Vayu: Learning to control the cloud

Abstract

In this paper we describe Vayu, a system for managing cloud applications from the performance, availability and capacity standpoints. The system automatically learns the behavior of cloud applications and the remediation actions required to avoid and resolve problems that may arise in the application by detecting and creating signatures of problems and mapping them to a finite set of automated remediation actions available in cloud environments. Vayu is based on a set of algorithms; anomaly detection, problem signature and similarity and classification based action learning.

1. Problem statement

Cloud services, such as Amazon's EC2, provide the ability to rapidly deploy and scale applications based on demand, without the need to go through the complex change processes common in traditional IT. In many enterprises, the lines of business owners are pushing IT to provide both internal and external clouds for many business applications.
Detecting and solving problems while an application is running is still mostly a manual operation. The cloud provides tools that make controlling and tuning applications much simpler compared to traditional IT. Adding, removing, restarting and controlling the different virtual machines (VMs) that make up the applications is achieved through easy to use interfaces and APIs. However, to leverage these capabilities, the application owners are still required to monitor these applications continuously, detect problems and either take manual control actions, or manually pre-configure sets of rules that map their application state to control actions, a complex and lengthy process [1]. Complicating the automation is the fact that most of the applications are multi-tier, with scaling and remediation actions that can be taking at various levels. Automating the detection of problems and the creation of the control and tuning action rules is the challenge this works tackles.

2. Our Solution

We propose a system that automatically learns the application behavior and how to control and tune it. Overtime, a knowledge base is created and improves upon the automated control actions. Learning how to control the application is done continuously during the operations and load testing phases. Problems detected in production are simulated in a sandbox, where remediation action learning is performed. Once a remediation action to a problem is learned, it is applied in the production environment whenever the same type of problem is detected. The learning process is composed a number of steps: detecting a problem, classifying the problem according to previous occurrences and remediating it, simulating the problem when there is no remediation and learning of a remediation action for unseen problems. The flow of the system, named *Vayu*[1], is shown in Figure 1. Following is a detailed description of each of the steps:

Problem Detection/Prediction and characterization of the application state:
The application is monitored from both the end user and the system utilization perspective. Adaptive self-learning baselines provide the normal behavior of the different monitors that measure the application and its resource consumption. Our system identifies a performance anomaly in the application by combining probabilistically multiple monitors that deviate from their normal behavior, taking into account temporal and topological information. An anomaly creates a signature of the problem – which is described as the topology of the application components and the monitors on each node that deviated from their normal behavior during the anomaly. We briefly describe this algorithm in later sections.

Control Action Learning:
The system automatically learns which control actions actually resolve problems detected in the problem detection step by mapping the successful/unsuccessful corrective actions to the problem fingerprints. K-nearest neighbor classifiers are used for this mapping, where the feature space is the state characterization defined in the problem fingerprint. During the learning phase in the sandbox environment, the system tries out various corrective actions and observes if they resolve the problem, storing the result in the knowledge base. The actions are those defined as possible to apply automatically in the cloud environment (such as adding, restarting, removing and increasing the size of VMs). Each action has a cost associated with it, which is a function of actual monetary cost, time to take the action and the risk involved in taking the action. The cost is taken into consideration as a weight on the classification function.

[1] http://en.wikipedia.org/wiki/Vayu : The Lord of the winds

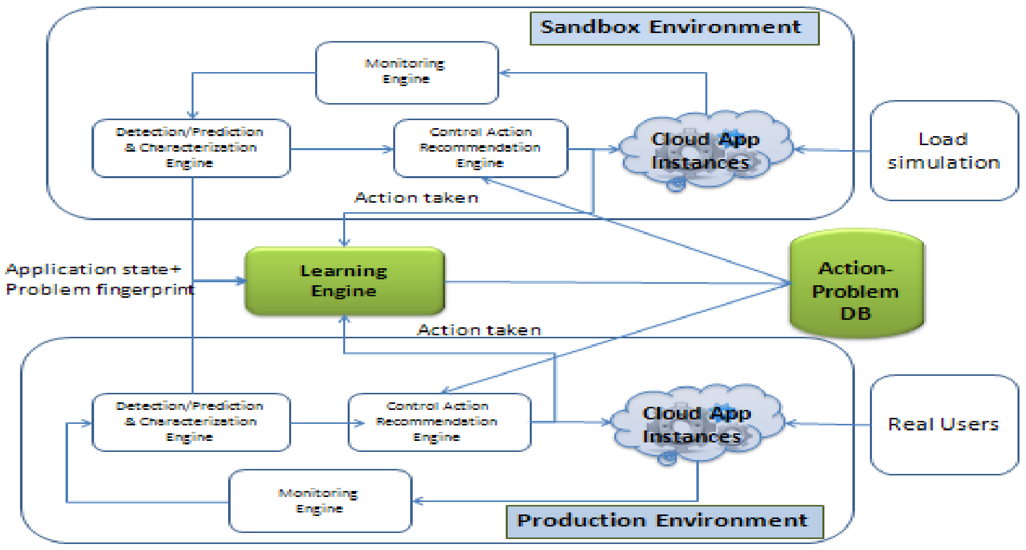

Sandbox Environment
Monitoring Engine
Detection/Prediction & Characterization Engine
Control Action Recommendation Engine
Cloud App Instances
Load simulation
Action taken
Application state+ Problem fingerprint
Learning Engine
Action-Problem DB
Action taken
Detection/Prediction & Characterization Engine
Control Action Recommendation Engine
Cloud App Instances
Real Users
Monitoring Engine
Production Environment

Figure 1 Vayu's system flow and basic components

To continuously tune the application, the system automatically learns and updates its baselines on the different monitors, describing normal behavior, so it can detect problems. When the system detects a problem for which it has not learned any control action, the system simulates the problem in a sandbox environment using a synthetic load. It tries-out the space of possible control actions until a remediation is discovered, and then applies it to the operational system and to future occurrences of a similar problem. The synthetic load is generated to match what has been measured on the production system before and during the problem occurs using an algorithm that maps real load patterns to synthetic load (the algorithm is beyond the scope of this paper). The system automatically recognizes whether the same problem was recreated in the sandbox environment by comparing the signature from production to the signature computed on the sandbox environment using a unique *problem similarity algorithm*. We now describe the different components of our solution in more detail.

2.1 Problem detection and signature creation:

In the first phase we monitor the application's state using both application performance monitors, such as response time and availability, and system level monitors, such as CPU, memory, disk and network monitors. For each monitor the normal values are learned using an adaptive baseline algorithm. When breaches to the normal behavior are detected for at least some application level performance monitors and system monitors, an anomaly is created using an anomaly detection algorithm. We now describe briefly both algorithms.

Adaptive baseline algorithm:

Our normal behavior model is a probability distribution conditional on the trend and seasonal behavior of the metric, that is, the probability density function of a metric at time t based on the season, *s*, and trend *tr* is given as: $p(m[t]|s,tr)\sim\mathcal{N}\left(\mu_{s,tr},\sigma_{s,tr}\middle|b[t]\right)$, where $\mathcal{N}(\mu,\sigma)$ is the normal distribution and *b[t]* is the time bucket to which the time *t* belongs to. The algorithm adaptively adjusts $\mu_{s,tr},\sigma_{s,tr}$ for each sample m[t] that is measured as follows:

1. Compute the number of standard deviations of m[t] from the previous model's mean: $numStd = |m[t]-\mu_{s,tr}|/\sigma_{s,tr}$
2. Update the forgetting factor α (between 0-1) such that it: a. increases (quicker forgetting) if there is a change event reported and/or there are many consecutive samples violating the normal model; b. decreases (slower forgetting) as the numStd is larger. The exact equation for updating alpha balances the two and allows for a soft update of the forgetting factor.
3. Update the mean and standard deviations according to: $\mu_{new} = \alpha\, m[t] + (1-\alpha)\mu_{current}$, $S^2_{new} = \alpha\, m[t]^2 + (1-\alpha)S^2_{current}$. The new standard deviation is therefore: $\sigma_{new} = \sqrt{S^2_{new} - {\mu_{new}}^2}$

The update of the forgetting factor in the second step ensures that known change events or a sequence of large deviations from normal behavior would cause a quick update to the model, while at the same time, short bursts of abnormal measurements are discounted and do not affect the model. The update ensures quick adaption without sacrificing robustness.

Anomaly detection algorithm and signature creation:

Given breaches to the normal behavior, an Anomaly Detection algorithm is used to determine if there is a real problem in the behavior of the application. Our anomaly detection algorithm combines the topology and temporal analysis using a statistical learning method to produce a single anomaly. At any given time, the algorithm computes the statistical significance of an anomaly occurring given the set of abnormal metrics observed on components that are topologically related, using the time since the abnormal behavior of each metric has been observed, the number of components involved and the statistical confidence of the baseline on each metric. A signature is then defined as a graph, with the topologically connected components of the application and the associated abnormal metrics on each one. The same algorithm reveals the end time of an anomaly.

2.1. Learning algorithms for control actions

The application is deployed on multiple components. We define two types of components: servers (VMs, including their operating systems) and middleware (e.g., database, web server, etc). An application uses one or more servers, and is deployed on one or more middleware components. For example, a web application may use a Tomcat web server and MySQL database. They may be installed on a single VM, or multiple VMs. We assume that applications may scale up or down by adding/removing identical VM+middleware components, with the part of application that uses those components. When multiple components exist, the load is balanced between these components identically.
We also define a finite set of actions that can be taken on a component. An initial set of actions are:

1. Add component type: add another instance of a component type (e.g., add web server)
2. Remove component type: remove an instance
3. Move component type: move an instance of a component type to a new VM.
4. Restart instance of a component
5. Change the configuration of an instance of a component. Most cloud providers have a predefined set of VM configurations (e.g., small, large, x-large, etc.)

In the cloud settings, operations 1-3 involve changing both the VM and the software/data required on the VM (middleware, etc). The restart operation may be on the middleware level (restarting Tomcat) or at the VM level. Note that the above actions are an initial set that can increase over time to include more complex actions, also at the application and middleware layer (e.g., increasing connection pools for databases). The characteristics of an action should be that there is an automated method to implement it in a reasonable timeframe. We assume that the placement of a VM in the cloud data center is done by the cloud provider, and is not part of Vayu's solution.
Given an application, we are given the type of components it requires to operate, and which ones can be scaled and in what manner. In addition, the application and its components are monitored by monitoring tools which report metrics describing the performance (e.g., transaction response time, user load, CPU utilization, memory utilization, etc). The performance of the application is monitored via availability and response time of transactions, and a problem is defined as one where there is degradation in the performance of the application, using the anomaly detection algorithm, which outputs a signature of the problem. Given a problem, a signature (s) of the problem is generated.
Let (s,a,r) be a triplet of problem signature (s), Action taken (a) and Result (r), where:
s – is a signature of symptoms representing a detected problem detected in by the anomaly detection algorithms. The set of all possible signatures is defined as S.
a – is an action taken to try to remediate the problem. a is taken from a finite set of possible actions, A={a1, a2, a3,…., an}. The set of actions is as defined above. For each action there is an associated cost {c1, c2, c3, …, cn}. The cost is a normalized value between 0-1, combining the actual $ cost of an action (increase/decrease), time to take the action and risk of taking the action on an application.
r – represents whether the problem was observed to have been resolved following the action taken (r is binary – R{ resolved/not resolved}).
The goal of the learning system is to learn a function f(s) -> a, such that the likelihood of r = resolved is the highest out of all possible actions in A with minimum cost. We use classifiers as the functions between f(s) and a.
In our case, we will transform the problem signature s to a set of features, and learn the mapping between the values of the features and the actions.
To define the features and classifiers, we first split the problem to two. We define two sets of action types: actions on component types (Ac) and actions on specific instances of a component type (Ai). An example of the first is adding a VM running a web server to the web server tier. An example of the latter is restart of a specific VM running a web server. Therefore, let Ac be the set of actions done on a component type (e.g., add component type, remove type) and let Ai be the set of actions done on a component type instance (e.g., restart tomcat, restart database, restart VM, etc). For each set of actions, Ac and Ai, we define different features as input for the classifiers.
Feature computations:

Step 1: *Metric abstraction: Map each metric measured to an abstract type. Examples are CPU Utilization, Memory Utilization, etc. There may be multiple metrics that map onto one abstract type.*
Step 2: *Compute features for component type action classification: Given a signature S, which is a list of metrics that were flagged as abnormal and their related instance information do:*

```
    - For each component type CTi in the application (index over all types of components)
        - For each abstract metric related to each component type, AMj (index over all types of abstract metrics
          measured on CTi)
            - Compute the average deviation from normal behavior for all abstract metrics, AMj on all CTi
              instances that were flagged as abnormal
            - Compute the percentage of abstract metrics, AMj, on all CTi instances that were flagged as
              abnormal
            - Store the result in as a feature vector.
```

Step 3: *Compute features for component instance action classification: Given a signature S, which is a list of metrics that were flagged as abnormal and their related instance information (Configuration Item, CI) do:*

```
    - For each component type instance CTij (j goes over all instance of component type i)
        - For each abstract metric related to each component type
            - Compute the average deviation from normal behavior for all metrics on instance CTij that are
              part of the abstract metric group and were flagged as abnormal
            - Compute the percentage of metrics that were flagged as abnormal that are part of the abstract
              metric group on instance CTij
            - Return a feature vector for each CTij
```

Classifier Learning
To learn classifiers, a training set is required. A training set is obtained by observing the application, either in production, in a sandbox environment. A training set is a set of N examples with the triplets (s_n, a_n, r_n) (signature, action taken and result), where n=1,…,N. There are several methods of classification that exist in the literature. There are two types classifiers that are learned from each training set: the component type classifier (Ac) and the instance specific classifier (Ai). In our current implementation we use the K-nearest neighbor classifiers [2], which require no training, other than computation and storage of the feature vectors with the actions and results. The classifiers are used to rank the actions using the following algorithm:

Action ranking algorithm:
Given a signature of a problem s:

- *Compute the feature vector for the component type action classifier (step 2)*
- *Compute the feature vectors for each instance of each component which exists currently in the application (Step 3)*
- *For each instance feature vector compute the likelihood of each action using the instance action classifiers.*
- *Compute the likelihood of each action using the component type classifier*
- *For each action in the list multiply the likelihood with (1- cost of the action (c)) to get a total score of the action*
- *Rank the list of actions based on the score, remove all actions with score < minimum threshold*
- *Pick the highest score action, recommend it as the remediation action*
- *Add the feature vector of the selected action to the training set with the action identifier and resolve/unresolved sign*

2.3 Problem similarity algorithm

Given two problem signatures, described as abnormal monitors on components of the application and the graph describing the topology of the application, the challenge becomes to determine a good similarity measure between the problems. We design our measure to take into account the topology similarity coupled with the similarity of the abnormal monitors on each component in the application. Our algorithm follows the following general steps:

1. *Abstract each monitor and component based on their ty*pe (e.g., monitor type: cpu time, available memory; Component type= Host, Database, Windows XP)
2. *Perform graph matching*: Find the best matching of Components from one problem to the other. For each pair of Components of the same type between the two problems we compute a matching score function based on the number of shared abnormal monitors of the same type and set of shared one-hop neighbor Components of the same Component type, normalized by the total number of monitors and one hop neighbors. With all pairwise scores between all Components, the overall matching is computed by finding the best assignment maximizing the overall sum of scores: we

use the Hungarian Algorithm (a combinatorial optimization algorithm which solves the assignment problem in polynomial time) for this step.

3. *Compute the overall similarity score:* We combine the similarity score which maximized the assignment in the previous step with the counts of matching links between Components that were deemed to match to each other. This step provides a refinement taking into account the link structure between the Components.

3. Initial Results

To demonstrate the working of Vayu, we deployed a web application (petshop.com) on the the Amazon cloud (AWS). We simulated real traffic on the application using LoadRunner, increasing the load to create various different performance problems. The application is built of multiple components: a web load balancer, tomcat server (web and application server on one machine), a MySQL database cluster, which includes a load balancer and replicated mysql database servers. All machines run on windows. Petshop.com is an open source e-commerce java application (jpetstore) widely used for experimentation. The two components that can be scaled in petshop.com are the tomcat and mysql servers. We monitor the application and components for response time, availability and infrastructure utilization, applying the baselining and anomaly detection on all monitors.

We generated two different kind of user related loads: one that creates a bottleneck on the tomcat layer (heavy browsing, mostly static pages, with not much buying), and a load which caused a DB related bottleneck (DB heavy transaction of inventory lookups). Figures 2-4 demonstrate some of the results of these experiments on the Vayu UI. In Figure 2, Vayu's knowledge base is empty (no signature-action pairs have been learned). Vayu detects an anomaly, but without any knowledge chooses to add a mysql server(due to the cost consideration). The problem is not resolved, and a rollback of the action is performed (the mysql server is removed), and a tomcat server is added. Very quickly the response time returns to normal values and the anomaly is closed. Vayu stores the signature-action pairs that resolved the problem (and the signature-action pair that didn't solve it) in the knowledge base for future use.

Figure 3 shows the next experiment, in which we ran a similar load (which causes a tomcat related bottleneck) at a later time. The anomaly is detected, but at in this case, Vayu chooses the correct action immediately, and the problem is resolved quickly. Figure 4 shows an experiment with a different load, causing a DB related bottleneck. In this example Vayu chooses the correct action first and the problem is quickly resolved.

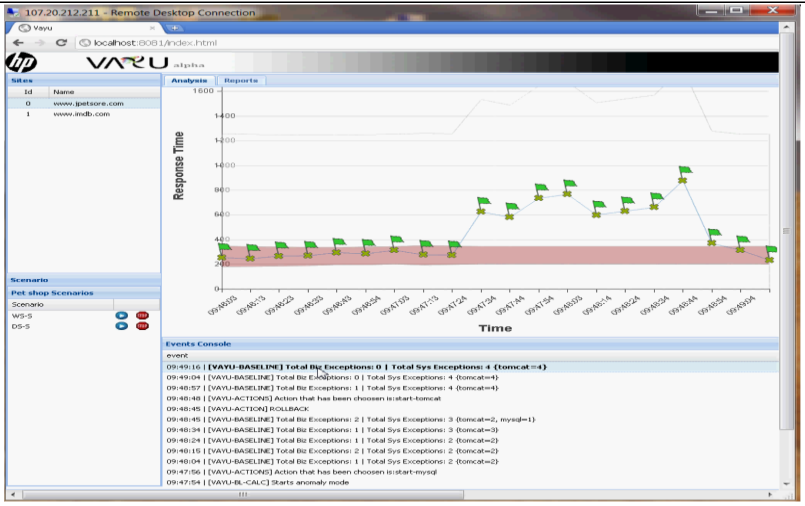
We have also tested the anomaly detection algorithm (including baseline), and the problem similarity algorithm, on various data sets collected from large scale production systems. Our anomaly detection algorithm displayed over 90% detection accuracy with very low false alarm (<1%). The problem similarity algorithm also showed robustness to noise and variations between environments, leading to 95% precision with a recall of 90-100% (depending on the level of noise). We found that the addition of topology provides a dramatic improvement in the similarity results. For lack of space we do not show all the details of these experiments.

4. Related Work

There has been a great deal of work around the use of machine learning for detection and characterization of problems in IT systems. The notion of retrieval of similar problems based on problem signatures, or fingerprints, has been explored in various works, e.g., [3][4].Our approach is different in that it represents the signature as a topological graph, performing similarity on the graphs. Automatic control in virtualized and cloud environments has been described in several works. Most related to Vayu are [5][6]. The main difference between these and Vayu is that in Vayu, the learning is performed with multiple control actions on multiple tiers of the application. In addition, compared to the method suggested in [5], which uses a simulator layer, Vayu learns both on the production system and in a sandbox environment. Vayu synthesizes the real load to recreate problems detected in production, and determines that it was recreated using the problem similarity algorithm. Compared to [6], the learning approach in Vayu is classification based, rather than a control based approach (Reinforcement learning). While the control based approach has more foundation for these types of problems, it is harder to learn in larger systems with multiple control actions and many states.

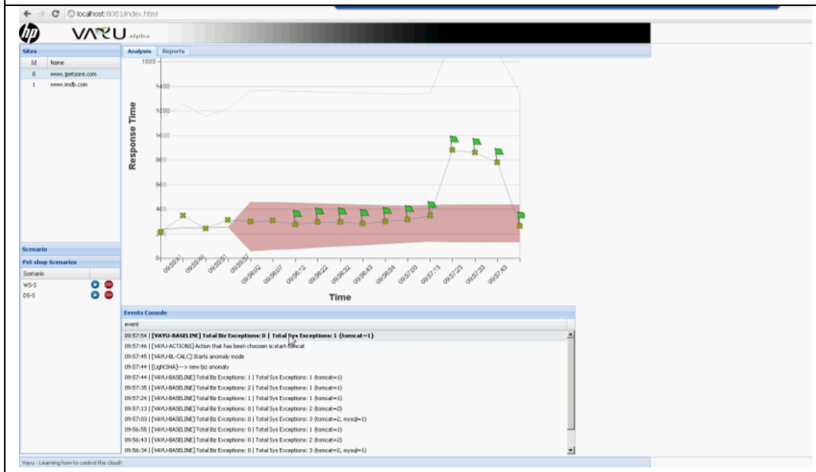
Summary

We presented Vayu, a system for automatic tuning of an application in cloud environments using an action learning mechanism that in composed of problem detection and remediation action learning. We presented initial results on a tested system deployed in Amazon's EC2, showing the efficacy of our approach. We are in the process of performing extensive experiments on a more complex real application running on amazon EC2, demostrating both the ability to identify and learn the right remediation action, leadind. Vayu is currently being developed as a product in a large management software vendor.



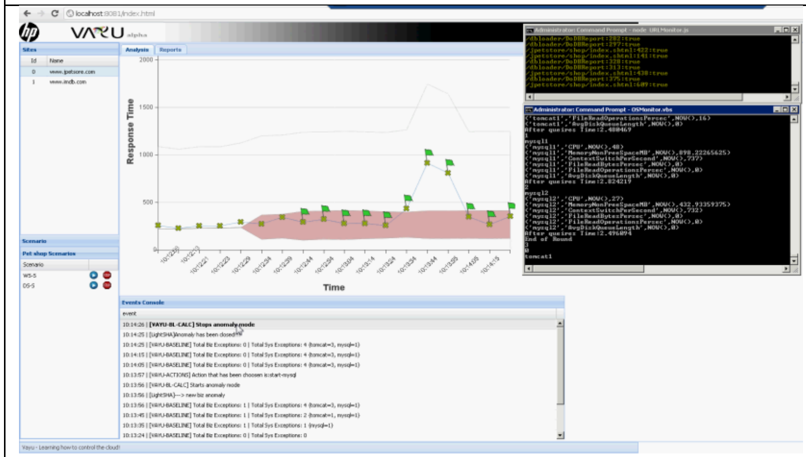
```
09:49:16 | [VAYU-BASELINE] Total Biz Exceptions: 0 | Total Sys Exceptions: 4 {tomcat=4}
09:49:04 | [VAYU-BASELINE] Total Biz Exceptions: 0 | Total Sys Exceptions: 4 {tomcat=4}
09:48:57 | [VAYU-BASELINE] Total Biz Exceptions: 1 | Total Sys Exceptions: 4 {tomcat=4}
09:48:48 | [VAYU-ACTIONS] Action that has been choosen is:start-tomcat
09:48:45 | [VAYU-ACTION] ROLLBACK
09:48:45 | [VAYU-BASELINE] Total Biz Exceptions: 2 | Total Sys Exceptions: 3 {tomcat=2, mysql=1}
09:48:34 | [VAYU-BASELINE] Total Biz Exceptions: 1 | Total Sys Exceptions: 3 {tomcat=3}
09:48:24 | [VAYU-BASELINE] Total Biz Exceptions: 1 | Total Sys Exceptions: 2 {tomcat=2}
09:48:15 | [VAYU-BASELINE] Total Biz Exceptions: 2 | Total Sys Exceptions: 2 {tomcat=2}
09:48:04 | [VAYU-BASELINE] Total Biz Exceptions: 1 | Total Sys Exceptions: 2 {tomcat=2}
09:47:56 | [VAYU-ACTIONS] Action that has been choosen is:start-mysql
09:47:54 | [VAYU-BL-CALC] Starts anomaly mode
```

Figure 2 Learning a Web server related anomaly: Vayu detects a business anomaly, attempts to resolve it by adding a DB machine (mysql), detects that the anomaly did not resolve and chooses the next action (adding Tomcat machine), which causes the return to normal operation.. The event console shows the stages towards the solution, the graph shows the response time monitor exceeding the baseline values and returning to normal after the correct action is taken.



```
09:57:54 | [VAYU-BASELINE] Total Biz Exceptions: 0 | Total Sys Exceptions: 1 {tomcat=1}
09:57:46 | [VAYU-ACTIONS] Action that has been choosen is:start-tomcat
09:57:45 | [VAYU-BL-CALC] Starts anomaly mode
09:57:44 | [LightSHA]---> new biz anomaly
09:57:44 | [VAYU-BASELINE] Total Biz Exceptions: 1 | Total Sys Exceptions: 1 {tomcat=1}
09:57:35 | [VAYU-BASELINE] Total Biz Exceptions: 2 | Total Sys Exceptions: 1 {tomcat=1}
09:57:24 | [VAYU-BASELINE] Total Biz Exceptions: 1 | Total Sys Exceptions: 1 {tomcat=1}
09:57:13 | [VAYU-BASELINE] Total Biz Exceptions: 0 | Total Sys Exceptions: 2 {tomcat=2}
09:57:03 | [VAYU-BASELINE] Total Biz Exceptions: 0 | Total Sys Exceptions: 3 {tomcat=2, mysql=1}
09:56:55 | [VAYU-BASELINE] Total Biz Exceptions: 0 | Total Sys Exceptions: 1 {tomcat=1}
09:56:43 | [VAYU-BASELINE] Total Biz Exceptions: 0 | Total Sys Exceptions: 2 {tomcat=2}
09:56:34 | [VAYU-BASELINE] Total Biz Exceptions: 0 | Total Sys Exceptions: 3 {tomcat=2, mysql=1}
```

Figure 3 The same problem as in Figure 2 is reconstructed. Vayu takes the correct action immediately and resolves the problem quickly.



localhost:8081/index.html
VAYU alpha
Sites
Analysis
Reports
Response Time
Time
Scenario
Pet shop Scenarios
WS-5
DS-5
Events Console
event
Vayu - Learning how to control the cloud!
Administrator: Command Prompt - node URLMonitor.js
Administrator: Command Prompt - OSMonitor.vbs

```
/dbloader/DoDBReport:282:true
/dbloader/DoDBReport:297:true
/jpetstore/shop/index.shtml:422:true
/jpetstore/shop/index.shtml:141:true
/dbloader/DoDBReport:328:true
/dbloader/DoDBReport:313:true
/jpetstore/shop/index.shtml:438:true
/dbloader/DoDBReport:375:true
/jpetstore/shop/index.shtml:609:true
```

```
('tomcat1','FileReadOperationsPersec',NOW(),16)
('tomcat1','AvgDiskQueueLength',NOW(),0)
After queires Time:2.480469
1
mysql1
('mysql1','CPU',NOW(),48)
('mysql1','MemoryNonFreeSpaceMB',NOW(),898.22265625)
('mysql1','ContextSwitchPerSecond',NOW(),737)
('mysql1','FileReadBytesPersec',NOW(),0)
('mysql1','FileReadOperationsPersec',NOW(),0)
('mysql1','AvgDiskQueueLength',NOW(),0)
After queires Time:2.824219
2
mysql2
('mysql2','CPU',NOW(),27)
('mysql2','MemoryNonFreeSpaceMB',NOW(),432.93359375)
('mysql2','ContextSwitchPerSecond',NOW(),732)
('mysql2','FileReadBytesPersec',NOW(),0)
('mysql2','FileReadOperationsPersec',NOW(),0)
('mysql2','AvgDiskQueueLength',NOW(),0)
After queires Time:2.496094
End of Round
3
0
tomcat1
```

```
10:14:26 | [VAYU-BL-CALC] Stops anomaly mode
10:14:25 | [LightSHA]Anomaly has been closed!!!
10:14:25 | [VAYU-BASELINE] Total Biz Exceptions: 0 | Total Sys Exceptions: 4 {tomcat=3, mysql=1}
10:14:15 | [VAYU-BASELINE] Total Biz Exceptions: 0 | Total Sys Exceptions: 4 {tomcat=3, mysql=1}
10:14:05 | [VAYU-BASELINE] Total Biz Exceptions: 0 | Total Sys Exceptions: 4 {tomcat=3, mysql=1}
10:13:57 | [VAYU-ACTIONS] Action that has been choosen is:start-mysql
10:13:56 | [VAYU-BL-CALC] Starts anomaly mode
10:13:56 | [LightSHA]---> new biz anomaly
10:13:56 | [VAYU-BASELINE] Total Biz Exceptions: 1 | Total Sys Exceptions: 4 {tomcat=3, mysql=1}
10:13:45 | [VAYU-BASELINE] Total Biz Exceptions: 1 | Total Sys Exceptions: 2 {tomcat=1, mysql=1}
10:13:35 | [VAYU-BASELINE] Total Biz Exceptions: 1 | Total Sys Exceptions: 1 {mysql=1}
10:13:24 | [VAYU-BASELINE] Total Biz Exceptions: 0 | Total Sys Exceptions: 0
```

Figure 4 A DB related anomaly, Vayu detects it and resolves it with the correct action (Add DB machine). Event console shows the stages towards the solution, the graph shows the response time monitor exceeding the baseline values and returning to normal.

References

[1] http://techblog.netflix.com/2012/01/auto-scaling-in-amazon-cloud.html

[2] http://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

[3] Ira Cohen , Steve Zhang , Moises Goldszmidt , Julie Symons , Terence Kelly , Armando Fox, Capturing, indexing, clustering, and retrieving system history, Proceedings of the twentieth ACM symposium on Operating systems principles, October 23-26, 2005, Brighton, United Kingdom

[4] Peter Bodik, Moises Goldszmidt, Armando Fox, Dawn B. Woodard, and Hans Andersen, Fingerprinting the datacenter: automated classification of performance crises, in EuroSys, 2010

[5] Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters
Peter Bodík, Rean Griffith, Charles Sutton, Armando Fox, Michael Jordan, and David Patterson, HotCloud 2009.

[6] Automatic exploration of datacenter performance regimes. P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. Patterson. First Workshop on Automated Control for Datacenters and Clouds (ACDC), Barcelona, Spain, 2009