Phase Aware Performance Modeling for Cloud Applications

Arnamoy Bhattacharyya
University of Toronto
arnamoy10@gmail.com

Cristiana Amza
University of Toronto
amza@ece.utoronto.ca

Eyal de Lara
University of Toronto
delara@cs.toronto.edu

***Abstract*—In this paper we propose a new methodology for performance modeling of applications deployed in the cloud based on automatically discovered *phases* along with their inputs. Our method is based on lightweight sampling that can predict the performance of applications with up to 95% accuracy for previously *unseen* input configurations at less than 5% overhead. We show the effectiveness of the performance modeling methodology in case of anomaly detection for a variety of real world workloads. As compared to the state-of-the-art, our method gives significant improvements in reducing both false positives and false negatives for anomalous test cases.**

***Index Terms*—Performance Modeling, Cloud Computing, Anomaly Detection**

I. Introduction

The use of Cloud environments is becoming increasingly popular due to the ability to provide high quality of service for millions of users, while taking the burden of designing and deploying an expensive hardware platform away from them. Performance modeling of cloud applications is necessary for both resource provisioning and anomaly diagnosis towards efficient management of cloud systems. Accurate performance or resource usage prediction for any given workload is difficult to achieve. An application may have several distinct phases with radically different performance or resource usage profiles. It is usually not feasible for the user to explicitly specify anything more than coarse grained upper or lower bounds for the expected application resource needs or other behavioral expectations.

In this paper, we introduce a novel methodology that can model and predict the behavior of application workloads automatically. Our method recognizes the application modeling granularity (phase) and the inputs that matter e.g., for performance, on the fly, without user specified information or other annotations. A *phase* is defined as a time slice in the application's life when it executes a stable set of functions using a stable number of threads, resulting in stable resource utilization. The per-phase application characterization is performed using periodic thread dumps in order to capture application stacktraces to characterize the phase. The relevant inputs (meta-inputs) to each phase are identified automatically by correlating the resource utilization and execution time behavior for the previous phases of the current application run.

Our approach is non-intrusive, does not require application source code and is offered online, life-long, for all applications, with the cloud service provided. Our experimental evaluation shows that our approach significantly improves the modeling accuracy (up to 10% improvement with a prediction accuracy of 95%) of traditional alternatives (RNN, ARIMA) [1]–[4] but transparently, without requiring any user input and at negligible, less than 5%, overhead. Our performance modeling approach can be used for anomaly detection, where it is able to reduce the number of false positives (normal runs flagged as anomalous) and false negatives (anomalies identified as normal execution) up to 60%.

II. Granularity of Performance Modeling: Phases

In this section, we discuss our first contribution, which is finding a proper granularity for performance modeling with the aim to achieve a better prediction accuracy. Our granularity of modeling is per *phase* of an application. An application running on a cloud goes through separate *phases* as workloads are run on it. For example on a Spark [5] deployment, semantically, a `wordcount` workload may show phases for: (a) reading the file (b) counting the words from the file (c) saving the output to a different file. Each of these phases can have a single function or a set of functions executed by a number of threads, performing the desired work, therefore resulting in a particular resource utilization. Therefore modeling at the phase granularity rather than the granularity of the entire run is a better choice for resource utilization prediction. We formally define a phase of an application as the following:

Definition 2.1: A phase $\mathbb{P}$ of an application is a time slice during the application life-cycle when the application executes a stable set of functions following a particular distribution by a stable number of threads.

By *stable*, we mean an entity (a set or a real number) whose value is not statistically significantly different from its last seen values.

A. Application Phases

For detecting the phase boundaries, we sample periodic *threaddumps* for the application, at 1HZ frequency (determined experimentally to trade-off prediction accuracy and throughput drop). Then we aggregate an *epoch* of n samples to extract statistical features from the threaddump data. We develop an adaptive way of finding the epoch size tuned per application, by mining the application trace generated by

appending the periodically collected threaddumps. In the next section, we describe the mining technique in details.

2.1: Identifying exact patterns: Long running functions and/or loops in the workload generate repeated stacktraces in the application trace. Our method for identifying such phases (exact repeats) is based on identifying *all non-overlapping repeated substrings* from a long string (the trace) using the method described by Crochemore et al. [6] in linear time. The epoch size for exact matches is the number of threaddumps that are needed to be collected for discovering the match.

2.2: Identifying fuzzy patterns: There are workloads that have a mixture of functions [7] or follow a specific distribution [8]. Capturing stacktraces from such workloads may not generate exact repeats in the trace, but fuzzy repetitions. The input to our function *distribution* (fuzzy repeats) finding algorithm is the trace *after* removing exact matches found in the previous step. Our goal is to identify time-slices within the trace that have similar distributions. We perform a greedy search to find such phases by sliding a window across the trace, starting with bigger window sizes and gradually evaluating smaller windows. For matching a couple of windows, we develop the following two metrics:

$$f_{sim} = \frac{|(T_{test} \bigcap T_{ref})|}{max(|T_{test}|, |T_{ref}|)} \quad (1)$$

$$t_{sim} = \sum_{i=1}^{n} \left(\sqrt{H_{cur}} - \sqrt{H_{next}}\right)^2 \quad (2)$$

f_{sim} represents the fraction of common functions between the two windows under comparison and t_{sim} calculates the similarity in the distribution of the set of common functions, using the Squared-Chord distance [9]. We consider a match when both values of f_{sim} and t_{sim} are ≥ 0.95 (indicating *at least* 95% match), a value that is empirically determined by us. The epoch size for fuzzy matches is the number of threaddumps that are needed to find the match.

B. Populating phase database and characterization

After the phase patterns (and therefore the corresponding epoch sizes for the phases) have been identified from the trace file using the text mining techniques described in the previous section, we populate a database with the discovered phases. We also track the execution order of the phases by recording information about the preceding two phases of any given phase.

While characterizing phases online, we pick the epoch size p_s of the smallest phase from the phase database. We match an epoch of p_s samples from the testing trace with all the phases from the phase database with epoch size p_s. If matches are found (using both f_{sim} and t_{sim} values), the same size is used for matching with the consecutive epochs. If any phase from the phase database reaches a certain confidence for the epoch size, we have characterized the phase. If not, we restart the characterization with the next bigger epoch size from the phase database. Each match for subsequent epochs increases the confidence by 1, while a mismatch decreases the confidence by 1. We buffer the trace data until we have found a characterization with confidence 5 (empirically determined). Once a match has been found with the above confidence, we have successfully characterized the phase. After successful characterization, we keep comparing the f_{sim} and t_{sim} values with the same epoch size until a point where a mismatch is found. This denotes a phase transition point, and we have to reset the epoch size and start the characterization again by comparing the phase database.

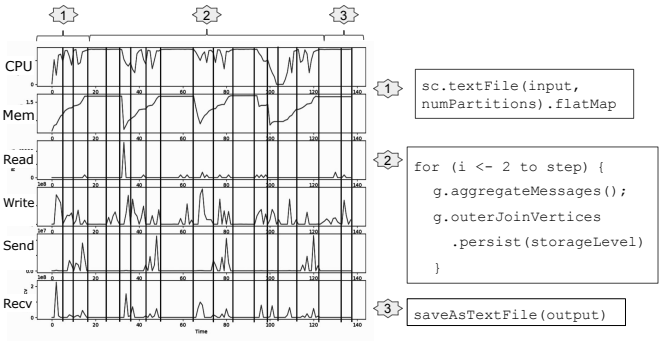

Figure 1: Resource Utilization profile and stages for NWeight workload.

If no accurate characterization has been done for the trace, we keep buffering the trace until we have reached a certain length that allows us to run our text mining method and update our database with newly discovered phases. This buffer size for learning can be selected by the performance engineer given the time and resources he has for training. Therefore, our method gives the opportunity to learn and relearn continuously in an online fashion and populate the phase database with newly discovered phases as new workloads are run on the deployed application. In the next section, we provide an example workload and its phases, along with experiments to validate our choice of stacktraces for identifying phases.

C. Phases in NWeight

In this section, we try to identify the correlation between the set of functions executed by a statistically constant number of threads (our definition of phase) and the various resource utilization of the application. We aim to justify the use of threaddumps in detecting phases. We discover phases in NWeight workload by varying the set of features. We use (1) resource utilization features (R) (2) stacktrace features (S) (3) resource utilization + stacktrace features (RS) ; and then identify how many phase change points are identified in each case. We also check whether the phase change points coincide. This gives us an idea about the correlation of features used for phase detection.

Figure 1 shows the resource usage profile of the NWeight workload and the identified phases while using different sets of features for the workload. The vertical lines show each phase change transition points identified when R features are used. This gives rise to a lot of fine grained phases detected due to small variations in any of the resource utilization

dimensions. These variations also affect the performance of the performance model as it becomes difficult for the model to identify performance repeatability across workload runs due to the fine grain non-determinism. The numbered braces on the top of the figure show the three phases identified while using the S and RS features. The corresponding code sections from the workload for the discovered phases are given on the right. The first phase is reading an input file and distributing it over the nodes for processing, giving rise to slow increase in memory and CPU utilization and the spike in network and disk accesses. The second phase is running a processing loop over the graph, which shows the recurring pattern across multiple dimensions of resource utilization. The persist operation across the loop iterations causes the spikes in disk writes. The third and final stage is writing the output to a file, which is identified as the spike in disk write towards the end. Therefore, for performing modeling at a granularity that provides better prediction accuracy, we choose to use S features. Though RS features provide the same benefit as S features, but it costs higher to generate models and perform prediction (due to significant increase in the number of features); that may be detrimental in an online setting. In the next section, we discuss the details of generating performance models per phase.

III. Input Aware Performance Modeling

In this section, our goal is to predict the behavior of future runs of an applications (and its *phases*) using previously *unseen* inputs without requiring user supplied information.

A. Input specification

Our assumption is that there are phases in the application whose *behavior* depends on the input to the application [10]. We generate performance models for (1) the resource behavior and (2) the execution time of phases. These behaviors are guided by the *values* of the input parameters. Some examples of input parameters are the number of records accessed from the database, percentages of reads or writes, the size of the file being read by the application, etc.

B. Meta-Input Identification

The original input parameter values to the application regulate the performance of the phases of that application. The main idea for *meta-inputs* comes from the fact that we can characterize the behavior of the current phase from the behavior of phases that are related by a *happens-before* relation with the current phase. We keep information about the execution order of phases in the phase database. In order to quantify the behavior of a phase, we first identify the set of functions, including their full stacktraces that are executed during the phase. We use information from the collected samples of the application to identify this set of functions $\mathbb{T}$. With this set $\mathbb{T}$, we create a *thread-vector* V, where each index of the vector corresponds to each function in $\mathbb{T}$ and the value for that index is the number of threads that executed that particular function during the phase.

$$V = (t_1^v, t_2^v, \ldots, t_n^v) \tag{3}$$

Here t_i^v represents the number of threads that executed a function t_i during the phase.

Along with this *thread-vector*, in order to create the complete *meta-input* set for the current phase, we also consider the (1) CPU, (2) memory usage (3) I/O behavior and (4) the execution time of the previous phases that happened-before the current phase. We extract aggregate statistical features (e.g. average, standard deviation, skewness etc.) to quantify the resource characteristics of the phases. The meta-input set is formed by a union of the meta-inputs (thread vectors and resource features) for *two* (found empirically) preceding input-dependent phases of the current phases. We build models for the execution time and resource consumption (CPU, Memory, Disk I/O, Network I/O) for different phases. We compare the performance of four algorithms that are popular and have been explored in performance modeling before: Logistic regression [11], LASSO Regression [12], Support Vector Regressor [13] and SGDRegressor.

IV. Experimental Evaluation

We present results for various HiBench workloads [14] running on a 4-node Spark [5] cloud and various YCSB applications running on a 4-node Cassandra [15] cloud. For each of the workloads, for the generation of performance models, we train with 200 runs using 15 different input configurations. For testing purposes, we run the workloads with 4 *unseen* inputs that have not been used during training. For the YCSB workloads, different read and write ratios are used as the input. For testing, each accuracy number in the next section represents the average accuracy from 16 prediction instances (4 test runs with each of the 4 unseen input configurations). LASSO performs best among the modeling algorithms we tried, therefore we only include the results using LASSO.

A. Accuracy Comparison Among Modeling Techniques

In this section, we compare the prediction accuracy among different techniques to check where our technique stands as compared to the state-of-the-art: (1) Naive: The prediction for the future time is same as the current value, (2) ARIMA models [1], [16], (3) Recurrent Neural Network model [2]–[4], [17]. The state of the art models are input and phase unaware.

Figure 2 shows the comparison among the prediction accuracies for CPU consumption of workload phases using different modeling techniques. The prediction accuracy is calculated by subtracting the Mean Absolute Percentage Error (MAPE) from 100. For reference, we also provide the accuracy from building models using user-supplied input (the best case) and LASSO as the modeling algorithm.

Our technique outperforms RNN and ARIMA without requiring any user supplied information about the running application. Also running algorithms, such as RNN, is computationally expensive than running LASSO, therefore RNN may not be suitable for online settings where faster learning and predictions are necessary. On average, our phase-aware modeling improves the prediction accuracy by 10% as compared to the state-of-the-art. Also, modeling using meta-inputs gives

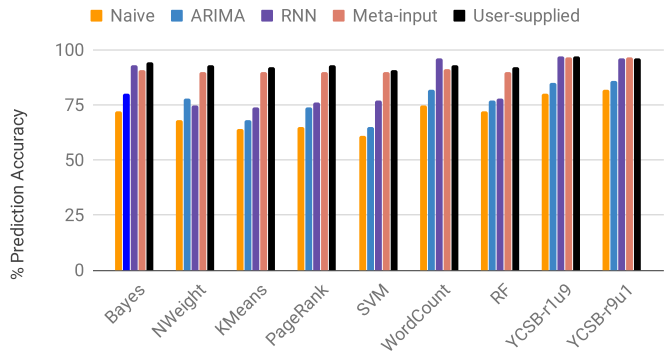

Figure 2: Comparison of prediction accuracy of State-of-the-art and our input-aware technique. For comparison we show the results from using both user supplied input and meta-inputs discovered by us.

an accuracy that is very close to the accuracy of the model generated by user-supplied inputs. This shows the strength of our technique, requiring very minimal information about the workload from the user. In the next section, we show how much impact this accuracy improvement has for the use case of anomaly detection.

B. Use Case: Anomaly Detection

We detect two types of anomalies as use case: (1) performance regression and (2) algorithmic imbalances (*stragglers* in Spark).

We assume that during training there are no/ statistically insignificant number of anomalous events. While testing, we identify the phase of the application and sample the target metric (resource utilization and/or execution time) for the phase. Then we compare the predicted value with the actual value for that metric, and based on a threshold difference, we flag the occurrence as anomalous/ normal.

1) Performance Regression: For software bugs, we consider two standard bug reports, i.e., Cassandra-6949 [18], which causes high CPU utilization due to redundant checks, and Cassandra-8559 [19], which is a memory leak bug. For hardware defects, we simulate a faulty disk in one of the Spark cluster nodes using Systemtap [20] tool. We also simulate an unusual traffic scenario that produces I/O discrepancy in Log-structure merge (LSM) systems where spilling to the disk will be the bottleneck when the update rate grows dramatically. For each anomaly type, we have 25 normal test runs and 25 anomalous test runs. Figure 3 shows the differences between a normal and an anomalous run.

Table I compares the anomaly detection accuracies of different state-of-the-art methods to our phase aware method. We report the False Positive Rate (FPR) and False Negative Rate (FNR) across all the modeling methods.

As seen in Table I, our method reduces both the FPR and FNR for the test cases significantly as compared with the state-of-the-art methods. Naive performs poorly with high FPR as it looks into the immediate past to take decisions. ARIMA and RNN both perform better than Naive because they learn the trend better than naive by looking at windows in the past. This effect is most visible in the unusual traffic scenario, where at each peak and trough, the Naive method generates a false positive (normal runs flagged as anomalous). Our phase aware method generates the lowest FPR.

Table I: FPR and FNR for different methods for all the anomalous cases.

	Naive		ARIMA		RNN		**Ours**	
	FPR	FNR	FPR	FNR	FPR	FNR	**FPR**	**FNR**
CPU	0.5	0.7	0.41	0.65	0.3	0.6	**0.02**	**0**
Mem	0.6	0.3	0.4	0.2	0.2	0.1	**0.01**	**0**
Disk	0.52	0.4	0.2	0.3	0.1	0.22	**0.01**	**0**
Traffic	0.65	0.6	0.43	0.53	0.4	0.4	**0.01**	**0**

But our method really shines while reducing false negatives (anomalous cases not detected). In cases of CPU anomaly and the unusual traffic scenario, the steady anomalous pattern results in poor performance of the naive method. For the CPU anomaly case, both the ARIMA and RNN suffer from high FNR without knowledge of the phase (as high CPU utilization from other workloads pollutes the model). But our method considers the phase information and is able to reduce the FNR to 0, thus resulting in a maximum of 80% reduction in false negatives.

2) Algorithmic Imbalances: In this section, we compare detection of *stragglers* using Spark default technique and our phase-based technique. For iterative workloads such as KMeans and NWeight, Spark generates stages that are too fine grained, due to the data dependency among them, while our epoch based phase detection technique consolidates them. This is particularly helpful because, due to data skew issues, the fine grained stages learn from execution time values that have high variation.

On the other hand, for workloads such as WordCount, the opposite happens. Our phase detection is able to identify and model garbage collections and shuffling phases separately, while in Spark stages, they often belong to a coarse grain stage, therefore affecting the performance of the generated performance model, therefore higher false positives in terms of detecting stragglers. Our phase based granularity lowers both the FPR (on average 58%) and FNR (on average 80%) for detecting Spark stragglers, as compared to Spark default straggler detection technique, in all the workloads.

C. Scalability

The main overhead for deploying our system at large scales in the real world comes from communicating the collected stacktraces to a central processing server. From our experience, the exchange of stacktraces produces, on average, 200Kbps of bandwidth on the network per node. Therefore, without causing significant interference to inter-node traffic for the workload, for a 25Gbps multi-flow traffic (very typical from cloud providers these days), our method can scale up to 25,000 nodes.

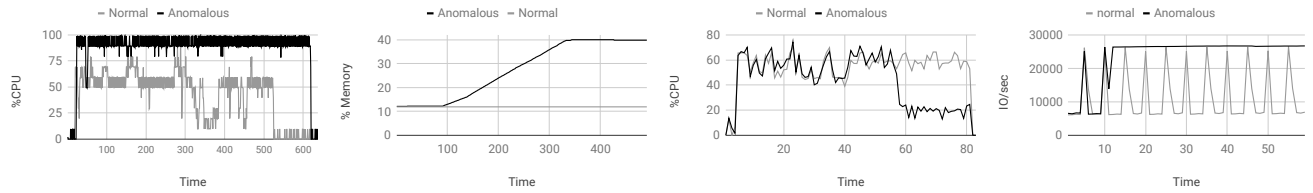

Figure 3: Different anomalies: (left to right) Cassandra bug 6949, Cassandra bug 8559, Faulty Disk, Unusual Traffic. Anomalous runs are dark colored lines.

V. Related Work

The use of performance modeling manually has been explored before. Hoefler et al. aimed to popularize performance modeling by defining a simple six-step process to create application performance models [21]. Bhattacharyya et al. [12] show the use of LASSO regression to build performance model of scientific applications. But none of these approaches talked about discovering inputs automatically from applications.

The Autoregressive Integrated Moving Average (ARIMA) model was applied to estimate the future need of applications [1]. There have been work on using hardware and software features to characterize applications and predict their performance using Recurrent Neural Networks [22], [23]. Although these methods have good prediction effects in the case of special scenarios, none of them addressed the issue of input awareness to the application behavior. Also, none of these approaches has good prediction accuracy for behavior with *unseen* inputs.

VI. Conclusion

In this paper we propose a new methodology for performance prediction for applications running in the cloud based on automatically identified *phases*. We introduce the concept of meta-inputs that drive the performance of different phases of a running application on the cloud. We show one use case of anomaly detection where we can detect the anomalies with a precision of 94.3% and a recall of 98% at very low (less than 5%) overhead.

References

[1] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, "Workload prediction using ARIMA model and its impact on cloud applications' QoS," *IEEE Transactions on Cloud Computing*, vol. 3, no. 4, pp. 449–458, 2015.

[2] Y.-C. Chang, R.-S. Chang, and F.-W. Chuang, "A predictive method for workload forecasting in the cloud environment," in *Advanced Technologies, Embedded and Multimedia for Human-Centric Computing*. Springer, 2014, pp. 577–585.

[3] A. Brown, A. Tuor, B. Hutchinson, and N. Nichols, "Recurrent neural network attention mechanisms for interpretable system log anomaly detection," in *Proceedings of the First Workshop on Machine Learning for Computing Systems*. ACM, 2018, p. 1.

[4] L. A. D. Bathen, S.-P. Genot, M. Qiao, and R. R. Routray, "Anomaly detection in multidimensional time series data," May 16 2019, US Patent App. 15/815,057.

[5] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, "Apache Spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[6] M. Crochemore and G. Tischler, "Computing longest previous non-overlapping factors," *Information Processing Letters*, vol. 111, no. 6, pp. 291–295, 2011.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

[8] "Memtier benchmark., url = https://redislabs.com/blog/memtier_benchmark-a-highthroughput-benchmarking-tool-for-redis-memcached."

[9] G. L. Simpson *et al.*, "Analogue methods in palaeoecology: using the analogue package," *Journal of Statistical Software*, vol. 22, no. 2, pp. 1–29, 2007.

[10] R. F. Da Silva, G. Juve, M. Rynge, E. Deelman, and M. Livny, "Online task resource consumption prediction for scientific workflows," *Parallel Processing Letters*, vol. 25, no. 03, p. 1541003, 2015.

[11] R. Ren, S. Tian, and L. Wang, "Online anomaly detection framework for spark systems via stage-task behavior modeling," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*. ACM, 2018, pp. 256–259.

[12] A. Bhattacharyya and T. Hoefler, "Pemogen: Automatic adaptive performance modeling during program runtime," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 2014, pp. 393–404.

[13] E. W. Fulp, G. A. Fink, and J. N. Haack, "Predicting Computer System Failures Using Support Vector Machines." *Workshop on the Analysis of System Logs*, vol. 8, pp. 5–13, 2008.

[14] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*. IEEE, 2010, pp. 41–51.

[15] "Apache Cassandra, url = http://cassandra.apache.org/."

[16] W. Fang, Z. Lu, J. Wu, and Z. Cao, "RPPS: a novel resource prediction and provisioning scheme in cloud data center," in *Services Computing (SCC), 2012 IEEE Ninth International Conference on*. IEEE, 2012, pp. 609–616.

[17] A. Bhattacharyya, S. A. J. Jandaghi, S. Sotiriadis, and C. Amza, "Semantic aware online detection of resource anomalies on the cloud," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2016, pp. 134–143.

[18] "Cassandra-6949, url = https://issues.apache.org/jira/browse/cassandra-6949."

[19] "Cassandra-8559, url = https://issues.apache.org/jira/browse/cassandra-8559."

[20] "Systemtap, url = http://sourceware.org/systemtap/."

[21] T. Hoefler, W. Gropp, W. Kramer, and M. Snir, "Performance modeling for systematic performance tuning," in *State of the Practice Reports*. ACM, 2011, p. 6.

[22] A. Bhattacharyya, S. Sotiriadis, and C. Amza, "Online phase detection and characterization of cloud applications," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2017, pp. 98–105.

[23] A. Bhattacharyya, S. A. J. Jandaghi, and C. Amza, "Semantic-aware online workload characterization and consolidation," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 210–219.