Cloud WorkBench: Benchmarking IaaS Providers based on Infrastructure-as-Code

Joel Scheuner, Jürgen Cito, Philipp Leitner, Harald Gall

software evolution & architecture lab
University of Zurich
Switzerland
{lastname}@ifi.uzh.ch

ABSTRACT

Optimizing the deployment of applications in Infrastructure-as-a-Service clouds requires to evaluate the costs and performance of different combinations of cloud configurations which is unfortunately a cumbersome and error-prone process. In this paper, we present Cloud WorkBench (CWB), a concrete implementation of a cloud benchmarking Web service, which fosters the definition of reusable and representative benchmarks. We demonstrate the complete cycle of benchmarking an IaaS service with the sample benchmark SysBench. In distinction to existing work, our system is based on the notion of Infrastructure-as-Code, which is a state of the art concept to define IT infrastructure in a reproducible, well-defined, and testable way.

Categories and Subject Descriptors

D.m [**Software**]: MISCELLANEOUS

Keywords

Cloud Computing; IaaS; Benchmarking; DevOps; IaC

1. INTRODUCTION

In the cloud computing model Infrastructure-as-a-Service (IaaS), *"processing, storage, networks, and other fundamental computing resources"* [7] are acquired on a pay-per-use basis, most commonly in the form of virtual machines (VMs). The functional similarities of these services are contrasted by significant variations in non-functional properties. Service performance not only varies between providers, as studies listed in [2] show, but also for services exhibiting the same specification [3]. Representative benchmarks (*i.e.*, performance tests) can be used to assess service performance and thus assist software engineers in service selection. However, testing multiple providers with variable configurations results in a large parameter space to explore, making benchmarking a labor-intensive task. Moreover, in fast moving cloud environments, continuous reevaluation is inevitable, when providers change their supported instance types or upgrade their hardware. Therefore, several research projects [1, 5, 9] aiming at extensible cloud benchmark automation were recently introduced. They all facilitate systematic cloud benchmarking. However, defining benchmarks is typically a tedious and error-prone activity, which often involves manually creating VM images for each benchmark, cloud provider, and region. This increases the time necessary to benchmark a given configuration, and reduces comparability and reproducibility of results.

In this paper we demonstrate Cloud WorkBench (CWB), a web-based framework that is grounded on the notion of Infrastructure-as-Code (IaC) to foster simple definition, execution, and repetition of benchmarks over a wide array of cloud providers and configurations. IaC was introduced by the current DevOps trend [4] and captures the complete provisioning and configuration of various middleware components, most importantly IaaS VMs, operating systems, and standard software, in provisioning code. Applying provisioning code reproducibly converges a system to a desired state, without the need for manual steps and irrespective of previous configurations of the same components.

We introduced Cloud WorkBench (CWB) in a companion full paper [8], presented the results of a large-scale cloud evaluation analyzing more than 33000 measurements in [6], and we now give an example of its capabilities by showing how a common benchmark from literature can be completely defined, scheduled, and automatically executed in a cloud environment.

2. SYSTEM OVERVIEW

This section introduces the CWB framework for defining, scheduling, and executing benchmarks.

2.1 Architecture

Defining and executing a benchmark in CWB involves interactions among the components shown in Figure 1.

The (human) *experimenter* defines benchmarks via the provisioning service and the CWB web interface, which subsequently allows one to schedule and manage executions of benchmarks. The CWB *server* is the main component consisting of a standard three-tiered web application. It provides the web interface, implements the business logic in collaboration with external dependencies, and stores its data (definitions and results of benchmarks) in a relational database. The scheduler component of the CWB *server* periodically triggers the execution of defined benchmarks.

WWW 2015 Companion, May 18–22, 2015, Florence, Italy.
ACM 978-1-4503-3473-0/15/05.
http://dx.doi.org/10.1145/2740908.2742833 .

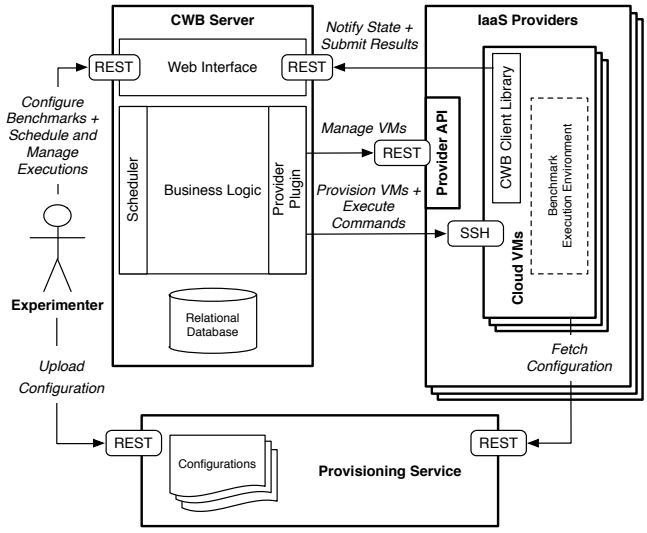

Figure 1: Architecture Overview

Benchmarks in CWB are typically defined across a multitude of different IaaS providers, which the CWB *server* interacts with over a *provider API*. Fundamentally, this API is mostly used to acquire and release cloud VMs of a given user-defined specification. These *cloud VMs* are the System Under Test (SUT) and execute the actual benchmarking code. To ease the interaction between the *cloud VMs* and the CWB *server*, a small CWB client library is installed in each VM. This client library, along with all other required code (*e.g.*, Linux packages required by a benchmark, or the benchmark code itself), is provisioned in the *cloud VMs* based on IaC configurations retrieved from a *provisioning service*. The *provisioning service* knows how to prepare a given bare VM to execute a given benchmark.

All interactions among these components happen typically over REST services to foster loose coupling and reusability, with the exception of the interaction between the CWB *server* and the *cloud VMs*. These components communicate over the standard Linux utilities `rsync` and `ssh` for simplicity reasons.

2.2 Benchmark Definition

One core feature of CWB is that benchmarks, including the cloud configuration they are evaluating, can be defined entirely in code, essentially following the ideas of DevOps and IaC. As argued in [4], this renders the process reproducible, modularizable, flexible, and testable using standard software engineering techniques. Common components among benchmarks can be easily shared and provisioning configurations from a large provisioning service community can be reused to efficiently describe the benchmark installation.

Logically, a benchmark definition requires the information depicted in the simplified UML class diagram in Figure 2. Every benchmark definition must specify one or more client VMs, which are brought into the expected configuration state via executing one or more provisioning configurations. Both, the definition of client VMs and provisioning configurations follow the established notions of standard IaC

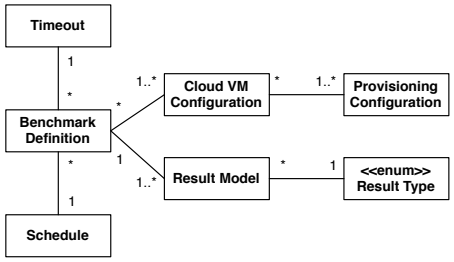

Figure 2: Structure of a Benchmark Definition

tooling (*e.g.*, Vagrant[1] and Opscode Chef[2]). In addition, every benchmark definition requires one or more result models, which capture the type of outcome (*i.e.*, either nominal, ordinal, interval, or ratio scale) a benchmark will deliver. Finally, benchmarks optionally also contain a schedule (benchmarks without a schedule are only triggered manually by the experimenter) and a timeout, after which the execution of a benchmark is terminated no matter whether it is finished or not. This timeout prevents potentially costly resource leaks caused by unforeseen exceptions during the execution.

CWB defines an interface to handle the interaction with user-defined benchmarks. Each benchmark must implement a callback (*i.e.*, a piece of code following a defined convention, which can be invoked by the CWB server) to start executing. Further, benchmarks should use the provided CWB client library to notify state updates (*e.g.*, when the benchmark run is completed or a failure has occurred and submit results back to the CWB server. The client library is transparently installed via a pre-defined provisioning configuration.

As the provisioning code is logically separated from the definition of the cloud VMs, it is easy to set up a large array of benchmarks that evaluate different cloud configurations, and be confident that the code and setup of each benchmark is in fact identical except for the facets that the experimenter specifically wants to vary.

2.3 Executing Benchmarks

Figure 3 illustrates the interactions when a new benchmark execution is triggered by the experimenter or the scheduler. For simplicity, we focus on a successful execution here (*i.e.*, neither provisioning nor benchmark execution fails, and the benchmark finishes before the defined timeout is exceeded). For further detail regarding the executional behavior, we refer to the benchmark state model presented in our accompanying paper [8].

A provider plugin in the business logic asynchronously acquires cloud resources (typically cloud VMs, but it may also comprise of cloud specific features such as dedicated block storage or dynamically mapped IP addresses). As soon as the business logic has managed to establish a remote shell connection to the cloud VM, it starts orchestrating the VM provisioning via the remote shell connection. Thereby, each cloud VM fetches its role dependent configurations from the provisioning service and applies them.

At this point, the benchmark is entirely prepared for execution and asynchronously started via a remote shell command. This command invokes a defined callback on the VM

[1] `https://www.vagrantup.com/`
[2] `https://www.chef.io/`

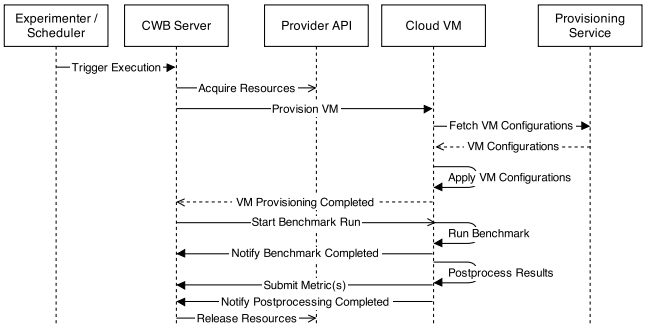

Figure 3: Executing a Benchmark

that any benchmark has to implement. Once the actual benchmark workload is completed, the benchmark should notify this state update to the CWB server via the client library. The benchmark results are then postprocessed, which typically involves textual result extraction, and submitted to the CWB server as individual metrics or as a collection of metrics via a CSV file. After completed work, the cloud VM notifies the state update to the CWB server in order to trigger all resources being released.

2.4 System Implementation

The CWB web application is implemented using the Ruby on Rails[3] framework. One of the fundamental distinctions between CWB and related work is that it strives to reuse as much existing DevOps tooling as possible, so that experimenters can build upon existing community artifacts (e.g., for provisioning configurations) and knowledge. Hence, it integrates Cron as the scheduler, Vagrant as the VM environment management tool, and Opscode Chef as the provisioning tool.

Vagrant was chosen to represent cloud VM configurations using an established Ruby-based domain specific language (DSL). It abstracts cloud provider APIs, provisioning orchestration, and the execution of remote shell commands. The DSL exposes all relevant configuration options in a declarative and easy-to-understand manner. Vagrant provides open source plugins for all relevant IaaS providers. The CWB web interface integrates a minimal web IDE with syntax highlighting for the Vagrant DSL.

Choosing Opscode Chef with a dedicated Chef server as provisioning service provides us with a flexible way to install and configure benchmark components in a reusable manner by exploiting Chef attributes. Experimenters can reuse software components (*e.g.*, database installation and setup code) in terms of cookbooks from the worldwide Chef community, and easily share benchmark infrastructure code with others. Furthermore, Chef integrates particularly well with Vagrant. The attribute passing mechanism from Vagrant to Chef allows to build configurable and thus reusable benchmark cookbooks. Since both, Chef and Vagrant, use an internal Ruby DSL, they not only ensure language consistency across the project but also offer the capabilities of a fully featured programming language that is exploited with the use of variables and utility functions.

The current version of CWB is available as an open source project on Github[4], including samples and installation instructions for an automated installation using Vagrant.

[3] `http://rubyonrails.org/`
[4] `https://github.com/sealuzh/cloud-workbench`

3. DEMONSTRATION

During the demonstration we will show how a sample benchmark is defined, scheduled, and executed with Cloud WorkBench. We use the CPU test of the SysBench[5] tool suite that calculates prime numbers up to a configurable maximum workload.

Defining the Benchmark Cookbook: We will show the key extracts of the provisioning code that is captured in a Chef cookbook and covers the installation and configuration of SysBench. The entire code being responsible for installing SysBench is shown in Listing 1. This code snippet exemplifies the cross-platform capabilities of IaC-based benchmark installation. Adding additional platforms is easy and often supported by Chef utilities out-of-the-box such as here where `package "sysbench"` is automatically translated to the respective package manager command of the underlying platform (*e.g.*, `apt-get install -y sysbench` on Debian or `yum install -y sysbench` on CentOS).

```
case node["platform_family"]
  when "debian"
    # Update package index
    include_recipe "apt"
end

package "sysbench" do
  action :install
end
```

Listing 1: SysBench Benchmark Installation via Chef

Specifying Cloud Resources: Once we have shown how the SysBench installation is described in a Chef cookbook, we switch to the CWB web interface for the cloud VM configuration. We will demonstrate how the CWB web interface facilitates creating variations of a previously defined benchmark by cloning an existing benchmark definition.

Figure 4 shows the cloud VM configuration for the sample benchmark which is intended to run in the Amazon EC2 cloud. This so called Vagrantfile specifies the desired cloud resources, references the SysBench Chef cookbook, and passes optional benchmark parameters for provisioning. The first section in Figure 4 defines that the VM to be used should be launched in the Amazon EC2 cloud, within a European data center, on a micro instance type, and with a bare Ubuntu 14.04 VM image.

Parametrizing the SysBench Provisioning: The second section in Figure 4 references the SysBench cookbook via `add_recipe` and takes advantage of the optional benchmark configuration parameters. Hereby, within the `sysbench` namespace, it is explicitly specified that SysBench uses the CPU test mode, sets the workload to 2000 (*i.e.*, calculating calculate prime numbers up to 2000), and runs in a single thread. Via the `cli_options` parameter, arbitrary command line arguments that are supported by the SysBench tool can be passed as key-value pairs.

Scheduling the Benchmark: We will demonstrate how to create and activate a schedule that periodically triggers executions of the SysBench benchmark using the cron expression syntax[6]. In our example, the expression `15 0,12 * * *` triggers a new execution of the benchmark a quarter

[5] `http://manpages.ubuntu.com/manpages/utopic/man1/sysbench.1.html`
[6] `http://linux.die.net/man/5/crontab`

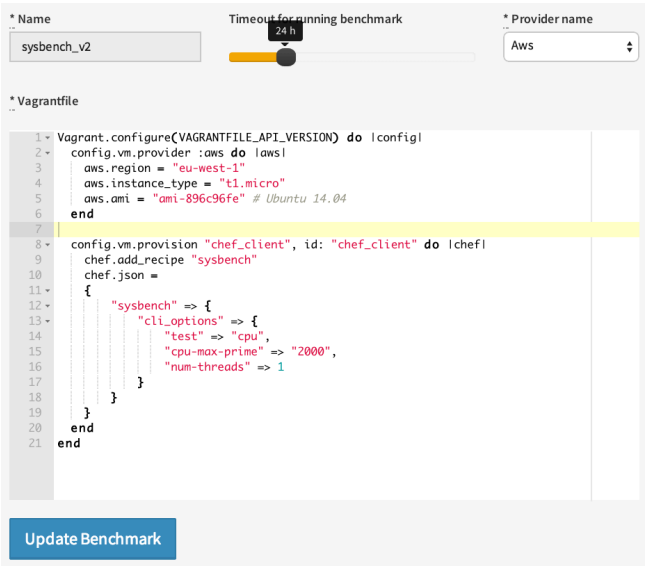

Figure 4: Cloud VM Configuration for SysBench

past midnight and lunchtime, every day of the month, every month of the year, and every day of the week.

Executing the Benchmark: We will manually start the execution of the previously adapted benchmark and demonstrate how CWB supports the experimenter in tracking the current status of the benchmark execution. Figure 5 gives an overview over a successfully finished execution of the SysBench benchmark. The top bar summarizes the current status of the execution (*i.e.*, finished), the duration of the actual benchmark (*i.e.*, less than a minute), and the duration of the entire execution (*i.e.*, 3 minutes) including preparation and termination. Preparation includes the VM startup time and time to provision (*i.e.*, install and configure) the benchmark, and termination includes post-processing and submission of benchmark results. The timeline underneath the top bar visualizes the key events of an execution in a comprehensible manner.

Showing the Results: Finally, the resulting metric that was generated in the previous execution will be shown. For further analysis of the results, the metrics associated to a benchmark definition can be downloaded as a CSV file.

The demonstration is available as a screencast on Youtube[7].

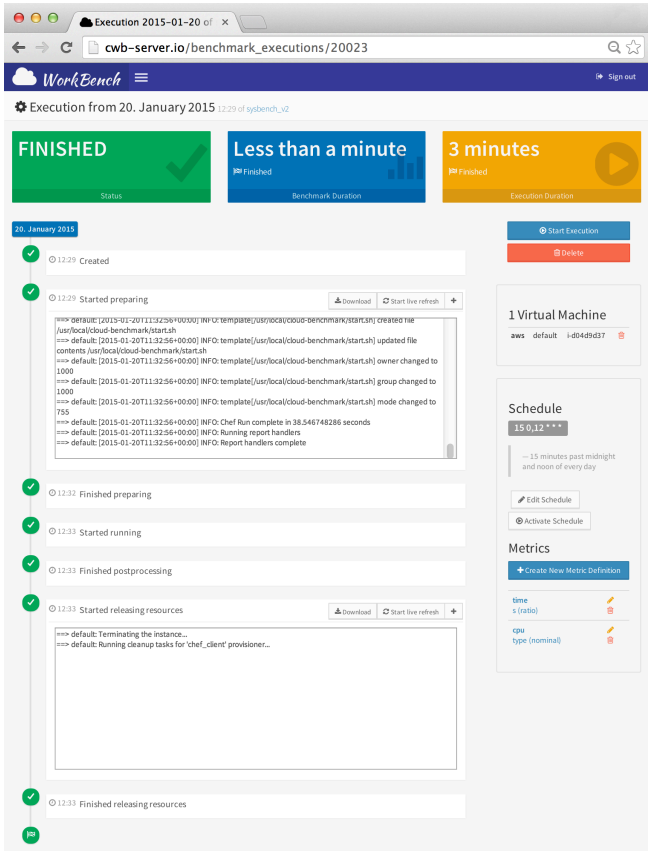

Figure 5: Finished SysBench Benchmark Execution

4. ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 610802 (CloudWave).

5. REFERENCES

[1] M. Cunha, N. Mendonça, and A. Sampaio. A Declarative Environment for Automatic Performance Evaluation in IaaS Clouds. In *6th IEEE International Conference on Cloud Computing (CLOUD)*, 2013.

[2] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift. More for Your Money: Exploiting Performance Heterogeneity in Public Clouds. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC'12)*, 2012.

[3] L. Gillam, B. Li, J. O'Loughlin, and A. Tomar. Fair Benchmarking for Cloud Computing Systems. *Journal of Cloud Computing: Advances, Systems and Applications*, 2013.

[4] M. Hüttermann. *DevOps for Developers*. Apress, 2012.

[5] D. Jayasinghe, J. Kimball, S. Choudhary, T. Zhu, and C. Pu. An Automated Approach to Create, Store, and Analyze large-scale Experimental Data in Clouds. In *14th IEEE International Conference on Information Reuse and Integration (IRI)*, 2013.

[6] P. Leitner and J. Cito. Patterns in the Chaos - a Study of Performance Variation and Predictability in Public IaaS Clouds. *CoRR*, abs/1411.2429, 2014.

[7] P. Mell and T. Grance. The NIST Definition of Cloud Computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), 2011.

[8] J. Scheuner, P. Leitner, J. Cito, and H. Gall. Cloud WorkBench - Infrastructure-as-Code Based Cloud Benchmarking. In *Proceedings of the 6th IEEE International Conference on Cloud Computing Technology and Science (CloudCom'14)*, 2014.

[9] M. Silva, M. Hines, D. Gallo, Q. Liu, K. D. Ryu, and D. Da Silva. CloudBench: Experiment Automation for Cloud Environments. In *IEEE International Conference on Cloud Engineering (IC2E)*, 2013.

[7] `http://youtu.be/0yGFGvHvobk`