Search-Based Stress Testing the Elastic Resource Provisioning for Cloud-Based Applications

Abdullah Alourani, Md Abu Naser Bikas, Mark Grechanik

University of Illinois at Chicago
Chicago, Illinois, 60607
aalour2,mbikas2,drmark@uic.edu

Abstract. One of the main benefits of cloud computing is to enable customers to deploy their applications on a cloud infrastructure that provisions resources (e.g., memory) to these applications on as-needed basis. Unfortunately, certain workloads can cause customers to pay for resources that are provisioned to, but not fully used by their applications, and as a result their performances then deteriorate beyond some acceptable thresholds and the benefits of cloud computing may be significantly reduced or even completely obliterated. We propose a novel approach to automatically discover these workloads to stress test elastic resource provisioning for cloud-based applications. We experimented with four non-trivial applications on the Microsoft Azure cloud to determine how effectively and efficiently our approach explores a very large space of the workload parameters' values. The results show that our approach discovers the first irregular workload faster in the search space of over 10^{40} input combinations compared to the random approach, and it discovers more irregular workloads that result in much higher costs and performance degradations for applications in the cloud.

Keywords: cloud computing; performance testing; cloud elasticity; genetic algorithms; multi-objective optimization; irregular workloads, stress testing.

1 Introduction

One of the main benefits of cloud computing is to enable customers to deploy their applications on a cloud infrastructure that provisions resources (e.g., *virtual machines (VMs)*) to these applications on as-needed basis [26]. That is, instead of buying and hosting expensive hardware, customers pay for renting resources for running these applications from cloud computing facilities [22]. A fundamental problem of cloud computing is to provision resources according to the application's runtime needs in order to ensure that its performance does not worsen below a predefined threshold, and it affects the technology spending in the excess of $1 trillion by 2020 [29].

The decisions to provision certain resources are typically made by engineers who create and maintain cloud-based applications, and they express their decisions in rules. A common and frequently used rule recommended by the Amazon and Google Cloud documentations is to provision one more VM when the CPU's utilization increases above 80% [13,3,24]. There are many different rules like that for controlling cloud *elasticity*, a term that designates on-demand resource provisioning to an application

[5,14]. Unfortunately, the behaviours of the nontrivial applications are very complex, so some rules may be far from optimal in terms of allocating best possible resources for maximizing the applications' performance.

In performance testing, input workloads are often created that resemble typical usages of applications and their performance characteristics are analyzed for regular workloads. In this paper, we are interested in *irregular workloads*, whose occurrences are rare and deviate beyond what is normally expected and they are extremely difficult to predict. Whereas test input workload generation techniques concentrate on finding patterns in the existing past workloads [20], there is no approach for finding new irregular workloads for *stress testing*, where applications are used beyond the normal operational capacity to a breaking point [4]. Unfortunately, when irregular workloads happen, customers pay for resources that are provisioned to, but not fully used by their applications [18], and the benefits of cloud computing may be significantly reduced or even completely obliterated [2].

We propose a novel approach for automatically discovering irregular workloads that result in situations when customers pay for resources that are not fully used by their applications while at the same time, some performance characteristics of these applications are not met, i.e., the *Cost-Utility Violations of Elasticity (CUVE)*. We implemented our approach for *Testing for Infractions of CLoud Elasticity (TICLE)* that combined a search-based heuristic with rule-guided resource provisioning to discover irregular workloads that led to CUVEs. These irregular workloads and rules can be reviewed by developers and performance engineers, who optimize the rules to improve the performance of the corresponding application. To the best of our knowledge, TICLE is the first fully automatic CUVE approach for discovering irregular workloads for applications deployed on the cloud. We TICLEd four nontrivial open-source applications in the Microsoft Azure cloud to determine how automatically and accurately TICLE explored a large search space of over 10^{40} input combinations while discovering CUVEs. The results show that TICLE finds the first irregular workload faster thus enabling stakeholders to investigate its impact sooner, and it finds more irregular workloads that lead to much higher costs and performance degradations for applications in the cloud compared to the random approach. TICLE's source code and all the experimental data are publicly available [1].

2 Problem Statement

In this section, we provide a background on workloads and rules for elastic resource provisioning, discuss sources of CUVE, and formulate the problem statement.

2.1 Rules and Workloads

In general, `if-then` elasticity rules contain antecedents that describe the level of resource utilization (e.g., CPU utilization $\geqslant 80\%$), and the consequents that specify (de)provisioning actions (e.g., to (de)provision a VM). Unfortunately, rule creation is an error-prone manual activity, and provisioning certain resources using manually created rules does not often improve the application's performance. For example, when the CPU utilization reaches some threshold due to a lot of page swapping or a lack of the storage space, provisioning more CPUs does not fix the underlying cause that requires giving more memory and storage to the application. That is, often rules are not optimal in terms of allocating required resources based on projected applications' needs [18].

It is very difficult to create rules that provision resources optimally to maximize the performance of the application while minimizing the cost of its deployment. Doing so requires the application's owners to understand which resources to (de)provision at what points in execution, how the cost of the provisioned resources varies, and how to make trade-offs between the application's performance and these costs [16]. Optimal provisioning is difficult even for five basic resource types (i.e., CPU, RAM, storage, VM, and network connections), where each type has many different attributes (e.g., the Microsoft Azure documentation mentions 30 attributes [24], which result in tens of millions of combinations).

Definition 1. *An application workload is a time-dependent collection of request tuples as shown in Figure 1 that contains a function of time that maps a time interval to the subset of input requests and its input data.*

The *application workload* includes not only the static part of the input to the application (i.e., combinations of HTTP requests with their parameter values) but also the dynamic part that comprises the number of HTTP requests submitted to the application per time unit and how this number changes as a function of time [23]. For example, a workload specifies how the number of requests to the application fluctuates periodically according to a circular function $y_t = \alpha \sin \omega t$, where α is the amplitude of the workloads that designates the maximum number of HTTP requests, t is the discrete time of the execution, and ω is the periodicity coefficient.

Application workloads are often characterized by *fast fluctuations* and *burstiness*, where the former designates a fast irregular growth and then a decline in the number of requests over a short period of time, and the latter means that many inputs occur together in bursts separated by lulls in which they do not occur [25]. By changing the coefficients of the function, irregular workloads can be generated for stress testing in varying degrees of burstiness and fluctuation.

2.2 Sources of Cost-Utility Violations of Elasticity

There are two main sources of CUVE. First, there is a problem of provisioning resources to an application that are not optimal for achieving the application's best performance. For example, the application may not perform better with additionally provisioned many CPUs instead of some more RAM [18]. Recall that cloud providers recommend some generic rules for resource provisioning [13,3,24]. Often, during stress testing, applications are run under regular heavy workloads that reflect the expected pattern of usage (e.g., loads peak during evening hours when people shop online), and they are unable to find CUVEs that result from irregular workloads. As a result, when these workloads occur during deployment, resources that are provisioned to an application may not improve its performance; however, its owner still has to pay the cloud provider for these needlessly provisioned resources.

Second, when the cloud infrastructure allocates resources, there is a delay between the moment when the cloud assigns a resource to an application and the moment when this application takes control of this resource. There are at least a couple of reasons for this delay: the startup time for a VM that hosts the application or its components includes the VM's loading and initialization time by the underlying infrastructure; assigning a new CPU to the existing VM requires its hosted operating system to recognize this CPU, which takes from seconds to tens of minutes [21]. Of course, the cloud infrastructure starts charging the customer for the resources at the moment it provisions them rather than when the application can control these resources [18]. However, all these may be done in vain – if the application rapidly changes its runtime behavior during a resource initialization time, this resource may not be needed any more by the time it is initialized to maintain the desired performance of the application. As a result, during irregular workloads, customers pay for resources that are not used by their applications for some period of time resulting in performance degradations.

2.3 The Problem Statement

Software engineers make performance enhancements routinely during perfective maintenance [19] when they use mostly exploratory random performance testing to identify when the performance of the *Application Under Test (AUT)* worsens. In this paper, we address a fundamental problem of performance testing in the cloud – *how to increase the effectiveness and efficiency of obtaining irregular workloads for software applications deployed on the cloud that lead to instances of the CUVE*. The root of this fundamental problem is that using only regular workloads for applications as part of random exploratory performance testing results in a large number of executions, many of which are not effective in determining CUVE instances. Selecting randomly a subset of workloads often results in a complete absence of the CUVE instances. To the best of our knowledge, there is no automatic approach to obtain irregular workloads that can produce instances of the CUVE.

Specifically, we want to construct irregular workloads automatically using combinations of inputs to which some functions are applied to cause fluctuations and burstiness to detect situations where the cost increases significantly while the average throughput (i.e., a measure inverse to the response time) of the application decreases beyond a certain threshold defined in a *service level agreement (SLA)* that indicates a desired performance level and the provisioned resources remain under-utilized or even completely unused at the same time. This is an instance of the *multiobjective optimization problem (MOOP)*. Automatically discovering irregular workloads is very difficult in general, especially when trying to satisfy multiple conflicting constraints.

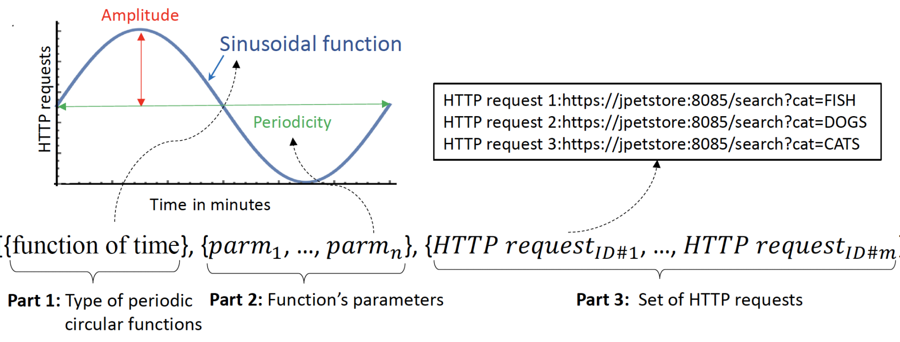
3 Our Approach

In this section, we state our key ideas for our approach for *Testing for Infractions of CLoud Elasticity (TICLE)*, explain the *genetic algorithm (GA)* with MOOP (*GAMOOP*), and describe the algorithm for TICLE.

3.1 Key Ideas

A goal of our approach is to automatically obtain irregular workloads for the AUT using GAMOOP. In general, GAs are based on natural selection techniques where solutions to optimization problems are obtained using a stochastic search [17]. The advantage of a GA is in evolving multiple candidate solutions in parallel thus allowing it to explore efficiently a large search space of possible solutions. Thus, TICLE is likely to scale well to modern AUTs with enormous search space.

In TICLE, a workload is represented by a *chromosome* that contains a sequence of *genes* divided into three parts as it is shown in Figure 1. The first part refers to the types of periodic circular functions (e.g., sinusoidal) that represent changes in the number of HTTP requests in the workload, the second part refers to the functions' parameters (e.g., amplitudes), and the third part refers to a set of HTTP requests, where each HTTP request is assigned to a unique ID, i.e., a HTTP request that includes various parameters is assigned to various IDs. For each application, we used a spider tool [8] to traverse the web interface of the application, log all unique HTTP requests sent to the backend of the application, and ensure these HTTP requests are valid. Each chromosome contains one function of time, two function parameters (e.g., amplitude and periodicity), and a set of HTTP requests, where each function of time uses only two function parameters. Therefore, modifying the values of these parameters in the second part of the chromosome by the GA is independent of changing the function of time in the first part of the chromosome. Once chromosomes are constructed, they are modified by GAs iteratively to find solutions that satisfy multiple objectives. That is, TICLE generates the combination of inputs (i.e., HTTP requests) plus the parameters of workloads for formulae that describe them.


Amplitude
Sinusoidal function
HTTP requests
Periodicity
Time in minutes
[{function of time}, $\{parm_1, ..., parm_n\}$, $\{HTTP\ request_{ID\#1}, ..., HTTP\ request_{ID\#m}\}$]
Part 1: Type of periodic circular functions
Part 2: Function's parameters
Part 3: Set of HTTP requests

HTTP request 1:https://jpetstore:8085/search?cat=FISH
HTTP request 2:https://jpetstore:8085/search?cat=DOGS
HTTP request 3:https://jpetstore:8085/search?cat=CATS

Fig. 1: The representation of the workload and the chromosome.

We use GAs for finding CUVEs that result from irregular workloads. In GAs, new solutions, or *offsprings* are generated using existing solutions, or *parents*. New solutions are often "fitter" to meet the objectives of the desired solution. A predefined *fitness function* is used to evaluate how close each solution is to being the optimal solution and fitter solutions have a better chance to "survive" multiple iterations [17]. In order to create a new generation of workload solutions, the operator selection, mutation, and crossover are applied to workloads, where a selection operator selects parents based on their fitness, a crossover operator recombines a pair of selected parents and generates new offspring workloads, and a mutation operator produces a mutant of one workload solution by randomly altering its gene. It is our hypothesis that GAMOOP can efficiently generate close to optimal workloads using the properties of their parents.

Our other key idea is to include user-defined rules for SLA violations as objective constraint functions for TICLE. For example, the Amazon's SLA rule limits the response time to 300ms for its web-based application [10]. Finding workloads that violate SLA thresholds is one of the main goals of performance testing. However, if finding workloads that break the SLA rules was the only objective, simply exponentially increasing the amplitude of the workloads with a very large burstiness would likely result in a sudden increase of the response time. Unfortunately, doing so results in ignoring the other two objectives (i.e., increasing the cost of the provisioned resources and decreasing the utilization of resources), since the cost is likely to remain the same if the cloud does not rapidly provision resources and the utilization will keep increasing with the increasing workloads. Thus, workload parameters should be chosen in such a way that delays between resource provisioning and resource availability are exploited by changing the fluctuations and the burstiness of the workloads in addition to differences in how applications use resources based on the workload content that includes HTTP requests, which trigger different execution paths in AUTs.

3.2 TICLE Algorithm

TICLE is shown in Algorithm 1 that includes the following major steps: (i) randomly generate an initial set of workloads, (ii) use these workloads to execute the cloud-deployed AUT and measure its performance, such as the utilization of the provisioned resources and the average response time, and (iii) use fitness functions, as described by Equation 2 [31] to evaluate the objectives and to select workload solutions using *the quality indicator* described by Equation 1 [31] to select solutions using GAMOOP. The fitness function is Pareto dominance compliant since it uses the quality indicator to rank solutions based on their usefulness regarding multiple objectives, amplifying the influence of dominating solutions over dominated solutions. A Pareto optimal solution dominates some other one if the dominating solution is better in some objectives and it is not worse in all the other objectives. Each solution can be represented as a point in a multidimensional space of orthogonal objectives. A curve can be drawn to connect non-dominated solutions that can be selected as optimal when no objective could be improved without sacrificing the other objectives. The curve is named a *Pareto optimal front* and is used by GAMOOP to choose winning workloads that result in CUVEs.

$$I(S,S') = \max\left\{\forall w' \in S'\, \exists w \in S : g_j(w) \geq g_j(w') \quad \text{for} \quad j \in \{1,\ldots,n\}\right\}, \qquad S,S' \in \Omega, \quad w,w' \in P \tag{1}$$

$$F(w) = \Sigma_{w' \in P \setminus \{w\}} - e^{-I\left(\{w'\},\{w\}\right)/k}, \quad k > 0 \tag{2}$$

Where Ω indicates the entirety of all Pareto sets, S is a Pareto set and S' is another Pareto set in all Pareto set approximations. P indicates the initial population P of workloads, w is a workload (i.e., solution), and w' is another workload in the population. I is the quality indicator function that compares the quality of two Pareto set approximations or

Algorithm 1 `TICLE`'s algorithm for automating workload search for instances of the CUVE problem.

```
1: Inputs: GAMOOP Configuration $\Omega$, Input Set $I$
2: $\mathcal{P} \leftarrow$ InitializePopulation($I$)
3: while $\neg$ Terminate do
4:     EvalFitnessObjectiveFunctions($\mathcal{P}$, $\Omega$)
5:     EvalConstraintsFunctions($\mathcal{P}$, $\Omega$)
6:     $\mathcal{F} \leftarrow$ FastNondominatedSort($\mathcal{P}$)
7:     CrowdingDistanceAssignment($\mathcal{F}$)
8:     $\mathcal{S} \leftarrow$ SelectParentsByRankDistance($\mathcal{F}$, $|\mathcal{P}|$)
9:     $\mathcal{R} \leftarrow$ RemoveLowerRankedSolutions($\mathcal{S}$)
10:    $\mathcal{C} \leftarrow$ CrossoverMutation($\mathcal{R}$, $\Omega$)
11:    $\mathcal{P} \leftarrow \mathcal{P} \cup$ Merge( $\mathcal{P}$, $\mathcal{C}$)
12: end while
13: return $\mathcal{P}$
```

solutions with respects to n objective functions $g_1,\ldots,g_n$ that are described below, k is a fitness scaling factor and is set to 0.05 experimentally.

We chose *Non-dominated Sorting Genetic Algorithm II (NSGA-II)* because previous evaluations showed that it finds a much better spread of solutions and it converges near the true Pareto optimal front. NSGA-II does not require the user to prioritize, scale, or weigh objectives like many other algorithms, which would be a major manual effort in TICLE. Finally, NSGA-II can generate new non-dominated solutions in unexplored parts of the Pareto front by applying the crossover operator to take advantage of good solutions with respect to multiple conflicting objectives [9].

That is, the space of workload parameters (e.g., the amplitude, periodicity) is explored to optimize three objectives in parallel by evaluating a fitness function (Equation 2) that maps workloads to the unused resources of provisioned VMs (objective 1), the cost of provisioned resources (objective 2), and the average response time (objective 3). An ideal solution is a workload that maximizes these objectives, as described by Equation 1, i.e., to achieve the maximum cost of the deployment with the minimum resource utilization and the application throughput that violates predefined SLA constraints. These objectives cannot be formally defined, since their values are obtained from the Microsoft Azure cloud. Since no solution exists to address this important problem, using NSGA-II to find a better solution and to compare it with a random performance testing approach is our major contribution.

The algorithm for TICLE takes in the complete set of input ranges for the subject AUT and the GAMOOP configurations Ω, including the crossover and mutation rates, fitness functions for their respective objectives, an SLA threshold, and the termination criterion. In Step 2, the algorithm generates an initial population of workloads by combining randomly selected HTTP requests. In TICLE, we create four types of workload fluctuation functions: sinusoidal, where the workload changes with periodicity, as described by the equation $y_t = \alpha \sin t$, where α is the amplitude of the workloads that designates the maximum number of HTTP requests, and t is the discrete time of the execution; linear, where the workload increases or decreases linearly, as described by the equation $y_t = \alpha \times t$; exponential, with a rapid rise or drop of the workload $y_t = \alpha^t$; and random, where a random number generator is used to define the amplitude and the HTTP requests for the workloads. In the RANDOM approach, a workload contains AUT's HTTP requests, the types of periodic circular functions that represent changes in the number of HTTP requests in the workload, and the functions' parameters (e.g., amplitudes and periodicities). Once workloads are constructed, their parameters are modified randomly to find solutions. Based on previous research, these functions represent a majority of workload shapes [23].

Starting from Step 3, the evolution process begins by evaluating if the termination condition is satisfied. In Step 4, fitness functions are applied to evaluate each individual workload and in Step 5 constraint functions are evaluated to determine if the SLA holds. After the evaluation, in Step 6 the population is sorted and in Step 7 the distances of the solutions on the Pareto front are estimated. Using those closest to the Pareto front, in Step 8 the solutions are ranked into a hierarchy of sub-populations based on the ordering of the Pareto dominance. In Step 9, lower ranked solutions are removed from the population. In Step 10, for each part of the chromosome, the mutation operator replaces the value of one random gene with another value within the specified range, thus creating a new (updated) individual.

All newly generated individual workloads are evaluated using the defined fitness functions, and the fittest workloads are selected for the next generation that is formed first by the order of dominating precedence of the Pareto front and then by using the distance within the front. Finally, the new workload solutions are added to the population. The cycle of Steps 3-12 repeats until the termination criterion is satisfied, and the final population is returned in Step 13 as the algorithm terminates.

4 Empirical Evaluation

In this section, we describe the design of the empirical study to evaluate `TICLE` and state threats to its validity. We pose the following three *Research Questions (RQs)*:

***RQ*$_1$: ** How effective is TICLE in finding irregular workloads that lead to the greater cost of the AUT's deployment?

***RQ*$_2$: ** How fast is TICLE in finding the first irregular workload that infracts the elasticity rules for the AUT?

***RQ*$_3$: ** Is TICLE more effective than the random approach in finding more CUVEs for different elasticity rules?

Table 1: Characteristics of the subject AUTs: their names followed by their versions, the number of lines of code (LOC), the number of classes, the number of methods and the approximate size of the search space of the input requests for the AUT.

AUT	Version	LOC	Classes	Methods	Space
JPetStore	*v*4.0.5	2,762	42	400	10^{31}
JForum	*v*2.1.9	36,401	397	3,487	10^{49}
PhotoV	*v*2.1.0	10,549	81	931	10^{36}
RUBiS	*v*1.4.3	83,640	641	4,396	10^{14}

4.1 Subject Applications

We evaluated TICLE on four web-based, open-source subject applications written in Java: JPetStore, JForum, PhotoV, and RUBiS. Their basic characteristics are shown in

Table 1. These applications are written by different programmers, come from different domains, and have high popularity indexes. Choosing up to 50 input requests from 100+ HTTP requests results in over 10^{40} combinations.

All subject AUTs have a three-tier architecture. Response time is measured between the moment when a sent request is received by the AUT and the moment when a response to the request is issued from the AUT, and the network latency time is not included. All components of the same AUT are deployed on the same VM. When the cloud provisions VMs to the AUT, each VM will have a replica of these three tiers to ensure full horizontal scalability of the AUT.

4.2 Methodology

We use the definition a *workload* from Section 2.1 to specify the set of input requests and how their quantities change over time. For example, the HTTP request `https://jpetstore:8085/search?cat=FISH` is an input to JPetStore, where `search` is the path component of the HTTP request, `cat` is the name of its parameter, and `FISH` is the value of this parameter. TICLE generates workloads and uses JMeter [15] that simulates users sending the workload requests to web servers of the AUT and collects performance measurements of the provisioned VMs that host AUT's components that execute the workload requests. In our experiments, we set the number of HTTP requests in a workload between 10 and 50 to observe a wide range of the AUT's behaviors.

Table 2: The set of predefined `if-then` elasticity rules.

Rule	**Provisioning Action**	
	Scale In	**Scale Out**
R_1	$CPU_{utilization} < 20\%$	$CPU_{utilization} > 50\%$
R_2	$CPU_{utilization} < 40\%$	$CPU_{utilization} > 60\%$
R_3	$CPU_{utilization} < 20\%$	$CPU_{utilization} > 80\%$

Also, we defined three elasticity rules with different ranges for VM (de)provisioning that are shown in Table 2 to determine how effectively TICLE finds irregular workloads that infract these elasticity rules for the AUTs. Since our goal is to find irregular workloads that lead to CUVEs, violating the predefined SLA threshold is an important objective of the experiments. We use the AUT's response time as the SLA. To determine the SLA threshold, we first run each subject AUT under heavy workloads in a single VM to determine the longest possible response time. Then, we repeat our experiments with 20%, 40%, and 60% of this longest response time as the SLA threshold.

The experiments for the AUTs were carried out using 10 small VMs/servers from the A-series in the Microsoft Azure cloud called Standard A1 with 1 GHz CPU and 1.75 GB of memory. We wrote a client for JMeter [15] that applied generated workloads to the subject AUTs, and JMeter clients were run externally on laptops. All experiments were conducted on the same experimental platform.

We implemented TICLE using `jMetal`, which is an open-source framework for multi-objective optimization with various evolutionary algorithms [11]. We used the following GAMOOP settings for TICLE: the crossover rate of 0.9, the mutation rate of 0.3, the population of 100 individuals, and the tournament selection of size two. The evolution was terminated if the workload solutions did not improve after 10 generations. The maximum number of generations was set to 30. We chose these values experimentally for the platform based on the limitations of the hardware.

4.3 Threats to Validity

A threat to the validity of our empirical study is that our experiments were performed on only four open-source, web-based applications, which makes it difficult to generalize the results to other types of applications that may have different logic, structure, or input types. However, the subject AUTs were used in other empirical studies on performance testing [27]. Therefore, we expect our results to be generalizable.

Our current implementation of TICLE deals with simple types of inputs, HTTP requests with basic parameter types (e.g., integer), whereas other programs may have complex input types (e.g., JSON or XML structures). While this is a threat, TICLE can be adapted to encode inputs of other types. In order to apply TICLE to other applications, the user needs to modify only the gene representation approach so that TICLE recognizes other types of inputs.

One threat to validity is that we deployed an AUT fully in a single VM. Indeed, deploying an AUT's components in multiple VMs may lead to performance bottlenecks since many shared resources are used in the application layer. This situation may result in more CUVEs, thus making it easier for TICLE to find them. However, deploying these layers on the same VM (i.e., it is scaled horizontally) puts TICLE at a disadvantage to find CUVEs since many bottlenecks do not show up easily, thus making our experiments robust.

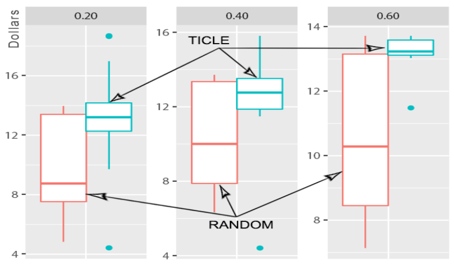
We experimented with only three generic elasticity rules using the recommendations from Amazon, Azure, and Google Cloud documentations. This is a threat for two reasons. First, users may create much more sophisticated rules that would make it difficult for TICLE to find CUVEs. Second, our rules provision only VMs, whereas real-world rules could also provision storage, RAM, network connections, and other virtual hardware. However, understanding the effect of various resources is currently out of scope for this paper and will be addressed in future work.

5 Empirical Results

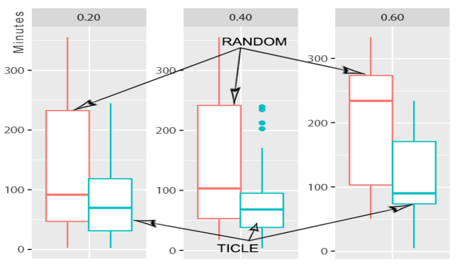
In this section, we describe and analyze the results of the experiments to answer the three RQs stated in Section 4.

5.1 Finding Workloads that Lead to Higher Costs

The results of the experiments are shown in the box-and-whisker plots in Figure 2a and Figure 2b that summarize the deployment costs and the time it takes to find the first CUVE for the subject AUTs using the `TICLE` and `RANDOM` approaches for three different SLA threshold values of the longest response time. We observe that the average costs for the found CUVEs using `TICLE` are consistently higher than the average costs of the CUVEs found by `RANDOM` among all SLA threshold values. The costs for CUVEs have the highest difference between `TICLE` and `RANDOM` at 60% of the SLA threshold, then at 40%, followed by 20%. This result suggests that the higher threshold values require more sophisticated workloads to break the threshold and to lead to a higher cost



(a)



(b)

Fig. 2: Box-and-whisker plots compare (a) the deployment costs and (b) the time to the first CUVE discovery for detected CUVEs that are computed using the `TICLE` and `RANDOM` approaches for the subject AUTs for three SLA thresholds (i.e., 0.2, 0.4, and 0.6) of the longest response time. The cost is measured in dollars and the time is measured in minutes.

Table 3: The comparison of the results of Mann-Whitney-Wilcoxon U-Tests for `TICLE` and `RANDOM` using three SLA thresholds. The first column designates the null hypothesis followed by the column for SLA thresholds, and the cells contain the *p*-values.

	SLA Threshold		
Null Hypothesis	**20%**	**40%**	**60%**
Cost	9.7×10^{-15}	8.2×10^{-3}	0.03
Detection Time	1.4×10^{-4}	5.5×10^{-4}	0.02

of deployment, because it is more difficult to construct workloads when longer response times are permitted. The cost variance for CUVEs computed by `TICLE` is significantly lower when compared to the `RANDOM` approach, which suggests that `TICLE` favors workloads that have the highest impact on increasing the cost of deployment.

Similarly, it is shown in the box-and-whisker plot in Figure 2b that `TICLE` is consistently faster than `RANDOM` in finding the first CUVE. This result is important not only to answer ***RQ***$_2$, but also to show that `TICLE` is efficient in practice, since taking less time to find the first CUVE shows that `TICLE` beats the `RANDOM` approach in notifying stakeholders faster that there is a workload that results in a CUVE. We expect that `TICLE` will be used by performance testers, and it is important for them to find CUVEs faster to report them to developers who will start looking for fixes to the detected CUVEs. Thus, a faster-to-find-CUVE approach is also more efficient in using fewer computer resources and stakeholders' time.

In our case, the data cannot be guaranteed to follow the normal distribution, therefore, we applied Mann-Whitney-Wilcoxon U-Tests to evaluate the statistical significance of the difference in the median value of deployment cost between `TICLE` and `RANDOM` for the subject AUTs. The results of Mann-Whitney-Wilcoxon U-Tests for `TICLE` and `RANDOM` are shown in Table 3. The results confirm that the values for the differences between `TICLE` and `RANDOM` are always statistically significant according to the Mann-Whitney-Wilcoxon U-Test, thus **positively addressing *RQ***$_1$.

5.2 Finding Workloads Faster

We applied Mann-Whitney-Wilcoxon U-Tests to evaluate the statistical significance of the difference in the median value of detection time, which indicates the execution time to find irregular workloads that lead to the CUVE, between `TICLE` and `RANDOM` for the subject AUTs. The results of Mann-Whitney-Wilcoxon U-Tests for `TICLE` and `RANDOM` are shown in Table 3. The results confirm that the values for the differences between `TICLE` and `RANDOM` are always statistically significant according to the Mann-Whitney-Wilcoxon U-Test, thus **positively addressing *RQ*$_2$**, which states that `TICLE` is more efficient in finding CUVE using significantly fewer computational resources compared to the `RANDOM` approach.

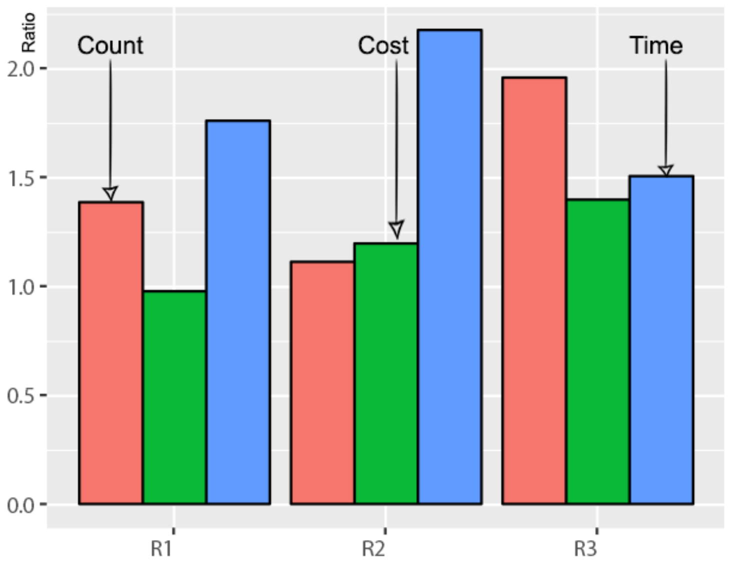


Fig. 3: Comparing `TICLE` and `RANDOM` for detecting CUVEs for the subject AUTs with different elastic rules that are shown in Table 2. The X-axis designates elasticity rules. The leftmost red bar represents the ratio of the total number of detected CUVEs using the approaches `TICLE` and `RANDOM`, $\frac{count_{TICLE}}{count_{RANDOM}}$. The middle green bar represents the ratio of the average costs for CUVEs, $\frac{cost_{TICLE}}{cost_{RANDOM}}$. The righmost blue bar represents the ratio of detection times for the first found CUVE, $\frac{time_{RANDOM}}{time_{TICLE}}$.

5.3 The Impact of the SLA Threshold

An interesting question is how an SLA threshold affects the process of finding CUVEs. As discussed in Section 4.2, a higher percentage of the SLA threshold means that longer response times are acceptable. Since one of the objectives is to find CUVEs where the SLA threshold is violated, the higher the percentage at which the SLA threshold is chosen, the more difficult it is to obtain CUVEs. Consider the box-and-whisker plots that are shown in Figure 2a and Figure 2b – the visual inspection clearly identifies the rise of the average cost and the detection time with the increase of the SLA threshold.

However, our analysis shows that the cost of the application deployment increases robustly when using `TICLE` whereas for `RANDOM`, the average cost stays approximately the same, but it shows a much wider variance. Our explanation is that `TICLE` is more effective in finding workloads for CUVEs with much higher SLA thresholds, since it systematically chooses workloads with a higher cost using the fitness functions.

Alternatively, the detection time to the first occurrence of the CUVE shows almost an opposite pattern. The detection time increases steadily when using `RANDOM` with a large variance of the measurements whereas for `TICLE`, the average detection time stays approximately the same, and it shows a much smaller variance. Again, this observation confirms the efficiency of `TICLE` when the SLA threshold increases.

5.4 Impact of Different Elasticity Rules

The results of the experiments to answer $\boldsymbol{RQ}_3$ are presented in the histogram plot in Figure 3 that shows ratios for the total numbers of detected CUVEs, deployment costs, and detection times computed using the approaches `TICLE` and `RANDOM` over subject AUTs for three elasticity rules, which allocate and deallocate resources in consonance with the user-specific conditions (i.e., the utilization of CPUs increases above 80%). We used three elasticity rules that are recommended by the Amazon, Microsoft Azure, and Google Cloud documentations [13,3,24], and these rules are shown in Table 2. The higher the ratios, the more effective and efficient `TICLE` is in finding CUVEs compared to the `RANDOM` baseline approach.

We observe that all ratios with the exception of one for the deployment cost of the rule $\mathbf{R}_1$ are greater than one meaning that `TICLE` finds faster and more CUVEs when compared to `RANDOM`. The highest count ratio is for $\mathbf{R}_3$ and $\mathbf{R}_1$, followed by $\mathbf{R}_2$, which suggests that a higher range value between the lower threshold that triggers the scale-in operation and the upper threshold that triggers the scale-out operation for elasticity rules results in more detected CUVEs. In summary, these experimental results demonstrate that `TICLE` is more effective and efficient in finding CUVEs for all elasticity rules than the `RANDOM` baseline approach, thus **positively addressing $\boldsymbol{RQ}_3$**.

6 Related Work

Gambi et al. developed a tool that uses predefined workloads to test the automation of cloud-based elastic systems [12]. Bodik et al. proposed a workload model that characterizes volume and data spikes to test the robustness of stateful systems [6]. Chen et al. developed a tool that uses user-defined workloads to analyze performance and energy consumption for cloud applications [7]. Snellman et al. developed a tool that uses user-defined test scripts to evaluate the performance and scalability of rich internet applications in the cloud[28]. Shen *et al.* presented an approach that uses genetic algorithms to find the combinations of inputs that lead to performance problems [27]. Xiao *et al.* presented an approach that uses complexity models to predict workload-dependent performance bottlenecks [30]. However, `TICLE` is the first fully automatic approach that finds irregular workloads that lead to the CUVEs for stress-testing applications deployed on the cloud.

7 Conclusion

We presented a novel approach for automating the discovery of situations when customers pay for resources that are not fully used by their applications while at the same time, some performance characteristics of these applications are not met, i.e., the cost-utility violations. We implemented our approach for *Testing for Infractions of CLoud Elasticity* (`TICLE`) and we `TICLE`d four nontrivial open-source applications in the Microsoft Azure cloud. The results show that TICLE is effective for automatic stress testing of elastic resource provisioning for applications deployed on the cloud to determine infractions of elastic rules. With TICLE, experts can analyze the discovered workloads to determine their impact on applications. To the best of our knowledge, TICLE is the first fully automatic approach for discovering irregular workloads that are very difficult to create using other approaches.

8 Acknowledgments

We warmly thank Prof. Márcio Barros and anonymous reviewers for their comments and suggestions that helped us to improve the quality of this paper. This work is supported by NSF EAGER-1650000, NSF CCF-1615563, NSF I-Corps-1547597, Microsoft, and Grammatech. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

1. Ticle source code and experimental data. ”`https://www.dropbox.com/s/c2rs5afh5g4icdl/TICLEProject.zip?dl=0`” (2018)
2. Albonico, M., Mottu, J.M., Sunyé, G.: Controlling the elasticity of web applications on cloud computing. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing. pp. 816–819. SAC '16, ACM, New York, NY, USA (2016)
3. AWS: What is auto scaling? ”`http://docs.aws.amazon.com`” (2018)
4. Beizer, B.: Software testing techniques. Dreamtech Press (2003)
5. Bikas, M.A.N., Alourani, A., Grechanik, M.: How elasticity property plays an important role in the cloud: A survey. Advances in Computers **103**, 1–30 (2016). https://doi.org/10.1016/bs.adcom.2016.04.001, `https://doi.org/10.1016/bs.adcom.2016.04.001`
6. Bodik, P., Fox, A., Franklin, M.J., Jordan, M.I., Patterson, D.A.: Characterizing, modeling, and generating workload spikes for stateful services. In: Proceedings of the 1st ACM symposium on Cloud computing. pp. 241–252. ACM (2010)
7. Chen, F., Grundy, J., Schneider, J.G., Yang, Y., He, Q.: Stresscloud: a tool for analysing performance and energy consumption of cloud applications. In: Proceedings of the 37th International Conference on Software Engineering-Volume 2. pp. 721–724. IEEE Press (2015)
8. crawler4j: Open source web crawler for java. `https://github.com/yasserg/crawler4j` (2018)
9. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: Nsga-ii. Trans. Evol. Comp **6**(2), 182–197 (Apr 2002)
10. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: ACM SIGOPS operating systems review. vol. 41, pp. 205–220. ACM (2007)

11. Durillo, J.J., Nebro, A.J.: jmetal: A java framework for multi-objective optimization. Adv. Eng. Softw. **42**(10), 760–771 (Oct 2011)
12. Gambi, A., Hummer, W., Dustdar, S.: Automated testing of cloud-based elastic systems with autocles. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on. pp. 714–717. IEEE (2013)
13. Google: Autoscaling groups of instances. ”`https://cloud.google.com`” (2018)
14. Grechanik, M., Luo, Q., Poshyvanyk, D., Porter, A.: Enhancing rules for cloud resource provisioning via learned software performance models. In: Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering, ICPE 2016, Delft, The Netherlands, March 12-16, 2016. pp. 209–214 (2016). https://doi.org/10.1145/2851553.2851568, `http://doi.acm.org/10.1145/2851553.2851568`
15. Halili, E.: Apache JMeter. Packt Publishing (2008)
16. Herbst, N.R., Kounev, S., Reussner, R.: Elasticity in cloud computing: What it is, and what it is not. In: Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13). pp. 23–27. USENIX, San Jose, CA (2013)
17. Holland, J.H.: Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence. U Michigan Press (1975)
18. Islam, S., Lee, K., Fekete, A., Liu, A.: How a consumer can measure elasticity for cloud platforms. In: Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering. pp. 85–96. ICPE ’12, ACM, New York, NY, USA (2012)
19. Lientz, B.P., Swanson, E.B.: Software Maintenance Management. Addison-Wesley (1980)
20. Liu, Z., Cho, S.: Characterizing machines and workloads on a google cluster. In: Proceedings of the 2012 41st International Conference on Parallel Processing Workshops. pp. 397–403. ICPPW ’12, IEEE Computer Society, Washington, DC, USA (2012)
21. Mao, M., Humphrey, M.: A performance study on the vm startup time in the cloud. In: Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing. pp. 423–430. CLOUD ’12, IEEE Computer Society, Washington, DC, USA (2012)
22. Mendelson, H.: Economies of scale in computing: Grosch’s law revisited. Commun. ACM **30**(12), 1066–1072 (Dec 1987)
23. Mian, R., Martin, P., Zulkernine, F., Vazquez-Poletti, J.L.: Towards building performance models for data-intensive workloads in public clouds. In: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering. pp. 259–270. ICPE ’13, ACM, New York, NY, USA (2013)
24. MSAzure: Autoscaling. ”`https://docs.microsoft.com`” (2018)
25. Perez-Palacin, D., Mirandola, R., Scoppetta, M.: Simulation of techniques to improve the utilization of cloud elasticity in workload-aware adaptive software. In: Companion Publication for ACM/SPEC on International Conference on Performance Engineering. pp. 51–56. ICPE ’16 Companion, ACM, New York, NY, USA (2016)
26. Peter Mell and Tim Grance: The NIST Definition of Cloud Computing. ”`http://csrc.nist.gov/groups/SNS/cloud-computing/cloud-def-v15.doc`” (2009)
27. Shen, D., Luo, Q., Poshyvanyk, D., Grechanik, M.: Automating performance bottleneck detection using search-based application profiling. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. pp. 270–281. ACM (2015)
28. Snellman, N., Ashraf, A., Porres, I.: Towards automatic performance and scalability testing of rich internet applications in the cloud. pp. 161–169. In SEAA ’11, IEEE (2011)
29. vanderMeulen, R.: Gartner says by 2020 ”cloud shift” will affect more than $1 trillion in it spending. ”`http://www.gartner.com/newsroom/id/3384720`” (2018)
30. Xiao, X., Han, S., Zhang, D., Xie, T.: Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In: ISSTA ’13. pp. 90–100 (2013)
31. Zitzler, E., Künzli, S.: Indicator-based selection in multiobjective search. In: International Conference on Parallel Problem Solving from Nature. pp. 832–842. Springer (2004)