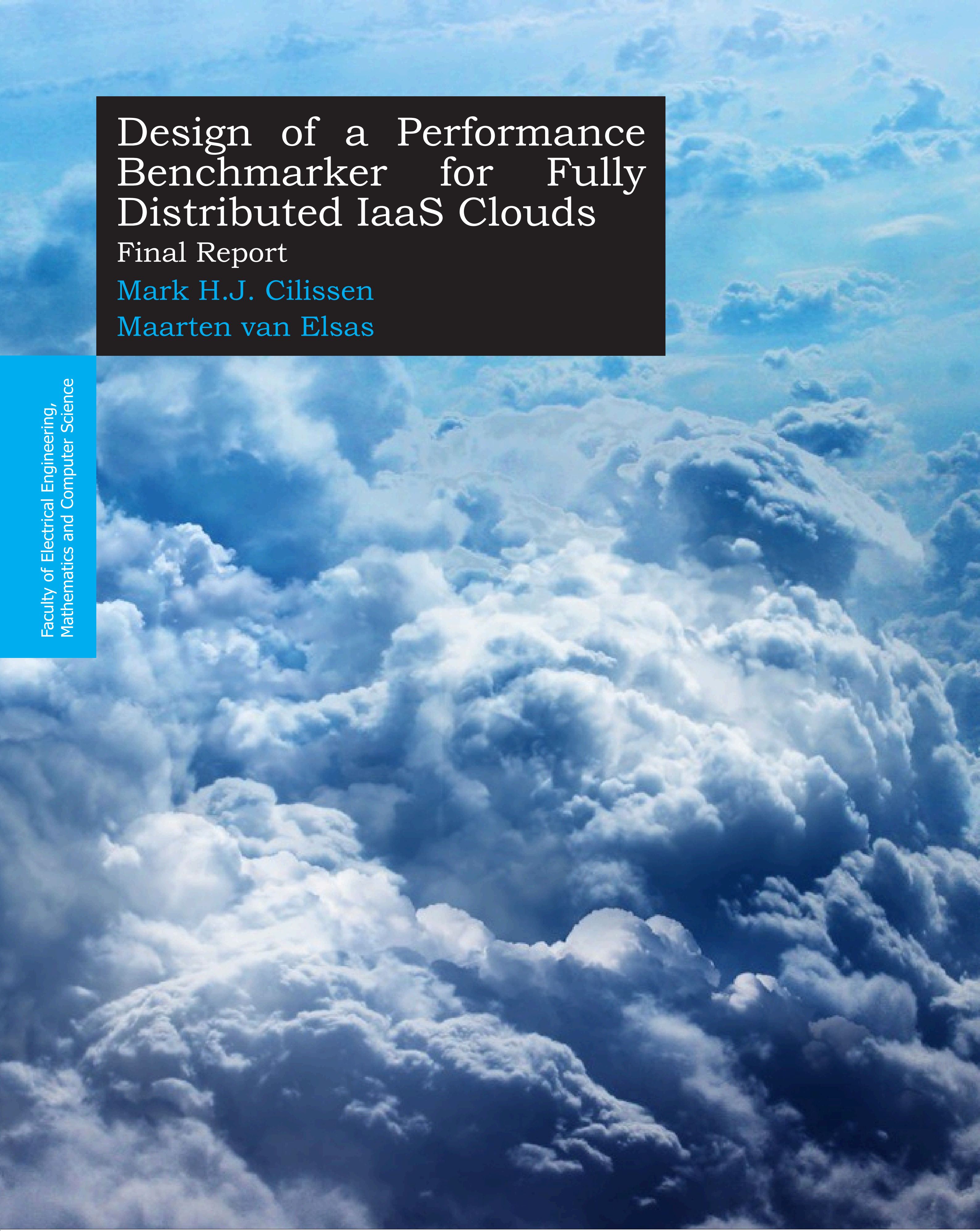


Design of a Performance Benchmarker for Fully Distributed IaaS Clouds

Final Report

Mark H.J. Cilissen

Maarten van Elsas

Faculty of Electrical Engineering, Mathematics and Computer Science

Delft University of Technology

Challenge the future

Design of a Performance Benchmarker for Fully Distributed IaaS Clouds

Final Report

by

Mark H.J. Cilissen
Maarten van Elsas

in partial fulfillment of the requirements for the degree of

Bachelor of Science
in Computer Science

at the Delft University of Technology,
to be defended publically on Wednesday September 30th, 2015 at 14:00.

Project duration:	July 6, 2015 – September 16, 2015	
Project supervisors:	Dr. ir. A. Iosup	TU Delft
	E. Feliksik, M.Sc.	Nerdalize
	M. de Meijer	Nerdalize
Project coordinator:	Dr. ir. M. Larson	TU Delft

This thesis is confidential and cannot be made public until December 31st, 2016.

Preface

This thesis is the conclusion and survival of our work done over the summer at Nerdalize, in the context of the final Bachelor Project in Computer Science, TI3800. A mandatory course in the Computer Science undergraduate program at Delft University of Technology, and usually the final project, it puts students into a situation of solving real problems for real companies, while maintaining a requirement of academic purpose. This unique combination leads to conflicting requirements from the company and university stakeholders at times, which is up to the students to weigh and balance.

Over the course of this project we learned a lot about working in a team, having real product owner interactions, dealing with real-life limitations and – most important of all – delivering a product that is used in production environments. Our hope is that this thesis gives insight into our development and thought process, endured difficulties and successes, and provides any potential future developers at Nerdalize background, documentation, and pointers to future improvements for the final product from this thesis.

Mark H.J. Cilissen
Maarten van Elsas
Delft, September 23, 2015

Summary

Nerdalize B.V. is an infrastructure-as-a-service (IaaS) cloud provider aiming to offer substantially lower prices than its competitors. In order to visualize its cost savings to customers and measure its own systems against competitors in a cloud market reigned by opaque pricing models, it would like to utilize an application benchmarker to give customers insight into the resource utilization and operation costs of their applications among various cloud providers.

We have done research into the fields of cloud computing, benchmarking and the intersection thereof, determined requirements for such a benchmarker, and assessed any existing solutions in the field. We then chose Nerdalize's internal prototype implementation as a suitable base to develop a fully featured, production-ready application benchmarker. We identified five main design goals of correctness, robustness, security, extensibility and maintainability.

We then analyzed and prioritized potential improvements and extensions to this prototype, and implemented them in an agile-driven Extreme Programming (XP) development process. The main contributions lie in designing and implementing a fully automated test suite and system, vastly improving the accuracy and stability of the benchmarker, re-designing the deployment model from monolithic to modularized and extensible, implementing support for provisioning to arbitrary Linux-based hosts, and deployment of complex workload architectures.

We then experimentally verify the accuracy of the benchmarker, and assess that its deployment overhead is very small to neglible, and run several tests against real-world cloud providers. The end result of this project is a stable, well-tested, featured benchmarker application that is used in production environments at Nerdalize.

Acknowledgements

We were helped by many people during the course of this project, but would like to thank some of them specifically.

First and foremost, we would like to thank Eric Feliksik, Director of the Cloud Orchestra, and Mathijs de Meijer, CTO, at Nerdalize for their patient guidance before, during and after the project, as well as their constant feature feedback, allowing for a truly agile development process to happen. In particular, we would like to thank Mr. Feliksik and Mr. Schoute, hardware engineer, for their off-hours guidance in progressing in the company's internal table football league.

Furthermore, we would like to thank our supervisor at Delft University of Technology, Dr. Alexandru Iosup, for his clear requirement analysis upfront and invaluable thesis, project and process feedback.

Finally, we would like to thank everyone we interacted with during our stay at Nerdalize for their unquestionable kindness, support and help, and finally allowing us to trial our thesis defense during their usual socialization times.

Table of Contents

List of Illustrations

Figures

Tables

1 Introduction

Nerdalize B.V. is a Delft-based technology startup that aims to provide infrastructure-as-a-service (IaaS) cloud platform services for substantially lower prices than its competitors. It does this instead of centralizing servers in a data center and spending significant amounts on cooling, distributing its compute nodes over houses, and using the generated heat to warm them. This allows for savings on operational cost, which can be used to both offer a cheaper cloud service and free heating for home owners.

Nerdalize would like to visualize and contextualize its cost savings to cloud consumers. However, pricing models in the cloud market are opaque, and providers work hard to keep their pricing algorithms secret. Additionally, cloud infrastructures are heterogenous, and can scale up and down as needed while influencing the price model, making predictionary cost analysis difficult. Finally, user applications are heterogenous too, and analysis models that may work for a certain type of application may not work for different types with different resource needs.

To this end, Nerdalize would like to utilize an application benchmarker: cloud consumers can supply representative 'slices' of their application, and Nerdalize will use this benchmarker to run it across several cloud providers, including itself. Resulting in them being able to provide insight into the resource utilization and cost of this application across all these providers. This allows Nerdalize to engage in client bonding and give real insight, backed by data, into the potential savings. Furthermore, Nerdalize can use this application internally to measure if its own services are still competitive with regards to other cloud providers, and possibly adjust its pricing model or hardware as a result.

Structure

Numerous cloud infrastructure benchmarker implementations already exist, but are unsuitable for running arbitrary user applications. This is expanded upon in chapter 2. Nerdalize already has an in-house prototype implementation that, while very basic, is suitable for running arbitrary user applications, and thus expansion upon. In chapter 2 we also investigate suitable improvements for the current prototype, which is the main subject of this project.

In chapter 3, we discuss the main design contributions for implemented improvements, including the overhaul of the deployment model, support for additional provisioning and deployment methods and the setup of a fully automated testing environment.

We then discuss the main implementation challenges in chapter 4, including the implementation of provisioning to arbitrary machines, improving the reporting accuracy and the numerous difficulties we ran into when trying out various test automation environments.

In chapter 5, we describe the testing process, and how we approached testing in not simply a qualitative approach, but also through performance and overhead metrics. In specific, we expand on the testing of measurement accuracy and deployment overhead.

We evaluate whether or not we have met the design goals set out in chapter 2 and provide insight into the development process in chapter 6. We also describe possible future improvements that could be made to make the benchmarker even more widely-applicable and insightful.

Finally, in chapter 8 we conclude and summarize the process and answer the initial research questions posed at the start of the thesis.

2 Research Study on Cloud Benchmarking

2.1. Overview

The first two weeks of this project have been fully devoted to research. We have done investigation into the problem background and scope, existing literature on the matter, existing implementations which may help in solving parts of the problem, project and infrastructure constraints and the feasibility of both existing and novel solutions.

In section 2.2 we discuss and analyze the project problem statement. From here, the necessary background information for the matter is discussed in the next three sections. In section 2.3 we give an overview of the general basics and ideas behind the concept of "The Cloud". We give necessary background information on benchmarking in section 2.4. Finally in section 2.5, we discuss the specifics and issues inherent to running benchmarks on cloud infrastructures.

Having discussed the necessary background information, section 2.6 moves on to describe the current available solutions, and why we choose Nerdalize's existing benchmarker as our base. During the research phase, these solutions, together with Nerdalize's infrastructure, were analyzed and taken into account while setting the design goals. We describe those goals in section 2.7. From the infrastructure and literature analysis, we formed a number of possible extensions to the current infrastructure. Those extensions, including the reasoning for including or rejecting them for this project, are listed in the final section 2.8.

2.2. The Problem

To be able to solve a problem it must first be well-defined. In this section we sketch the context in which the problem resides and give the problem definition. We then analyze the problem to lay the foundation for the rest of this report.

2.2.1. Problem Definition

Nerdalize offers Compute as a Service with a competitor-based pricing model. This means Nerdalize provides Infrastructure as a Service with an architecture that will be most attractive for CPU-intensive batch jobs, at a cost lower than alternative cloud-providers.

The Nerdalize cloud architecture deploys 4 rack servers in a household. The households are interconnected with a VPN via using consumer-grade internet connectivity. Although this creates some new infrastructural bottlenecks as compared to a centralized Data Center approach, Nerdalize will have an advantage in terms of operational cost of the servers. This advantage in cost is used to offer a cheaper service.

Customers want to easily compare the cost of running its compute batch jobs at different cloud providers. However, the cloud market is opaque:

- Heterogeneous infrastructure: different virtualization technology, CPU, networking, and disk types are used.
- Heterogeneous cost models: different prices and models for machine resource and time consumption
- Heterogeneous computational jobs: different jobs have different performance profiles (resource utilization and the influence of infrastructure on turnaround times).

A benchmark that uses a representative computational job — as defined by the customer — can help the customer gain insight in the cost of its example job at various cloud platforms.

A first version of the benchmark system has been implemented in Python, using Docker to deploy workloads on Cloud instances. Along with the workload a monitoring agent is deployed on the instance, which reports resource utilization to the benchmark orchestrator. After the workload is finished, reports are generated that show the resource utilization, runtime, amount of disk and network I/O performed, and the associated cost. To calculate the cost, a cost model is implemented that is modeled per cloud provider.

The goal of the project is to improve this benchmarker to be more generally useful for different situations and circumstances.

2.2.2. Problem Analysis

Nerdalize is interested in a benchmarker for running customer-supplied applications on various Infrastructure-as-a-Service (see section 2.3.5) providers. By obtaining the results of the various metrics, a cost model for their clients can be derived for easy comparison. Nerdalize can use this cost model to undercut their competitors and for internal reference, as their operational costs are lower due the nature of their servers.

Interviews with employees show they currently have an existing benchmarker implementation developed in-house, but it is very basic and not quite suitable for production deployment yet. It is unclear whether this is a suitable base to develop a more advanced benchmarker from, or if there exist better current solutions to use. We will explore this in section 2.6. If no suitable base can be found, developing the benchmarker from scratch is another option.

Vital to the implementation of the benchmarker is the ability to not just run predetermined benchmark suites with predetermined metrics, but arbitrary user applications, even if they misbehave. This is an aspect that makes this a unique project worth extra consideration.

The requirements as set forth by Nerdalize lead to the formulation of the following research questions:

1. How can one measure the performance and cost difference of running an arbitrary application across several cloud infrastructure providers, and different instance types within these providers?
2. How can one verify the accuracy of such a measurement tool?

2.3. Background on Cloud Computing

In recent years, the usage of the term "The Cloud" has become commonplace. Despite the recent emergence of this trend, some of the concepts from cloud computing date back to the ideas behind grid computing from the 1990s and earlier[1, 2]. In this section, we make an attempt to give a narrower definition of the term "cloud computing", as well as an explanation of the basic concepts and how it differs from earlier paradigms.

2.3.1. Origins

Advocates of cloud computing promise a world in which computational and storage resources are not typically owned by the vast majority of its consumers, but instead are centralized in massively transparently scalable resources operated by third parties for rent[1]. The idea of such resources as an utility is not new by any means: in the 1960s and 1970s mainframes were a very common method to centralize computation for easy use by individuals who would otherwise have no access to such resources. Mainframes are not an appropriate paradigm to compare cloud computing with.

One difference between cloud computing and mainframes is that the former allows the resources to scale at will, thanks to advancements in virtualization technology: Foster *et al.* [2] define Cloud Computing as "*a large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet*". This clearly differs from typical mainframe operations — in that while the resources *appear* centralized — they are actually distributed over a large amount of virtualized machines. This allows for massive transparent scaling benefits that mainframes do not offer.

An additional difference is that mainframes were often only accessible for individuals with the proper credentials: university researchers or employees at large companies are the two most prominent examples of this. In contrast, following the definition given by Foster *et al.*, any person with an accepted payment method can order and utilize these resources.

2.3.2. The Grid and the Cluster

Because of these differences, it is perhaps more useful to find a different paradigm to compare cloud computing to. Buyya *et al.* [1] propose a comparison of cloud computing with "grid computing", which they define as "*a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed 'autonomous' resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements*".

This paradigm draws more analogies to cloud computing, decentralizing the actual computational resources. However, it does not take into account the appearance of the distributed resources as a single unified resource. For this, Buyya *et al.* consider an additional comparison with "cluster computing", defined as "*a type of parallel and distributed system, which consists of a collection of inter-connected stand-alone computers working together as a single integrated computing resource*"[3].

In fact, Foster *et al.* argue that the cloud computing paradigm *evolved* from grid computing, while Buyya *et al.* argue that it finds its roots in both grid computing and cluster computing.

2.3.3. Economy of Scale

It would be incorrect to state that cloud computing is merely a *combination* of grid and cluster computing. An important aspect of cloud architectures, as noted by Buyya *et al.* [1], is the market interaction with customers in order to be able to provide for more generalized needs.

As customers rely more on cloud computing to satisfy their resource requirements, the quality-of-service (QoS) resource allocation models need to change in order to adapt to per-customer, per-job and even in-job QoS requirement changes.

Instead of simply treating every resource aspect for every customer as equal, cloud providers need to be able to dynamically adapt and scale their resources, to provide an optimal allocation at all times. This is an important factor in the often-cited[4] high scalability of cloud architectures, a prominent property which sets cloud computing apart from *merely* being a combination of grid and cluster computing.

2.3.4. Automation

The other defining characteristic of cloud computing — as set forth by Wang *et al.* [5] — is the degree of automation involved and the amount of influence by the consumer in the matter. Large cloud providers offer programmatic interfaces (APIs)[1][2][3], so provisioning and deployment of computing instances can be fully automated. This provisioning is often instant and on-demand[5], with no manual intervention required on either side[6].

2.3.5. Service Architecture

Cloud computing architectures can vary in abstraction levels, in order to provide maximum flexibility. Mell and Grance [7] describe the service architecture in three layers, shown as a hierarchical pyramid in figure 2.1:

- **Level 1: Infrastructure-as-a-Service (IaaS)**
  The lowest level in cloud services. On this level, the physical resources such as processing power, physical memory, long-term storage, networking et cetera are provided and abstracted by the cloud provider. Examples include Amazon Elastic Compute Cloud (EC2) and Google Compute Engine.

- **Level 2a: Platform-as-a-Service (PaaS)**
  One level above IaaS, the user is provided with "development and administration"[6] platforms that operate on top of the virtualized hardware resources. These may include programming language environments and runtimes. Examples include Amazon Elastic Beanstalk and Google App Engine.

- **Level 2b: Data-as-a-Service (DaaS)**
  In parallel with PaaS, data-as-a-service architectures provide the user with database and file storage engines, which are typically accessed from IaaS or PaaS services. Examples include Amazon Relational Database Service and Google BigQuery.

- **Level 3: Software-as-a-Service (SaaS)**
  The final and highest level in the "cloud architecture pyramid", in software-as-a-service architectures the provider runs a full software stack on top of the PaaS and DaaS layers. The management of this application is the responsibility of the provider as a service to the user. This can optionally be combined with lower layers of the pyramid to provide ready-made applications as part of a larger system. Examples include Google Apps and Microsoft Office Online.

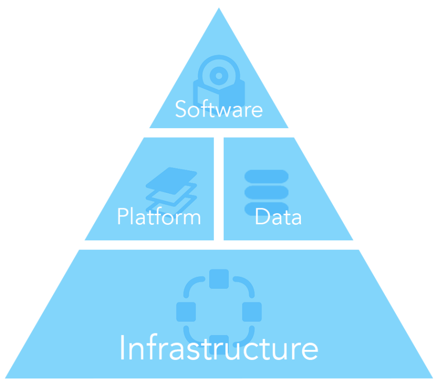


Figure 2.1: A visual representation of the cloud architecture "pyramid".

[1]http://docs.aws.amazon.com/AWSEC2/latest/APIReference/Welcome.html
[2]https://cloud.google.com/compute/docs/reference/latest/
[3]https://msdn.microsoft.com/en-us/library/azure/ee460799.aspx

A higher level in this pyramid amounts to a higher amount of abstraction for the customer, but also more limited possibilities in customization and self-deployment. The very lowest level abstracts only the hardware, while the highest level abstracts away the entire application stack.

2.3.6. Deployment Architecture

Furthermore, cloud computing deployment models can differ. Mell and Grance [7] describe the possible deployment models in terms of ownership of the infrastructure and accessibility by customers:

- **Private cloud:** The infrastructure for a cloud of this kind is privately managed, for exclusive use by one or multiple entities. It may physically be located on or off organization premises, and can either be managed and operated by the organization itself, a third party, or a combination.
- **Community cloud:** This infrastructure is publically managed, within a community, for exclusive use by members. This community consists of members that have some kind of shared concern. Like a private cloud, it may be physically located on or off community premises, and can be managed and operated by either members of the community itself, a third party, or a combination.
- **Public cloud:** This infrastructure is available to the general public. It is managed and operated by a business or government organization and is located on the premises of this provider.
- **Hybrid cloud:** This infrastructure is a composition of two distinct earlier-named infrastructures that are still separate, but are joined together by a technology that allows for application and data portability between these infrastructures.

2.4. Background on Benchmarking

Benchmarking a system is measuring and evaluating the performance and other "non-functional characteristics" of a system, called the system under test (SUT)[8]. A traditional use for benchmarking is comparing various systems to each other by the use of a special benchmarking application. Recently benchmarking has also seen prominence in areas of system design and tuning[8]. In this section we will briefly discuss the basics of benchmarking.

2.4.1. Utility

As stated in the introduction, a very common use of benchmarking is using a single or multiple workloads and running it across different kinds of systems, in order to get an overview of the performance characteristics of these systems. These workloads are often specifically designed to test certain aspects one at a time.

Recently, benchmarking has also seen a rise in use in the areas of system design and optimization. By periodically benchmarking an application on various configurations, it may be possible to weed out performance bottlenecks early in resource-critical applications.

Following from these examples, benchmarking can essentially be divided in two purposes: finding and comparing the performance characteristic for a variety of infrastructures, and finding similar characteristics for a certain application. In both cases the 'other' factor is a known or fixed value.

2.4.2. Requirements

Designing a good benchmark can be a challenge. Folkerts *et al.* [9] describe requirements for a good benchmark, collected from a number of sources but primarily from Gray [10]. These requirements are divided into three categories:

- **General requirements:** A benchmark should have a tight target audience, be relevant, economical and simple.
- **Implementation Requirements:** A benchmark should be fair and portable, repeatable, realistic and comprehensive, and configurable.
- **Workload Requirements:** The workload for a benchmark should be representative, scalable, and contain one or more metrics.

Balancing these requirements can be a challenging task, and even common well-used benchmarks may do concessions in some aspects in favor of others.

2.4.3. Workloads

A workload is "the operational load to which the SUT is subjected"[8]. Two different kinds of workloads can be distinguished: simple ones where a single application is combined with a job arrival process, and complex workloads where multiple users potentially use different applications that are dissimilar[8].

An example of a simple workload is a bag of tasks (BoT), which is a set of tasks that are similar but can be executed independently. An example of this is running a single application with different parameters. BoTs are very common in scientific computing[11], for instance for the purpose of folding proteins.

A complex workload on the other hand could be one user hosting a website and another one folding proteins on the same system, or running multiple dissimilar bags of tasks at once.

2.4.4. Tasks

There are a number of approaches to individual benchmark tasks, each with their own advantages and downsides. They will be briefly discussed in the following subsections.

Microbenchmarks

A microbenchmark analyzes a specific component or action of a system, like writing a value to memory or performing a floating point operation. There are a few challenges in performing microbenchmarks: the amount of time taken by a single one of these operations is so low that measuring it is impossible on real systems. Instead, many identical operations are performed and the average time of a single operation is measured.

This in turn introduces a new problem where compilers optimize the code, resulting in a unrepresentative runtime per operation[12]. Some compilers allow turning off optimizations, but even then the operating system may influence the benchmark by, for instance, caching a file into RAM when performing a disk input/output benchmark.

Kernels

In a kernel benchmark, the most CPU-intensive part of an application is extracted and run independently. Gray [10, Chapter 9] has shown that this is a valid approach by determining that, on average, 10% of the code uses 80% of the resources.

This method too has several downsides. Aside from also being subject to compiler optimizations, only the CPU resource usage ‘decides’ the part of the application to extract. As a result, the kernel may not be representative of the application for any metric but CPU usage. In addition, depending on the application architecture, the process of kernel extraction may be non-trivial.

Applications

Another way to benchmark a SUT is to run the actual applications on it. If more information than simply the running time is desired, the SUT can be monitored for all the required resource metrics while the application is running. This method can result in a good estimate of the runtime if the application runtime is consistent[8].

While accurate, this approach has the downside of requiring a full application install, with all possible prerequisites. Moreover, the benchmark size and scale is barely flexible or non-flexible depending on the application, and often requires a ‘full application run‘ to produce workable results. Finally, the benchmarked applications can introduce side-effects upon being run, which would make repeated running problematic.

In Nerdalize’s problem, user applications is what they wish to run on their infrastructure, and what we would need to benchmark to get the most accurate results. It is not possible to have repeated runs of the same application in all cases, since user applications can, in the example case of scientific computing software, grab their jobs from a centralized job server and ‘consume’ them. This would make repeated runs of the same applications non-identical, since they may get different jobs with different performance profiles.

2.4.5. Performance Prediction

The performance of an application on a SUT can be somewhat predicted. One way to do this is to take the microbenchmarks of a system and make a vector out of these. Then the application itself is profiled to make a characteristic vector of the application. This vector has a value for the application’s relative use of all of the microbenchmark characteristics. By taking the dot product of an application vector with a system vector and normalizing the result, an estimate of the runtime of the application on different systems can be retrieved[12]. There are several challenges in vector based benchmarking. It has to be assumed that the application can be accurately profiled, which is not always the case, for instance because one decision branch is much more work than another. Additionally, a universal set of operations to microbenchmark and a standardized way to measure them has to be devised.

Another common method of predicting performance is to take a similar application or workload and assume the performance will be equivalent. For this purpose synthetic benchmarks are created in order the characterize a group of applications or workloads.

Analogous to the vector method is measuring a benchmark on all systems and the application on a single (consistent) one. Then the system vectors are approximated using a genetic algorithm [13]. The advantage of this method is that only a single benchmark needs to be run in order to be able to characterize the system. For a fair number of different systems, same or similar benchmarks are available.

2.5. Challenges in Benchmarking Cloud Infrastructures

The distributed and transparently scaling architecture of cloud computing makes it difficult to directly apply common benchmarking knowledge and tools to cloud infrastructures. In this section we will discuss some of the specific issues, along with potential solutions and new techniques for result analysis and processing.

2.5.1. Transparent Scaling

Due to the transparent scaling property of cloud architectures as shown in section 2.3.3, it can be hard to exactly figure out the bottlenecks of a cloud system. The supervising system may decide to utilize more hardware on-the-fly to meet the computational demands of a benchmark, which makes finding limits of a certain system hard or even impossible. A possible solution here would be to artificially limit the bottleneck in the benchmark, in order to keep the benchmark economic[9].

Contrarily, the infrastructure may also decide to utilize less resources when the benchmark has just started, or in a period of low computational demands. This 'turn-around' time between the start of more intensive calculations and the up-scaling of the resources may also influence the result negatively[8]. A possible approach to minimizing this issue would be to run the benchmark numerous times in succession on the target system. This may eliminate the start-up scaling issues while maintaining the otherwise realistic environment of a cloud system, where dynamic down- and upscaling is a matter that has to be taken into account for benchmarking, as it will affect performance in the "real world" too.

2.5.2. Consistency

The multi-tenant nature of cloud architectures discussed in section 2.3.1 leads to a consistency issue: while transparent scaling strategies are utilized to minimize large performance hits, the current other jobs processed by one or more cloud instances cause the performance of a benchmark to vary unpredictably in most metrics[9, 14]. This is especially hurtful for comparative benchmarks, where small to medium performance differences between providers might greatly influence the outcome.

A proposed solution involves running the benchmark multiple times over an extended period of time, and comparing the different results from these runs. Iosup *et al.* [14] have observed that while there is non-trivial performance variability, common cloud providers also have periods of stability. If there is too much variability between measurements, it can be useful to do additional runs over an extended period in order to rectify the discrepancy in the dataset.

This method can be combined with running the benchmark multiple times in parallel, so the instances would be deployed over different (identical in configuration) hardware instances. This way an unusual load on a specific system would not influence the measurement disproportionately.

2.5.3. Additional Metrics

The unique aspects of cloud architectures lend to some interesting new metrics that have to be taken into account to get a good overview of the performance. The believed to be more important ones are listed and briefly described.

Elasticity and Scalability

The aforementioned scaling property exhibited by cloud architectures may negatively affect existing benchmark metrics, but measuring scalability and elasticity on their own is an interesting consideration metric when benchmarking cloud architectures. For customers with applications whose resource demands can fluctuate wildly, the capability of the infrastructure to scale on-demand and within reasonable time is an important consideration[15]. Kossmann *et al.* [16] note that while cloud architectures offer the illusion of 'infinite scale', this does not match with the practical architectures that cloud providers deploy, thus it is an important metric.

Hwang *et al.* [15] offer some possible cloud performance models which include this metric, comparing the different properties of various scaling methods, and show that scalability is directly proportional to overall productivity, making it an important metric to include. However, their approach might be considered too complicated and involved with domain knowledge for some.

Contrarily, Folkerts *et al.* [9] approach the issue by only modeling for an increase of application instances per node, keeping benchmark analysis relatively simple. Deploying multiple instances of a benchmark on the same node would be a good approach to give an approximation of the scalability and elasticity of the target cloud architecture.

Deployment Latency

Another new potential metric in cloud architectures is the time it takes to deploy the application onto the cloud infrastructure[11]. This deployment latency, while easily measurable, can give some valuable insight into the performance of the supervising and orchestrating systems of the cloud provider. In addition, being able to respond quickly to changing requirements is an important requirement in a lot of applications. Take for instance a security vulnerability, where it is often desired to immediately patch the application upon discovery. Every minute that an infrastructure takes to deploy a new version of the application is a minute that a malicious attacker can use to exploit the vulnerability.

2.6. Existing Benchmarking Solutions

It is good engineering principle to not needlessly repeat any past work that has already been researched and implemented. In this section, we will discuss some of the existing solutions in this field , and their suitability for the problem analyzed in section 2.2 will be addressed. Finally, a decision will be argued on whether to base the benchmarker for this project or an existing solution, or to write it from scratch.

2.6.1. CloudCmp

CloudCmp is a cloud benchmarker and systematic comparator developed by Li *et al.* [17] with the explicit purpose of allowing consumers to review and compare cloud services for their needs. Emphasis is placed on three metrics: runtime, cost and deployment latency. The runtime metric determines how much time it took to run a certain workload on each provider. The cost metric is the total monetary cost, measured using a provider-specific cost model, required to run a certain workload on each provider. Finally, the deployment latency metric refers to the delay in creating new instances: see section 2.5.3. It uses a set of predefined workloads to measure various specific aspects of the supported providers.

Despite seeming very promising at first, it is considered unsuitable for the project goal since the intention is to run user-supplied applications or kernels to get the most exact metrics for a specific client situation (section 2.2.2). The flexibility to add benchmarks without adding new code to the benchmarker application should be considered vital as a result, and CloudCmp can not offer this flexibility.

In addition, the project seems abandoned: the project website[4] is not available at time of writing, and the project code repository[5] was last updated over three years ago, and never got past version 0.1.

2.6.2. Google PerfKit

Google PerfKit is a relatively new tool, open-sourced by Google [18]. It consists of PerfKit Benchmarker and PerfKit Explorer. The former allows one to run actual benchmarks, while the latter performs visualization of the results. It is cross-platform and has built-in support for all the common cloud providers. The visualization is very user-friendly and allows for a historic metrics overview and export of data.

However, it has the same main issue CloudCmp does: benchmark workloads are hardcoded into the benchmarker, losing immense amounts of flexibility, which is not considered appropriate for this project. The existing predefined benchmarks seem biased towards microbenchmarks that measure a specific aspect of the system, and may be harder turn into a full application profile. This is especially true as the workloads are grouped into existing synthetic benchmark suites, which are hard to adapt to a specific application.

In addition, the only supported way to set up PerfKit Explorer is by using Google's own cloud services. There seems to be no supported way of running the visualization system locally or on own servers, which may be important for confidentiality of benchmark data.

2.6.3. Nerdalize Benchmarker

Nerdalize currently has a custom in-house benchmarker implementation. Its architecture is described in figure 2.2. It is accessed through a web interface, denoted as the "FrontEnd'. It takes Docker[6] containers and deploys them using Docker Machine[7]. Along with the container, a monitor is deployed to keep track of several system metrics such as the CPU usage, runtime and bandwidth. The monitoring data is sent to an InfluxDB[8] database. From the metrics data, a price approximate is calculated using a provider-specific cost model.

The advantage is that this benchmarker actually measures specific applications and already implements Docker input, so that not only can any provider that supports Docker be measured, the benchmarker itself can be run from any provider that supports Docker, although the cost model wouldn't be able to be applied.

[4]http://cloudcmp.net
[5]https://github.com/angl/cloudcmp
[6]https://www.docker.com
[7]https://docs.docker.com/machine
[8]https://influxdb.com/

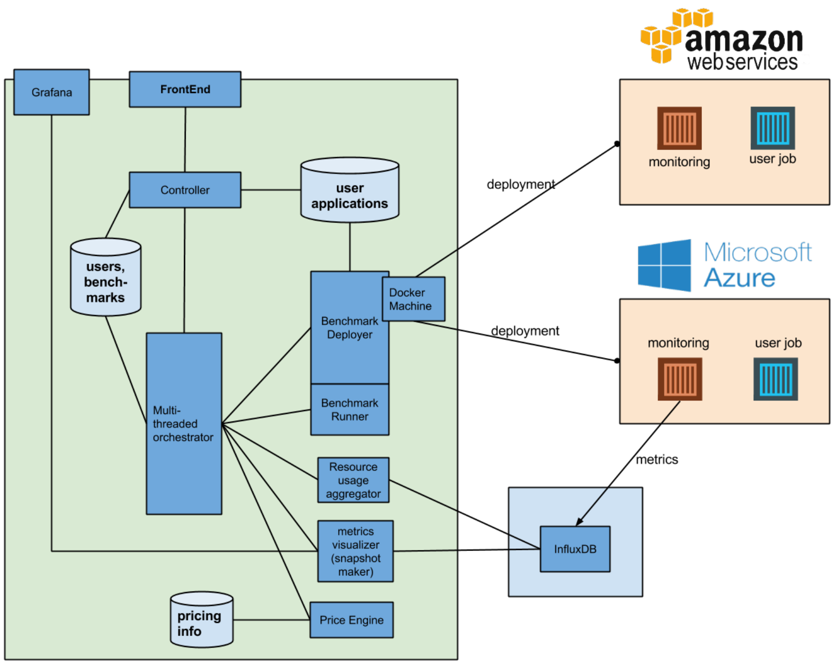


Figure 2.2: A schematic overview of the architecture of the Nerdalize Benchmarker.

Table 2.1: Comparison of the features of the reviewed implementations.

Name	**Last updated**	**Implemented in**	**Workload types**	**Overhead (expected)**	**Cost Model**	**Automated Provisioning**
CloudCmp	2011-12-12	Java	Only built-in suites	Large	Simple	AWS, GCE, Azure, RS
Google PerfKit	2015-09-10	Python	Only built-in microbenchmarks	Small	None	AWS, GCE, Azure, DO, RS
Nerdalize Benchmarker	2015-07-01	Python	Docker-packaged user applications	Small	Medium	AWS, Azure

Provisioning key: AWS: Amazon Web Service, GCE: Google Compute Engine, RS: RackSpace, DO: Digital Ocean

However, there are some points for improvement, both functionality and design wise. The deployment is currently limited to Docker and Docker Machine for provisioning, and metrics reporting is not very advanced yet. Moreover, its cost model is currently somewhat simple, albeit more advanced than CloudCmp's. However, it does seem specialized for specific applications and designed to be adaptable.

2.6.4. Conclusion

Having considered a few benchmarker implementations, we conclude that Nerdalize's own benchmarker is a suitable base to base further work on. We have summarized the features in table 2.1.

While being lacking in advanced features, its benchmarking philosophy is consistent with the aim of this project, and Nerdalize has own in-house knowledge about its architecture, resulting in the code base being easier to grasp, and modifications easier to make. We feel modifying the other benchmarkers to fit the vision is a harder task, especially for their more complex and not simply grasped code bases.

The project goal will thus consist of improving Nerdalize's benchmarker to bring it up to speed on the state-of-the-art features required by Nerdalize and its customers.

2.7. Design Goals

Having selected an implementation to base further work on, the principal design goals are now decided upon and explained. These goals serve as the pillars for the implementation and guide the design and re-design of the system. A primary design goal is identified, with four additional goals being decided on as secondary goals.

Correctness

The primary aim is to get accurate measurements of the application metrics on the system under test. It is important the code runs correctly in order to make sure that we do not get measurements that are wrong. The most important measurement errors to avoid are those that only impact certain cloud providers. It is more important that the comparison is accurate than that the actual values are. Furthermore, the benchmarker should work for as many applications as possible, since they are user-provided and thus not necessarily adaptable.

Robustness

The benchmarker deals with large-scale infrastructures and unpredictable user applications, so the possibility of failure is very real and has to be dealt with. It is very useful for both the consumer and Nerdalize to be notified of any failures that occurred while benchmarking, and if possible given information on the cause. Furthermore, the benchmarker has to be able to resume when a failure occurred if possible, so that a single failure can not halt the entire benchmarking process. Failure rates could finally be included as a provider metric.

Extensibility

The cloud market is still developing and varying. New cloud providers are still emerging. The jobs that are run and the way in which they can be run are also continuously expanding. The ways in which benchmarking in the cloud can be accurately done are an ongoing research area and as such, new methods can be expected. All of this leads to the requirement of having an application that is easily extensible, to be able to quickly and easily adapt to changes.

Security

Since the workloads provided are from the consumer, the biggest security concern is keeping them safe and isolated, since they might contain confidential or proprietary information. In addition, the credentials used to run jobs on different cloud providers need to be kept safe and secure, to prevent credential abuse leading to potential monetary losses. It's of concern that the measurements themselves are not influenced. Influence could consist of other applications running on the same system, or by forging measurement events.

Maintainability

Lastly, it is also important that our software in maintainable. This includes a clear and well-engineered benchmarker architecture, and building on supported technologies and frameworks. Ones that still get bug fixes and community support, to keep development focused on the benchmarker application proper. Finally, it should be easy to verify that a code change does not introduce any regressions, through for example automated testing and continuous deployment.

2.8. Extensions to the Nerdalize Benchmarker

The Nerdalize benchmarker has been chosen as a base for this project implementation, but it is basic and lacking in required functionality. In this section the possible extensions that can be implemented to its code base are described, to make it fulfill and exceed the project requirements. The proposed extensions have been discussed with the project employer, and the selected ones will be implemented as part of this project.

2.8.1. Proposed Extensions

Measure Deployment Latency

As described in section 2.5.3, the deployment latency is a useful and important metric for insight into the provider infrastructure and rapid deployment possibilities. This extension is also relatively trivial to implement.

Measure Elasticity

Elasticity as described in section 2.5.3 is a potential useful metric for determining the scale-up and scale-down possibilities of the target cloud infrastructure. This provides useful insight into the dynamic possibilities.

Deploy Complex Architectures

Deploying application architectures that consist of more than one container is currently not possible. Allowing for deployment of more complex architectures using a description format for containers and their interdependencies would allow for benchmarking more types of real user applications, that are not necessarily just a single service. Docker Compose could be of use for this as a standard format and deployment backend.

Deploy Cluster Architectures

Deploying application architectures over more than one instance node is currently not possible. Allowing applications to be run over a cluster of nodes would allow for benchmarking more types of scientific computing applications, where distributed computing is a more important common architecture. Docker Swarm could be of use for this to manage the deployment.

Extra Provisioning Methods

The current Nerdalize base benchmarker implementation can only deploy to the infrastructures supported by Docker Machine[9]. The deployment mechanism should be made generic and pluggable, so extending deployment to platforms not supported by Docker Machine is possible and relatively easy to implement. At least one extra deployment method should be implemented.

[9]https://docs.docker.com/machine/#drivers

Extra Provisioning Platforms

Currently, deployment is only supported to a select few cloud platforms. This can be extended to include more cloud providers, so they can be added to the comparison to give the customer a broader insight into the market. This is necessary to be able to keep up with market changes and emerging new providers. This involves not only possible deployment methods as above, but also updates to the cost model and authentication keys. At least one extra deployment platform should be implemented.

Extra Workload Formats

Instead of the customer having to provide a Docker image, it should be possible to utilize different deployment methods, such as AWS CloudFormation. Certain common applications types could also be allowed to be uploaded and automatically deployed, in a Platform-as-a-Service (see 2.3.5) fashion. The image input backend should be made generic and pluggable, and at least one extra deployment format should be implemented.

Benchmarking Performance Variability

As noted in section 2.5.2, cloud benchmarking can prove problematic for single measures, as performance of cloud architecture can vary over time and node. Additions to the benchmarker should be implemented so that it can automatically run benchmarks a number of times, possibly with a time delay, and aggregate the results for more accurate measurements.

Benchmarking Region Variability

Cloud providers offer different regions to deploy to, to allow customers to pick an infrastructure closest to their location. This can be useful for network throughput to external systems, but also for local laws and regulations regarding the privacy and handling of sensitive data.

However, for some customers this difference is not relevant. Since different regions have different price models and often varying performance, it is important for those consumers to take the difference between these regions into account in order to find the most efficient fit for their application.

Multi-User Support

The base benchmarker interface is accessible by uploading an application and having it benchmarked. It would be more user-friendly if this interface could be directly exposed to the users, and allowing them to see benchmark results for themselves. Forms of account restriction are also important to implement, so that separate accounts can be made that can only, for instance, view the results of certain benchmarks.

Python 3 Re-architecture

The current chosen base benchmarker implementation is written in the Python 2 programming language. While still being in support mode for at least five more years[19], it is considered a developmental dead end and the official upgrade path is Python 3[20].

Since Python 3 is a backwards-incompatible upgrade[21], a part of the current codebase will need to be refactored and changed to be Python 3 compatible in order to guarantee future compatibility and support.

Advanced Cost Model

The accuracy of the price prediction is dependent on that of the price model. The current one does not take factors such as long term usage and cheaper prices for leftover capacity into account. Long term pricing could be statically predicted. For leftover capacity a cost model based on historic data or fetching current pricing is required.

Resource-Cost Extrapolation

When benchmarking a bag of tasks, similar tasks are run many times. For instance an application is run many times with different parameters. These parameters influence the runtime greatly. By sampling from these parameters and applying statistics, an estimate of the runtime of the entire bag of tasks can be made. If the application differs in more than just it's input, we could also sample from the different tasks to arrive at an estimate.

Improving Reporting

The current base benchmarker's reporting is meager and shows no information at all when a failure occurs. It is also not able to correlate data from multiple benchmarking runs together, or show runs on different infrastructures for a same workload together. Improving this reporting would vastly improve the user experience, and contribute to the ability to diagnose issues and log possible security issues.

Improving Testing

The current implementation of the base benchmarker has no form of automated testing. In order to make sure everything works in the first place, and that nothing is broken by refactoring and extension implementation, automated tests will need to be added. This will bring familiarity with the existing code base as a bonus. Mocking can be used to simulate the cloud providers, so that no expenses are made running these tests.

Continuous Integration / Deployment

In addition to writing automated tests, a mechanism to run all tests on every commit and automatically deploy the benchmarking system can be put in place to guarantee the quality of the code base continuously, even if developers neglect to run tests themselves. It could also generate test coverage statistics and run various code analysis tools, and report on test failure, insufficient coverage or major issues identified by these tools.

An overview of how these extensions relate to the design goals is given in table 2.2. We believe that implementing these extensions give a significant and sufficient improvement to the success of the design goals.

Table 2.2: Comparison of proposed extensions and their fulfillment of design goals.

Feature	Correctness	Robustness	Security	Extensibility	Maintainability
Deployment Latency	*				
Elasticity	*				
Complex Architectures	*			*	
Cluster Architectures	*			*	
Provisioning Methods				*	*
Provisioning Platforms				*	*
Workload Formats				*	*
Performance Variability	*				
Region Variability	*				
Multi-User Support		*	*		
Python 3		*			*
Cost Model	*				
Cost Extrapolation	*				
Reporting	*	*	*		
Testing	*				*
CI / CD	*				*

2.8.2. Chosen Extensions

After consulting the product owners (listed in table 2.4), the list of extensions was prioritized, and the top items in the priority list were further refined as part of the agile process of backlog refinement. This is visualized in table 2.3.

Table 2.3: Early list of prioritized and refined extensions, sorted by priority.

D	O	Abstract	Refined
*		Testing Improvement	Write tests for code base for full coverage
*		Continuous Integration / Deployment	Set up Strider and SonarQube, integrate with existing services
*		Python 3 Re-architecture	Port existing code base to Python 3.3
*		Development Setup	Set-up easy deployment for existing benchmarker
		Extra Provisioning Methods	Implement provisioning through Apache Libcloud
		Extra Provisioning Platforms	Implement support for Google Cloud Engine
		Benchmarking Region Variability	Refactor code base and data model for regions
		Complex Architecture Deployment	
		Extra Workload Formats	
		Deployment Latency Measurement	
		Reporting Improvement	
	*	Performance Variability Measurement	
	*	Multi-User Support	
	*	Advanced Cost Model	
	*	Elasticity Measurement	
	*	Cluster Architecture Deployment	
	*	Resource-Cost Extrapolation	

D: Development Process related
O: Optional

Emphasis was put on extensions that improve the development process in general, as described in section **??**. The top extensions in the priority list were formalized and refined, to give an appropriate and concrete feature to work on.

Table 2.4: Overview of consulted Nerdalize personnel.

Role	Name	Contact
Co-founder / CTO	Mathijs de Meijer	Face-to-face meeting
Director of the Cloud Orchestra	Eric Feliksik	Face-to-face meeting

Over the course of the project, numerous refinements were done for extensions lower on the list as they rise, in consultation with the product owners and in accordance with an agile process. Priority was given for development process-related goals, as they can massively improve the throughput of the rest of the project. An extra goal called "Development Setup" was added, to allow time for familiarization with the code base and deployment.

A few goals were seen by the product owners as not important to the benchmarker, and thus were marked as optional goals and placed at the bottom of the list, to be implemented when all the important goals have been if there is time left.

Some extension ideas were added during the implementation phase, while others were eliminated entirely. This is discussed in section 6.2.

3 Design of a Cloud Benchmarker

3.1. Overview

During the implementation of improvements on the existing benchmarker implementation, we have done a significant amount of re-architecturing and re-factoring. In section 3.2 we lay out some of the unique challenges in designing this system. In section 3.4 we describe the major changes and improvements to this architecture, which is explained in section 3.3.

In addition, we set up a full deployment and testing environment. In section 3.5 we describe the initial setup, the issues experienced with it, and the final testing and deployment architecture. We conclude in section section 3.6 by showing the final system architecture, and how it corresponds better to the stated design goals.

3.2. Design Challenges

Instead of creating a fresh system and architecture from scratch, there was an existing design and an existing system which we would base our efforts on. This brought a few unique challenges that each require their own skill set to deal with:

- The existing architecture needed to be investigated and critically analyzed for a fit with any new features.
- The existing architecture needed to be re-engineered in case of incompatibility with new features.
- Any re-architecturing or implemented features could not break the existing system, or introduce severe regressions.

Another challenge, connected to the main one, is that the existing system was not well-documented or tested. Identifying if a change broke something in the system, and whether that was acceptable, would be a manual and laborious task. As such, changes had to be introduced with extra precision with respect to the existing architecture, at least until a comprehensive test suite was written.

The system not being extensively used in production yet alleviated some of these concerns, as subtle breakage that would only be notified after very intensive testing would not directly lead to issues that affect the running of the system live. However, time gained in not having to worry too much about this was definitely lost in trying to figure out the existing architecture of the system, and how it was all put together. The lack of documentation and testing made this task even harder, and any changes still had to be thoroughly manually checked in the initial stage of the project.

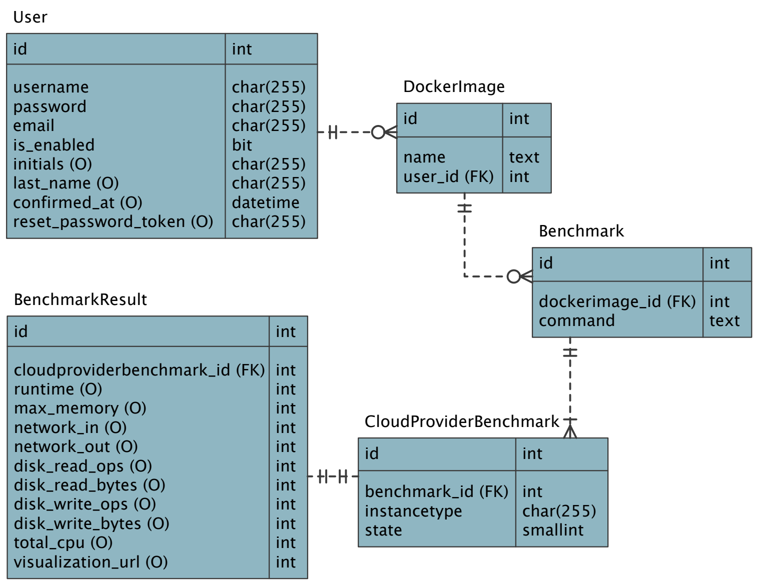


Figure 3.1: Overview of the existing data model.

3.3. Prior System Architecture

In this section, we describe the architecture of the system as received at the start of the project. It was composed of a web application front-end, an orchestrator back-end, and two analyzers: a price engine to calculate benchmark run prices for a given provider, and a resource usage aggregator to compile the resource usage statistics into a pleasant report. The architecture is displayed in figure 3.2.

The web application was used by the end-user to upload images and start benchmarks, while the orchestrator back-end was responsible for running the benchmarks. A PostgreSQL[1] relational database system was used to store general data about users, images, benchmarks and aggregated results. The data model of this database is shown in figure 3.1. An InfluxDB[2] 0.8 time series database system was used to store and retrieve resource usage data from the system under test. Finally, a Grafana[3] installation was used for the visualization of the aggregated data.

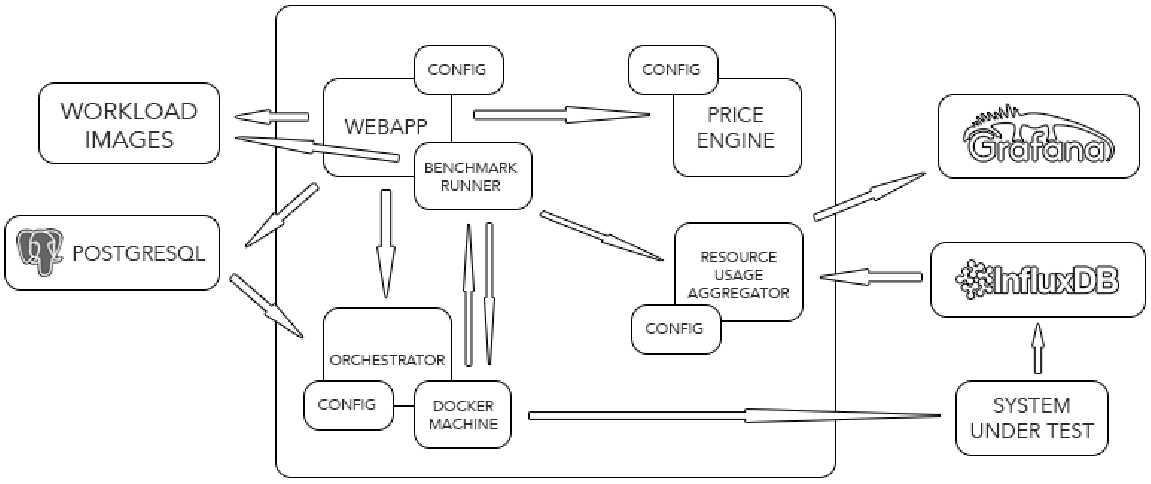


Figure 3.2: High-level overview of the prototype benchmarker architecture.

A large part of the application logic was concentrated in the web application: it contained the full database model objects used by the Object-Relational Mapper (ORM) also used in the other parts. Furthermore, the majority of the logic for running benchmarks was tucked away in the 'tasks' submodule, invoked by the orchestrator. This means that in practice, while being the front-end of the system, a large part of the benchmarking system was directly dependent on the web application subsystem.

The web application front-end was responsible for being the user-facing part of the application, and provide a simple interface for uploading images and starting benchmarks, as well as checking on their status and results. To this end, it used the Flask[4] microframework, and the SQLAlchemy ORM to abstract away the PostgreSQL database backend. It communicated with the other parts by directly importing them: the benchmarker was a single process of which the control flow was orchestrated by the web application.

The orchestration back-end was mostly responsible for determining how to provision the system under test and deploy the workloads. This was hardcoded to use Docker Machine[5]. The provisioning and deployment process took place by using shell scripts that invoked Docker Machine after manipulating certain given parameters. The Celery[6] task queue system was used to manage benchmarker runs and have them be independent from the web application. For managing the benchmark, it invoked the tasks stored in the web application package.

Each part of the benchmarker had its own configuration submodule, storing local settings relevant to the subsystem.

[1]http://www.postgresql.org/
[2]https://influxdb.com/
[3]http://grafana.org/
[4]http://flask.pocoo.org
[5]https://docs.docker.com/machine/
[6]http://www.celeryproject.org/

3.4. Architectural Changes

Both for the objective of improving the development process, as well as implementing the required extensions as described in section 2.8 and chapter 4, significant architectural changes were made to better organize, decouple and modularize the benchmarker. In this section the architectural changes to the system that were done as part of the project are described.

3.4.1. Data Model Re-design

We found the existing data model as shown in figure 3.1 limiting in numerous ways. For instance, a benchmark could only be bound to a single image, which would give issues when modeling more complex architecture deployments which require multiple images. Furthermore, we found the naming of certain models odd and sometimes very unnecessarily specific (e.g. `DockerImage` for workload images). Furthermore, 'ownership' of resources by a user was complicated: it always led back to which image a benchmark used in order to indicate ownership. We implemented the following changes:

- Rename the models to have clearer and more generic names: DockerImage → Image; CloudProviderBenchmark → Run; BenchmarkResult → Result.
- Turn the N:1 relationship of Benchmark to Image into a N:N relationship. This results in a benchmark being able to have multiple images, which is useful and necessary for complex architecture deployment.
- Remove the `command` field from the Benchmark and add a new Workload model: this model stores the actual workload a Benchmark will run and how it has to run it, including links to the images. We therefor placed it in between the Benchmark and the Image models, severing the original link between Benchmark and Image.
- Split up the `instancetype` field in Run to its proper components: `provider`, `instance` and `region`. The field was formerly a composition of these three components, joined together by a special delimiter character. Normalization practices prescribe that these should be separate fields, as they are separate data components.
- Add a `provision_time` field to Result, since we do want to measure provisioning time as well.
- Add useful fields to Image, such as the image 'tag' (used for Docker images), and the actual path where the image is stored. This was previously calculated by the benchmarker application from the image ID at various different points.

As result, figure 3.3, shows the new data model. While the changes, except for the Workload introduction, are relatively minor in nature, they made the data model a lot more pleasant to work with and reason about.

3.4.2. Deployment Model

The prototype implementation had a very simple deployment model, consisting of two steps: deploying the workload, and running the workload. Furthermore, this was hard coded to use Docker Machine for both steps, leading to only having support for the cloud providers supported by Docker Machine, and Docker-packaged workloads. In this change, the deployment model was revisited and redesigned to be modularized, pluggable and easily extensible.

The first step was to separate the deployment steps further to enable abstraction of different parts. The deployment model was changed to have four steps:

- Provisioning: obtaining a machine from a cloud provider to run workloads on.
- Configuration: preparing the obtained machine for the workload type, for example by installing required prerequisite software.
- Deployment: getting the workload resources and monitoring software on the machine.
- Running: running the workload and measuring resource utilization with the monitoring software.

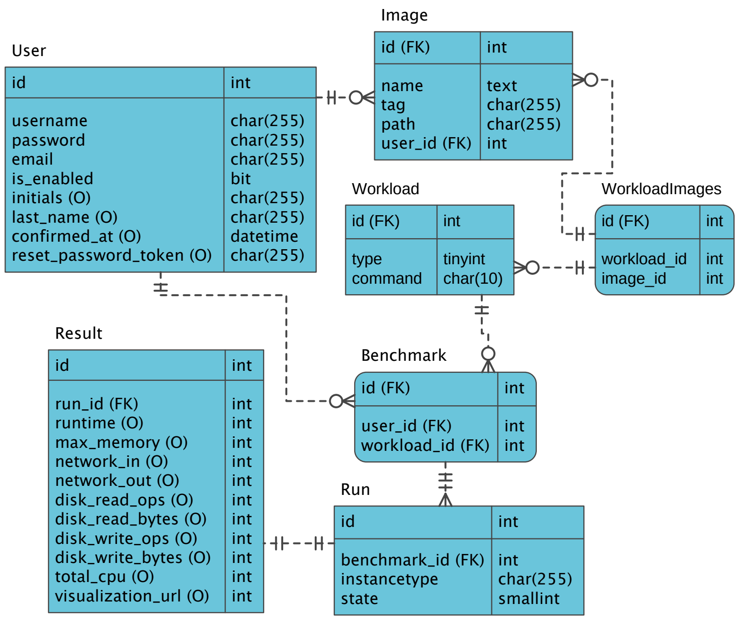


Figure 3.3: Overview of the new data model.

These four steps were then distributed over three pluggable deployment classes:

- Provisioner: perform the provisioning and deprovisioning steps.
- Configurator: perform the configuration and cleanup steps.
- Deployer: perform the deployment, run and cleanup steps.

Each class is also responsible for cleaning up the changes it has made. For a provisioner, this would mean making sure the provisioned machine is also destroyed after the run is done. For a configurator, this would mean removing any additional software it has installed. Finally, for a deployer, this would mean cleaning up after the workload and removing the workload and monitoring software.

Each class implementation is fully independent and can be swapped out for another. This architectural change allowed us to quickly and efficiently add support for new cloud providers and workload formats without meddling in unrelated modules. An overview of the new architecture is depicted in figure 3.4.

A model where the configurator and deployer were merged into one class was also considered, but rejected because of dissimilar workload types requiring identical configuration, such as Docker and Docker Compose workload types.

The second step was to implement automatic selection of the appropriate method for a given benchmark. Instead of a centralized place were this was decided, we opted for decentralized selection, where every provisioning, configuration and deployment method can state if it supports the given benchmark or not. The orchestrator then iterates over all methods and collects a supported combination. This makes selection flexible and keeps support logic isolated in the method implementations.

Finally, the existing Docker Machine method had to be ported over to this new model. This is discussed in section 4.5.

3.4.3. Orchestrator Separation

In the pre-existing implementation, the web application front-end and orchestrator back-end were heavily coupled, with some orchestration functionality even being implemented within the web application. In order to be able to guarantee stability and cleanly separate out different functionality, we would have to split up the orchestrator and separate it entirely from the web application. This would allow

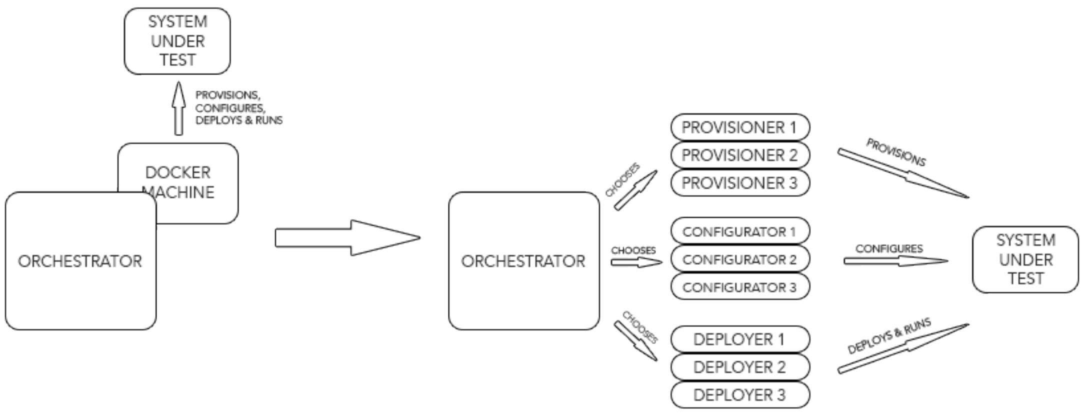


Figure 3.4: Overview of the changes in the orchestrator deployment model.

it to run independently from the web application as well, focusing on what it's good at: orchestrating benchmarks.

As described in section 3.3, the prototype implementation contained four modules: the web application, the orchestrator and two analyzers. The first step in separating the web application and the orchestrator was deciding which analyzer(s) were needed by either, if any.

Inter-application Communication

In addition, the web application would still have to communicate with the orchestrator, for example to inform it of a new benchmark being created. Previously this was done by directly importing the orchestrator module, and calling its methods. However, with the orchestrator running in a separate process or even container, a new way for the two to communicate would have to be devised. Some methods we considered were:

- **Sockets:** Setting up a bi-directional pipe (socket) between the two processes and sending serialized messages over this pipe. This has the pro of being bidirectional, where any party can initiate a message, but a protocol would have to be invented and made stable. In addition, it was uncertain how how well listening on a pipe would work with the HTTP server within the web application.

- **Database triggers:** Sharing a database instance between the web application and the orchestrator, and setting database triggers when new data was inserted. This way the orchestrator would be able to directly respond to new data in the database, without directly needing a notification from the web application. However, it could be the case that data was inserted into the database that was not yet ready to be ran as benchmark. Furthermore, it would make analyzing the communication between the orchestrator and web application very difficult and opaque, and exactly how the database would call back to the orchestrator was hard to figure out, and possibly not portable across database platforms.

- **HTTP API:** The orchestrator exposing a RESTful (as described by Fielding and Taylor [22]) HTTP API. This would allow us to use a standard protocol to communicate, although the message specifics and API would still have to be designed. Furthermore, because of this standardization it would be easy to implement using any of the various HTTP libraries available, and would be portable across any implementation of the orchestrator, being able to survive severe architectural changes or rewrites.

After having considered our findings, summarized in table 3.1, we chose to implement communication as a RESTful HTTP API. From the requirements we determined that the web application only ever needed to inform the orchestrator of events, so bi-directional message initiation was not needed. A RESTful HTTP API made it easy to follow the communications between the web application and orchestrator, and was easy to implement using any of the various Python micro-frameworks available.

Table 3.1: Comparison of the reviewed orchestrator communication techniques.

Method	Portable	Standardized	Ease of Implementation	Bidirectional
Socket	Yes	No	Medium	Yes
Triggers	No	No	Hard	Yes
HTTP API	Yes	Partially	Easy	No

Furthermore, we determined a HTTP orchestrator API would also be useful in the implementation of custom provisioning, as described in section 3.4.4.

From analyzing the interactions between the web application module and the orchestrator module, we determined the API had to make it possible to fulfill three tasks:

- **Start a run:** For a specific given run in a benchmark: provision, configure and deploy the machine, and run the workload and resource analysis software.
- **Retry a run:** For a specific given run in a benchmark that failed: retry this run according to the steps above.
- **Cancel a run:** For a specific given run in a benchmark: cancel the run and clean up all resources.

We deemed implementing starting, cancellation or retries for whole benchmarks as unimportant, as it could be performed by performing the same action for all runs in that benchmark.

Allocation of Data Sources

Another important consideration was data sharing between the separated applications. We had to decide on which of the data sources available to the benchmarker would be made available to which application, and due the possibly concurrent usage of these sources data synchronization issues had to be taken into account.

- **Relational database:** Containing the basic data structures for common models, such as users, images, benchmarks and runs, this would be needed by both the web application and the orchestrator. The web application could use the decided-upon communication protocol to transfer data over to the orchestrator, but this would require deciding upon a (de)serialization protocol. This was furthermore not needed, as the database was well-equipped to handle synchronization of data and race conditions, following ACID[23][24] principles. We decided to make the database available to both the web application and the orchestrator.
- **Time series database:** Containing the measurement data from a benchmark run, this would be needed for the aggregation and processing of resource usage. In our new split-up design, this was fully handled by the orchestrator, and the web application had no use for it. It was thus only made available for the orchestrator.
- **Workload images:** Containing the actual data and applications that belonged to workloads, this was needed by both the web application so users could upload images, and the orchestrator so it could upload and run those images to the system under test. We saw no significant advantage in using the communication protocol to transmit the images. Furthermore, as only the web application produces new images, data synchronization issues are not relevant.

The decided-upon division of data sources is summed up in table 3.2. The architectural changes from this features are visualized in figure 3.5.

3.4.4. Custom Provisioning

A big new feature in the benchmarker is being able to run workloads on any Linux host on which commands can be manually run. Called 'custom provisioning', this allows running workloads on cloud providers for which support has not been added yet, or own machines optionally behind network firewalls. This would greatly improve the extensibility of the benchmarker, not requiring any programming changes to run workloads on new kinds of systems.

Table 3.2: Division of the data sources between the separated applications.

Data Source	Web Application	Orchestrator
Relational Database	Yes	Yes
Time Series Database	No	Yes
Workload Images	Yes	Yes

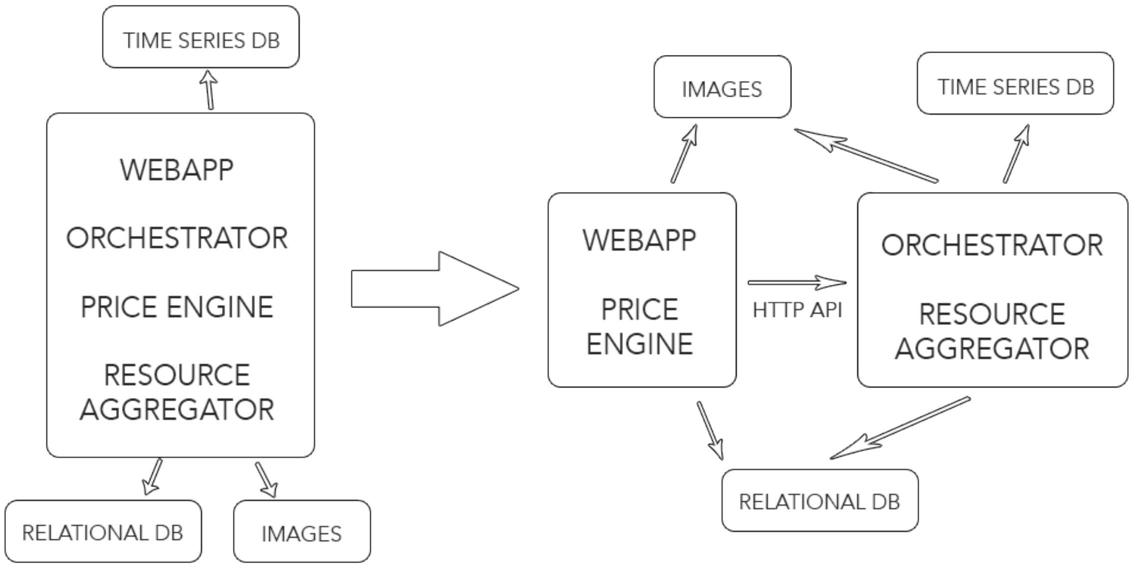


Figure 3.5: The separation of the orchestration component.

Communication Program

A big factor considered in the design was ease of use: ideally running the entire workload would only require a single command, with the orchestrator taking care of everything on the machine after that. To this end we opted to have the single command consist of downloading and running a single program that would communicate with the orchestrator to run the workload. The alternative of having the user run a bunch of commands manually was found too cumbersome. Having decided this, we had to consider an implementation language for this program, taking into account portability:

- **Python:** Considering the rest of the prototype implementation was written in Python too, this would make the code base consistent. Python being a high-level language would make the script trivial to write, too. However, we consider requiring a Python runtime to be installed on any system we might want to test unreasonable.

- **C:** The de facto implementation language for Linux and other UNIX-like environments, C programs can be expected to run everywhere once compiled. However, being a low-level language would make implementation difficult. Furthermore, we would have to compile a different binary for every target operating system workloads would be ran on.

- **POSIX shell:** The de facto scripting language for Linux and UNIX-like environments, guarded by the POSIX[25] standard. If written carefully, programs written in this language are portable across all POSIX compliant systems, and no further runtime except that mandated by POSIX is required.

We summarized our findings in table 3.3. From our design considerations, we value portability and lack of extra runtime requirement highly, as ideally as little as possible is required from the target system under test. To this end, we decided to use POSIX shell to implement the communication program.

Table 3.3: Considered languages for implementing the orchestrator communication program.

Language	Difficulty	Portable	Needs Extra Runtime
Python	High-level	Yes	Yes
C	Low-level	Only source code	No
POSIX shell	High-level	Yes (if careful)	No (POSIX)

Communication Protocol

Another consideration was the protocol used to communicate with the orchestrator. Since the system under test may be behind network firewalls or Network Address Translation[26] (NAT), we can not expect the orchestrator to be able to initiate connections to the system under test to send commands. To solve this, the client should initiate a TCP connection, which can then be used by the server to 'push' new commands and data. A number of standardized and less-standardized protocols such as Reverse HTTP[27] , WebSockets[28] and BOSH[29] were considered. The issue that we found at the time was that these all required additional software or libraries on top of the relatively slim requirements we had made for systems under test for custom provisioning. Furthermore, we did not need bidirectional message initiation, as the orchestrator would only be sending commands to the system under test. As a result, we devised a simpler protocol on top of HTTP:

1. **Initiation phase:** The system under test (client) opens a HTTP connection to the orchestrator (server) and sends a POST request indicating its availability. The connection is kept open.
2. **Messaging phase:** The server is now free to start sending messages. These messages are sent as HTTP responses, with the HTTP content body containing the message contents. Once the client receives a message, it closes the HTTP connection, processes the message, and opens a new HTTP connection again, sending its response (if any) as a HTTP POST request. The connection is kept open and the server is now free to send a new message.
3. **Termination phase:** Once the server is done with the client, it simply sends a HTTP response with a message telling the client to terminate. The client will close the connection and not open a new connection again.

This protocol can be seen as a variant of Reverse HTTP, treating HTTP responses as requests, and HTTP requests as responses, except for the initiation and termination messages. However, it requires nothing more than a standard HTTP client on the system under test and a HTTP server on the orchestrator. In line with keeping the requirements for the system under test minimal, we deem this a good solution for our architecture. From section 3.4.3, the orchestrator had already gained a HTTP API that could be used in the implementation of this. Furthermore, the system under test already needed some way to retrieve the communication program from the orchestrator in the first place: HTTP was deemed the most standardized protocol to do this, so the system under test would already require a HTTP client.

3.4.5. Other changes

A lot of other, more modest changes to the design were made as well. In this section they will be all briefly discussed.

Improving Logging

The prototype implementation received no kind of feedback from the orchestrator about the status of a benchmark run, aside from a basic keyword like 'running', 'finished' or 'error'. We opted to improve logging of runs, not only to introduce clarity for the user, but also to make it easier to spot faults and errors for the developer. In the new split-up architecture, we had to therefore decide upon a method to transfer orchestrations logs to the web application. Because the orchestrator could not call back into the web application, and because a database field was seen as excessive and unfit for a potentially large log, we opted to share a data volume between the web application and the orchestrator. The orchestrator could write its logs to simple files, and the web application could simply read those files and show them in the web interface.

Configuration Centralization

In the prototype implementation, each module had its own configuration submodule: see figure 3.2. This proved cumbersome for the end-developer and system administrator, as finding out where exactly a certain configuration value needed to be set could be time-consuming. Furthermore, certain configuration items were duplicated across the several submodules, so you had to be careful to modify the right values at all points in the configuration.

A new design was put into action where all configuration was centralized in a single module, which would then be accessed by all other modules. This merged configuration included among others private keys, cloud provider credentials, cloud provider pricing information and file paths. The result of this merge is shown in figure 3.6.

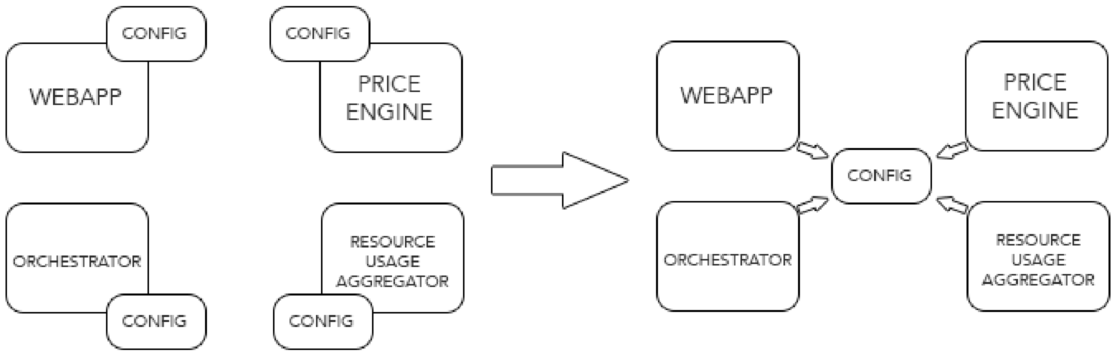


Figure 3.6: Overview of configuration architecture changes.

3.5. Testing Environment

Adopting the Extreme Programming methodology, a large focus was put on testing and quality assurance. However, the prototype implementation had no form of automatic testing whatsoever. This means we had to set up an entire testing environment. In compliant with eXtreme Programming practices, we opted to go for frequent integration, as described by Booch *et al.* [30]. This avoids "integration hell", where integrating any changes from the developer's testing branch back into the main branch or a testing environment proves very cumbersome, because the branch has been isolated for so long.

It is efficient for an agile team as opposed to its traditional counterpart, a dedicated quality assurance (QA) phase. This is because it happens continuously, and thus consists of frequent but tiny changes and fixes, as opposed to all needed changes being piled up at the end of a development phase. It allows catching any integration mistakes early and fixing them as soon as possible. In this project, we opted to go even further, using *continuous* integration (CI). This means that every developer merges changes from the main branch into their feature repository on every new commit they make. This way, problems due merge conflicts could be kept to a minimum.

An important good practice in frequent and continuous integration is continuous automated testing[30]. There was no such setup for the prototype implementation, so we devised an architecture for continuous testing. It is shown in figure 3.7. Whenever a developer pushes a change to the code server, it is automatically tested by the CI system using the code base's unit and integration tests. When the test run is finished, the developer is notified of the status using the team messaging system. This way, a developer is very quickly notified if a change they made broke or changed features unintentionally, and even developer branches can be kept stable.

Additionally, we opted to use *continuous deployment* (CD) as well: a revision of the main branch that passed all tests would be automatically deployed to a testing environment for further manual testing whenever appropriate. This way the product stakeholder would also have an immediate view into the progress. When deciding on a new release to deploy to the production environment, one can look at the releases that passed automated testing and can perform additional manual testing. It also allows for faster feedback cycles from the product stakeholder, as they can directly see the product working live in the testing environment.

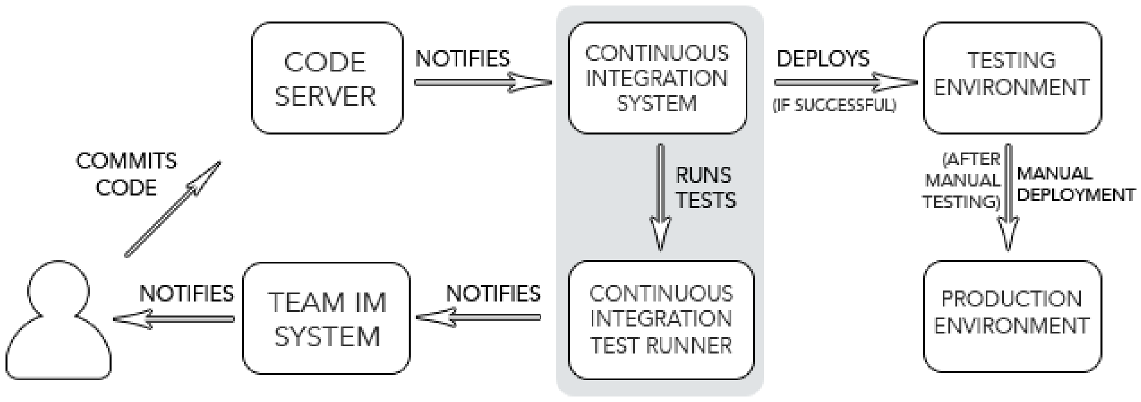


Figure 3.7: Overview of testing and deployment architecture.

For tests proper, we decided on three types of automated testing, each of which approached using any of the two methods of *white-box testing* (testing with knowledge of inner parts of the system [31]) and *black-box testing* (testing without knowledge of inner parts of the system, only checking input and visible output [31]):

- **Unit testing:** Testing the individual functionality of code units that work alone[31]. This would be typically approached using white-box testing, since any effects might not be immediately visible at the unit level. An example of this is testing if the price engine outputs the correct price, given resource usage data for a certain provider.
- **Integration testing:** Testing the interactions between units of code[31]. This would be approached using a mix of white-box and black-box testing, but mostly black-box testing. An example of this is that given resource usage data, the orchestrator successfully calls Grafana for visualization of the data.
- **System testing:** Testing interactions of the user with the entire system[31]. This would be approached using black-box testing. An example of this is running a complete benchmark through the benchmarker and ensuring that a machine is provisioned, workloads are ran and results are generated.

This hierarchy of tests is shown in figure 3.8. Other test types like user acceptance tests and usability tests were left out because they were considered very hard to automate.

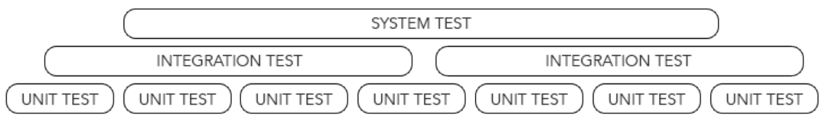


Figure 3.8: Testing type hierarchy for automated tests.

3.6. Final System Architecture

In figure 3.9, we depict the design of the re-architectured benchmarker. We believe this new design fits our original design goals as described in section 2.7 a lot better:

- It is more **correct** because of the implemented test suite that makes sure the code does what it is supposed to, and the automated testing to ensure it keeps doing this. Additionally, the new deployment model makes it easier to support more workload types and cloud providers, easily increasing the number of applications supported by the benchmarker.

- It is more **robust** because the orchestrator and web application are now fully independent, separated applications: a mistake in one can not crash the other, and the split makes external dependencies better organized.
- It is more **extensible** because the deployment model has been changed to a pluggable architecture, where support for new cloud providers and workload formats can be trivially added.
- It is more **secure** because the new deployment model guarantees clean-up of potentially proprietary workload images, making sure no third party can access them even if the cloud provider's machine cleanup procedures are lacking. Furthermore, assigning a user to benchmarks in the data model allows for better and easier access checking.
- It is more **maintainable** because both the internal and external architecture have been abstracted and split up better, allowing for better understanding what a piece of code does and easier swapping out of modules. Furthermore, the addition of a full test suite and automated testing make finding regressions introduced by new changes a lot easier.

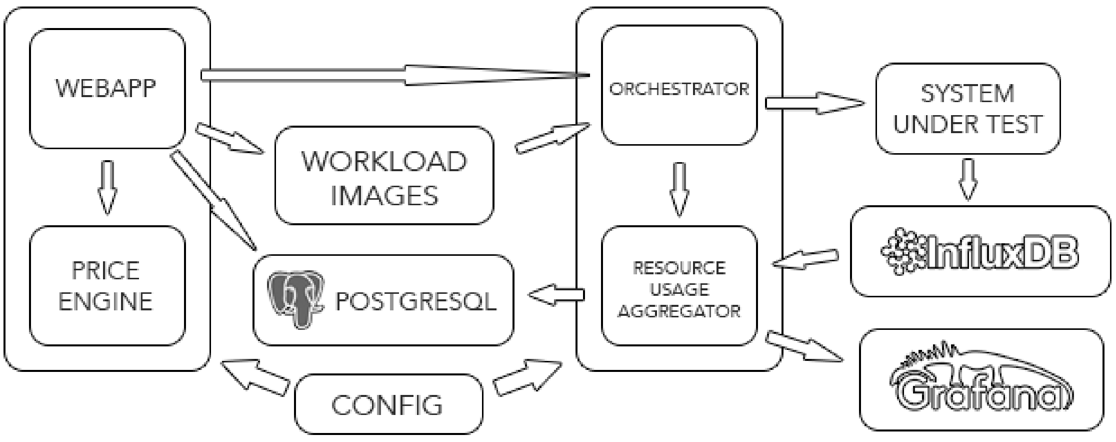


Figure 3.9: High-level overview of re-architectured benchmarker.

4

Implementation of the Cloud Benchmarker

4.1. Overview

Being an agile project, the implementation of the benchmarker involved the bulk of the time spent on the project, with the design being improved iteratively. In this chapter, implementation specifics and difficulties are discussed for the more difficult parts of the project. First, we discuss the employed development methodology and software in section 4.2. Then, in sections section 4.3, section 4.4, section 4.5 and section 4.6, we discuss the implementation of some of the largest changes made from the prototype. In section section 4.7 we discuss the setting up of the testing environment and the associated difficulties. Finally, in section 4.8 we discuss the process of making the benchmarker both Python 2 and Python 3 compatible.

4.2. Development Methodology

Central to any development process is the methodologies used to turn the process into a streamlined and effective one. Numerous methodologies, tools, and paradigms have been considered, and in this section the findings will be presented.

4.2.1. Development Process

For the development period, eXtreme Programming (XP) was chosen as the agile development methodology. Agile principles have proven highly effective for small teams[32]. It allows teams to quickly adapt to requirement changes and to rapidly get features up and running, which is especially useful in the relatively short eight-week development period. The short sprints allows splitting up the implementation of extensions effectively, and adopting a test-driven approach.

Sprints of two weeks were chosen, with flexible milestones for extensions set at the start of every week. eXtreme Programming engineering paradigms such as test-first development, extensive code review and simple code were seen as very beneficial to the overall engineering process.

Waterfall was considered as development methodology, but eventually rejected due its inflexibility over the short development process, the requirement to specify full architectural changes up-front and the lack of engineering guidelines. Scrum was a very debated addition, but was eventually also rejected in favor of eXtreme Programming due the latter featuring shorter sprints and more flexibility within these sprints, as well as a focus on good engineering principles, leading to a feeling of redundancy in adding Scrum concepts.

4.2.2. Communication

The team communicated directly, being present at the Nerdalize office at YES!Delft every weekday. Working hours from around 09:00 to around 17:00 were decided on, with some flexibility allowed as long the weekly quotum of forty hours was met. A Trello[1] board was set up to keep track of tasks and progress. Furthermore, Slack[2] was already used by Nerdalize as team communication tool.

Partial remote working was briefly proposed by Nerdalize, but quickly rejected because of the perceived benefits of face-to-face communication and regular working hours.

We integrated with the rest of the Nerdalize development team by using Slack and face-to-face communication, but we were the only two developers working on the benchmarker project. However, other developers were more than willing to help think issues over or do dry user testing of the benchmarker, for which we are very grateful. This aside, semi-regular stakeholder meetings with Eric Feliksik, Director of the Cloud Orchestra, and Mathijs de Meijer, CTO, were had to discuss progress and refine and reorder backlog items.

4.2.3. Source Control

Git[3] was chosen as the source control system for this project. It provides distributed source control, where everyone can work independently on their own feature and merge it later. Furthermore, the existing benchmarker base source code was already hosted on a Git repository, making the transition easy.

Subversion was also considered for this role, but rejected due its very centralized nature. Extensions to the benchmarker base are in principle independent and can be implemented in parallel, but having to push every change to a central repository can make this very hard. Mercurial was also considered, but not chosen due to the developers' unfamiliarity with it, and it not being perceived as having significant advantages to Git.

4.2.4. Source Quality

It is important to keep an eye on the overall quality of the source code during the development process. Source code should not only work, but also be well-organized, cleanly designed and maintainable (see section 2.7). Two tools were chosen for continuous inspection of source code quality.

Pylint[4] was chosen as the tool that can be run locally and do quick checks on code quality while

[1]https://trello.com
[2]https://slack.com
[3]https://git-scm.com
[4]http://www.pylint.org

developing, using IDE integration mechanics. Alternatives such as Pyflakes and flake8 were considered, but they were found to be lacking in metrics. A fast and lean tool like this allows for rapid checking and improvement during the development of a feature, and catching errors before they are committed. Since Python, the programming language used in the base implementation, is dynamically typed and relies on runtime checks instead of compile-time ones, having a static analysis tool that runs periodically in the IDE is very useful for catching issues early.

In addition to a quick local checker, a more elaborate architecture inspection tool was also required. This tool would be run on every commit and output a full report on various metrics of source code quality. It was decided to use SonarQube[5] due to its elaborate featureset and ability to be run on an own server. Numerous alternatives like Kiuwan, Parasoft and Semmie were looked at, but they were all commercial and proprietary tools, of which a potential purchase and evaluation was not within the scope of this project. Another potential candidate was Yasca, but its analysis skills for Python source code seemed limited, and integration with other systems was not implemented.

Table 4.1: Initial and final overview of the used tools and software.

Category	Name	Location
Source code management	Git	https://git-scm.com
Source code hosting	Bitbucket	https://bitbucket.org
Task management	Trello	https://trello.com
Team communication	Slack	https://slack.com
Continuous integration	Strider	http://stridercd.com
Continuous deployment	Strider	http://stridercd.com
Source code quality inspection	Pylint	http://www.pylint.org
	SonarQube	http://www.sonarqube.org
Editor	Eclipse	https://eclipse.org
	Vim	http://www.vim.org
Source code management	Git	https://git-scm.com
Source code hosting	GitLab	https://gitlab.com
Task management	Trello	https://trello.com
	Backlog board	Office
Team communication	Slack	https://slack.com
Continuous integration	GitLab CI	https://gitlab.com
Continuous deployment	GitLab CI	https://gitlab.com
Source code quality inspection	Pylint	http://www.pylint.org
	SonarQube	http://www.sonarqube.org
Editor	PyCharm	https://www.jetbrains.com/pycharm
	Vim	http://www.vim.org

4.2.5. Testing

A test-driven development (TDD) approach was chosen for this project. Being one of the key points of eXtreme Programming, it allows the developers to quickly iterate over benchmarker features. Its behavior-driven nature makes the implementation more likely to follow the design, as the behavior is specified upfront and the implementation is continuously tested against this.

We set up a continuous integration (CI) server and configured it to automatically test every commit against the test suite, and notify the developers if a test failed with these new changes. We initially chose Strider[6] as the implementation of choice due its integration with Git services, customization, ability to be self-hosted, and its pleasant user interface. However, we quickly ran into issues, as described in section 4.7, and eventually opted to use GitLab CI instead. This was paired with a switch to GitLab for source code hosting, being a prerequisite to use GitLab CI.

[5]http://sonarqube.org
[6]https://github.com/Strider-CD/strider

We opted to utilize the feature branching model commonly combined with Git, where functionality is separated into features. Features are worked on independently in isolated source code branches, which are merged back into the main ('master') branch when they have been completed and tested. The main branch should always be stable and contain a working a product: experimental features should be merged into a special development ('edge') branch when they have been completed.

4.3. Separating the Orchestrator

The orchestrator separation as described in section 3.4.3 was one of the heaviest changes implementation-wise. Any and all links between the web application and the orchestrator had to be separated and replaced with API calls. Furthermore, any shared resources had to be carefully factored out.

The prototype implementation was deployed using application containers via Docker Compose[7]. The once-single container containing the entire application now had to be split up in two containers, one for the web application and one for the orchestrator. The additional containers containing the relational database and other data sources had to be re-linked to one or both of the new containers, according to the layout in section 3.4.3. Furthermore, the code shared between the two new applications had to be included in both containers. An overview of the code module division between the applications is given in table 4.2.

Table 4.2: Module division between separated web application and orchestrator.

Module	Web Application	Orchestrator
`config`	*	*
`models`	*	*
`price_engine`	*	
`webapp`	*	
`resource_usage_aggregator`		*
`orchestrator`		*

We analyzed the interactions between the web application and the orchestrator, and designed an orchestrator HTTP API, outlined in table 4.3. The web application would use this API to 'kickstart' an operation in the orchestrator, which would give the web application feedback by updating the relevant database entries with new information. The overall interaction when running a benchmark since the separation of the orchestrator is displayed in figure C.1.

Table 4.3: Orchestrator RESTful HTTP API.

Endpoint	HTTP Method	Input	Description
`/run`	POST	`id`: Run ID	Run the benchmark run identified by `id`.
`/retry`	POST	`id`: Run ID	Retry the benchmark run identified by `id`.
`/cancel`	POST	`id`: Run ID	Cancel the benchmark run identified by `id`.

An issue we ran into is that in the testing environment, the application still had to be ran as one container, or coverage information would not be recorded properly. As a result, we could not rely on Docker's hostname resolution to connect to the 'orchestrator' container, and had to add configuration options to override the orchestrator's address and port, and set these in the testing environment.

4.4. Improving Data Storage

The prototype implementation ran the collectd[8] system metric collection software on the system under test, gathering data every ten seconds and sending it to the orchestrator's InfluxDB[9] time series

[7]https://docs.docker.com/compose/
[8]http://collectd.org
[9]https://influxdb.com/

database. Data collection in this setup proved to be very failure-sensitive, occasionally dropping entire collectd packets, or even worse, randomly locking up entirely, resetting any connection attempted to it.

We approached this issue in two ways: first, we upgraded the aged InfluxDB 0.8 to InfluxDB 0.9, greatly improving reliability and durability of the database engine. Sadly, InfluxDB 0.9 was in many ways a rewrite: a lot of functionality was stripped out, moved or renamed. As such, a significant amount of the benchmarker code that query InfluxDB had to be rewritten. Example query rewrites are shown in figure 4.1.

```
0.8: SELECT DERIVATIVE(value) FROM "benchmark-1/disk-sda/disk_octets"
     WHERE dsname='read' GROUP BY time(10s) ORDER ASC;
0.9: SELECT DERIVATIVE(value) FROM 'disk_read' WHERE host = "benchmark-1"
     AND instance = "sda" AND type = "disk_octets";

0.8: SELECT MAX(value) FROM "benchmark-1/interface-eth0/if_octets"
     WHERE dsname='tx';
0.9: SELECT LAST(value) - FIRST(value) FROM 'interface_tx'
     WHERE host = "benchmark-1" AND instance = "eth0" AND type ="if_octets";
```

Figure 4.1: Differences between example InfluxDB 0.8 and 0.9 queries.

Additionally, we decreased the collectd statistic collection interval to one second, down from ten. This would lead to increased network traffic, but a lot more accurate measurements. Now, the various resource utilizations could be followed by the second, which proved very useful for applications with quickly alternating processor usage patterns.

4.5. Implementing Additional Provisioning Methods

As part of extending the prototype to allow provisioning to more cloud services, two additional provisioning backends were created. In this section we describe the implementation details and pitfalls we ran into.

4.5.1. Custom Provisioning

The first additional provisioning method we implemented was one that allows the orchestrator to deploy to any Linux host on which a command can be manually ran. We called this the 'custom' provisioning backend, and implemented it first because it would allow us to test on any cloud provider, albeit with some extra manual work of provisioning the machine and entering the command. As described in 3.4.4, a long-polling HTTP API was chosen as the communication method of choice. This meant that a HTTP client had to be installed on the server. Furthermore, TCP keep-alive support would be useful to detect if the connection has dropped, especially in the case of long polling with significant amounts during which no data is transferred. After doing a survey of client features, shown in table 4.4, we decided to require the curl HTTP client to be installed on the machine. Luckily, this client comes pre-installed on all the operating system images we encountered.

Table 4.4: Overview of HTTP client capabilities.

Client	TCP Keep-Alive	HTTP POST	Long Polling	Typically Pre-installed
curl[10]	Yes	Yes	Yes	Yes
GNU wget[11]	No	Yes	Not Ideal	Sometimes
BusyBox wget[12]	No	No	No	No
HTTPie[13]	No	Yes	Yes	No

The orchestrator HTTP API was extended to add the required endpoints for custom provisioning. These endpoints are relatively simple and shown in table 4.5. The `/util` endpoint is where the client

```
<?xml version="1.0" encoding="utf-8"?>
<message>
        <type>command</type>
        <command>docker pull maartenve/prime</command>
        <quiet>false</quiet>
</message>

<?xml version="1.0" encoding="utf-8"?><message><type>command</type><command>docker
    pull maartenve/prime</command><quiet>false</quiet></message>
```

```
{
        'type': 'command',
        'command': 'docker pull maartenve/prime',
        'quiet': false,
}

{'type':'command','command':'docker pull maartenve/prime','quiet':false}
```

Figure 4.2: Message format encoded in XML and JSON for comparison, pretty-printed and compacted.

can download the custom provisioning client software from, and in a later provisioning stage, also the workload images.

Table 4.5: Additions to the orchestrator API for custom provisioning.

Endpoint	Method	Description & Arguments
`/communicate`	POST	Communicate with orchestrator to perform benchmark. `token`: A security token identifying the machine. `result`: The response to the orchestrator message, or the initiation message.
`/util/<name>`	GET	Retrieve supplemental benchmark software and data. `name`: The identifier of the software or data to fetch.

While HTTP provides a suitable transport layer for messages, a message format had to be devised. We decided to use a custom protocol formatted in JSON[33], due its ubiquity among web developers and wide support, as well as its relative brevity compared to, for instance, XML documents. We compare these two formats by example in figure 4.2.

The contents of the JSON-formatted messages were a simple JSON-object-encapsulated key-value protocol, the type of which identified by the `type` entry. An overview of all message types is given in table 4.6.

As described in 3.4.4, the protocol consists of messages by the server and responses by the client. Initially, we ran into issues while deploying to Microsoft Azure machines, because they come pre-shipped with a firewall that disconnects any TCP connection after four minutes of inactivity. Some of our workloads took longer than four minutes to run, and thus would get cut off. We solved this by decreasing the TCP keep-alive timeout between the client and the orchestrator to two minutes, sending empty packets to prevent the firewall from disconnecting the connection.

4.5.2. Libcloud

In order to be able to provision to Google Compute Engine and several other providers unsupported by Docker Machine, we implemented a provisioning backend using Apache Libcloud[14]. Touting "the avoidance of vendor lock-ins" and "using the same API to talk to multiple providers", it claims to support over thirty cloud providers[34].

During the implementation of the Libcloud backend, it became obvious that its provisioning model differed from the benchmarker's: instead of providing a procedural API to talk to the provisioned machine in real time, all operations to be executed on the machine have to be supplied beforehand.

[14]https://libcloud.apache.org/

Table 4.6: Orchestrator ↔ System under Test communication protocol.

Type	Description & Arguments
Client Messages	
`started`	Indication to the orchestrator that the client is available. <no arguments>
`ok`	Server command was executed successfully with no further info. <no arguments>
`error`	Server command failed to execute. `message`: The error message
`result`	The result of the command executed by a `command` message. `code`: The exit code of the executed command `output`: The output of the command, unless `quiet` was true. `error`: The error output of the command.
`file`	The contents of the requested file by a `get` message. `data`: File contents
Server Messages	
`command`	Execute a command in the shell. `command`: The command to execute `quiet`: Whether or not to return the command output
`download`	Download a file over HTTP. `url`: The URL to download from. `path`: The file path to save in.
`get`	Upload the contents of a file. `path`: File path to upload
`destroy`	Benchmark run is over, end and clean up this script. <no arguments>

Libcloud being incompatible with the usual model used for orchestration, we had to think of a work-around to make Libcloud fit in, as this was not a model we found suitable to change the orchestrator to. Eventually, we came up with a solution: re-use part of the custom provisioning method described in section 4.5.1, so we can give a single command beforehand to Libcloud: the command normally used to download and launch the custom provisioning script on a machine. With this solution, the custom provisioning method can handle the usual provisioning operations and fit in properly with the benchmarker's orchestration model.

Another issue we ran into was that while claiming to support the same API across multiple providers, in reality the API documentation and implementation did not match, in various ways across various providers. For instance, a "deployment" operation, giving the specific set of operations that have be performed after the machine has been provisioned, is documented to be specified as a list of instantiated objects of classes from the `libcloud.compute.deployment` module. However, for the Google Compute Engine provider, the `Driver.deploy_machine()` method took no such argument: instead, it took the filename of a POSIX shell script to execute on the machine.

To solve this issue, we split up the provisioning implementations for each provider supported by Libcloud that we could test. Formerly one generic method, it was now split into small dedicated methods, each taking care of the individual derivations from the standard API by Libcloud's driver implementations.

4.6. Implementing Additional Deployment Methods

In addition to support for more providers, we also added support for more workload types. The prototype implementation only supported Docker. While an easy and standardized way to package applications for universal deployment, it does not scale well for more complex deployment architectures that involve multiple services interacting with each other. To this end, we implemented Docker

Compose[15] workload support.

Docker Compose is an application that uses Docker to orchestrate container dependencies and inter-links. Containers are specified using declarative YAML[35] specification files; refer to figure 4.3 for an example.

```
postgresql:
    image: sameersbn/postgresql:9.4-3
    environment:
    - DB_USER=gitlab_ci
    - DB_PASS=password
    - DB_NAME=gitlab_ci_production
    volumes:
    - /srv/docker/gitlab-ci/postgresql:/var/lib/postgresql

redis:
    image: sameersbn/redis:latest
    volumes:
    - /srv/docker/gitlab-ci/redis:/var/lib/redis

ci:
    image: sameersbn/gitlab-ci:7.14.3
    links:
    - redis:redisio
    - postgresql:postgresql
     ports:
    - "10081:80"
    volumes:
    - /srv/docker/gitlab-ci/gitlab-ci:/home/gitlab_ci/data
```

Figure 4.3: Example Docker Compose YAML specification file, specifying three containers..

In other to differentiate between different workload types, a `type` field was added to the `Workload` database model. This is a free-text string that could be used by deployment methods to determine if they support the given workload format. Furthermore, a method to store the YAML specification had to be devised. Luckily, there was already a field for storing a workload specification of some kind: for Docker workloads, the `command` field was used to specify the command required to start the container. We re-purposed this field to store workload specifications in.

With the required database changes done, a `Deployer` subclass was implemented. One issue was that Docker Compose specification files, unlike regular Docker workloads, may refer to images that do not exist on the orchestrator yet, like external public images used for supplementary services like PostgreSQL. Furthermore, the specification files can contain certain 'dangerous' stanzas that might provide a user with more privileges than strictly desirable.

To solve both of these issues at once, we implemented a simple Docker Compose YAML parser in the orchestrator, which validates the specification against these bad stanzas, and looks up any container images specified within to see if they are stored on the orchestrator for that user. If so, they will be uploaded and loaded into the system under test. Any non-existing images specified will be fetched from public repositories on the system under test.

4.7. Setting Up a Testing Environment

Because the prototype was completely untested and a general testing environment was not yet in place at Nerdalize, an environment had to be designed and created. Several options were considered and tested, and we will describe our experiences with each in this section. The environment had to be able to run tests on each new Git changes, and be able to create and run Docker containers, as this was what we used to test and deploy the benchmarker.

[15]https://docs.docker.com/compose/

4.7.1. Strider

Our initial choice of testing environment was Strider[16], a continuous integration and delivery system written in Javascript. We initially chose it for its seeming flexibility and attractive interface, despite being a relatively new software package. Initial setup was easy, and it seemed to respond fine to Git changes.

However, after adding Docker integration via its built-in plug-in system, fetching new changes from the Git server suddenly started breaking. It seemed the Docker plug-in interfered with the Git module, so that the Git module tried to run some commands locally on the server, and some in the Docker application container, breaking the test procedure.

A bug was filed in Strider's bug tracker, but this turned out to be a deep dependency issue that was not trivially solvable. Since both Git and Docker support were vital to our project and testing procedure, we set out to find an alternative to Strider for our needs. Our next candidate was an established, well-known solution in the automated testing world: Jenkins.

4.7.2. Jenkins

Being unable to use Strider for our needs, we set out to try an established solution, the Java-based Jenkins[17] automated testing system. It too had Git and Docker integration, as well as support for a lot of different programming languages and testing software.

Having learned from our experience with Strider, we tested the Docker integration first. It seemed to work, but was limited in its possibilities: it could only run the tested software in an already existing application container, instead of creating a container from the software code base first.

As the automated instructions for creating a container were contained within the software repository for the benchmarker, this meant it could not test the application proper within the testing container designed for it. This meant we could not properly test the benchmarker, and fixing this would require a rewrite of Jenkin's Docker plug-in. This made Jenkins unsuitable for our needs as well, and we moved on to the final candidate on our shortlist: GitLab CI.

4.7.3. GitLab CI

GitLab CI[18] is a relatively new player in the continuous integration market, a spin-off project from the GitLab[19] source code management system. While very promising in features and interface, it was initially lower in our candidate list due to requiring a GitLab installation where the source code is hosted for its integration. However, after the unsuccessful experiences with the other two systems, we decided to try it out.

GitLab CI's method of determining how to run tests was different from both Strider and Jenkins: where the latter two required language and deployment-specific plugins, GitLab CI allows one to simply specify the commands they needed to run in a declarative file called `.gitlab-ci.yml` in the code base. Aside from being very flexible, this had the additional advantage of changes to the testing procedure being recorded as part of your source code history, so that older versions can be re-tested just as easily when the procedure has changed.

While GitLab CI also had Docker integration, this was similarly inflexible to Jenkins's, and we ended up not significantly using up. In contrast, the simple declarative file contained all that was needed to run the tests in a very straight-forward way, using Docker manually. This also allowed us to trivially add SonarQube integration, despite no such plug-in existing for GitLab CI.

4.7.4. Conclusion

Our experiences with the three automated testing systems we tested are summarized in table 4.7. We settled on GitLab CI due it being the only continuous integration suite that could fulfill our three requirements: Git integration, Docker integration and SonarQube integration. It achieved this thanks to its flexible testing procedure configuration.

We did have to move the source code hosting to a custom GitLab instance, however. This ended up being beneficial as well, as it provided us with easy issue and milestone tracking, and in-line code

[16]http://stridercd.com
[17]https://jenkins-ci.org/
[18]https://about.gitlab.com/gitlab-ci/
[19]https://about.gitlab.com/features/

reviews.

Table 4.7: Feature summary of tested CI systems.

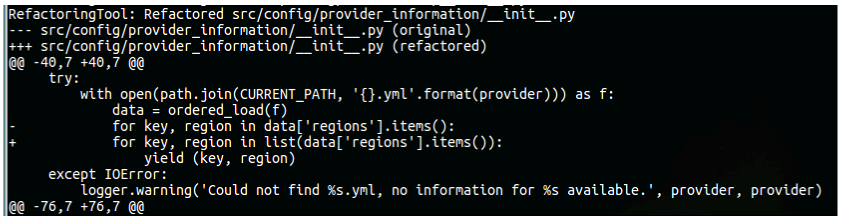
Software	Open Source	Git Integration	Docker Integration	SonarQube integration
Strider	*	Breaks with Docker	*	Unknown
Jenkins	*	*	Inflexible	*
GitLab CI	*	*	*	*

4.8. Porting to Python 3

As part of portability and support improvements to the orchestrator, we decided it had to be ported to Python 3, a source-incompatible successor to the original programming language used, Python (afterwards referred to as Python 2). Because some of the deployments of the benchmarker may still run on non-Python 3-compatible environments, the source code had to be kept compatible with Python 2 as well. Thus, we set out to create a code base compatible with both Python 2 and 3.

We decided on an approach of first porting to Python 3, then fixing what is broken under Python 2. Because we already had the prototype running under Python 2, spotting breakages under Python 2 would be easier than under Python 3, where a breakage might potentially be due changed Python 3 semantics.

We started by using a conversion tool included with Python 3, `2to3`. This tool performs, as much as possible, an automated conversion of Python 2 source code to Python 3, and indicates issues that were not automatically convertible. Example partial output of this tool is shown in figure 4.4. While far from being a perfect tool, it provided us with a good base to start the porting efforts from. After manually addressing the non-convertible segments, we did multiple full syntax check runs over the source code base with Python 3 to confirm at least the source code syntax was conformant.



```
RefactoringTool: Refactored src/config/provider_information/__init__.py
--- src/config/provider_information/__init__.py (original)
+++ src/config/provider_information/__init__.py (refactored)
@@ -40,7 +40,7 @@
     try:
         with open(path.join(CURRENT_PATH, '{}.yml'.format(provider))) as f:
             data = ordered_load(f)
-            for key, region in data['regions'].items():
+            for key, region in list(data['regions'].items()):
                 yield (key, region)
     except IOError:
         logger.warning('Could not find %s.yml, no information for %s available.', provider, provider)
@@ -76,7 +76,7 @@
```

Figure 4.4: Example `2to3` output.

After this initial conversion, we started automatically and manually testing the new source code to ensure semantics hadn't changed. This caught quite a few issues that were left unaddressed by `2to3`. For instance, a text string and raw binary data had the essential same type in Python 2: `str`. In Python 3, this was split up into the `bytes` and `str` types, and so at certain points in the source code semantic decision had to be made about what kind of data was being handled.

With the port to Python 3 complete, we began the reverse process to see what functionality was broken in Python 2 by the changes. This was mostly painless, as a lot of the changes made were backwards-compatible. The most significant change was the change of some standard library locations, requiring us to accommodate both possible locations. The approach we took is shown in figure 4.5.

```
## First:
from pipes import quote

## Now:
try:
    # Python 3 location.
    from shlex import quote
except ImportError:
    # Python 2 location.
    from pipes import quote
```

Figure 4.5: Importing from multiple possible locations in Python.

5 Experimental Testing of the Cloud Benchmarker

5.1. Overview

During the course of the project, we have continuously kept track of the quality of the code base and its functioning. We detail the measurements we used for code quality in section 5.2. section 5.3 expands on our testing methodology.

When creating a measurement tool, making sure the overhead of said tool doesn't drastically influence the metrics is important. In section 5.4 we describe our measurement of the overhead of the benchmarker.

We have also performed some experiments on cloud providers to demonstrate that the benchmarker is capable of performing these. We compare different providers for a single workload in section 5.6. Additionally, we run multiple workloads on different zones of the same provider in section 5.7, to show that these zones have performance differences. Finally, we compare the provisioning time of different providers in section 5.8. Full measurement data for all experiments can be found in chapter D.

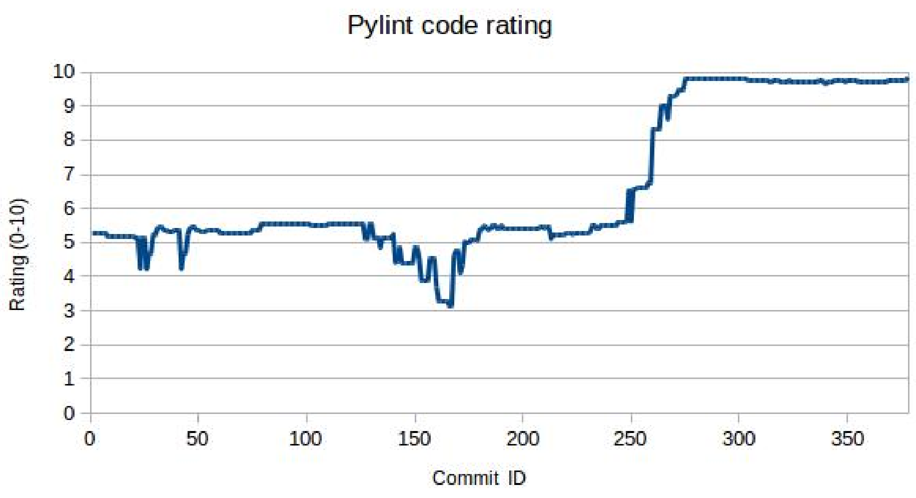


Figure 5.1: Pylint rating for the code base over time.

5.2. Code Quality

In order to measure the quality of our code we have used three tools: SonarQube[1], PEP8[2] and Pylint[3]. For SonarQube, we used the default configuration with all warnings enabled. For PEP8 we ignored certain whitespace rules, as the implications of these rules seem irrelevant to us and we do not agree with the aesthetic impact of these rules. In Pylint, we ignored several rules:

- The rules for enforcing variable name style contained too much false positives, such as the case of assigning a class object to a variable.
- The rules forbidding wildcard imports, as this was used in the configuration module to allow local override of configuration variables.
- The rules forbidding relative imports. Relative imports do not break anything in Python 2 and are actually mandatory in Python 3, so we had to use those for compliance.
- The rules forbidding unused arguments. The provisioning and deployment classes had to follow a strict API contract, and subclasses would not always use the arguments given to them. This was normal and to be expected.
- Finally, the rules forbidding too broad exception catch clauses. In certain parts of the orchestrator, we do not know nor care about the kind of exception that was thrown, but we know something went wrong in the run regardless and do not want to crash the entire orchestrator, but cleanly terminate the run.

We use all three of these tools because in part they do different things and we consider the overlap useful. None of these tools are perfect and one might catch what the others miss. For instance, all three check code style, but only Pylint and SonarQube delve deeper into code semantic analysis

Pylint assigns a rating to the code based on the amount of violations that are present. Throughout the project, this rating has fluctuated. In figure 5.1 it is apparent that special focus was put on the code quality after commit 250, after several features were already implemented. It is also clear that the code quality has increased significantly from where the prototype was at.

[1]http://www.sonarqube.org/
[2]https://pypi.python.org/pypi/pep8
[3]http://www.pylint.org/

5.3. System Testing

We have performed a myriad of different tests on the benchmarker during development. For automated testing, we created unit tests for the appropriate modules, integration tests for multiple modules and system tests to check if the entire system processes are integrated properly.

At the start of the project we would run all of these tests for every commit. However, they have significant disadvantages: cost and time. Some tests actively provisioned a machine at a cloud provider to make sure provisioning works. Before the implementation of the custom provisioning method, this was the only way to test that running a benchmarked worked at all. Aside from the obvious cost issue, this would lead to several minutes added to our test time. After we completed the custom provisioning method, we switched to running the tests locally on the CI server, so that at least the non-provisioning parts of the benchmarker can be tested quickly and for free. We still run the (automated) provisioning tests with instances in the cloud when we are testing manually or for a stable release.

A consequence of not running provisioning tests to cloud providers on every commit was that the test coverage for these runs was not representative. Without these tests, the coverage percentage would remain hovering around 80% total system coverage. However, running with the costly tests included, it would jump to 90 to 95% coverage. While we could use the every-commit coverage to track increases and decreases in coverage, it was sadly not useful anymore as an absolute indicator.

Not all testing has been automated. For instance, the trade-off for GUI testing would involve a lot of time upfront for automated testing, but relatively little time for testing it manually. Especially considering that developing new features usually always involves interacting with the GUI. For features, the relevant GUI parts are tested manually and for stable releases, all features in the GUI are tested.

5.4. Measurement Overhead of the Benchmarker

When benchmarking, our aim is to measure how an application performs on a system under test. Unfortunately, the benchmarker itself incurs resource usage on the system under test. The deployment of the workload and the monitoring of the resources used by the system under test are considered separately. For the deployment of the benchmark we use our custom implementation, which we'll refer to as the deployment tool, and Docker Machine. Since the Libcloud deployment method is heavily based on the custom deployment, we assume these to be equal. For the monitoring of the resources we use the collectd[4] monitoring program, which we will refer to s the measuring tool. In our benchmarker, we are interested in the metrics displayed in table 5.1.

Table 5.1: Interesting metrics for overhead testing.

Metric	Unit	Description
Provision time	seconds (s)	The duration from requesting a machine to having access to it.
CPU time	seconds (s)	The duration of the test.
CPU load	jiffies	The amount of total CPU consumed by the test.
Memory usage	bytes	The memory usage over the time of the test.
Disk read/write operations	-	The amount of disk read and write operations performed.
Disk read/write bytes	bytes	The amount of bytes moved by disk operations.
Network in/out packets	-	The amount network packets sent and received.
Network in/out bytes	bytes	The amount of bytes moved by network packets.

The most important of these metrics are the runtime and the network output, because they are the main cost drivers for cloud computation in current common cloud provider cost models.

We formulate a hypothesis for the test plan: "The overhead of the deployment system with regards to these metrics does not depend on the content of the benchmark".

We expect this because the deployment tool simply performs some operations before and after the workload has been run. We do not know the inner workings of Docker Machine but we expect it has been implemented in a similar way. The operations are the same for every benchmark of the same format and as such should not be influenced by the workload itself.

[4]https://collectd.org/

5.4.1. Test Plan

We will run Docker-based workloads, and assume that Docker is already installed on the system under test such that no configuration is necessary. The installation of Docker as done by the deployment tool is done in the same as a manual installation would be. The same applies to Docker Machine. We perform these tests on a physical computer located on-premises. This is preferred to deploying to the cloud, because cloud providers might allocate different resources to different machines between runs, or workloads with different usage patterns.

Table 5.2: Tested workloads in overhead measurement.

Workload	Image	Parameters	Systems
A	Sleeping	60 seconds	All
B	Sleeping	180 seconds	All
C	Prime calculation	Eight cores, numerical range 100-80,000,000.	All

For the actual test we ran the workloads depicted in table 5.2. We ran each of these workloads three times across every method. We defined the benchmarker's overhead as the amount of resources the deployment method uses on top of running the workload manually. From the different duration of the 'sleep' workloads we could easily derive whether the overhead of the benchmarker is runtime-dependent or not. From the comparison between the sleep workload and the prime workload we could derive whether the overhead depends on the system load.

Of course, our measuring tool has some overhead as well. Unfortunately, the operating system does perform additional other operations during our workloads. Because of the difficulties involved in measuring our measurement tool, this is left out of scope.

For our manual run, we launched a monitoring container and a workload container. The former consisted of a collectd image sending the monitoring data to our InfluxDB time series database. We did this rather than storing the data locally so that we could measure the network overhead of the benchmarker. On the system under test we performed the sequence of operations depicted in figure 5.2.

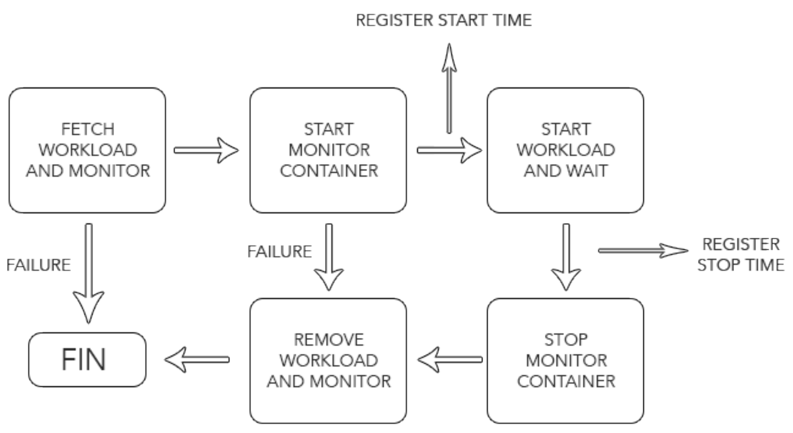


Figure 5.2: Overhead test plan sequence diagram.

The time measurements takes place after starting the monitor and before stopping it, because starting and stopping the monitor image also incurs overhead.

For the deployment tool and Docker Machine, the runs are orchestrated from a virtual machine on the same local network. This means that in these tests the network latency is also included in the runtime. The monitor image is not stopped until the system under test communicates to the orchestrator that the workload is done. The orchestrator then sends a stop command for the monitor. This results in some more data being sent by the monitor and some measurement inaccuracy. This inaccuracy increases as the network latency increases. Because the workload has stopped running at this point, the amount of resources used should be limited. We do not explore the effects of increased network latency.

The tests have been run on a laptop running Ubuntu[5] 14.04 with the graphical user interface disabled. This is not ideal as there were still some background processes running. However, these should not interfere in any significant measure with the runtime and network out.

5.4.2. First Run Results

Table 5.3: Average overhead and standard deviation for each metric, compared to manual deployment original measurement.

Combined Overhead for each Deployment Method	Custom		Docker Machine	
	Average	Standard Deviation	Average	Standard Deviation
Runtime (s)	1.56	2.38	2.33	1.15
Max memory used (Byte)	9055345.78	24568962.97	-12273436.44	23987835.98
Network out (Byte)	23864.56	80052.75	43763.78	91603.45
Network in (Byte)	102.56	235.15	163.11	531.74
Disk read operations	-109.6	566.5	-110.9	566.5
Disk write operations	1478.7	3036.5	1388.2	3469.4
Disk read (Byte)	-453290.7	2320756.6	-458752.0	2320650.6
Disk write (Byte)	278226716.4	157110972.6	172975900.4	268633417.4
Total cpu (jiffies)	528.11	626.95	192.78	453.65

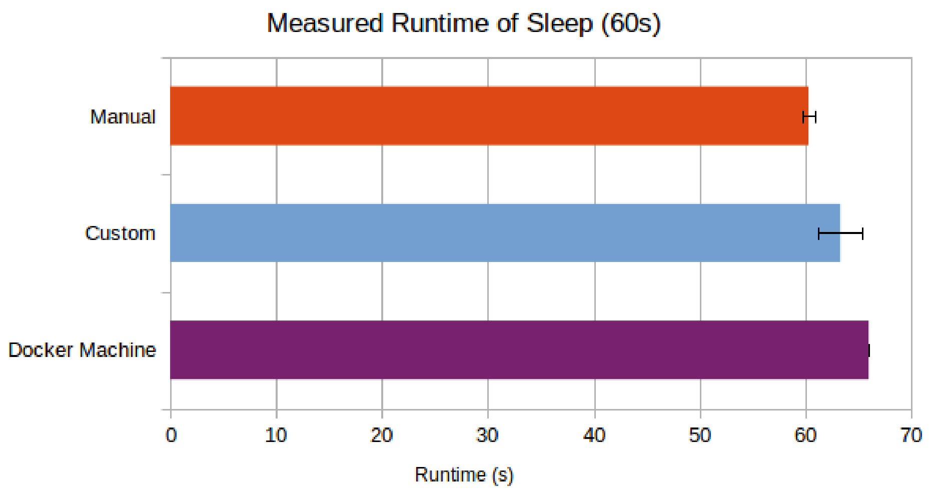


Figure 5.3: Runtime of a 60-second sleep workload across various provisioning methods original measurement.

As can be seen in table 5.3, the overhead of the runtime is rather large. This is mostly due to the results of the 'sleep' benchmark as depicted in figure 5.3. However, we derive this overhead to be constant, seeing as longer benchmarks have the same or less absolute overhead.

On the other hand, the overhead of the memory usage is limited, within a couple of dozen MiB. The overhead in network out is even more constrained, being under one MiB.

For the disk operations, the background processes have introduced so much noise we can not say anything conclusive about this.

[5]http://www.ubuntu.com

5.4.3. Second Run Results

Because of the overhead on the measurement of the runtime, we adjusted the way in which this was measured in the orchestrator, putting the measurement point closer to the actual deployment. Accordingly, we ran these tests again to see to which degree this has improved. The results of this re-run are shown in table 5.4.

Table 5.4: Average overhead and standard deviation for each metric, compared to manual deployment re-run measurement.

Combined Overhead for each Deployment Method	Custom		Docker Machine	
	Average	Standard Deviation	Average	Standard Deviation
Runtime (s)	-1.22	1.83	-0.67	1.83
Max memory used (Byte)	-13322012.44	8563068.51	-14346467.56	6401454.84
Network out (Byte)	1390.67	29542.26	11917.67	60851.47
Network in (Byte)	111.89	702.61	169.67	696.60
Disk read operations	-0.7	9.0	0.0	8.3
Disk write operations	-742.4	5049.3	-696.0	4278.4
Disk read (Byte)	-9102.2	52026.2	-6371.6	50053.9
Disk write (Byte)	187862584.9	293713378.5	102311708.4	168806883.6
Total cpu (jiffies)	-674.67	889.25	-747.00	810.60

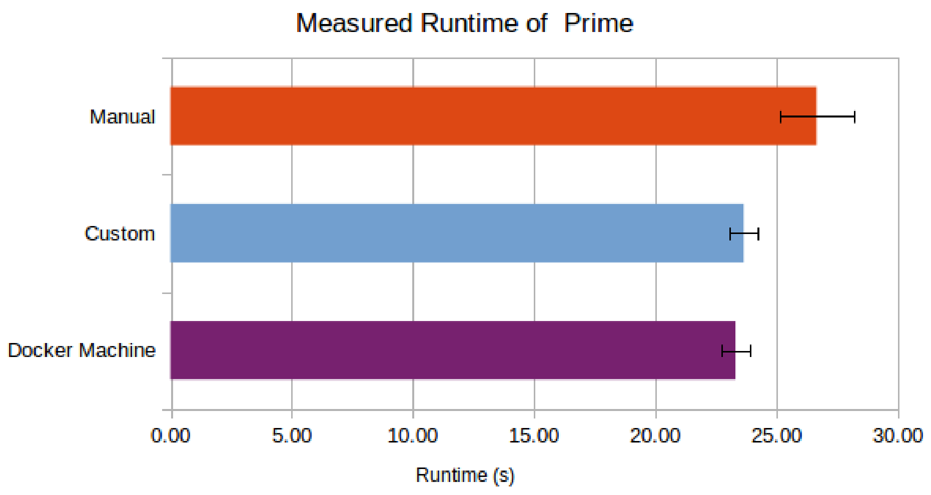


Figure 5.4: Runtime of a prime workload across various provisioning methods re-run measurement.

From this, it shows that the runtime is now shorter using our deployment methods. This is mainly caused by the runtime on manual deployment being very long. This might be due to a variety of external factors. For instance, the system temperature at the start of the test may cause the CPU not to be turbo boosted, degrading its performance. The runtime of prime on the earlier manual run was a consistent 24 seconds, to which the results of our deployment tools are very close. From these tests we can conclude that now our measurement of the runtime is more accurate with no significant difference to manual deployment.

5.4.4. Provisioning Overhead

We have also measured the overhead of provisioning to a local computer using Docker Machine. The measured provisioning time is depicted in table 5.5. The overhead is on average seventeen seconds, which is a significant time, but as shown in section 5.8 still smaller than the inter-provider differences.

Table 5.5: Provisioning time measured using Docker Machine on a computer in the LAN.

Provisioning time(s)	Average (s)	Standard Deviation(s)
17	17	1
16		
18		

5.5. Measurement Accuracy

While performing initial testing, it turned out that several measurements were not interpreted correctly in the prototype. One main reason for this was mistaking absolute values that only keep increasing for values that represent a differential between two points in time.

In addition, the CPU counters were read out without specifying which exact counter type to read. As such the difference between two different types of CPU counters measurements would be taken, i.e. the system CPU time counter and the user CPU time counter. The benchmarker now uses the difference between the user CPU time counters, as this metric indicates the amount of jiffies used by the user-space workload most accurately.

Another interesting inaccuracy in the measurements was causes by the disk measurements. In Linux and UNIX-like systems, drives have a name and partitions on that drive are denoted by that name followed by a number. For example, a drive `sda` with partitions `sda1` and `sda2`. The prototype implementation summed all partition values *and* the disk value together. This resulted in the disk measurements being twice the actual value.

5.6. Benchmarking across Different Providers

In order to show that the benchmarker can in fact compare providers we make a case study. We compare the cost and runtime of a 'Psipred' workload on different instance types of the three main cloud providers: GCE, Amazon EC2 and Azure. This test setup is depicted in table 5.6. We choose Pspred because it folds 5000 proteins. Protein folding is not a dependent process, making it conveniently parallelizable.

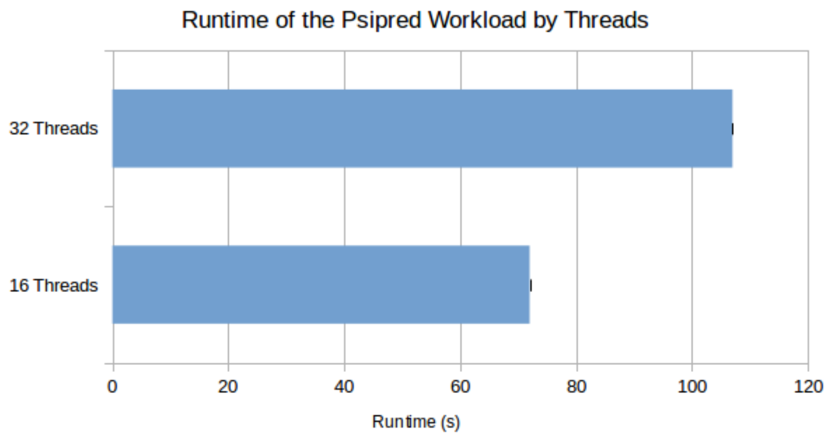


Figure 5.5: Result of running the Psipred benchmark on a test server at Nerdalize with 16 physical and 32 virtual cores.

First, we determined whether running a number of threads equal to the amount of virtual or physical cores would be more effective. As can be seen in figure 5.5 the workload is finished faster when run with a number of threads equal to the amount of virtual cores. For each provider we have chosen four instances: the two smallest instances of the two instance types most suited for CPU-heavy purposes. Examples of instance types are Google Compute Engine 'high compute', Azure 'Ax' and Amazon 'c4'. Instances within these types tend to scale linearly in their resources such as the amount of virtual cores or memory available. Our chosen instances all have this characteristic, as depicted in in table 5.6. Our hypotheses is that the workload will cost the same to run on the chosen instances of the same instance type.

Table 5.6: Different instances on which the Psipred workload has been run.

Provider	Instance	Cores	Memory (GB)
GCE	n1-standard-1	1	3.75
	n1-standard-2	2	7.5
	n1-highcpu-2	2	1.8
	n1-highcpu-4	4	3.6
Azure	D1	1	3.5
	D2	2	7
	A1	1	1.75
	A2	2	3.5
EC2	c4.large	2	3.75
	c4.xlarge	4	7.5
	c3.large	2	3.75
	c3.xlarge	4	7.5

5.6.1. Results

As can be seen in figure 5.6, the costs for each instance type are within the standard deviation of eachother, with the exception of Google Compute Engine's n1-standard. We note that n1-standard-2 took the same amount of time as n1-highcpu-2, as can be seen in figure 5.7. We expect that the CPUs in both instances are the same and that n1-standard-2 simply has more memory. The memory usage for all of the runs was under 500 MiB, so we can determine that this extra memory remains unused. This workload is cheapest to run on the Google Compute Engine's n1-highcpu instance type, although Amazon's EC2 instances are not far off. To run it on Azure, on the other hand would cost more than twice as much. With this, we have shown that we can use the benchmarker to get a price comparison for our workload.

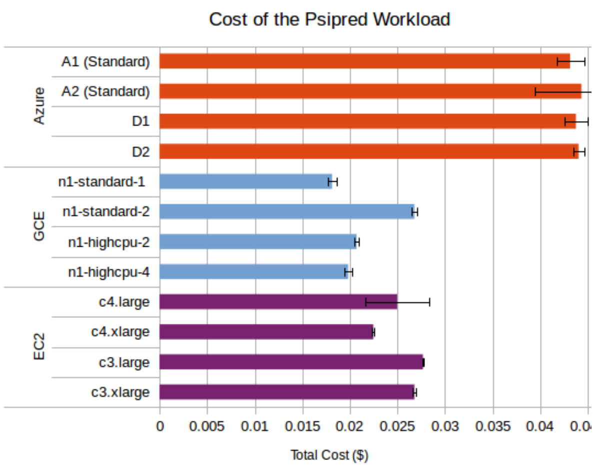


Figure 5.6: Cost result of running the Psipred workload on different instances at different providers.

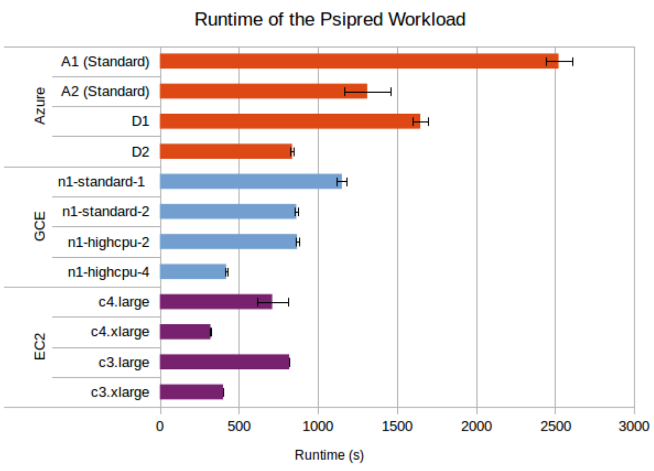


Figure 5.7: Runtime result of running the Psipred workload on different instances at different providers.

5.7. Benchmarking Different Workloads across Different Zones

Google Compute Engine has an interesting approach to handling multiple generations of processors. Rather than creating a new instance type, they use the same in a different zone with the same pricing [6]. The newer generations processors have slightly lower clockspeeds, Sandy Bridge runs at 2.6GHz, Ivy Bridge at 2.5GHz and Haswell at 2.3GHz [7].

We expect that this will lead to performance differences in running different workloads. Some workloads will benefit more from a new architecture than others. In order to verify this we run two workloads on each of the zones in europe-west1. The first is an image that computes prime numbers, the second is a raytracer. Because the cost is identical for these instances in different zones, we only consider the runtime. All tests have been run on n1-highcpu-2 instances.

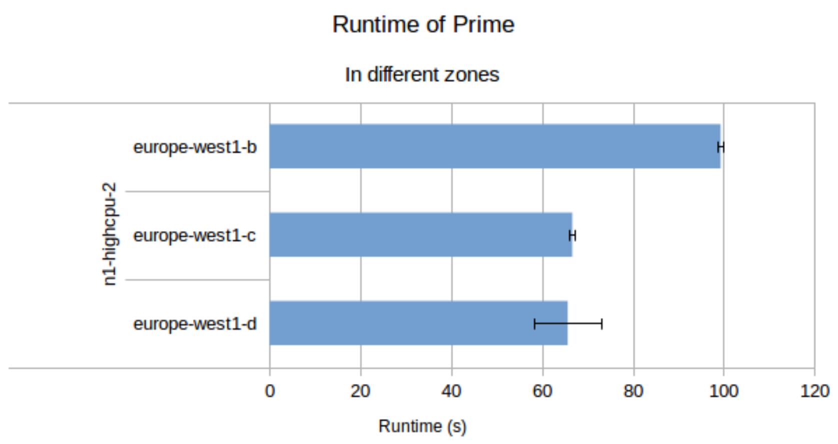


Figure 5.8: Runtime result of running the prime workload on n1-highcpu-2 instances in different GCE zones.

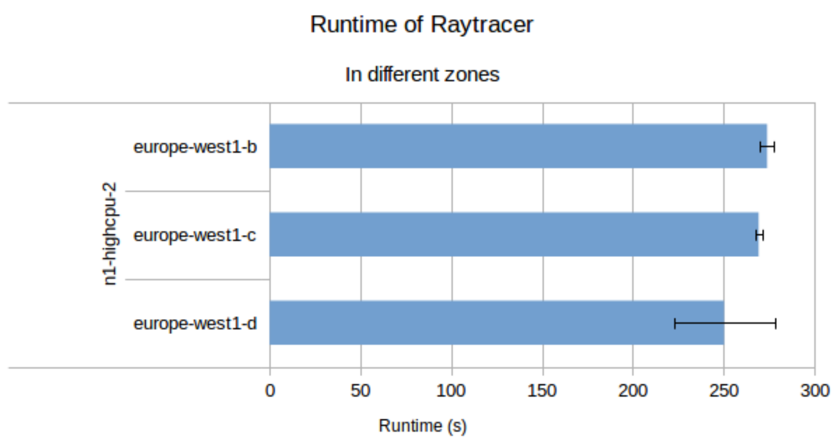


Figure 5.9: Runtime result of running the raytracer workload on n1-highcpu-2 instances in different GCE zones.

5.7.1. Results

As can be seen in figure 5.8, zone B takes significantly longer to finish the prime workload. Zone C and D perform about the same, although the variance of D is significantly larger. For the raytracer on the other hand, as can be seen in figure 5.9, zone B and C are very close in performance and zone D again finishes the workload the fastest on average, with the highest variance. From these results we conclude that there can be significant performance differences between zones on different workloads.

[6]https://cloud.google.com/compute/docs/zones
[7]https://cloud.google.com/compute/docs/machine-types

5.8. Provisioning Time

Another interesting factor in cloud computing is the provisioning time, or it takes before the machine you are provisioning is available. During the last phase of our project we have measured this time for the instances we have provisioned. For Azure and Amazon EC2, we chose Docker Machine as the provisioning method. For Google Compute Engine, we Libcloud. We have measured the overhead of Docker Machine in section 5.4.4. As such, we have added this overhead to the lower bound of the error bar. Libcloud has not been measured in a similar way. However, Google Compute Engine is by far the fastest in provisioning, even with Libcloud's overhead. Because we have done experiments on different Google Compute Engine zones, we have data concerning the provisioning time on different zones. As can be seen from figure 5.10 the difference between zones is very small. Amazon EC2 takes over three times as long and Azure takes over fifteen times as long.

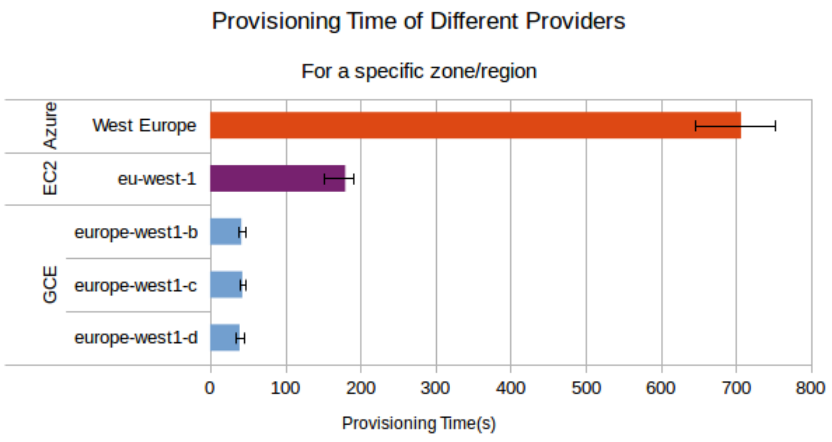


Figure 5.10: Provisioning time results on different providers in specific zones.

This experiment shows that by far, Google Compute Engine is the fastest in provisioning a machine, somewhat closely followed by Amazon EC2, with Azure coming in last.

5.9. Summary

Using these test plans, we have not only done interesting experiments on various cloud providers in the wild, but also shown the effectiveness of the final benchmarker implementation in being able to create these measurements. We have compared providers and their instance types, CPU generations across fixed- and floating-point benchmarks, and provisioning times. We have also measured the deployment overhead of the benchmarker and have shown it to be insignificant.

6 Qualitative Product Evaluation

6.1. Overview

In this section, we look back on the final implementation of the benchmarker and verify it complies with the stated goals in chapter 2 and chapter 3. First, we list the extensions that we implemented in section 6.2. In section 6.3, we then match up those extensions with the design goals as specified in section 2.7. Finally, we formulate and discuss the success criteria for this project in section 6.4, and determine whether or not according to them this project was successful.

6.2. Implemented Extensions

In this section we evaluate whether the given extensions were successfully implemented. The extensions we refined and implemented are shown in table 6.1. This differs from the initial list in table 2.3 by a few points:

- Certain extensions are moved up or down in priority. This is a result of the stakeholder priorities changing during the agile development process.
- New extensions have been added. During the development process, new ideas and priorities arose, and after meetings with the stakeholder they were added to the list.
- Almost all extensions are refined. This was the result of regular stakeholder meetings.

Table 6.1: Final list of prioritized, refined and implemented extensions, sorted by priority.

#	I	§	Abstract	Refined
1	*	5.3	Testing Improvement	Write tests for code base for full coverage
2	*	3.5, 4.7	Continuous Integration, Continuous Deployment	Set up GitLab, GitLab CI (see section 4.7) and SonarQube, integrate with existing services
3	*	4.8	Python 3 Re-architecture	Port existing code base to Python 3.3
4	*		Development Setup	Set up easy deployment for existing benchmarker using Docker Compose.
5	*	3.4.3, 4.3	Orchestrator Split	Isolate orchestrator to separate container, split from web application.
6	*	3.4.5	Centralize Configuration	Move split-up configuration modules to one central configuration module, convert end-user or machine-editable to YAML format.
7	*	4.4	Measurement Accuracy	Upgrade InfluxDB to 0.9, improve and tweak collectd configuration.
8	*	4.5	Extra Provisioning Methods	Implement provisioning through Apache Libcloud
9	*	4.5	Extra Provisioning Platforms	Implement support for Google Cloud Engine
10	*	4.5	Custom Provisioning Platforms	Implement support for deploying to any Linux host
11	*		Benchmarking Region Variability	Refactor code base and data model to support comparing regions
12	*	4.6	Complex Architecture Support	Implement support for Docker Compose workloads
13	*	4.6	Extra Workload Formats	Implement support for Docker Compose workloads
14	*	3.4.1	Latency Measurement	Measure the latency in provisioning a machine
15	*	3.4.5	Reporting Improvement	Provide more detailed status information steps: add 'provisioning' and 'configuring' steps; show complete benchmark run log in web application.
16	*		Production Setup	Set up production environment for the benchmarker.
17	*		Performance Variability	Allow running the same benchmark at a later time and comparing the results.
18			Multi-User Support	Implement client accounts so that clients can run their applications themselves.
19			Advanced Cost Model	Implement automatic price updates
20			Cluster Architecture Deployment	Implement support for using Docker Swarm.
21			Resource-Cost Extrapolation	Implement linear scaling prediction of price with size of application, relative to tested application.

I: Implemented; §: Discussed in paragraph

For the sake of brevity, not all implemented extensions were described in chapter 3 and chapter 4. Only the ones with significant design and/or implementation impact were chosen to be expanded upon, as a lot of features were relatively simple to design and implement.

Comparing to the initial list in section 2.8, we ended up implementing more than initially envisioned. All of these additional extensions were a result of priorities and desires of the stakeholder changing over the development process, and the team adapting the schedule and sprint planning to it.

Having implemented more than the initial schedule planned for, and after discussion with the stakeholder, we have confirmed that all extensions required for a minimum viable product (MVP) were implemented.

6.3. Design Goals

Now, we relate the implemented extensions to the design goals and aspects laid out in section 2.7.

Table 6.2: Addressing of design goals from section 2.7 and their aspects from impemented extensions.

Design Goal	Aspect	Addressed in extension
Correctness	Accuracy Compatibility Error detection	Measurement Accuracy, Latency Measurement Extra Provisioning Methods, Platforms, Extra Workload Formats Testing Improvement, Continuous Integration, Continuous Deployment
Robustness	Error handling Error logging	Orchestrator Split Reporting Improvement
Extensibility	Provider support Workload support	Extra Provisioning Methods, Platforms Extra Workload Formats
Security	Workload confidentiality Credential confidentiality Measurement integrity	Orchestrator Split Centralize Configuration
Maintainability	Support Testing	Python 3 Re-architecture, Development Setup, Production Setup Testing Improvement, Continuous Integration, Continuous Deployment, Development Setup

As seen in the overview in 6.2, all aspects of all design goals have been addressed, except for one, most in more than one way. Measurement integrity has been left unaddressed in any new extensions partially due the fact that local integrity was already guaranteed by the use of containerized applications, running fully isolated from other processes on the host operating system.

We now evaluate the success of meeting the design goals individually:

1. **Correctness:** Vast improvements have been made to the measurement accuracy of the benchmarker, both in terms of the number of metrics as the accuracy of these individual metrics. In addition, automated testing has lead to a far greater rate of error detection within the benchmarker, and the orchestrator split has made error handling easier and more robust. We evaluate this design goal as *met*.

2. **Robustness:** Automated testing has made breakage in changes to the code base far easier to detect, analyze and correct. Furthermore, the orchestrator split properly isolates the vital

apparatus of the benchmarker, the orchestration mechanism, from the user-facing part, making sure they can not influence each other beyond strict API bounds. In addition, the benchmark run logging provides detailed logs in case something does go wrong, for analysis by the users or developers. We evaluate this design goal as *met*.

3. **Extensibility:** Having made large improvements to the orchestration architecture of the benchmarker, it is now fully pluggable with a separation between provisioning, configuration, and deployment. This separation made it far easier to implement the two extra provisioning methods, custom provisioning and Libcloud provisioning, and extra workload format, Docker Compose for complex workload architectures. We evaluate this design goal as *met*.

4. **Security:** With the orchestrator isolation and configuration centralization, the handling of cloud provider credentials is now a lot more secure, and the new orchestrator architecture ensures that customer workloads get properly cleaned up, no matter what happens. This prevents sensitive data from leaking. Although we were not able to implement more measures for ensuring measurement integrity, we still evaluate this design goal as *met*.

5. **Maintainability:** With an automated testing environment, an easy-to-deploy development environment and a working production environment, the benchmarker has become significantly more mature and easy to develop and test for. The Python 3 port has ensured the underlying software will remain supported for the years to come. We evaluate this design goal too as *met*.

We conclude that all the design goals that we initially laid out in section 2.7 have been met, and that the implementation of them was successful.

6.4. Success Criteria

Finally, with the required extensions implemented and the formulated design goals met, we take a closer look at the success criteria for each three involved parties, and see if they have been met.

- **Students:** For us, the success criteria involves meeting all the stated design goals and being able to leave Nerdalize behind with a working production-level implementation of the benchmarker they described in the initial assignment. As shown by section 6.3, and the fact that the benchmarker is running in production internally at Nerdalize, we feel our success criteria have been widely *met*.

- **Nerdalize:** From meetings with the stakeholders at Nerdalize, we learned Nerdalize's success criteria involved the implementation of the desired functionality within the project timeframe. As shown by section 6.2, all desired extensions, and more that have come into play during the development phase, have been implemented. As such, we consider Nerdalize's success criteria *met*.

- **TU Delft:** The university requires an academic component to the project, as well as the students undergoing a full development process, from start to finish. The learning objectives as stated in the project manual[1] are as follows:

    1. The students can carry out an entire software development cycle with success, from researching solutions through testing the product, in a team of developers addressing a real-world problem. We consider this learning objective *met*, as described in this thesis.
    2. The students can effectively, in collaboration with a coach and a client, choose a development strategy and execute a development process according to that strategy We consider this learning objective *met*, as described in this thesis and evidenced by he final product.
    3. The students can establish the necessary quality requirements for a product and carry out the tests necessary to determine that the product fulfills those requirements. We consider this learning objective *met*, as described in chapter 2, chapter 5 and chapter 6 in this thesis.

[1]https://homepage.tudelft.nl/q22t4/Resources/GeneralGuideTUDelftCSBachelorProject.pdf

4. The students can present a complete and convincing explanation of the development process and the product results. We also consider the learning objective *met*, as described in this thesis.

From meetings with Dr. Iosup, our university supervisor, we learned of additional success criteria:

1. Being able to demonstrate that the benchmarker is flexible and can provision to multiple cloud services. This criterion has been *met*, as described in section 4.5 and the tests in section 5.6.
2. Being able to demonstrate that the benchmarker is flexible and can work with multiple workload formats. This criterion has been *met*, as described in section 4.5 and the tests in section 5.7.
3. Being able run a workload supplied by Nerdalize representing one of their customer's applications, to prove effectiveness of the benchmarker in Nerdalize's domain. This benchmark has been ran internally in Nerdalize and is not further discussed in this report for customer privacy and integrity reasons. This criterion has thus been *met*.

We consider TU Delft's success criteria *met* in all aspects.

We can conclude that all success criteria from all parties involved have been met, and that the project as such can be called a success.

7 Process Evaluation and Recommendations

7.1. Overview

In this final chapter, we will discuss and evaluate the development process, highlighting strong and weak points experienced during this project, in terms of development methodology, product management and personal communication. We will finish with providing pointers for future developers with information about potential future features on the benchmarker project.

7.2. Development Process Evaluation

In this section we will expand on the various highs and lows of various process aspects that we encountered in this project.

7.2.1. Development Methodology

During the development phase Extreme Programming (XP) and agile principles, as described in section 4.2 were followed. This generally provided us with a well-working and effective process, but occasionally did lead to issues.

Agile principles allowed us to quickly adapt to the client's changing needs or requirements. For instance, a few weeks in into the project Nerdalize received a customer they wished to benchmark an application from, for which the benchmarker was not yet ready. Agile and Extreme Programming principles of backlog reordering and iterative refinement allowed us to switch our focus to the parts of the benchmarker that contained the features that said client needed to benchmark their application, and as such Nerdalize ended up successfully benchmarking the client application in time.

Furthermore, the good usage of a backlog board together with GitLab for issue tracking allowed us to keep focus on the work ahead of us, instead of getting drowned in details and potential new features. The sprint milestones kept us motivated to get a feature or product increment done by the deadlines. However, considering the size of some of the changes we have done in the project, we feel that sprints of one week may have been too short, even for a project as short of this one. We often ended up moving features to the next sprint because they could not be completed in time, and felt rushed. In hindsight, we feel that two week sprints may have worked a lot better in giving us the breathing room required to implement large features.

In addition, due the increased focus by the short sprints, we often ended up focusing on implementing the features high on the backlog and the to-do board, but not spending enough time refining the features lower in priority. This resulted eventually in having badly-defined features at the top of the backlog, having to waste time scheduling an appointment with the stakeholder to refine these. More attention to this on our part will be needed for this in the future, but near the end of the project we felt that we had found a good balance between refinement and implementation.

7.2.2. Source Code Management and Testing

Git proved to be a great and competent source code management tool, being very adept in handling code base conflicts and merging different code versions together. Its decentralized nature also helped us work in other places occasionally, where no internet or no access to the internal GitLab server was available.

Due our decision to work using feature branches in Git, and the tendency to work on big features in parallel, we sometimes found our code bases had diverged too much, and had to be manually merged and fixed up. This wasted some precious time, and could have been avoided by integrating from each other's branches more often, instead of just from the main code branch.

The automated tests proved to be very helpful in automatically catching various kinds of breakage and bugs introduced by our changes. However, the continuous integration server proved occassionally unstable, and would fail tests for no reason, or lose internet connectivity. This resulted in wasted time attempting to restart and fix the test server that could have been spent designing and implementing features.

Furthermore, the initial setup took a significant chunks of the first three weeks to get right, while doing research and exploring the inherited code base. However, Nerdalize has now showed interested in applying the system to their development process in general, so we have made another lasting contribution to the development environment within Nerdalize, which is definitely a very positive consequence.

7.2.3. Communication

Most communication within the team ended up being by face-to-face communication or Slack. Not much effort was spent keeping the Trello board up-to-date, as the office provided us with a real-life backlog board. As described in section 7.2.1, communication with the stakeholder was occasionally less frequent than it should have been, leading to not only a lack of feature refinement on our part, but also a potential frustration on their part for not being kept up to date on the process. We have definitely learned from this experience and found a working balance near the end of the project.

Due the rather unique situation of this project taking place over the summer holidays, communication with our university supervisor, Dr. Iosup, was also less frequent than ideal. Compounded with some miscommunication, this led to just a few real-life meetings taking place over the course of the project. Fortunately, Dr. Iosup was able to take this up very well and nevertheless was able to provide us with very useful feedback on the requirements, product, process and thesis. We would like to thank him again for being so accommodating in this situation, and have learned that direct face-to-face contact is not always necessary to provide useful insights.

7.3. Future Work and Recommendations

We expect that the benchmarker implementation will succeed us, and while we feel like a lot was done during the course of this project, it is of course not finished. As this implementation is used in production, we would like to provide a few pointers and subjects for future work for Nerdalize and any other developers continuing this project in the future.

7.3.1. Multi-User Support

Currently, the benchmarker still relies on a single 'administrator'-level user account to upload images and start benchmarks. While the database structure and all internal structures are made with multi-user support in mind, actual user management is not yet fully implemented. Implementing this would give the potential advantage of customers being able to run their applications directly, without requiring Nerdalize intervention. This could optionally be paired with a privilege system, restricting what the user can and cannot do.

7.3.2. System under Test Self-Destruction

When the orchestrator itself suddenly crashes, machines at the cloud providers it is currently benchmarking are never shut down and can incur significant costs. While the current implementation ships with a helper script that can automatically locate and destroy any running benchmark systems at all supported providers, except custom-provisioned ones, a far nicer approach would be to install a self-destruct timer on the system under test, where it automatically shuts itself down or terminates itself, using some internal API call to the cloud provider, if it does not receive any data from the orchestrator over a certain period of time.

7.3.3. Automatic Pricing Updates

With the configuration centralization feature, several file formats were rewritten to declaratively contain cloud provider pricing information in machine-writable configuration files. One nice feature would be a background process that automatically fetches the latest prices from the various cloud providers and updates their configuration files with this information. This would decrease the chance of pricing information generated by benchmarking runs being inaccurate or seen as misleading, as well as remove a maintenance burden from the developers.

8 Conclusion

While cloud computing is becoming more and more ubiquitous, insight into the market is still difficult to come by, due to opaque hardware architectures and pricing models. Existing benchmarking were not sufficient to gain accurate information on specific user applications, as they were solely targeted at existing, predefined benchmarking suites testing a variety of predefined metrics. In the consumer world, requirements shift quickly and vary wildly, and being able to test user applications was an absolute requirement for Nerdalize.

It was clear from the first meeting on that this was project was not ordinary and would involve a fair amount of research into existing academic and commercial solutions. From this research it would have to be decided whether to build on an existing implementation or design a benchmarker from scratch. Having chosen an existing prototype to build on, additional challenges laid ahead in re-architecturing the existing design, instead of starting from a clean slate.

We can now answer the research questions posed in the start of this thesis, in section 2.2:

1. **How can one measure the performance and cost difference of running an arbitrary application across several cloud infrastructure providers, and different instance types within these providers?**
   For arbitrary user applications that can be packaged as Docker containers or Docker Compose infrastructures, we developed a benchmarker from an existing prototype that is able to accurately assess resource usage patterns and pricing information for applications. We have analyzed the desired architectural improvements and implemented the most desirable reachable within the project time frame, chief among which modularization of the deployment architecture, accuracy and overhead improvements, a fully automated testing suite and environment, provisioning to any custom Linux-based machine and deployment of complex workload architectures.

2. **How can one verify the accuracy of such a measurement tool?**
   For validation of the benchmarker design, numerous automated and manual tests have been performed, comparing benchmarker-deployed applications to manually deployed applications, across several cloud infrastructure providers and system types. These tests showed that the benchmarker can measure application performance while not degrading it significantly. Furthermore, measurement data has been obtained from a real-life application running over several zones, indicating a hardware difference between zones for certain cloud providers.

The end result of this thesis is a benchmarker application that Nerdalize can run in production environments, for assessing their strategic position in the wildly growing cloud market, but also as a sales tool to customers in order to visualize their pricing benefit and be able to back it up by actual data, and analyze how applications can potentially be ran more efficiently.

A

Project Formulation from Nerdalize

1. Background

Nerdalize offers Compute as a Service with a competitor-based pricing model. This means Nerdalize provides Infrastructure as a Service with an architecture that will be most attractive for CPU-intensive batch jobs, at a cost lower than alternative cloud-providers.

The Nerdalize cloud architecture deploys 4 rack servers in a household. The households are interconnected with a VPN via using consumer-grade internet connectivity. Although this creates some new infrastructural bottle-necks as compared to a centralized Data Center approach, Nerdalize will have an advantage in terms of operational cost of the servers. This advantage in cost is used to offer a cheaper service.

Customers want to easily compare the cost of running its compute batch jobs at different cloud providers. However, the cloud market is opaque:

- Heterogeneous infrastructure: different virtualization technology, CPU, networking, and disk types are used.
- Heterogeneous cost models: different prices and models for machine resource and time consumption
- Heterogeneous computational jobs: different jobs have different performance profiles (resource utilization and the influence of infrastructure on turnaround times).

A benchmark that uses a representative computational job, as defined by the customer, can help the customer gain insight in the cost of its example job at various cloud platforms. Given a batch job provided by a customer, the Nerdalize benchmarker runs the job on the specified cloud platforms and instances (including Nerdalize), reporting:

- The resource utilization of the job (CPU, memory, network, disk)
- Turnaround time of the job, per cloud provider
- The associated costs, per cloud provider, differentiated per resource (CPU, data transfer, …)

2. Current Status

A first version of the benchmark system has been implemented in Python, using Docker to deploy workloads on Cloud instances. Along with the workload a monitoring agent is deployed on the instance, which reports resource utilization to the benchmark orchestrator. After the workload is finished, reports are generated that show the resource utilization, runtime, amount of disk and network I/O performed, and the associated cost. To calculate the cost, a cost model is implemented that is modeled per cloud provider.

3. Project Goal

The goal of the project is to improve this benchmarker to be more generally useful for different situations and circumstances. The initial steps are envisioned are:

1. Studying of benchmarking challenges from literature, and existing solutions to those challenges (e.g. the theory and implementation for the publication by Iosup et al).
2. Assessing the current Benchmarker in terms of functionality and implementation quality (both strengths and weaknesses).
3. Assessing requirements for extensions to the benchmarker and validating them.

After the students reported on the above, Nerdalize uses the advice of students and university supervisor to select the improvements, after which the following steps will take place:

1. Development and implementation of the decided-upon requirements into a prototype. This is not necessarily a linear per-improvement process, as multiple improvements may be developed and implemented concurrently.
2. Validation of the decided-upon design and implementation, internally from peer review and experimental evaluation, as well as at the end of the project at SIG.

Some examples of possible improvements include:

- Generally improve/refactor the code base, where necessary to extend it properly.
- Allow for deploying more complex job architectures (e.g. multiple Docker containers, using Docker Compose).
- Allow running jobs over multiple machines (i.e. a cluster).
- Allow for deployment on more additional cloud providers.
- Create more advanced reporting mechanisms (e.g. resources to be monitored).
- Benchmark variability of results, over time.
- Implement a login/multi-user backend.
- Implement a cost approximation/extrapolation method: when running a benchmark on one instance, use the results of previous jobs and the cost model to estimate the price/runtime on instances.

Alongside work on the prototype implementation of these improvements, a report is expected to be written over the course of the project. The contents of this report will be subject to a defense at the end of the project. In addition, the prototype codebase will be subject to a evaluation by the Software Improvement Group (SIG). Time will be allotted to ensure enough work can be spent on writing the report and preparing the codebase for SIG evaluation.

B Software Improvement Group Code Evaluation

First Evaluation

De code van het systeem scoort bijna 4 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door lagere scores voor Unit Interfacing en Unit Size.

Voor Unit Interfacing wordt er gekeken naar het percentage code in units met een bovengemiddeld aantal parameters. Doorgaans duidt een bovengemiddeld aantal parameters op een gebrek aan abstractie. Daarnaast leidt een groot aantal parameters nogal eens tot verwarring in het aanroepen van de methode en in de meeste gevallen ook tot langere en complexere methoden.

In jullie code valt op dat jullie een aantal implementaties van `supports_methods()` hebben die allemaal 5 parameters nodig hebben, die in de meeste gevallen helemaal niet gebruikt worden. Je zou dit op een aantal manieren kunnen aanpakken, bijvoorbeeld: - introduceer een parameter-object die duidelijk maakt dat deze velden bij elkaar horen - zoek de kleinste deler voor parameters die nodig zijn, en verwijder de overige parameters - een logger geef je meestal niet als parameter mee, daar zou je eerder een constante verwachten

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt. Binnen de langere methodes zijn meestal aparte stukken functionaliteit te vinden welke ge-refactored kunnen worden naar aparte methodes.

Jullie scoren hier al redelijk goed, maar er zijn nog een paar punten waarop verdere verbetering mogelijk is. Kijk bijvoorbeeld eens naar `print_graph()` in `visualize.py` of `get_resource_usage()` in `aggregator.py`. Deze methodes bestaan eigenlijk uit een aantal deelmethodes die verschillende dingen doen. Het is aan te raden kritisch te kijken naar de langere methodes binnen dit systeem en deze waar mogelijk op te splitsen.

Over het algemeen scoort de code dus bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase.

De aanwezigheid van test-code is ook veelbelovend, hopelijk zal het volume van de test-code ook groeien op het moment dat er nieuwe functionaliteit toegevoegd wordt.

– Dennis Bijlsma, Software Improvement Group

Response

We see and appreciate the concerns for the Unit Size issues in the resource usage aggregator. The function that aggregates all resources is one big chunk of code, that could be easily split up in handling different aspects of the aggregation, for different metrics. As a result of refactoring this code, we also deduplicated and rewrote a large chunk of the surrounding code in the resource usage aggregator, so that everything factored together a lot more nicely.

Furthermore, we also split up the other large function in the code base, `orchestrator.tasks.run_benchmark()`, to separate the running of the workload and result processing. Using Pylint and SonarQube, we were not able to find any other significantly large functions in the code base.

For Unit Interfacing, we were somewhat skeptical. The given example does not receive five parameters, but four: in Python, `self` or `cls` is a mandatory method parameter that would normally be implicit in other languages, like `this` in Java or C++. Nevertheless, we see the concern for the other four parameters, and refactored the orchestrator to collect the provisioning, configuration and deployment method into a `orchestrator.strategy.OrchestrationStrategy` object. This could be simply passed around, representing the orchestration strategy for a given run.

We searched for other instances of methods with a large amount of parameters, and found one stand-out: `orchestration.provisioning.Provisioner.supports_machine(cls, logger, provider, instance, region)`. However, seeing as provider, instance and region are later grouped together into a `models.Run` object and this is the only method call where they are separated, we did not consider it worth refactoring for.

Second Evaluation

In de tweede upload zien we dat zowel de omvang van het systeem als de score voor onderhoudbaarheid ongeveer gelijk zijn gebleven. Jullie zitten dus nog steeds op 4 sterren.

Bij de deelscores voor Unit Size en Unit Interfacing, die tijdens de analyse van de eerste upload als verbeterpunt zijn aangemerkt, zien we dat jullie de door ons genoemde voorbeelden hebben verbeterd. De stijging in score is echter niet heel groot, aangezien jullie deze refactorings niet structureel hebben doorgevoerd.

Een positief punt is dat jullie gedupliceerde code hebben opgeruimd, terwijl we dit bij de eerste upload niet als verbeterpunt hadden genoemd. Jullie hebben dit dus zelf gevonden en verbeterd.

Tot slot is het goed om te zien dat de kleine stijging in de hoeveelheid productiecode samen gaat met een bijbehorende stijging in de testcode.

Uit deze observaties kunnen we concluderen dat de aanbevelingen van de vorige evaluatie zijn meegenomen in het ontwikkeltraject.

– Dennis Bijlsma, Software Improvement Group

C

Diagrams

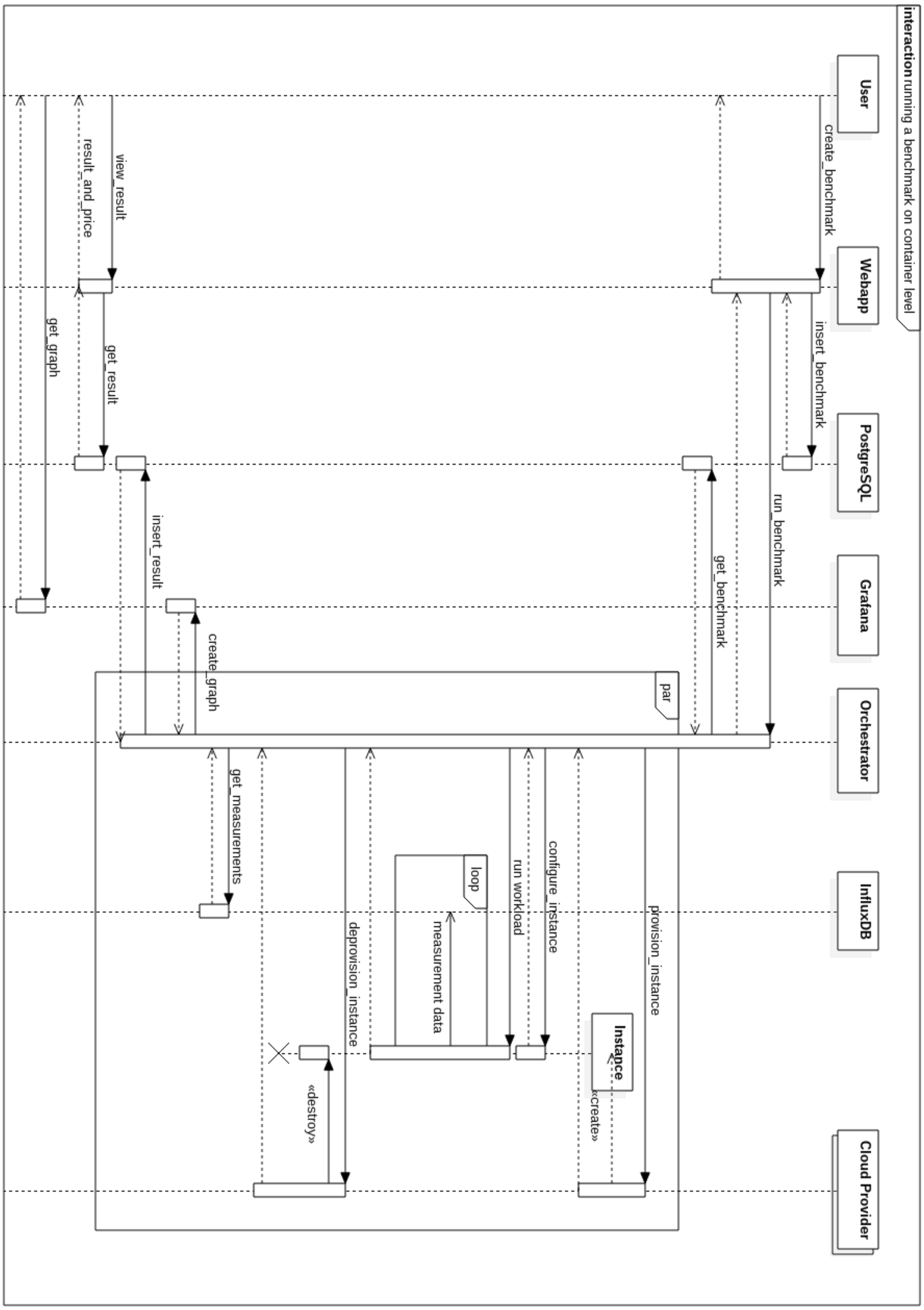


Figure C.1: Sequence diagram showing the running of a benchmark.

D
Raw Measurement Data

Table D.1: Original measurement results for manual provisioning.

		Manual				
Workload		Run 1	Run 2	Run 3	Average	Standard Deviation
Prime	Runtime (s)	24.0	24.0	24.0	24.0	0.0
	Max memory used (Byte)	381648896.0	382595072.0	380129280.0	381457749.3	1243959.5
	Network out (Byte)	211926.0	201367.0	198464.0	203919.0	7084.6
	Network in (Byte)	5109.0	5001.0	5001.0	5037.0	62.4
	Disk read operations	8.0	0.0	0.0	2.7	4.6
	Disk write operations	38.0	1940.0	2072.0	1350.0	1138.1
	Disk read (Byte)	73728.0	0.0	0.0	24576.0	42566.9
	Disk write (Byte)	6987776.0	10633216.0	11329536.0	9650176.0	2331843.8
	Total cpu (jiffies)	18247.0	17584.0	17575.0	17802.0	385.4
Sleep	Runtime (s)	60.0	61.0	60.0	60.3	0.6
	Max memory used (Byte)	351674368.0	378482688.0	380092416.0	370083157.3	15962783.3
	Network out (Byte)	565881.0	565398.0	567114.0	566131.0	884.9
	Network in (Byte)	5250.0	5250.0	5250.0	5250.0	0.0
	Disk read operations	8.0	6.0	988.0	334.0	566.4
	Disk write operations	5282.0	3452.0	3680.0	4138.0	997.3
	Disk read (Byte)	32768.0	24576.0	4046848.0	1368064.0	2319898.6
	Disk write (Byte)	135913472.0	28860416.0	19570688.0	61448192.0	64655883.0
	Total cpu (jiffies)	69.0	75.0	94.0	79.3	13.1
Sleep 180	Runtime (s)	181.0	180.0	181.0	180.7	0.6
	Max memory used (Byte)	352600064.0	384507904.0	380694528.0	372600832.0	17425799.8
	Network out (Byte)	1834759.0	1715174.0	1715206.0	1755046.3	69033.2
	Network in (Byte)	5874.0	5977.0	5790.0	5880.3	93.7
	Disk read operations	16.0	0.0	12.0	9.3	8.3
	Disk write operations	12814.0	8434.0	10056.0	10434.7	2214.4
	Disk read (Byte)	65536.0	0.0	49152.0	38229.3	34106.0
	Disk write (Byte)	71106560.0	47300608.0	56008704.0	58138624.0	12045051.1
	Total cpu (jiffies)	240.0	67.0	92.0	133.0	93.5

Table D.2: Original measurement results for overhead using custom provisioning.

		Custom				
Workload		Run 1	Run 2	Run 3	Average	Standard Deviation
Prime	Runtime (s)	26.0	26.0	25.0	25.7	0.6
	Max memory used (Byte)	391802880.0	390434816.0	397836288.0	393357994.7	3938176.4
	Network out (Byte)	257392.0	267843.0	254324.0	259853.0	7087.5
	Network in (Byte)	4959.0	5109.0	5043.0	5037.0	75.2
	Disk read operations	10.0	12.0	4.0	8.7	4.2
	Disk write operations	2040.0	1294.0	1980.0	1771.3	414.5
	Disk read (Byte)	40960.0	49152.0	16384.0	35498.7	17053.0
	Disk write (Byte)	569024512.0	425041920.0	549019648.0	514362026.7	77997498.3
	Total cpu (jiffies)	19299.0	19710.0	18750.0	19253.0	481.7
Sleep	Runtime (s)	61.0	65.0	64.0	63.3	2.1
	Max memory used (Byte)	370278400.0	379322368.0	378937344.0	376179370.7	5114015.2
	Network out (Byte)	578465.0	610615.0	654770.0	614616.7	38309.6
	Network in (Byte)	5476.0	5823.0	5715.0	5671.3	177.6
	Disk read operations	2.0	0.0	2.0	1.3	1.2
	Disk write operations	2486.0	4466.0	3844.0	3598.7	1012.5
	Disk read (Byte)	8192.0	0.0	8192.0	5461.3	4729.7
	Disk write (Byte)	23846912.0	230563840.0	230621184.0	161677312.0	119364631.3
	Total cpu (jiffies)	98.0	124.0	139.0	120.3	20.7
Sleep 180	Runtime (s)	181.0	180.0	181.0	180.7	0.6
	Max memory used (Byte)	383373312.0	380801024.0	381136896.0	381770410.7	1398274.7
	Network out (Byte)	1730345.0	1723110.0	1713206.0	1722220.3	8604.1
	Network in (Byte)	5684.0	5790.0	5826.0	5766.7	73.8
	Disk read operations	12.0	10.0	0.0	7.3	6.4
	Disk write operations	14254.0	14704.0	16008.0	14988.7	911.0
	Disk read (Byte)	49152.0	40960.0	0.0	30037.3	26333.6
	Disk write (Byte)	284975104.0	285442048.0	293216256.0	287877802.7	4629127.6
	Total cpu (jiffies)	160.0	257.0	259.0	225.3	56.6

Table D.3: Original measurement results for overhead using Docker Machine provisioning.

		Docker Machine				
Workload		Run 1	Run 2	Run 3	Average	Standard Deviation
Prime	Runtime (s)	24.0	24.0	25.0	24.3	0.6
	Max memory used (Byte)	382291968.0	380932096.0	383533056.0	382252373.3	1300932.0
	Network out (Byte)	243928.0	250269.0	247875.0	247357.3	3202.0
	Network in (Byte)	4632.0	5127.0	5277.0	5012.0	337.5
	Disk read operations	0.0	0.0	2.0	0.7	1.2
	Disk write operations	2614.0	54.0	70.0	912.7	1473.4
	Disk read (Byte)	0.0	0.0	8192.0	2730.7	4729.7
	Disk write (Byte)	455221248.0	1138688.0	7725056.0	154694997.3	260284201.5
	Total cpu (jiffies)	17906.0	18175.0	18323.0	18134.7	211.4
Sleep	Runtime (s)	66.0	66.0	66.0	66.0	0.0
	Max memory used (Byte)	349863936.0	351350784.0	356872192.0	352695637.3	3692612.3
	Network out (Byte)	620124.0	619192.0	620617.0	619977.7	723.7
	Network in (Byte)	5923.0	5865.0	5548.0	5778.7	201.9
	Disk read operations	6.0	0.0	6.0	4.0	3.5
	Disk write operations	7118.0	5774.0	4478.0	5790.0	1320.1
	Disk read (Byte)	24576.0	0.0	24576.0	16384.0	14189.0
	Disk write (Byte)	232316928.0	222756864.0	223395840.0	226156544.0	5344606.7
	Total cpu (jiffies)	206.0	165.0	136.0	169.0	35.2
Sleep 180	Runtime (s)	181.0	182.0	182.0	181.7	0.6
	Max memory used (Byte)	352534528.0	352063488.0	352522240.0	352373418.7	268478.1
	Network out (Byte)	1788944.0	1729409.0	1848805.0	1789052.7	59698.1
	Network in (Byte)	6034.0	5475.0	6089.0	5866.0	339.7
	Disk read operations	12.0	4.0	10.0	8.7	4.2
	Disk write operations	14212.0	13616.0	12326.0	13384.7	964.0
	Disk read (Byte)	49152.0	16384.0	40960.0	35498.7	17053.0
	Disk write (Byte)	273522688.0	269467648.0	258949120.0	267313152.0	7521875.5
	Total cpu (jiffies)	336.0	293.0	238.0	289.0	49.1

Table D.4: Re-run measurement results for manual provisioning.

		Manual				
		Run 1	Run 2	Run 3	Average	Standard Deviation
Prime	Runtime (s)	25	27	28	26.67	1.53
	Max memory used (Byte)	382562304	380743680	383254528	382186837.33	1296850.35
	Network out (Byte)	226929	228783	242007	232573.00	8222.51
	Network in (Byte)	5109	3898	5001	4669.33	670.17
	Disk read operations	2	4	0	2.00	2.00
	Disk write operations	5386	2396	1524	3102.00	2025.48
	Disk read (Byte)	8192	65536	0	24576.00	35708.10
	Disk write (Byte)	306143232	22036480	11001856	113060522.67	167305529.54
	Total CPU (jiffies)	19172	19975	20778	19975.00	803.00
Sleep	Runtime (s)	60	61	60	60.33	0.58
	Max memory used (Byte)	381612032	381730816	382836736	382059861.33	675409.58
	Network out (Byte)	563308	562859	563273	563146.67	249.74
	Network in (Byte)	5358	5292	5250	5300.00	54.44
	Disk read operations	2	2	8	4.00	3.46
	Disk write operations	4676	3638	2494	3602.67	1091.43
	Disk read (Byte)	8192	8192	32768	16384.00	14188.96
	Disk write (Byte)	25444352	22167552	16154624	21255509.33	4711541.99
	Total CPU (jiffies)	77	106	91	91.33	14.50
Sleep 180	Runtime (s)	181	180	180	180.33	0.58
	Max memory used (Byte)	383184896	379015168	378597376	380265813.33	2536615.91
	Network out (Byte)	1706519	1703278	1708055	1705950.67	2438.68
	Network in (Byte)	5790	5898	5958	5882.00	85.14
	Disk read operations	14	6	12	10.67	4.16
	Disk write operations	10218	16358	12156	12910.67	3138.80
	Disk read (Byte)	65536	24576	49152	46421.33	20616.08
	Disk write (Byte)	54648832	88670208	66789376	70036138.67	17241508.26
	Total CPU (jiffies)	247	124	238	203.00	68.56

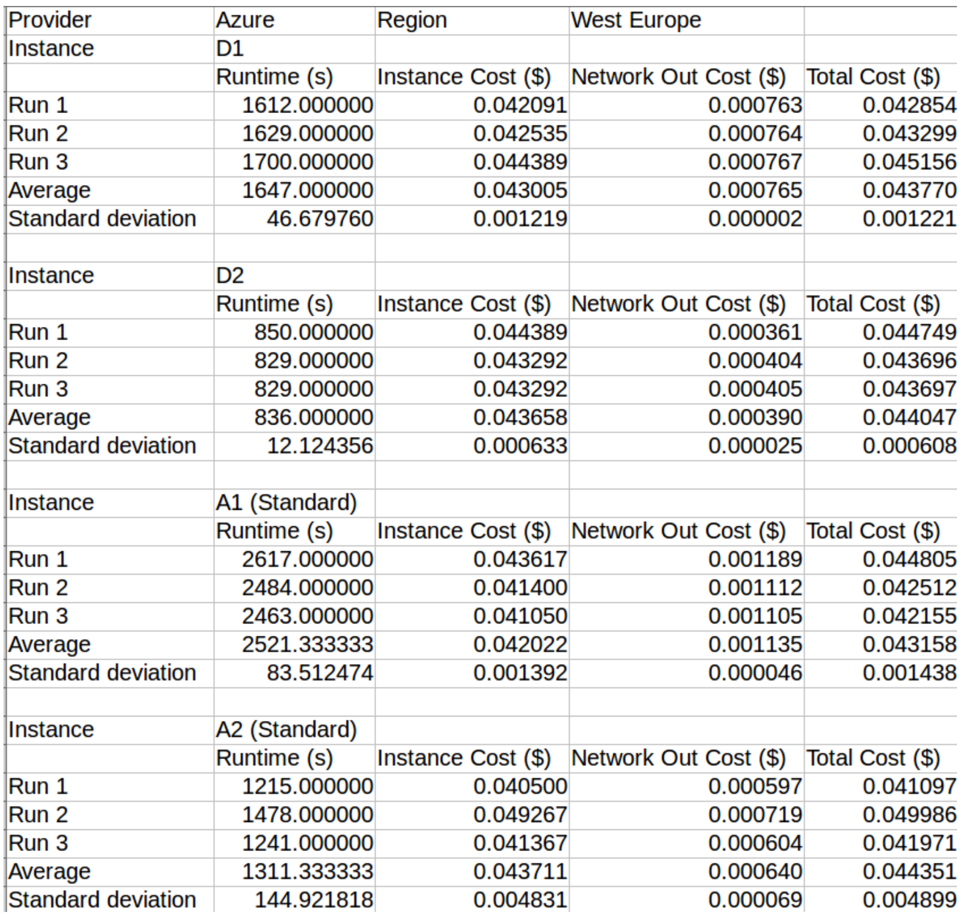
Table D.5: Re-run measurement results for overhead using custom provisioning.

		Custom				
		Run 1	Run 2	Run 3	Average	Standard Deviation
Prime	Runtime (s)	23	24	24	23.67	0.6
	Max memory used (Byte)	380043264	393572352	393318400	388978005.33	7738754.74
	Network out (Byte)	206645	210860	223056	213520.33	8522.81
	Network in (Byte)	5109	4959	5001	5023.00	77.38
	Disk read operations	0	8	0	2.67	4.62
	Disk write operations	1520	1568	1666	1584.67	74.41
	Disk read (Byte)	0	32768	0	10922.67	18918.61
	Disk write (Byte)	11239424	420798464	430653440	287563776.00	239354633.90
	Total CPU (jiffies)	17562	17910	18302	17924.67	370.22
Sleep	Runtime (s)	60	60	60	60.00	0.00
	Max memory used (Byte)	356556800	358572032	354922496	356683776.00	1828078.34
	Network out (Byte)	568432	615111	569738	584427.00	26581.15
	Network in (Byte)	5108	5250	5292	5216.67	96.42
	Disk read operations	2	0	2	1.33	1.15
	Disk write operations	4042	3454	4878	4124.67	715.59
	Disk read (Byte)	8192	0	8192	5461.33	4729.65
	Disk write (Byte)	214130688	214228992	221347840	216569173.33	4138738.61
	Total CPU (jiffies)	53	80	99	77.33	23.12
Sleep 180	Runtime (s)	180	180	180	180.00	0.00
	Max memory used (Byte)	357748736	360198144	358707200	358884693.33	1234312.68
	Network out (Byte)	1703127	1711976	1708582	1707895.00	4464.32
	Network in (Byte)	5856	6106	5880	5947.33	137.93
	Disk read operations	6	10	16	10.67	5.03
	Disk write operations	8620	14886	11530	11678.67	3135.64
	Disk read (Byte)	24576	40960	65536	43690.67	20616.08
	Disk write (Byte)	235528192	284975104	270917632	263806976.00	25478821.56
	Total CPU (jiffies)	180	254	296	243.33	58.73

Table D.6: Re-run measurement results for overhead using Docker Machine provisioning.

		Docker-Machine				
		Run 1	Run 2	Run 3	Average	Standard Deviation
Prime	Runtime (s)	23	23	24	23.33	0.58
	Max memory used (Byte)	391249920	396562432	396431360	394747904.00	3030051.82
	Network out (Byte)	217702	221498	224284	221161.33	3303.89
	Network in (Byte)	5079	4959	5049	5029.00	62.45
	Disk read operations	6	4	0	3.33	3.06
	Disk write operations	234	248	234	238.67	8.08
	Disk read (Byte)	24576	16384	0	13653.33	12513.49
	Disk write (Byte)	4128768	10813440	4079616	6340608.00	3873664.10
	Total CPU (jiffies)	17744	17690	17764	17732.67	38.28
Sleep	Runtime (s)	61	61	61	61.00	0.00
	Max memory used (Byte)	346832896	346537984	353103872	348824917.33	3708616.06
	Network out (Byte)	569299	571177	571721	570732.33	1270.75
	Network in (Byte)	5448	5458	5250	5385.33	117.31
	Disk read operations	0	8	6	4.67	4.16
	Disk write operations	5244	3714	3586	4181.33	922.52
	Disk read (Byte)	0	32768	24576	19114.67	17053.01
	Disk write (Byte)	235642880	223297536	221519872	226820096.00	7692279.28
	Total CPU (jiffies)	50	92	59	67.00	22.11
Sleep 180	Runtime (s)	181	181	181	181.00	0.00
	Max memory used (Byte)	361402368	355627008	356671488	357900288.00	3077524.63
	Network out (Byte)	1710388	1711230	1814971	1745529.67	60139.43
	Network in (Byte)	5916	6048	5874	5946.00	90.80
	Disk read operations	12	6	8	8.67	3.06
	Disk write operations	14578	13200	11544	13107.33	1519.12
	Disk read (Byte)	49152	24576	32768	35498.67	12513.49
	Disk write (Byte)	288186368	279019520	267173888	278126592.00	10534660.38
	Total CPU (jiffies)	308	163	215	228.67	73.46

Table D.7: Measurement results for a Psipred workload on Azure.



Provider	Azure	Region	West Europe	
Instance	D1			
	Runtime (s)	Instance Cost ($)	Network Out Cost ($)	Total Cost ($)
Run 1	1612.000000	0.042091	0.000763	0.042854
Run 2	1629.000000	0.042535	0.000764	0.043299
Run 3	1700.000000	0.044389	0.000767	0.045156
Average	1647.000000	0.043005	0.000765	0.043770
Standard deviation	46.679760	0.001219	0.000002	0.001221
Instance	D2			
	Runtime (s)	Instance Cost ($)	Network Out Cost ($)	Total Cost ($)
Run 1	850.000000	0.044389	0.000361	0.044749
Run 2	829.000000	0.043292	0.000404	0.043696
Run 3	829.000000	0.043292	0.000405	0.043697
Average	836.000000	0.043658	0.000390	0.044047
Standard deviation	12.124356	0.000633	0.000025	0.000608
Instance	A1 (Standard)			
	Runtime (s)	Instance Cost ($)	Network Out Cost ($)	Total Cost ($)
Run 1	2617.000000	0.043617	0.001189	0.044805
Run 2	2484.000000	0.041400	0.001112	0.042512
Run 3	2463.000000	0.041050	0.001105	0.042155
Average	2521.333333	0.042022	0.001135	0.043158
Standard deviation	83.512474	0.001392	0.000046	0.001438
Instance	A2 (Standard)			
	Runtime (s)	Instance Cost ($)	Network Out Cost ($)	Total Cost ($)
Run 1	1215.000000	0.040500	0.000597	0.041097
Run 2	1478.000000	0.049267	0.000719	0.049986
Run 3	1241.000000	0.041367	0.000604	0.041971
Average	1311.333333	0.043711	0.000640	0.044351
Standard deviation	144.921818	0.004831	0.000069	0.004899

Table D.8: Measurement results for a Psipred workload on Google Compute Engine.

Provider	GCE	Region	europe-west1-c	
Instance	n1-highcpu-2			
	Runtime (s)	Instance Cost ($)	Network Out Cost ($)	Total Cost ($)
Run 1	866.000000	0.020207	0.000436	0.020642
Run 2	858.000000	0.020020	0.000523	0.020543
Run 3	878.000000	0.020487	0.000473	0.020960
Average	867.333333	0.020238	0.000477	0.020715
Standard deviation	10.066446	0.000235	0.000044	0.000218
Instance	n1-highcpu-4			
	Runtime (s)	Instance Cost ($)	Network Out Cost ($)	Total Cost ($)
Run 1	411.000000	0.019180	0.000273	0.019453
Run 2	428.000000	0.019973	0.000279	0.020253
Run 3	418.000000	0.019507	0.000277	0.019784
Average	419.000000	0.019553	0.000277	0.019830
Standard deviation	8.544004	0.000399	0.000003	0.000402
Instance	n1-standard-1			
	Runtime (s)	Instance Cost ($)	Network Out Cost ($)	Total Cost ($)
Run 1	1187.000000	0.018135	0.000549	0.018684
Run 2	1141.000000	0.017432	0.000549	0.017981
Run 3	1126.000000	0.017203	0.000536	0.017739
Average	1151.333333	0.017590	0.000545	0.018135
Standard deviation	31.785741	0.000486	0.000008	0.000491
Instance	n1-standard-2			
	Runtime (s)	Instance Cost ($)	Network Out Cost ($)	Total Cost ($)
Run 1	870.000000	0.026583	0.000459	0.027042
Run 2	851.000000	0.026003	0.000442	0.026445
Run 3	866.000000	0.026461	0.000439	0.026900
Average	862.333333	0.026349	0.000446	0.026796
Standard deviation	10.016653	0.000306	0.000011	0.000312

Table D.9: Measurement results for a Psipred workload on Amazon EC2.

Provider	Amazon	Region	eu-west-1	
Instance	c3.large			
	Runtime (s)	Instance Cost ($)	Network Out Cost ($)	Total Cost ($)
Run 1	819.000000	0.027300	0.000422	0.027722
Run 2	816.000000	0.027200	0.000418	0.027618
Run 3	819.000000	0.027300	0.000425	0.027725
Average	818.000000	0.027267	0.000422	0.027689
Standard deviation	1.732051	0.000058	0.000003	0.000061
Instance	c3.xlarge			
	Runtime (s)	Instance Cost ($)	Network Out Cost ($)	Total Cost ($)
Run 1	401.000000	0.026622	0.000254	0.026876
Run 2	397.000000	0.026356	0.000261	0.026618
Run 3	401.000000	0.026622	0.000255	0.026877
Average	399.666667	0.026533	0.000257	0.026790
Standard deviation	2.309401	0.000153	0.000004	0.000149
Instance	c4.large			
	Runtime (s)	Instance Cost ($)	Network Out Cost ($)	Total Cost ($)
Run 1	654.000000	0.022708	0.000296	0.023005
Run 2	655.000000	0.022743	0.000300	0.023043
Run 3	822.000000	0.028542	0.000334	0.028876
Average	710.333333	0.024664	0.000310	0.024975
Standard deviation	96.707463	0.003358	0.000021	0.003379
Instance	c4.xlarge			
	Runtime (s)	Instance Cost ($)	Network Out Cost ($)	Total Cost ($)
Run 1	321.000000	0.022381	0.000185	0.022566
Run 2	318.000000	0.022172	0.000186	0.022358
Run 3	320.000000	0.022311	0.000187	0.022498
Average	319.666667	0.022288	0.000186	0.022474
Standard deviation	1.527525	0.000107	0.000001	0.000106

Table D.10: Measurement results of workload runtime on Google Compute Engine zones.

Workload	Prime		Workload	Raytracer
Provider	GCE		Provider	GCE
Instance	n1-highcpu-2		Instance	n1-highcpu-2
Region	europe-west1-b		Region	europe-west1-b
	Runtime (s)			Runtime (s)
Run 1	100.00		Run 1	270.00
Run 2	99.00		Run 2	274.00
Run 3	99.00		Run 3	278.00
Average	99.33		Average	274.00
Standard deviation	0.58		Standard deviation	4.00
Region	europe-west1-c		Region	europe-west1-c
	Runtime (s)			Runtime (s)
Run 1	67.00		Run 1	271.00
Run 2	67.00		Run 2	270.00
Run 3	66.00		Run 3	267.00
Average	66.67		Average	269.33
Standard deviation	0.58		Standard deviation	2.08
Region	europe-west1-d		Region	europe-west1-d
	Runtime (s)			Runtime (s)
Run 1	74.00		Run 1	230.00
Run 2	60.00		Run 2	239.00
Run 3	63.00		Run 3	282.00
Average	65.67		Average	250.33
Standard deviation	7.37		Standard deviation	27.79

Table D.11: Measurement results for provisioning time.

Provider	Amazon EC2	Azure	GCE		
Zone	eu-west-1	West Europe	europe-west1-b	europe-west1-c	europe-west1-d
Provisioning Time (s)	182	701	38	46	35
	182	709	36	46	42
	187	700	47	46	35
	183	756	45	38	45
	210	704	38	40	35
	173	753	45	50	35
	168	681		47	40
	183	632		37	47
	177	718		44	
	180	674		44	
	163	675		44	
	178	805		47	
	172	682		38	
				38	
				35	
				45	
				38	
				45	
				45	
Average (s)	179.85	706.92	41.50	42.79	39.25
Standard Deviation (s)	11.31	44.14	4.68	4.29	4.98

Information Sheet

Design of a Performance Benchmarker for Fully Distributed Infrastructure-as-a-Service Clouds

A bachelor project by Nerdalize, defended on September 30th, 2015.

Plot

Nerdalize B.V. is an infrastructure-as-a-service (IaaS) cloud provider aiming to offer substantially lower prices than its competitors. In order to visualize its cost savings to customers and measure its own systems against competitors in a cloud market reigned by opaque pricing models, it would like to utilize an application benchmarker to give customers insight into the resource utilization and operation costs of their applications among various cloud providers.

We have done research into the fields of cloud computing, benchmarking and the intersection thereof, determined requirements for such a benchmarker, and assessed any existing solutions in the field. We then chose Nerdalize's internal prototype implementation as a suitable base to develop a fully featured, production-ready application benchmarker from. We identified five main design goals of correctness, robustness, security, extensibility and maintainability.

We then analyzed and prioritized potential improvements and extensions to this prototype, and implemented them in an agile-driven Extreme Programming (XP) development process. The main contributions lie in designing and implementing a fully automated test suite and system, vastly improving the accuracy and stability of the benchmarker, re-designing the deployment model from monolithic to modularized and extensible, implementing support for provisioning to arbitrary Linux-based hosts, and deployment of complex workload architectures.

We have experimentally verified the accuracy of the benchmarker, and assessed that its deployment overhead is very small to neglible. Moreover, we have pitted various cloud providers against each other in the arena, and tested them on metrics such as provisioning latency, runtime costs, CPU performance and much more.

The end result of this process is a stable, well-tested, featured benchmarker application that is used in production environments at Nerdalize.

Starring

Mark Cilissen <m.h.j.cilissen@student.tudelft.nl> as orchestration engineer, API manager and software porter.
Maarten van Elsas <m.vanelsas@student.tudelft.nl> as database engineer, resource analyst and testing manager.

With Special Appearances By

Dr. ir. Alexandru Iosup <a.iosup@tudelft.nl> as university supervisor.
Dr. ir. Martha Larson <m.a.larson@tudelft.nl> as project supervisor.
Eric Feliksik, MSc <e.feliksik@nerdalize.com> as company supervisor.
Mathijs de Meijer <m.demeijer@nerdalize.com> as company supervisor.

Coming Soon to a Repository Near You...

The final report for this project can be found at: `http://repository.tudelft.nl/`.

Bibliography

[1] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, *Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility,* Future Generation Computer Systems **25**, 599 (2009).

[2] I. Foster, Y. Zhao, I. Raicu, and S. Lu, *Cloud computing and grid computing 360-degree compared,* in *Grid Computing Environments Workshop, 2008. GCE'08* (IEEE, 2008) pp. 1–10.

[3] R. Buyya, *High Performance Cluster Computing: Architectures and Systems*, Vol. 1 (Prentice Hall, Upper SaddleRiver, NJ, USA, 1999).

[4] J. Cáceres, L. Vaquero, L. Rodero-Merino, A. Polo, and J. Hierro, *Service scalability over the cloud,* in *Handbook of Cloud Computing* (Springer US, 2010) pp. 357–377.

[5] L. Wang, G. Von Laszewski, A. Younge, X. He, M. Kunze, J. Tao, and C. Fu, *Cloud computing: A perspective study,* New Generation Computing **28**, 137 (2010).

[6] Q. Hassan, *Demystifying cloud computing,* CrossTalk, The Journal of Defense Software Engineering **24**, 16 (2011).

[7] P. M. Mell and T. Grance, *SP 800-145. The NIST Definition of Cloud Computing*, Tech. Rep. (Gaithersburg, MD, United States, 2011).

[8] A. Iosup, R. Prodan, and D. Epema, *IaaS cloud benchmarking: Approaches, challenges, and experience,* in *Cloud Computing for Data-Intensive Applications* (Springer New York, 2014) pp. 83–104.

[9] E. Folkerts, A. Alexandrov, K. Sachs, A. Iosup, V. Markl, and C. Tosun, *Benchmarking in the cloud: What it should, can, and cannot be,* in *Selected Topics in Performance Evaluation and Benchmarking*, Lecture Notes in Computer Science, Vol. 7755 (Springer Berlin Heidelberg, 2013) pp. 173–188.

[10] J. Gray, *Benchmark Handbook: For Database and Transaction Processing Systems* (Morgan Kaufmann Publishers Inc., 1992).

[11] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. Epema, *Performance analysis of cloud computing services for many-tasks scientific computing,* Parallel and Distributed Systems, IEEE Transactions on **22**, 931 (2011).

[12] X. Zhang, *Application-specific benchmarking*, Ph.D. thesis, Harvard University Cambridge, Massachusetts (2001).

[13] S. Sharkawi, D. DeSota, R. Panda, R. Indukuru, S. Stevens, V. Taylor, and X. Wu, *Performance projection of HPC applications using SPEC CFP2006 benchmarks,* in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (2009) pp. 1–12.

[14] A. Iosup, N. Yigitbasi, and D. Epema, *On the performance variability of production cloud services,* in *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on* (IEEE, 2011) pp. 104–113.

[15] K. Hwang, X. Bai, Y. Shi, M. Li, W. Chen, and Y. Wu, *Cloud performance modeling and benchmark evaluation of elastic scaling strategies,* IEEE Transactions on Parallel and Distributed Systems (2015), 10.1109/TPDS.2015.2398438.

[16] D. Kossmann, T. Kraska, and S. Loesing, *An evaluation of alternative architectures for transaction processing in the cloud,* in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (ACM, 2010) pp. 579–590.

[17] A. Li, X. Yang, S. Kandula, and M. Zhang, *CloudCmp: Comparing public cloud providers,* in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10 (2010) pp. 1–14.

[18] Google, *New open source tools for measuring cloud performance,* (retrieved 2015-07-15), `http://googlecloudplatform.blogspot.nl/2015/02/new-open-source-tools-for-measuring-cloud-performance.html`.

[19] B. Peterson, *PEP 0373 – Python 2.7 release schedule,* (retrieved on 2015-07-14), `https://www.python.org/dev/peps/pep-0373/`.

[20] B. Warsaw, *PEP 0404 – Python 2.8 un-release schedule,* (retrieved on 2015-07-14), `https://www.python.org/dev/peps/pep-0404/`.

[21] G. van Rossum, *PEP 3000 – Python 3000,* (retrieved on 2015-07-14), `https://www.python.org/dev/peps/pep-3000/`.

[22] R. T. Fielding and R. N. Taylor, *Principled design of the modern web architecture,* in *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00 (ACM, New York, NY, USA, 2000) pp. 407–416.

[23] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, 1st ed. (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992).

[24] T. Haerder and A. Reuter, *Principles of transaction-oriented database recovery,* ACM Comput. Surv. **15**, 287 (1983).

[25] *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) Base Specifications, Issue 7,* IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004) (2008), 10.1109/IEEESTD.2008.4694976.

[26] P. Srisuresh and M. Holdrege, *IP Network Address Translator (NAT) Terminology and Considerations*, RFC 2663 (1999).

[27] M. Lentczner, *Reverse HTTP*, RFC draft-lentczner-rhttp-00 (2009).

[28] I. Fette and A. Melnikov, *The Websocket Protocol*, RFC 6455 (2011).

[29] I. Paterson, D. Smith, P. Saint-Andre, J. Moffitt, L. Stout, and W. Tilanus, *Bidirectional-streams Over Synchronous HTTP (BOSH)*, XEP 0124 (2003).

[30] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Connallen, and K. A. Houston, *Object-oriented Analysis and Design with Applications, Third Edition* (Pearson Education, 2007).

[31] British Computer Society Specialist Interest Group in Software Testing, *Glossary of Software Testing Terms*, British Standard 7925-1 (1998).

[32] L. Rising and N. Janoff, *The scrum software development process for small teams,* Software, IEEE **17**, 26 (2000).

[33] Ecma International, *The JSON Data Interchange Format*, ECMA 404 (2013).

[34] Apache Foundation, *Apache Libcloud homepage,* (retrieved on 2015-09-22), `https://libcloud.apache.org/`.

[35] O. Ben-Kiki, C. Evans, and I. döt Net, *YAML Ain't Markup Language (YAML™) Version 1.2*, Tech. Rep. (2009).