Economically Deploying Applications in Elastic Clouds

by

Abdullah Alourani
M.S., Computer Science, DePaul University, 2013
B.S., Computer Science, Qassim University, 2007

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2020

Chicago, Illinois

Defense Committee:
Prof. Ajay D. Kshemkalyani, Chair and Advisor
Prof. A. Prasad Sistla
Prof. V.N. Venkatakrishnan
Prof. Balajee Vamanan
Prof. Jalal Alowibdi, University of Jeddah

This thesis is dedicated to my father, Ibrahim, my mother, Khadijah, and my family.

ACKNOWLEDGMENT

First and foremost, I would like to thank my Ph.D. advisor, Prof. Ajay D. Kshemkalyani, for his continuous guidance, support, dedication, and encouragement. Working with him during the past years has been my greatest honor. This thesis would not have been possible without his supervision and endless help. I will always be indebted to him for giving me the great opportunity to pursue my Ph.D. degree.

In addition, I would like to thank the other members of my dissertation committee, Prof. A. Prasad Sistla, Prof. V.N. Venkatakrishnan, Prof. Balajee Vamanan, and Prof. Jalal Alowibdi, for spending their valuable time serving on the committee. I am grateful for their insightful discussions and suggestions that contributed to this thesis. Also, I would like to express my sincere gratitude and appreciation to the former director of graduate studies, Prof. Robert Kenyon, for helping me in many ways during my Ph.D. study. I also want to thank my collaborators for their valuable time and efforts that contributed to this thesis.

Last but not least, I would like to express my deepest gratitude and appreciation to my family for their unconditional love, continuous encouragement, and endless support. I especially want to thank my parents for everything they have done for me.

AIA

CONTRIBUTIONS OF AUTHORS

Chapter 1 presents the thesis introduction, states the thesis statement, and highlights the research contributions.

Chapter 2 provides the necessary background on cloud instance types and highlights the rules that contribute to economically deploying applications in elastic clouds.

Chapter 3 compares the related work with our work in this thesis.

Chapter 4 presents a published paper (Alourani et al., 2018 [1]), for which I was the first author and the primary investigator. Md. Abu Naser Bikas contributed to the implementation of the framework, the design of the illustrative example, and the writing with respect to the preliminary ideas, along with the planning and structure of the work. Mark Grechanik contributed to the writing of the paper, along with the planning and structure of the work.

Chapter 5 presents a published paper (Alourani et al., 2019 [2]) and a published paper (Alourani et al., 2020 [3]), for which I was the first author and the primary investigator. My advisor, Ajay D. Kshemkalyani, contributed to the writing of the papers, in addition to the planning and structure of the work. Mark Grechanik contributed to the writing with respect to the problem, along with the planning and structure of the work with respect to the problem.

Chapter 6 presents P-SIWOFT, Provisioning Spot Instances WithOut employing Fault-Tolerance mechanisms.

Chapter 7 concludes this thesis and highlights future work.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (Continued)

FIGURE **PAGE**

LIST OF ABBREVIATIONS

AUT Application Under Test

BASIR Bugs of cloud-based Applications resulting from Spot Instance Revocations

CUVE Cost-Utility Violations of Elasticity

DCATO Deployment Cost And Time Overhead

GA Genetic Algorithm

GAMOOP Genetic Algorithm with Multiobjective Optimization Problem

IaaS Infrastructure as a Service

JVM Java Virtual Machine

KM Kernel Modules

MOOP Multiobjective Optimization Problem

MTTR Mean Time To Revocation

NSGA-II Non-dominated Sorting Genetic Algorithm II

P-SIWOFT Provisioning Spot Instances WithOut employing Fault-Tolerance mechanisms

PaaS Platform as a Service

LIST OF ABBREVIATIONS (Continued)

RAT Resources Affected by Termination

SaaS Software as a Service (SaaS)

SLA Service Level Agreement

T-BASIR Testing for Bugs of Cloud-Based Applications Resulting from Spot Instance Revocations

TICLE Testing for Infractions of Cloud Elasticity

VM Virtual Machine

SUMMARY

Cloud computing provides key features of cloud platforms to enable customers to economically deploy their applications. First, customers can deploy their applications on a cloud infrastructure that provisions resources (e.g., memory) to these applications on as-needed basis. However, certain workloads can result in situations when customers pay for resources that are provisioned, but not fully used by their applications, and as a result, some performance characteristics of these applications are not met, i.e., the Cost-Utility Violations of Elasticity (CUVE). Second, customers can economically deploy their applications on cloud spot instances (i.e., virtual machines (VMs)) in cloud computing at much lower costs than that of other types of cloud instances. In exchange, spot instances are often exposed to revocations (i.e., terminations) by cloud providers; thus, when applications that run in spot instances are being irregularly terminated due to spot instance revocations, these applications might lose their states that lead to certain bugs, i.e., Bugs of cloud-based Applications resulting from Spot Instance Revocations (BASIR). Also, applications often employ different fault-tolerance mechanisms to minimize the lost work for each spot instance revocation. However, these fault-tolerance mechanisms incur additional overhead related to application completion time and deployment cost, i.e., the Deployment Cost And Time Overhead (DCATO). Unfortunately, cloud-based applications are not designed or tested to deal with CUVE, BASIR, and DCATO problems in the cloud environment, and as a result, the benefits of economically deploying applications in elastic clouds may be significantly reduced or even completely obliterated. In this thesis, we propose a novel model that reduces the impact of CUVE, BASIR, and DCATO problems in the cloud environment to economically deploy applications in elastic clouds, and this model leads to practical frameworks for optimizing cloud elasticity, improving the design of the shutdown process, and reducing the deployment cost and completion time for cloud-based applications. This ensures efficient cloud computing services that lead to greater economies of scale.

In the first work, we develop a novel approach for Testing for Infractions of Cloud Elasticity (`TICLE`) that combines a search-based heuristic with rule-guided resource provisioning by stress testing the elastic resource provisioning for cloud-based applications to automatically discover irregular workloads that led to CUVE. We conduct our experiments with four nontrivial open-source applications in the Microsoft Azure cloud to determine how automatically and accurately `TICLE` explores 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.

In the second work, we implement a novel approach for Testing for Bugs of Cloud-Based Applications Resulting from Spot Instance Revocations (`T-BASIR`) that uses kernel modules to automatically find BASIR and locate their causes in the source code. We evaluate `T-BASIR` using 10 popular open-source applications. Our results show that `T-BASIR` not only finds more instances and different types of BASIR (e.g., data loss) compared to the random approach, but it also locates the causes of BASIR to help developers improve the design of the shutdown process for cloud-based applications during the testing of these applications.

In the third work, we develop a novel cloud market-based approach that leverages features of cloud spot markets for Provisioning Spot Instances WithOut employing Fault-Tolerance mechanisms

(P-SIWOFT) to reduce the overhead related to application completion time and deployment cost (i.e., DCATO) and, as a result, reduces the deployment cost and completion time of applications. We evaluate P-SIWOFT in simulations and use Amazon spot instances that contain jobs in Docker containers and realistic price traces from EC2 markets. Our simulation results show that our approach reduces the deployment cost and completion time compared to approaches based on fault-tolerance mechanisms.

CHAPTER 1

INTRODUCTION

1.1 Introduction

Cloud computing enables cloud customers to rent resources (e.g., CPU, memory,virtual machines (VMs)) on as-needed basis to run their applications [4]. That is, customers do not have to buy and host expensive hardware to run their applications, and instead they pay for renting resources for these applications from cloud computing facilities [5, 6]. This is a fundamental difference between cloud computing systems and distributed systems, which require application owners, i.e., cloud customers, to buy and host expensive hardware to run their applications. As the deployment cost is an integral part of applications deployed on the cloud, the cost-efficiency of provisioning resource to these applications becomes a priority, and it is of growing significance, since the total spending that will be affected by cloud computing is over $1 trillion by 2020 [7].

Three major problems may prevent cloud customers from economically deploying their applications in elastic clouds. First, 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. 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% [8–10]. There are many different rules like that for controlling cloud *elasticity*, a term that designates on-demand resource provisioning to an application [5, 11]. 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. As a result, resources that are provisioned to an application may not improve its performance; however, its owner (i.e., a cloud customer) still has to pay for these needlessly provisioned resources.

Second, although cloud spot instances in cloud computing allow stakeholders to economically deploy their applications at much lower costs than those of other types of cloud instances, spot instances are often exposed to revocations (i.e., terminations) by cloud providers. With spot instances becoming pervasive, terminations have become a part of the normal behavior of cloud-based applications; thus, these applications may be left in an incorrect state leading to certain bugs, such as data loss, inconsistent states, performance bottlenecks, hangs, crashes, deadlocks, locked resources, or these applications that cannot restart/terminate. On top of poor user experience from seeing these bugs, other bugs result in situations where cloud-based applications could not be restarted without manual interventions. Cloud-based applications that run in spot instances are not designed or tested to deal with this behavior in the cloud environment. The shutdown sequence of a cloud-based application is often left untested because developers often assume that a cloud-based application is properly terminated as long as its processes are terminated [12]. It is very difficult to find these bugs because a termination signal can be initiated at every execution state of a cloud-based application, leading to a significantly larger search space of application states [13]. Unfortunately, the absence of testing the effect of spot instance revocations on cloud-based applications will likely lead to a large number of these bugs. As a result, the advantages of economically deploying applications on cloud spot instances could be significantly minimized or even entirely negated [14].

Third, cloud computing offers a variable-cost payment scheme that allows cloud customers to specify the price they are willing to pay for renting spot instances to run their applications at much lower costs than fixed payment schemes, and depending on the varying demand from cloud customers, cloud platforms could revoke spot instances at any time. To alleviate the effect of spot instance revocations, applications often employ different fault-tolerance mechanisms to minimize or even eliminate the lost work for each spot instance revocation. However, these fault-tolerance mechanisms incur additional overhead related to application completion time and deployment cost. As a result, even though cloud customers sometimes rent spot instances at 90% lower prices than on-demand prices [15], their applications that run on spot instances can be terminated based on price fluctuations that happen frequently; thus, those applications may incur additional overhead related to application completion time and deployment cost from re-executing lost work for each spot instance revocation.

In summary, if many cloud-based applications are affected negatively by inefficient cloud elasticity, spot instance revocations, or the overhead of employing fault-tolerance mechanisms, the demand from cloud customers will eventually decrease, leading to a loss in both cloud providers'and cloud customers' revenues. Therefore, my thesis is dedicated to ensuring efficient cloud computing operations and services to enable cloud customers to deploy their applications in elastic clouds economically, resulting in greater economies of scale.

1.2 Research Contributions

The main contributions of this thesis are:

- We formulate challenging new problems that prevent cloud customers from deploying their applications in elastic clouds economically.
  - We investigate situations when customers pay for resources that are provisioned to, but not fully used by their applications, and as a result, some performance characteristics of these applications are not met, i.e., the Cost-Utility Violations of Elasticity (CUVE).
    - We develop a novel approach for Testing for Infractions of Cloud Elasticity (`TICLE`) that combines a search-based heuristic with rule-guided resource provisioning by stress testing the elastic resource provisioning for cloud-based applications to automatically discover irregular workloads that led to CUVE.
    - We evaluate `TICLE` using four nontrivial open-source applications in the Microsoft Azure cloud to determine how automatically and accurately `TICLE` explores 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.
  - We investigate situations when applications that run in spot instances are being irregularly terminated due to spot instance revocations. These applications might lose their states that lead to certain bugs, i.e., Bugs of cloud-based Applications resulting from Spot Instance Revocations (BASIR).

  * We implement a novel approach for Testing for Bugs of Cloud-Based Applications Resulting from Spot Instance Revocations (`T-BASIR`) that uses kernel modules to automatically find BASIR and locate their causes in the source code.
  * We evaluate `T-BASIR` using 10 popular open-source applications. Our results show that `T-BASIR` not only finds more instances and different types of BASIR (e.g., performance bottlenecks, data loss, locked resources, and applications that cannot restart) compared to the random approach, but it also locates the causes of BASIR to help developers improve the design of the shutdown process for cloud-based applications during the testing of these applications.

- We investigate situations when applications employ fault-tolerance mechanisms to minimize the lost work for each spot instance revocation. These applications incur additional overhead related to application completion time and deployment cost resulting from employing these fault-tolerance mechanisms, i.e., the Deployment Cost And Time Overhead (DCATO).

  * We develop a novel cloud market-based approach that leverages features of cloud spot markets for Provisioning Spot Instances WithOut employing Fault-Tolerance mechanisms (`P-SIWOFT`) to reduce the deployment cost and completion time of applications.
  * We evaluate `P-SIWOFT` in simulations and use Amazon spot instances that contain jobs in Docker containers and realistic price traces from EC2 markets. Our simulation results show that our approach reduces the deployment cost and completion time compared to approaches based on fault-tolerance mechanisms.

1.3 Thesis Statement

The thesis statement is formulated as follows.

With cloud-based applications becoming pervasive, the impact of inefficient cloud elasticity, spot instance revocations, and fault-tolerance mechanisms has become a very important concern for cloud customers. A solution based on the model that we proposed can be utilized to reduce or even eliminate the impact of CUVE, BASIR, and DCATO problems in the cloud environment to economically deploy applications in elastic clouds, and this model can lead to practical frameworks for optimizing cloud elasticity, improving the design of the shutdown process, and reducing the deployment cost and completion time for cloud-based applications. This ensures efficient cloud-computing services that lead to greater economies of scale.

1.4 Thesis Outline

The thesis is organized as follows: In chapter 2, we provide the necessary background on cloud instance types and highlight the rules that contribute to economically deploying applications in elastic clouds. Chapter 3 compares the related work with our work in this thesis. Chapter 4 presents TICLE, Testing for Infractions of Cloud Elasticity. Chapter 5 presents T-BASIR, Testing for Bugs of cloud-based Applications resulting from Spot Instance Revocations. Chapter 6 presents P-SIWOFT, Provisioning Spot Instances WithOut employing Fault-Tolerance mechanisms. Finally, we conclude this thesis and highlight future work in Chapter 7.

CHAPTER 2

BACKGROUND

This chapter presents some portions of the following papers.

- *Abdullah Alourani, Md Abu Naser Bikas, and Mark Grechanik. "Search-Based Stress Testing the Elastic Resource Provisioning for Cloud-Based Applications." In International Symposium on Search Based Software Engineering, pp. 149-165. Springer, Cham, 2018. [Online]. Available:* `https://doi.org/10.1007/978-3-319-99241-9_7`.
- *Abdullah Alourani, Ajay D. Kshemkalyani, and Mark Grechanik. "Testing for Bugs of Cloud-Based Applications Resulting from Spot Instance Revocations." In 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pp. 243-250. IEEE, 2019. [Online]. Available:* `https://doi.org/10.1109/CLOUD.2019.00050`. ***Best Student Paper Award.***
- *Abdullah Alourani, Ajay D. Kshemkalyani, and Mark Grechanik. "T-BASIR: Finding Shutdown Bugs for Cloud-Based Applications in Cloud Spot Markets." in IEEE Transactions on Parallel and Distributed Systems (TPDS), 2020. [Online]. Available:* `https://doi.org/10.1109/TPDS.2020.2980265`.

In this chapter, we provide the necessary background on cloud instance types and highlights the rules that contribute to economically deploying applications in elastic clouds.

2.1 Cloud Instance Types

We provide an overview of cloud instance types and discuss the challenges of finding optimal provision of different instance types.

2.1.1 Overview

Many cloud providers such as Amazon Web Services offer four types of instances (i.e., Servers) [16]: on-demand, reserved, dedicated, and spot (also known as preemptible [17]). Cloud customers can pay for renting on-demand instances per hour without long-term commitments and they cost the most. The reason for the highest cost is that cloud providers do not know the future demand for cloud resources from many cloud customers in advance, so it is difficult for cloud providers to make instances available without any prior notice from cloud customers. Also, cloud providers do not offer a discount for on-demand instances compared to other types of instances (e.g., reserved, spot), since cloud providers do not require long-term commitments from cloud customers to use on-demand instances, and cloud providers guarantee the availability of on-demand instances until they are released by their owners, i.e., cloud customers or simply customers. Cloud customers can rent reserved instances for a long term by making an upfront payment to cloud providers and thus pay a much lower rate than on-demand instances. For example, Amazon offers a three-year contract at a 75% discount relative to its on-demand prices. Cloud providers guarantee the availability of both reserved and on-demand instances. A variation of reserved instances is a dedicated host, which is a physical server that is assigned only to a specific cloud customer, and nobody besides this customer can use the resources of this host [18]. Dedicated hosts allow cloud customers to use their server-bound licenses (e.g., Windows Server) to reduce costs. Cloud customers can rent dedicated hosts per hour or for a long term. Therefore, although cloud providers guarantee the availability of on-demand, reserved, and dedicated instances until they are released by their owners, they result in higher deployment costs for owners.

Unlike the fixed-cost paying schemes mentioned above, a variable-cost paying scheme allows cloud customers to specify the price they are willing to pay for renting a spot instance to run their applications [15], and, depending on the varying demand from cloud customers, the price of this spot instance can go up if the demand increases and the number of available instances that can be supported by a finite number of physical resources in a data center of cloud providers decreases [19]. Conversely, the price of this spot instance can go down if the demand decreases and the number of available instances increases. Therefore, if the customer's price is greater than the cloud provider's price that depends on the demand, a spot instance will be provisioned to cloud customers' applications at the customer's price. However, when spot instances are already provisioned to cloud customer applications and the cloud provider's price goes above the customer's price, the cloud providers will terminate those spot instances within two minutes by sending termination notification signals [20]. As a result, even though cloud customers sometimes rent spot instances at 90% lower costs compared to on-demand [15], their applications that run on spot instances can be terminated based on price fluctuations that happen frequently, thus the services of those applications that run on spot instances will not be provided to their customers.

2.1.2 Challenges of Finding Optimal Provision of Different Instance Types

Cloud customers face a major challenge in choosing between different types of instances to run their applications. When cloud customers only use on-demand instances to run their applications, they will incur high deployment costs since on-demand instances cost the most. If cloud customers choose to run their applications using only reserved instances, they will save up to 75% of the deployment costs compared to on-demand prices, but they will need to know the demands of their customers in advance, which is often very difficult. When cloud customers only use spot instances to run their applications, they can save up to 90% of the deployment costs compared to on-demand prices. However, the availability of spot instances is not guaranteed since the cloud providers will terminate spot instances when the demand increases and the number of available instances decreases. That is, the services of cloud customers' applications that run on spot instances will not be provided to their customers. As a result, the fundamental problem for cloud customers is how to find an optimal provision of different types of instances for their applications that effectively balance the availability of services and the cost of deployment.

In addition, it is very difficult to determine an optimal allocation of different types of instances based on the application's needs. It requires application owners, i.e., cloud customers to understand what application components need to be run on instances that their availabilities are guaranteed, how the price of spot instances change at runtime, and how to make trade-offs between the cost of deployment and the availability of applications. Suppose that a web store application has multiple components (e.g., microservices) deployed on the cloud where each component has different impacts on quality of service requirements. For example, the payment processing component requires strict completion deadlines and has higher impacts on quality of service requirements compared to the shipping cost calculation component. Therefore, the major challenge for cloud customers is to determine how to allocate different types of instances to application components in order to reduce the deployment cost while maximizing the availability of components; thus, the services of those components will often be provided to applications' customers.

2.2 Elastic Rules

We provide an overview of elastic rules and discuss the challenges of creating optimal elastic rules.

2.2.1 Overview

In general, if-then elasticity rules contain antecedents that describe the level of utilization of some resources (e.g., CPU utilization >80%) and the consequents that specify (de)provisioning actions (e.g., (de)provision a VM) [11]. Unfortunately, rule creation is an error-prone manual activity, and provisioning certain resources using manually created rules may not improve the application's performance significantly. For example, when the CPU utilization reaches some threshold due to a lot of swapping or a lack of the storage, provisioning more CPUs does not fix the underlying cause that requires giving more storage to the application. That is, often rules are not optimal in terms of allocating required resources based on projected applications' needs [5].

2.2.2 Challenges of Creating Optimal Elastic Rules

It is very difficult to create rules that provision resources optimally to enhance the performance of the application while reducing 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. Doing so 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 [21,22], which result in tens of millions of combinations). Furthermore, suppose that the performance of an application falls below some desired level that is specified by the application's owners. Since there are multiple possible combinations of resources that could be allocated to the application, the challenge is to find the rules that provision only minimally needed resources to maintain the desired level of performance (i.e., the average response time). Conversely, provisioning resources that are not optimal often leads to a loss in customers' revenues.

In addition, it is very difficult to create rules that provision resources optimally to maximize the performance of a multi-tier web application (i.e., the user interface tier, the application server tier, and the database tier) while minimizing the cost of its deployment because the lack of resources that leads to degradation in its performance can occur at multiple tiers of this application. Therefore, applications' owners would need to analyze their applications to determine how to allocate resources (i.e., CPU, memory, VM) to different tiers of these applications in a way that maximizes their performance and reduces their deployment costs. For instance, the database tier (e.g., MySQL) is often I/O intensive whereas the application server (e.g., Tomcat) is rather CPU-intensive. Consider a typical scenario for a multi-tier web application where an application server interacts with multiple database servers, and the performance of this application drops at heavy loads (e.g., high CPU utilization). That is, allocating CPU instead of VM would not only improve the performance of this application, but also, reduce the cost of allocated resources. As a result, although some resources that are allocated to an application may improve its performance, they result in higher deployment costs for application's owners.

In summary, creating elastic rules that provision resources, based on applications' behaviors to optimize the performance of applications while minimizing the deployment cost, is an undecidable problem because it is impossible to determine in advance how an application will use available resources unless its executions are analyzed with all combinations of input values, which is often a huge effort [23]. Currently, many applications' owners manually determine the rules to (de)provision resources that approximate a very small subset of the application's behavior, and clouds often (de)provision resources inefficiently in general, thus resulting in major application service degradations and the loss of cloud customers' time and money, leading to inefficient cloud computing services that reduce the utility of cloud based applications.

2.3 Financial Rules

We provide an overview of financial rules and discuss the challenges of creating optimal financial rules.

2.3.1 Overview

If-then financial rules contain antecedents that specify the price condition of provisioning spot instances (e.g., the customer's price $>$the cloud provider's price) and the consequents that determine (de)provisioning actions (e.g., (de)provision a spot instance). It is very difficult for cloud customers to decide a price they are willing to pay for renting a spot instance to run their applications in such a way that reduces the deployment cost and the number of spot instance revocations [24, 25]. When spot instances are already provisioned to cloud customer applications and the customer's price is close to zero, there is a high probability that those spot instances will be revoked by cloud providers. Also, when a cloud customer requests spot instances and the customer's price is close to zero, there is a very low probability that those spot instances will be provisioned to cloud customer applications. Conversely, if cloud customers set their prices close to on-demand instances' prices, cloud customers may reduce the number of revocations of spot instances that are provisioned to their applications, but cloud customers may not benefit from a significant discount of spot instances that is up to 90% compared to on-demand instances [15]. As a result, without knowing a demand from different cloud customers in advance, the challenge for cloud customers is to choose a price of spot instances that is both significantly lower than the price of on-demand instances and greater than the cloud provider's price to minimize the cost of the deployment and the number of spot instance revocations.

2.3.2 Challenges of Creating Optimal Financial Rules

It is very difficult to create financial rules that provision spot instances optimally to reduce the number of spot instance revocations and the cost of the deployment since the revocations of spot instances are based on price fluctuations that happen based on demand of spot instances from many cloud customers. The cloud providers often revoke spot instances when the demand increases and the number of available spot instances that can be supported by a finite number of physical resources in a data center of cloud providers decreases. It is very difficult to determine in advance spot instance revocations that depend on the varying demands of cloud customers [26]. Doing so requires cloud customers (i.e., application's owners) to understand how the demands of the spot instances change, how the costs of the allocated spot instances change, and how to make trade-offs between the demands and these costs [1]. As a result, price fluctuations that depend on the demand have a high influence on the number of spot instance revocations.

2.4 Consistency Rules

We provide an overview of consistency rules and describe the interactions between consistency and financial rules.

2.4.1 Overview

Replication is the process of copying and distributing data objects of cloud applications from one instance to other instances (i.e., replicas) in distributed systems deployed on the cloud, and then synchronizing these data objects among these instances to maintain the consistency of these distributed systems. Consistency rules specify when and how this synchronization occurs to ensure that any new updates made to any data object of cloud applications will be visible in all replicas in distributed systems deployed on the cloud. The CAP theorem [27] states that in the presence of partitions (i.e., the network connection is broken), we cannot have both availability and strong consistency, i.e., any new updates made to any data object will instantaneously be visible in all replicas. To preserve availability guarantees, consistency rules can be defined based on eventual consistency instead of strong consistency, which guarantees that all replicas of a data object will eventually converge if no new updates are submitted to this data object for some time. A fundamental model of cloud environment that is used in the CAP theorem is shown in Figure 1 that contains a data object D and its replica D_R, a writer who writes and updates the data object, and a reader who reads from the replica. The data object D and its replica D_R are synchronized over a network to exchange messages that designate the states of these data objects. However, when there is a partition in the network, the data object D and its replica D_R cannot be synchronized because these messages will be delayed or lost.

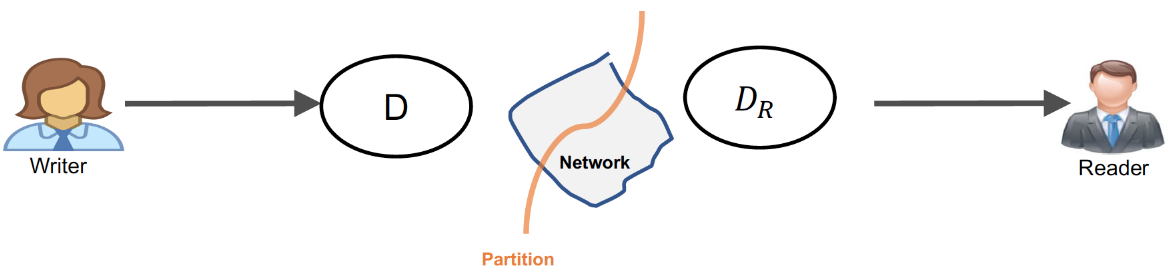


Figure 1: A fundamental model of cloud environment.

2.4.2 Interactions between Consistency and Financial Rules

We enhance the model that is shown in Figure 1 with a special type of a message (i.e., termination message) that disconnects this distributed data object from the other data objects in the model of the cloud environment, and this message models the terminations of spot instances to demonstrate when consistency rules trigger. Consider a scenario where a writer (i.e., Alice) sends a message to update the data object D, and then a termination message is sent to the data object D. As a result, this data object D is disconnected before it synchronizes with its replica D_R, and while the data object D is disconnected, another writer (i.e., Bob) sends a message to update the replica D_R. Suppose that the data object D is reconnected again. If the old update by Alice is sent to the replica D_R, this distributed data object will be in an inconsistent state because it will overwrite the latest update by Bob that happens when the data object D was disconnected. Then, consistency rules will be triggered to resolve these conflicts in order to ensure consistency of this distributed data object. As a result, when spot instances are frequently terminated based on price fluctuations that depend on the varying demand from cloud customers, more consistency rules will be triggered that consume more resources to resolve conflict states of data objects in a cloud-based application and achieve consistency, resulting in overloading of resources.

CHAPTER 3

RELATED WORK

This chapter presents some portions of the following papers.

- *Abdullah Alourani, Md Abu Naser Bikas, and Mark Grechanik. "Search-Based Stress Testing the Elastic Resource Provisioning for Cloud-Based Applications." In International Symposium on Search Based Software Engineering, pp. 149-165. Springer, Cham, 2018. [Online]. Available:* `https://doi.org/10.1007/978-3-319-99241-9_7`*.*
- *Abdullah Alourani, Ajay D. Kshemkalyani, and Mark Grechanik. "Testing for Bugs of Cloud-Based Applications Resulting from Spot Instance Revocations." In 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pp. 243-250. IEEE, 2019. [Online]. Available:* `https://doi.org/10.1109/CLOUD.2019.00050`*.* ***Best Student Paper Award.***
- *Abdullah Alourani, Ajay D. Kshemkalyani, and Mark Grechanik. "T-BASIR: Finding Shutdown Bugs for Cloud-Based Applications in Cloud Spot Markets." in IEEE Transactions on Parallel and Distributed Systems (TPDS), 2020. [Online]. Available:* `https://doi.org/10.1109/TPDS.2020.2980265`*.*

In this chapter, we discuss the related work concerning elasticity rules, genetic algorithms, performance testing, spot instance revocations, application bugs, and fault-folerance mechanisms.

3.1 Elasticity Rules

Elasticity rules are a key element in the reactive provisioning technique [28, 29], which is the most commonly used and offered by popular cloud service providers [30–32]. Gambi et al. developed a tool that uses predefined workloads to test the automation of cloud-based elastic systems [33]. Breitgand *et al.* designed an algorithm based on the logistic regression model that redefines threshold values when a violation of performance parameters occurs to improve the elasticity property of the cloud [34]. However, testing the elasticity rules has not yet been investigated. Islam *et al.* first observed a situation when provisioning resources do not alleviate the service level agreement (SLA) violations [35]. Testing for Infractions of Cloud Elasticity (`TICLE`) is the first approach that obtains workloads that lead to Cost-Utility Violations of Elasticity (CUVE).

3.2 Genetic Algorithms

Genetic Algorithms (GAs) are extensively used in many areas of software engineering [36], such as software maintenance [37–39], cloud computing [40], regression testing [41], quality assurance [42], mutation testing [43, 44], textual analysis [45], test generation [46–52], stress testing [53], coverage testing [54], fault detection [55], and performance testing [56, 57]. Although these genetic algorithms are used in many areas of software engineering, they have not been applied to our problem, since it required multiobjective optimization.

Only a few works have been conducted on applying multiobjective optimization in software engineering [58–60]. Mondal et al. designed an approach for enhancing fault detection, which prioritizes the selection of test cases by maximizing test case diversity and code coverage based on a multiobjective optimization algorithm [61]. Linares-Vasquez et al. designed a multiobjective approach that generates color compositions for Android app GUIs to improve energy consumption [62]. Almhana et al. proposed an approach that locates potential relevant classes for bug reports by applying a multiobjective optimization algorithm [63]. `TICLE` uses a multiobjective algorithm with rule-guided provisioning of resources to determine irregular workloads that lead to CUVEs.

3.3 Performance Testing

One of the critical goals of performance testing is to automatically generate test cases that may trigger performance problems [64]. Several papers focused on generating test cases to find performance problems [65–69]. Burnim *et al.* presented an approach for the symbolic test generation tool to find inputs that lead to performance bottlenecks [70]. Bodik et al. proposed a workload model that characterizes volume and data spikes to test the robustness of stateful systems [71]. Chen et al. developed a tool that uses user-defined workloads to analyze performance and energy consumption for cloud applications [72]. 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 [73]. Shen *et al.* presented an approach that uses genetic algorithms to find the combinations of inputs that lead to performance problems [74]. Xiao *et al.* presented an approach that uses complexity models to predict workload-dependent performance bottlenecks [75]. 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.

3.4 Spot Instance Revocations

To the best of our knowledge, Testing for Bugs of Cloud-Based Applications Resulting from Spot Instance Revocations (`T-BASIR`) is the first automated solution for testing the effect of spot instance revocations on cloud-based applications. Most of the prior works focused on reducing the effect of

TABLE I: Comparison of `T-BASIR` with the related work concerning spot instance revocations and application bugs. The top part of Table (a) indicates existing works that aim to mitigate spot instance revocations. The following row designates the methodology of the proposed solution, followed by a row that designates specific methods. The bottom part of Table (b) indicates existing works that aim to find application bugs. The next row designates the methodology of the proposed solution and the cells contain the name of the proposed solutions.

(a) Spot Instance Revocations

Modeling Spot Markets		**Employing Fault-tolerance Mechanisms**			**Testing Impact on Applications**
Bidding Strategies	Prediction Schemes	Replication	Checkpointing	Migration	
PADB [76]	DrAFTS [77]	Multifaceted Policy [78]	Spoton [79]	Smart Spot Instances [80]	
DBA [81]	Calibration [82]	Proteus [83]	Checkpointing Schemes [84]	Hotspot [85]	`T-BASIR`
AMAZING [86]	Quantitative Models [87]	Spotcheck [88]	ExoSphere [89]		

(b) Application Bugs

Buggy Templates	**Rules and Specifications**	**Historical Bugs**	**RAT**
Metal Checkers [90]	Alattin [91]	HCM [92]	
PMD [93]	Pr-miner [94]	FixCache [95]	`T-BASIR`
FindBugs [96]	AFG [97]	Bug Prediction [98]	

spot instance revocations using fault-tolerance methods, such as replication [78, 83, 88, 88, 89, 99, 100], checkpointing [79, 84, 89, 101], and VM migration [80, 85]. Voorsluys et al. [78] proposed a fault-aware resource allocation approach that applies the price of spot instances, runtime estimation of applications, and task duplication mechanisms to economically run batch jobs in spot instances. Yi et al. [84] proposed checkpointing schemes to reduce the computation price of spot instances and the completion time of tasks. Shastri et al. [85] proposed a resource container that enables applications to self-migrate to new spot VMs in a way that optimizes cost-efficiency as the spot prices change.

In addition, other researchers worked on modeling spot markets to reduce the spot instance cost and the performance penalty that results from a high number of revocations, by designing optimal bidding strategies [76, 81, 86, 102–108] and developing prediction schemes [77, 82, 87, 109]. Song et al. [76]

proposed an adaptive bidding approach that leverages the spot price history information to choose the bid strategy that increases the profit for brokers of the cloud service. Javadi et al. [82] proposed a statistical approach to analyze changes in spot price variations and the time between price variations to explore characterization of spot instances that are required to design fault-tolerant algorithms for applications deployed on cloud spot instances.

3.5 Shutdown Bugs of Applications

`T-BASIR` is the first automated solution to identify instances of Bugs of cloud-based Applications resulting from Spot Instance Revocations (BASIR). `T-BASIR` measures the impact on the state of Resources Affected by Termination (RAT) when the application is irregularly terminated to identify BASIR, as discussed in Section 5.4.3. Existing bug finding tools are not applicable to BASIR because they rely on searching through the application's execution paths for certain inputs to check if the state value of an application varies from the expected value that represents the input value of the next instruction in this execution path [13]. However, a termination signal can be initiated at every execution state of applications, leading to a significantly larger search space of these states. Prior works required users to provide the buggy templates in order to find application bugs [90, 93, 96, 110], whereas other works automatically inferred rules and specifications by mining existing applications in order to find application bugs [91, 94, 97, 111]. Kermenek et al. [97] proposed a probabilistic approach that automatically infers specifications from a source code of an application and uses them to detect incorrect and missing properties in specifications. Other researchers focused on predicting application bugs using historical data of reported bugs [92, 95, 98, 112]. Giger et al. [98] proposed a bug prediction approach that learns from source code and change metrics to predict application bugs.

In summary, Table I briefly gives a comparison of `T-BASIR` from different existing works that aim to mitigate spot instance revocations and find application bugs. While many of the prior works focused on reducing the effect of spot instance revocations by modeling spot markets and using fault-tolerance methods, these works are subject to altering pricing algorithms and are exposed to incurring overhead related to application completion time and deployment cost, respectively. In contrast, `T-BASIR` focuses on testing the effect of spot instance revocations on cloud-based applications. Also, although the other prior works focused on finding application bugs using buggy templates, rules and specifications, and historical bugs, these works are subject to limited inputs. However, `T-BASIR` measures the impact on the state of RAT when the application is irregularly terminated to identify BASIR, as discussed in Section 5.4.3. As a result, `T-BASIR` is the first tool that sheds light on the effect of spot instance revocations on cloud-based applications.

3.6 Fault-Tolerance Mechanisms

To the best of our knowledge, Provisioning Spot Instances WithOut employing Fault-Tolerance mechanisms (`P-SIWOFT`) is the first approach that leverages cloud spot market's features to provision spot instances without employing fault-tolerance mechanisms to reduce the deployment cost and completion time of applications. Most of the prior works focused on reducing the effect of spot instance revocations using fault-tolerance methods, such as replication [78, 83, 88, 88, 89, 99, 100], checkpointing [79, 84, 89, 101], and VM migration [80, 85]. Voorsluys et al. [78] proposed a fault-aware resource allocation approach that applies the price of spot instances, runtime estimation of applications, and task duplication mechanisms to economically run batch jobs in spot instances. Yi et al. [84] proposed checkpointing schemes to reduce the computation price of spot instances and the completion time of tasks.

Shastri et al. [85] proposed a resource container that enables applications to self-migrate to new spot VMs in a way that optimizes cost-efficiency as the spot prices change.

CHAPTER 4

TESTING FOR INFRACTIONS OF CLOUD ELASTICITY (TICLE)

This chapter presents a published paper, Abdullah Alourani, Md Abu Naser Bikas, and Mark Grechanik. "Search-Based Stress Testing the Elastic Resource Provisioning for Cloud-Based Applications." In International Symposium on Search Based Software Engineering, pp. 149-165. Springer, Cham, 2018. [Online]. Available: `https://doi.org/10.1007/978-3-319-99241-9_7`.

In this chapter, we propose a novel approach for Testing for Infractions of Cloud Elasticity (TICLE) that combines a search-based heuristic with rule-guided resource provisioning by stress testing the elastic resource provisioning for cloud-based applications to automatically discover irregular workloads that led to CUVE.

4.1 Overview

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.

4.2 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 [4]. That is, instead of buying and hosting expensive hardware, customers pay for renting resources for running these applications from cloud computing facilities [6]. 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 [7].

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% [8–10]. There are many different rules like that for controlling cloud *elasticity*, a term that designates on-demand resource provisioning to an application [5, 11]. 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 work, 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 [113], 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 [114]. Unfortunately, when irregular workloads happen, customers pay for resources that are provisioned to, but not fully used by their applications [35], and the benefits of cloud computing may be significantly reduced or even completely obliterated [115].

Contributions: 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 combines 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.

4.3 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.

4.3.1 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 [35].

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 [116]. 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 [10], which result in tens of millions of combinations).

Definition 4.3.1. *An application workload is a time-dependent collection of request tuples as shown in Figure 3 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 [117]. 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 [118]. By changing the coefficients of the function, irregular workloads can be generated for stress testing in varying degrees of burstiness and fluctuation.

4.3.2 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 [35]. Recall that cloud providers recommend some generic rules for resource provisioning [8–10]. 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 [119]. 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 [35]. 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.

4.3.3 An Illustrative Example

The CUVE problem with a cloud-deployed application is illustrated in Figure 2. The operations of the *genetic algorithm (GA)* will be discussed in Section 4.4 as a part of our solution, and they can

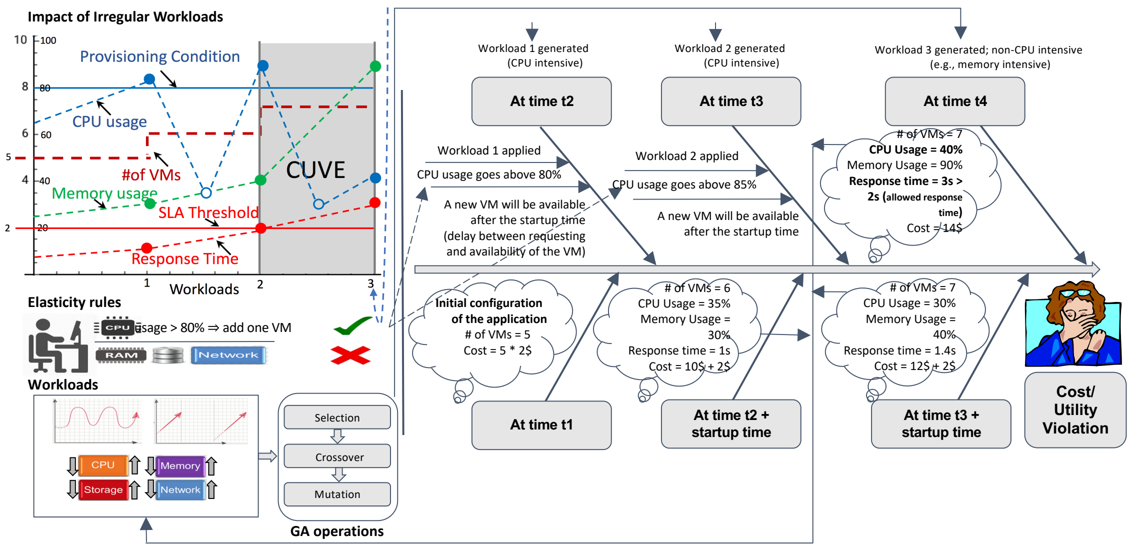


Figure 2: An illustrative example of the CUVE for a cloud-based application. The timeline of the operations is shown with the horizontal block arrow in the middle. The process starts with the customer who defines elasticity rules on the left and the events are shown in the fishbone presentation sequence that lead to the CUVE on the right.

be ignored for now. On the left side, the input is a set of rules for elastic resource provisioning and workloads for the application. The top leftmost embedded graph that is shown in Figure 2 summarizes how workloads' fluctuations and burstiness reduce the effectiveness of the elasticity rules. The horizontal axis shows numbered workloads, and the inner measurements on the vertical axis indicate the utilization of CPU and memory in percents. The solid blue line shows the provisioning condition that describes the level of CPU utilization. The outer measurements on the vertical axis indicate the number of the provisioned VMs and the response time in seconds, and the solid red line shows the threshold of a *service level agreement (SLA) that indicates a desired performance level (i.e., the response time).*

We show that a rapid change from the workload `2` to the workload `3` results in a situation where the cloud allocates resources according to the rule based on workload `2`, whereas different resources are needed to maintain a desired level of performance for workload `3`, a short moment after the provisioning is made for workload `2`. Finding such irregular workloads that lead to the CUVE is very important during stress testing, where the SLA is violated and the cost of deployment is high because of the provisioned resources. The cost and the performance move in opposite directions. Once known, these irregular workloads and rules can be reviewed by developers and performance engineers, who optimize the rules to achieve a better performance of the corresponding application. We show how the interactions between workloads and rules lead to the CUVE problem.

Consider what happens in the illustrative example with the commonly recommended rule that specifies that the cloud infrastructure should allocate one more VM if the utilization of the CPUs in already provisioned VMs exceeds 80%. As an example, we choose the initial configuration of five VMs at the cost of \$2 at the time t_1. We rounded off the cost for the ease of calculations and based it on the pricing of various cloud computing platforms [8–10]. Then, a CPU-intensive workload triggers the rule at the time t_2. A new VM will be provisioned after some startup time while the owner of this application is charged an additional \$2 at the time t_2. The VM will become available to the application at $t_2 + t_{VM_s}$, where t_{VM_s} is the VM startup time. Suppose that allocating one more VM in this example decreases the CPU utilization to 35% whereas the memory utilization remains the same at 30%. The new workload 2 leads to a significantly increased CPU utilization, and another VM is allocated at the time t_3. This is in a nutshell how an elastic cloud works.

Suppose that the response time for the application should be kept under two seconds according to the SLA that is specified by the applications' owners, and a goal of the elastic rules is to provision resources to the application to maintain the SLA. The SLA is maintained below the threshold until the time t_4 when the workload rapidly changes. The new workload 3 leads to a significant burst in the memory usage whereas the utilization of the CPUs in already provisioned VMs remains low at 40%. The memory utilization increases to 90%, and there is no rule that can be triggered in response, thus, subsequently, there is no action taken by the cloud to alleviate this problem. The CPUs wait for data to be swapped in and out of memory, and they spend less time executing the instructions of the application. As a result, the application's response time increases, thus eventually breaking the SLA threshold. Furthermore, at the 40% higher cost, the SLA is violated and the performance of the application worsened, while the application's owner pays for resources that are under-utilized or completely unused.

4.3.4 The Problem Statement

Software engineers make performance enhancements routinely during perfective maintenance when they use mostly exploratory random performance testing to identify when the performance of the *Ap-*

plication Under Test (AUT) worsens. In this work, 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 the SLA 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.

4.4 Our Approach

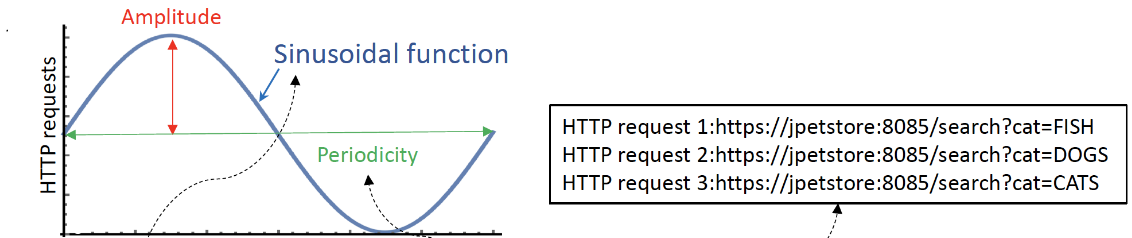
In this section, we state our key ideas for our approach for *Testing for Infractions of CLoud Elasticity (TICLE)*, explain *GA* with MOOP (*GAMOOP*), and describe the algorithm for TICLE.

4.4.1 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. 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 3. 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 [120] 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. Hence, existing test input data generation techniques are not applicable to TICLE. For example, model-based fuzzing or monkey testing require a complete model of software, which is often unavailable.

Using GAs for finding the CUVEs is illustrated in Figure 2 with the label `GA operations`. In GAs, new solutions, or *offsprings* are generated using existing solutions, or *parents*. New solutions



$$[\underbrace{\{\text{function of time}\}}_{\textbf{Part 1: }\text{Type of periodic circular functions}}, \underbrace{\{parm_1, \ldots, parm_n\}}_{\textbf{Part 2: }\text{Function's parameters}}, \underbrace{\{HTTP\ request_{ID\#1}, \ldots, HTTP\ request_{ID\#m}\}}_{\textbf{Part 3: }\text{Set of HTTP requests}}]$$

Figure 3: The representation of the workload and the chromosome.

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. 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 [121]. 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.

4.4.2 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 4.2) [122] to evaluate the objectives and to select workload solutions using *the quality indicator* described by (Equation 4.1) [122] 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.

$$\begin{gathered} I(S,S') = \max\left\{\forall w' \in S' \, \exists w \in S : g_j(w) \geq g_j(w') \quad \text{for} \quad j \in \{1,\ldots,n\}\right\}, \\ S,S' \in \Omega, \quad w,w' \in P \end{gathered} \tag{4.1}$$

$$F(w) = \Sigma_{w' \in P\setminus\{w\}} - e^{-I(\{w'\},\{w\})/k}, \quad k > 0 \tag{4.2}$$

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 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 [123].

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 4.2) that maps workloads to the

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

1: **`Inputs:`** GAMOOP Configuration Ω, 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** $\mathcal{P}$

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 4.1), 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. Determining if such an irregular workload is realistic is a task for subject-matter experts, and its investigation is beyond the scope of this work. 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 [117].

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, and the crossover operator randomly selects a crossover point and exchanges the remaining genes for selected parent individuals, thus creating two new offspring individuals for a new generation.

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.

4.5 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?

We introduce the null hypothesis H_0 and an alternative hypothesis H_A to evaluate the statistical significance of the difference in the median value of the dependent variables:

H_0: There is no statistical difference in the median values of the dependent variables triggered by workloads generated randomly and by TICLE.

H_A: There is a statistically significant difference in the median values of the dependent variables triggered by workloads generated randomly and by TICLE.

TABLE II: 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	$v4.0.5$	2,762	42	400	10^{31}
JForum	$v2.1.9$	36,401	397	3,487	10^{49}
PhotoV	$v2.1.0$	10,549	81	931	10^{36}
RUBiS	$v1.4.3$	83,640	641	4,396	10^{14}

4.5.1 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 II. These applications are written by different programmers, come from different domains, and have high popularity indexes. JPetStore is a PetStore application that is widely used as a performance benchmark. JForum is a discussion board forum software. PhotoV is a photo database system that allows users to catalogue, sort, and display photos. RUBiS is an online auction system that is written in Java and PHP. 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.

TABLE III: 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	$CPU_{utilization} < 20\%$	$CPU_{utilization} > 80\%$

4.5.2 Methodology

We use the definition a *workload* from Section 4.3.1 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 [124] 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.

Also, we defined three elasticity rules with different ranges for VM (de)provisioning that are shown in Table III to determine how effectively TICLE finds irregular workloads that infract these elasticity rules for the AUTs. For example, the rule R_3 gives us a wider range of the CPU utilization (i.e., 60%) than the rule R_2 (i.e., 20%), thus allowing us to control how easy it is to find a workload that triggers provisioning of the VMs (or scaling out). Respectively, it is easier to trigger the rule R_2 than the rule R_1 to deprovision VMs (or scale in). Evaluating TICLE with these different rules is one of the goals of this work to answer $\boldsymbol{RQ}_3$.

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. That is, if we use 100% of the longest response time as the SLA threshold, there will be very few observed CUVEs, if any, since the response time for all experiments will be less than or equal to the SLA threshold. Conversely, setting the SLA threshold at 20% of the longest response time will likely make finding CUVEs easier. Experimenting with different SLA thresholds in the controlled environment enables us to answer $\boldsymbol{RQ}_1$.

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 [124] 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 [125]. 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.

4.5.3 Variables

Independent variables include the SLA violation threshold, i.e., the AUT's response time, the set of HTTP requests, the costs of the cloud virtual resources, the functions that describe the burstiness and the fluctuations of the workloads, the subject applications, and the set of user-defined elasticity rules that are illustrated in Table III. Dependent variables include the cost and the utilization level of resources provisioned to the AUTs, the average response time of the AUT, the average execution time to find irregular workloads that led to the first obtained CUVE, and the total count of the detected CUVEs.

4.5.4 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 [74]. 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 work and will be addressed in future work.

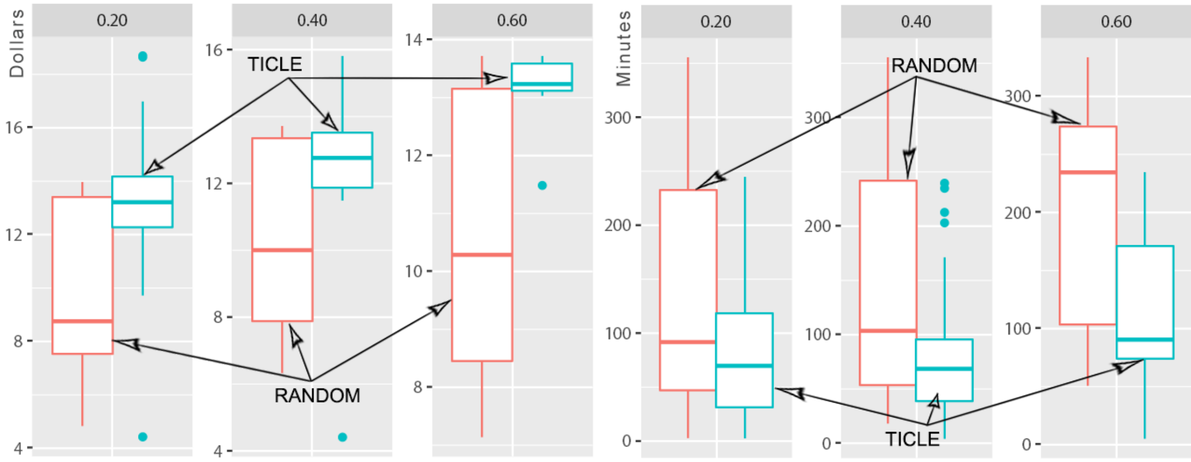
Our experiments were performed only on the *Infrastructure as a Service (IaaS)* cloud model, whereas applications may be deployed on *Platform as a Service (PaaS)* or *Software as a Service (SaaS)* cloud models. Even thought it is a potential threat to validity, TICLE is perfectly applicable for the PaaS model as long as the PaaS supports auto-scaling features and provides access to resource utilization such as App Service Plan in Azure.

4.6 Empirical Results

In this section, we describe and analyze the results of the experiments to answer the three RQs stated in Section 4.5.

4.6.1 Finding Workloads that Lead to Higher Costs

The results of the experiments are shown in the box-and-whisker plots in Figure 4a and Figure 4b 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



(a) (b)

Figure 4: 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.

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 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 4b that `TICLE` is consistently faster than `RANDOM` in finding the first CUVE. This result is important not only to answer $\boldsymbol{RQ_2}$, but also to show that `TICLE` is efficient in practice, since taking less time to find the first CUVE shows that `TICLE`

TABLE IV: 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

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 IV. 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**.

4.6.2 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 IV. 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.

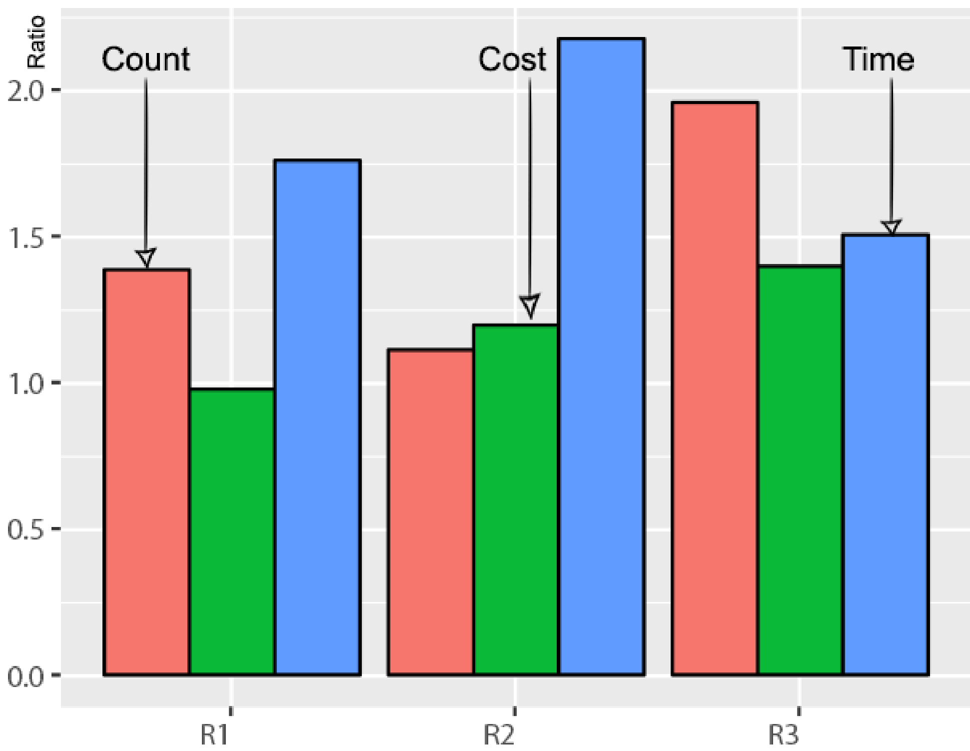


Figure 5: Comparing `TICLE` and `RANDOM` for detecting CUVEs for the subject AUTs with different elastic rules that are shown in Table III. The X-axis designates elasticity rules. The leftmost red bar represents the ratio of the total number of detected CUVEs using the approaches `TICLE` and `RANDOM`, $\frac{\text{count}_{\text{TICLE}}}{\text{count}_{\text{RANDOM}}}$. The middle green bar represents the ratio of the average costs for CUVEs, $\frac{\text{cost}_{\text{TICLE}}}{\text{cost}_{\text{RANDOM}}}$. The righmost blue bar represents the ratio of detection times for the first found CUVE, $\frac{\text{time}_{\text{RANDOM}}}{\text{time}_{\text{TICLE}}}$.

4.6.3 The Impact of the SLA Threshold

An interesting question is how an SLA threshold affects the process of finding CUVEs. As discussed in Section 4.5.2, 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 4a and Figure 4b – 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.

4.6.4 Impact of Different Elasticity Rules

The results of the experiments to answer $\boldsymbol{RQ}_3$ are presented in the histogram plot in Figure 5 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 [8–10], and these rules are shown in Table III. 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** $\boldsymbol{RQ}_3$.

4.6.5 Impact of Different Workload Types

Further details about the results of the `TICLE` and `RANDOM` comparison are shown in Table V, where statistical information is provided on the deployment costs and the time it takes to find the first CUVE in the context of four workload types using the `TICLE` and `RANDOM` approaches. We observed that the median cost of the found CUVEs using `TICLE` is consistently higher than the median cost of the CUVEs found by `RANDOM` for all workload types. Similarly, the median detection time using `TICLE` is consistently shorter than `RANDOM` in finding the first CUVE for all workload types. This result suggests that `TICLE` is more effective and efficient in finding CUVEs for all workload types when compared to

TABLE V: Results of experiments on subject AUTs using the SLA threshold at 15% of the longest response time. The first column represents the subject AUTs, and the second column represents the circular functions for workloads. The third column represents the approach `TICLE` and `RANDOM` followed by deployment cost values, and the detection time. We report the Min, Max, Mean, Median, and the standard deviation. We observe that the effectiveness of `TICLE` is higher in finding CUVEs.

App	Workload	Method	Deployment cost, $					Detection Time, mins				
			Min	Max	Mean	Med	SD	Min	Max	Mean	Med	SD
JForum	CIRCULAR	TICLE	6.217	11.400	9.722	9.883	0.995	15.828	98.028	44.403	41.871	18.320
		RANDOM	2.100	12.050	8.007	7.717	2.583	17.244	138.000	78.281	77.730	35.122
	LINEAR	TICLE	5.217	11.450	9.698	9.963	1.281	15.810	134.658	45.273	37.917	26.009
		RANDOM	1.200	12.200	7.603	7.600	2.459	16.230	140.130	76.308	74.466	35.150
	EXPON	TICLE	6.325	11.463	9.256	**9.388**	1.220	16.362	127.332	45.380	**38.184**	26.268
		RANDOM	1.600	12.300	7.225	**6.900**	2.930	16.122	144.000	74.507	**78.375**	34.279
	RANDOM	TICLE	7.350	12.275	9.771	10.000	1.089	17.442	144.426	49.581	46.833	26.559
		RANDOM	1.200	12.325	7.538	7.200	2.957	18.546	142.710	75.800	78.828	34.948
RUBiS	CIRCULAR	TICLE	8.975	10.575	10.047	10.356	0.596	19.068	97.836	46.605	40.476	24.601
		RANDOM	6.975	10.825	9.501	9.856	1.108	20.088	153.744	80.982	81.420	41.857
	LINEAR	TICLE	4.788	11.425	9.869	10.231	0.989	16.824	86.628	52.091	50.598	24.260
		RANDOM	6.100	11.475	9.335	9.471	1.406	16.476	147.240	77.943	80.544	37.398
	EXPON	TICLE	8.350	10.900	9.874	10.009	0.798	16.524	95.028	51.830	48.831	24.507
		RANDOM	5.600	10.544	9.031	9.663	1.540	16.164	154.392	86.550	82.614	52.569
	RANDOM	TICLE	8.850	10.817	9.954	10.263	0.638	25.848	133.668	68.843	64.764	36.083
		RANDOM	6.350	10.450	8.818	9.075	1.428	16.272	146.364	73.317	72.000	37.806
JPetstore	CIRCULAR	TICLE	6.779	12.400	10.126	10.611	1.501	16.149	51.579	32.433	29.390	11.027
		RANDOM	3.225	12.425	8.343	8.350	2.228	15.587	111.615	53.574	50.783	25.838
	LINEAR	TICLE	6.475	12.606	9.257	9.121	1.584	15.701	60.958	33.386	34.537	12.006
		RANDOM	1.600	12.583	7.764	7.663	2.220	15.887	106.530	55.631	55.680	22.255
	EXPON	TICLE	7.100	12.483	9.236	9.075	1.614	15.878	66.586	33.232	29.933	13.384
		RANDOM	4.683	11.494	8.375	8.340	2.190	16.773	92.292	46.916	47.219	25.568
	RANDOM	TICLE	6.267	11.533	9.578	9.791	1.343	18.510	60.020	34.576	31.086	12.919
		RANDOM	5.350	12.367	8.232	7.850	1.776	17.664	112.632	57.449	56.055	26.521
PhotoV	CIRCULAR	TICLE	7.492	11.425	9.833	9.950	0.733	17.592	74.616	34.468	34.236	13.935
		RANDOM	7.300	11.150	9.565	9.388	1.193	15.786	116.460	66.627	62.304	31.797
	LINEAR	TICLE	6.575	10.200	9.629	9.825	0.724	15.972	62.628	39.351	39.546	13.389
		RANDOM	8.325	11.588	9.822	9.575	0.933	23.646	119.880	68.300	71.880	27.909
	EXPON	TICLE	6.925	11.300	9.595	*9.763*	0.967	15.774	79.800	37.567	*35.937*	14.896
		RANDOM	7.700	11.800	9.635	*9.450*	0.977	15.846	99.972	49.992	*44.004*	31.022
	RANDOM	TICLE	7.388	10.200	9.427	9.669	0.743	15.732	64.248	33.467	31.476	12.326
		RANDOM	7.825	11.525	9.827	9.638	0.846	18.420	127.860	64.753	61.494	37.469

RANDOM. The standard deviation of cost and detection time for CUVEs computed by TICLE is lower when compared to the RANDOM approach for all workload types. This result suggests that TICLE favors workloads that have the highest impact on the cost of deployment.

We show in bold the median values for workload types that give the highest differences between the TICLE and RANDOM. The median costs and detection times for CUVEs have the highest difference between TICLE and RANDOM for JForum when employing exponential workloads. Since JForum has the largest search space of the input HTTP requests as shown in Table II, it is likely that randomly selecting workloads from a very large space of input requests results in many misses and TICLE zeros in on the CUVE-revealing workloads much faster in such large input spaces. Interestingly, the median costs and detection time for CUVEs have the smallest difference between TICLE and RANDOM for PhotoV, the subject AUT with the smallest number of the input HTTP requests, thus confirming our theory that, if the number of combinations of inputs for one AUT is larger than the number of combinations of inputs for some other AUT, then the effectiveness of TICLE for the former is higher than for the latter AUT.

4.7 Summary

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 TICLEd 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.

CHAPTER 5

TESTING FOR BUGS OF CLOUD APPLICATIONS (T-BASIR)

This chapter presents the following papers:

- *Abdullah Alourani, Ajay D. Kshemkalyani, and Mark Grechanik. "Testing for Bugs of Cloud-Based Applications Resulting from Spot Instance Revocations." In 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pp. 243-250. IEEE, 2019. [Online]. Available:* `https://doi.org/10.1109/CLOUD.2019.00050`. ***Best Student Paper Award.***
- *Abdullah Alourani, Ajay D. Kshemkalyani, and Mark Grechanik. "T-BASIR: Finding Shutdown Bugs for Cloud-Based Applications in Cloud Spot Markets." in IEEE Transactions on Parallel and Distributed Systems (TPDS), 2020. [Online]. Available:* `https://doi.org/10.1109/TPDS.2020.2980265`.

In this chapter, we propose a novel approach for Testing for Bugs of Cloud-Based Applications Resulting from Spot Instance Revocations (T-BASIR) that uses kernel modules to automatically find BASIR and locate their causes in the source code.

5.1 Overview

One of the major advantages of cloud spot instances in cloud computing is to allow stakeholders to economically deploy their applications at much lower costs than that of other types of cloud instances. In exchange, spot instances are often exposed to revocations (i.e., terminations) by cloud providers. With spot instances becoming pervasive, terminations have become a part of the normal behavior of cloud-based applications; thus, these applications may be left in an incorrect state leading to certain bugs. Unfortunately, these applications are not designed or tested to deal with this behavior in the cloud environment, and as a result, the advantages of cloud spot instances could be significantly minimized or even entirely negated. We propose a novel solution to automatically find these bugs and locate their causes in the source code. We evaluate our solution using 10 popular open-source applications. The results show that our solution not only finds more instances and different types of these bugs compared to the random approach, but it also locates the causes of these bugs to help developers improve the design of the shutdown process and is more efficient in finding instances of these bugs since it interposes at the system call layer.

5.2 Introduction

Cloud computing enables cloud customers to rent resources (e.g., virtual machines (VMs)) on as-needed basis to run their applications. That is, cloud customers do not have to buy and host expensive hardware to run their applications, and instead they rent resources for their applications from cloud computing facilities. This is an essential difference between cloud computing systems and distributed systems, which require application owners to buy and host expensive hardware to run their applications. As the deployment cost is an integral part of applications deployed on the cloud, the cost-efficiency of provisioning resource to these applications becomes a priority, and it is of growing significance, since the total spending that will be affected by cloud computing is over $1 trillion by 2020 [7].

Many cloud providers such as Amazon Web Services offer four types of instances (i.e., VMs) [15]: on-demand, reserved, dedicated, and spot (also known as preemptible). Cloud customers can pay for renting on-demand instances per hour without long-term commitments, and they cost the most. Also, they can rent reserved instances for a long term by making an upfront payment to cloud providers and thus pay a much lower rate than on-demand instances. A variation of reserved instances is a dedicated host, which is a physical server that is assigned only to a specific customer, and nobody besides this customer can use the resources of this host. Unlike the fixed-cost paying schemes mentioned above, a variable-cost paying scheme allows cloud customers to specify the price they are willing to pay for renting a spot instance to run their applications [15], and, depending on the varying demand from cloud customers, the price of this spot instance can go up if the demand increases and the number of available instances that can be supported by a finite number of physical resources in a data center of cloud providers decreases [126]. Conversely, the price of this spot instance can go down if the demand decreases and the number of available instances increases. If the customer's price is greater than the cloud provider's price that depends on the demand, a spot instance will be provisioned to customers' applications at the customer's price. However, when spot instances are already provisioned to customer applications and the cloud provider's price goes above the customer's price, the cloud providers will revoke those spot instances within two minutes by sending termination signals, thus resulting in *revocations of those spot instances* [126], whose occurrences are very difficult to predict [14]. As a result, even though cloud customers sometimes rent spot instances at 90% lower costs compared to on-demand [15], their applications that run in spot instances can be terminated based on price fluctuations that happen frequently, thus these applications may switch to an incorrect state leading to certain bugs [127, 128].

In general, terminations could be seen as *regular* when an application receives a termination signal in the context of predefined protocols, or *irregular* when an application receives a termination signal without using any context of predefined protocols. Hence, the revocations of spot instances often lead to irregular terminations of cloud-based applications. Note that an application can be irregularly terminated in two modes. We assume that the reason for executing an application is to run an algorithm that implements the requirements of this application to provide the required results. First, an application could be irregularly terminated during the execution of the application's algorithm. Second, an application could be irregularly terminated during the execution of the shutdown sequence of the application when the execution of the application's algorithm is completed. Moreover, irregular terminations do not affect stateless applications but often affect stateful applications relying on the results of ongoing calculation by applications under irregular terminations. These stateful applications might change to incorrect states when they are terminated before their shutdown sequences are entirely executed. In general, resources utilized by an application under irregular termination can be called ***R****esources* ***A****ffected by* ***T****ermination (RAT)*. When an application (A) encounters irregular terminations while interacting with another application (B), B is considered RAT because it might be left in an incorrect state until it identifies that A is already terminated.

EC2 spot markets contain approximately 7600 independent spot prices for different types of instances among 44 availability zones (i.e., data centers) in 16 regions [129]. With spot instances becoming pervasive, irregular terminations have become a part of the normal behavior of cloud-based applications. ***B****ugs of cloud-based* ***A****pplications resulting from* ***S****pot* ***I****nstance* ***R****evocations (BASIR)* result from errors in the implementation of the shutdown instructions of these applications that occur only during spot instance revocations. When these applications are being irregularly terminated, they might lose their states that lead to BASIR, such as data loss, inconsistent states, performance bottlenecks, hangs, crashes, deadlocks, locked resources, or these applications that cannot restart/terminate. On top of poor user experience from seeing these bugs, other bugs result in situations where cloud-based applications could not be restarted without manual interventions. In finer detail, when an application invokes synchronization system calls to lock a file and perform an update on the file inode's field that specifies if the file shall be persisted and this application is being irregularly terminated before the update is completed, system calls (i.e., fsync) of this application that are responsible for synchronizing the data of an open file to the storage device will become a ”no-op”, causing data loss of this file [130].

In general, heartbeat or timeout mechanisms might reduce the number of BASIR that require interaction between external processes (or threads) that run in different instances and an application's processes (or threads) run in a spot instance under irregular terminations, i.e., deadlocks, hangs, and performance bottlenecks. However, these mechanisms may not be useful for other types of BASIR that solely depend on ongoing calculations by applications deployed on a spot instance under irregular terminations, i.e., data corruption, data loss, crashes, and inconsistent states of shared data objects. Cloud-based applications that run in spot instances are not designed or tested to deal with this behavior in the cloud environment. The shutdown sequence of a cloud-based application is often left untested because developers often assume that a cloud-based application is properly terminated as long as its processes are terminated. It is very difficult to find BASIR because a termination signal can be initiated at every execution state of a cloud-based application, leading to a significantly larger search space of application states [13]. Unfortunately, the absence of testing the effect of spot instance revocations on cloud-based applications will likely lead to a large number of BASIR. As a result, the advantages of cloud spot instances could be significantly minimized or even entirely negated.

We propose a novel solution to automatically find BASIR and locate their causes in the source code of cloud-based applications. We develop our solution for ***Testing for BASIR*** *(`T-BASIR`)* that uses kernel modules (KMs) [131] to find these bugs and generate traces of their causes in the source code. `T-BASIR` is comprised of two major components. (1) Automating BASIR detection using KMs that contain the following main phases: (i) sending termination signals to certain system calls of a cloud-based application, and (ii) measuring the impacts on the state of RAT when the cloud-based application is irregularly terminated during the execution of these system calls. (2) Identifying the causes of BASIR using Tracer KM, which modifies the flow of executions through intercepting a termination signal to collect execution traces from the stack of a cloud-based application before the application receives the termination signal. BASIR and the traces of BASIR can be analyzed during application testing by developers, who look for fixes for these bugs to reduce or even eliminate the number of these bugs when cloud-based applications encounter irregular terminations. The motivation behind this work is to design a technique enabling developers to test the effect of spot instance revocations on cloud-based applications.

Contributions: We address a new and challenging problem for cloud-based applications that results from irregular terminations due to spot instance revocations. To the best of our knowledge, `T-BASIR` is the first automated solution to find bugs of applications resulting from cloud spot instance revocations. We evaluate `T-BASIR` using 10 popular open-source applications. Our results show that `T-BASIR` not only finds more instances and different types of BASIR (e.g., performance bottlenecks, data loss, locked resources, and applications that cannot restart) compared to the random approach, but it also locates the causes of BASIR to help developers improve the design of the shutdown process for cloud-based applications during their testing.

5.3 Problem Statement

In this section, we provide a background on shutdown processes and revocation notifications, discuss sources of BASIR, illustrate the BASIR problem, and formulate the problem statement.

5.3.1 Shutdown Processes and Revocation Notifications

The shutdown process of an application is often initiated during the execution of application instructions in response to termination signals. This allows the application to switch its execution control to execute predefined shutdown instructions that save the state of the application and the state of its artifacts within a certain timeout before the operating system removes the application process from the memory. It is very difficult to specify in which sequence instructions should be executed during the shutdown of an application. Doing so requires the knowledge of the execution state of an application at any point when this application receives a termination signal. Also, specifications describing the shutdown process of an application and which states are incorrect are rarely documented. The shutdown process of an application is often left untested because developers often assume that an application is properly terminated as long as its processes are terminated. As a result, the shutdown process of applications may fail to be completed within a certain timeout, leading to an incorrect state that affects the execution of subsequent instances of this application.

In general, cloud providers revoke (i.e., terminate) spot instances after a brief two-minute notification. The revocation notifications are often sent to spot instances when the demand from cloud customers increases and the number of available spot instances that can be supported by a finite number of physical resources in a data center of cloud providers decreases. If the customer's price is greater than the cloud provider's price that depends on the demand, a spot instance will be provisioned to customers' applications at the customer's price. However, when spot instances are already provisioned to customer applications and the cloud provider's price goes above the customer's price, the cloud providers will revoke those spot instances within two minutes by sending termination signals [126]. The cloud providers give spot instances two-minute revocation notifications to enable applications that run in spot instances to be gracefully shut down within the two-minute revocation notice time. However, the brief two-minute revocation notice is not often enough to complete the shutdown process of applications, especially when the applications' memory footprints are greater than 4GB [79]. As a result, when these applications are being terminated during the execution of the shutdown process of these applications, they might lose their states that lead to BASIR.

5.3.2 Sources of BASIR

There are two primary sources of BASIR. The first one is spot instance revocations. The other one is shutdown bugs of cloud-based applications.

5.3.2.1 Spot Instance Revocations

The revocations of spot instances are based on price fluctuations that happen based on demand of spot instances from many cloud customers. The cloud providers often revoke spot instances when the demand increases and the number of available spot instances that can be supported by a finite number of physical resources in a data center of cloud providers decreases. It is very difficult to determine in advance spot instance revocations that depend on the varying demands of cloud customers [25]. Doing so requires cloud customers (i.e., application's owners) to understand how the demands of the spot instances change, how the costs of the allocated spot instances change, and how to make trade-offs between the demands and these costs [1]. As a result, price fluctuations that depend on the demand have a high influence on the number of spot instance revocations.

In addition, it is very difficult for cloud customers to decide a price they are willing to pay for renting a spot instance to run their applications in such a way that reduces the deployment cost and the number of spot instance revocations [106]. When spot instances are already provisioned to cloud customer applications and the customer's price is close to zero, there is a high probability that those spot instances will be revoked by cloud providers. Also, when a cloud customer requests spot instances and the customer's price is close to zero, there is a very low probability that those spot instances will be provisioned to cloud customer applications. Conversely, if cloud customers set their prices close to on-demand instances' prices, cloud customers may reduce the number of revocations of spot instances that are provisioned to their applications, but cloud customers may not benefit from a significant discount of spot instances that is up to 90% compared to on-demand instances [15]. As a result, without knowing a demand from different cloud customers in advance, the challenge for cloud customers is to choose a price of spot instances that is both significantly lower than the price of on-demand instances and greater than the cloud provider's price to minimize the cost of the deployment and the number of spot instance revocations.

5.3.2.2 Shutdown Bugs of Cloud-Based Applications

The shutdown bugs of applications often result from errors in the implementation of a cleanup process of these applications that occurs only during their shutdowns [132]. The shutdown sequence of an application is often left untested because developers often assume that an application is properly

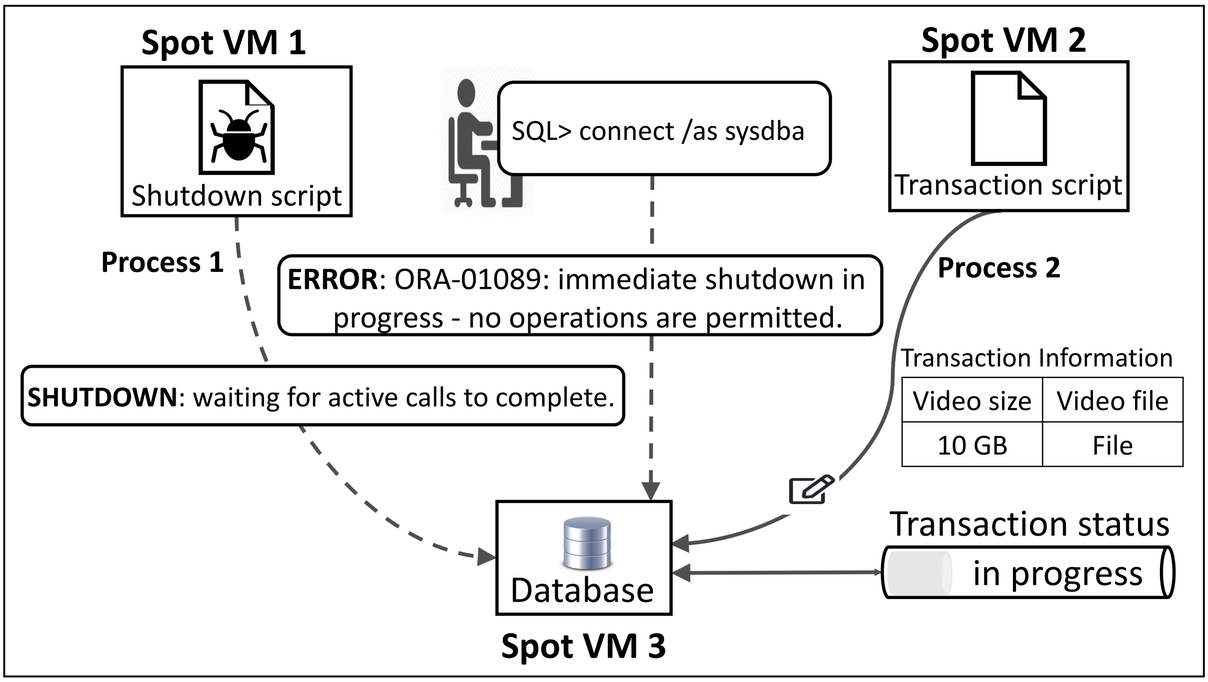


Figure 6: An illustrative example of BASIR.

terminated as long as its processes are terminated. Developers often depend on the assumption that the operating system cleans the process space to a certain extent in any case. Also, specifications describing the shutdown process of an application and which states are incorrect are rarely documented. Unfortunately, existing bug finding tools (e.g., PMD [93] and FindBugs [96]) are not applicable to BASIR because they rely on searching through the application's execution paths for certain inputs to check if the state value of an application varies from the expected value that represents the input value of the next instruction in this execution path [13]. However, a termination signal can be initiated at every execution state of applications, leading to a significantly larger search space of these states. On top of that, the shutdown sequence of an application varies based on the type of termination signals [131].

In addition, it is very difficult to analyze irregular terminations, even for a single execution path of an application for certain inputs since termination signals can be initiated at every point during the execution of the path resulting in deviations from the execution path [133]. For example, termination signals that are initiated during the execution of the third-party's instructions could change the application state, resulting in BASIR. Also, it is very difficult to specify in which sequence instructions should be executed during the shutdown of an application. Doing so requires the knowledge of the execution state of an application at any point when this application receives a termination signal. Furthermore, multiple termination signals can be initiated during the execution of the shutdown instructions of an application, leading to a significantly larger search space.

5.3.3 Illustrative Example

The BASIR problem with a cloud-based application is illustrated in Figure 6. As discussed in Section 5.3.2, BASIR results from two primary sources: shutdown bugs of applications and spot instance revocations. We show an instance of BASIR that arises from the interactions between a shutdown bug of an application, which comes from a real shutdown bug [127], and the revocation of a spot VM that represents the normal behavior of spot VMs. Our illustrative example shows a typical cloud-based application where a cloud-based application and its artifacts are often replicated across multiple VMs to improve its fault tolerance and reduce its network latency. The cloud-based application and its artifacts are deployed on three spot VMs, where spot VM 1 contains an Oracle shutdown script that reflects a routine script for databases in production, spot VM 2 contains a transaction script that uploads a video file with a large size (e.g., 10GB), and spot VM 3 contains an Oracle database.

Suppose that the Oracle shutdown script in spot VM 1 that runs on a particular process (Process 1) is executed to terminate the Oracle database that runs in spot VM 3 at the same time another process (Process 2) in spot VM 2 is holding the lock on this Oracle database to perform the transaction. Hence, Process 1 will be waiting until Process 2 releases the lock from the Oracle database. However, consider what happens when spot VM 2 is revoked as a part of the normal behavior of spot VMs while the transaction that is executed by Process 2 is still ongoing. Since Process 2 does not release the lock before the revocation of spot VM 2, the Oracle database will hang and consume needlessly resources until Process 1 determines that Process 2 is gone. The Oracle database prevents users from performing other operations (see the error message in the middle of Figure 6), since the database is waiting for active calls to be finished (see the log on the left side of Figure 6). Furthermore, if the spot VM 3 that contains the database is also revoked, this revocation (i.e., an irregular termination of the database) may not only produce an inconsistent state of various data or an incorrect state of artifacts in the database but also may affect the execution of subsequent instances of the database.

Additionally, we point out to multiple real-world bugs resulting from irregular terminations to shed light on the effect of spot instance revocations on applications. Irregular revocations could cause severe bugs, such as EX file system corruption [134], data loss on Atom editor [135], data loss on XFS file system [136], data corruption on Docker container [137], SQLite file corruption [138], database corruption on Docker [139], Leveldb database corruption [140], and Mosquitto database corruption [141]. Also, the Linux documentations describe that although Linux can often repair file system corruption due to a power failure, some situations may require manual interventions to repair non-recoverable file system issues [142].

5.3.4 The Problem Statement

With spot instances becoming pervasive, bugs of cloud-based applications resulting from spot instance revocations have become a very important concern for cloud customers (i.e., application's owners). In this work, we address a new and challenging problem of testing the effect of spot instance revocations on cloud-based applications – *how to find bugs of cloud-based applications that result from spot instance revocations*. Also, (Equation 5.1) and (Equation 5.2) describe how to search through RAT for certain execution points (i.e., system calls) to check if the value t'_{ij} of RAT j during the execution of an execution point i when a cloud-based application is irregularly terminated varies from the expected value t_{ij} that represents the value of RAT j during the execution of an execution point i when a cloud-based application is regularly terminated. Once a difference b_{ij} is found, this difference is added to the matrix B of potential BASIR.

$$B := T - T' \tag{5.1}$$

$$b_{ij} = \begin{cases} 0 & t_{ij} = t'_{ij} \\ (t_{ij} - t'_{ij}) & t_{ij} \neq t'_{ij} \end{cases}$$

$$\forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, m\}, \tag{5.2}$$

$$t_{ij} \in T, \quad t'_{ij} \in T', \quad b_{ij} \in B$$

Here, T is a matrix of size $n \times m$, n and m designate the total number of execution points (i.e., system calls) and RAT, respectively, for regular terminations of a cloud-based application, t_{ij} is the value of RAT j during the execution of an execution point (i.e., a system call) i when a cloud-based application is regularly terminated. T' is another matrix of size $n \times m$ for irregular terminations of a cloud-based application, t'_{ij} is the value of RAT j during the execution of an execution point i when a cloud-based application is irregularly terminated. Also, B is another matrix of size $n \times m$ for potential BASIR, b_{ij} is the difference between t_{ij} and t'_{ij}.

The root of this major problem is that cloud-based applications that are exposed to irregular terminations are not designed or tested to deal with this behavior in the cloud environment. Thus, when cloud-based applications are being irregularly terminated, their current state might be lost, which leads to certain bugs, such as data loss, inconsistent states, performance bottlenecks, hangs, crashes, deadlocks, or locked resources. On top of poor user experience from seeing these bugs, other bugs result in situations where cloud-based applications could not be restarted without manual interventions. As a result, the advantages of cloud spot instances could be significantly minimized or even entirely negated. To the best of our knowledge, T-BASIR is the first automated solution to identify instances of BASIR. BASIR results from two primary sources: shutdown bugs of applications and spot instance revocations. However, since spot revocations are unpredictable and cloud-based applications are not designed or tested to deal with cloud spot revocations, BASIR is a critical problem for cloud customers, and T-BASIR is an essential tool to shed light on the effect of spot instance revocations on cloud-based applications. Thus, when the number of spot instance revocations or the number of shutdown bugs of applications increase, the number of BASIR will likely increase, and vice versa.

Specifically, we use kernel modules to find these bugs and generate traces of their causes in the source code. With our solution, developers can analyze the found bugs and their traces to improve the design of the shutdown process for cloud-based applications during the testing of these applications. Automatically finding these bugs is extremely difficult, in general, especially since a termination signal can be initiated at every execution state of applications, leading to a significantly larger search space.

5.4 Our Approach

In this section, we introduce KMs, explain why we use KMs, describe how we utilize KMs in `T-BASIR` and outline the architecture and workflow of `T-BASIR`.

5.4.1 Why We Use Kernel Modules in `T-BASIR`

A KM is a mechanism for (un)loading some codes into an operating system at runtime without rebooting the operating system to extend its functionalities [131]. KMs facilitate modifying the flow of executions, handling the interruption of termination signals, and accessing the information of kernel space functions. There are three main reasons for using KMs rather than modules in the user space. First, using modules in the user space, it is very difficult to synchronize between a process of a cloud-based application that performs a specific operation (e.g., write) on certain resources and a process that sends a termination signal to this application. Second, it is very difficult to time the execution of a particular instruction of a cloud-based application in the user space because an operating system that runs in the kernel space determines the schedule of executing this instruction. Third, some termination signals (e.g., SIGKILL) often invoke the signal handlers in the kernel space instead of the signal handler in the user space (i.e., a signal handler that is defined in the source code of a cloud-based application) [131]. In contrast, KMs have complete control over the execution in the kernel space at runtime. As a result, `T-BASIR` uses KMs to ensure termination signals are sent to certain points in the execution of a cloud-based application and to measure the impact on the state of RAT at these points of the execution in order to find BASIR.

5.4.2 Why We Use Synchronization System Calls in `T-BASIR`

In general, the synchronization system calls are responsible for managing the access of shared data objects among multiple processes (or threads). `T-BASIR` focuses on the synchronization system calls since the irregular terminations of synchronization system calls may negatively affect not only the state of shared data objects causing bugs (e.g., data loss, data corruption) but also the state of external processes (or threads) that run on different instances and interact with the process of terminated system calls, causing bugs (e.g., deadlocks and performance bottlenecks). However, although a write system call is another important type of system call that is responsible for modifying the value of data objects, the irregular termination of write system calls may negatively affect only the modified data objects, causing bugs (e.g., data loss, data corruption). Thus, the irregular termination of synchronization system calls may cause more bugs that are related to data objects and processes (or threads) within the critical section of synchronization system calls, compared to the irregular termination of the write system calls that may cause bugs related to only the modified data objects.

5.4.3 Automating BASIR Detection Using KMs

In `T-BASIR`, our terminator KM specifies when we send a termination signal during the execution of cloud-based applications that mimics the irregular terminations, as discussed in Section 5.2. An essential goal is to identify which instructions of applications are more likely to lead to BASIR in order to send termination signals during the executions of these instructions. Given that BASIRs are more likely to be exposed when instructions use resources to perform certain operations (e.g., write) that are often accessed when specific system calls [131] (e.g., acquire-lock) are invoked, we favor instructions whose executions access these resources. Our terminator KM sends a termination signal during the execution of these system calls, which correspond to specific instructions in the source code. Our terminator KM uses the number of a system call with KProbe and JProbe interfaces [131] to intercept the execution of these system calls and, hence, ensures that a termination signal is sent to certain points of the execution. In summary, our terminator KM sends termination signals only during the execution of these instructions to increase the degree of precision for finding BASIR. In the RANDOM approach, a termination signal is sent to any point in the execution of a cloud-based application. Our hypothesis is that our terminator KM is more effective than randomly sending termination signals to any instructions because determining to which instruction a termination signal should be sent is highly correlated to the probability of finding BASIR. We verify our hypothesis with the experimental data in Section 5.6.

$$B(T,T') = \sum_{i=1}^{n}\sum_{j=1}^{m} D(t_{ij},t'_{ij}) \text{ where } t \in T,\ t' \in T' \tag{5.3}$$

$$D\left(t_{ij},t'_{ij}\right) = \begin{cases} 0 & t_{ij} = t'_{ij} \\ \\ 1 & t_{ij} \neq t'_{ij} \end{cases} \tag{5.4}$$

Here, T is a matrix of size $n \times m$, n and m designate the total number of system calls and RAT, respectively, for regular terminations of a cloud-based application, t_{ij} is the value of RAT j during the execution of a system call i when a cloud-based application is regularly terminated. T' is another matrix of size $n \times m$ for irregular terminations of a cloud-based application, t'_{ij} is the value of RAT j during the execution of a system call i when a cloud-based application is irregularly terminated. D is the delta function that evaluates the presence of BASIR by comparing the difference between the value of RAT when a cloud-based application is regularly terminated and the value of the same RAT when this application is irregularly terminated during the execution of the same system call. B is the summation function that computes the total number of BASIR by analyzing executions between irregular and regular terminations of a cloud-based application for m RAT and n system calls.

In `T-BASIR`, our detector KM determines when irregular terminations lead to BASIR. We use the values of RAT (e.g., variables and artifacts) for cloud-based applications to identify the presence of BASIR. Initially, we randomly select a set of system calls of a cloud-based application. Then, we use our identifier KM to record the values of RAT that are used by these system calls when a cloud-based application is regularly terminated. For each system call, we run this application to collect the values of the RAT when this application is irregularly terminated. Our detector KM uses (Equation 5.4) to measure the difference between the value of RAT when the cloud-based application is regularly terminated and the value of the same RAT when the cloud-based application is irregularly terminated during the execution of the same system call. We use the difference operation to evaluate the presence of BASIR by analyzing executions between irregular and regular terminations, since we assume that running a single execution path of a cloud-based application for certain inputs multiple times leads to the same values of the RAT in different runs. When the value of the RAT after irregular terminations varies from the expected value of the RAT at the same point in the execution after regular terminations, it indicates a potential instance of BASIR. Hence, once a difference is found, the detector KM uses (Equation 5.3) to add this difference to the total number of potential BASIR and collects the traces of this BASIR, as discussed in Section 5.4.4. As a result, with T-BASIR, developers can analyze the found instances of BASIR and their traces to improve the design of the shutdown process for cloud-based applications during the testing of these applications.

`T-BASIR` is illustrated in Algorithm 1 that contains the following main phases: (i) send termination signals to certain system calls of a cloud-based application, and (ii) measure the impacts on the state of RAT when the cloud-based application is irregularly terminated during the execution of these system

Algorithm 2 `T-BASIR`'s algorithm for finding BASIR and locating their causes.

1: **`Inputs:`** KM Configuration Ω, Application $\mathcal{A}$
2: **`LoadIdentifierKMs`**(Ω)
3: **while** $\mathcal{A}$ $\neg$ **`Terminate`** **do**
4:     $\mathcal{T} \leftarrow$ **`IdentifySyscallRAT`**($\mathcal{A}$, Ω)
5: **end while**
6: **`UnloadIdentifierKMs`**(Ω)
7: **`LoadTerminatorDetectorKMs`**(Ω)
8: **for** each system call i **in** $\mathcal{T}$ **do**
9:     **for** each RAT j **in** $\mathcal{T}$ **do**
10:         $t'_{ij} \leftarrow$ **`MeasureSyscallRAT`**($\mathcal{A}$, Ω)
11:         **if** $t_{ij} \neq t'_{ij}$ **then**
12:             $\mathcal{B} \leftarrow \mathcal{B} + 1$
13:             $\mathcal{C} \leftarrow$ **`CollectTraces`**(t'_{ij})
14:         **end if**
15:         **`RestoreAppInitialState`**($\mathcal{A}$)
16:     **end for**
17: **end for**
18: **`UnloadTerminatorDetectorKMs`**(Ω)
19: **return** $\mathcal{B}$, $\mathcal{C}$

calls. The algorithm for `T-BASIR` takes in the entire set of inputs for the cloud-based application, its snapshot, and the KM configurations Ω, containing the identifier, terminator and detector KMs. Starting from Step 2, the algorithm loads the identifier KM into an operating system. In `T-BASIR`, we use lock system calls, where a thread locks certain resources to perform read or write operations. In Steps 3-5, the identifier KM randomly selects a set of system calls and records the values of RAT that are used by these system calls when the cloud-based application is regularly terminated. In Step 6, the identifier KM is unloaded from the operating system. In Step 7, the terminator and the detector KMs are loaded into the operating system. In Steps 8-17, for each system call and RAT, the algorithm repeatedly runs the snapshot of the cloud-based application, and then the terminator KM sends a termination signal to the cloud-based application during the execution of this system call. For each run, the detector KM uses (Equation 5.4) to measure the difference between the value of RAT when the cloud-based

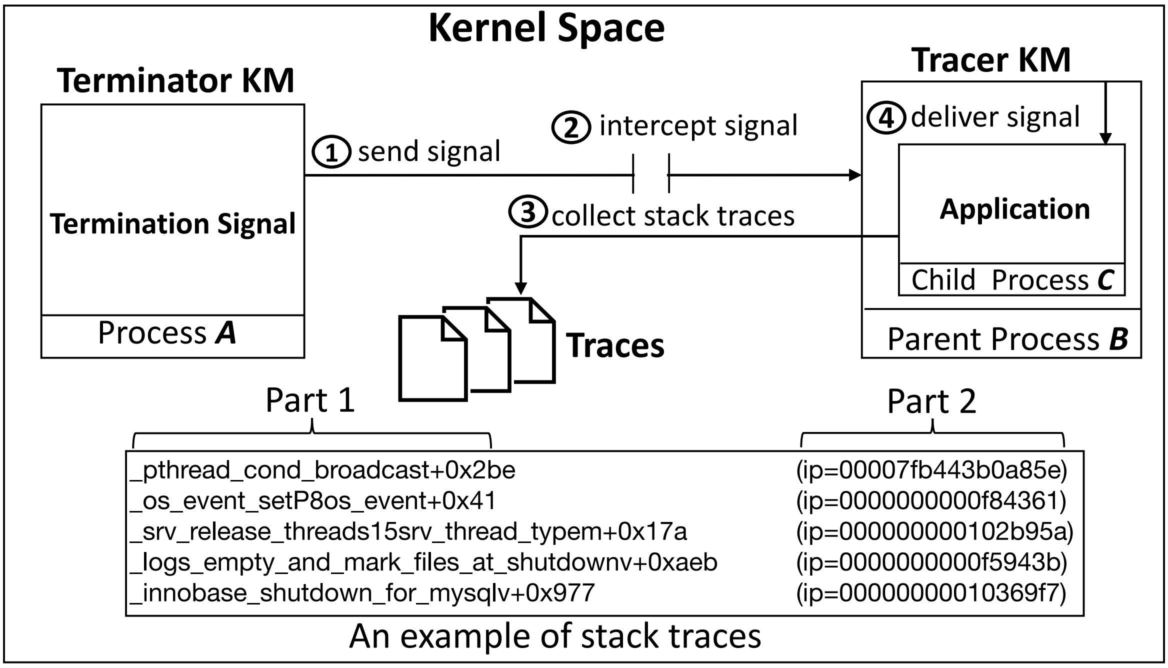


Figure 7: The implementation design of T-BASIR tracer.

application is regularly terminated and the value of the same RAT when this application is irregularly terminated during the execution of the same system call. Once a difference is found, the detector KM uses (Equation 5.3) to add this difference to the total number of potential BASIR and collects its traces, as discussed in Section 5.4.4. The cycle of Steps 8-17 repeats until the set of system calls is completed. Finally, in Step 18, the terminator and the detector KMs are unloaded from the operating system. The found instances of BASIR and their traces are returned in Step 19 as the algorithm ends.

5.4.4 Identifying the Causes of BASIR

Tracer KM is at the core of the T-BASIR tracer to identify the causes of BASIR. We provide an overview and describe the implementation design of the T-BASIR tracer.

5.4.4.1 Overview of T-BASIR tracer

Our goal is to automatically determine specific instructions in the source code of cloud-based applications that lead to BASIR when these applications encounter irregular terminations. In order to contrast instructions that lead to BASIR, we rely on the stack trace approach [143] that can be used to collect execution traces from the stack in the memory when a cloud-based application is irregularly terminated. The stack traces contain a sequence of method calls with corresponding instructions, which often represents the current point in the execution path. These traces are often difficult to capture because termination signals can be initiated at every point in the execution of a cloud-based application, leading to a significantly larger search space. Hence, existing tracing tools [143] are not applicable to BASIR because the stack traces of applications are gone as soon as these applications are terminated. However, our tracer KM in `T-BASIR` can intercept a termination signal before this signal is delivered to a cloud-based application, as discussed in Section 5.4.4.2. As a result, during application testing, developers can use these traces to identify corresponding instructions in the source code that lead to instances of BASIR.

5.4.4.2 Implementation Design of the T-BASIR Tracer

The implementation design of the `T-BASIR` tracer in the kernel space is illustrated in Figure 7. The rectangles denote Terminator KM, Tracer KM, and a cloud-based application with their processes. The arrows indicate the actions between these KMs and the cloud-based application, and the numbers in the circles show the sequence of operations in the `T-BASIR` tracer.

Our tracer KM can intercept a termination signal before this signal is delivered to the cloud-based application because this application runs inside our tracer KM, as illustrated on the right side of Figure 7 (i.e., this application runs on a child process of the tracer/parent process). In particular, when a termination signal is sent (1) to the cloud-based application, our tracer KM first intercepts (2) this signal to collect and store (3) the execution traces of the cloud-based application in files (e.g., text files) and then delivers (4) this signal to terminate this application.

In general, it is very difficult to time the execution of a particular instruction of a cloud-based application in the user space because an operating system that runs in the kernel space determines the schedule of executing this instruction. Conversely, KMs have complete control over the execution in the kernel space at runtime. Hence, our tracer KM modifies the flow of executions through intercepting a termination signal to collect execution traces from the stack of a cloud-based application before the application receives the termination signal. In particular, our tracer KM uses Libunwind interfaces [144] to generate execution traces from the stack memory of the cloud-based application. An example of stack traces for a cloud-based application (e.g., MySQL) is illustrated in the bottom of Figure 7. The first part of these traces refers to the sequence of method calls with corresponding instructions that represents the current point in the execution path when the cloud-based application is being irregularly terminated, and the second part of these traces refers to the methods' instruction pointers. As a result, the traces of BASIR can be reviewed by developers during application testing to identify which instructions in the execution path may lead to instances of BASIR.

5.4.5 T-BASIR's Architecture and Workflow

The architecture of `T-BASIR` is illustrated in Figure 8. The rectangles indicate components of `T-BASIR`, the arrows denote the data flow between components, and the numbers in the circles show the sequence of processes in the workflow.

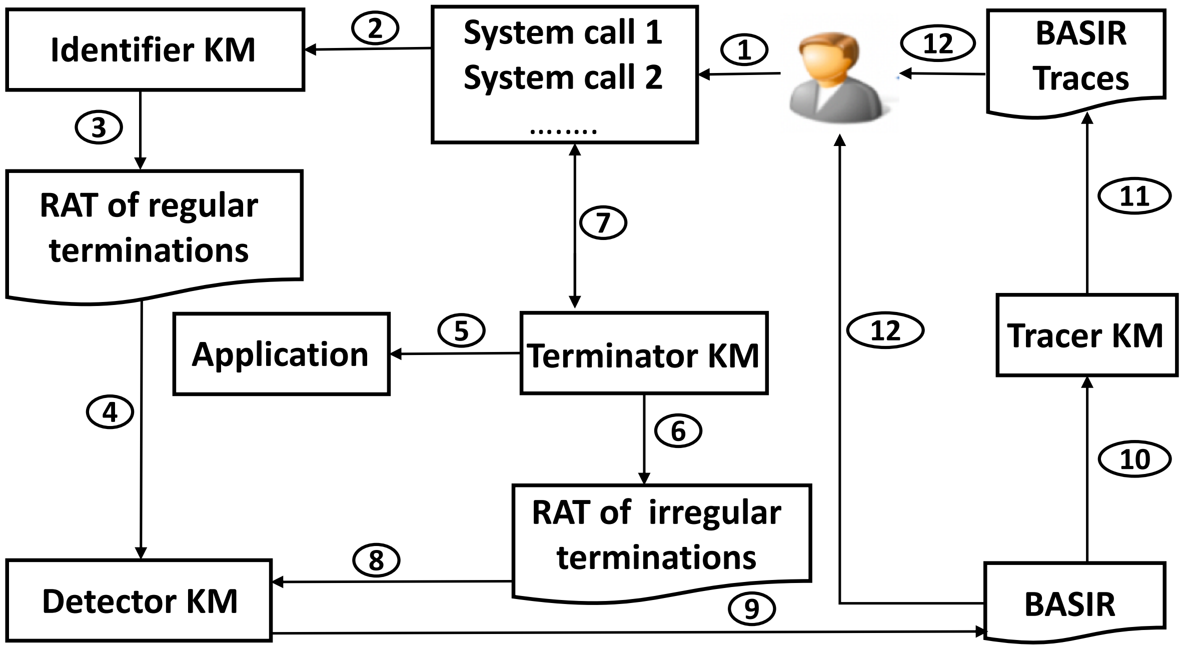


Figure 8: The architecture and workflow of T−BASIR.

TABLE VI: Overview of the applications: their names followed by the versions of the applications, and the total number of accessed futexes and their system calls when these applications restart after regular terminations.

Application	Version	Futexes	Syscalls
MySQL	v5.7.25	58	132
Cassandra	v3.0.17	35	138
PostgreSQL	v10.6	3	5
CouchDB	v2.3.0	25	11920
MongoDB	v3.0.6	61	1201
Hbase	v2.1.2	53	808
Docker	v18.09.0	45	1583
Hadoop	v3.0.3	34	1716
ZooKeeper	v3.4.12	35	910
Hive	v2.1.1	32	874

The input to `T-BASIR` is the entire set of inputs for a cloud-based application that performs specific operations (e.g., write) on certain resources (i.e., RAT), which often invoke particular system calls (e.g., acquire-lock) to use these resources. Initially, a set of system calls of the cloud-based application is chosen at random (1). For each system call, RATs are identified (2), and *Identifier KM* records (3) the values of RAT that are used by this system call when the cloud-based application is regularly terminated. These values of RAT that represent the expected values of the RAT, as discussed in Section 5.4.3, are passed (4) to *Detector KM*. *Terminator KM* sends (5) a termination signal to the cloud-based application during the execution of each system call. The values of RAT that are used by these system calls when the cloud-based application is irregularly terminated are collected (6). The evaluation is evolved using *Terminator KM* until the set of system calls is completed (7).

When the values of RAT for all system calls are collected, these values of RAT are passed (8) to *Detector KM*. *Detector KM* uses (Equation 5.4) to measure the difference between the value of RAT when the cloud-based application is irregularly terminated and the expected value of the same RAT when the cloud-based application is regularly terminated during the execution of the same system call. When the value of the RAT after irregular terminations varies from the expected value of the RAT at the same point in the execution after regular terminations, it indicates a potential instance of BASIR. Then, when a difference is found, *Detector KM* uses (Equation 5.3) to add this difference to the list of potential BASIR (9). Once the list of potential BASIR is obtained (10), *Tracer KM* collects (11) the traces of BASIR that contain a sequence of method calls with corresponding instructions, as discussed in Section 5.4.4. The found instances of BASIR and their traces are given to the developers for further evaluation (12).

5.5 Empirical Evaluation

In this evaluation section, we state our *Research Questions (RQs)*, illustrate subject applications, describe our methodology to evaluate `T-BASIR`, and outline threats to its validity.

***RQ_1*:** How effective is `T-BASIR` compared to the random approach in finding more instances of BASIR?

***RQ_2*:** How effective is `T-BASIR` in finding different types of BASIR?

***RQ_3*:** Do irregular terminations result in different impacts on the behaviors of the applications compared to the regular terminations?

***RQ_4*:** Is `T-BASIR` more effective than the random approach in causing more impacts on the application behaviors?

5.5.1 Subject Applications

We evaluated T-BASIR on 10 open-source subject applications. An overview of the subject applications is shown in Table VI. These applications are multithreaded, have high popularity indexes, come from different domains, and are written by different programmers. The synchronization mechanism of these applications relies on a futex system call [145], which is a fast user-space synchronization method that puts specific threads to sleep/wait or wakes waiting threads when specific conditions become true. Each critical section in these applications often uses certain futex variables that are stored in particular memory addresses and are used by multiple threads to access this critical section through futex system calls [145].

5.5.2 Methodology

For each application, we first use the Strace tool [131] to ensure that its synchronization mechanism relies on futex system calls. As discussed in Section 5.4.3, T-BASIR analyzes the values of the RAT between regular and irregular terminations at the same point in the execution to identify BASIR. RATs are the logs of the subject applications, the logs of the Linux kernel, the number of accessed futexes, and the number of futex system calls. An application is irregularly terminated using the RANDOM approach, where a termination signal is sent to any point in the execution of this application, and in T-BASIR, where a termination signal is sent to specific points in the execution of this application (i.e., during the executions of futex system calls). T-BASIR uses the logs to identify different types of BASIRs that lead to different effects on the behaviors of applications to answer $\boldsymbol{RQ}_1$ and $\boldsymbol{RQ}_2$. T-BASIR also identifies other cases of BASIR when the logs do not contain error messages. For example, T-BASIR identifies when applications cannot restart without manual interventions using the process status tool [131]. Also, we measure the impacts on the behaviors of the subject applications to answer ***RQ***$_3$ and ***RQ***$_4$. When an application restarts after irregular terminations, we check if values for the total number of accessed futexes and their system calls vary from the expected values when this application restarts after regular terminations for 20 seconds, which is set experimentally. Once a significant change is identified, as discussed in Section 5.4.3, `T-BASIR` adds this change to the total number of potential BASIR and collects its traces. `T-BASIR` is implemented using KMs, KProbe, and JProbe interfaces [131]. The experiments for the subject applications were carried out using 10 virtual machines. Each subject application was deployed on Ubuntu 18.04 LTS VM with 4 GB of memory and 4 GHz CPU. For each application, we created a snapshot to ensure a similar state of the test environment after irregular terminations.

5.5.3 Threats to Validity

Our implementation of `T-BASIR` deals with only futex system calls, whereas other applications may use different synchronization mechanisms (e.g., semaphore system calls [131]). While this is a potential threat, it is unlikely a major threat, since `T-BASIR` can be adjusted to support other types of synchronization mechanisms. In order to use `T-BASIR` with other applications, the developer can change only the system call type in the KMs so that `T-BASIR` identifies other types of system calls.

We experimented with only synchronization system calls, whereas other types of system calls (e.g., information flow, creation, preparatory, and termination) could also result in different effects on the behaviors of applications when these applications are terminated during the execution of other types of system calls. In contrast, understanding the effect of different types of system calls on the behavior of the applications is beyond the scope of this empirical study and shall be considered in future studies.

5.6 Empirical Results

In this section, we discuss the experimental results to answer the RQs listed in Section 5.5.

5.6.1 Finding more instances of BASIR

The experimental results to answer $\boldsymbol{RQ}_1$ are shown in Table VII and summarize the found instances of BASIR when the subject applications encounter irregular terminations using `T-BASIR` and RANDOM approaches. We focus on determining whether these applications restart without manual interventions after they are irregularly terminated using `T-BASIR` and RANDOM. The experimental results show that `T-BASIR` causes MySQL, CouchDB, MongoDB, HBase, Hadoop, and ZooKeeper not to restart without manual interventions, whereas the RANDOM approach causes only CouchDB to not restart without manual interventions. Our explanation is that the RANDOM approach was able to cause CouchDB not to restart without manual interventions, since CouchDB uses an extremely high number of futex system calls, as shown in Table VI. Hence, the RANDOM approach may accidentally hit these futex system calls, resulting in an instance of BASIR.

On the other hand, `T-BASIR` was not able to cause PostgreSQL, Cassandra, Docker, and Hive not to restart without manual interventions. Our explanation is that PostgreSQL uses an extremely low number of futex system calls as shown in Table VI. This situation puts `T-BASIR` at a disadvantage to find BASIRs since causing BASIR often requires more interactions among threads that often occur when a large number of futex system calls are executed. Cassandra runs on Java processes using a Java Virtual Machine (JVM), and `T-BASIR` uses Java processes instead of the application name processes (i.e., Cassandra) to specify the desired process of an application for receiving termination signals. Subsequently, JVM may play some roles in reducing the effect on Cassandra since Cassandra receives termination sig-

TABLE VII: The comparison of the results of BASIR for `T-BASIR` and RANDOM. The first column specifies the name of the applications followed by columns for `T-BASIR` and RANDOM, and the cells indicate whether irregular terminations using these approaches lead to BASIR (i.e., an application cannot restart without manual interventions).

Application	**T-BASIR**	**RANDOM**
MySQL	✓	✗
Cassandra	✗	✗
PostgreSQL	✗	✗
CouchDB	✓	✓
MongoDB	✓	✗
Hbase	✓	✗
Docker	✗	✗
Hadoop	✓	✗
ZooKeeper	✓	✗
Hive	✗	✗

nals through the JVM. Docker uses the resource isolation features for the kernel. `T-BASIR` uses KMs to send termination signals to the process of the subject applications. Hence, these features may play some roles in reducing the effect on Docker when Docker receives termination signals. Even though the Hive server restarts after irregular terminations using `T-BASIR`, its HCatalog component fails to restart. This observation allows us to conclude that even though irregular terminations may not show an impact on the restart state of an application, it does not mean that the other components of this application have no impacts too. In summary, our results show that `T-BASIR` causes six subject applications not to restart without manual intervention, whereas the RANDOM approach causes only one subject application not to restart without manual intervention, thus **positively addressing** $\boldsymbol{RQ}_1$.

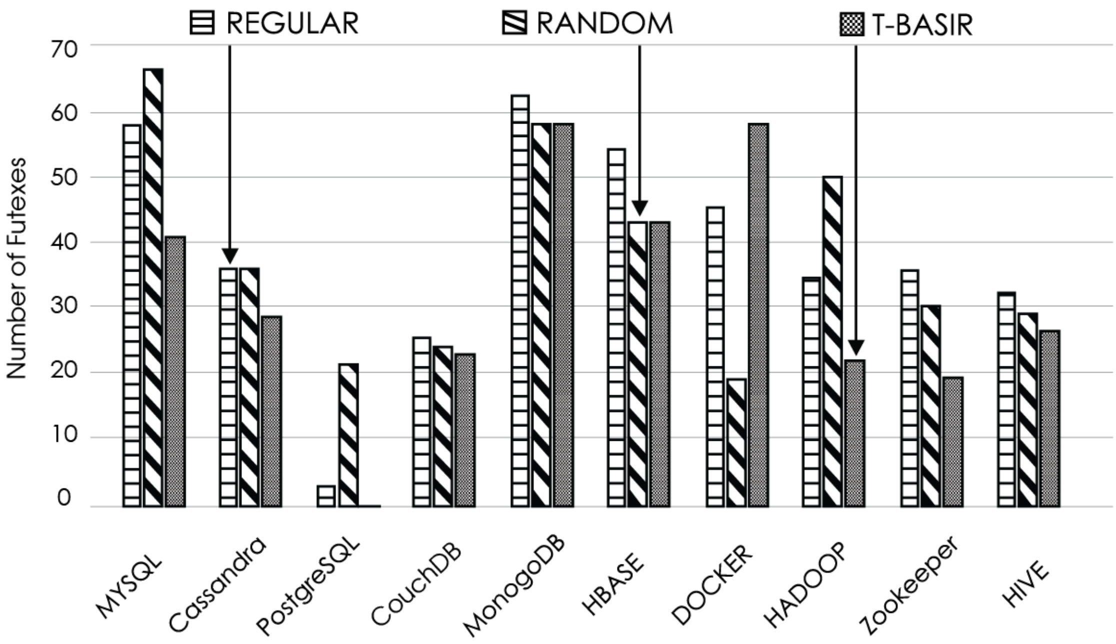


Figure 9: Comparing the total number of accessed futexes when the subject applications restart after regular and irregular terminations using `T-BASIR` and RANDOM.

5.6.2 Finding different types of BASIR

When we investigate ***RQ***$_2$, we observe that unlike the RANDOM approach, `T-BASIR` leads to other types of BASIR. Since we are more familiar with the MySQL components, we further analyze and discuss the effects of other types of BASIR for MySQL. We observe that the logs of MySQL report the following message. `[Note] InnoDB: page_cleaner: 1000ms intended loop took 848417ms` [146]. The message shows that the `page_cleaner` method that is responsible for writing data from memory into the disk takes a very long time from 1 second, which is expected, to 848 seconds (~14 minutes). This result demonstrates a major problem, since it results in not only performance bottlenecks but also data loss. We analyze the effect of data loss by creating a virtual machine with 1 GB of memory, and we use `MySQLlap client` to perform large write operations (e.g., inserting hundreds of records) using multiple threads. We then load `T-BASIR` into the operating system to send the termination signals during the execution of these system calls. Interestingly, we observed that once MySQL restarts, the recently written data is lost. This bug is also reported on the following web page [128]. Also, we observed the following error message: `[ERROR] InnoDB: Unable to lock ./ibdata1 error: 11` [146]. The error message shows that `T-BASIR` prevents MySQL from performing a clean shutdown and hence results in locked ibdata1, which is a file that includes the shared tablespace containing the internal data of InnoDB. Unlike the RANDOM approach, `T-BASIR` also leads to other types of BASIR, such as performance bottlenecks, data loss, and locked resources. This result confirms that `T-BASIR` also results in different types of BASIR, compared to the RANDOM approach, thus **positively addressing *RQ*$_2$**. As a result, when irregular terminations result in BASIR, `T-BASIR` collects the traces that contain a sequence of method calls with corresponding instructions, as discussed in Section 5.4.4. Hence, developers can use these traces to improve the design of the shutdown process for the subject applications during the testing of these applications.

5.6.3 Impact of irregular terminations on the behaviors of applications

The results of the experiments are presented in the histogram plot in Figure 9 that summarizes the number of accessed futexes for the subject applications when these applications restart after regular and irregular terminations using `T-BASIR` and RANDOM approaches. These futexes often control the access of shared resources in critical sections across various threads/processes of an application. Different futexes often correspond to different execution paths since these futexes control the access of critical sections in different methods of an application. We observe that the number of accessed futexes varies between regular and irregular terminations using `T-BASIR` and RANDOM approaches. This observation suggests that the execution paths between regular and irregular terminations of an application

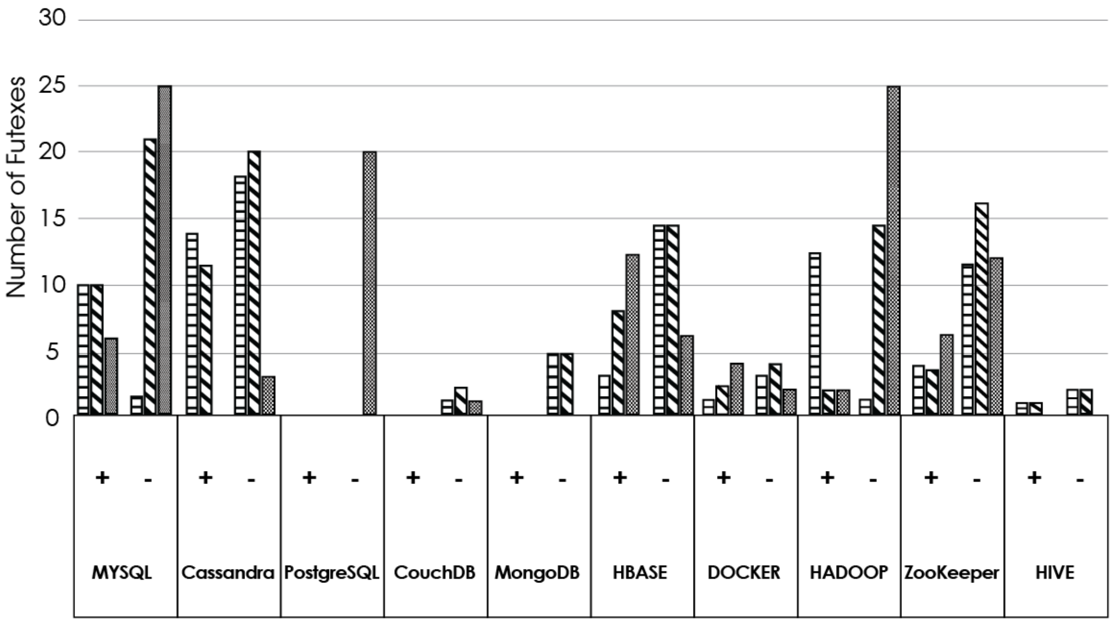


Figure 10: The change in the total number of accessed futexes between regular and irregular terminations using `T-BASIR` and RANDOM approaches for the subject applications. The plus and minus symbols specify extra and missing futexes, respectively. The horizontal stripes, diagonal stripes, and dotted bars represent the change of accessed futexes between RANDOM and REGULAR, `T-BASIR` and REGULAR,and `T-BASIR` and RANDOM approaches, respectively.

change where newly accessed futexes (i.e., extra futexes) may have been accessed in the recovery execution paths, or other futexes that are often used during the execution of the application startup may not have been accessed (i.e., missing futexes) [147]. We observe that, except for Docker, most numbers of accessed futexes when applications are irregularly terminated using `T-BASIR` are lower than the number of accessed futexes when applications are regularly terminated or irregularly terminated using the RANDOM approach. A higher change in the number of accessed futexes often indicates a higher change in the execution paths when an application restarts after regular and irregular terminations. Further details about the results for all applications are shown in Figure 11, where the number of extra and missing futexes are provided. Interestingly, we observe that there is a change in the number of accessed futexes between T-BASIR and RANDOM approaches, which suggests when an application encounters irregular terminations using different approaches, it often leads to different execution paths for the application. Hence, this observation confirms that the change in the execution paths not only indicates the recovery execution paths but also indicates other execution paths that may result in instances of BASIR [133, 148]. As a result, these experimental results demonstrate that when applications encounter irregular terminations using different approaches, it often leads to different execution paths, which result in different impacts on the behaviors of these applications, thus **positively addressing *RQ*$_3$**.

5.6.4 Impact of T-BASIR on the behaviors of applications

We present the change in the number of futex system calls for the subject applications in Table VIII when these applications restart after regular and irregular terminations using T-BASIR and RANDOM. We assume that running recovery execution paths of an application multiple times leads to the same values of the futex system calls for certain futexes in different runs. Hence, when the number of these futex system calls of an application after irregular termination using T-BASIR varies from the number of these futex system calls of this application after irregular termination using the RANDOM approach, it suggests the former recovery execution paths deviate from the latter recovery execution paths, which often indicates different impacts on the behaviors of this application. In particular, we observed that the number of futex system calls varies between regular and irregular terminations using T-BASIR and RANDOM approaches. This observation suggests the number of futex system calls may increase when specific threads do not release the lock from resources, resulting in thread contentions, or decrease when specific threads prevent other threads that use these futexes from reaching advanced points in the

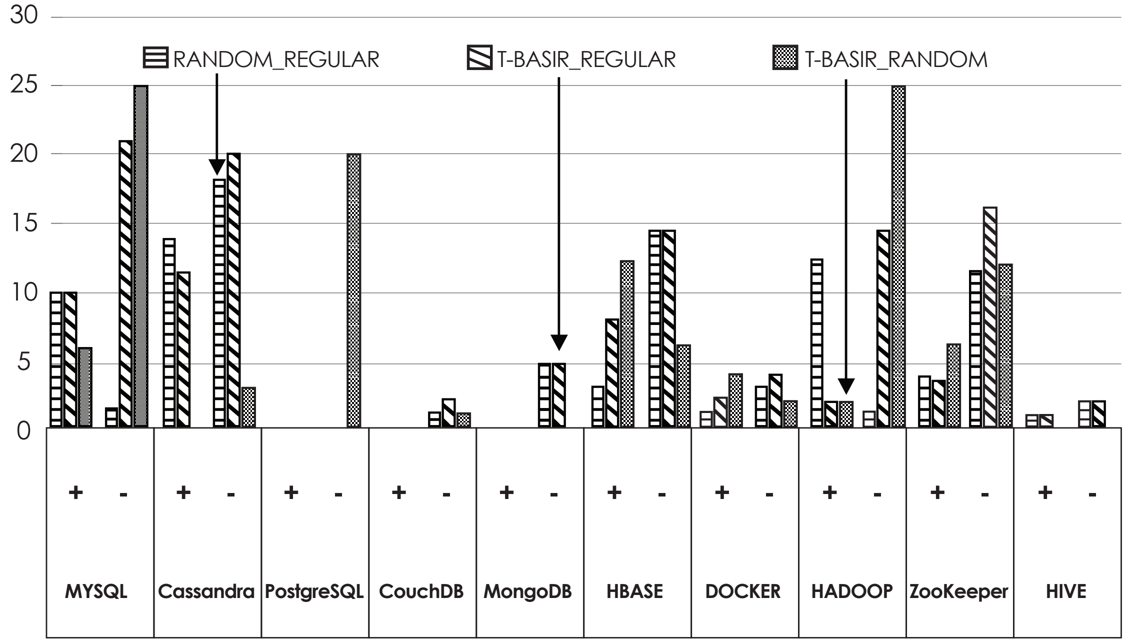


Figure 11: The change in the total number of accessed futexes between regular and irregular terminations using T-BASIR and RANDOM approaches for the subject applications. The plus and minus symbols specify extra and missing futexes, respectively. The horizontal stripes, diagonal stripes, and dotted bars represent the change of accessed futexes between RANDOM and REGULAR, T-BASIR and REGULAR,and T-BASIR and RANDOM approaches, respectively.

execution. In general, we observe that the number of futex system calls when Hive, Docker, Hadoop, Cassandra, MongoDB, and ZooKeeper are irregularly terminated using T-BASIR, except for a few futexes (they may correspond to the third-party's instructions (i.e., JVM)), is often less than the number of futex system calls when these applications are regularly terminated or irregularly terminated using the RANDOM approach. This observation suggests that irregular terminations that are initiated by T-BASIR often lead to more impacts on the behaviors of applications compared to the RANDOM approach since the lower number of futex system calls indicates not only a lack of thread executions but also incomplete recovery executions.

Conversely, we observe that the number of the futex system calls when CouchDB and Hbase are irregularly terminated, except for a few futexes, is often greater than the number of futex system calls when these applications are regularly terminated. This result suggests that irregular terminations often lead to more impacts on the behaviors of applications compared to the regular terminations since the higher number of futex system calls indicates not only more thread contentions but also a higher chance of locked resources. Interestingly, we observe that a futex with the last four digits of the memory address 0x0610 for CouchDB has a significant decrease in the number of its futex system calls between regular and irregular terminations, which suggests some threads that use this futex may be prevented (i.e., locked) from reaching this point in the execution. We also observe that a futex with the last four digits of the memory address of 0x0020 appears in extra futexes across different applications, such as Hadoop, HBase, and Hive, when they are restarted after irregular terminations. This observation suggests that this futex is invoked by recovery instructions of JVM, which is also reported on the collected traces of these applications [146]. Hence, fixing these recovery instructions of JVM will reduce or even eliminate the number of BASIR for all applications that rely on JVM. Also, we observe that the number of the futex system calls when MySQL and PostgreSQL are irregularly terminated is not significantly different from the number of futex system calls when these applications are regularly terminated. Our explanation is that PostgreSQL uses an extremely low number of the futex system calls, as shown in Table VI, and MySQL, unlike other applications, uses asynchronous I/O system calls. These situations make it more difficult to show different impacts on the behaviors of these applications in terms of the number of the futex system calls when these applications encounter irregular terminations.

Finally, we observed that, except for Docker, the number of missing futexes when the subject applications are irregularly terminated using `T-BASIR` is often higher than the number of missing futexes when these applications are irregularly terminated using the RANDOM approach. Similarly, we observed that, except for Hadoop and Postgres, the number of extra futexes when the subject applications are irregularly terminated using `T-BASIR` is often higher than the number of extra futexes when these applications are irregularly terminated using the RANDOM approach. This observation suggests that irregular terminations that are initiated by `T-BASIR` often lead to more impacts on the behaviors of applications compared to the RANDOM approach since a higher change in the number of accessed futexes often indicates a higher change in the execution paths when an application restarts after irregular terminations. In summary, these experimental results demonstrate that `T-BASIR` not only results in different impacts on the behaviors of these applications but also leads to more impacts on the behaviors of these applications compared to the RANDOM approach, thus **positively addressing** $\boldsymbol{RQ_4}$. As a result, when certain futexes result in significant changes in the behavior of applications, the traces of these futexes can be reviewed by developers to analyze how the changes of these futexes and their traces may lead to BASIR.

TABLE VIII: The comparison of the total number of futex system calls for the subject applications after regular and irregular terminations. The first column specifies the name of the applications followed by the memory address for a futex. The following columns designate REGULAR (T1), RANDOM (T2), and `T-BASIR` (T3), respectively.

App	Address	T1	T2	T3	Address	T1	T2	T3	Address	T1	T2	T3	Address	T1	T2	T3
CouchDB	0x7fbd804812c8	3	0	0	0x7fbd817802d0	412	512	528	0x7fbd81780450	392	523	530	0x7fbd817805d0	11	15	13
	0x7fbd81780190	400	512	522	0x7fbd81780310	397	526	534	0x7fbd81780490	563	705	686	0x7fbd81780610	5245	3402	3315
	0x7fbd817801d0	417	516	528	0x7fbd81780350	402	518	528	0x7fbd817804d0	396	520	528	0x94df78	3	3	0
	0x7fbd81780210	396	518	526	0x7fbd81780390	449	578	584	0x7fbd81780510	391	506	507	0x94f7f8	3	6	3
	0x7fbd81780250	409	506	518	0x7fbd817803d0	403	520	532	0x7fbd81780550	382	514	522	0x9595c8	19	5	9
	0x7fbd81780290	414	522	530	0x7fbd81780410	405	522	538	0x7fbd81780590	6	8	10	0x9595cc	1	1	1
	0x959660	1	1	1												
Hbase	0x7ff1b8014d28	4	64	56	0x7ff1b8014d78	2	26	24	0x7ff1b8014d7c	2	20	19	0x7ff1b8014f28	2	0	0
	0x7ff1b8014f78	1	0	0	0x7ff1b8015028	12	18	15	0x7ff1b8015078	8	7	6	0x7ff1b801507c	8	6	5
	0x7ff1b8015228	3	138	0	0x7ff1b8015278	2	57	0	0x7ff1b8026728	2	28	28	0x7ff1b8026778	2	12	10
	0x7ff1b802677c	3	13	11	0x7ff1b8028328	2	16	13	0x7ff1b8028378	2	6	6	0x7ff1b802837c	3	7	5
	0x7ff1b8055c28	34	1	139	0x7ff1b8055c78	18	2	62	0x7ff1b8055c7c	12	56	61	0x7ff1b80d1628	39	72	46
	0x7ff1b80d1678	21	34	27	0x7ff1b80d167c	12	25	15	0x7ff1b80db128	4	9	168	0x7ff1b80db178	3	4	63
	0x7ff1b80db17c	2	2	64	0x7ff1b80ddb28	4	21	13	0x7ff1b80ddb78	2	7	5	0x7ff1b80ddb7c	3	8	6
	0x7ff1b80ddf28	9	0	0	0x7ff1b80ddf78	4	0	0	0x7ff1b80ddf7c	2	0	0	0x7ff1b8111178	1	0	0
	0x7ff1b8115128	49	41	30	0x7ff1b8115178	26	28	18	0x7ff1b811517c	25	30	18	0x7ff1b8117428	65	182	4
	0x7ff1b8117478	25	67	2	0x7ff1b811747c	25	66	2	0x7ff1b811a378	1	0	1	0x7ff1b8123e28	162	209	186
	0x7ff1b8123e78	162	208	185	0x7ff1b83d5728	1	0	0	0x7ff1b83d577c	2	0	0	0x7ff1b83d7788	1	0	0
	0x7ff1b83d77d8	2	0	0	0x7ff1b83d77dc	1	0	0	0x7ff1b9644128	5	0	0	0x7ff1b9644178	4	0	0
	0x7ff1b9644328	13	4	1	0x7ff1b9644378	9	2	2	0x7ff1b964437c	1	2	0	0x7ff1be5c5540	1	1	1
	0x7ff1bf21e9d0	1	1	1	0x7fde20111278	0	1	1	0x7fde25870280	0	3	4	0x7f06c8000020	0	0	6
	0x7f06c811ac28	0	0	3	0x7f06c811ac78	0	0	2	0x7f06c811fc78	0	0	1				

Table VIII – Continued from previous page

App	Address	T1	T2	T3	Address	T1	T2	T3	Address	T1	T2	T3	Address	T1	T2	T3
Hive	0x7f6fe801d228	27	28	22	0x7f6fe801d278	17	20	16	0x7f6fe801d27c	18	20	16	0x7f6fe801d428	6	0	0
	0x7f6fe801d478	2	0	0	0x7f6fe801d47c	2	0	0	0x7f6fe8081928	47	52	36	0x7f6fe8081978	22	24	22
	0x7f6fe808197c	13	15	8	0x7f6fe808b428	3	9	3	0x7f6fe808b478	2	3	2	0x7f6fe808b47c	1	4	1
	0x7f6fe808de28	3	9	3	0x7f6fe808de78	1	3	1	0x7f6fe808de7c	2	4	2	0x7f6fe80c2f28	102	84	97
	0x7f6fe80c2f78	43	40	37	0x7f6fe80c2f7c	42	38	38	0x7f6fe80c3128	3	3	3	0x7f6fe80c3178	2	2	2
	0x7f6fe80c5228	101	122	107	0x7f6fe80c5278	44	43	41	0x7f6fe80c527c	43	44	42	0x7f6fe80c5428	10	6	0
	0x7f6fe80c5478	4	2	0	0x7f6fe80c547c	4	2	0	0x7f6fe80c8578	1	1	1	0x7f6fe80cd928	153	157	143
	0x7f6fe80cd978	152	156	142	0x7f6ff0bf4540	1	1	1	0x7f6ff0c0d280	3	0	0	0x7f6ff184e9d0	1	1	1
	0x7f5494000020	0	6	12												
Docker	0x557f25a478b0	9	120	9	0x557f25a47980	5	7	2	0x557f25a47990	8	116	6	0x557f25a479a8	2	2	2
	0x557f25a47f48	2	3	2	0x557f25a47f90	2	0	2	0x557f25a483e8	101	121	79	0x557f25a4b860	1	0	0
	0x557f25a68ba0	2	0	0	0x557f25a68bb8	7	0	4	0x557f25a68ca0	5	0	3	0x55c2866db8b0	43	0	35
	0x55c2866db990	96	0	69	0x55c2866dc3e8	60	0	0	0x55c2866df840	8	11	0	0x55c2866df860	73	115	74
	0x55c2866df8c0	3	0	0	0x55c2866df8e0	65	65	53	0x55c2866fcbb8	1	7	2	0x55c2866fcca0	3	6	2
	0x55c4b7c8b8b0	10	0	9	0x55c4b7c8b980	4	0	4	0x55c4b7c8b990	10	0	6	0x55c4b7c8b9a8	5	0	3
	0x55c4b7c8bf48	2	0	1	0x55c4b7c8bf90	2	0	2	0x55c4b7c8c3e8	98	0	78	0x55c4b7c8f860	1	0	1
	0x55c4b7cacba0	1	0	0	0x55c4b7cacbb8	6	0	4	0x55c4b7cacca0	4	3	0	0xc42005e948	250	216	156
	0xc42005ed48	260	179	124	0xc42005f548	53	279	0	0xc42005f948	1	0	0	0xc42005fd48	1	0	0
	0xc420096548	101	0	0	0xc420096948	10	0	0	0xc420097148	9	0	0	0xc42016a148	14	0	0
	0xc42016a548	57	0	0	0xc420404548	72	0	0	0xc420464548	47	0	0	0xc420464948	12	0	0
	0xc42049b948	57	0	0	0x55e687f8e2a0	0	7	3	0x556aab5498e0	0	0	1	0x556aab566bb8	0	0	3
	0x556aab566ca0	0	0	3	0x5578ba7b98b0	0	0	8	0x5578ba7b9980	0	0	4	0x5578ba7b9990	0	0	6
	0x5578ba7b99a8	0	0	3	0x5578ba7b9f48	0	0	2	0x5578ba7b9f90	0	0	2	0x5578ba7ba3e8	0	0	82
	0x5578ba7bd8e0	0	0	1	0x557bdda3e980	0	0	2	0x557bdda3e9a8	0	0	2	0x557bdda3ef48	0	0	1
	0x55c52254fca0	0	0	3	0xc420022148	0	0	66	0xc42005d148	0	0	50	0xc4200921d8	0	0	1
	0xc420094d48	0	0	15	0xc420095148	0	0	25	0xc420095948	0	0	35	0xc420095d48	0	0	65
	0xc420098d48	0	0	28	0xc4201bc548	0	0	108	0xc4201bdd48	0	0	67	0xc4202b6148	0	0	65

Table VIII – Continued from previous page

App	Address	T1	T2	T3	Address	T1	T2	T3	Address	T1	T2	T3	Address	T1	T2	T3
	0xc420e8d948	0	0	10												
Hadoop	0x7f4c48013528	43	57	0	0x7f4c48013578	26	34	0	0x7f4c4801357c	24	32	0	0x7f4c48013728	4	7	0
	0x7f4c48013778	2	6	0	0x7f4c4801377c	2	4	0	0x7f4c48077a28	63	65	20	0x7f4c48077a78	31	35	20
	0x7f4c48077a7c	22	24	15	0x7f4c48081528	16	9	7	0x7f4c48081578	7	3	2	0x7f4c4808157c	6	6	1
	0x7f4c48083f28	18	9	0	0x7f4c48083f78	7	6	1	0x7f4c48083f7c	6	3	0	0x7f4c48084328	12	9	0
	0x7f4c48084378	4	4	0	0x7f4c4808437c	4	2	0	0x7f4c480b8f28	243	225	0	0x7f4c480b8f78	99	101	0
	0x7f4c480b8f7c	100	102	0	0x7f4c480b9128	3	3	1	0x7f4c480b9178	2	2	1	0x7f4c480bb228	282	293	15
	0x7f4c480bb278	102	107	11	0x7f4c480bb27c	102	107	9	0x7f4c480bb428	3	6	6	0x7f4c480bb478	2	2	5
	0x7f4c480be07c	1	0	2	0x7f4c480c3428	238	243	283	0x7f4c480c3478	238	243	283	0x7f4c4e2e1540	1	1	3
	0x7f4c4e2fa280	3	4	0	0x7f4c4ef3a9d0	1	1	4	0x7f8014000020	0	21	0	0x7f8014013228	0	4	0
	0x7f8014013278	0	5	0	0x7f801401327c	0	1	1	0x7f80140be078	0	3	0	0x7f80140d767c	0	6	0
	0x7f80140d7ac8	0	1	0	0x7f80141bd228	0	6	1	0x7f80141bd278	0	2	1	0x7f80141bd27c	0	2	0
	0x7f801d2e2280	0	4	0	0x7f801d6d8ba4	0	1	0	0x7f96840be57c	0	0	1				
	0x7f9685263328	0	0	1	0x7f9685263378	0	0	1								
Cassandra	0x7faff400d928	2	4	4	0x7faff400d978	2	2	2	0x7faff400d97c	2	2	2	0x7faff400db28	22	22	13
	0x7faff400db78	9	9	7	0x7faff400db7c	8	9	4	0x7faff4071a28	4	6	3	0x7faff4071a78	2	2	1
	0x7faff4071a7c	2	2	0	0x7faff407b628	1	3	0	0x7faff407b678	2	2	5	0x7faff407b67c	7	7	0
	0x7faff407ba28	3	1	0	0x7faff407ba78	2	2	0	0x7faff407df28	3	3	1	0x7faff407df78	2	2	2
	0x7faff407df7c	6	6	5	0x7faff407e328	1	1	0	0x7faff407e378	2	2	0	0x7faff40b0428	1	3	3
	0x7faff40b0478	2	2	2	0x7faff40b2528	7	7	9	0x7faff40b2578	4	4	4	0x7faff40b257c	3	3	3
	0x7faff40b4728	7	5	5	0x7faff40b4778	4	4	4	0x7faff40b477c	3	3	3	0x7faff40b7628	1	1	1
	0x7faff40b7678	2	2	2	0x7faff40b767c	1	1	1	0x7faff40ba028	2	4	4	0x7faff40ba078	2	2	2
	0x7faff40ba07c	2	2	2	0x7faff994abc0	6	5	5	0x7faffa7b89d0	9	8	5				
	0x55870f1194c8	1	1	0	0x55870f14d908	2	1	0	0x55870f175700	1	0	0	0x5587108d6f68	1	1	1
	0x55871098ef28	19	18	18	0x55871098ef78	19	18	19	0x5587109cb708	1	0	0	0x5587109cb758	2	0	0
	0x558710aff978	3	4	3	0x558710b16cd8	8	9	9	0x558710bc5f98	2	1	1	0x558710bc5f9c	1	0	0
	0x558710bc5fa0	3	0	0	0x558710bef1d8	3	0	0	0x558710ce2d28	18	18	17	0x558710ce2d78	19	19	18

Table VIII – Continued from previous page

App	Address	T1	T2	T3	Address	T1	T2	T3	Address	T1	T2	T3	Address	T1	T2	T3
MongoDB	0x558710eeb088	94	88	92	0x558710eeb0d8	85	84	83	0x558710eeb0dc	5	5	5	0x558710eebe88	21	21	21
	0x558710eebed8	22	22	22	0x558710eebf08	164	162	160	0x558710eebf58	165	163	161	0x558710eebf88	31	28	32
	0x558710eebfd8	32	29	33	0x558710ef14d0	164	146	160	0x558710ef1520	83	94	87	0x558710ef1524	84	71	75
	0x558710ef18d0	1	1	1	0x558710ef9860	1	1	1	0x558710ef98f0	17	17	16	0x558710ef9940	18	18	17
	0x558710efa898	11	12	12	0x558710efb198	9	9	10	0x55871451f008	3	1	3	0x55871451f058	2	2	2
	0x55871451f088	3	1	3	0x55871451f0d8	2	2	2	0x55871451f108	3	3	3	0x55871451f158	2	2	2
	0x55871451f188	3	3	3	0x55871451f1d8	2	2	2	0x55871451fe88	9	9	8	0x55871451fed8	10	10	9
	0x55871451ff08	3	3	3	0x55871451ff58	2	2	2	0x55871451ff88	3	3	3	0x55871451ffd8	2	2	2
	0x558714520008	3	3	3	0x558714520058	2	2	2	0x558714520088	3	3	3	0x5587145200d8	2	2	2
	0x558714520108	13	12	18	0x558714520158	13	12	18	0x7f2690d3c9d0	1	1	0	0x7f269153d9d0	1	1	0
	0x7f2691d3e9d0	1	1	0	0x7f26998fe0bc	1	1	0	0x7f26998fe124	1	1	0	0x7f26998fe214	1	1	0
	0x7f26998fe21c	1	1	0	0x563244a0b4c8	0	3	1	0x563248893088	0	3	3	0x5632488930d8	0	2	2
	0x5560c940f908	0	0	1	0x7f7f91bc09d0	0	0	1	0x7f7f923c19d0	0	0	1	0x7f7f92bc29d0	0	0	1
	0x7f7f9a7820bc	0	0	1	0x7f7f9a782124	0	0	1	0x7f7f9a782214	0	0	1				
ZooKeeper	0x7f8a34000e28	10	0	0	0x7f8a34000e78	11	0	0	0x7f8a34000e7c	1	0	0	0x7f8a7000cb28	24	5	0
	0x7f8a7000cb78	10	5	0	0x7f8a7000cb7c	10	3	0	0x7f8a7000cee8	1	1	1	0x7f8a70071428	25	21	21
	0x7f8a70071478	21	19	19	0x7f8a7007147c	3	1	1	0x7f8a7007af78	5	4	1	0x7f8a7007d97c	5	4	1
	0x7f8a700b2828	89	11	3	0x7f8a700b2878	36	7	3	0x7f8a700b287c	32	4	0	0x7f8a700b4b28	82	13	3
	0x7f8a700b4b78	34	7	3	0x7f8a700b4b7c	31	4	0	0x7f8a700b7878	5	2	1	0x7f8a7020b628	323	325	328
	0x7f8a7020b678	323	325	327	0x7f8a702bf328	1	1	10	0x7f8a702bf378	2	2	10	0x7f8a702cbf28	2	1	1
	0x7f8a702cbf78	2	0	0	0x7f8a702cbf7c	2	0	0	0x7f8a702cf528	1	10	0	0x7f8a702cf578	2	11	0
	0x7f8a702cf9e8	2	0	0	0x7f8a702d2d28	1	1	0	0x7f8a702d2d78	2	2	0	0x7f8a702d31b8	1	0	0
	0x7f8a789ad540	5	4	1	0x7f8a789c6280	3	0	0	0x7f8a796069d0	5	4	1	0x7f80d03262f8	0	2	1
	0x7f80d00c9d78	0	0	4	0x7f80d0329ca8	0	0	1								
	0x1e06c28	1	1	1	0x3e935b8	6	7	0	0x3e93608	3	3	0	0x3e9360c	2	3	0
	0x3e937d8	1	3	0	0x3e93828	2	2	0	0x3e93868	4	4	1	0x3e938b8	4	3	4
	0x3e938f8	6	6	0	0x3e93948	4	3	5	0x3e9394c	2	2	0	0x3e93988	2	6	0

Table VIII – Continued from previous page

App	Address	T1	T2	T3	Address	T1	T2	T3	Address	T1	T2	T3	Address	T1	T2	T3
MySQL	0x3e939d8	4	3	3	0x3e939dc	2	2	0	0x3e93a18	6	6	0	0x3e93a68	4	3	3
	0x3e93a6c	2	2	0	0x3e93aa8	3	3	9	0x3e93af8	1	1	10	0x3e93af8	1	1	0
	0x3e93b38	1	1	1	0x3e93b88	1	1	2	0x3e93bc8	1	1	0	0x3e93c18	2	2	2
	0x3e93c58	4	4	10	0x3e93ca8	5	6	6	0x3e93cac	1	1	10	0x3e93ce8	1	3	0
	0x3e93d38	2	2	2	0x41786f8	3	0	0	0x4505318	6	6	9	0x4505368	2	2	4
	0x450536c	2	2	2	0x4505be8	1	1	3	0x4505c38	7	3	2	0x45234a8	1	1	9
	0x45234f8	1	1	10	0x4526ff8	3	3	0	0x4527048	1	1	1	0x4527088	3	3	0
	0x45270d8	2	2	0	0x452ccb8	1	3	0	0x452cd08	2	2	0	0x45732d8	3	1	1
	0x4573328	1	1	2	0x4573368	1	0	0	0x45733b8	2	2	0	0x7fd1d13a89d0	1	1	0
	0x7fd1d9e0d1a0	1	1	1	0x7fd1da53207c	1	1	1	0x7fd1da532088	1	1	1	0x7fd1dad7c0c8	1	1	1
	0x7fff01d02c54	2	0	0	0x1dccce0	0	3	0	0x1ddc9e0	0	5	6	0x39ba428	0	7	10
	0x39ba478	0	4	4	0x39ba47c	0	2	4	0x39e14b8	0	3	0	0x39e1508	0	2	0
	0x7fb443d1a348	0	8	0	0x7ffdf1a2cdd4	0	2	0	0x1dcc748	0	0	3	0x1dcc760	0	0	3
	0x1dcc848	0	0	1	0x1dcc908	0	0	2	0x1dcc920	0	0	3	0x7fff17a2dbe4	0	0	2
PostgreSQL	0x7f4491bb7ba4	0	1	0	0x7f44937506ec	0	1	0	0x7f44937520b0	0	1	0	0x7f44937520bc	0	1	0
	0x7f4493752124	0	1	0	0x7f4493752214	0	1	0	0x7f449375221c	0	1	0	0x7f4493752230	0	1	0
	0x7f4493752248	0	1	0	0x7f4493752254	0	1	0	0x7f449375225c	0	1	0	0x7f449375226c	0	1	0
	0x7f4493752278	0	1	0	0x7f4493752340	0	1	0	0x7f4493752394	0	1	0	0x7f449375282c	0	1	0
	0x7f44939bd648	0	1	0	0x7f44939bd720	0	1	0	0x7f44939bd7f0	0	1	0	0x7f44939bd7fc	0	1	0
	0x7f44941ab370	0	1	1												

5.7 Summary

We addressed a new and challenging problem for cloud-based applications that results from spot instance revocations. We proposed a novel solution to automatically find Bugs of cloud-based Applications that result from Spot instance Revocations (BASIR) and to locate their causes in the source code. We developed our solution for Testing the BASIR (`T-BASIR`), and we evaluated it using 10 popular open-source applications. The results show that `T-BASIR` finds more instances of BASIR and different types of BASIR, such as performance bottlenecks, data loss and locked resources, and applications that cannot restart, compared to the Random approach. With `T-BASIR`, developers can analyze the traces of BASIR to improve the design of the shutdown process for cloud-based applications during their testing and, hence, to gain the advantage of cloud spot instances in the cloud. This enables stakeholders to economically deploy their applications on the cloud spot instances. To the best of our knowledge, `T-BASIR` is the first automated solution to find bugs of cloud-based applications resulting from spot instance revocations.

CHAPTER 6

PROVISIONING SPOT INSTANCES IN CLOUD MARKETS (P-SIWOFT)

In this chapter, we propose a novel cloud market-based approach that leverages features of cloud spot markets for Provisioning Spot Instances WithOut employing Fault-Tolerance mechanisms (P-SIWOFT) to reduce the deployment cost and completion time of applications.

6.1 Overview

Cloud computing offers a variable-cost payment scheme that allows cloud customers to specify the price they are willing to pay for renting spot instances to run their applications at much lower costs than fixed payment schemes, and depending on the varying demand from cloud customers, cloud platforms could revoke spot instances at any time. To alleviate the effect of spot instance revocations, applications often employ different fault-tolerance mechanisms to minimize or even eliminate the lost work for each spot instance revocation. However, these fault-tolerance mechanisms incur additional overhead related to application completion time and deployment cost. We propose a novel cloud market-based approach that leverages cloud spot market features to provision spot instances without employing fault-tolerance mechanisms to reduce the deployment cost and completion time of applications. We evaluate our approach in simulations and use Amazon spot instances that contain jobs in Docker containers and realistic price traces from EC2 markets. Our simulation results show that our approach reduces the deployment cost and completion time compared to approaches based on fault-tolerance mechanisms.

6.2 Introduction

Cloud computing offers a variable-cost payment scheme that allows cloud customers to specify the price they are willing to pay for renting spot instances to run their applications at much lower costs than fixed payment schemes, and depending on the varying demand from cloud customers, cloud platforms could revoke spot instances at any time. The price of a spot instance can go up if the demand increases and the number of available instances that can be supported by a finite number of physical resources in a data center of cloud providers decreases. Conversely, the price of this spot instance can go down if the demand decreases and the number of available instances increases. Therefore, if the customer's price is greater than the cloud provider's price that depends on the demand, a spot instance will be provisioned to cloud customers' applications at the customer's price. However, when spot instances are already provisioned to cloud customer applications and the cloud provider's price goes above the customer's price, the cloud providers will terminate those spot instances within two minutes by sending termination notification signals [2]. As a result, even though cloud customers sometimes rent spot instances at 90% lower prices than on-demand prices [15], their applications that run on spot instances can be terminated based on price fluctuations that happen frequently; thus, those applications may incur additional overhead related to application completion time and deployment cost (i.e., DCATO) from re-executing lost work for each spot instance revocation.

Applications may benefit from different fault-tolerance mechanisms to alleviate the work lost for each spot instance revocation. However, these fault-tolerance mechanisms incur additional overhead related to application completion time and deployment cost (i.e., DCATO). Fault-tolerance mechanisms are typically divided into three types: migration, checkpointing, and replication. First, migration mechanisms are often employed to reactively migrate the state of an application (i.e., memory and local disk state) to another instance prior to a spot instance revocation. The overhead of a migration mechanism is determined based on the migration time of an application and the number of spot instance revocations during the application execution. The migration time of an application mostly depends on the resource usage of the application, whereas the number of spot instance revocations depends on the volatility of cloud spot markets. A larger resource usage of an application often results in a higher overhead of a migration mechanism, and vice versa. A similar explanation is applicable for the volatility of cloud spot markets; thus, a higher overhead of a migration mechanism will lead to a higher overhead of an application's completion time and deployment cost. Second, checkpointing mechanisms are often employed to proactively checkpoint an application's state to remote storage (e.g., AWS S3). The overhead of a checkpointing mechanism is specified based on the time to checkpoint an application's state and the number of checkpoints, which represents how often an application's state is stored in remote storage during the application execution, along with the time to re-execute the lost work from the last checkpoint for each spot instance revocation. The checkpointing time of an application relies on the resource usage of the application and the number of checkpoints typically specified by engineers who maintain applications deployed on spot instances. If engineers specify a large number of checkpoints, the overhead time to re-execute the lost work from the last checkpoint for each spot instance revocation will likely decrease, whereas the overhead time to checkpoint the state of an application will likely increase. Conversely, if engineers specify a small number of checkpoints, the overhead time to checkpoint the state of an application will likely decrease, whereas the overhead time to re-execute the lost work from the last checkpoint for each spot instance revocation will likely increase. Hence, checkpointing mechanisms require analyzing cloud spot markets and the resource usage of applications to optimize the tradeoff between the overhead of actual checkpoints and the overhead of re-executing lost work. Third, replication mechanisms are often employed to replicate the computations of an application among different instances. The overhead of a replication mechanism is based on the degree of replication (i.e., the number of replicated instances) and the number of revocations that depends on the volatility of cloud spot markets, and is independent of the resource usage of an application. As a result, a higher overhead of these fault-tolerance mechanisms leads to a higher overhead related to application completion time and deployment cost (i.e., DCATO).

Contributions: We address a challenging problem for applications deployed on cloud spot instances that results from the overhead of employing fault-tolerance mechanisms. We propose a novel cloud market-based approach that leverages features of cloud spot markets for provisioning spot instances without employing fault-tolerance mechanisms (`P-SIWOFT`) to reduce the deployment cost and completion time of applications. We develop `P-SIWOFT` based on cloud spot market features, such as the spot instance lifetime, revocation probability, and revocation correlation between cloud spot markets and provision spot instances, without employing fault-tolerance mechanisms. We evaluate `P-SIWOFT` in simulations and use Amazon spot instances that contain jobs in Docker containers and realistic price traces from EC2 markets. Our simulation results show that our approach reduces the deployment cost and completion time compared to approaches based on fault-tolerance mechanisms.

6.3 Problem Statement

In this section, we discuss sources of overhead of fault-tolerance mechanisms, describe an illustrative example, and formulate the problem statement.

6.3.1 Sources of Overhead of Fault-Tolerance Mechanisms

There are three main sources of overhead of fault-tolerance mechanisms. First, various resource usage of an application imposes various overhead of fault-tolerance mechanisms depending on the settings of each fault-tolerance mechanism type. A larger resource usage of an application (i.e., memory footprint) often results in a higher overhead of a fault-tolerance mechanism, and vice versa. The time to migrate/checkpoint the state of an application depends on the sizes of the application's memory and local disk state. Additionally, the choice of the type of fault-tolerance mechanism depends on the resource usage of an application. For example, a live migration requires a limited size of an application's memory footprint and cannot be employed when the application's memory footprint is greater than 4 GB [79]. As a result, the resource usage of an application not only affects the overhead of a fault-tolerance mechanism but also affects the choice of the type of fault-tolerance mechanism.

Second, the volatility of cloud markets is represented by the number of spot instance revocations over the application runtime. A higher number of spot instance revocations often results in higher overhead of fault-tolerance mechanisms, and vice versa. Checkpointing mechanisms will re-execute the lost work from the last checkpoint for each spot instance revocation, whereas migration mechanisms will reactively migrate an application to another instance prior to each spot instance revocation. Unlike migration and checkpointing mechanisms, a replication mechanism might re-execute the lost work from the beginning of an application's runtime for each spot instance revocation when all replicated instances are being revoked. As a result, the volatility of cloud markets has an impact on the overhead of various types of fault-tolerance mechanisms.

Third, the overhead of fault-tolerance mechanisms relies on the settings of each type of fault-tolerance mechanism. A main parameter of replication settings is the degree of replication, which represents the number of replicated servers needed to execute the same application's job across these replicated servers. When the degree of replication is small, the overhead that results from re-executing the lost work from the beginning of an application's runtime for each spot instance revocation will likely increase. In contrast, when the degree of replication is large, the overhead that results from a high number of servers will likely increase. A main parameter of checkpointing settings is the number of checkpoints, which represents how often an application's state is stored in remote storage over the application runtime. When the number of checkpoints is small, the overhead that results from re-executing the lost work from the last checkpoint for each spot instance revocation will likely increase. In contrast, when the number of checkpoints is large, the overhead that results from the time to checkpoint an application's state will likely increase. A main parameter of migration settings is the number of migrations, which represents how often an application's state migrates to another server over the application runtime. When the number of migrations is small, the overhead that results from re-executing the lost work from the beginning of an application's runtime for each spot instance revocation will likely increase. In contrast, when the number of migrations is large, the overhead that results from the time to migrate an application's state will likely increase. As a result, the fundamental problem for cloud customers is determining how to find the optimal settings of various types of fault-tolerance mechanisms to reduce the overhead resulting from employing fault-tolerance mechanisms.

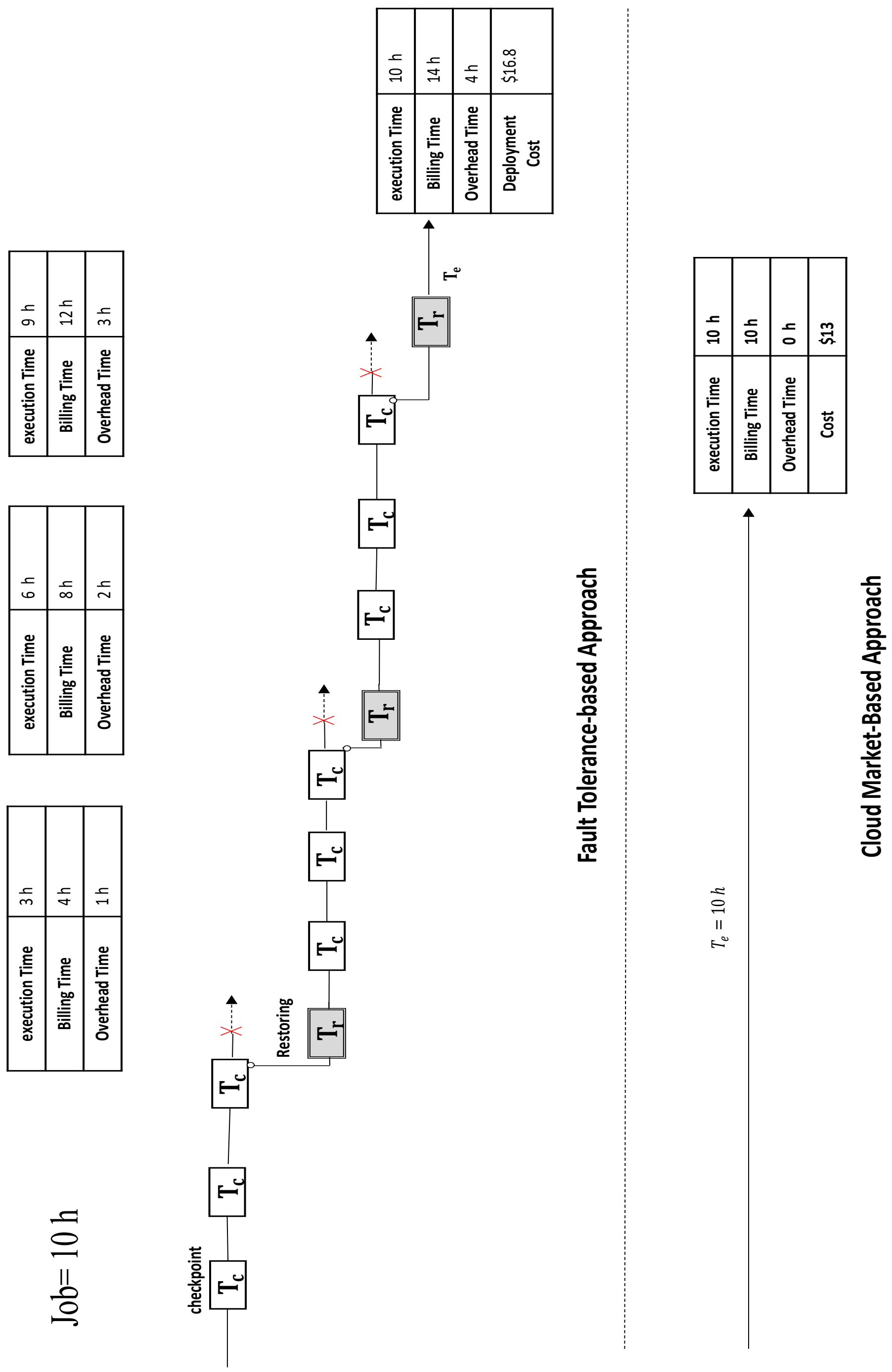


Figure 12: An illustrative example of P-SIWOFT.

6.3.2 An Illustrative Example

An illustrative example is shown in Figure 12. Applications deployed on cloud spot instances are often exposed to revocations by cloud providers, and as a result, these applications often employ various fault-tolerance mechanisms to alleviate the effect of spot instance revocations. However, these fault-tolerance mechanisms often incur additional overhead related to application completion time and deployment cost (i.e., DCATO). Our illustrative example shows a comparison of deployment costs for provisioning spot instances using a fault-tolerance approach and a cloud market-based approach (i.e., P-SIWOFT). Since cloud spot instances are often used to run batch job applications, we use a batch job application throughout the illustrative example to compute the deployment cost for provisioning spot instances using these approaches. As an example, we assume a cloud spot market contains three spot instances (i.e., VM1, VM2, and VM3) that meet the resource requirements for a batch job (i.e., a job of 10 h execution length and 64 GB of memory footprint). For ease of calculation, we assign a fixed price per hour for each spot instance throughout the entire job runtime. The prices of VM1, VM2, and VM3 are \$1.2, \$1.25, and \$1.3, respectively, and the lifetimes of VM1, VM2, and VM3 are 4, 16, and 24 h, respectively. First, we run the job using a fault-tolerance approach that employs a checkpointing mechanism and a cost-driven selection policy that selects a spot instance with the lowest price. To employ the checkpointing mechanism, we need to specify the number of checkpoints in a way that balances the overhead of actual checkpointing and the overhead of re-executing the lost work from the last checkpoint for each spot instance revocation. Since the deployment cost depends on the number of billing cycles, we specify the number of checkpoints for a job based on the number of billing cycles (i.e., a checkpoint is taken in each billing cycle). Suppose the time to checkpoint the state of a job to remote storage is five minutes and the time to restore a checkpoint from remote storage (i.e., recovery time) is also five minutes. Initially, VM1 will be selected based on the cost-driven selection policy to run the job until VM1 is revoked after four hours according to the lifetime of VM1. Additionally, a checkpoint will be taken/stored in each billing cycle (i.e., an hour based on the billing policies of various cloud computing platforms [15]). VM1 will complete executing three hours of the job and 15 min for storing three checkpoints before VM1 is revoked at its fourth hour of execution according to the lifetime of VM1, and there will be 45 min of lost work that was executed but not saved into remote storage (i.e., a checkpoint). Thus, the billing time is four hours, whereas the completed execution time of the job is three hours and the overhead time resulting from checkpoints and lost work is one hour. To resume the job execution, VM1 will again be selected based on the cost-driven selection policy; then, the last checkpoint will be restored, which takes five minutes, to resume the execution for another three hours plus 15 min for storing three checkpoints before this VM is revoked at its fourth hour of execution, and there will be 40 min of lost work that was executed but not saved in remote storage. Thus, the billing time increases by four hours to become eight hours, whereas the completed execution time of the job increases by three hours to become six hours in total and the overhead time increases by one hour to become two hours in total. Similarly, the next run will complete executing another three hours, 20 min for storing/restoring checkpoints, and 40 min of lost work. At this point, the billing time is 12 h, whereas the completed execution time of the job is nine hours and the overhead time is three hours. Again, VM1 will be selected, and the last checkpoint will be restored to resume the remaining execution of the job for the last hour; then, VM1 will be revoked due to the completion of the job execution. Since the last execution time is one hour and five minutes, the billing time will round up to two hours based on the billing policy that charges are counted per billing cycle (i.e., a complete hour). The billing time is 14 h, whereas the completed execution time of the job is 10 h and the overhead time is four hours. As a result, the total cost of executing this job using the fault-tolerance approach will be \$16.8.

Second, we run the job using a cloud market-based approach that uses the spot instance lifetime and a lifetime-driven selection policy that selects the spot instance with the highest lifetime. To reduce the revocation risk of this policy, we limit the selection of spot instances to instances whose lifetimes are at least twice the job's execution length. When using the cloud market-based approach, if a spot instance is revoked, the job will be re-executed from the beginning and the work before the revocation will be lost. When the job is executed using the cloud market-based approach, VM3 will be selected based on the lifetime-driven selection policy to execute the job until the job execution is completed or VM3 is revoked after 24 h according to the lifetime of VM3. VM3 will complete 10 h of the job execution and will be terminated before it is revoked according to the lifetime of VM3. Thus, the total cost of executing this job using the cloud market-based approach will be \$13. In summary, even though the fault-tolerance approach selects the most inexpensive VM in the cloud spot market to run the job, this approach leads to a higher deployment cost resulting from the overhead of the fault-tolerance approach (i.e., the checkpointing mechanism). On the other hand, the cloud market-based approach selects the most expensive VM in the cloud spot market but results in a lower deployment cost since this approach does not incur any additional overhead resulting from employing fault-tolerance mechanisms.

6.3.3 The Problem Statement

Cloud computing offers a variable-cost payment scheme that allows cloud customers to specify the price they are willing to pay for renting spot instances to run their applications at much lower costs than fixed payment schemes. In exchange, applications deployed on spot instances are often exposed to revocations by cloud providers, and as a result, these applications often employ different fault-tolerance mechanisms to minimize or even eliminate the lost work for each spot instance revocation. However, the overhead resulting from employing fault-tolerance mechanisms (i.e., periodic checkpointing) has become a very important concern for cloud customers (i.e., application owners). In this work, we address a challenging problem for applications deployed on cloud spot instances that results from the overhead of employing fault-tolerance mechanisms—determining how to effectively deploy applications on spot instances without employing fault-tolerance mechanisms to reduce the deployment cost and completion time of applications. The root of this problem is that applications often employ fault-tolerance mechanisms to minimize the lost work for each spot instance revocation without taking into consideration the overhead of fault-tolerance mechanisms, leading to significantly larger deployment costs and completion times of applications, and as a result, the advantages of cloud spot instances could be significantly minimized or even completely eliminated. To the best of our knowledge, there is no automated solution to provision spot instances without employing fault-tolerance mechanisms to reduce the deployment cost and completion time of applications.

6.4 Our Approach

In this section, we state our key ideas for `P-SIWOFT`, outline the architecture of `P-SIWOFT`, and explain the `P-SIWOFT` algorithm.

6.4.1 Key Ideas

A goal of our approach is to automatically provision spot instances without employing fault-tolerance mechanisms to reduce the deployment cost and completion time of applications. Our approach leverages features of cloud spot markets such as the spot instance lifetime, revocation probability, and revocation correlation between cloud spot markets to provision spot instances for applications. The spot instance lifetime represents the average time until a spot instance's price rises above the corresponding on-demand instance price (i.e., mean time to revocation (MTTR)) because cloud customers are often not willing to pay more than the on-demand price to rent spot instances. The revocation probability of each spot instance represents the estimated lifetime of a spot instance during a job execution and is calculated by dividing the job's execution length by the MTTR of the provisioned spot instance. The revocation correlation between cloud spot instances represents how often these spot instances were revoked at the same time (i.e., the same hour representing a single billing cycle in cloud platforms [15]) over the past three months.

In general, cloud spot markets show a broad range of characteristics. These important characteristics are at the core of our approach. First, revocations rarely happen in some cloud spot markets, so the MTTR of these markets is very high (i.e., $>$ 600 h) [149]. Second, employing fault-tolerance mechanisms often results in additional overhead related to application completion time and deployment cost [79]. Third, cloud spot markets exhibit variations in price characteristics for a similar type of spot instance across various cloud spot markets (i.e., availability zones and regions). Thus, a spot instance in a cloud market is often independent of a spot instance in another cloud market, which suggests that a spot instance's revocation in a cloud market is often uncorrelated with a spot instance in another cloud market [89]. Based on these characteristics, our key idea is that we could eliminate the additional overhead resulting from employing fault-tolerance mechanisms by provisioning the spot instance with the highest MTTR as long as the spot instance's MTTR is at least twice the application's execution length.

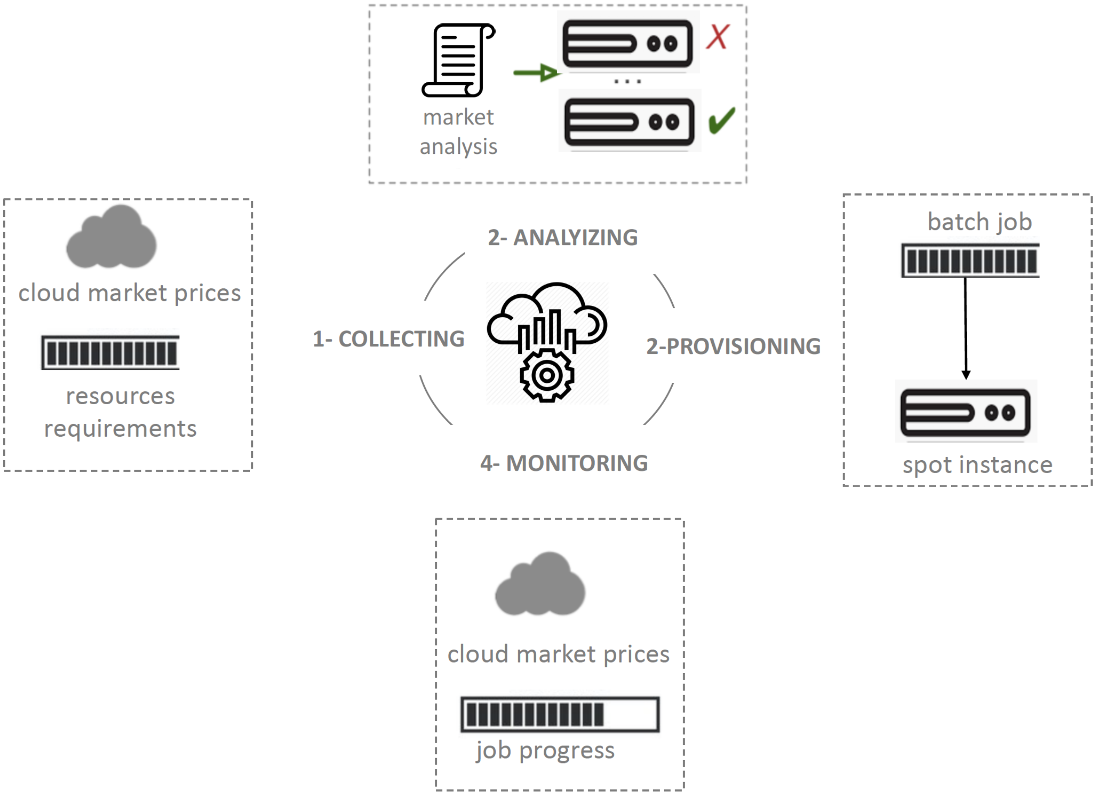


Figure 13: The architecture of `P-SIWOFT`.

Another idea is that we could reduce consequent revocations when a spot instance is revoked by provisioning a new spot instance with the next highest MTTR and a low revocation correlation with the revoked spot instance. When we provision a spot instance that is uncorrelated with the revoked spot instance, it is more unlikely that the new spot instance will be revoked again than another spot instance that is highly correlated with the revoked spot instance. As a result, these key ideas enable cloud customers to avoid unnecessary overhead resulting from employing fault-tolerance mechanisms; hence, cloud customers can execute jobs with a completion time near that of on-demand instances but at a cost of only spot instances.

6.4.2 Overview of P-SIWOFT

The architecture of P-SIWOFT is illustrated in Figure 13. Cloud market features are at the core of P-SIWOFT to provision spot instances for applications. Provisioning spot instances for applications based on cloud market features reduces the deployment cost of jobs compared to the deployment cost of jobs using a fault-tolerance approach or on-demand instances, in addition to maintaining a completion time near that of on-demand instances. There are four main phases in P-SIWOFT. 1) Collecting cloud market prices and the resource requirements for a job. Initially, P-SIWOFT uses EC2's REST API to collect cloud market prices for all instances (i.e., servers) across all markets (i.e., availability zones and regions) for the past three months. P-SIWOFT supports a predefined resource usage of a job to guide the selection of spot instances and assumes a job's resource usage does not change significantly (i.e., unphased jobs) over runtime. 2) Analyzing cloud spot market's features to identify a suitable spot instance for a job. P-SIWOFT first filters cloud spot markets to identify spot instances that satisfy the job's resource usage requirements and then computes the MTTR for each spot instance, the revocation probability for the job and a certain spot instance, and the revocation correlation between cloud spot instances. P-SIWOFT sorts the spot instances' MTTRs in descending order to provision the spot instance with the highest MTTR as long as the MTTR of the spot instance is at least twice the job's execution length. P-SIWOFT uses the revocation probability to determine when a spot instance might be revoked during its execution. Additionally, P-SIWOFT uses the revocation correlation between a pair of cloud spot instances when the provisioned spot instance is revoked to provision a new spot instance that is less correlated or even uncorrelated with the revoked spot instance to reduce the likelihood that the new spot instance will again be revoked over the job's runtime. 3) Provisioning a suitable spot instance for the job. P-SIWOFT uses the features of cloud spot markets and the resource requirements of spot instances to provision a suitable spot instance for a job. 4) Monitoring cloud market prices and the job execution progress over the job's execution. P-SIWOFT monitors cloud market prices to determine when a spot instance is revoked based on the revocation probability of the provisioned spot instance. When the provisioned spot instance is revoked, P-SIWOFT provisions a new spot instance with the next highest MTTR and a low revocation correlation with the revoked spot instance. P-SIWOFT also monitors the progress of the job execution to determine when the job execution is completed. Finally, our hypothesis is that leveraging cloud market features without employing fault-tolerance mechanisms to provision spot instances for applications reduces the deployment cost compared to the deployment cost using fault-tolerance approaches or on-demand instances and maintains the completion time near that of on-demand instances.

6.4.3 P-SIWOFT Algorithm

P-SIWOFT is illustrated in Algorithm 3 that takes in the batch job set J; the resource requirement set R; and the entire set of cloud markets M, containing on-demand instance types, prices of on-demand instances, spot instance types, their availability zones, their regions, and spot instance prices over the past three months. Starting from Step 2, the algorithm finds a suitable set of spot instances U that meet the resource requirements. In P-SIWOFT, we use the memory size to determine suitable types of spot instances that are supported by EC2 markets [15]. In Step 3, for each suitable spot instance, the spot instance's lifetime (i.e., the spot instance's MTTR) is computed based on the corresponding on-demand instance price, as discussed in Section 6.4.1. L is the set of such lifetimes. In Steps 4-20, for each job, the algorithm is executed until the jobs in the job set are completed. In Step 5, the cloud spot markets

Algorithm 3 `P-SIWOFT`'s algorithm for provisioning spot instances without employing fault-tolerance mechanisms.

1: **`Inputs:`** Jobs J, Cloud Markets M, Resources R
2: $U \leftarrow$ **`FindSuitableServers`**(J, R)
3: $L \leftarrow$ **`ComputeLifeTime`**(M, U)
4: **for** each j **in** J **do**
5: $\quad S_j \leftarrow$ **`ServerBasedLifeTime`**(j, M, L)
6: $\quad$ **while** $j \neg$ **`Completed`** **do**
7: $\quad\quad s_j \leftarrow$ **`Highest`**(S_j)
8: $\quad\quad$ **if** $length(s_j) >> length(j)$ **then**
9: $\quad\quad\quad \nu_{s_j} \leftarrow$ **`RevocationProbability`**(j, s_j)
10: $\quad\quad\quad$ **`ProvisionHighestLifeTime`**(j, s_j)
11: $\quad\quad\quad$ **if** s_j $encounters$ ν_{s_j} **then**
12: $\quad\quad\quad\quad C_j, T_j \leftarrow C_j \cup \{c_{s_j}\}, T_j \cup \{t_{s_j}\}$
13: $\quad\quad\quad\quad W_{s_j} \leftarrow$ **`FindLowCorrelation`**$(j, s_j))$
14: $\quad\quad\quad\quad S_j \leftarrow (S_j \setminus \{s_j\}) \cap W_{s_j}$
15: $\quad\quad\quad$ **end if**
16: $\quad\quad$ **end if**
17: $\quad$ **end while**
18: $\quad C_j, T_j \leftarrow C_j \cup \{c_{s_j}\}, T_j \cup \{t_{s_j}\}$
19: $\quad C, T \leftarrow$ **`ComputeCostExeTime`**(C_j, T_j)
20: **end for**
21: **return** C, T

are first filtered to include only a set of suitable spot instances S_j for the job j according to their lifetimes L, as discussed in Section 6.4.1, and then these spot instances are sorted in descending order based on their lifetimes. In Steps 6–17, the job j is executed until the job's execution is completed. In Step 7, the algorithm selects a spot instance s_j with the highest lifetime. In Step 8, we ensure that the highest lifetime for the spot instance s_j is at least twice the job j's execution length to reduce the revocation probability of the provisioned instance during the job execution. In Step 9, the algorithm computes the revocation probability of the provisioned instance ν_{s_j} by dividing the job j's execution length by the lifetime of the provisioned instance s_j. In Step 10, the spot instance s_j with the highest lifetime is provisioned to (re)start executing the job j. In Steps 11–15, the algorithm checks if the provisioned spot instance s_j is revoked based on its revocation probability ν_{s_j} during the job execution j. When a spot instance s_j is revoked, the deployment time t_{s_j} and cost c_{s_j} are added to the total deployment time set T_j and cost set C_j, respectively, in Step 12. In P-SIWOFT, the deployment time represents the job's execution time until the spot instance is revoked, the deployment cost of a spot instance represents the price of the provisioned spot instance at a certain execution point, and the cost is computed at a per hour rate (i.e., a single billing cycle in cloud platforms [15]). In Step 13, the low revocation correlation set W_{s_j} with the revoked spot instance is computed using the revocation correlation between cloud spot instances, as discussed in Section 6.4.1. In Step 14, the revoked spot instance is removed from the set of suitable spot instances S_j, and the set of suitable spot instances S_j is filtered based on a low revocation correlation set W_{s_j}. The cycle of Steps 6–17 repeats until the job j's execution is completed. When the job j's execution is completed, the deployment time t_{s_j} and cost c_{s_j} are added to the total deployment time set T_j and cost set C_j, respectively, in Step 18. In Step 19, the total deployment time set T_j and cost set C_j are computed and then added to the overall deployment time T and cost C, respectively. The cycle of Steps 4–20 repeats until the jobs in the job set are completed. Finally, the total deployment time T and cost C are returned in Step 21 as the algorithm ends.

6.5 Empirical Evaluation

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

***RQ*$_1$:** How efficient is P-SIWOFT compared to a fault-tolerance approach in executing applications?

***RQ*$_2$:** How effective is P-SIWOFT compared to a fault-tolerance approach in reducing the deployment cost of applications?

***RQ*$_3$: Do different settings of a fault-tolerance approach contribute to different types of overhead?**

6.5.1 Subject applications

We evaluate `P-SIWOFT` in simulations and use Amazon spot instances that contain jobs in Docker containers and realistic price traces from EC2 markets. `P-SIWOFT` packages jobs in Docker containers to simplify restoring and checkpointing. We use a load generator called Lookbusy [150] to create synthetic jobs with different amounts of resource usage. In addition, `P-SIWOFT` uses EC2's REST API to collect realistic price traces for all spot instances across all markets (i.e., availability zones and regions) for the past three months. We conduct some analysis on the collected cloud market prices to compute a spot instance's MTTR to identify the spot instance's lifetime based on its revocations over the past three months and to seed our `P-SIWOFT` for provisioning spot instances (i.e., `P-SIWOFT` looks for the spot instance with the highest MTTR to provision it for a job as long as the job's execution length is at least twice the MTTR of this spot instance). We also use the collected cloud market prices to compute the revocation correlation between cloud spot instances to identify how often a pair of spot instances were revoked at the same time (i.e., the same hour representing a single billing cycle [15]) over the past three months and to seed our `P-SIWOFT` for re-provisioning spot instances, i.e., `P-SIWOFT` looks for a spot instance that has a low revocation correlation with the revoked spot instance to reduce the revocation probability of the provisioned spot instance over the job's execution. In other words, when we provision a spot instance that is less correlated with the revoked spot instance, it is more unlikely that the new spot instance will be revoked again than another spot instance that is highly correlated with the revoked spot instance. As a result, our `P-SIWOFT` simulator utilizes these analyses of cloud markets to (re)provision spot instances without employing fault-tolerance mechanisms and hence reduces the deployment cost and completion time of applications.

6.5.2 Methodology

Some objectives of the experiments are to demonstrate that `P-SIWOFT` can efficiently execute applications and can effectively decrease the deployment cost of applications compared to a fault-tolerance approach. For these objectives, we use different combinations of job execution length (i.e., 13, 25, 51, and 101 h) and job memory footprint (i.e., 8, 16, 32, and 64 GB) to show the impact on the completion time and the deployment cost when a spot instance is provisioned for the job using `P-SIWOFT` and the fault-tolerance approach. We define two revocation rules with different ranges for `P-SIWOFT` and the fault-tolerance approach to show the impact on the completion time and the deployment cost for different numbers of revocations during a job's execution. When a spot instance is provisioned for a job using the fault-tolerance approach, we randomly send a fixed number of terminations (i.e., revocations) per day of the job's execution length, as suggested by prior work [79]. Conversely, when a spot instance is provisioned for a job using `P-SIWOFT`, we use the revocation probability of a spot instance that relies on realistic price traces from the Amazon cloud to revoke the provisioned spot instance. The deployment cost/completion time for `P-SIWOFT` is derived from the price/execution time of spot instances during the startup of a spot instance, the job's execution, and the job's re-execution after the provisioned spot instance is revoked. On the other hand, the deployment cost/completion time for the fault-tolerance approach is derived from the price/execution time of spot instances during the startup of a spot instance, the job's execution, the job's re-execution, the job's checkpointing, and the job's recovery (i.e., checkpoint restoring). Evaluating P-SIWOFT with different combinations of job settings (i.e., job execution length and job memory footprint) enables us to answer RQ_1 and RQ_2.

Since another goal is to understand how different settings of jobs and different settings of the fault-tolerance approach contribute to different types of overhead (e.g., checkpoint overhead), we investigate how different job execution lengths, job memory footprints, numbers of revocations, and numbers of checkpoints contribute to different overhead types that are related to a job's completion time and deployment cost (i.e., DCATO). In general, the time/cost overhead mainly falls into four categories: 1) the startup time/cost overhead that represents additional startup time/cost, which occurs when starting a new spot instance after each revocation; 2) the re-execution time/cost overhead that represents the lost work for each revocation (i.e., lost work using P-SIWOFT refers to unsaved and executed work from the beginning of a job, whereas lost work using the fault-tolerance approach refers to unsaved and executed work from the last checkpoint); 3) the checkpointing time/cost overhead that represents the time/cost to checkpoint a job's container into remote storage (i.e., AWS S3); 4) the recovery time/cost overhead that represents the time/cost to restore a checkpoint of a job's container from remote storage (i.e., AWS S3) into a container deployed on a spot instance for each revocation. Furthermore, the time overhead is divided into the startup time, the job's re-execution time, the job's checkpointing time, and the job's recovery time (i.e., checkpoint restoring time). The cost overhead is divided into the startup cost, the job's re-execution cost, the job's checkpointing cost, and the job's recovery cost (i.e., checkpoint restoring cost). Both P-SIWOFT and the fault-tolerance approach encounter the time/cost of startup overhead and the time/cost of re-execution overhead, whereas the time/cost of checkpointing overhead and the time/cost of recovery overhead are only encountered by the fault-tolerance approach. Understanding how different job settings and different settings of the fault-tolerance approach contribute to different types of overhead enables us to answer RQ_3.

P-SIWOFT is implemented using a load generator API (Lookbusy), EC2's REST API, Docker containers, AWS S3, and EC2 spot instances. The experiments for the subject applications were carried out using spot instances from Amazon EC2 called m5ad.12xlarge with a 48 GHz CPU and 192 GB of memory. We package jobs in Docker containers that run on Ubuntu 18.04 LTS with a limited CPU and memory capacity for the provisioned spot instances to assess the effectiveness of P-SIWOFT for different job memory footprints and job execution lengths. All experiments were performed on the same experimental platform to ensure a fair comparison between P-SIWOFT and the fault-tolerance approach. We used the following checkpointing settings: the number of checkpoints is equal to the number of billing cycles of a job's execution length because the deployment cost relies on the number of billing cycles instead of the actual completion time of the job.

6.5.3 Variables

The independent variables include the job execution length, i.e., the time required to complete the job execution; the job memory footprint, i.e., the size of a job's memory usage; the price of spot instances (i.e., price traces from EC2 markets); the price of on-demand instances; functions that describe instance selection policies depending on the cost or MTTR; the number of revocations based on the MTTR or user-defined rules, and the number of checkpoints, which represents how often the state of a job is stored in remote storage over the job runtime. The dependent variables include the deployment cost of instances provisioned for a job until the job's execution; the total runtime to execute a job in instances; types of overhead related to job completion time and deployment cost including startup, re-execution, checkpointing, and recovery overhead; and the time to restore/checkpoint a Docker container over a range of job settings (i.e., the job's memory footprint and the job's execution length).

6.5.4 Threads to validity

One potential threat to our empirical evaluation is that our experiments were conducted only on batch job applications, which may make it difficult to generalize the results of the experiments to other types of applications (e.g., interactive job applications) that may have various workflows and behaviors. However, cloud spot instances are often used to run batch job applications. As a result, we expect the results of the experiments to be generalizable.

Another threat to validity is that our experiments were performed in a simulation environment. While this is a potential threat, it is unlikely a major one since the average revocation time of spot instances in a cloud environment (e.g., Amazon EC2) often exceeds hundreds of hours, which makes it difficult to assess the effectiveness of `P-SIWOFT` for smaller job execution lengths that often reflect job execution lengths in production [151]. That is, we use realistic price traces from the Amazon cloud to define the revocation probability of spot instances for all spot instances across all markets (i.e., availability zones and regions) for the past three months. Additionally, we use a realistic time to restore/checkpoint a Docker container deployed on a spot instance in Amazon EC2 to seed our `P-SIWOFT`. For example, we measure the time to restore/checkpoint a Docker container that packages jobs with different job execution lengths and job memory footprints in/from S3 storage in Amazon EC2. Our experiments were performed only on Docker containers. While this is a potential threat, it is unlikely a major one since `P-SIWOFT` is perfectly applicable to other types of containers, such as Linux Containers, as long as those containers support checkpointing and restoring container images.

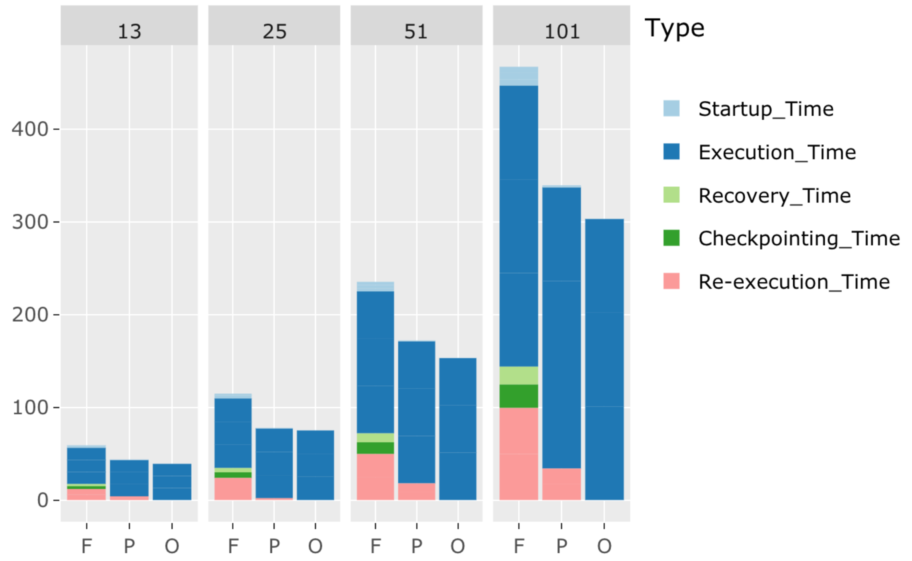
We experimented with a certain price ratio between spot instances and on-demand instances that is based on realistic price traces from EC2 markets, whereas other ratios between spot instances and on-demand instances could result in different effects on the deployment cost and completion time of jobs when spot instances are provisioned using P-SIWOFT and the fault-tolerance approach. However, understanding the effect of various price ratios between spot instances and on-demand instances is beyond the scope of this empirical study and shall be considered in future studies.

6.6 Empirical Results

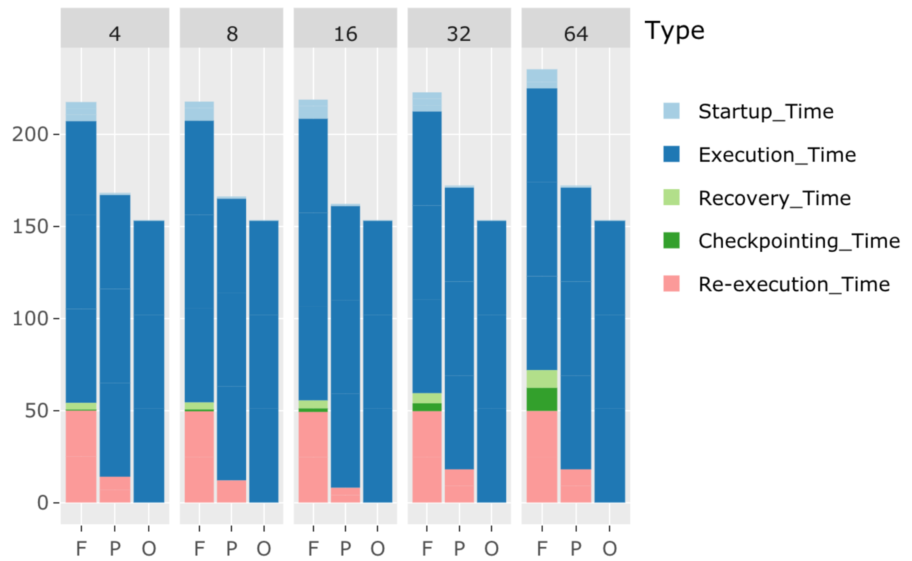
In this section, we describe and analyze the results of the experiments to answer the RQs listed in Section 6.5.

6.6.1 Completion Time

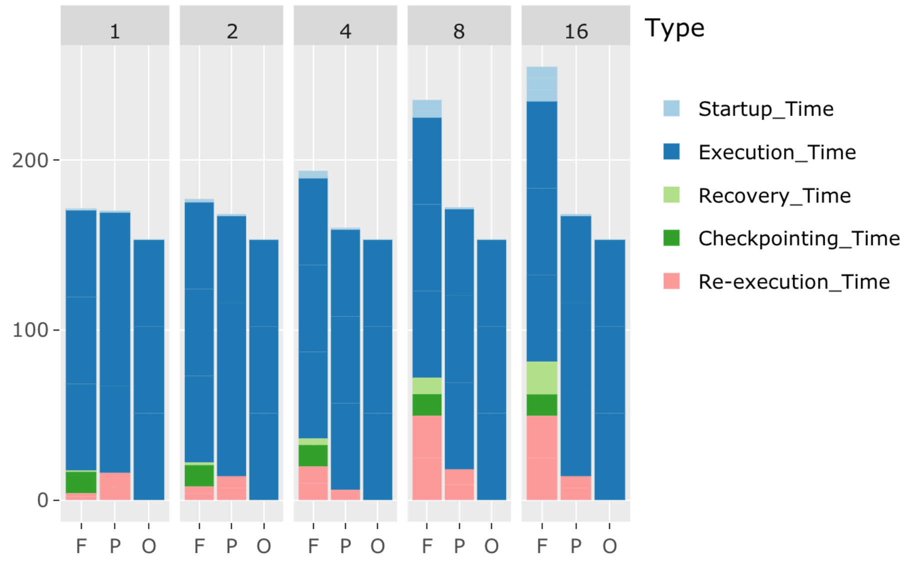
The experimental results that summarize the completion time for the subject applications using P-SIWOFT, the fault-tolerance approach, and on-demand instances for different job execution lengths are shown in the stacked bar plots in Figure. 14a. We observe that the completion time using P-SIWOFT is consistently shorter than the completion time using the fault-tolerance approach, and the completion time using P-SIWOFT is consistently near that of on-demand instances, which do not incur any additional overhead [15]. This result shows that a higher job length leads to a steadily higher overhead of completion time resulting from the job's checkpointing, recovery, and re-execution times, as well as the startup time of a spot instance when using the fault-tolerance approach. However, a higher job length leads to a slightly higher overhead of the completion time, as a result of the job's re-execution time and the startup time of a spot instance when using P-SIWOFT. Our explanation is that P-SIWOFT does not incur frequent job re-execution time and the startup time of a spot instance since the startup



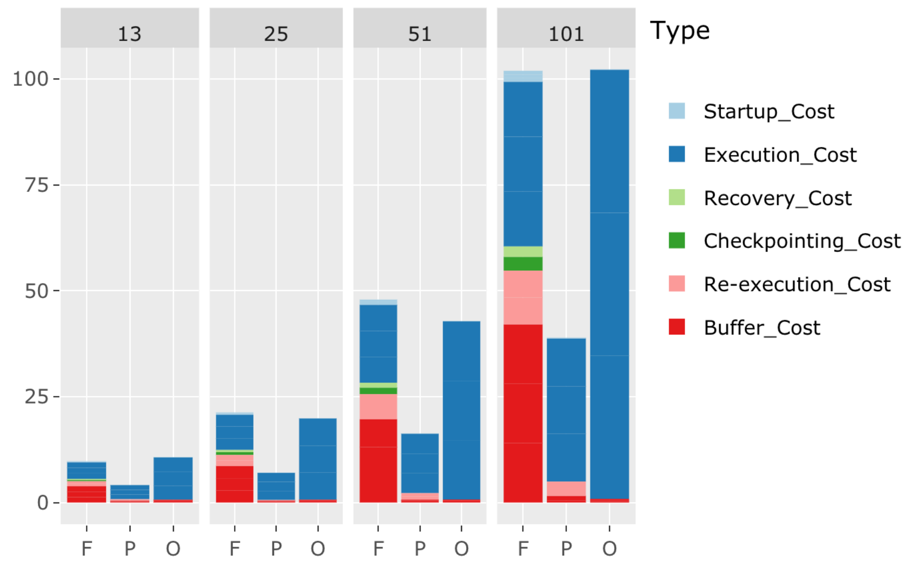
(a) Job Length (Time)



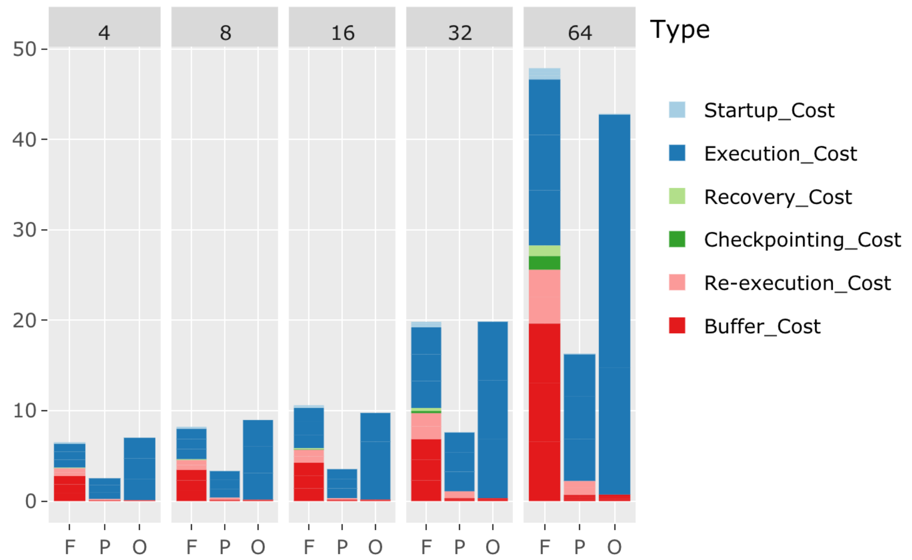
(b) Memory Footprint (Time)



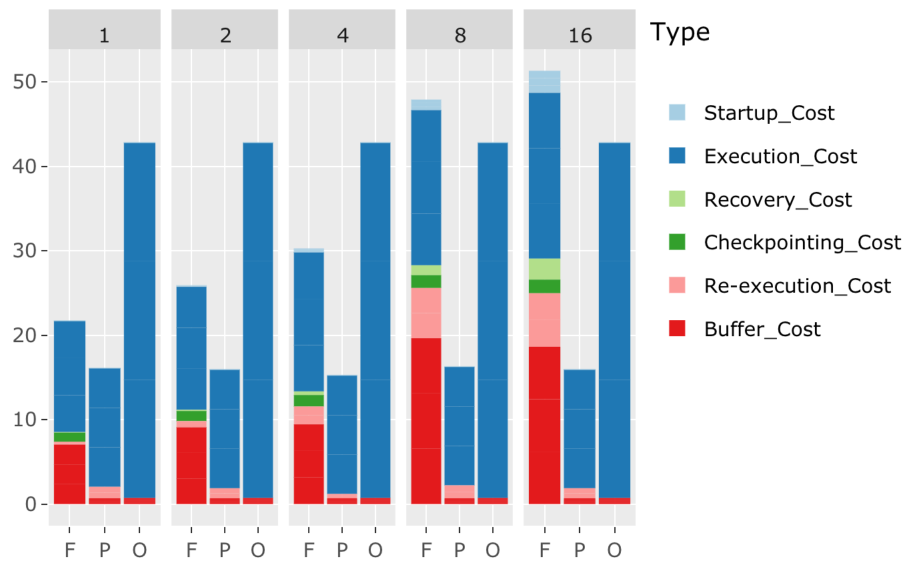
(c) Revocation Number (Time)



(d) Job Length (Cost)



(e) Memory Footprint (Cost)



(f) Revocation Number (Cost)

Figure 14: Comparing the completion time (top row) and the deployment costs (bottom row) for the subject applications using `P-SIWOFT` (P), the fault-tolerance approach (F), and on-demand instances (O) for different job execution lengths (a and d), memory footprints (b and e), and revocation numbers (c and f), while keeping other job features constant.

time of a spot instance using P-SIWOFT does not increase with the increase in job execution length. This is expected based on the way P-SIWOFT provisions a spot instance with the highest MTTR.

The experimental results that summarize the completion time for the subject applications using P-SIWOFT, the fault-tolerance approach, and on-demand instances for different job memory footprints are shown in the stacked bar plots in Figure. 14b. We observe that the completion time for P-SIWOFT is consistently shorter than the completion time for the fault-tolerance approach, and the completion time for P-SIWOFT is consistently near that of on-demand instances, which do not incur any additional overhead [15]. This result shows that a higher job memory footprint leads to a higher overhead of the completion time resulting from the job's checkpointing time and recovery time when using the fault-tolerance approach. In contrast, the overhead of the completion time resulting from the job's re-execution time and the startup time of a spot instance when using the fault-tolerance approach stays approximately the same across various job memory footprints, which suggests that the overhead resulting from the job's re-execution time and the startup time of a spot instance for the fault-tolerance approach is independent of the job resource usage. Also, the overhead of an application's completion time resulting from the job's re-execution time and the startup time of a spot instance when using P-SIWOFT stays approximately the same across various job memory footprints, which suggests that the completion time for the subject applications when using P-SIWOFT is also independent of the job resource usage.

The experimental results that summarize the completion time for the subject applications using P-SIWOFT, the fault-tolerance approach, and on-demand instances for different numbers of revocations are shown in the stacked bar plots in Figure. 14c. We observe that the completion time for P-

SIWOFT—except for when the number of revocations equals one—is consistently shorter than the completion time for the fault-tolerance approach, and the completion time for P-SIWOFT is consistently near that of on-demand instances, which do not incur any additional overhead [15]. When the number of revocations equals one, the job's checkpointing time for the fault-tolerance approach balances the job's re-execution for P-SIWOFT. This result suggests that the fault-tolerance approach incurs additional overhead due not only to the number of revocations, but also the number of checkpoints. It also suggests that the effectiveness of P-SIWOFT may decrease when the number of revocations decreases, and it is very difficult to guarantee that the number of revocations is small [129]. The job's recovery time, the job's re-execution time, and the startup time of a spot instance—except for the job's checkpointing time—all increase steadily when using the fault-tolerance approach, whereas in P-SIWOFT, the job's re-execution time and the startup time of a spot instance stay approximately the same. This observation suggests that the job's checkpointing time for the fault-tolerance approach as well as the job's re-execution time and the startup time of a spot instance for P-SIWOFT, are independent of the number of revocations. In summary, these experimental results allow us to conclude that P-SIWOFT is more efficient in executing applications for different job execution lengths, job memory footprints, and numbers of revocations than the fault-tolerance approach, thus **positively addressing *RQ*$_1$**.

6.6.2 Deployment Costs

The experimental results that summarize the deployment costs for the subject applications using P-SIWOFT, the fault-tolerance approach, and on-demand instances for different job execution lengths are shown in the stacked bar plots in Figure. 14d. We observe that the deployment costs using P-SIWOFT are consistently lower than the deployment costs using the fault-tolerance approach or those of on-demand instances. This result identifies the steady rise in overhead related to deployment costs that result from the job's checkpointing costs, its recovery costs, its re-execution costs, the startup costs of spot instances, and the buffer costs of billing cycles when using the fault-tolerance approach with the increased job length. However, this result also identifies a slight rise in the overhead of deployment costs that result from the job's re-execution cost, the startup costs of spot instances, and the buffer costs of billing cycles when using `P-SIWOFT` with the increased length. Our explanation is that `P-SIWOFT` does not frequently incur the job's re-execution costs and the startup costs of spot instances since the startup costs of spot instances using `P-SIWOFT` do not increase with the increase of the job execution length, which is expected based on the way that `P-SIWOFT` provisions a spot instance with the highest MTTR. Interestingly, we observe that unlike `P-SIWOFT`, the buffer costs of billing cycles significantly increase compared to the other types of overhead costs when using the fault-tolerance approach with the increase of the job length, which suggests that the fault-tolerance approach incurs not only overhead related to the settings of the fault-tolerance approach (e.g., the job's checkpointing cost) but also additional overhead related to the cloud billing policies (i.e., the buffer costs of billing cycles). Also, we observe that the deployment costs of the fault-tolerance approach across all job lengths are equal to or higher than the deployment costs of on-demand instances [15], which suggests using on-demand for larger job lengths may reduce deployment costs and the completion time when compared to the fault-tolerance approach.

The experimental results that summarize the deployment costs for the subject applications using `P-SIWOFT`, the fault-tolerance approach, and on-demand instances for different job memory footprints are shown in the stacked bar plots in Figure. 14e. We observe that the deployment costs using `P-SIWOFT` are consistently lower than the deployment costs using the fault-tolerance approach and on-demand instances. This result demonstrates the steady rise of the overhead related to deployment costs resulting from the job's checkpointing, recovery, re-execution, and startup costs of spot instances, as well as the buffer costs of billing cycles when using the fault-tolerance approach with the increase of job memory footprint. However, this result demonstrates a slight rise of the overhead of deployment costs resulting from the job's re-execution and startup costs of spot instances, and the buffer costs of billing cycles when using P−SIWOFT with the increase of job memory footprint. Our explanation is that P−SIWOFT does not incur the job's re-execution and startup costs of spot instances, since the startup costs of spot instances using P−SIWOFT do not increase with the increase of the job memory footprint, which is expected based on the way that P−SIWOFT provisions a spot instance with the highest MTTR. We observe that, unlike the buffer costs of billing cycles for P−SIWOFT, the buffer costs of billing cycles for the fault-tolerance approach significantly increase with the higher job memory footprints (i.e., 32 and 64 GB), suggesting that the buffer costs increase when there is a significant change in deployment time between consecutive job memory footprints (i.e., exceeds the period for a billing cycle). Additionally, we observe that the deployment costs of the fault-tolerance approach across all job memory footprints are equal or higher than the deployment costs of on-demand instances [15], which suggests provisioning on-demand for large job memory footprints may result in lower deployment costs and completion time than the fault-tolerance approach.

The experimental results that summarize the deployment costs for the subject applications using P−SIWOFT, the fault-tolerance approach, on-demand instances for different numbers of revocations are shown in the stacked bar plots in Figure. 14f. We observe that the deployment costs using P−

SIWOFT and that of on-demand instances are consistently lower than the deployment costs using the fault-tolerance approach. The job's recovery and re-execution costs, the startup costs of spot instances, and the buffer costs of billing cycles, except for the job's checkpointing costs, increase steadily when using the fault-tolerance approach whereas, for P-SIWOFT, the job's re-execution costs, the startup costs of spot instances, and the buffer costs of billing cycles stay approximately the same. This observation suggests that the job's recovery time and re-execution costs, the startup costs of spot instances, and the buffer costs of billing cycles depend on the number of revocations when using the fault-tolerance approach. However, the job's checkpointing costs for the fault-tolerance approach and the job's re-execution costs, the startup costs of spot instances, and the buffer costs of billing cycles for P-SIWOFT, are independent of the number of revocations, respectively. Our explanation is that P-SIWOFT does not incur the job's re-execution costs and the startup costs of spot instances. We observe that unlike the buffer costs of billing cycles for P-SIWOFT, the buffer costs of billing cycles for the fault-tolerance approach significantly increase with the higher numbers of revocations (i.e., 8 and 16 times per day), which suggests that the buffer costs increase when there is a significant change in deployment time between consecutive numbers of revocations (i.e., exceeds the period for a billing cycle). Interestingly, we observe that the deployment costs for the fault-tolerance approach when the number of revocations is high (i.e., 8 and 16 times per day) is significantly higher than the deployment costs for on-demand instances [15], which confirms that provisioning on-demand for a large number of revocations may result in lower deployment costs and completion time than the fault-tolerance approach. In summary, these experimental results allow us to conclude that P-SIWOFT is more effective in reducing the deployment costs of applications for different job execution lengths, job memory footprints, and numbers of revocations than the fault-tolerance approach, thus **positively addressing RQ_2**.

6.6.3 Impact on Different Types of Overhead

An interesting question is how different job execution lengths, job memory footprints, and numbers of revocations, contribute to different overhead types that are related to a job's completion time and deployment cost (i.e., DCATO) when using the fault-tolerance approach. Consider the stacked bar plots that are shown in Figure. 14a, Figure. 14b, and Figure. 14c — the visual inspection identifies the highest overhead related to the completion time results from the job's re-execution time, then the job's checkpointing time and the job's recovery time, followed by the startup time of a spot instance, with the increase of the job execution length. Also, with the rise of the job memory footprint, the highest overhead related to the completion time when using the fault-tolerance approach results from the job's checkpointing time and the job's recovery time. With the increase of the number of revocations, the highest overhead related to the completion time when using the fault-tolerance approach results from the job's re-execution time, then the job's recovery time, followed by the startup time of a spot instance.

Similarly, it is shown in the stacked bar plots in Figure. 14d, Figure. 14e, and Figure. 14f that the highest overhead related to the deployment costs when using the fault-tolerance approach results from the buffer costs of billing cycles, the job's re-execution costs, then the job's checkpointing cost, the job's recovery cost, followed by the startup costs of spot instances, with the increase of the job execution length. With the rise of the job memory footprint, the highest overhead related to the deployment costs when using the fault-tolerance approach results from the buffer costs of billing cycles, the job's re-execution costs, then the job's checkpointing and recovery costs, followed by the startup costs of spot instances. With the increase of the number of revocations, the highest overhead related to the deployment costs when using the fault-tolerance approach results from the buffer costs of billing cycles, the job's re-execution costs, then its recovery costs, followed by the startup costs of spot instances. The results confirm that different job execution lengths, job memory footprints, and numbers of revocations contribute to different overhead types related to a job's completion time and deployment cost (i.e., DCATO) when using the fault-tolerance approach, thus **positively addressing RQ_3**.

6.7 Summary

We addressed a challenging problem for applications deployed on cloud spot instances that results from the overhead of employing fault-tolerance mechanisms. We proposed a novel cloud market-based approach that leverages features of cloud spot markets for provisioning spot instances without employing fault-tolerance mechanisms (`P-SIWOFT`) to reduce the deployment cost and completion time of applications. We evaluated `P-SIWOFT` in simulations and used Amazon spot instances that contain jobs in Docker containers and realistic price traces from EC2 markets. Our simulation results show that our approach reduces the deployment cost and completion time compared to approaches based on fault-tolerance mechanisms. To the best of our knowledge, `P-SIWOFT` is the first approach that leverages cloud spot market's features to provision spot instances without employing fault-tolerance mechanisms to reduce the deployment cost and completion time of applications.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

Cloud computing provides key features of cloud platforms to enable customers to economically deploy their applications. First, customers can deploy their applications on a cloud infrastructure that provisions resources (e.g., memory) to these applications on as-needed basis. Second, customers can economically deploy their applications on cloud spot instances (i.e., virtual machines (VMs)) in cloud computing at much lower costs than that of other types of cloud instances.

In this thesis, we formulated challenging new problems that prevent cloud customers from deploying their applications in elastic clouds economically. First, we investigated situations when customers pay for resources that are provisioned, but not fully used by their applications, and as a result, some performance characteristics of these applications are not met, i.e., the Cost-Utility Violations of Elasticity (CUVE). Second, we investigated situations when applications that run in spot instances are being irregularly terminated due to spot instance revocations. These applications might lose their states that lead to certain bugs, i.e., Bugs of cloud-based Applications resulting from Spot Instance Revocations (BASIR). Third, we investigated situations when applications employ fault-tolerance mechanisms to minimize the lost work for each spot instance revocation. These applications incur additional overhead related to application completion time and deployment cost resulting from employing these fault-tolerance mechanisms, i.e., the Deployment Cost And Time Overhead (DCATO).

Therefore, we proposed a novel model that reduces the impact of CUVE, BASIR, and DCATO problems in the cloud environment to economically deploy applications in elastic clouds, and this model leads to practical frameworks for optimizing cloud elasticity, improving the design of the shutdown process, and reducing the deployment cost and completion time for cloud-based applications. This ensures efficient cloud computing services that lead to greater economies of scale.

Chapter 4 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 TICLEd 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.

Chapter 5 presented a novel solution to automatically find Bugs of cloud-based Applications that result from Spot instance Revocations (BASIR) and to locate their causes in the source code. We developed our solution for Testing the BASIR (T-BASIR), and we evaluated it using 10 popular open-source applications. The results show that T-BASIR finds more instances of BASIR and different types of BASIR, such as performance bottlenecks, data loss and locked resources, and applications that cannot restart, compared to the Random approach. With `T-BASIR`, developers can analyze the traces of BASIR to improve the design of the shutdown process for cloud-based applications during their testing and, hence, to gain the advantage of cloud spot instances in the cloud. This enables stakeholders to economically deploy their applications on the cloud spot instances. To the best of our knowledge, T-BASIR is the first automated solution to find bugs of cloud-based applications resulting from spot instance revocations.

Chapter 6 proposed a novel cloud market-based approach that leverages features of cloud spot markets for provisioning spot instances without employing fault-tolerance mechanisms (`P-SIWOFT`) to reduce the deployment cost and completion time of applications. We evaluated `P-SIWOFT` in simulations and used Amazon spot instances that contain jobs in Docker containers and realistic price traces from EC2 markets. Our simulation results show that our approach reduces the deployment cost and completion time compared to approaches based on fault-tolerance mechanisms. To the best of our knowledge, `P-SIWOFT` is the first approach that leverages cloud spot market's features to provision spot instances without employing fault-tolerance mechanisms to reduce the deployment cost and completion time of applications.

This thesis addressed CUVE, BASIR, and DCATO problems that prevent cloud customers from deploying their applications in elastic clouds economically. However, there are many other problems and challenges that need to be addressed to ensure efficient cloud computing operations. Here, we highlight two research directions to extend our work.

- **Building Reliable Applications against Revocations.** We plan to build revocation-robust applications in cloud spot markets to reduce the number of BASIR when these applications encounter irregular terminations. In particular, our goal is to optimize the design of the shutdown sequence for these applications using certain specifications that describe the shutdown sequence. These specifications can be defined based on the developers' recommendations of the found instances of BASIR or common design flaws in the applications' shutdown process (i.e., bug reports in code repositories, as discussed in Chapter 5). For example, applications should make their buffered writes short (i.e., reducing the dirty data buffer time), and applications should first flush the primary data of applications in volatile buffers and then flush the secondary data of these applications (e.g., log files) in volatile buffers. Also, partitions used in applications should be mounted as read-only and temporary remounted as read and write during the write operations. However, if an irregular termination occurs during the write operations, partitions should be remounted as read-only, which will likely force flushing volatile-buffers faster, and additional writes to volatile buffers should be blocked. Hence, closing files that are opened for writing may reduce the negative effect on these files due to irregular terminations, whereas files that are open for reading will likely not be affected by irregular terminations. In general, applications should operate based on the magnitude of the termination interval to determine whether buffered writes can be stored in permanent stores, and they should also indicate whether the shutdown process is completed successfully. Finally, although such specifications cannot guarantee BASIR-free applications, they will likely reduce the number of BASIR when these applications encounter irregular terminations.

- **Exploring the Impact of other Types of System Calls.** We plan to study the effect of I/O system calls that are responsible for reading/writing data from/to storage on applications when these applications are irregularly terminated during the execution of I/O system calls. For example, we will test the irregular termination of sync system calls that are responsible for synchronizing cached writes from volatile buffers to non-volatile buffers (i.e., persistent storage) to ensure that changes on volatile buffers are successfully flushed and committed to persistent storages on an irregular revocation. When sync system calls (i.e., fsync) are interrupted, due to irregular termination, we expect that cached writes in volatile buffers will likely not be committed to non-volatile storage, causing data loss. Another example of I/O system calls is write system calls. Let us suppose that concurrent write system calls are executed by separate processes/threads writing into a single buffer. However, consider what happens when one of these write system calls is interrupted by irregular termination. These processes/threads may not put all the data in a row, which causes data corruption.

CITED LITERATURE

1. Alourani, A., Bikas, M. A. N., and Grechanik, M.: Search-based stress testing the elastic resource provisioning for cloud-based applications. In Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings, volume 11036 of Lecture Notes in Computer Science, pages 149–165. Springer, 2018.

2. Alourani, A., Kshemkalyani, A. D., and Grechanik, M.: Testing for bugs of cloud-based applications resulting from spot instance revocations. In 12th IEEE International Conference on Cloud Computing, CLOUD 2019, Milan, Italy, July 8-13, 2019, pages 243–250. IEEE, 2019.

3. Alourani, A., Kshemkalyani, A. D., and Grechanik, M.: T-basir: Finding shutdown bugs for cloud-based applications in cloud spot markets. In IEEE Transactions on Parallel and Distributed Systems, 2020.

4. Peter Mell and Tim Grance: The NIST Definition of Cloud Computing. `http://csrc.nist.gov/groups/SNS/cloud-computing/cloud-def-v15.doc`, 2020.

5. Bikas, M. A. N., Alourani, A., and Grechanik, M.: How elasticity property plays an important role in the cloud: A survey. Advances in Computers, 103:1–30, 2016.

6. Mendelson, H.: Economies of scale in computing: Grosch's law revisited. Commun. ACM, 30(12):1066–1072, 1987.

7. vanderMeulen, R.: Gartner says by 2020 "cloud shift" will affect more than $1 trillion in it spending. `http://www.gartner.com/newsroom/id/3384720`, 2020.

8. Google: Autoscaling groups of instances. `https://cloud.google.com/compute/docs/autoscaler/`, 2020.

9. AWS: What is auto scaling? `http://docs.aws.amazon.com/autoscaling/latest/userguide/WhatIsAutoScaling.html`, 2020.

10. MSAzure: Autoscaling. `https://docs.microsoft.com/en-us/azure/architecture/best-practices/auto-scaling`, 2020.

11. Grechanik, M., Luo, Q., Poshyvanyk, D., and 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, pages 209–214. ACM, 2016.

12. Kim, S. and Ernst, M. D.: Which warnings should I fix first? In Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007, pages 45–54. ACM, 2007.

13. Johnson, B., Song, Y., Murphy-Hill, E. R., and Bowdidge, R. W.: Why don't software developers use static analysis tools to find bugs? In 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, pages 672–681. IEEE Computer Society, 2013.

14. Toosi, A. N., Vanmechelen, K., Ramamohanarao, K., and Buyya, R.: Revenue maximization with optimal capacity control in infrastructure as a service cloud markets. IEEE Trans. Cloud Computing, 3(3):261–274, 2015.

15. AWS: Amazon ec2 spot instances pricing. `https://aws.amazon.com/ec2/spot/pricing/`, 2020.

16. AWS: Amazon instance types. "`https://aws.amazon.com/ec2/instance-types/`", 2020.

17. GoogleCloud: Google preemptible instances. "`https://cloud.google.com/compute/docs/instances/preemptible`", 2020.

18. AWS: Amazon ec2 pricing. "`https://aws.amazon.com/ec2/pricing/`", 2020.

19. Agmon Ben-Yehuda, O., Ben-Yehuda, M., Schuster, A., and Tsafrir, D.: Deconstructing amazon EC2 spot instance pricing. ACM Trans. Economics and Comput., 1(3):16:1–16:20, 2013.

20. Barr, J.: New – ec2 spot instance termination notices. "`https://aws.amazon.com/blogs/aws/new-ec2-spot-instance-termination-notices/`", 2020.

21. MSAzure: Sizes for cloud services. `https://docs.microsoft.com/en-us/azure/cloud-services/cloud-services-sizes-specs`, 2020.

22. MSAzure: Sizes for windows virtual machines in azure. https://docs.microsoft.com/en-us/azure/virtual-machines/virtual-machines-windows-sizes?toc=%2fazure%2fvirtual-machines%2fwindows%2ftoc.json, 2020.

23. Alourani, A., Bikas, M. A. N., and Grechanik, M.: Input-sensitive profiling: A survey. Advances in Computers, 103:31–52, 2016.

24. Irwin, D. E., Grit, L. E., and Chase, J. S.: Balancing risk and reward in a market-based task service. In 13th International Symposium on High-Performance Distributed Computing (HPDC-13 2004), 4-6 June 2004, Honolulu, Hawaii, USA, pages 160–169. IEEE Computer Society, 2004.

25. Sharma, B., Thulasiram, R. K., Thulasiraman, P., and Buyya, R.: Clabacus: A risk-adjusted cloud resources pricing model using financial option theory. IEEE Trans. Cloud Computing, 3(3):332–344, 2015.

26. Columbus, L.: Roundup of cloud computing forecasts and market estimates, in forbes. ”https://www.forbes.com/sites/louiscolumbus”, 2020.

27. Gilbert, S. and Lynch, N. A.: Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM Special Interest Group on Algorithms and Computation Theory (SIGACT), 33(2):51–59, 2002.

28. Ghanbari, H., Simmons, B., Litoiu, M., and Iszlai, G.: Exploring alternative approaches to implement an elasticity policy. In IEEE International Conference on Cloud Computing, CLOUD 2011, Washington, DC, USA, 4-9 July, 2011, pages 716–723. IEEE Computer Society, 2011.

29. Gandhi, A., Dube, P., Karve, A. A., Kochut, A., and Zhang, L.: Adaptive, model-driven autoscaling for cloud applications. In 11th International Conference on Autonomic Computing, ICAC ’14, Philadelphia, PA, USA, June 18-20, 2014, pages 57–64. USENIX Association, 2014.

30. MSAzure: Microsoft azure. https://azure.microsoft.com, 2020.

31. Google: Google cloud platform. https://cloud.google.com, 2020.

32. Amazon: Amazon web services. http://aws.amazon.com, 2020.

33. Gambi, A., Hummer, W., and Dustdar, S.: Automated testing of cloud-based elastic systems with autocles. In 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013, pages 714–717. IEEE, 2013.

34. Breitgand, D., Henis, E., and Shehory, O.: Automated and adaptive threshold setting: Enabling technology for autonomy and self-management. In Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on, pages 204–215. IEEE, 2005.

35. Islam, S., Lee, K., Fekete, A., and Liu, A.: How a consumer can measure elasticity for cloud platforms. In Third Joint WOSP/SIPEW International Conference on Performance Engineering, ICPE'12, Boston, MA, USA - April 22 - 25, 2012, pages 85–96. ACM, 2012.

36. Harman, M.: Search based software engineering for program comprehension. In 15th International Conference on Program Comprehension (ICPC 2007), June 26-29, 2007, Banff, Alberta, Canada, pages 3–13. IEEE Computer Society, 2007.

37. Li, Z., Harman, M., and Hierons, R. M.: Search algorithms for regression test case prioritization. IEEE Trans. Software Eng., 33(4):225–237, 2007.

38. Mitchell, B. S. and Mancoridis, S.: Using heuristic search techniques to extract design abstractions from source code. In GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, New York, USA, 9-13 July 2002, pages 1375–1382. Morgan Kaufmann, 2002.

39. O'Keeffe, M. K. and Cinnéide, M. Ó.: Search-based software maintenance. In 10th European Conference on Software Maintenance and Reengineering (CSMR 2006), 22-24 March 2006, Bari, Italy, pages 249–260. IEEE Computer Society, 2006.

40. Frey, S., Fittkau, F., and Hasselbring, W.: Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, pages 512–521. IEEE Computer Society, 2013.

41. Xu, Z., Cohen, M. B., and Rothermel, G.: Factors affecting the use of genetic algorithms in test suite augmentation. In Genetic and Evolutionary Computation Conference, GECCO 2010, Proceedings, Portland, Oregon, USA, July 7-11, 2010, pages 1365–1372. ACM, 2010.

42. Memon, A. M., Porter, A. A., Yilmaz, C., Nagarajan, A., Schmidt, D. C., and Natarajan, B.: Skoll: Distributed continuous quality assurance. In 26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom, pages 459–468. IEEE Computer Society, 2004.

43. Harman, M., Jia, Y., and Langdon, W. B.: Strong higher order mutation-based test data generation. In SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011, pages 212–222. ACM, 2011.

44. Schwarz, B., Schuler, D., and Zeller, A.: Breeding high-impact mutations. In Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings, pages 382–387. IEEE Computer Society, 2011.

45. Panichella, A., Dit, B., Oliveto, R., Penta, M. D., Poshyvanyk, D., and Lucia, A. D.: How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, pages 522–531. IEEE Computer Society, 2013.

46. Ali, S., Briand, L. C., Hemmati, H., and Panesar-Walawege, R. K.: A systematic review of the application and empirical investigation of search-based test case generation. IEEE Trans. Software Eng., 36(6):742–762, 2010.

47. Alshahwan, N. and Harman, M.: Automated web application testing using search based software engineering. In 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011, pages 3–12. IEEE Computer Society, 2011.

48. Fraser, G., Arcuri, A., and McMinn, P.: A memetic algorithm for whole test suite generation. J. Syst. Softw., 103:311–327, 2015.

49. Harman, M. and McMinn, P.: A theoretical and empirical study of search-based testing: Local, global, and hybrid search. IEEE Trans. Software Eng., 36(2):226–247, 2010.

50. McMinn, P.: Search-based software test data generation: a survey. Softw. Test., Verif. Reliab., 14(2):105–156, 2004.

51. McMinn, P.: Search-based software testing: Past, present and future. In Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings, pages 153–163. IEEE Computer Society, 2011.

52. McMinn, P., Harman, M., Lakhotia, K., Hassoun, Y., and Wegener, J.: Input domain reduction through irrelevant variable removal and its effect on local, global, and hybrid search-based structural test data generation. IEEE Trans. Software Eng., 38(2):453–477, 2012.

53. Briand, L. C., Labiche, Y., and Shousha, M.: Stress testing real-time systems with genetic algorithms. In Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005, pages 1021–1028. ACM, 2005.

54. Baars, A. I., Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., Tonella, P., and Vos, T. E. J.: Symbolic search-based testing. In 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011, pages 53–62. IEEE Computer Society, 2011.

55. Iqbal, M. Z. Z., Arcuri, A., and Briand, L. C.: Empirical investigation of search algorithms for environment model-based testing of real-time embedded software. In International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012, pages 199–209. ACM, 2012.

56. Cain, H. W., Miller, B. P., and Wylie, B. J. N.: A callgraph-based search strategy for automated performance diagnosis (distinguished paper). In Euro-Par 2000, Parallel Processing, 6th International Euro-Par Conference, Munich, Germany, August 29 - September 1, 2000, Proceedings, volume 1900 of Lecture Notes in Computer Science, pages 108–122. Springer, 2000.

57. Wegener, J., Grimm, K., Grochtmann, M., Sthamer, H., and Jones, B.: Systematic testing of real-time systems. In 4th International Conference on Software Testing Analysis and Review (EuroSTAR 96), 1996.

58. Wang, S., Ali, S., Yue, T., Bakkeli, Ø., and Liaaen, M.: Enhancing test case prioritization in an industrial setting with resource awareness and multi-objective search. In Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume, pages 182–191. ACM, 2016.

59. Epitropakis, M. G., Yoo, S., Harman, M., and Burke, E. K.: Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015, pages 234–245. ACM, 2015.

60. Mao, K., Harman, M., and Jia, Y.: Sapienz: multi-objective automated testing for android applications. In Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016, pages 94–105. ACM, 2016.

61. Mondal, D., Hemmati, H., and Durocher, S.: Exploring test suite diversification and code coverage in multi-objective test case selection. In 8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015, pages 1–10. IEEE Computer Society, 2015.

62. Vásquez, M. L., Bavota, G., Bernal-Cárdenas, C. E., Oliveto, R., Penta, M. D., and Poshyvanyk, D.: Optimizing energy consumption of guis in android apps: a multi-objective approach. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015, pages 143–154. ACM, 2015.

63. Almhana, R., Mkaouer, W., Kessentini, M., and Ouni, A.: Recommending relevant classes for bug reports using multi-objective search. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016, pages 286–295. ACM, 2016.

64. Zaman, S., Adams, B., and Hassan, A. E.: A qualitative study on performance bugs. In 9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland, pages 199–208. IEEE Computer Society, 2012.

65. Jin, G., Song, L., Shi, X., Scherpelz, J., and Lu, S.: Understanding and detecting real-world performance bugs. In ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012, pages 77–88. ACM, 2012.

66. Chen, T., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M. N., and Flora, P.: Detecting performance anti-patterns for applications developed using object-relational mapping. In 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014, pages 1001–1012. ACM, 2014.

67. Zhang, P., Elbaum, S. G., and Dwyer, M. B.: Automatic generation of load tests. In 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011, pages 43–52. IEEE Computer Society, 2011.

68. Pradel, M., Schuh, P., Necula, G. C., and Sen, K.: Eventbreak: analyzing the responsiveness of user interfaces through performance-guided test generation. In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014, pages 33–47. ACM, 2014.

69. Nguyen, T. H. D., Nagappan, M., Hassan, A. E., Nasser, M. N., and Flora, P.: An industrial case study of automatically identifying performance regression-causes. In 11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India, pages 232–241. ACM, 2014.

70. Burnim, J., Juvekar, S., and Sen, K.: WISE: automated test generation for worst-case complexity. In 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings, pages 463–473. IEEE, 2009.

71. Bodík, P., Fox, A., Franklin, M. J., Jordan, M. I., and Patterson, D. A.: Characterizing, modeling, and generating workload spikes for stateful services. In Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010, pages 241–252. ACM, 2010.

72. Chen, F., Grundy, J. C., Schneider, J., Yang, Y., and He, Q.: Stresscloud: A tool for analysing performance and energy consumption of cloud applications. In 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2, pages 721–724. IEEE Computer Society, 2015.

73. Snellman, N., Ashraf, A., and Porres, I.: Towards automatic performance and scalability testing of rich internet applications in the cloud. In 37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011, Oulu, Finland, August 30 - September 2, 2011, pages 161–169. IEEE Computer Society, 2011.

74. Shen, D., Luo, Q., Poshyvanyk, D., and Grechanik, M.: Automating performance bottleneck detection using search-based application profiling. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015, pages 270–281. ACM, 2015.

75. Xiao, X., Han, S., Zhang, D., and Xie, T.: Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013, pages 90–100. ACM, 2013.

76. Song, Y., Zafer, M., and Lee, K.: Optimal bidding in spot instance market. In Proceedings of the IEEE INFOCOM 2012, Orlando, FL, USA, March 25-30, 2012, pages 190–198. IEEE, 2012.

77. Wolski, R., Brevik, J., Chard, R., and Chard, K.: Probabilistic guarantees of execution duration for amazon spot instances. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, Denver, CO, USA, November 12 - 17, 2017, pages 18:1–18:11. ACM, 2017.

78. Voorsluys, W. and Buyya, R.: Reliable provisioning of spot instances for compute-intensive applications. In IEEE 26th International Conference on Advanced Information Networking and Applications, AINA, 2012 , Fukuoka, Japan, March 26-29, 2012, pages 542–549. IEEE Computer Society, 2012.

79. Subramanya, S., Guo, T., Sharma, P., Irwin, D. E., and Shenoy, P. J.: Spoton: a batch computing service for the spot market. In Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015, pages 329–341. ACM, 2015.

80. Jia, Q., Shen, Z., Song, W., van Renesse, R., and Weatherspoon, H.: Smart spot instances for the supercloud. In Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms, CrossCloud@EuroSys 2016, London, United Kingdom, April 18-21, 2016, pages 5:1–5:6. ACM, 2016.

81. Zafer, M., Song, Y., and Lee, K.: Optimal bids for spot vms in a cloud for deadline constrained jobs. In 2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, June 24-29, 2012, pages 75–82. IEEE Computer Society, 2012.

82. Javadi, B., Thulasiram, R. K., and Buyya, R.: Statistical modeling of spot instance prices in public cloud environments. In IEEE 4th International Conference on Utility and Cloud Computing, UCC 2011, Melbourne, Australia, December 5-8, 2011, pages 219–228. IEEE Computer Society, 2011.

83. Harlap, A., Tumanov, A., Chung, A., Ganger, G. R., and Gibbons, P. B.: Proteus: agile ML elasticity through tiered reliability in dynamic resource markets. In Proceedings of the

Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017, pages 589–604. ACM, 2017.

84. Yi, S., Kondo, D., and Andrzejak, A.: Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud. In IEEE International Conference on Cloud Computing, CLOUD 2010, Miami, FL, USA, 5-10 July, 2010, pages 236–243. IEEE Computer Society, 2010.

85. Shastri, S. and Irwin, D. E.: Hotspot: automated server hopping in cloud spot markets. In Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017, pages 493–505. ACM, 2017.

86. Tang, S., Yuan, J., and Li, X.: Towards optimal bidding strategy for amazon EC2 cloud spot instance. In 2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, June 24-29, 2012, pages 91–98. IEEE Computer Society, 2012.

87. Wang, C., Liang, Q., and Urgaonkar, B.: An empirical analysis of amazon EC2 spot instance features affecting cost-effective resource procurement. ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS), 3(2):6:1–6:24, 2018.

88. Sharma, P., Lee, S., Guo, T., Irwin, D. E., and Shenoy, P. J.: Spotcheck: designing a derivative iaas cloud on the spot market. In Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015, pages 16:1–16:15. ACM, 2015.

89. Sharma, P., Irwin, D. E., and Shenoy, P. J.: Portfolio-driven resource management for transient cloud servers. In Proceedings of the 2017 ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, Urbana-Champaign, IL, USA, June 05 - 09, 2017, page 59. ACM, 2017.

90. Chelf, B., Engler, D. R., and Hallem, S.: How to write system-specific, static checkers in metal. In Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'02, Charleston, South Carolina, USA, November 18-19, 2002, pages 51–60. ACM, 2002.

91. Thummalapenta, S. and Xie, T.: Alattin: Mining alternative patterns for detecting neglected conditions. In ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009, pages 283–294. IEEE Computer Society, 2009.

92. Hassan, A. E.: Predicting faults using the complexity of code changes. In 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings, pages 78–88. IEEE, 2009.

93. PMD: An extensible cross-language static code analyzer. ”`http://pmd.sourceforge.net`”, 2020.

94. Li, Z. and Zhou, Y.: Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005, pages 306–315. ACM, 2005.

95. Kim, S., Zimmermann, T., Jr., E. J. W., and Zeller, A.: Predicting faults from cached history. In 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007, pages 489–498. IEEE Computer Society, 2007.

96. FindBugs: Findbugs documents and publications. ”`http://findbugs.sourceforge.net/publications.html`”, 2020.

97. Kremenek, T., Twohey, P., Back, G., Ng, A. Y., and Engler, D. R.: From uncertainty to belief: Inferring the specification within. In 7th Symposium on Operating Systems Design and Implementation (OSDI ’06), November 6-8, Seattle, WA, USA, pages 161–176. USENIX Association, 2006.

98. Giger, E., D’Ambros, M., Pinzger, M., and Gall, H. C.: Method-level bug prediction. In 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM ’12, Lund, Sweden - September 19 - 20, 2012, pages 171–180. ACM, 2012.

99. Huang, B., Jarrett, N. W., Babu, S., Mukherjee, S., and Yang, J.: Cümülön: Matrix-based data analytics in the cloud with spot instances. Proceedings of the VLDB Endowment, 9(3):156–167, 2015.

100. Sharma, P., Guo, T., He, X., Irwin, D. E., and Shenoy, P. J.: Flint: batch-interactive data-intensive processing on transient servers. In Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016, pages 6:1–6:15. ACM, 2016.

101. Khatua, S. and Mukherjee, N.: Application-centric resource provisioning for amazon EC2 spot instances. In Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings, volume 8097 of Lecture Notes in Computer Science, pages 267–278. Springer, 2013.

102. Zaman, S. and Grosu, D.: Efficient bidding for virtual machine instances in clouds. In IEEE International Conference on Cloud Computing, CLOUD 2011, Washington, DC, USA, 4-9 July, 2011, pages 41–48. IEEE Computer Society, 2011.

103. Menache, I., Shamir, O., and Jain, N.: On-demand, spot, or both: Dynamic resource allocation for executing batch jobs in the cloud. In 11th International Conference on Autonomic Computing, ICAC '14, Philadelphia, PA, USA, June 18-20, 2014, pages 177–187. USENIX Association, 2014.

104. Mazzucco, M. and Dumas, M.: Achieving performance and availability guarantees with spot instances. In 13th IEEE International Conference on High Performance Computing & Communication, HPCC 2011, Banff, Alberta, Canada, September 2-4, 2011, pages 296–303. IEEE, 2011.

105. Zheng, L., Joe-Wong, C., Tan, C., Chiang, M., and Wang, X.: How to bid the cloud. In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015, pages 71–84. ACM, 2015.

106. Liang, Q., Wang, C., and Urgaonkar, B.: Spot characterization: What are the right features to model. In International Workshop on System Analytics and Characterization (SAC), 2016.

107. Wolski, R. and Brevik, J.: Providing statistical reliability guarantees in the AWS spot tier. In Proceedings of the 24th High Performance Computing Symposium, Pasadena, HPC 2016, part of the 2016 Spring Simulation Multiconference, SpringSim '16, CA, USA, April 3-6, 2016, page 13. ACM, 2016.

108. Zhang, Q., Zhu, Q., and Boutaba, R.: Dynamic resource allocation for spot markets in cloud computing environments. In IEEE 4th International Conference on Utility and Cloud Computing, UCC 2011, Melbourne, Australia, December 5-8, 2011, pages 178–185. IEEE Computer Society, 2011.

109. Flor, S. A., Pires, F. L., and Barán, B.: A comparative evaluation of algorithms for auction-based cloud pricing prediction. In 2016 IEEE International Conference on Cloud Engineering,

IC2E 2016, Berlin, Germany, April 4-8, 2016, pages 99–108. IEEE Computer Society, 2016.

110. Engler, D. R., Chen, D. Y., and Chou, A.: Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP 2001, Chateau Lake Louise, Banff, Alberta, Canada, October 21-24, 2001, pages 57–72. ACM, 2001.

111. Wasylkowski, A., Zeller, A., and Lindig, C.: Detecting object usage anomalies. In Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007, pages 35–44. ACM, 2007.

112. Moser, R., Pedrycz, W., and Succi, G.: A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, pages 181–190. ACM, 2008.

113. Liu, Z. and Cho, S.: Characterizing machines and workloads on a google cluster. In 41st International Conference on Parallel Processing Workshops, ICPPW 2012, Pittsburgh, PA, USA, September 10-13, 2012, pages 397–403. IEEE Computer Society, 2012.

114. Beizer, B.: Software testing techniques. Dreamtech Press, 2003.

115. Albonico, M., Mottu, J., and Sunyé, G.: Controlling the elasticity of web applications on cloud computing. In Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016, pages 816–819. ACM, 2016.

116. Herbst, N. R., Kounev, S., and Reussner, R. H.: Elasticity in cloud computing: What it is, and what it is not. In 10th International Conference on Autonomic Computing, ICAC'13, San Jose, CA, USA, June 26-28, 2013, pages 23–27. USENIX Association, 2013.

117. Mian, R., Martin, P., Zulkernine, F. H., and Vázquez-Poletti, J. L.: Towards building performance models for data-intensive workloads in public clouds. In ACM/SPEC International Conference on Performance Engineering, ICPE'13, Prague, Czech Republic - April 21 - 24, 2013, pages 259–270. ACM, 2013.

118. Perez-Palacin, D., Mirandola, R., and 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, ICPE 2016 Companion, Delft, The Netherlands, March 12-16, 2016, pages 51–56. ACM, 2016.

119. Mao, M. and Humphrey, M.: A performance study on the VM startup time in the cloud. In 2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, June 24-29, 2012, pages 423–430. IEEE Computer Society, 2012.

120. crawler4j: Open source web crawler for java. `https://github.com/yasserg/crawler4j`, 2020.

121. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W.: Dynamo: amazon's highly available key-value store. In Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007, pages 205–220. ACM, 2007.

122. Zitzler, E. and Künzli, S.: Indicator-based selection in multiobjective search. In Parallel Problem Solving from Nature - PPSN VIII, 8th International Conference, Birmingham, UK, September 18-22, 2004, Proceedings, volume 3242 of Lecture Notes in Computer Science, pages 832–842. Springer, 2004.

123. Deb, K., Agrawal, S., Pratap, A., and Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evolutionary Computation, 6(2):182–197, 2002.

124. Halili, E.: Apache JMeter. Packt Publishing, 2008.

125. Durillo, J. J. and Nebro, A. J.: jmetal: A java framework for multi-objective optimization. Adv. Eng. Softw., 42(10):760–771, 2011.

126. Singh, R., Irwin, D. E., Shenoy, P. J., and Ramakrishnan, K. K.: Yank: Enabling green data centers to pull the plug. In Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013, pages 143–155. USENIX Association, 2013.

127. Burleson, D.: Fix hanging shutdown: waiting for active calls to complete. "`http://www.dba-oracle.com/t_hanging_shutdown_waiting_for_active_tasks_to_complete.htm`", 2020.

128. Mäkelä, M.: Move the innodb doublewrite buffer to flat files. "https://jira.mariadb.org/browse/MDEV-11659", 2020.

129. Shastri, S. and Irwin, D. E.: Cloud index tracking: Enabling predictable costs in cloud spot markets. In Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018, pages 451–463. ACM, 2018.

130. Mohan, J., Martinez, A., Ponnapalli, S., Raju, P., and Chidambaram, V.: Finding crash-consistency bugs with bounded black-box crash testing. In 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018, pages 33–50. USENIX Association, 2018.

131. Keniston, J.: The linux kernel documentation. "https://www.kernel.org/", 2020.

132. Ieee standard classification for software anomalies. IEEE Std 1044-1993, pages i–, 1994.

133. Armstrong, D. N., Kim, H., Mutlu, O., and Patt, Y. N.: Wrong path events: Exploiting unusual and illegal program behavior for early misprediction detection and recovery. In 37th Annual International Symposium on Microarchitecture (MICRO-37 2004), 4-8 December 2004, Portland, OR, USA, pages 119–128. IEEE Computer Society, 2004.

134. Juniper: Ex file system corruption. "https://kb.juniper.net/InfoCenter/index?page=content&id=KB20570", 2020.

135. Musubi, P.: Data loss on atom editor. "https://github.com/atom/atom/issues/11406", 2020.

136. Lebedev, D.: Data loss on xfs file system. "https://superuser.com/questions/84257/xfs-and-loss-of-data-when-power-goes-down", 2020.

137. Patry, C.: Data corruption on docker container. "https://github.com/scality/cloudserver/issues/662", 2020.

138. Duddridge, B.: Sqlite file corruption. "https://github.com/couchbase/couchbase-lite-ios/issues/1482", 2020.

139. Fay, R.: Database corruption on docker. "https://github.com/drud/ddev/issues/748", 2020.

140. Benny: Leveldb database corruption. ”https://github.com/google/leveldb/issues/733”, 2020.

141. thanhvtruong: Mosquitto database corruption. ”https://github.com/eclipse/mosquitto/issues/189”, 2020.

142. Frampton, S.: File system corruption after power outage or system crash. ”https://www.tldp.org/LDP/lame/LAME/linux-admin-made-easy/crash-repair.html”, 2020.

143. Moreno, L., Treadway, J. J., Marcus, A., and Shen, W.: On the use of stack traces to improve text retrieval-based bug localization. In 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014, pages 151–160. IEEE Computer Society, 2014.

144. Watson, D.: Libunwind documentation. ”https://www.nongnu.org/libunwind/docs.html”, 2020.

145. Franke, H., Russell, R., and Kirkwood, M.: Fuss, futexes and furwocks: Fast userlevel locking in linux. In AUUG Conference Proceedings, volume 85. AUUG, Inc. Kensington, NSW, Australia, 2002.

146. Our experimental data. ”https://www.dropbox.com/s/0z4qndkv6dwkzhu/T-BASIR.zip?dl=0”, 2020.

147. Wang, N. J., Fertig, M., and Patel, S. J.: Y-branches: When you come to a fork in the road, take it. In 12th International Conference on Parallel Architectures and Compilation Techniques (PACT 2003), 27 September - 1 October 2003, New Orleans, LA, USA, pages 56–66. IEEE Computer Society, 2003.

148. Akkary, H., Srinivasan, S. T., and Lai, K.: Recycling waste: exploiting wrong-path execution to improve branch prediction. In Proceedings of the 17th Annual International Conference on Supercomputing, ICS 2003, San Francisco, CA, USA, June 23-26, 2003, pages 12–21. ACM, 2003.

149. Sharma, P., Irwin, D. E., and Shenoy, P. J.: How not to bid the cloud. In 8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, June 20-21, 2016. USENIX Association, 2016.

150. Carraway, D.: lookbusy – a synthetic load generator. ”https://devin.com/lookbusy/”, 2020.

151. Google: Cluster workload traces. ”https://github.com/google/cluster-data”, 2020.

VITA

NAME	Abdullah Alourani
EDUCATION	Ph.D, Computer Science, University of Illinois at Chicago, Chicago, IL, USA, 2020 Advisor: Prof. Ajay D. Kshemkalyani M.S., Computer Science, DePaul University, Chicago, IL, USA, 2013 B.S., Computer Science, Qassim University, Qassim, Saudi Arabia, 2007
PUBLICATIONS	**Abdullah Alourani** and Ajay D. Kshemkalyani. "Provisioning Spot Instances Without Employing Fault-Tolerance Mechanisms." arXiv:2003.13846, 2020. [Online]. Available: https://arxiv.org/abs/2003.13846. **Abdullah Alourani**, Ajay D. Kshemkalyani, and Mark Grechanik. "T-BASIR: Finding Shutdown Bugs for Cloud-Based Applications in Cloud Spot Markets." In IEEE Transactions on Parallel and Distributed Systems (TPDS), 2020. [Online]. Available: https://doi.org/10.1109/TPDS.2020.2980265. **Abdullah Alourani**, Ajay D. Kshemkalyani, and Mark Grechanik. "Testing for Bugs of Cloud-Based Applications Resulting from Spot Instance Revocations." In 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pp. 243-250. IEEE, 2019. [Online]. Available: https://doi.org/10.1109/CLOUD.2019.00050. **This paper was awarded as Best Student Paper Award.** **Abdullah Alourani**, Md Abu Naser Bikas, and Mark Grechanik. "Search-Based Stress Testing the Elastic Resource Provisioning for Cloud-Based Applications." In International Symposium on Search-Based Software Engineering, pp. 149-165. Springer, Cham, 2018. [Online]. Available: https://doi.org/10.1007/978-3-319-99241-9_7. **Abdullah Alourani**, Md Abu Naser Bikas, and Mark Grechanik. "Input-Sensitive Profiling: A Survey." In Advances in Computers, vol. 103, pp. 31-52. Elsevier, 2016. [Online]. Available: https://doi.org/10.1016/bs.adcom.2016.04.002.

Md Abu Naser Bikas, **Abdullah Alourani**, and Mark Grechanik. “How elasticity property plays an important role in the cloud: a survey.” In Advances in Computers, vol. 103, pp. 1-30. Elsevier, 2016. [Online]. Available: `https://doi.org/10.1016/bs.adcom.2016.04.001`.

Md Abu Naser Bikas, **Abdullah Alourani**, and Mark Grechanik. “TestIng CLoud Elasticity.” In 5th Greater Chicago Area Systems Research(GCASR). University of Chicago,USA, 2016.(poster)

Md Abu Naser Bikas, **Abdullah Alourani**, and Mark Grechanik. “Enhancing Cloud Elasticity Via Software Performance Test Automation and Learning.” In 4th Greater Chicago Area Systems Research(GCASR). University of Chicago,USA, 2015.(poster)

Awards and Memberships

Best Student Paper Award: for the paper ”Testing for bugs of cloud-based applications resulting from spot instance revocations” at 12th International Conference on Cloud Computing (CLOUD), 2019

Scholarship for Doctorate Degree: Faculty scholarship program, Ministry of Higher Education, Saudi Arabia, 2014-2019

Scholarship for Master Degree: Faculty scholarship program, Ministry of Higher Education, Saudi Arabia, 2011-2013

Scholarship for Diploma in English Language: Faculty scholarship program, Ministry of Higher Education, Saudi Arabia, 2010-2011

Association for Computing Machinery (ACM): Learning society for computing, USA, 2007-Present

Institute of Electrical and Electronics Engineers (IEEE): Learning society for computing, USA, 2007-Present

Saudi Computer Society: Learning society for computing, Saudi Arabias, 2003-Present

Consent to Publish

Lecture Notes in Computer Science

Title of the Book or Conference Name: 10th International Symposium on Search-Based Software Engineering

Volume Editor(s) Name(s): Prof. Phil McMinn and Prof. Thelma Elita Colanzi Lopes

Title of the Contribution: Search-Based Stress Testing the Elastic Resource Provisioning for Cloud-Based Applications

Author(s) Full Name(s): Abdullah Alourani, Md Abu Naser Bikas, Mark Grechanik

Corresponding Author's Name, Affiliation Address, and Email:

University of Illinois at Chicago, Chicago, Illinois, 60607

aalour2,mbikas2,drmark@uic.edu

When Author is more than one person the expression "Author" as used in this agreement will apply collectively unless otherwise indicated.

The Publisher intends to publish the Work under the imprint **Springer**. The Work may be published in the book series **Lecture Notes in Computer Science (LNCS, LNAI or LNBI)**.

§ 1 Rights Granted

Author hereby grants and assigns to **Springer International Publishing AG, Gewerbestrasse 11, 6330 Cham, Switzerland** (hereinafter called **Publisher**) the exclusive, sole, permanent, world-wide, transferable, sub-licensable and unlimited right to reproduce, publish, distribute, transmit, make available or otherwise communicate to the public, translate, publicly perform, archive, store, lease or lend and sell the Contribution or parts thereof individually or together with other works in any language, in all revisions and versions (including soft cover, book club and collected editions, anthologies, advance printing, reprints or print to order, microfilm editions, audiograms and videograms), in all forms and media of expression including in electronic form (including offline and online use, push or pull technologies, use in databases and data networks (e.g. the Internet) for display, print and storing on any and all stationary or portable end-user devices, e.g. text readers, audio, video or interactive devices, and for use in multimedia or interactive versions as well as for the display or transmission of the Contribution or parts thereof in data networks or search engines, and posting the Contribution on social media accounts closely related to the Work), in whole, in part or in abridged form, in each case as now known or developed in the future, including the right to grant further time-limited or permanent rights. Publisher especially has the right to permit others to use individual illustrations, tables or text quotations and may use the Contribution for advertising purposes. For the purposes of use in electronic forms, Publisher may adjust the Contribution to the respective form of use and include links (e.g. frames or inline-links) or otherwise combine it with other works and/or remove links or combinations with other works provided in the Contribution. For the avoidance of doubt, all provisions of this contract apply regardless of whether the Contribution and/or the Work itself constitutes a database under applicable copyright laws or not.
The copyright in the Contribution shall be vested in the name of Publisher. Author has asserted his/her right(s) to be identified as the originator of this Contribution in all editions and versions of the Work and parts thereof, published in all forms and media. Publisher may take, either in its own name or in that of Author, any necessary steps to protect the rights granted under this Agreement against infringement by third parties. It will have a copyright notice inserted into all editions of the Work according to the provisions of the Universal Copyright Convention (UCC).
The parties acknowledge that there may be no basis for claim of copyright in the United States to a Contribution prepared by an officer or employee of the United States government as part of that person's official duties. If the Contribution was performed under a United States government contract, but Author is not a United States government employee, Publisher grants the United States government royalty-free permission to reproduce all or part of the Contribution and to authorise others to do so for United States government purposes. If the Contribution was prepared or published by or under the direction or control of the Crown (i.e., the constitutional monarch of the Commonwealth realm) or any Crown government department, the copyright in the Contribution shall, subject to any agreement with Author, belong to the Crown. If Author is an officer or employee of the United States government or of the Crown, reference will be made to this status on the signature page.

§ 2 Rights Retained by Author

Author retains, in addition to uses permitted by law, the right to communicate the content of the Contribution to other research colleagues, to share the Contribution with them in manuscript form, to perform or present the Contribution or to use the content for non-commercial internal and educational purposes, provided the original source of publication is cited according to the current citation standards in any printed or electronic materials. Author retains the right to republish the Contribution in any collection consisting solely of Author's own works without charge, subject to ensuring that the publication of the Publisher is properly credited and that the relevant copyright notice is repeated verbatim. Author may self-archive an author-created version of his/her Contribution on his/her own website and/or the repository of Author's department or faculty. Author may also deposit this version on his/her funder's or funder's designated repository at the funder's request or as a result of a legal obligation. He/she may not use the Publisher's PDF version, which is posted on the Publisher's platforms, for the purpose of self-archiving or deposit. Furthermore, Author may only post his/her own version, provided acknowledgment is given to the original source of publication and a link is inserted to the published article on the Publisher's website. The link must be provided by inserting the DOI number of the article in the following sentence: "The final authenticated version is available online at https://doi.org/[insert DOI]." The DOI (Digital Object Identifier) can be found at the bottom of the first page of the published paper.
Prior versions of the Contribution published on non-commercial pre-print servers like ArXiv/CoRR and HAL can remain on these servers and/or can be updated with Author's accepted version. The final published version (in pdf or html/xml format) cannot be used for this purpose. Acknowledgment needs to be given to the final publication and a link must be inserted to the published Contribution on the Publisher's website, by inserting the DOI number of the article in the following sentence: "The final authenticated publication is available online at https://doi.org/[insert DOI]".
Author retains the right to use his/her Contribution for his/her further scientific career by including the final published paper in his/her dissertation or doctoral thesis provided acknowledgment is given to the original source of publication. Author also retains the right to use, without having to pay a fee and without having to inform the Publisher, parts of the Contribution (e.g. illustrations) for inclusion in future work. Authors may publish an extended version of their proceedings paper as a journal article provided the following principles are adhered to: a) the extended version includes at least 30% new material, b) the original publication is cited, and c) it includes an explicit statement about the increment (e.g., new results, better description of materials, etc.).

§ 3 Warranties

Author agrees, at the request of Publisher, to execute all documents and do all things reasonably required by Publisher in order to confer to Publisher all rights intended to be granted under this Agreement. Author warrants that the Contribution is original except for such excerpts from copyrighted works (including illustrations, tables, animations and text quotations) as may be included with the permission of the copyright holder thereof, in which case(s) Author is required to obtain written permission to the extent necessary and to indicate the precise sources of the excerpts in the manuscript. Author is also requested to store the signed permission forms and to make them available to Publisher if required.
Author warrants that Author is entitled to grant the rights in accordance with Clause 1 "Rights Granted", that Author has not assigned such rights to third parties, that the Contribution has not heretofore been published in whole or in part, that the Contribution contains no libellous or defamatory statements and does not infringe on any copyright, trademark, patent, statutory right or proprietary right of others, including rights obtained through licences; and that Author will indemnify Publisher against any costs, expenses or damages for which Publisher may become liable as a result of any claim which, if true, would constitute a breach by Author of any of Author's representations or warranties in this Agreement.
Author agrees to amend the Contribution to remove any potential obscenity, defamation, libel, malicious falsehood or otherwise unlawful part(s) identified at any time. Any such removal or alteration shall not affect the warranty and indemnity given by Author in this Agreement.

§ 4 Delivery of Contribution and Publication

Author agrees to deliver to the responsible Volume Editor (for conferences, usually one of the Program Chairs), on a date to be agreed upon, the manuscript created according to the Publisher's Instructions for Authors. Publisher will undertake the reproduction and distribution of the Contribution at its own expense and risk. After submission of the Consent to Publish form signed by the Corresponding Author, changes of authorship, or in the order of the authors listed, will not be accepted by the Publisher.

§ 5 Author's Discount for Books and Electronic Access

Author is entitled to purchase for his/her personal use (if ordered directly from Publisher) the Work or other books published by Publisher at a discount of 40% off the list price for as long as there is a contractual arrangement between Author and Publisher and subject to applicable book price regulation.
Resale of such copies is not permitted.

§ 6 Governing Law and Jurisdiction

If any difference shall arise between Author and Publisher concerning the meaning of this Agreement or the rights and liabilities of the parties, the parties shall engage in good faith discussions to attempt to seek a mutually satisfactory resolution of the dispute. This agreement shall be governed by, and shall be construed in accordance with, the laws of Switzerland. The courts of Zug, Switzerland shall have the exclusive jurisdiction.

Corresponding Author signs for and accepts responsibility for releasing this material on behalf of any and all Co-Authors.

Signature of Corresponding Author: **Date:**

Abdullah Alourani .... Jun 23 2018 ....

☐ I'm an employee of the US Government and transfer the rights to the extent transferable (Title 17 §105 U.S.C. applies)

☐ I'm an employee of the Crown and copyright on the Contribution belongs to the Crown

For internal use only:
Legal Entity Number: 1128 Springer International Publishing AG
Springer-C-CTP-01/2018

IEEE

IEEE SERVICES

IEEE computer society

IEEE International Conference on Cloud Computing (CLOUD 2019)

2019 IEEE World Congress on Services

July 8-13, 2019 Milan, Italy

http://conferences.computer.org/services/2019/

To whom It May Concern:

On behalf of the Organizing Committee, this document is to certificate that Abdullah Alourani presented the following paper in the IEEE International Conference on Cloud Computing (CLOUD 2019) and inclusion of this paper in his thesis is permitted:

Abdullah Alourani, Ajay D. Kshemkalyani and Mark Grechanik. "Testing for Bugs of Cloud-Based Applications Resulting from Spot Instance Revocations," In 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pp. 243-250. IEEE, 2019.

If you have any more questions, please feel free to contact me at oyama.katsunori@nihon-u.ac.jp.

Sincerely,

Kat Oyama

Katsunori Oyama
Nihon University
Publication Chair of IEEE SERVICES 2019