Al-Said Ahmad and Andras *Journal of Cloud Computing: Advances, Systems and Applications* (2022) 11:1
https://doi.org/10.1186/s13677-021-00277-z

Journal of Cloud Computing: Advances, Systems and Applications

RESEARCH **Open Access**

Scalability resilience framework using application-level fault injection for cloud-based software services

Amro Al-Said Ahmad[1,2*] and Peter Andras[3]

Abstract

This paper presents an investigation into the effect of faults on the scalability resilience of cloud-based software services. The study introduces an experimental framework using the Application-Level Fault Injection (ALFI) to investigate how the faults at the application level affect the scalability resilience and behaviour of cloud-based software services. Previous studies on scalability analysis of cloud-based software services provide a baseline of the scalability behaviour of such services, allowing to conduct in-depth scalability investigation of these services. Experimental analysis on the EC2 cloud using a real-world cloud-based software service is used to demonstrate the framework, considering delay latency of software faults with two varied settings and two demand scenarios. The experimental approach is explained in detail. Here we simulate delay latency injection with two different times, 800 and 1600 ms, and compare the results with the baseline data. The results show that the proposed approach allows a fair assessment of the fault scenario's impact on the cloud software service's scalability resilience. We explain the use of the methodology to determine the impact of injected faults on the scalability behaviour and resilience of cloud-based software services.

Keywords: Resilience, Scalability, Software-as-a-service (SaaS), Application-level fault-injection

Introduction

As cloud-based software services have become more popular and dependable, evaluating the performance of such services is more critical than before. Previous research studies [1, 2] have focused on measuring the scalability performance of such services to collect the right measurements and set up specific metrics such as technical evaluation metrics and infrastructure-monitoring metrics. These metrics are important to set a baseline for the scalability performance behaviour of these services.

Performance and scalability assessment by using the fault injection technique allows evaluation of the impact of faults on aspects of cloud-based software services that pertain to the quality, such as performance, scalability, and security [3]. However, most studies in the area focused on injecting the faults on the Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) level [4, 5], or introducing a test environment system that injects faults into hardware devices or VMs levels [6].

Fault injection is a method to test the performance of software systems [7, 8]. Fault injection can take place at various times: at runtime, compile-time, or the loading time of external components [9]. Fault injection approaches have been used extensively to characterise the behaviour of systems under faults [4]. Furthermore, Fault injection has been utilized to analyse the dependability and reliability of cloud-based software systems [10–12].

Application-level fault injection (ALFI) is one of the most common techniques to study the application's resilience to faults [3]. It has been used to evaluate the application's vulnerability [3] based on its application

* Correspondence: asaid@philadelphia.edu.jo; a.al-said.ahmad@keele.ac.uk
[1]Faculty of Information Technology, Philadelphia University, Amman, Jordan
[2]School of Computing and Mathematics, Keele University, Newcastle-under-Lyme, UK
Full list of author information is available at the end of the article

responses. Moreover, the ALFI technique is used for testing the application's resilience to ascertain how applications tolerate random instance failures [13], which is a discipline of experimenting on software systems' ability to tolerate failures in unexpected conditions that have been referred to as "chaos engineering" [14]. In this work, we aim to present the use of scalability performance measurements and metrics to evaluate the resilience of cloud-based software services, and establish a corresponding framework for scalability performance measurements. For this we will be injecting the faults into the running cloud-based application by using fault injection tools to emulate potential problems at the application level to assess how the faults influence the scalability resilience and behaviour of the cloud-based software service.

The experimental evaluation of the results shows that the injected faults impacted the scalability behaviour. The established scalability resilience framework has highlighted the impact of those faults on the scalability and clarified how the ALFI significantly impacted the scalability resilience and behaviour of the targeted services. As a result, the scalability performance of the cloud services has significantly dropped in terms of quality (i.e. average response time) and volume (i.e. number of scaling instances). Incorporating the scalability measurements with application-level fault injection at runtime allows a clear assessment of the resiliency of the scalability behaviour of cloud software services and draws a fair indicator on how the faults will affect the scalability resilience and behaviour.

The structure for the remainder of the paper is as follows. First, Section 2 presents related works. Section 3, presents the scalability performance metrics and demand scenarios used in this paper. Section 3 discusses the proposed framework using the Application-Level Fault Injection for Scalability Resilience. Section 5 presents the results of an application example. This is proceeded by a discussion of the study in Section 6, including the implications, limitations, and importance of the work. The final section, Section 7, presents the conclusions and future directions.

Related works

Technical scalability metrics provide the baseline for more detailed investigations of cloud-based software services' scalability performance. Fault injection at the application level would help to evaluate the application's response to those artificial faults [3] over the quality aspects of cloud-based software services, such as performance, scalability, and security. Therefore, comparing the scalability performance of a cloud-based software service after a fault-injection attack with the performance analyses with normal workload will indicate the resiliency of that software service and how the scalability behaviour of such application will be impacted in such fault scenarios.

Technically oriented scalability measurements and metrics for cloud-based software service are limited. The work [15] provides an elasticity-driven metric that measures the sum of over and under-provisioned resources over the total length of time of service provision as a technical scalability metric. Even though the work [16] does not specify or formulate specific metrics of technical scalability, it does provide a technical measurements approach, which depends on throughput in the system with and without multiple VMs. A graphical model approach to evaluate the Software-as-a-Service (SaaS) performance and scalability is presented in [17]. The performance is evaluated from the system capacity perspective, which includes the system capacity and load as measurements for scalability. A case study using a sample of Java-based program hosted on EC2 has been reported.

While [18] provides a technical approach to scalability measurement in terms of throughput and CPU utilization, it does not present a well-defined metric. Instead, the work focuses on presenting performance variations through experimental analysis, using three public cloud platforms and two cloud applications, and another set of comparisons based on three private clouds that have been built using the three mainstream hypervisors. The work [19] focuses on building a model that allows measuring and comparing different delivery configurations in terms of capacity, elasticity, and cost. The work evaluated the proposed metrics using CloudStore application on Amazon EC2. On the other hand, they identified the scalability in terms of the number of simultaneously simulated users as a current limitation. While the work [20] proposed two scalability metrics, one based on the relationship between the services capacity and its use of resources; while the other is the cost scalability metric function that based on the services capacity and its cost, the work used the CloudStore hosted in EC2 with different configurations in order to demonstrate the proposed metrics.

In terms of fault injection, related survey studies [8, 21] show that most of the research is focused on measuring fault tolerance in cloud computing by using fault injection. The majority of the studies use the technique of injecting the fault on IaaS and PaaS levels [4, 5, 22, 23], testing the resilience specific types of cloud applications [24], or by introducing a test environment system that injects faults into hardware devices or VMs levels [6]. However, there have been some studies that address the fault injection technique on the cloud applications level. These studies describe either prototypes or the use of this technique to build fault detection and diagnosis models. Herscheid et al. [11] suggested a draft architecture for “fault injection as a service” within the OpenStack, the implementation of the service itself is still under further development. Ye et al. [12] proposed a fault injection framework for artificial intelligence applications in container-based clouds, in order to detect the fault behaviour and interference phenomenon, however, the work focuses on presenting fault detection models that can distinguish the injected faults. On the other hand, Zhang et al. [25] presented a novel fault injection framework for system calls level, however, the study aims to evaluate the reliability of applications in relation to system call invocation errors in production, and did not specify that the work is focused on Cloud-based applications.

Scalability performance metrics and demand scenarios

We follow the approach to measuring and quantifying the scalability of cloud-based software services and explaining the metrics based on the measurement approach, as presented in Al-Said Ahmad and Andras (2019) [2]. The measurement approach explains both scalability metrics, volume and quality scaling scalability of cloud-based software services.

Here, D and D′ are two service demand volumes. D′ is greater than D. While I and I′ are the corresponding number of software instances deployed to deliver the software services, t_r and t'_r are the corresponding average response times for the services. Assuming the same service demand scenarios, we consider a series of increasing demand levels, D_k. The corresponding performance indicators are t_k the service average response times and I_k the volume of software instances, and I^*_k is the corresponding ideal volume of software instances. We can calculate the corresponding ideal volume of software instances as the following:

$$I*_k = (D_k/D_0) \cdot I_0$$

The volume metrics (η_I) is defined as follows:

$$A^* = \sum\nolimits_{k=1,\dots,n} (D_k – D_k\text{-}1) \quad \cdot \quad (I^*_k + I^*_{k-1})/2 \quad (1)$$

$$A = \sum\nolimits_{k=1,\dots,n} (\mathrm{D_k - D_{k-1}}) \cdot \left(\mathrm{I_k} - 2 \cdot [\,\mathrm{I_k - I^*{}_k}]_+ + \mathrm{I_{k-1}} - 2 \cdot [\,\mathrm{I_{k+1} - I^*{}_{k+1}}]_+\right)/2 \tag{2}$$

$$\eta_I = A/A^* \tag{3}$$

Where $[x]_+$ represents the value of x if it is positive and 0 otherwise. This alteration of the calculation avoids the distortion of the metric caused by the potential overprovision of services.

The quality metric is defined (η_t) as follows:

$$B^* = \sum\nolimits_{k=1,\dots,n} (D_k - D_{k-1}) \cdot t_0 = (D_n - D_0) \cdot t_0 \tag{4}$$

$$B = \sum\nolimits_{k=1,\dots,n} (D_k - D_{k-1}) \cdot (t_k + t_{k-1})/2 \tag{5}$$

$$\eta_t = B^*/B \tag{6}$$

The performance measures consider; the number of scaling instances, and average response times for cloud-based software services scalability, to provide a practical measure of these features of such systems. This is important to support effective measurement and testing of the scalability of cloud-based software systems.

Figure 1 illustrates the calculation of the two scalability performance metrics (quality and volume) [2]. In Fig. 1a, A* is the area that represents the ideal expectation about the scalability behaviour, and A is the area that

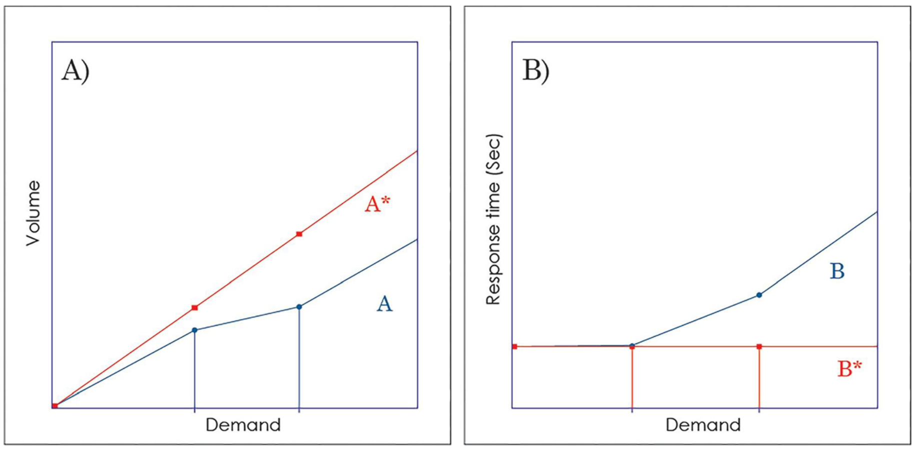

Fig. 1 The calculation of the scalability performance metrics: **a**) the volume scalability metric is η_I, which is the ratio between the areas A and A*; **b**) the quality scalability metric is η_t, which is the ratio between the areas B* and. The red lines indicate the ideal scaling behaviour and the blue curves show the actual scaling behaviour [2; page 5]

represents the corresponds to the actual volume scaling of the software services. In Fig. 1b, B* is the area under that represents the expected ideal behaviour, and B is the area that illustrates the actual quality scaling behaviour of the system (in terms of average response time). We chose to present the actual scaling behaviour as nonlinear curves to indicate that the actual scaling of the software services is likely to respond in a nonlinear manner to changing demand [2].

Two kinds of demand scenarios have been used in this paper. These demand scenarios follow the patterns recommended by Fehling et al. [26], which include static, periodic, once-in-a-lifetime, unpredictable, or continuously changing workload patterns. Any demand scenario or workload pattern must represent a real customer workload. So in this paper, we have adopted and followed those patterns and developed our versions of these recommended patterns. The first scenario is a steady increase followed by a steady decrease in the workload with a set level of the peak. This scenario follows the static workload pattern, which is suitable for private cloud-based applications of small and medium-sized companies; these systems are usually used internally by employees or a small user group [26]. The second scenario is a stepped increase and decreases, again with a set peak level of workload; with this scenario, we schedule to start with 10% of the total demand size, then increase 10% stepwise over time, followed by a 10% stepped decrease over time. These kinds of scenarios are suitable for cloud-based software services that follow growing and changing demand with peaks. This is important to show the scalability of cloud-based software services is adjusted automatically to the rate at which growing or changing happened [26]. These two demand scenarios are shown in Fig. 2.

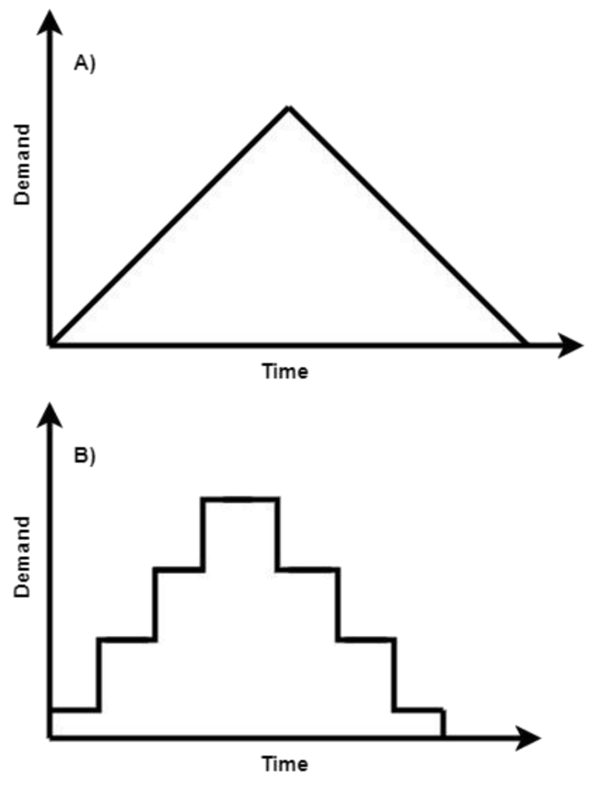

Fig. 2 Demand scenarios: **A**) steady rise and fall of demand; **B**) stepped rise and fall of demand

Application-level fault injection for scalability resilience

This paper aims to establish a framework for measuring the scalability resilience of cloud-based software services and to investigate the effect of runtime fault injection at the application level on the scalability behaviour of such services. An Auto-Scaling service is used to support the software services to deal with the sudden workload. In addition, a Load-Balancing service is used to determine the fault tolerance of software services by ensuring that the incoming application's traffic is distributed across multiple applications instances [27]. Previous studies [2] investigated the scalability performance of cloud-based software services, which set a baseline for the scalability behaviour of these services. In the study reported in this paper, the use of ALFI provides data to compare the scalability performance with the baseline performance following the scalability metrics discussed in [2] and explained in section 3.

In general, the aim here is not to crash the application at runtime. Our methodology is focused on measuring and evaluating the effect of the injected faults on the cloud-based software services' scalability over a sustained period. We collect the measurements that have been defined in Section 3, the number of scaling instances and average response times, to calculate the volume and quality scalability metrics. This will provide fair comparisons of the calculated average number of instances and average response time under normal operation and the behaviour of the two measurements during fault injection. This will provide useful behaviour benchmarking about the scalability performance that can assess the impact of faults in the delivery of the cloud-based software service from a scalability resilience perspective. Figure 3 illustrates the general set-up of the experimental framework approach.

This framework incorporates four main components: workload generator, software fault, scalability measures, and the system under test and its environment. A workload generator (such as JMeter or/and Redline13) is used to simulate a realistic workload demand scenario that reflects the real usage of services. A set of software faults should represent a repeatable and generally accepted set of faults (such as adding latency/bandwidth, HTTP traffic, database traffic, or terminating requests). The

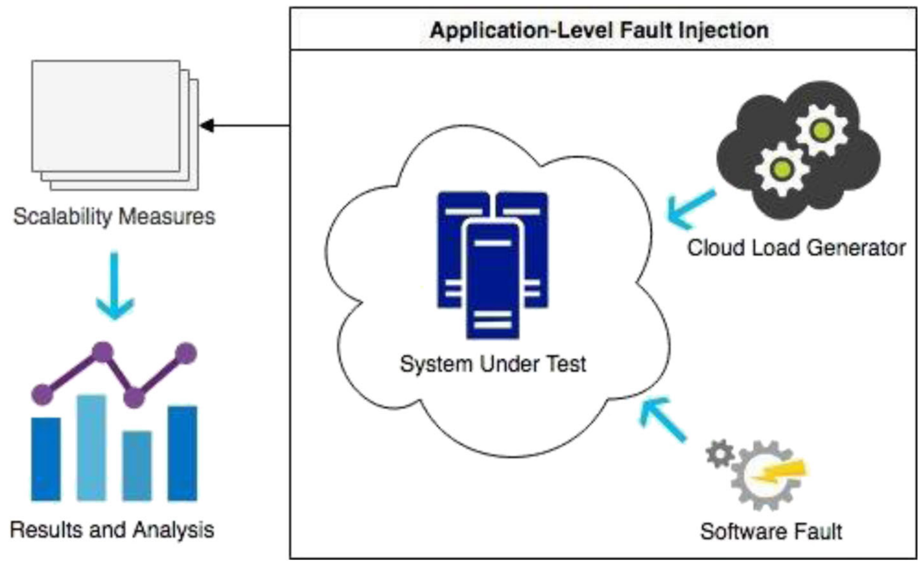

Fig. 3 Scalability Resilience framework using application-level fault injection

software fault is defined as *"An error is that part of the system state which is liable to lead to subsequent failure: an error affecting the service is an indication that a failure occurs or has occurred. The adjudged or hypothesised cause of an error is a fault."* [28]. Scalability measures are the indicators that are used to quantify the scalability of cloud-based software services. Finally, the system under test and its environment include connecting both Auto-Scaling and Load-Balancing services to ensure the scaling provision of services.

Incorporating scalability measurements with application-level fault injection (ALFI) to measure the resilience of cloud-based software services is very important. This framework can provide a useful behaviour benchmarking in relation to the scalability performance that can be used to assess the impact of the injected faults on the delivery and the resilience of such service from a scalability perspective.

Application example and results

We follow a two-stage preparation process in order to use the ALFI approach to validate the framework proposed in section 4 for measuring the scalability resilience of cloud-based software services. The first stage is preparing the workload scenario, scalability measures, and the system under test and its environment; the second stage is preparing the set of the software fault(s) injected in parallel with the workload on the system under test. Following the preparatory stages, we execute the experiments and measure the scalability performance.

System set-up stage

An Amazon EC2 instance was configured in order to host the OrangeHRM (https://www.orangehrm.com/) service through the AWS management console. OrangeHRM is an open-source application built using both PHP and MySQL. It has been optimized to fit cloud environment use, and it has influenced its architecture by offering a scalable human resource (HR) solution [29]. OrangeHRM is considered the most popular HR software globally, with more than 4 million current users across the globe (*OrangeHRM.com*). In addition, the application is based on REST-caching architecture, which is highly adopted and used by cloud applications and service providers. The REST architecture allows improving the performance and scalability of cloud software services by caching the data and code [30]. That will help improve the response time of such services by reducing the amount of time required to execute the HTTP requests [31].

The application instance was connected with auto-scaling and Load-Balancing services. In addition, the CloudWatch service was attached to monitor the scaling parameters. Table 1 shows the parameters of the instance and the Auto-Scaling policies that were used for the experiments. The auto-Scaling policies adopted in this paper are the default policies used in AWS when

Table 1 EC2 instance parameters and Auto-Scaling policies

Instance (VM) Parameters			
Instance type: t2.micro			
vCPUs	RAM (GB)	CPU Credits/hour	Storage (GB)
1	1.0	6	10
Auto-Scaling Policies			
Add Instance		80% >= CPU Utilization < + infinity	
Remove Instance		30% <= CPU Utilization > – infinity	

creating EC2 virtual machines. Although, in the previous study [2], a comparison has drawn between two options of auto-scaling policies (i.e., default AWS and custom auto-scaling policies). The work concluded that efficiency is increased when used the default auto-scaling policies offer by AWS [2]. Therefore, we relied on using the default Auto-Scaling policies offered by AWS. Some parameters have been considered to connect the auto-scaling to the software instance on EC2:

- The capacity of the auto-scaling group is used to determine the maximum number of scaling instances;
- Launch configuration: is a configuration template that uses by an auto-scaling group to lunch software instances at runtime (include the ID of Amazon Machine Image (AMI); the instance type; a security key pair; security group(s); and a block device mapping);
- Scaling policies are instructions for making the scale up and down in response to a workload. In this paper, we relied on CPU Utilization; and
- Attach the Load-balancers to the Auto Scaling group. This will help deal with HTTP and HTTPS traffic and automatically distribute incoming application traffic across multiple targets, such as EC2 instances [27].

We use the Apache JMeter script to simulate the demand scenarios illustrated in Fig. 2. Furthermore, to ensure the repeatability of the demand scenarios, RedLine13 services were used. This allows us to deploy the test scripts easily using our AWS account and repeat the tests without resetting the test parameters. In addition, this allows systematic extraction of the data. We used both Redline13 and AWS' CloudWatch services to collect The scalability measurements data.

Here we report the behaviour of the OrangeHRM in response to the HTTP request. The JMeter allows targeting the system-under-test (SUT) with a basic HTTP/HTTPS request, parsing HTML web pages for images and other embedded resources in the application, including applets, scripts, etc., and sends HTTP retrieval requests [32]. The service requests consisted of HTTP requests to the main page of the application by gaining login access using the following steps from Apache JMeter script:

- Path = /
- Method = GET
- Parameters = username, password, and a login button

Fault preparation stage

To simulate the injected faults, we used Charles version 4.5.4 (https://www.charlesproxy.com/), which is an HTTP proxy, an HTTP monitor; a reverse proxy; and a web traffic simulator, to simulate application delay latency (in milliseconds [ms]). The latency delay simulates the latency experienced on slower connections, which is the delay between making an HTTP request(s) from the application side and receiving the request at the cloud server-side. In the experiments reported in this paper, the delay latency times were varied: 800 ms and 1600 ms. For our purposes, it was sufficient to simulate the latency delay using Charles; also, we found this HTTP proxy easy to use, free and available. However, it should be noted that there are some other HTTP proxy alternatives, including James, Fiddler, TinyProxy, and mitmproxy, etc.

Here we simulate a delayed latency that is the delay in time between the request being made and received at the other end. This is called just before executing the HTTP request against the targeted cloud-based software service. This provides an insight into how calling applications behave when their dependency goes slow, as requests accumulated causes congestion at the requests queue. Although this type of fault focuses on affecting the quality measurement (i.e. response time), delay latency can affect the system capacity (i.e., volume measurement). For instance, the targeted application requires computation to process the requests and to receive their response from the cloud side. This process causes an increase in the number of HTTP requests being initiated to respond to the service responses, increasing the possibility of requesting more service instances (volume) to handle the number of HTTP requests targeting the software services. In addition, this type of fault can cause a termination of service delivery processes, requiring the re-issue of service requests (i.e., connection timed out).

In this paper, the delay latency may be set to any random milliseconds of time. Here the delay simulates the latency experienced on slower connections [charlesproxy.com]. The latency delay is the delay between making any request and the request received on the server-side. Therefore, each request is subjected to the same delay, i.e., if we assign 100 requests to hit the system in 10 s with an 800 milliseconds latency delay, we expect that each request will be delayed for 800 ms before reaching the server-side. Due to the latency delay (i.e., 800 ms or 1600 ms), there are some re-issues of the same request caused by the delay with the answer from the server side. These additional re-issues of the requests cause the clogging of the system. Given that there is a variation in the response times, there will be a variation in the number of re-issues of the same requests and a variation in the cancellations of the re-issues of requests, when the service delivery arrives.

Experimental process

An example of the experimental demand pattern at runtime is illustrated in Fig. 4, and these patterns were captured after applying the two-stage ALFI experimental approach. While Fig. 5 represents an example of the experimental demand pattern at runtime for the baseline experiments (without fault injection). We note that there is a delay in terms of starting the user running in real-time. This is due to the delay between the request being made and received at the cloud-based software service side. In this paper's detailed set of experiments, each fault injection experiment was conducted ten times (× 10), and the baseline experiments were performed ten times. Each demand scenario varies the volume of demand, and we used experiments with four demand sizes: 100, 200, 400, and 800 service requests. A total of 160 (fault injection) experimental were conducted and 80 basline experiments (without fault), the averages and standard deviations of the simultaneously scaled number of instances and average response times over ten experimental runs have been calculated. We note over the ten runs that the standard deviations were minimal concerning the averages.

To consider the collected results of any performance indicator as benchmark data, the value of one test should be obtained and compared with previous tests. Therefore, to ensure that our test results are statistically significant, all tests have been repeated ten times. Thus, Table 2 shows the details and duration of the experiments conducted in this paper. Each user does twenty iterations, i.e., if we assigned 200 virtual users to hit the system in an x time, we expect that all 200 will do ten iterations, which means 4000 times. Each experiment took 1.15 h (in average) to complete without considering the management time for uploading the application into our cloud platform, setting up both auto-scaling and load-balancing settings, and setting up the fault simulator for each experiment.

The measured scalability results

This section will present the scalability measurements collected following the scalability resilience technical measurements framework discussed in section 4. The baseline benchmark data was collected from the experiments without fault injection, following the first stage of the approach. We note that some of the 800 service requests for 800 ms and 1600 ms delay latency experiments crashed due to "connection timed out." Table 3 shows the successful and failed experiments. Failed experiments are defined as those in which all virtual users in one test do not complete the test successfully in the allocated time for each test, or the test ended in the case of "connection timed out," i.e., if we assigned 400 virtual users to hit the system in an x period of time, we expect that all 400 will successfully finish. Otherwise, it is considered a failed experiment.

The average number of software instances for each of the four demand levels (100, 200, 400, and 800) is shown in Fig. 6. The figure illiterates the comparison between the baseline, 800 ms, and 1600 ms delay latency experiments for both demand scenarios. The average response times for the four demand levels are shown in Fig. 7, which illustrates the comparison between the baseline, 800 ms, and 1600 ms delay latency experiments for both demand scenarios.

It is noted that the average number of instances for the 800 ms experiments caused a similar scaling behaviour to the baseline, while in the case of the 1600 ms experiments, the behaviour changed by increasing the number of provisioned instances at the 400 service requests. Thus, in terms of quality, there was not a big variation in average response times in both cases (800 and 1600 ms). However, in the case of 800 ms, the scaling started increasing significantly from the demand size of 200. Then once the demand size reached 400, the average response time stabilized around the same pattern as the baseline. In contrast, the response time values for the 1600 ms experiment, shown in Fig. 7, increased gradually with bigger variations.

This investigation of scalability is designed to determine the impact of using other ways to study the performance of cloud software services, such as using the

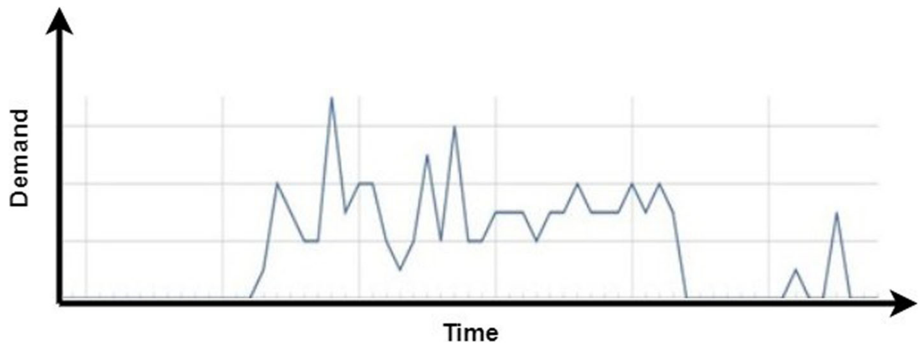

Fig. 4 Typical experimental demand patterns: OrangeHRM/EC2 – series of stepwise increases and decreases of demand – ALFI

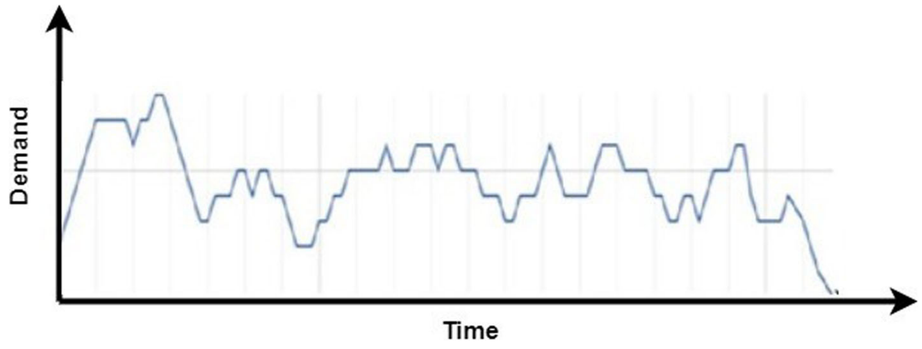

Fig. 5 Typical experimental demand patterns: OrangeHRM/EC2 – series of stepwise increases and decreases of demand – Baseline (2, page 8)

fault-injection technique. Figure 8 shows the average number of instances of the 800 and 1600 ms latency injection experiments compared with the ideal scaling behaviour in relation to both technical scalability metrics: volume (η_I) and quality (η_t) calculation, while Fig. 9 illustrates the ideal and average response times of the 800 and 1600 ms latency injection experiments for both demand scenarios. Here, we compare the ideal scaling of the baseline experiments with the actual scaling behaviour of the latency injection experiments.

We note that both 800 and 1600 ms fault injection experiments have drawn similar scalability behaviour in terms of volume scalability. However, we have noted a big variation in response times for both scenarios in terms of quality. This is not surprising, given our intervention (fault injection) is adding time delays, which will

Table 2 Experiments Duration

Scenario	Experiment type	No. of running users	Iteration (in each test)	Duration (sec)	Total time (×10)
Steady rise and fall (Demand Scenario 1)	**Baseline (without fault injection)**	100	20	100	20,000 s
		200	20	200	40,000 s
		400	20	400	80,000 s
		800	20	800	160,000 s
	800 ms delay latency	100	20	100	20,000 s
		200	20	200	40,000 s
		400	20	400	80,000 s
		800	20	800	160,000 s
	1600 ms delay latency	100	20	100	20,000 s
		200	20	200	40,000 s
		400	20	400	80,000 s
		800	20	800	160,000 s
Stepwise increase and decrease (Demand Scenario 2)	**Baseline (without fault injection)**	100	20	10	2000 s
		200	20	20	4000 s
		400	20	40	8000 s
		800	20	80	16,000 s
	800 ms delay latency	100	20	10	2000 s
		200	20	20	4000 s
		400	20	40	8000 s
		800	20	80	16,000 s
	1600 ms delay latency	100	20	10	2000 s
		200	20	20	4000 s
		400	20	40	8000 s
		800	20	80	16,000 s

Table 3 The successful/failed experiments

Scenario	Experiment type	100	200	400	800
Steady rise and fall (Demand Scenario 1)	**Baseline (without fault injection)**	Successful	Successful	Successful	Successful
	800 ms delay latency	Successful	Successful	Successful	80% Successful (8 out of 10)
	1600 ms delay latency	Successful	Successful	Successful	70% Successful (7 out of 10)
Stepwise increase and decrease (Demand Scenario 2)	**Baseline (without fault injection)**	Successful	Successful	Successful	Successful
	800 ms delay latency	Successful	Successful	Successful	60% Successful (6 out of 10)
	1600 ms delay latency	Successful	Successful	Successful	40% Successful (4 out of 10)

directly affect the average response time of the service. For example, in Fig. 8A and B for the demand size 100, over-provision cases have been accrued, while in Fig. 8A, which illustrates the average number of instances of the steady rise and fall of demand (the simpler demand scenario), both 800 and 1600 ms draw a similar pattern. However, in the context of the second scenario (see Fig. 8B) - which is likely to be a more realistic scenario for many software services – response time (quality) scaling is changing when we reach 400 demand size for the 1600 ms fault injection experiments.

The values for the scalability metrics η_I and η_t for the both baseline and the two fault injection set of experiments that were conducted are shown in Table 4. The calculated metrics show that the fault injection experiments display over-provisioning behaviour in volume scaling, with a notably decreased volume performance in the 1600 ms experiment for both scenarios. This is because the volume metric values for the fault injection scaling behaviour is based on the metric that considers the over-provision (see equation number 2). This is because part of the volume results are equivalent to over-provision according to our definition of this (i.e. see Fig. 8A and B for demand size 100).

In terms of quality scalability, the system scales much better in the baseline context than the fault injection experiments. It was noted that as a result of the variations in response times for the 1600 ms experiments, the quality metric (η_I) value dropped by 0.5609, a percentage decrease of 62% (first scenario), and 0.4121 with a 79% percentage deceased (second scenario). It was also noted that by using 1600 ms latency injection, the volume (η_t) decreased as expected; however, the quality dropped significantly. If the decrease in quality and volume scaling is taken into account, this shows that the overall performance of the scalability behaviour and resilience has dropped.

It should be noted that the latency faults cause a negative impact in terms of quality. In contrast, volume decreases between 36% and 49% in relation to the baseline for the 1600 ms experiments for the first and second scenarios, respectively. Furthermore, the quality indicator shows a significant drop in the performance of the services between 62% and 79% in relation to the baseline for the 1600 ms experiments for the first and second scenarios, respectively.

Based on the above percentage drops in terms of scalability metrics values, this helps to provide a clear assessment of the scalability's resilience. By considering the baseline as 1, and see how much the values drop after applying ALFI incorporated with scalability testing. This will help to define a scalability performance resilience measure. This established a framework for using ALFI to measure the resiliency of scalability performance

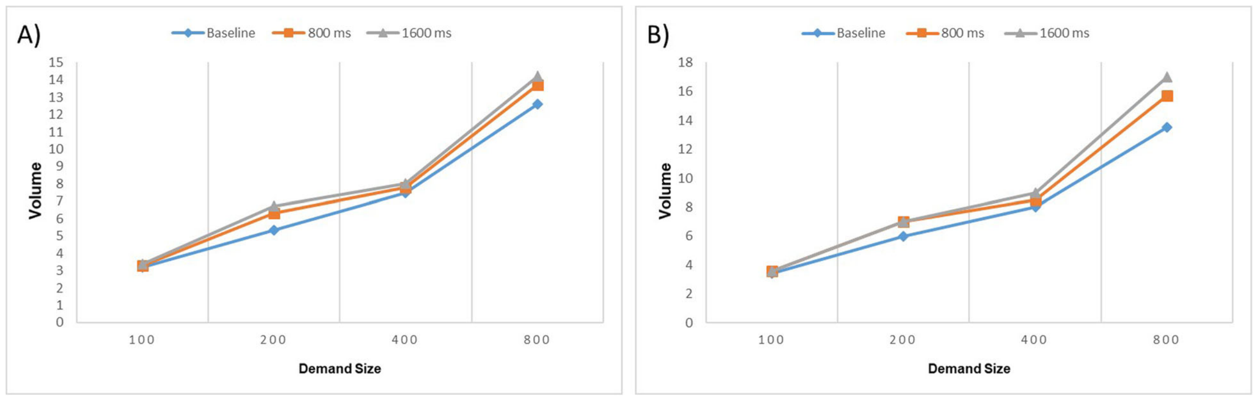

Fig. 6 The average number of software instances for the baseline, 800 ms, and 1600 ms delay latency experiments: **A**) steady rise and fall of demand; **B**) series of stepwise increases and decreases of demand

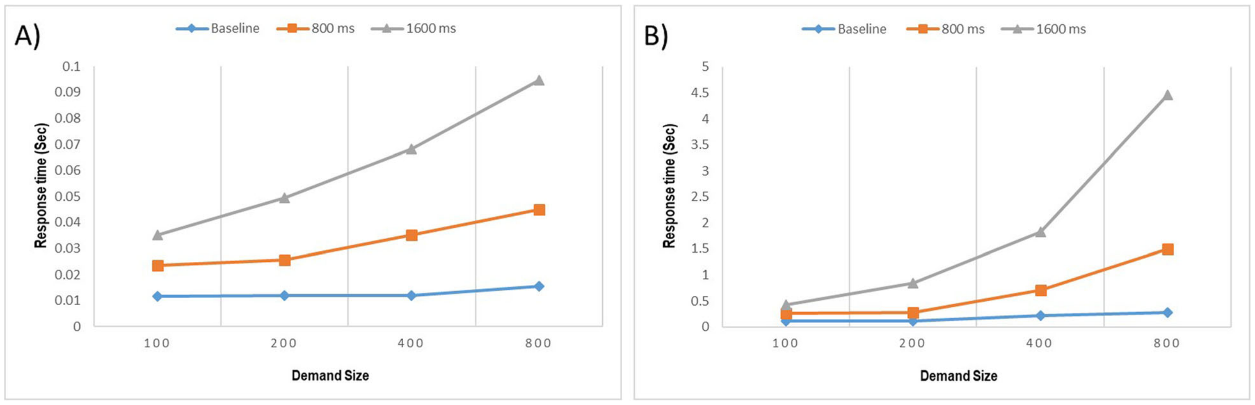

Fig. 7 The average response times for the baseline, 800 ms, and 1600 ms delay latency experiments: **A**) steady rise and fall of demand; **B**) series of stepwise increases and decreases of demand

and create a clear way to assess the resilience of scalability. The values for the scalability resilience calculated based on the ηI and ηt for the baseline and the two fault injection experiments conducted are shown in Table 5.

The above results of the resilience values show that the values drop gradually when the injected faults are stronger (i.e., 1600 ms delay has more impact than 800 ms). As we expected from these faults (i.e., latency delay), the impact on the quality (i.e., response time) resiliency is greater, howerver, we note the decreasing values of the resilience in scalability in terms of volume and quality. As noted, the quality resilience has been dropped 41% and 62% for the first scenario. In the contract, in terms of volume, the resilience dropped by 15% and 36% for the 800 ms and 1600 ms fault scenarios, respectively. In the context of the second scenario - which is likely to be a more realistic scenario for many services – we note that the drop of quality resiliency around 66% to 79%, and by 19% to 49% for the volume resiliency; for the 800 ms and 1600 ms fault scenarios respectively in relation to the baseline.

Discussion

This paper presents an experimental analysis of the impact of fault injection on the scalability resilience of cloud-based software services. The experimental framework based on the use of the ALFI has been explained, combining four components: workload generator, software fault, scalability measures, and the system under test and its environment. Previous studies on the

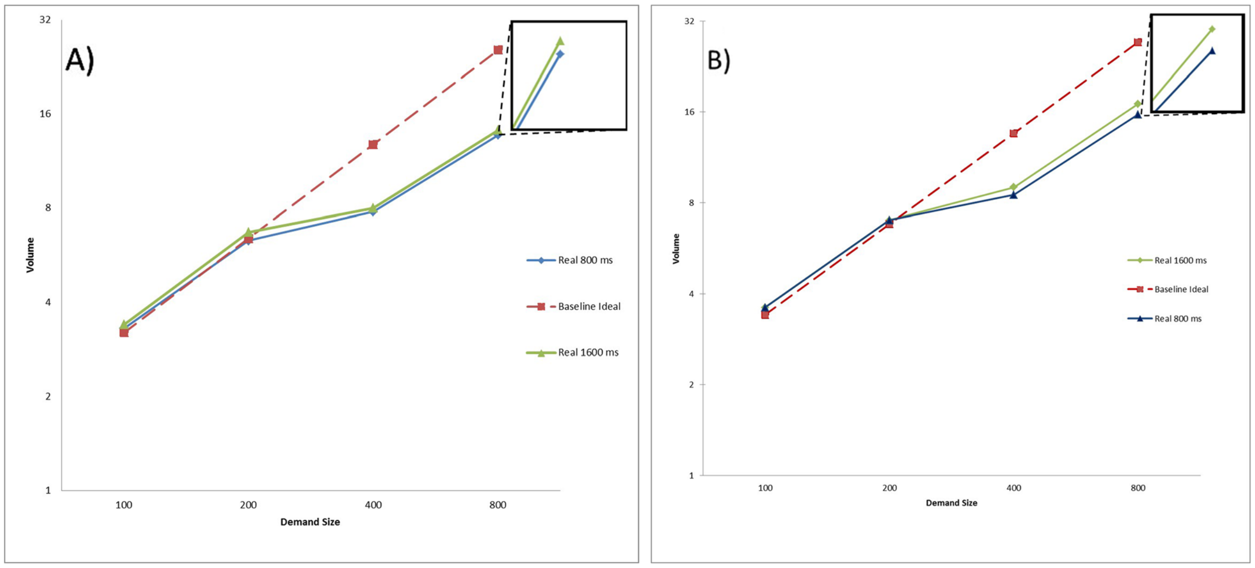

Fig. 8 The average number of software instances for baseline, 800, and 1600 ms experiments. **A**) steady rise and fall of demand; **B**) series of stepwise increases and decreases of demand

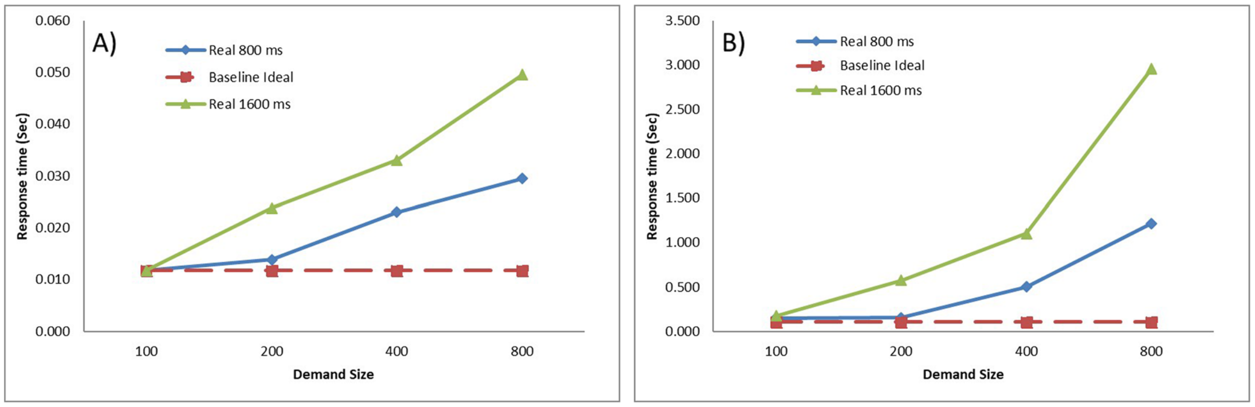

Fig. 9 The average response times for baseline, 800, and 1600 ms experiments. **A** steady rise and fall of demand; **B** series of stepwise increases and decreases of demand

scalability performance of cloud-based software services provide a baseline for the scalability behaviour of those services. An example using Amazon EC2 and OrangeHRM as a cloud-based software service has been employed to demonstrate the approach using delay latency injection with two different times, 800 and 1600 ms, and the data has been compared with the baseline data. This is important to determine whether the fault injection experiments significantly impact the scalability resilience of the software service. It should be noted that the delay latency faults cause a negative impact in terms of quality. Moreover, while the volume scaling is decreased in relation to the baseline, the quality indicator shows a significant drop in the performance of the service in terms of quality. The calculation of the scalability resilience values clearly indicates the impact of using the ALFI approach.

In this paper, the fault injection is considered by injecting delay latency into the software service at runtime. Other faults (such as adding latency/bandwidth, HTTP traffic, database traffic, or terminating requests) at the application level could also be considered to assess the true impact of faults on the scalability resilience of cloud-based software services and ascertain the type of impact on the scalability based on the nature of the fault. As noted in this study, the use of delay latency faults has affected the quality aspect of the scalability (i.e., response time) more than the scaling performance in terms of scaling instances.

This would provide useful behaviour benchmarking in relation to the scalability performance and resilience that can be used to assess the impact of faults on the delivery of the cloud-based software service from the perspective of scalability. This could help identify likely problems with the software or the cloud environment that delivers the cloud-based software service. Expanding the range of faults provides better benchmark data and a more comprehensive picture of cloud-based software services' scalability resilience under fault scenarios and techniques.

In this work, two demand scenarios were used to demonstrate the effect of demand patterns in the fault injection approach. In principle, considering a further set of fault injections incorporated with different scalability workload scenarios can also be used to pinpoint changes in such scenarios that might trigger interventions in terms of system upgrades or maintenance for the system under test.

The constraints in these results are due to the limited nature of the experimental investigation presented. First, the framework was demonstrated using one cloud-based

Table 4 Scalability metrics values

Scenario		Metric	
		η_I	η_t
Steady rise and fall (Demand Scenario 1)	Baseline (without fault injection)	0.5687	0.9041
	800 ms delay latency	0.4830	0.5318
	1600 ms delay latency	0.3621	0.3432
Stepwise increase and decrease (Demand Scenario 2)	Baseline (without fault injection)	0.5882	0.5201
	800 ms delay latency	0.4730	0.1768
	1600 ms delay latency	0.2988	0.1080

Table 5 Scalability resilience values

Scenario		Metric	
		η_I	η_t
Steady rise and fall (Demand Scenario 1)	Baseline (without fault injection)	1.0000	1.0000
	800 ms delay latency	0.8493	0.5882
	1600 ms delay latency	0.6367	0.3796
Stepwise increase and decrease (Demand Scenario 2)	Baseline (without fault injection)	1.0000	1.0000
	800 ms delay latency	0.8041	0.3399
	1600 ms delay latency	0.5080	0.2077

software service hosted into one public cloud environment. Naturally, further developing the experiments to cover multiple cloud environments and multiple software services will provide a better overview of the impact of the fault on the scalability resilience of such services. Moreover, two demand scenarios and one type of fault (delay Latency) were used with two settings. In contrast, a broader range of faults would give us a more profound understanding of how the approach varies depending on the nature of faults and the scalability demand scenario. Finally, one particular cloud instance's specifications and one fault generator were used to demonstrate the framework. Alternative workload and faults generators might impact the calculated metrics values due to their implementation details and preferences, although, in principle, it is not expected that these would significantly impact the reported results.

Conclusions and future directions

In this paper, an experimental framework of using ALFI to investigate the scalability resilience of cloud-based software services is presented. The experimental approach is explained, combining four components: workload generator, software fault, scalability measures, and the system under test and its environment. The framework was demonstrated using OrangeHRM hosted into EC2 and considering two demand scenarios incorporated with one type of fault with two settings (800, and 1600 ms delay latency). The results show that the proposed approach allows clear assessment of the impact of a fault scenario on the cloud service's scalability performance and resilience.

A major part of the method implemented in the ALFI approach is derived from the findings of previous studies [2], which set the baseline for measuring the scalability of cloud-based software services, which draws comparisons with the result of the fault injection experiments to assess the impact of this methodology. This allows for a clear assessment of the scalability resilience of cloud-based software services.

Naturally, future work will consider other cloud environments, other workload generators, fault types, and other cloud software services to obtain a wider range of scalability and fault measurements of the proposed framework, extending the practical validity of the work. An alternative of using commercial cloud infrastructure such as Amazon or Microsoft Azzure is to use private cloud infrastructures such as Openstack or Hewlett Packard Enterprise (HPE), or to use Cloud simulators (e.g. CloudSim) in order to conduct scalability experiments in combination with a set of experiments on public cloud platforms. The latter alternatives may provide lower cost experimental alternatives compared to commercial cloud services. Moreover, consider further demand patterns incorporated with faults to show how they impact the scalability resilience of cloud-based software services. This could help to establish volume and quality scalability metrics conditional on fault injection patterns.

Abbreviations

ALFI: Application-level fault injection; SaaS: Software as services; VMs: Virtual machines; η_I: Volume scalability metric; ηt: Quality scalability metric; D and D': Service demand volumes; I and I': The corresponding number of software instances; t_r and t'_r: The corresponding average response times

Acknowledgements

The publication of this research has been supported by The Deanship of Scientific Research and Graduate Studies at Philadelphia University - Jordan.

Authors' contributions

Both authors have participated in conception and design of the framework, ASA carried out the experimental work and results analysis. The work revised by PA. Both authors read, edited, and approved the final manuscript.

Authors' information

Amro Al-Said Ahmad, is an assistant professor at the Software Engineering Department, the Faculty of Information Technology at Philadelphia University in Jordan. He completed his Ph.D. in Cloud Computing (Scalability analysis of cloud-based systems) at the School of Computing and Mathematics of Keele University, UK. His main research interest is focused on technical scalability of cloud-based software services and delivery of such services from a technical perspective. Besides being a faculty member at PU, he is currently a collaborative researcher on Cloud Computing with the School of Computing and Mathematics at Keele University, UK.
Peter Andras has a BSc in computer science (1995), an MSc in artificial intelligence (1996) and a PhD in mathematical analysis of neural networks (2000), all from the Babes-Bolyai University, Cluj, Romania. He is a Professor of Computer Science and Dean of the Schools of Computing and Engineering & Built Environment at Edinburgh Napier University, UK. He has published 2 books and over 100 papers. He works in the areas of complex systems, computational intelligence and computational neuroscience. Dr. Andras is Senior

Member of IEEE, member of the International Neural Network Society, of the Society for Artificial Intelligence and Simulation of Behaviour, and Fellow of the Royal Society of Biology.

Funding
Not applicable.

Availability of data and materials
Not applicable.

Declarations

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Faculty of Information Technology, Philadelphia University, Amman, Jordan. [2]School of Computing and Mathematics, Keele University, Newcastle-under-Lyme, UK. [3]Schools of Computing, Engineering & the Built Environment, Edinburgh Napier University, Edinburgh, UK.

Received: 27 March 2021 Accepted: 16 December 2021
Published online: 03 January 2022

References

1. Al-Said Ahmad A, Andras P (2018) Measuring and testing the scalability of cloud-based software services. In: 2018 fifth international symposium on innovation in information and communication technology (ISIICT). IEEE, Amman, pp 1–8
2. Al-Said Ahmad A, Andras P (2019) Scalability analysis comparisons of cloud-based software services. J Cloud Comput 8. https://doi.org/10.1186/s13677-019-0134-y
3. Guo L, Liang J, Li D (2016) Understanding ineffectiveness of the application-level fault injection. In: Poster in ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. SC
4. Feinbube L, Pirl L, Tröger P, Polze A (2017) Software fault injection campaign generation for cloud infrastructures. In: proceedings - 2017 IEEE international conference on software quality, reliability and security companion. QRS-C, pp 622–623
5. Sheridan C, Whigham D, Artač M (2016) DICE fault injection tool. In: Proceedings of the 2Nd International Workshop on Quality-Aware DevOps. ACM, New York, pp 36–37
6. Banzai T, Koizumi H, Kanbayashi R et al (2010) D-cloud: design of a software testing environment for reliable distributed systems using cloud computing technology. CCGrid 2010 10th IEEE/ACM Int Conf Clust Cloud Grid Comput: 631–636. https://doi.org/10.1109/CCGRID.2010.72
7. Avizienis A, Laprie J-., Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Secur Comput 1:11–33 .https://doi.org/10.1109/TDSC.2004.2
8. Natella R, Cotroneo D, Madeira HS (2016) Assessing dependability with software fault injection: a survey. ACM Comput Surv 48:44:1--44:55. https://doi.org/10.1145/2841425
9. Piscitelli R, Bhasin S, Regazzoni F (2017) Fault attacks, injection techniques and tools for simulation. In: Hardware Security and Trust. Springer, pp 27–47
10. Huber N, Brosig F, Dingle N et al (2012) Providing Dependability and Performance in the Cloud: Case Studies. In: Katinka W, Alberto A, Vieira M, van Aad M (eds) Resilience Assessment and Evaluation of Computing Systems, pp 391–412
11. Herscheid L, Richter D, Polze A (2015) Experimental assessment of cloud software dependability using fault injection. In: Doctoral Conference on Computing, Electrical and Industrial Systems. Springer, pp 121–128
12. Ye K, Liu Y, Xu G, Xu C-Z (2018) Fault injection and detection for artificial intelligence applications in container-based clouds. In: Luo M, Zhang L-J (eds) CLOUD computing – CLOUD 2018. Springer International Publishing, Cham, pp 112–127
13. Chaos Monkey (2021) Chaos Monkeyhttps://github.com/netflix/chaosmonkey. Accessed 12 Dec 2020
14. Chaos Engineering (2019) Principles of chaos engineering. https://principlesofchaos.org/. Accessed 1 Feb 2021
15. Herbst NR, Kounev S, Reussner R (2013) Elasticity in Cloud Computing: What It Is , and What It Is Not. In: Presented as part of the 10th international conference on autonomic computing. USENIX, San Jose, pp 23–27
16. Hasan Jamal M, Qadeer A, Mahmood W et al (2009) Virtual machine scalability on multi-core processors based servers for cloud computing workloads. In: proceedings - 2009 IEEE international conference on networking, architecture, and storage. NAS, pp 90–97
17. Gao J, Pattabhiraman P, Bai X, Tsai WT (2011) SaaS performance and scalability evaluation in clouds. In: proceedings - 6th IEEE international symposium on service-oriented system engineering, SOSE. IEEE, pp 61–71
18. Jayasinghe D, Malkowski S, Li J et al (2014) Variations in performance and scalability: an experimental study in IaaS clouds using multi-tier workloads. IEEE Trans Serv Comput 7:293–306. https://doi.org/10.1109/TSC.2013.46
19. Lehrig S, Sanders R, Brataas G et al (2018) CloudStore — towards scalability, elasticity, and efficiency benchmarking and analysis in cloud computing. Futur Gener Comput Syst 78:115–126. https://doi.org/10.1016/j.future.2017.04.018
20. Brataas G, Herbst N, Ivansek S, Polutnik J (2017) Scalability analysis of cloud software services. In: proceedings - 2017 IEEE international conference on autonomic computing, ICAC, pp 285–292
21. Kumari P, Kaur P (2018) A survey of fault tolerance in cloud computing. J King Saud Univ - Comput Inf Sci. https://doi.org/10.1016/j.jksuci.2018.09.021
22. Xiaoyong Y, Ying L, Zhonghai W, Tiancheng L (2014) Dependability analysis on open stack IaaS cloud: bug anaysis and fault injection. In: 2014 IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, pp 18–25
23. Deng Y, Mahindru R, Sailer A, et al (2017) Providing fault injection to cloud-provisioned machines
24. Heorhiadi V, Rajagopalan S, Jamjoom H et al (2016) Gremlin: systematic resilience testing of microservices. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), pp 57–66
25. Zhang L, Morin B, Baudry B, Monperrus M (2021) Maximizing error injection realism for Chaos engineering with system calls. IEEE Trans Dependable Secur Comput 1. https://doi.org/10.1109/TDSC.2021.3069715
26. Fehling C, Leymann F, Retter R et al (2014) Cloud computing patterns: fundamentals to design, build, and manage cloud applications. Springer
27. Amazon EC2 (2021) Elastic Load Balancing. https://aws.amazon.com/elasticloadbalancing/. Accessed 17 Aug 2021
28. Avizienis A, Laprie J-C, Randell B (2001) Fundamental concepts of dependability. University of Newcastle upon Tyne, Computing Science
29. Woodside M (2001) Scalability metrics and analysis of Mobile agent systems. In: Wagner T, Rana OF (eds) Infrastructure for agents, multi-agent systems, and scalable multi-agent systems. Springer, Berlin Heidelberg, pp 234–245
30. OrangeHRM OrangeHRM REST APIS. https://api.orangehrm.com/?url=/apidoc/index.html.
31. Microsoft Azure (2017) Caching. https://docs.microsoft.com/en-us/azure/architecture/best-practices/caching. Accessed 15 Aug 2021
32. JMeter (2021) JMeter HTTP Request. https://jmeter.apache.org/usermanual/component_reference.html#HTTP_Request. Accessed 1 Feb 2021

Publisher's Note