IN4392: Automatic resource management of Python web applications at Amazon Web Services

Mircea Cadariu
EEMCS Faculty
Delft University of Technology
Email: M.Cadariu@student.tudelft.nl

Zmicier Zaleznicenka
EEMCS Faculty
Delft University of Technology
Email: D.V.Zhaleznichenka@student.tudelft.nl

***Abstract*—This paper discusses a system intended for the automatic deployment and monitoring of Python web applications using Amazon Web Services infrastructure. The proposed system was developed as a lab assignment for IN4392 Cloud Computing course at TU Delft. The functionality of a system resembles Elastic Beanstalk service provided by Amazon though the discussed system has less functionality. The performed evaluation shows that AWS is a reliable cloud computing platform for hosting applications of virtually arbitrary complexity.**

I. Introduction

In recent years cloud computing became a hot and emerging topic both in industry and academia. A large amount of cloud providers offer now their computational capacities for reasonable prices and the services offered in the clouds gain more and more popularity. Cloud computing platforms have many advantages over the traditional approaches in building infrastructure for the IT projects like purchasing own hardware, leasing it from data centers or utilizing the capacities of computational grids. These advantages include elasticity, high capacity, redundancy, ease of maintenance, flexible payment schemas and more.

Apart from providing pure computational power and virtually unlimited storage on demand, the cloud providers usually give their customers access to the additional services intended to automate the daily tasks of the system administrators working with the cloud platform. These services may add such functionality as automatic scaling, load balancing, real-time monitoring and e-mail notifications. Thus, one of the leading cloud computing providers Amazon Web Services (AWS) by the time this paper is being written supports over 25 services augmenting its major Simple Storage Service (S3) and Elastic Compute Cloud (EC2) services.

One of the important problems for the engineers willing to deploy their applications at AWS and similar cloud computing platforms is the need to get acquainted with all the diversity of services available at their cloud provider to work efficiently. While the large companies may afford having a dedicated specialist or a whole team busy with establishing and maintaining cloud infrastructure, for independent developers or small engineering or research groups this is usually impossible. Such teams want to benefit from all the advantages offered by the additional services in the cloud and spend as little time as possible in studying the proper ways to work with them in the meantime.

One of the solutions to this problem is the introduction of an additional service that would be used for rapid creation and easy management of rich environments utilizing most of the services available at the given cloud provider. Such a service should have a relatively small number of options and easy API allowing the developers to start working with the services offered by their cloud provider almost immediately.

AWS has its own service that is responsible for rapid application deployment. It is called Elastic Beanstalk (EB) and it allows to deploy .Net, Java, PHP, Python and Ruby applications at AWS premises in minutes. The applications deployed with EB benefit from immediately available detailed monitoring, e-mail notifications, a number of database services, auto-scaling and more. The similar services are also offered by many third parties, like RightScale[1] or Scalarium[2]. An additional benefit of these services is the support of many cloud providers. This means that if an engineering team decides to change their provider, they do not need to study the services offered by their new supplier as the only thing needed is to update the configuration of the intermediary service.

The project described in this paper was conducted as a lab assignment for IN4392 Cloud Computing course at TU Delft, The Netherlands and is an attempt to implement a service similar to AWS EB which should allow software developers to deploy and monitor Python web applications at AWS cloud. The implemented system is less advanced that EB. Due to the lab requirements it does not utilize APIs of many AWS services but provides an own implementation of them instead. The course was taught by dr. D.H.J. Epema and dr. A. Iosup, the lab work was supervised by B. Ghit ({D.H.J.Epema,A.Iosup,B.I.Ghit}@tudelft.nl).

The source code of this project and the latest version of this report are publicly available on Github[3].

The remainder of this paper is organized as follows. Section 2 contains a description of a reference web application built to test the forthcoming system. Section 3 discusses the system design and policies. Section 4 contains the evaluation. Section 5 is a discussion of general cloud computing platform

[1] http://www.rightscale.com
[2] http://www.scalarium.com
[3] https://github.com/dzzh/IN4392

properties and trade-offs. Section 6 contains suggestions for the future work. Section 7 reports the conclusions.

II. Background on application

Considering the time limitations for the project implementation, it was decided not to develop a very general service supporting many languages and frameworks available for the deployment but rather restrict to a particular class of the applications. Thus, the discussed system supports web applications written in Python and managed by Apache web server with mod_wsgi installed.

Before working on the infrastructural services responsible for AWS interaction, we have implemented a simple web service to be used as a reference web application for AWS environment. We have chosen matrices multiplication operation to be provided by the reference application as this operation can be implemented in a straightforward way and requires a lot of computational resources for completion thus allowing to perform load testing easily.

The reference application has simple user interface allowing a client to choose matrices dimension to multiply and start the computation. Before starting the computation, two random square matrices of given dimension are generated. After the computation is completed, the client is shown a web page with the time spent on computation. The results of the multiplication are not stored and if a user issues a new request, the whole computation is performed again with the newly generated matrices.

III. System design

A. Resource management architecture

IN4392 system consists of three main parts, namely Environment manager, System monitor and Monitoring GUI. All of these components run at a client machine and do not need in allocating any resources in the cloud for their proper operation.

Environment manager is responsible for creating and deleting the AWS environments. In the terms of this project an environment means a virtual organization of running and stopped instances that have the same application deployed and a load balancer which is used to direct client requests to one of the running computational instances.

A user of IN4392 starts working with the system by creating an environment using the Environment manager. While creating an environment, a minimal number of instances required for its proper operation is created at AWS data center. After the instances are launched, Environment manager starts an instance of AWS Elastic Load Balancer (ELB) and attaches the computational instances to it. In addition, at this stage the Environment manager sets up a security group and establishes test connection with the instance to ensure it started correctly.

When the environment is created, a client should ask Environment manager to deploy his application to it. At this stage, the client application and the instance configuration files get transferred to AWS instances. Environment manager connects to the instances using SSH protocol and performs the required management tasks to set up the application and start a web server. After this stage is done, the application becomes available online and may accept the incoming HTTP connections.

When the client does not need in an environment anymore, he can use Environment manager to delete it. When deleting the environment, all the computational instances and ELB instance terminate.

System monitor has to be started by the client after the application is deployed. The monitor is implemented as a script running an endless loop. It is responsible for auto scaling, health checks and informing about CPU utilization at the running computational instances. The System monitor adds to the system all the basic features that are requested in the assignment description. These features are implemented as follows.

After the monitor is started, no additional user interaction is needed for its proper operation. The System monitor constantly analyzes the state of a system and makes decisions based on the data retrieved from AWS CloudWatch (CW) monitoring service and the system configuration. This was the requirement for the *Automation* feature.

Elasticity and *Performance* requirements are addressed with the auto-scaling part of the System monitor. When the system needs to scale up, it looks for the available VMs in the resource pool. If the pool is not empty, one VM from it gets started. Otherwise a new VM is started and the application gets deployed to it. If the system exceeds the maximum number of VMs allowed to be run simultaneously, System monitor issues a warning. When down-scaling, one of the running instances is stopped and placed into the resource pool until reaching the minimum number of running instances.

Each VM may have only one application instance running but the number of allowed simultaneous WSGI processes and threads inside the VM can be adjusted in the system configuration.

Reliability feature is implemented with the health check procedure in the System monitor. In each monitoring iteration, the monitor requests the load balancer for the current state of its instances. If any of the instances reports any errors for several consequent checks, it gets stopped and the monitor logs this event. Later, a new instance is automatically added instead of the unhealthy one during the auto-scaling check.

The existence of the monitoring service itself solves the *Monitoring* requirement. So far, the functionality available in the Service monitor allows only to monitor the CPU utilization at the environment and certain computational instances, but it can be easily expanded. In addition, Monitoring GUI component provides visualization data for the environment CPU utilization.

Monitoring GUI is used to visualize the aspects that define the behavior of the application. This tool can be used to analyze the evolution of CPU utilization for the running instances of the computational environment. The data is displayed in a form of time-series chart, therefore the user can observe the trends in CPU utilization of the environment instances.

Monitoring GUI is implemented as MVC application where

the Model part is written as queries to AWS CW service, the Controllers are the Python wrappers that encapsulate queries to CloudWatch and serve the requests coming from the user via the browser and the Views are the web pages where the user can see charts and issue requests for their update.

IN4392 system has a number of configuration options that can be either default or specific for the given environments. The decisions made by Environment manager and System monitor are based on the application configuration and can be easily adjusted. All the components of the system are tightly integrated and reuse the code wherever possible.

B. System policies

The system currently supports two system policies applied to the System monitor operations. The first policy is related to the size of the resource pool. The system allows the user to specify minimum and maximum number of computational instances that can be run concurrently. The minimum limit is needed to guarantee certain system performance, the maximum limit is used to control the spendings on the environment.

The second policy defines CPU utilization limits used for auto-scaling. When the utilization falls below the lower limit, the monitor makes decision to scale down. Reciprocally, when the utilization exceeds the upper limit, the monitor makes decision to scale up.

The current version of the System monitor is rather simple and straightforward to support the additional system policies, but if it is improved for monitoring the other metrics, the respective system policies can be added to it.

C. Additional system features

While working on the project some effort was put into implementing *Security* and *Benchmarking* additional requirements.

The *Security* requirement is solved by applying the EC2 security groups policies while launching new computational instances. Currently, the security policy applied to the instance allows only HTTP and SSH access to it. For SSH access it is required to have private key file in possession.

Security can be improved further by adjusting the security policy after the instance is launched. When all the maintenance operations with it are done, there is no more need to keep SSH port opened and the respective rule can be removed from the policy. However, due to the time pressure this has not been implemented.

A number of tests were created in Apache JMeter tool for analyzing the behavior of the system under the peak load and verify the correctness of System monitor implementation. The existence of these tests and their description in the consequent chapter address the *Benchmarking* requirement.

IV. Experimental results

A. Experimental setup

All the experiments discussed in this section were conducted at the environments created with IN4392 system in Ireland region of AWS cloud. The reference application discussed in Section 2 was deployed on a number of t1.micro AWS instances and used ami-6d555119 VM (64-bit Amazon Linux with Python 2.6) and Apache 2.2 web server. The environment was configured to use at least one and at most five computational instances simultaneously.

No specific automated tests were written to test the Environment manager component. Instead, a thorough manual testing was performed after its implementation to make sure the manager correctly creates and deletes the environments as well as deploys the reference application to the launched computational instances.

While testing the Environment manager, the results of its work were validated against the output of the AWS management console. The correct application deployment procedure was verified by opening the launched application in the browser, interacting with it through the UI and analyzing web server logs.

To test the behavior of System monitor, a number of test plans were created using Apache JMeter[4] load testing tool. Each JMeter test is represented as a sequence of thread groups connecting to the ELB instance balancing the load of the environment and requesting the application to multiply two 100x100 random matrices. At different stages, various number of simultaneous threads and requests per thread are used to simulate different load and test the correctness of the System monitor, namely its ability to react on computational instance failures and need in auto-scaling.

For the tests, up-scaling CPU utilization limit was set to 90%, down-scaling limit was set to 40%. Minimum number of running instances was set to 1, maximum number was set to 5.

Monitoring GUI was tested manually. CPU Utilization graphs available from it were compared with the graphs provided at AWS CW console.

IN4392 system is fully implemented in Python programming language. For interacting with AWS, boto[5] framework was used. Data visualization at Monitoring GUI component is generated using Google Visualization Python API[6]. To create web applications (both reference matrix multiplication application and Monitoring GUI), web.py[7] framework was used. Twitter Bootstrap[8] was used as UI library for the reference web application.

B. Experiments

1) 7-stage JMeter test description: For the purposes of thorough testing of a System monitor and analyzing the system behavior under various loads, a seven-stage JMeter test plan was created. Its parameters are presented in Table I. The results of its execution are provided at Figure 1.

At each testing stage, different number of threads were run concurrently, each performing N requests to the load balancer.

[4]http://jmeter.apache.org
[5]https://github.com/boto/boto
[6]http://code.google.com/p/google-visualization-python/
[7]http://webpy.org
[8]http://twitter.github.com/bootstrap/

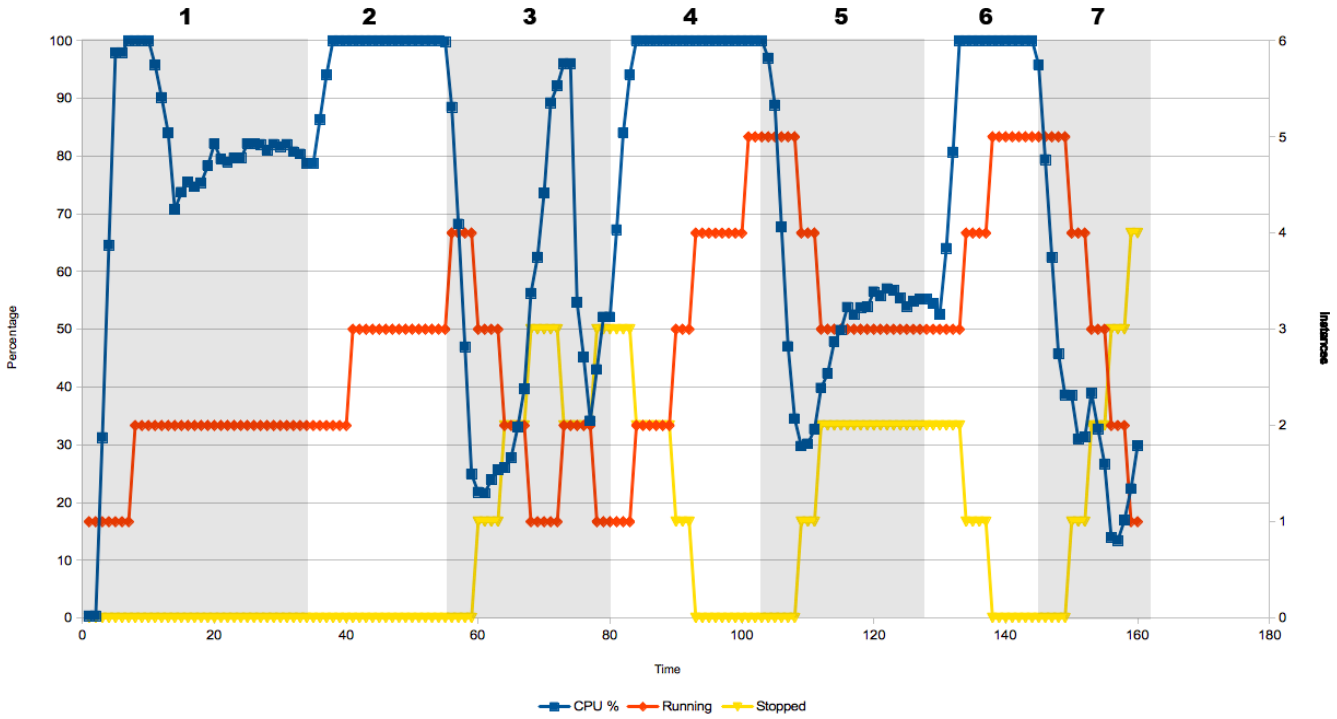

Fig. 1. Execution results for 7-stage JMeter test.

Stage	Threads	Samples/Thread	Delay
1	2	500	5000
2	5	300	3000
3	1	500	5000
4	10	200	2000
5	2	500	5000
6	10	150	2000
7	1	170	10000

TABLE I
7-STAGE JMETER TEST PARAMETERS.

Between two consequent requests, a thread was delayed for random time, with maximum delay in milliseconds reported in Table I.

The first stage was designed to be a warm-up, with just two threads and quite large delay between the requests. During the second stage, the load was increased by introducing three more threads and reducing the delay. Stage 3 was introduced to decrease the load to its very minimum and thus test the down-scaling functionality of System monitor. Stage 4 was a stress test exceeding the capabilities of the system. It was planned that all the available instances would start and the monitor would warn the user about the system overloading. Stage 5 was a relaxation with just 2 threads running. Stage 6 was another stress test testing the VMs allocation from the resource pool. Stage 7 was the final down-scaling stage testing that the environment is able to scale down to one single running instance.

These different execution stages are marked at the top of Figure 1 and can be distinguished by different background colors. The figure contains three graphs. Blue line shows the CPU usage of the environment in percents, red line shows the number of running instances and yellow line shows the number of stopped instances. Time is measured in minutes.

2) 7-stage JMeter test analysis: From the testing results, the following conclusions can be made. We see that during the first stage, the system was under high load and had to scale up by one instance. This means that the selected reference application consumes available CPU resources very aggressively. After the load was increased to five threads, the system scaled up to four running instances. We see, that between two upscaling events almost 15 minutes passed. The length of this period is caused by two factors. First of all, after the system reports that it has scaled up, a new instance does not accept new connections for a couple of minutes while being registered at the load balancer. Then, the monitor waits for AWS CW service to start sending monitoring statistics for the new instance. In some situation, this happens fast (see upscaling events around minutes 92 and 100), sometimes this lag is longer. This means that the system is not able to react to the changes in the load patterns quickly, time lag is always present and it may take up to 15 minutes (or even more) between two upscaling events.

It can also be seen that the upscaling event at minute 56 was reported when the system had already switched to stage 3. This is related to the fact that upscaling process takes around two minutes for instance launch and application deployment. The system reports that upscaling is completed when a new instance is already running and attached to the load balancer,

but the signal for it to launch is usually issued around two minutes earlier. For the instances running from the pool, the launching period was approximately one minute long.

At stage 3 we see that the system becomes underloaded and within ten minutes there are three down-scaling events. The smallest period between two auto-scaling events can be configured in the application settings to prevent the monitor from very frequent changes in the environment structure, to scale up and scale down gradually and have enough time for the system analysis between the auto-scaling events.

After scaling down to one instance, during stage 3 the system upscales and downscales back. To prevent it from such behavior, a lower down-scaling CPU utilization limit has to be set. It was set to 40% to see more auto-scaling events in this test. In production, this limit can be set to 20% CPU utilization or even lower.

During stage 4 we see that the utilization is raised up to 100% and remained there till this stage ends. During this time, the system gradually scales up, first from the pool of the stopped instances, then by initiating the new instances until it exceeds maximum number of them. When it is done (around minute 100), the system writes warnings to the log file, informing the user that further upscaling is needed.

At stage 5 the system gradually scales down to three running instances. Two instances get stopped and become available in the pool. During stage 6, the system scales up again and reports about the need to scale up further. During stage 7 it gradually scales down and places four stopped instances at the pool.

Generally, the performed load test have shown that the reference application selection was not entirely correct as matrix multiplication operation consumes resources in a different way than most web applications do. Mathematical operations quickly load the processor by 100% and it is difficult to execute similar operations at a single-core machine in parallel efficiently. Thus, even one user issuing requests to the server may load it by 100% quickly. On the other hand, many typical web applications are rather I/O-bound than CPU-bound (due to simple computational logic and database queries) and may be executed at a single-core computational instance simultaneously by many parallel threads while keeping CPU utilization low.

Another conclusion from the load test is that the monitoring service provided by AWS is not able to gather the data from the instances in real-time, in most situations the results were provided with a lag of 1-2 minutes. This lag has to be considered while designing the auto-scaling strategy and it is needed to know that it is impossible to up-scale the application immediately. At least several minutes are needed from the environment to understand that the application is overloaded, apply the auto-scaling policy and wait for the instance start-up and its registration at the load balancer. Thus, if an application has weakly predictable load patterns, it is better to keep more running instances that is needed at the moment to diminish the effect of start-up lag.

Time-cost analysis for 7-stage JMeter test is provided in Table II. From this analysis it is evident that due to the hourly charging system at AWS there is no need to scale the instances down until they work till the end of a full hour. This may even lead to cost savings for certain auto-scaling patterns if comparing with the policy when the instances are scaled down immediately after the load is decreased below the threshold.

Inst.	Runtime,min	Launches	Charged time,hrs	Cost,USD
1	160	1	3	0.06
2	145	3	5	0.1
3	92	2	3	0.06
4	48	3	3	0.06
5	24	2	2	0.04
total	**469**	**11**	**16**	**0.32**

TABLE II
7-STAGE JMETER TEST COST.

3) Testing basic system requirements: With the help of the described test it was possible to analyze most of the basic features required from the system. *Automation* was reached as no human intervention was needed during the test. It was only needed to run a short shell script to create an environment, deploy the application to it, start JMeter and System monitor and delete the environment when the test was done.

Elasticity and *Performance* were tested by applying different load patterns to the environment. From the test results we see that the monitor was able to launch the new VMs, stop them and add them back from a resource pool upon a need by a predictable pattern based on the CPU utilization of the environment. *Monitoring* was tested by analyzing the behavior of System monitor and its logs. From them it is clear that the monitor was constantly checking the environment load and reacted accordingly.

For *Reliability* requirement, a simple one-stage stress test was created in JMeter. After this test was started and a number of instances were launched, one instance was manually rebooted via AWS management console. The System monitor reported this event and stopped the instance. A new instance was later added to the environment during normal auto-scaling process.

V. DISCUSSION

Cloud computing platforms are being used for several years already for provision of computational power and storage services for different applications, including very large and complex ones. During this time, some cloud providers proved to be reliable suppliers for IaaS solutions, some did not, but the current level of technological development reached in the best and the most expensive data centers available for cloud computing customers allows to host there the applications of virtually arbitrary complexity with satisfactory service level guarantees.

When deciding whether to place an application at the data center of a cloud provider, a number of issues have to be taken into account. These issues may be categorized into financial and technological.

Placement of the application in the cloud has its advantages and disadvantages both for financial and technological aspects.

From financial point of view, a definite advantage of leasing the resources from the cloud is the elimination of high upfront expenses on building the own service infrastructure and paying high salaries for its support staff from the very beginning of the project lifecycle. For many of the projects being created while this report is filled in (i.e. deeply in the night), especially for the trendy start-ups raising venture investments it is difficult to predict the popularity of the final product and plan the infrastructural expenses in advance. Cloud computing platforms allow their customers to pay for what they use and adjust their environments in minutes if any errors were made during the planning stage. A disadvantage of cloud computing platforms is that for large long-term projects with predictable load patterns their placement in the cloud is generally more expensive than creating the own service infrastructure. There are some cloud providers that offer rather low rates but it is not always safe to place 100%-availability-dependent applications there due to their lower fault tolerance and guarantees if comparing with the leaders of the market.

From technological prospect, cloud platforms have a number of interesting properties that are difficult to achieve if having all the infrastructure on the own premises or built into general-purpose data centers. However, these platforms also have their drawbacks.

One of the important cloud platforms advantage already mentioned while discussing the financial aspects is their elasticity. If a customer plans to release a new application to the market and make an advertising campaign for it, it is almost impossible to predict its popularity and the load patterns. Cloud platforms allow to scale literally on-the-fly thus helping system administrators to struggle with extremely high load or to reduce amount of work if the load is low.

Another advantage of IaaS solutions is possibility to deploy global applications near to the end customers. Some of cloud computing providers, e.g. AWS, have their data centers located all over the world and a system administrator can easily manage his environments in different regions using same software or API calls. If building a platform at application vendor premises, its proximity to the end customers is often simply not achievable though it can be an important requirement for some classes of applications, i.e. online games servers which are very susceptible to network latencies by design.

Also, the cloud platforms provide certain service level guarantees that are hard to achieve at the local data centers. For instance, each AWS region has a number of availability zones, with each zone placed at its own premises having different power and network connectivity suppliers. Thus, even in case of serious natural or technological disasters there is possibility that at least some of the availability zones will remain in service. AWS allows to share the computational instances supporting one application within different availability zones in one region thus providing potentially best failure tolerance guarantees possible.

The technological disadvantages of using IaaS solutions are the following. First of all, the available hardware is usually limited by rather small number of configurations and it is impossible for the customers to build an own configuration that would serve their needs in the best way. Secondly, cloud computing platforms usually provide access not to hardware itself but to the virtual machines launched on top of it which may sometimes impose problems with the declared performance of the computational instances. The new instances may be added to the resource pools from different racks which can be inappropriate for the applications requiring high inter-node throughput. There are some solutions nowadays when it is possible to lease real hardware instances and guarantee the proximity of the newly added instances but such options are usually very expensive if comparing with having own racks in a general-purpose data center.

It can be concluded that the decision on whether to design an application for deployment at IaaS provider premises or in some other way should be based on a number of financial and technological considerations. In this section we have listed just a few of them but there are more. Large and small application with different usage patterns are successfully hosted in the clouds, in general-purpose data centers and at vendors' premises nowadays which means that there is no general advice that can be given about the selection of a hosting provider without the knowledge of the application requirements and a thorough financial analysis.

We did not perform the cost computations for the reference web application aimed to serve up to 10 000 000 of users as for this application this sort of planning is inappropriate. To analyze the spendings for this amount of customers there is a need to develop another application that would better resemble the load patterns usual for typical high-loaded multi-user applications in the Internet.

VI. Future work

The application built for IN4392 course can already help its users with deploying Python application at AWS clusters and monitoring their state almost in real-time but many improvements can still be made to it. One of such improvements is the addition of S3 support. With S3, it will be possible to launch new instances in approximately thirty seconds instead of approximately two minutes needed for the launch now as there will be no need to transfer files there from a local machine and wait for the SSH connection to be established. Instead, all the configuration could be done using init scripts. Also, this will allow to make the environment more secure, as there will be no need to keep SSH port open anymore.

Another important improvement to be done for IN4392 is addition of versioning support. Now, it is impossible to deploy a new version of the application to all the computational instances without terminating the environment though this functionality will definitely be required by the end customers, if any.

Also, the system would benefit from the integration with AWS Simple Notification Service (SNS) to inform the administrators about certain events via email, not only logging the warnings to the files.

VII. Conclusion

During the lab project for IN4392 course we have developed a system allowing to deploy Python web applications developed to use with Apache web server at AWS cloud computing platform. The performed evaluation shows that the built system is able to correctly create and terminate the computational environments as well as to monitor their state while they are in service and react to a number of events thus helping system administrators to manage AWS computational instances.

AWS is a reliable cloud computing platform with a rich set of services and a number of easy-to-use APIs that allow to deploy applications of virtually arbitrary complexity. It can be recommended as a powerful and flexible IaaS solution.

Appendix

A. Time sheets

The time sheets as requested can be found in Table III.

Time	**Mircea**	**Zmicier**
total-time	60	76
think-time	15	3
dev-time	30	2 (ref. app.), 30 (aws scripts)
xp-time	0	1 (building tests), 14 (testing)
analysis-time	0	3
write-time	5	8
wasted-time	10	15 (playing with Elastic Beanstalk)

TABLE III
Time sheets.

References

[1] AWS documentation. http://aws.amazon.com/documentation/

[2] Boto documentation. http://boto.cloudhackers.com.

[3] M. Garnaat. *Python and AWS Cookbook*. ISBN 9781449305444. O'Reilly Media, 2011.

[4] The Adaptive Web - How to setup mod_wsgi on EC2 Amazon AMI. http://theadaptiveweb.org/2011/11/04/how-to-setup-mod_wsgi-on-ec2-amazon-ami/.