodical changes of stable states) of the SUT.

5.2 Definition of test cases

To the best of our knowledge, no dedicated test cases for PaaS platforms currently exist. As such, it is necessary to rely on application benchmarks. Because of the diversity of application

frameworks supported by PaaS platforms, a separate benchmark should be proposed for each of the frameworks. In addition, there are benchmarks that target validation of particular PaaS features. While selecting test cases for the scenarios described in section 6, we took into account multiple factors such as the ability to obtain objective results, simplicity of measurements and interpretation of their results, as well as the scope of the measured performance characteristics.

5.2.1 *Three-tier web applications – TC1.* The benchmark can use many example applications such as Spring Travel (Java Spring framework), Yet Another Blog Engine (Scala Play framework), Pet Clinic (Grails framework) [Davis 2014] or Node.js [Wilson 2013] benchmarks (part of the framework). These benchmarks rely on modularization of functionality by domain, reflecting the best practices of software development. They are multi-layered applications that can be scaled up and down in terms of web and database servers, thus enabling execution of sophisticated test cases differing in terms of the scalability and efficiency of session and persistence management services.

5.2.2 *Session persistence policy – TC2.* PaaS platforms support session affinity or sticky sessions for managing applications' users requests. The implementation of such behavior results in all requests from a given client being routed to the same application instance. However, if a particular AEA fails, PaaS might not persist or replicate session data. Session data that must be available after an application crashes or stops, or data that needs to be shared by all instances of an application, is stored by a PaaS internal service. The benchmark application should be written in two modes: (i) with session persistence built into the container, and (ii) using PaaS data services to maintain state. Besides High Availability (HA) aspects, comparison of performance of the application executed in these two modes might yield hints regarding selection of the appropriate mode for real-life applications.

5.2.3 *Memory-intensive application – TC3.* This case concerns a simple web application that, for each request, stores a specified amount of randomly generated data in its runtime memory over a specified period of time. The application works asynchronously. HTTP requests allocate memory, schedule its release at some point in the future and send HTTP responses to the clients. The goal of this benchmark is to test autoscaling as a response to memory shortages. When the cloud runs out of memory the Cloud Management System should automatically increase the amount of memory allocated for the underlying machines. The benchmark also provides for detection of autoscaling latency.

5.2.4 *CPU-intensive application – TC4.* This is a simple web application that, for each request, executes a simple synthetic computation in a loop over a specified period of time. The application works asynchronously. HTTP requests schedule execution of fake code and send scheduling confirmations to the clients. This benchmark enables testing autoscaling based on CPU load. It measures the time elapsed between a CPU usage increase and the assignment of additional CPUs to the underlying virtual machine by the Cloud Management System.

5.2.5 *Database-intensive application – TC5.* This is a simple web application bundled with a database schema and some sample test data. For each HTTP request it executes a complex query on that data. The application works asynchronously. HTTP requests do not wait until a query finishes. The query result is irrelevant, since the goal of querying is simply to generate fake load in order to test autoscaling based on service node load. Heavy queries will also use a lot of CPU and memory, triggering autoscaling actions. In addition, this test case can help verify how one query affects the response time of concurrent ones, i.e., provide information about performance isolation.

5.2.6 *Static content server application – TC6.* To be able to test routers/load balancers (layer-7 switches) and auto scaling at the AEA level, a simple web application that serves static content can be used. For example, it can respond to HTTP requests with some predefined text with-

PaaS component	Test category		
	Stress	Dynamicity	Stability
Load balancer	TC6	TC1	TC6
Router		TC1	
AEA	TC3, TC4	TC1,TC3, TC4	TC3, TC4
Health Manager			TC6
Service Node	TC5	TC1, TC5	TC5

Table II: Prefered assignment of test cases to specific PaaS components and aspect testing.

out interaction with databases or other external services where data is stored. Such tests could provide metrics illustrating the utilization of the infrastructure elements responsible for layer-7 switching. In respective test scenarios only the efficiency of interaction between cluster of AEAs and routers should be analysed. This test case covers most simple load balancing algorithms (round-robin, weighted round-robin), but for others which depend on server load reports TC4 may also be considered. In the case of load balancers that rely on response time, the test case can be slightly modified to use some simple "sleep" function parameterized with random values.

Suggestions regarding the assignment of test cases to particular SUT components in specific test categories are summarized in Table II. The presented test cases were used to assess the quantitative metrics of PaaS platforms as elaborated in the following sections.

5.3 Tools assisting in the test procedure

Effective performance evaluation of PaaS can be automated with a Test Management System (TMS) environment responsible for generation of synthetic load and acquisition of measurement results. The proposed architecture is depicted in Fig. 4. In the TMS, preparation of automated testing procedures consists of the following activities: (i) definition of testing scenarios based on particular test cases, (ii) deployment of test cases to the PaaS, (iii) scheduling of test scenario according to the number of Virtual Users (VUs), load distribution function, think time and others, (iv) execution of test scenario with metrics collection process to support visualization and comparison of results. The presented activities are defined with "plugins" that can be later reused in the form of libraries and implemented in accordance with TMS requirements. If required, the TMS also provides tools for the white-box methodology evaluation exposing advanced monitoring services for IaaS and PaaS layers to check the CPU, memory, network and storage utilization, and isolate potential bottlenecks. The monitoring services exploit well-known tools such as Ganglia[Ganglia 2014] and JMX/jmxtrans[JMX 2006]. The TMS also utilizes the JMeter[Erinle 2013] framework which provides services for distributed load generation and supports many protocols used by various types of test cases deployed over PaaS. The TMS can be deployed on a remote site to facilitate remote testing, or locally when local testing is performed.

6. SELECTED TEST SCENARIOS

The performance tests described in Section 5 can be customized in order to test specific PaaS elements. The proposed tests can be conducted either separately for particular elements of the PaaS, or in compound scenarios. By testing separate SUT components, analysts can identify performance bottlenecks. Compound test scenarios evaluate PaaS characteristics in an integrated way and illustrate end-user experience. This approach was specifically chosen for the experimental study presented in Section 7.

This section presents the test scenarios that were applied in the experimental study and discusses the expected (foreseen) results. As such, it facilitates interpretation of measured data presented in the following section. The test scenarios of interest are oriented towards providing answers to the following questions:

—is there any performance difference between operation of PaaS in public and private cloud

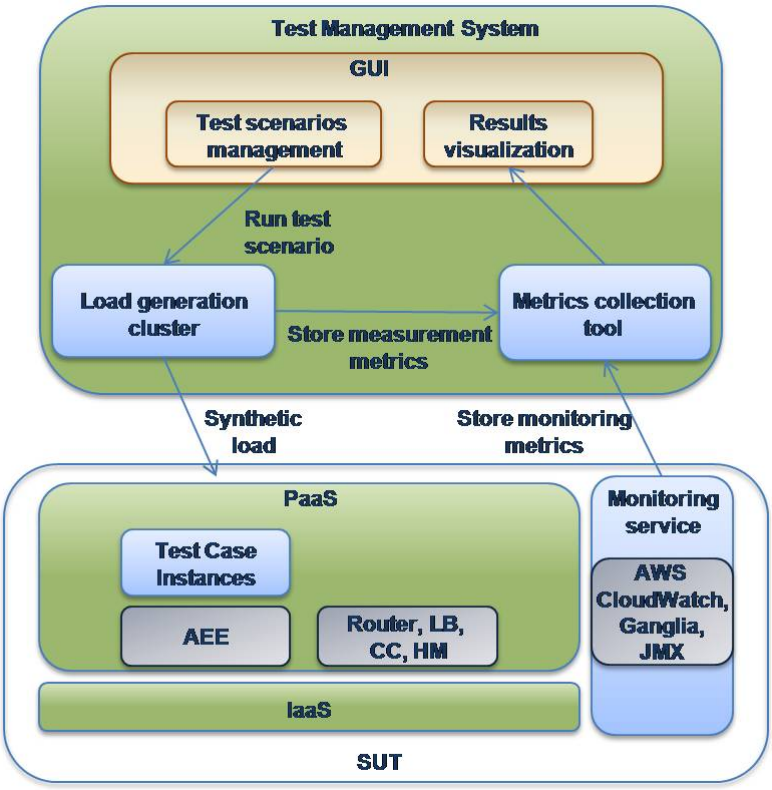

Figure 4: Architecture of the TMS.

deployments?

—what is the expected maximum throughput and response time?

—are tenants properly isolated, i.e., is the performance of one tenant affected by others?

—what is the influence of message bus-based communications on PaaS behavior?

—how does scaling a single SUT component out or up affect performance metrics?

—what is the delay in effecting scaling decisions?

The system response patterns represent specification of expected PaaS behavior under different load conditions. They can be used as conceptual reference points when real measured data is analysed (as discussed below). Basic knowledge of control theory[Kosinski et al. 2012] has been applied in this specification. The following subsections present the expected response patterns with regard to distinct categories of tests outlined in Section 6.

6.1 Stress testing - TS1

In stress testing two key factors are taken into account: time and load. In order to observe the reaction of a particular SUT component to increasing load, the suggested load pattern is as simple as possible – it is an affine function of time (starting from a specified value and increasing in a linear manner).

When observing changes in the SUT component behavior as a function of time given linear load growth, a critical load value emerges where response time increases rapidly. This indicates that the capacity of the SUT has been exhausted.

Figure 5 depicts the predicted response pattern for a constant number of application instances.

This test scenario could be performed for an increasing number of application instances. This would produce the data that describes the performance impact of newly deployed instances and

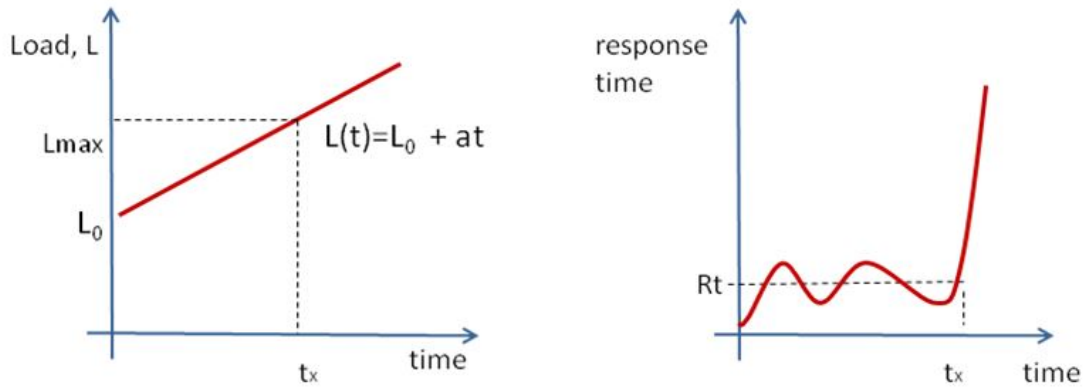

Figure 5: Affine load, used for stress testing, and the expected changes in response time.

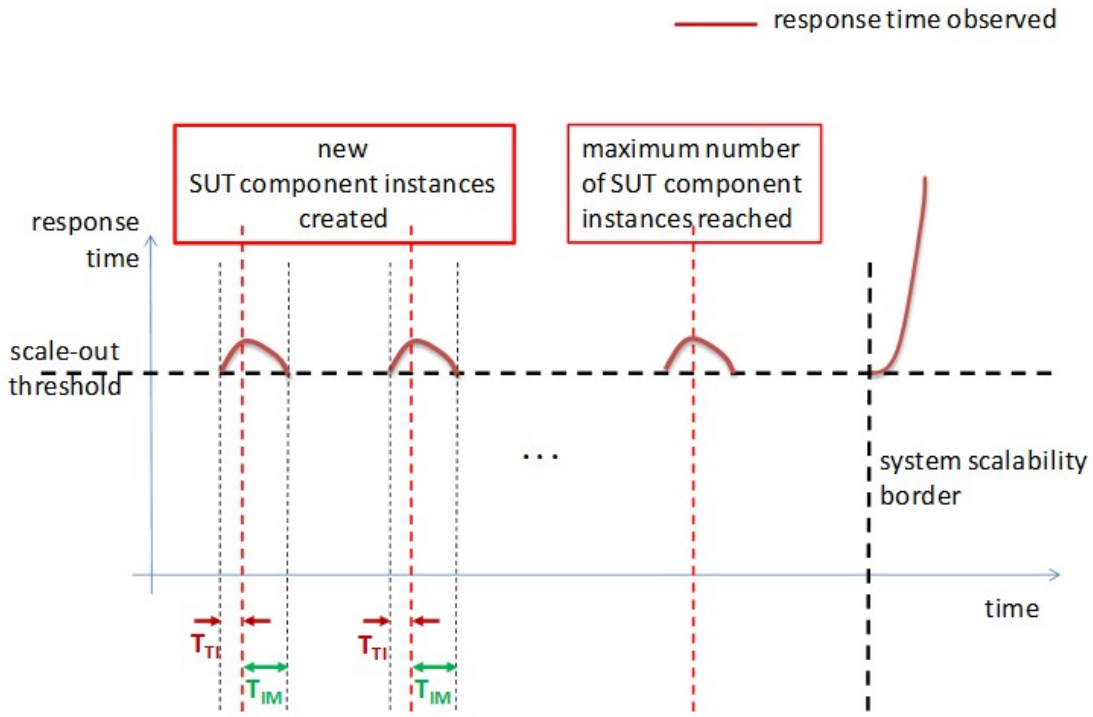

Figure 6: Expected response pattern for stress tests of an auto-scalable SUT. Affine load definition function assumed.

help determine to what extent it is possible to maintain the response time with autoscaling.

6.1.1 *Stress testing with autoscaling - TS2.* The autoscaling test scenario assumes that load increases linearly, much as in the TS1 scenario. When response time reaches the scale-out threshold, a new instance of the application is deployed. This increases the execution capacity of the SUT and causes a decrease in the measured response time. However, because no SUT is able to scale indefinitely, the limit of scalability is eventually reached and response time begins to grow rapidly. These phenomena are presented in Fig. 6.

During scalability tests, time-stamped response times and numbers of SUT component in-

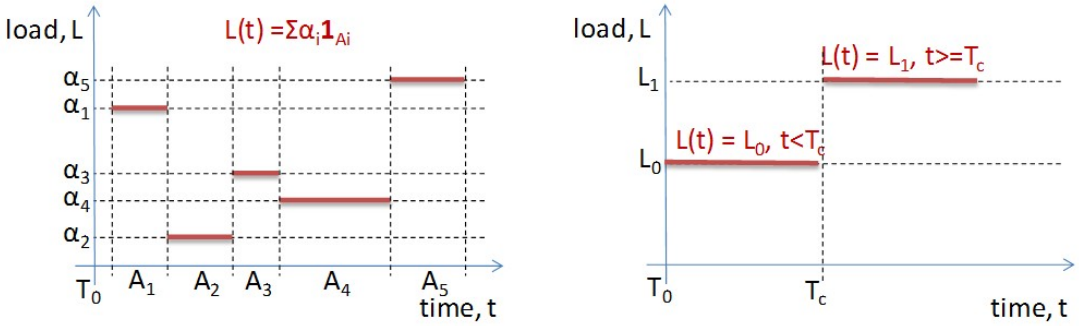

Figure 7: Load definitions based on a generalized stepped function and Heaviside-like function used for dynamicity testing.

stances are collected. Based on the obtained measurements, values of the following attributes can be calculated:

—T_{TI}: time elapsed between the response time exceeding a given threshold and instantiation of a new instance of the SUT component under consideration,

—T_{IM}: time elapsed between instantiation of a new instance of the SUT component and a drop in response time below the the acceptability threshold,

—T_{TR}=T_{TI}+T_{IM}: user-perceived system reaction time.

Additionally, the time (and load) at which the system can no longer maintain the specified response time can be identified as the time when the scale-out threshold is exceeded, having reaching the maximum number of SUT component instances. It is expected that once this capacity is exceeded, system stability may also degrade.

6.2 Dynamicity testing – TS3

Dynamicity testing is essentially about observing SUT reaction to sudden changes in load. Accordingly, the load pattern used in dynamicity testing can be described as a stepped function. However, because the effects of load changes are predictable only to a limited extent, it is advisable to use single-step functions (such as Heaviside's function) – especially at the beginning of the test.

The SUT response to changing load is delayed due to inertia, which is typical for every dynamic system. It takes a SUT-specific time to observe the response. Therefore, the duration of specific load conditions (the length of a single step in figure 7) must not be too short – otherwise the test would be assessing SUT's stability instead of dynamicity.

Figure 8 illustrates the predicted response for a system tested with a single-step load(time) function. For the sake of clarity, the response time metric and the autoscaling scenario used in the previous section were reapplied here.

In order to obtain useful data prior to conducting a dynamicity test, the analysts needs a proper estimate of the lower and higher loads. The lower load value (L_0) should result in a stable response time; such condition can be checked by performing a stress test with constant load, as described in section 6.1. The higher load value (L_1) should result in overcoming the scale-out threshold. Moreover, a consistent increase in response time should be observed prior to the scale-out operation.

Based on the measurements, the following attributes can be calculated:

—T_{CT}: time elapsed between a change in load conditions and crossing the scale-out threshold, T_{CT}, which depends on the difference between L_0 and L_1,

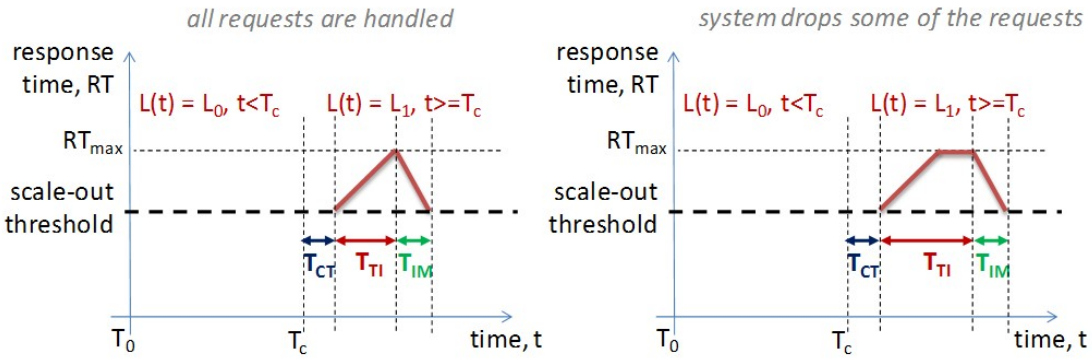

Figure 8: System response time changes in a dynamicity test.

—T_{TI}: time elapsed between crossing the threshold and instantiation of a new instance of the SUT component under consideration, T_{TI}, which does not depend on the load values,

—T_{IM}: time elapsed between instantiation of a new instance of the SUT component and the response time again dropping below the scale-out threshold,

—RT_{max}: maximum response time, i.e., the time it takes to handle a request when the maximum capacity of the request buffers is reached; RTmax can be measured when no scaling operation is permitted or when the operation takes longer than filling up the request buffers and the system starts dropping incoming requests. This vale indirectly illustrates the size of the queues at the performance bottleneck (application instance, router, load balancer etc.)

The values listed above can form a basis for estimating several characteristics of the SUT, such as:

—system usage/capacity ratio (based on L_0, L_1 and T_{CT}),

—the size of buffers needed to handle rapid changes in system load (based on T_{TI} and RT_{max}).

Moreover, one could determine whether the preconfigured auto-scaling operation was satisfactory based on the response time decrease ratio, which can be measured once the operation is completed.

Similar characteristics can be obtained when performing a scale-in test.

6.3 Stability testing – TS4

Oscillations of load in proximity to a decision threshold may reveal much about the stability of SUTs. Applying oscillating load (whether sine-, square- or sawtooth-like) can result in frequent changes in SUT configuration observed as a varying number of instances, response time, throughput or other metrics (see Fig. 9).

In order to set the lower and upper load boundaries properly it is important to know the decision thresholds (which should fall between the values) as otherwise the test would not yield useful results. The oscillations should be handled by the request buffers and the variations in response time should not exceed the acceptability thresholds.

The described, ideal SUT behavior patterns under test scenarios TS1-TS4 can only be observed if the following conditions are met:

—the performance of each SUT component instance is independent from the performance of others,

—load distribution between instances remains fair,

—there are no bottlenecks between the application clients and the tested SUT component.

The following section, which discusses experimental results, presents real-life characteristics that illustrate cases in which the pointed conditions are not always met.

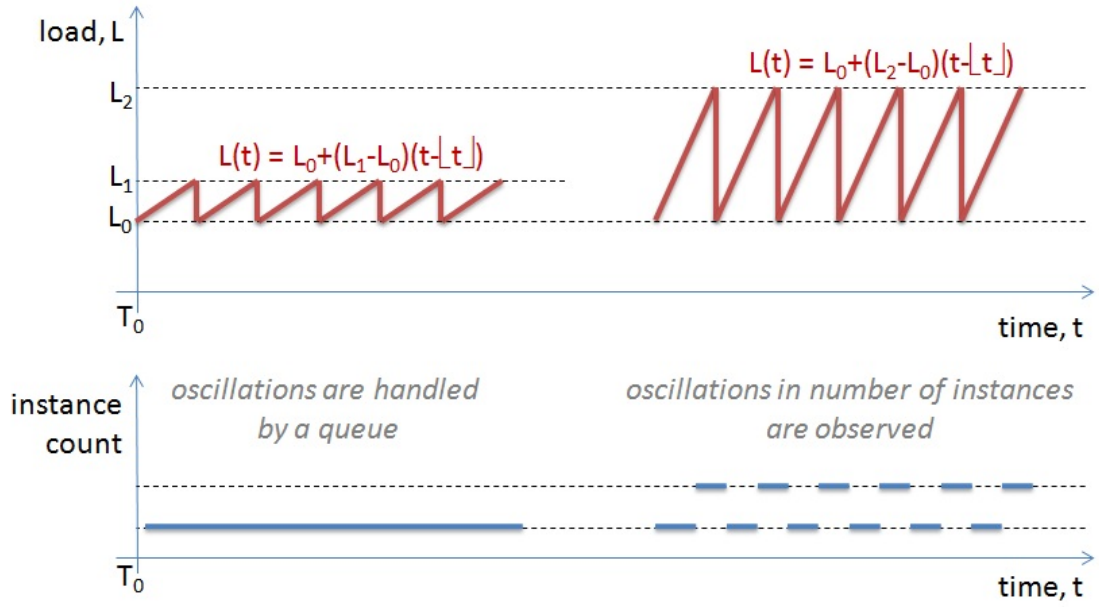

Figure 9: Expected instance count response patterns under oscillating load conditions.

7. DISCUSSION OF PRELIMINARY RESULTS

The methodology introduced in this paper was evaluated in a scenario that included a comparison of two cloud configurations – a public one and a private one. The clouds needed to be evaluated in the context of their applicability for hosting a large-scale application. The basic characteristics of the application were as follows:

—CPU power was the main resource requested,

—the application needed to scale out in cases of CPU usage exceeding a predefined threshold,

—the sessions between the application clients and servers were very short, typically consisting of just a few requests and responses.

In order to offer a reasonable SLA to application clients we needed to estimate the average response time (RT), maximum response time (RT_{max}) and the time it takes to scale out the application in specific clouds. A CPU-intensive application conforming to TC4 (see 5.2) was selected as the case study. The application performed a merge sort on a randomly generated array of integers; it took the size of array as the sole argument specified in its clients' requests.

Because conducting all the presented tests would be economically wasteful, we selected cases which yielded the most information about the cloud installations and configurations in the context of the specified test categories and application requirements. Based on the discussion presented in Section 5 (see Table II), we decided to perform:

—a maximum throughput test (TS1) using a CPU-intensive application (TC4),

—a dynamicity test (TS2) using the same application (TC4).

7.1 Testing environment

Test cases were based on Java/Spring frameworks deployed into Tomcat containers (AEE) and MySQL persistence services each running in an Ubuntu Linux instance. Evaluation was performed for booth private and public PaaS solutions integrated with the SCALR cloud management platform, facilitating single-access interfaces for both deployment options. The private PaaS was based on the CloudFoundry platform while the public one utilized Amazon BeanStalk services.

CloudFoundry was configured with a Router and AEE running on m1.medium (1 vCPU, 3.75 GB RAM) and m1.large (2 vCPUs, 7.5 GB RAM) profiles respectively. The AEE autoscaling policies mandated "five to twenty" VM instances. Amazon BeanStalk was assigned an m1.large profile with scaling policies set to "one to ten" VMs.
The TMS comprised four VMs with core services (GUI, metrics database, JMeter master) and twenty VMs working as JMeter slaves ensuring that the framework can handle a large number of virtual users. The TMS was deployed on a remote site for both private and public PaaS evaluation scenarios.

7.2 Maximum throughput measurement analysis

In this section we focus on analyzing the output of the stress test scenario (TS1) using a CPU-intensive application (TC4). Tests were conducted on two clouds (one of them public and one private) from the client's perspective. Conclusions were drawn without detailed knowledge of the configuration of the test environment. Instead, conclusions relied on the conceptual model of PaaS introduced in section 4 since its exact configuration – especially in the case of public PaaS – was not known to the authors.

7.2.1 *Public cloud assessment.* Tests conducted in the public cloud assumed linear load growth. The number of application instances was (respectively) 1,2,5 and 10. In the analysis we assumed that load balancers and request routers shared the load evenly and that the application queues were of equal length. Under such conditions application instance throughput was the only limiting factor. Accordingly, once maximum throughput is reached the queues should begin to fill up with requests and the response time should correlate to the square of the load throughput until the queues are filled up and the system begins dropping requests.

Analysis of throughput and response time for one application instance reveals the maximum theoretical throughput of a greater number (N) of instances (which should, in an ideal case, be N times greater). Following the experiment illustrated in the top left-hand corner of Fig. 10 one can observe that the maximum throughput of one instance can be estimated as a number between 38 and 40 requests per second (assuming that all requests before were processed successfully). Starting at roughly 38 requests per second the response time begins growing rapidly and then stabilizes at some level (RT_{max}).

The growth in the response time is due to the request queue filling up. Stabilization occurs when the queue is completely full and the application instance begins to drop incoming requests. It is worth noting that if instances are independent and there are no bottlenecks other than application instance performance, the values of RT_{max} observed for one instance should be the same for any number of instances. In the case of one instance, the observed RT_{max} was about 3.5 seconds.

As depicted in the top right-hand section of Fig. 10., for two instances running concurrently the maximum throughput estimation range is 76-95. From the presented data one could theorize that 76 requests per second (i.e. twice as much as in the single-instance scenario) is the maximum throughput for the tested configuration.

Note, however, that RT_{max} stabilizes at about 12 seconds, which is rather unexpected because if the throughput of application instances is the only limiting factor, RT_{max} should be independent of the number of instances. Note also that the time between the onset of the rapid response time growth and its stabilization should be longer than in the single-instance scenario (assuming the same linear load function). Results indicate, however, that the length of this interval is about 5 times greater, which is quite unexpected.

The case of five instances is rather difficult to analyze because it does not reveal any obvious RT_{max} value. Due to the variable hit(time) ratio it is also difficult to estimate the upper maximum throughput boundary. A broad estimate of maximum throughput is between 175 and 220 requests per second, with RTmax on the order of 6 seconds. The lack of a clear estimation for RTmax suggests that although some requests are dropped, the volume of application queues was not

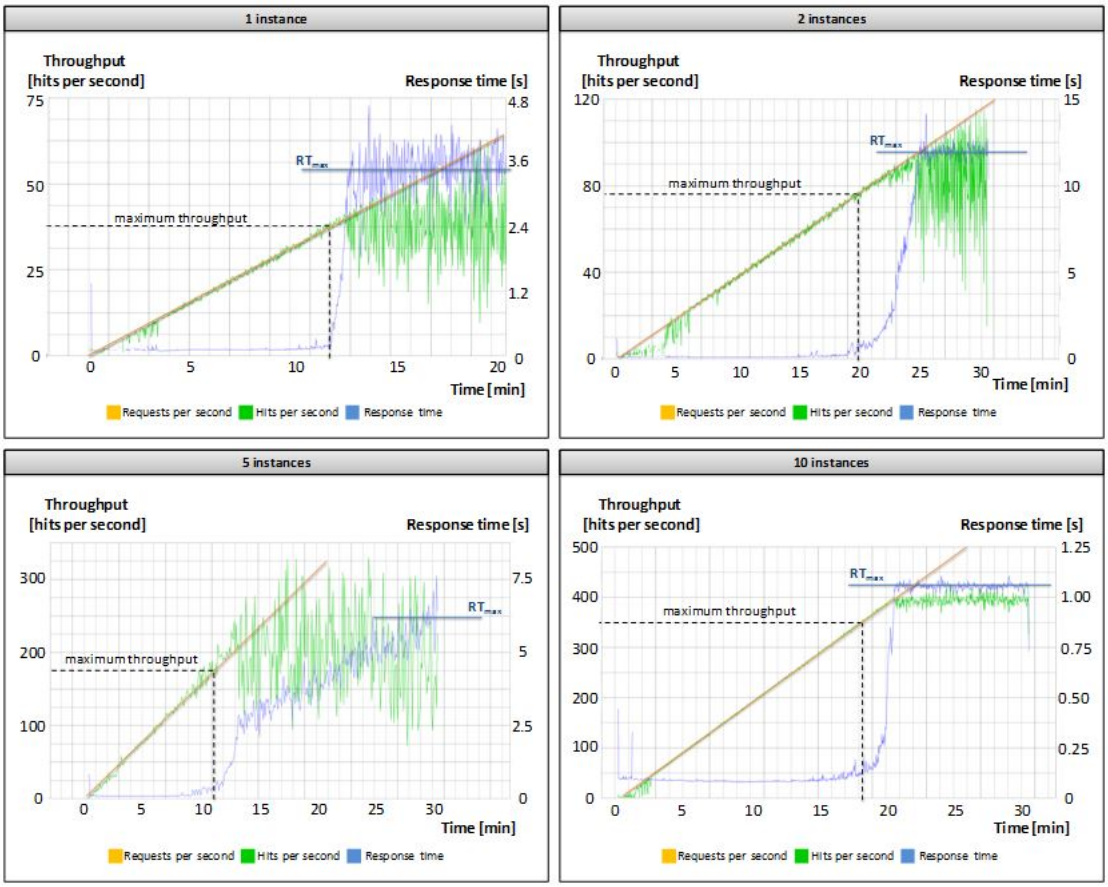

Figure 10: Public cloud CPU-intensive stress test results.

the only limiting factor. In other words, an entity other than the application instance (e.g. the request router, or the load balancer) also experienced trouble with handling requests.

The duration of rapid growth in response time is about two minutes, which is slightly more than twice the value obtained in the single-instance case.

In comparison with the five-instance case, 10 instances behaved in a very predictable way. The value of maximum throughput can be estimated as 350-400 requests per second range, i.e., 35-40 requests per second per instance. RT_{max} however, stabilized at about 1 second, which is much lower than in any other case. One reason for that is that the overflow occurred not within the application execution engines, but earlier on in the pipeline, e.g. in the request router or load balancer and therefore those requests which were not dropped did not need to spend much time in the application queue.

The duration of rapid growth in response time was about 3 minutes.

To sum up, even if the configuration details are unknown, the TS1 scenario proved to be a useful tool in assessing the public PaaS platform. The overall throughput scaled linearly with the number of application instances. For a linear load(time) function the system behaved as predicted (i.e. the response time initially remained stable and below the throughput limit, and then grew rapidly). Moreover, in the 5- and 10-instance cases, several effects were observed that suggested the existence of performance bottlenecks other than in the application queues. These phenomena could be explained only if a more detailed (white box) analysis is conducted.

Performance-wise, the public cloud scales almost perfectly. The differences between per-instance capacities are negligible. The variations in RTmax suggest, however, that the volume of

Application instance count	RT [s]	RT_{max} [s]	Estimated throughput (total)	Estimated throughput per instance	One instance throughput = 100%
1	0.2	3.5	38	38	100%
2	0.2	12	76	38	100%
5	0.2	6	175	35	92%
10	0.1	1	350	35	92%

Table III: Key measured characteristics of the public cloud.

PaaS element queues does not grow linearly with the number of instances.

7.2.2 *Private cloud assessment.* Similarly to the public cloud case, the tests conducted in a private cloud applied a linear load(time) function. The application instance counts, however, were 1, 10, 20 and 30 respectively.

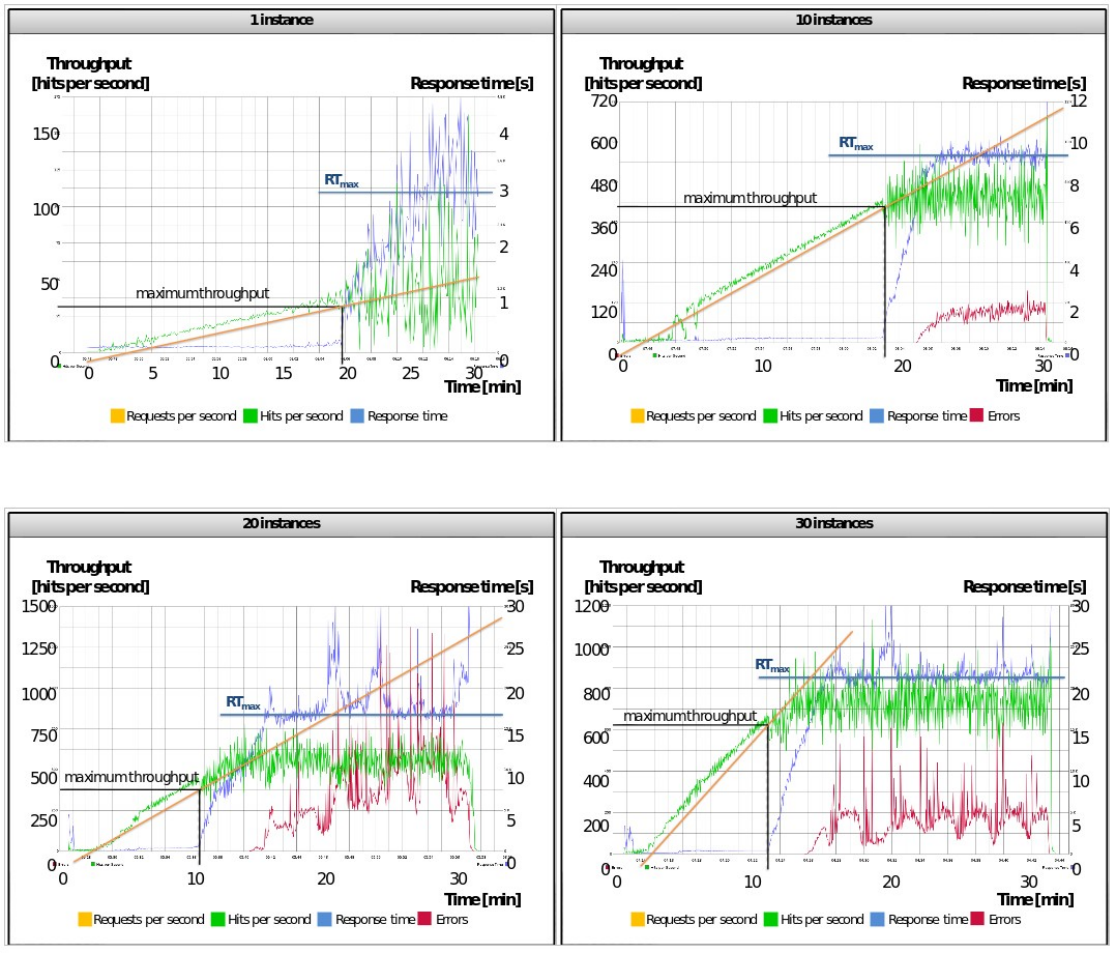

Figure 11: Private cloud CPU-intensive stress test results.

From among the two PaaS configurations the private one was characterized by better single-instance performance, but – on the other hand – the values presented in Table IV indicate that its configuration was clearly suboptimal. Per-instance performance in a 10-instance configuration was better than in the single-instance scenario. It therefore appears that in smaller deployments the instance remains underutilized. Additionally, the sudden drop in performance separating

Application instance count	RT [s]	RT_{max} [s]	Estimated throughput (total)	Estimated throughput per instance	One instance throughput = 100%
1	0.1	30	37	37	100%
10	0.25	9.5	425	42.5	115%
20	0.4	17	435	21.75	59%
30	0.45	22	660	22	60%

Table IV: Key measured characteristics of the private cloud

10-instance and 20-instance configurations suggests that the application instances are again underutilized and the performance bottleneck is on the level of router(s) or load balancer(s).

7.3 Dynamicity and stability measurements

In this section we focus on analyzing the output of the dynamicity test scenario (TS2) using a CPU-intensive application (TC4). Much like in the case of stress testing, public cloud tests required us to draw conclusions without detailed knowledge of the configuration of the test environment.

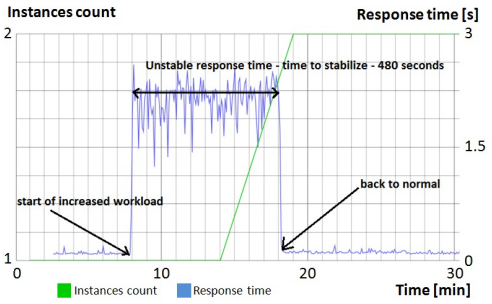

Figure 12: Dynamicity test results (public cloud).

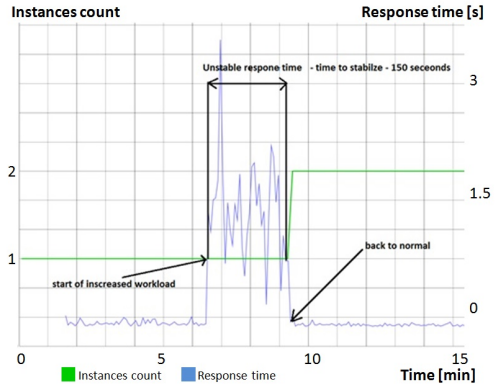

Figure 13: Dynamicity test results (private cloud).

In the test illustrated in Fig. 12 a Heaviside function based load pattern was applied to a system that was preconfigured to scale out if the CPU load exceeded 80% over a period of 5 minutes, while the load measurement was taken in one-minute intervals. By comparing the illustration with the model described in section 6.2 one can deduce that the request queue was filled very rapidly (perhaps the difference between the applied loads was too high) and then the system began to drop requests until a new VM was ready to accept them. After about 480 seconds the response time dropped back to the initial value. In the case of the private cloud configuration the respective time was only about 150 seconds (see Fig. 13). Therefore the conclusion was that the private setup scaled up faster and the instability period was significantly shorter. Similar tests conducted for a larger number of instances resulted in no instability in either setup – the request buffers were large enough to handle the increase in the number of requests without dropping any of them; only a temporary, gradual increase of response time was observed.

The increase in response time which resulted from the request queue gradually filling up was observed as a side effect in another test targeted at testing stability (Fig. 14).

Here, an oscillating load was applied with a constant average but high amplitude and – therefore – high variance of requests per second. By observing the results of the test one can deduce that:

—the response time grew due to the request queues filling up,

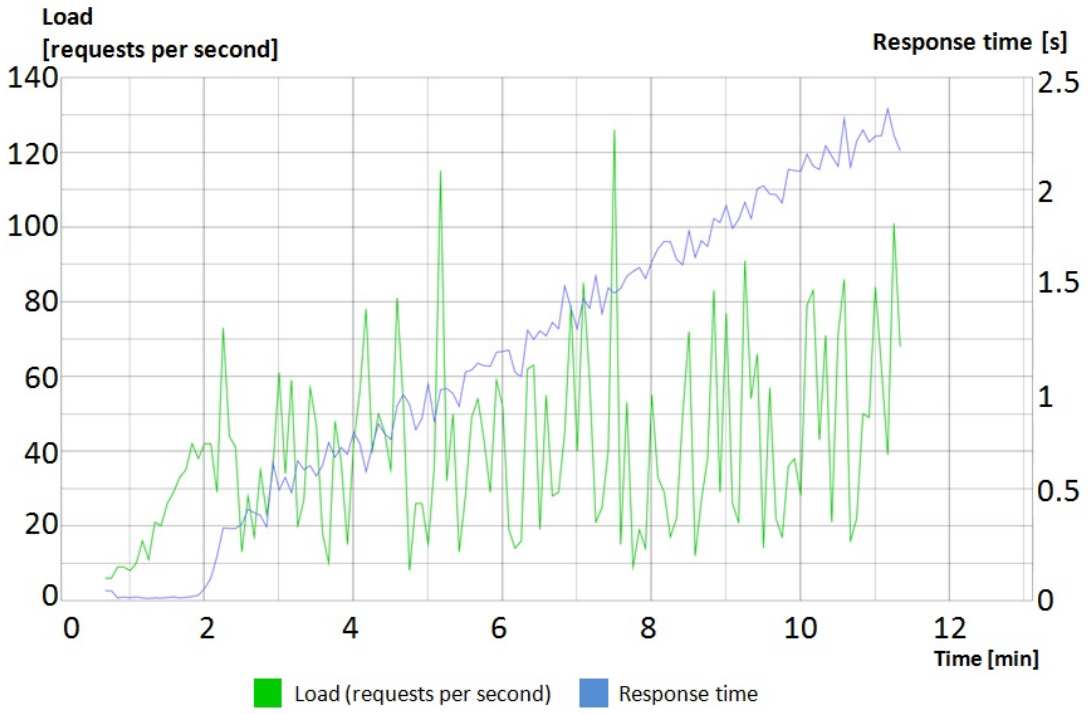

Figure 14: Gradual increase in response time resulting from request queuing.

—the variance in the response time was relatively low, so the system was well prepared to handle oscillations (i.e. the size of the buffers was enough to dampen the oscillation effects).

Fluctuations in response time were similar in both clouds and no significant differences between test setups could be observed in this experiment.

The stability tests revealed excessive sensitivity of the private cloud in terms of instantiating and deistantiating application instances in response to load changes (see Fig. 15). The load function was a sum of a linearly growing component and a randomly oscillating function (within a pre-configured range). Evidently, the scale-out and scale-in thresholds were too close to each other.

The PaaS dynamic behaviour is significantly influenced by:

—Organization of internal communication between PaaS components. The asynchronous nature of this communication and the length of message queues have substantial impact on response time and stability of the system under variable load conditions.

—Scalability of PaaS supported by IaaS works relatively well and the system scales out properly, although substantial delays are observed in the scaling-down process.

The presented study leads to the following conclusions:

—The theoretical model presented in Section 6 has been validated.

—Real data has been obtained which can be used as a reference point for qualitative analyses.

—The influence of internal PaaS communications on its performance has been assessed.

—A PaaS configuration methodology has been proposed.

8. CONCLUSIONS

The presented research confirms that the PaaS performance evaluation procedure should address many aspects such as stress, dynamicity and stability testing. The proposed testing scenarios

Figure 15: Oscillations in VM count close to decision thresholds (private cloud).

are universal and could be applied to different categories of services such as web applications and any memory-, CPU- or database-intensive components. PaaS performance is sensitive to a multitude of parameters referring to the generic PaaS model such as router efficiency, load balancer strategies, message queue length, internal communications etc. Their influence on qualitative performance aspects could be predicted based on theoretical analysis but real-life data has to be captured in experimental testing. Identification of performance limitations and system behavior under extreme load provide useful information for defining SLAs. Performing the same experiments in public and private clouds confirms similar behavior in both types of systems, although the results are not identical. This justifies the conclusion that the proposed performance evaluation procedure could be considered an obligatory step in validating deployment configurations. The PaaS performance testing procedure is quite complex and, in order to be successful, requires proper selection and assembly of tools. The proposed TMS system is composed of Ganglia, JMX and JMeter, each satisfying the stated requirements in that respect. These tools have to be supplemented with test scenario management modules and metrics visualization tools. Supporting rapid definition and automation of test scenarios in cloud environments remains a challenge and might require additional effort.

Acknowlegdements

The research presented in this paper was partially supported by Samsung Research and Development Poland, grant name "Cloud Validation and Verification Methodology", and the Polish Ministry of Science and Higher Education under AGH University of Science and Technology Grant 11.11.230.124 (statutory project).

REFERENCES

Stegun Abramowitz. 1964. *Handbook of Mathematical Functions With Formulas, Graphs, and Mathematical Tables*. National Bureau of Standards Applied Mathematics Series - 55. 1020 pages.

Nishant Agnihotri and Aman Kumar Sharma. 2014. Evaluating PaaS Scalability and Improving Performance Using

Scalability Improvement Systems. *IJRET: International Journal of Research in Engineering and Technology* 03, 03 (2014).

Abel Avram. 2010. Measuring and Comparing the Performance of 5 Cloud Platforms. (2010). http://www.infoq.com/news/2010/07/Benchmarking-5-Cloud-Platforms

Kees Blokland. 2013. Testing cloud services. How to test SaaS, PaaS and IaaS. (2013). http://www.polteq.com/wp-content/uploads/2013/11/2013_EuroSTAR-Ebook.pdf

B. Cohen. 2013. PaaS: New Opportunities for Cloud Application Development. *Computer* 46, 9 (2013), 97–100.

Adam L Davis. 2014. *Modern Java: Java 7 and Polyglot Programming on the JVM Paperback*. CreateSpace Independent Publishing Platform.

D.Jayasinghe, S.Malkowski, J.li, Q.Wang, Z.Wang, and Calton Pu. 2013. Variation In Performance and Scalability: An Experimental Study in IaaS Clouds using Multi - Tier Workloads. *IEEE Transactions on Services Computing* (2013), 1–14.

Bayo Erinle. 2013. *Performance Testing With JMeter 2.9*. Packt Publishing.

Ganglia. 2014. Ganglia Monitoring System. (2014). http://ganglia.sourceforge.net/

Mark Geene. 2012. 5 Steps To Selecting a Platform-as-a-Service (PaaS). (2012). http://www.cloud-elements.com/considerations-in-selecting-a-platform-as-a-service-paas/

Mark Geene. 2013. PaaS First; IaaS Second: Five Reasons to Select Your PaaS First. (2013). http://blog.appfog.com/paas-first-iaas-second-five-reasons-to-select-your-paas-first/

Neil J Gunther. 2007. *Guerrilla Capacity Planning*. Springer.

Shigeru Hosono, Jiafu He, Xuemei Liu, Lin Li, He Huang, and Shuichi Yoshino. 2011. Fast development platforms and methods for cloud applications. In *Services Computing Conference (APSCC), 2011 IEEE Asia-Pacific*. IEEE, 94–101.

JMX. 2006. *Java Management Extensions Specification - version 1.4*. http://docs.oracle.com/javase/7/docs/technotes/guides/jmx/JMX_1_4_specification.pdf

J. Kosinski, R. Szymacha, T. Szydlo, K. Zielinnski, J. Kosinska, and M. Jarzab. 2012. Adaptive SOA Solution Stack. *IEEE Transactions on Services Computing* 5, 2 (2012), 149–163.

Peter Mell and Tim Grance. 2009. The NIST definition of cloud computing. *National Institute of Standards and Technology* 53, 6 (2009), 50.

Atif Farid Mohammad and Hamid Mcheick. 2012. Cloud Service Testing: An Understanding. In *The 2nd International Conference on Ambient Systems, Network and Technologies*. 513–520.

Dan Orlando. 2011. Cloud computing service models, Part 2: Platform as a Service. (2011). http://www.ibm.com/developerworks/cloud/library/cl-cloudservices2paas/

Rajesh Ramchandani. 2012. Choosing and implementing suitable enterprise PaaS. (2012). http://jaxenter.com/choosing-and-implementing-suitable-enterprise-paas.1-43978.html

Alois Reitbauer, Andreas Grabner, and Michael Kopp. 2011. *Java Enterprise Performance*. Entwickler. Press.

Spec. 2012. Report on Cloud Computing to the OSG Steering Committee. (2012). https://www.spec.org/osgcloud/docs/osgcloudwgreport20120410.pdf

VMware. 2011. VMware vFabric PaaS Planning Service. (2011). http://www.vmware.com/files/pdf/services/vmware-vfabric-paas-planning-service.pdf

Jim R. Wilson. 2013. *Node.js the Right Way: Practical, Server-Side JavaScript That Scales*. Pragmatic Bookshelf.

Wenbo Zhang, Xiang Huang, Ningjiang Chen, Wei Wang, and Hua Zhong. 2012. PaaS-Oriented Performance Modeling for Cloud Computing. In *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*. 395–404.

Marcin Jarząb received his PhD in Computer Science, at the University of Science and Technology (AGH-UST) in Krakow, Poland, in 2011. He worked as a software consultant at ConsolSolutions and Software from 2000 - 2002, participating in many projects for Telco companies. He was an intern at Sun Labs in the latter half of 2003, investigating the application of the Multi-tasking Java Virtual Machine to the J2EE environment. Between 2003-2014 he worked as researcher at the AGH-UST participating in many commercial an scientific projects. Currently he is Cloud, Big Data architect and Principal Engineer at Samsung Research Poland. His research interests include the tuning and performance evaluation of distributed systems, design patterns, frameworks, and architectures of adaptive computing environments.

Krzysztof Zieliński is a full professor and head of the Department of Computer Science at AGH-UST. His interests focus on networking, mobile and wireless systems, distributed computing, and service-oriented distributed systems engineering. He is an author of over 200 papers in this area. He has been Project/Task Leader numerous EU-funded projects, like e.g.: PRO-ACCESS, 6WINIT, Ambient Networks. He served as an expert with Ministry of Science and Education. Now he is leading SOA oriented research performed by IT-SOA Consortium in Poland. In this area his research interest concerns: Adaptive SOA Solution Stack, Services Composition, Service Delivery Platforms and Methodology. His a member of IEEE, ACM and Polish Academy of Science Computer Science Chapter.

Slawomir Zieliński is an Assistant Professor at the Department of Computer Science at the AGH University of Science and Technology, Krakow, Poland. In 2009 he completed PhD on dynamic, semantics based deployment of overlay networks in peer-to-peer environments. His main areas of interest are cloud computing platforms, computer networking and development of distributed systems. In addition, he is working as an instructor at the Cisco Networking Academy.

Karol Grzegorczyk is a research assistant at the Department of Computer Science at the AGH University of Science and Technology, Krakow, Poland. He received the MSc degree in computer science from the Warsaw University of Technology, Poland, in 2006. His interests focus on the large scale distributed systems as well as machine learning algorithms and techniques.

Marek Piascik received MSc in Computer Science at the Warsaw University of Technology, he also holds a postgraduate diploma in cloud computing from Warsaw School of Economics. Between 2004-2013 he worked as software engineer and software architect in many commercial projects. Currently he held technical leadership positions in cloud and big data related projects at Samsung Research Poland. His professional interests include performance evaluation of cloud based services, application performance tuning and software design patterns.