A Framework for BPMS Performance and Cost Evaluation on the Cloud

Guillaume Rosinosky, Samir Youcef, François Charoy

▶ To cite this version:

Guillaume Rosinosky, Samir Youcef, François Charoy. A Framework for BPMS Performance and Cost Evaluation on the Cloud. Workshop "Business Process Monitoring and Performance Analysis in the Cloud", IEEE CloudCom, Dec 2016, Luxembourg, Luxembourg. hal-01379167

HAL Id: hal-01379167
https://hal.science/hal-01379167

Submitted on 11 Oct 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Framework for BPMS Performance and Cost Evaluation on the Cloud

Guillaume Rosinosky
Bonitasoft
Grenoble, France
Email: guillaume.rosinosky@bonitasoft.com

Samir Youcef, Francois Charoy
Universit de Lorraine, Inria, LORIA
Nancy, France
Email: samir.youcef@loria.fr, francois.charoy@loria.fr

***Abstract*—In this paper, we describe a framework that allows to automate and repeat business process execution on different cloud configurations. We present how and why the different components of the experimentation pipeline -like Ansible, Docker and Jenkins- have been set up, and the kind of results we obtained on a large set of configurations from the AWS public cloud. It allows us to calculate actual prices regarding the cost of process execution, in order to compare not only pure performance but also the economic dimension of process execution.**

1. Introduction

A Business Process Management System (BPMS) is usually deployed on traditional software stacks composed of an application engine and databases. Sizing the resources it requires to support a given load or number of transactions per second is not easy, since it may imply to test it on different computer size with different configurations for the engine and its components. Thus, most of the time, BPMS deployment is done on oversized systems. Thanks to the public cloud, it is possible today to conduct these tests in a semi-controlled environment that provides a wide variety of possible deployment configurations and to repeat benchmarking activities as much as one wants. Even better, it is possible to associate a cost to these configurations. Still, setting up a framework that would allow repeatable and controllable execution is not easy. In this paper, we describe and demonstrate that the framework we have designed allows us to evaluate the performances and the cost of the execution of processes on the BonitaBPM execution engine on a whole set of configurations in the cloud, such as the used cloud instance types and the parallel number of executed business processes, combining different well known open source software including Docker and Ansible. Our framework allows to generate data that relate the operating performances to a cost per BPM transactions. It could be adapted to other kinds of BPM systems or frameworks

In the section 2, we describe the context and the hypothesis that we made regarding the deployment and the execution of BPM engines. Then we describe the different initiative that are proposing solutions to similar problems. In section 4 we describe the technical principles of our framework and how its components are combined. Section 5 describes the experimentation and, in the last part, we discuss them in regard of our objectives.

2. Motivation and hypothesis

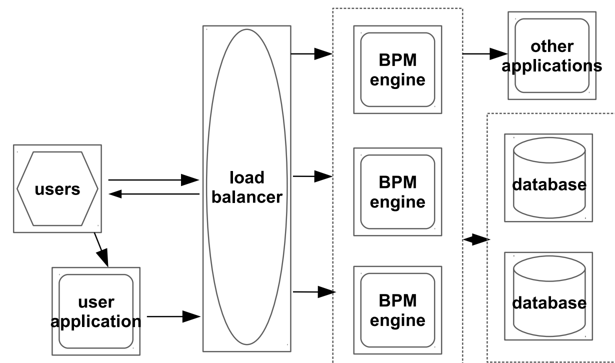

Figure 1. Simple BPMS architecture. This figure shows a clustered version of a BPMS using a clustered database, behind a load balancer.

Our original goal is to propose elastic resource allocation and scheduling methods for a BPMaaS provider. In order to size our infrastructure in a realistic way, we have to estimate different cloud configurations. Thus, we set up a framework to test multiple combinations of cloud resources by deploying in the Amazon Web Services (AWS) public cloud all the components needed for a BPMS (see figure 1 for an example of BPMS architecture) including the testing tools. Here, we test only configurations with one node for the database and one node for the application server and the BPMS engine. More precisely, we want to be able to :

- allocate three separated IaaS virtual machines, one for the database, one for the BPMS engine, and one the BPMN process injector
- deploy the relational database on its cloud instance
- deploy the application server and the BPMS on their cloud instance
- deploy the testing tool (process injector) on its cloud instance
- launch the tests
- get the results and the meta informations of the test

- get other useful data (database dump, logs, etc.)
- archive them
- deallocate the cloud instances

At the best of our knowledge, there is no existing tool today that can manage all these steps. It has to be automated since we want to repeat the tests, and later reuse this framework on different cloud providers, databases vendors, BPMS, with different cloud configurations. Furthermore, the tool needs to be customizable on several parameters such as the number of parallel BPMN processes instances launched against the BPM engine, the type and number of cloud resources used for the database or the BPMS engine, the size of the number of working threads of the BPM engine, etc. Some parameters of the application server and the database need to be tuned in regard of multiple cloud configurations with varying number of CPU cores, memory, etc. For instance, the number of available working threads for the BPMS to execute work, the number of connections, or the memory assigned to the application server are important parameters that can completely change the behaviour and the processing speed. These criteria must be passed as parameters or directly as files that have to be be incorporated.

This high variability and the large amount of tests that we have to conduct and to repeat oblige us to set up a highly automated environment. From the execution to the data collection, we must be able to execute the whole test pipeline automatically. In this part, we describe these components, how they are deployed and used for our experimentations. Figure 2 presents examples of configurations we want to obtain.

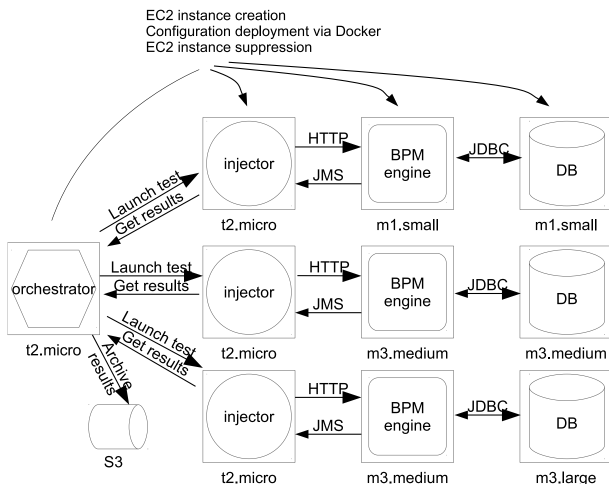

Figure 2. Example of test launches on three different configurations for the database tier and the BPMS tier. m1.small, t2.micro, m3.medium and m3.large are AWS instances classes with different memory, computing power, etc.

3. Related work

Since its inception, the Cloud has been recognized as an excellent platform to evaluate different kind of software stacks and to provide a good potential to conduct reproducible experimentation. As a programmable infrastructure, it allows to automate the experimentation pipeline in a very seamless and cost effective way, competitive with large international infrastructures like grid5k[1] or Planetlab[2]. Public Cloud have also the advantage of being actually used by companies for production deployment. The result that we can obtain can be exploited almost directly.

On the case of BPM engine evaluation, Benchflow [1] is a very interesting approach. It is a benchmarking framework based on Faban[3] - a generic measurement framework, and Docker. The authors use a process composed of an empty automated task and a timer event. Their main metric is the process throughput. They have tested two anonymized BPMS for a defined duration with various user loads, and observed differences between the performances and the general behaviour of these two systems. However, their tests are not deployed on the cloud, and the framework seems to lack an orchestrator for intensive testing. Betsy [2] is another interesting approach, but it focuses mainly on the BPMN or BPEL compliance of BPMS, even if a performance benchmarking is planned.

There exist several cloud-related generic attempts such as Smart CloudBench [3], where it is possible to test generic application on cloud resources. However these solutions are commercial and, even if they could theoretically be used in our case they are not directly BPMS related.

4. The framework components

Our framework for performance testing is primarily dedicated to benchmark Cloud configurations performance in order to compare different combinations and parameters. It is composed or uses the following components.

4.1. BonitaBPM BPMS

We used Bonita BPM[4] 7.3.1 in its community version as the BPMS engine. Bonita BPM is an opensource BPM solution originally developed in Inria [4] and now supported by the Bonitasoft company. It is compliant with BPMN and can be deployed on many SQL databases like H2, MySQL, PostgreSQL, Oracle, or SQL Server. In our case, we conducted our tests on Postgresql database. It is the reference database for BonitaBPM.

4.2. The Process injector

We need a component able to inject BPMN schemas, execute processes and measure the performances of the engine. For this, we used a testing tool developed by Bonitasoft. It allows to deploy business process model in a BonitaBPM installation, to execute them by mocking process variables,

1. https://www.grid5000.fr
2. https://www.planet-lab.org/
3. http://faban.org/
4. http://www.bonitasoft.com/

generate metrics about performance, and then undeploy the processes. The results this tool gives are process centric i.e. basic statistics on the processes runtime.

4.3. Containerization of software units

There exists several of libraries and parameters in the three main components : it can be difficult to master them, and we need a way to create isolated and repeatable tests. We decided to use Docker. Docker[5] is an open source project combining LXC containers, virtualization, and a configuration management platform. As [5] explains, Docker can be very useful for scientific experimentation. It is now used in many production cases, and allows to make repeatable and isolated executions faster than with a virtual machine. A Dockerfile contains the list of commands needed to initialize an environment such as installing libraries, setting environment variables, and execute the required programs. It is possible to version the Docker images, and to save them in a tar file and load afterwards in order to reproduce seamlessly the whole execution environment each time it is needed.

We have adapted a BonitaBPM Docker image in order to be able to select the database vendor, and to use the performance tool. We have developed a Docker image for the performance tool and we have specified environment variables as parameters, such as the type of injected process, the number of parallel processes run by the injector in the BPM engine, the total number of processes runned, etc.

4.4. IaaS provider

For our tests, we have used Amazon Web Services [6]. AWS provides a very well defined API to automate our tests and a wide selection of virtual machine configurations. It can also be considered as the reference public Cloud provider. Since our first goal is not to compare Clouds we did not reproduce the experiments on another IaaS but as we will see in the next part the framework could be easily adapted to another environment. We have mainly used here Elastic Compute Cloud (EC2) instances on which we have deployed the different needed components.

4.5. Orchestration tool

Thanks to Docker, we are able to manipulate only containers instead of deploying a lots of files. This is an important step, but not sufficient for our needs. Indeed we need to be able to allocate and de-allocate cloud resources, and keep the different IP addresses for each instance for various references. For instance the BPMS instance needs the address of the database, including some credentials. The performance tool instance needs the BPMS instance address. We needed a tool able to execute scripts, deploy, instantiate cloud, keep the inventory of the various instances and is able to inject in an instance configuration other instance data. Many tools exists for these kind of automation and could be considered, such as Puppet, Ansible, Chef, Salt [6]. We selected Ansible for our framework. As written in Ansible's documentation, Ansible[7] is an IT automation tool programmed in Python. It is able to configure systems, deploy software and orchestrate IT tasks with files usually in YAML. It does not need a component on the client side, as it uses internally SSH for the communications with the target instances.

Several concepts exists in Ansible :

- hosts and groups : target instances and their types - who can be used to obtain a subset of hosts
- inventory : current list of hosts and groups. It can be statically defined or dynamically defined
- task : a call to an Ansible module or a script
- role : a reusable group of tasks
- playbook : a model of configuration linking hosts and roles, and execute the latter on the concerned hosts

It is possible to use variables inside a playbook or a role. These can be defined at several places (for instance, in parameter of the playbook, in the host, the group, the role, the playbook, etc.) and can also be overloaded when calling the command line. The template engine Jinja is used by Ansible for variable operations.

We have set up several roles and playbooks for our needs, such as a database role, a benchmark role, etc. We have used the EC2 modules for the allocation and deallocation of EC2 resources, and the Docker module for image instantiation.

For the customization part, we have mainly used default variable definition, variables files for the mandatory customization parameters such as the size of the memory of the Java Virtual Machine used in the application server for the BPMS engine. Indeed this value should be adapted to the memory the instance type is able to provide. For instance, an EC2 m3.medium instance has 3.75 Go of RAM, and a c4.xlarge instance has 7.5 Go de RAM. For these we have assigned respectively 3 Go for the memory allocation pool of the m3.medium and 6.75 Go for the c4.xlarge. A list of the main internal parameters is in table 1.

We have set default values for our various parameters, but needed to be able to overload several with the needed variation in the tests. For this, we have used the possibility to pass variables directly in the command line (the highest priority), in order to make variation in the studied internal variables such as the number of launches, or the types of EC2 instances.

For the resource part, as it is possible to launch roles against a subset of hosts with the group concept, we have prepared one group for each tier (database, BPMS, bench tool). For instance, the database role will be launched against the hosts of the database group only. The dynamic inventory gets each hosts and its corresponding groups by querying

5. https://www.docker.com/
6. https://aws.amazon.com/
7. https://www.ansible.com/

Name	Role	Usage
name	global	test name
db_vendor	db & BPMS	database vendor
db_port	db & BPMS	database port
db_name	db & BPMS	database name
db_user	db & BPMS	database user
db_pass	db & BPMS	database password
ds1_minpoolsize	BPMS	database min pool size
ds1_maxpoolsize	BPMS	database max pool size
scheduler_poolsize	BPMS	BPMS scheduler pool size
java_opts	BPMS	Java options
max_threads	BPMS	max threads
bonita_version	BPMS	BPMS version
worker_size	BPMS	number of assigned threads
bonita_version	BPMS	version
perf_tests	benchmark	test type
perf_nb_parallel_launch	benchmark	nb of parallel instances
perf_nb_launch	benchmark	total number of instances
test_userid	global	unique test identifier

TABLE 1. ANSIBLE INTERNAL PARAMETERS.

AWS inventory servers. Furthermore, in order to identify and be able to launch several tests simultaneously, we have assigned a unique identifier to each current test. For this we have added a variable named test_userid that we use in the instantiation part as a supplementary group. For this, we have also modified the generic AWS dynamic inventory so it filters the hosts with this identifier. EC2 instances are declared with this additional EC2 tag in the creation playbook. Furthermore, we have added this value as a filter in the AWS dynamic inventory.

4.6. Test result collection

For this part, we used the s3cmd tool to send to a Amazon S3 bucket all our current results. For obvious speed and cost reasons, the scripts synchronizes only the files not already present in the bucket. We use the same command to download results for analysis.

4.7. Jenkins

Even with the orchestration offered by Ansible, launching all these scripts manually is not very efficient and can be complicated, mostly since we want to execute a lot of tests. In this case we have launched tests on more than 70 different configurations, each one launched 6 times. Indeed, we need to test a lot of instance types combinations for the BPMS and the database, including several different numbers of injected parallel processes. It was necessary to find a tool able to launch tests for us in an easy way, with a defined list of parameters and able to show logs feedback.

Jenkins[8] is an open source tool designed for continuous integration. It is possible to use it to make tests runs within a web-based user interface. We used it for the sake of simplicity, its scripting ability, and the capability to show different runs, and their log files. We used several plugins to simplify even more the processing, namely : Ansible plugin, Rebuilder plugin, Environment injector plugin, Conditional Buildstep plugin. We have two parts : parameterized jobs who call their Ansible counterparts, a job calling the command for result archiving and a parametrized Jenkins pipeline who launches the jobs and is able to launch multiple times the test. A Jenkins pipeline is a customizable job orchestration command where it is possible to put a Groovy script for calling jobs. The scripting language is very powerful, for instance it is possible to put exception catching, and we have used to trigger in all the cases the destroy job. Indeed in case of blocking errors, the reserved instances continue to be rented and it can become very expensive : it is important to destroy as soon as they are not needed anymore. We have also added a parameter for the number of test launches, adding simply a loop who will call the specified number of times the test job.

8. https://jenkins.io/

4.8. Overview

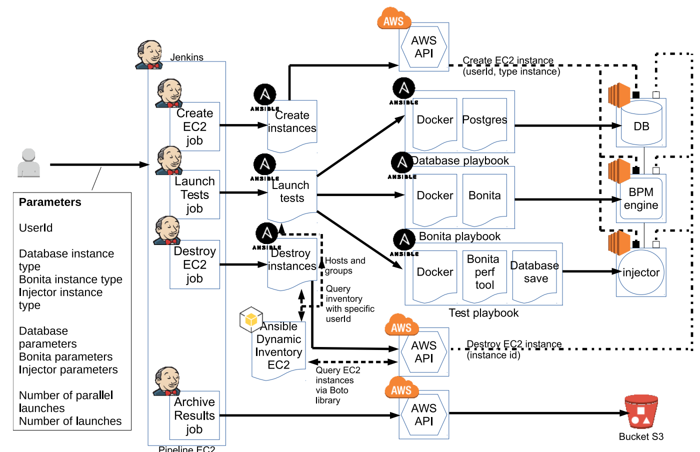

Figure 3. Test architecture used for the cloud configuration capability determination for EC2. We show here the orchestration units and a target generated configuration. A similar manner, without the Ansible database playbook, and with RDS instance creation and destruction is used for the RDS scenario described in chapter 5.

Figure 3 presents the global architecture. Setting up this entire test pipeline to conduct performance evaluation of different deployments remains a tedious task but we have seen that thanks to the cloud and of current administration automation tools it is possible to deploy and test a software stack under different configuration in a fully automated way and to conduct repeatable execution. In the next section, we describe the kind of result that we were able to produce based on this environment. At this point it is interesting to note that changing the software to test is mainly a matter of Docker Image production and to define the needed parameters in the Ansible playbook and the Jenkins jobs and pipeline.

5. Experimentation

5.1. Overview

A first version of this framework was used for the resource size estimation in [7]. In this paper, various AWS m3 family EC2 instances were used for the database and the application server. For this experimentation, we have launched tests against RDS Postgres, with db.r3 family instances (memory optimized) for the persistence tier, and with c4 (computing power optimized) family instances for the BPMS part.

For our tests, we have used the standard process, the reference process used for performance comparison between versions of BonitaBPM. This process is composed of 20 sequential automated tasks each one updating 15 string process variables from a constant and executing afterwards a connector repeatingly computing the Fibonacci number of 25 with a recursive method, until 150 milliseconds or more have passed.

The goal of our experimentation is to find the capacity of cloud resources in tasks per seconds. As a task could be very fast to execute, and the main metric used for size estimation in BonitaBPM benchmark tool is process based, we have used a reference time for the mean process duration. More precisely, we wanted our process to have a mean duration of 10 seconds : an arbitrary but realistic value for the standard process. Indeed, if we look only at the duration of the Fibonacci connectors, we obtain 3 seconds where we must add the process time of each process instantiation, variables allocation, workflow evaluation, and process termination. When launched with only one parallel inject business process, the duration of a process is about 5 seconds, as we will see in the figure 4.

We tested with different numbers of parallel process execution, and instance types for the database and application server. We have tuned BonitaBPM on the number of threads reserved for connectors and execution of tasks and on the database connection count, each one based on the parallel process number. The memory assigned to the Java(tm) VM of the BPMS' application server has also been tuned for each cloud configuration. We have also kept a unique specific parameter group for each RDS database, this one having most important parameters calculated from the size of the memory of the instance type. A parameter group is an AWS RDS reusable list of parameters used to tune the database. The list of parameters we used is listed in table 2.

Name	Tier	Usage
name	global	test name
nb_launch	global	number of launches of the test
database_instance_type	database	instance type file for database
bonita_instance_type	BPMS	instance type file for Bonita
parallel_launch	benchmark	number of parallel instances
configuration	BPMS	Bonita configuration file
worker_size	BPMS	number of assigned threads
bonita_version	BPMS	version
test_userid	global	unique test identifier

TABLE 2. USED PARAMETERS.

As the tests can be long, and expensive - we pay the cloud configurations during the tests - we have experimented a limited number of parallel processes, and then simply done a linear regression between the nearest upper and lower tests results around 10 seconds. For instance, if we obtain for 10 parallel processes a meantime of 5 seconds and for 20 parallel processes a meantime of 15 seconds, we will deduce we can use 15 process for a meantime of 10 seconds. We have then looked at the mean number of tasks per seconds for this number of processes.

5.2. Results

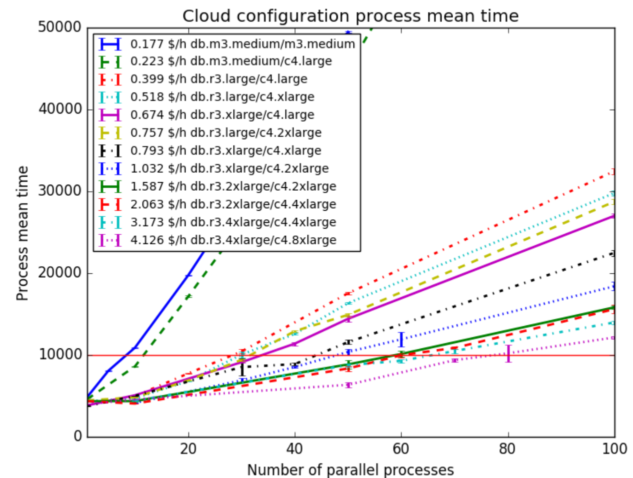

Figure 4. Process mean time for number of parallel process. Errors bars represent minimum and maximum values obtained. The red line represent the 10 seconds reference.

In figure 4, we can find the mean process execution time compared to the number of injected parallel processes. The throughput per second is simply obtained by dividing the total duration by the total number of executed tasks, 60000 in this case.

The task throughput and task throughput for one dollar is in table 3. This last metric is important to see which type of cloud instances are the cheapest to use for a given quality of service. We can see here that less powerful instances are more interesting, except for the generic instance type m3.medium used for the database tier or both the database tier and BPMS tier. However this configuration remains less expensive for a low throughput of tasks. More powerful resources are more expensive to use, but are still useful when higher task throughput is required. The configuration db.r3.large / c4.large is the cheapest configuration to use (with the higher throughput for one dollar).

Another interesting thing to notice here is that the db.r3.large / c4.2xlarge is a lot more expensive to use than the other db.r3.large based configurations, and is not able provide a throughput as high as the db.r3.xlarge / db.r3.large while being more expensive. This can be correlated with the fact that a cloud configuration uses both the database tier

DB inst. type	AS inst. type	price	task TP	task TP per $
db.m3.medium	m3.medium	0.177	16.400	92.656
db.m3.medium	c4.large	0.223	23.157	103.845
db.r3.large	**c4.large**	**0.399**	**55.164**	**138.255**
db.r3.large	c4.xlarge	0.518	58.067	112.100
db.r3.xlarge	c4.large	0.674	65.113	96.607
db.r3.large	c4.2xlarge	0.757	61.474	81.208
db.r3.xlarge	c4.xlarge	0.793	83.236	104.963
db.r3.xlarge	c4.2xlarge	1.032	89.149	86.384
db.r3.2xlarge	c4.2xlarge	1.587	105.794	66.663
db.r3.2xlarge	c4.4xlarge	2.063	107.585	52.150
db.r3.4xlarge	c4.4xlarge	3.173	115.283	36.332
db.r3.4xlarge	c4.8xlarge	4.126	129.279	31.332

TABLE 3. PRICE, MEAN TASK THROUGHPUT, AND MEAN TASK THROUGHPUT BY DOLLAR FOR A MEAN STANDARD PROCESS DURATION OF 10 SECONDS.

and the BPMS tier, and paying more only for one of the tier could become counterproductive. This is visible in the figure 5. However, this needs to be tested more, as noisy neighbor effects could be partially at the origin of these results, as we can see in the error bars of figure 4.

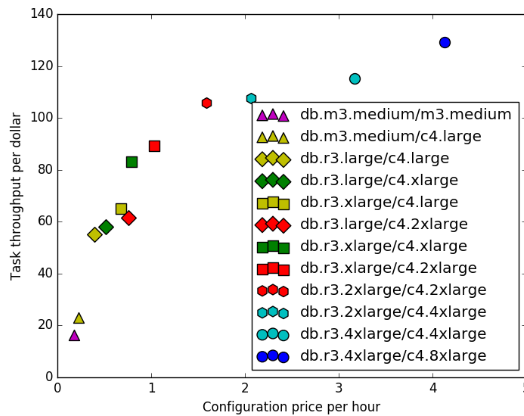

Figure 5. Price vs mean task throughput for each tested configuration. Shapes represent the database tier resource type, colors the BPMS tier resource type.

6. Conclusion

In this paper, we have presented our framework that allows to conduct extensive tests of the execution of a BPMS engine under different cloud and engine configuration and to collect data in order to compare the result. Thanks to the entire automation of the test pipeline, based only a collection of open source tools it is possible to execute the test with as many configuration as we need the number of time we need. It remains also very flexible and cost effective. We spent around 250 $ in AWS credit to conduct all the tests needed for this paper, including runs that were required to validate the framework. This price could be reduced on AWS by using spot instances, that would allow to use VM at a fraction of the price. This framework also works with Vagrant for testing purposes, and with on premises computers, with a static inventory including hard coded IP adresses for each tier, and without the cloud instance creation and destruction.

In the near future, we plan to enhance the framework with the capability of testing clustered configurations and other database vendors. These operations are easy to add, since we just have to rely on the creation and orchestration of new docker images. We also want to add the possibility to test other BPMS, and to couple this framework with a load balancer and an resource allocation and scheduling algorithm as in [7]. Finally, we plan to make more intensive benchmarks in order to better estimate the price and efficiency of BPMS on cloud configurations.

Acknowledgments

The authors would like to thank Amazon Web Services for the free credits (this paper is supported by an AWS in Education Research Grant Award).

References

[1] V. Ferme, A. Ivanchikj, and C. Pautasso, "A framework for benchmarking BPMN 2.0 workflow management systems," in *International Conference on Business Process Management*. Springer, 2015, pp. 251–259, 00006. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-319-23063-4_18

[2] M. Geiger, S. Harrer, and J. Lenhard, "Process Engine Benchmarking with BetsyCurrent Status and Future Directions," *ZEUS 2016*, p. 37, 2016, 00000. [Online]. Available: http://www.infosys.tuwien.ac.at/zeus2016/proceedings.pdf#page=42

[3] M. Baruwal Chhetri, S. Chichin, Q. B. Vo, and R. Kowalczyk, "Smart CloudBenchA framework for evaluating cloud infrastructure performance," *Information Systems Frontiers*, vol. 18, no. 3, pp. 413–428, Jun. 2016, 00003. [Online]. Available: http://link.springer.com/10.1007/s10796-015-9557-2

[4] F. Charoy, A. Guabtni, and M. V. Faura, "A dynamic workflow management system for coordination of cooperative activities," in *Business Process Management Workshops*. Springer, 2006, pp. 205–216, 00024. [Online]. Available: http://link.springer.com/chapter/10.1007/11837862_21

[5] C. Boettiger, "An introduction to Docker for reproducible research, with examples from the R environment," *arXiv preprint arXiv:1410.0846*, 2014, 00002. [Online]. Available: http://arxiv.org/abs/1410.0846

[6] P. Venezia, "Review: Puppet vs. Chef vs. Ansible vs. Salt," Nov. 2013. [Online]. Available: http://www.infoworld.com/article/2609482/data-center/data-center-review-puppet-vs-chef-vs-ansible-vs-salt.html

[7] G. Rosinosky, S. Youcef, and F. Charoy, "An Efficient Approach for Multi-tenant Elastic Business Processes Management in Cloud Computing environment," 2016, 00000. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01300188/