Theory and practice of rapid elasticity in cloud applications

Mika Majakorpi

MSc Thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, March 1, 2013

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Mika Majakorpi

Työn nimi — Arbetets titel — Title

Theory and practice of rapid elasticity in cloud applications

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
MSc Thesis	March 1, 2013	61

Tiivistelmä — Referat — Abstract

Abstract goes here.

ACM Computing Classification System (CCS):

Networks → Cloud computing
Software and its engineering → Software performance
Computer systems organization → Reliability
General and reference → Metrics

Avainsanat — Nyckelord — Keywords
cloud computing, scalability, elasticity

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

*Todo list

1 Introduction

Computer science is a discipline built on layers upon layers of abstraction. We build entire worlds out of combinations of binary states. When complexity increases over a practical threshold, we apply another abstraction layer and continue until we face another technological or conceptual limit. Progress happens when new abstractions emerge either leveraging existing ones or replacing and simplifying them.

The context of this thesis is scalability in cloud computing, a recent abstraction built on virtualization and distributed computing [15]. Technologies related to cloud computing accelerate the provisioning of computing resources by several orders of magnitude compared to a non-virtualized process. Resources are provided to users as virtual units which draw on a pool of distributed physical resources collectively called a cloud. The lead time to acquire a virtual server instance is measured in seconds or minutes instead of days or even weeks [27]. When the server is no longer needed or during times of inactivity the resources reserved for the server are allocated to other virtual resources or released back to the cloud as free capacity. This flexibility drives down costs and provides the possibility for new kinds of agile ICT.

The apparent unlimited supply and instant delivery of resources has inspired researchers to consider cloud computing as a utility similar to water and electricity [6]; It's ubiquitously available and billed based on usage. The ease at which cloud resources can be provisioned makes it possible to run applications with an adjustable amount of server instances depending on the current or anticipated usage level of the application. This flexibility and the speed with which the deployment can be adjusted have enabled e.g. web applications to scale from a handful of concurrent sessions to millions and back without committing to a large amount of computing resources citation? which would remain deployed but unused during periods of low usage. A deployment capable of serving millions of users is understandably expensive to maintain, but the cloud approach with its prevalent pay per use pricing enables such scenarios to be realized without large upfront investment in computing resources as would be the case with dedicated hardware servers.

The ability of an application deployment on a cloud platform to change in size dynamically at runtime is referred to as elasticity or rapid elasticity [29].

This capability to automatically scale the deployment in (smaller) or out (larger) depending on current demand is a major proponent in the hype and success of cloud platforms in recent years.

citation for elasticity importance in cloud success

The goal of this thesis is to explore the theory and practice of rapid elasticity. The concept of quality of elasticity is developed and put to test using a prototype implementation of an elasticity controller, a piece of cloud infrastructure software whose responsibility is to decide on and implement cloud provisioning actions. The focus is on hybrid infrastructure as a service (IaaS) clouds and the provisioning of virtual machines in such clouds. The elasticity controller concept is a step towards a more service oriented cloud offering. Rather than provide infrastructure with an interface modeled exactly after the operations performed on IaaS VMs, a service-oriented approach aims to provide more abstract interfaces which address the cloud customer's problem domain rather than the cloud provider's. This includes e.g. cross-cloud capabilities [31].

The thesis is structured as follows. Section 2 discusses the different forms of scalability in system and software architectures. Section 3 presents cloud scalability in detail. Section 4 provides a software architecture viewpoint and introduces the notion of quality of elasticity for applications. Section **??** presents the elasticity controller in theory and discusses prerequisites, benefits and drawbacks.

A prototype elasticity controller is presented in section **??** along with test results for two scaling scenarios using it. Monitoring a cloud application externally and internally via a performance monitoring aspect (aspect oriented programming, AOP) are compared in the test scenarios. Related work is discussed in section **??** and section 7 presents conclusions.

1.1 Cloud computing terminology

At this point in time, cloud computing is often referred to quite vaguely as a massively scalable model for infrastructure services in information technology. As academic research and practical use grows, more and more terms and conceptual frameworks related to cloud computing are emerging, some of them short lived or focused on marketing. Published taxonomies [18] offer a snapshot to a fast moving target. The following terms for deployment models

and abstraction levels are fixed in common usage [29] [15] and essential to understanding the scope of cloud computing.

A cloud is *public* if it is available for the general public to access and *private* if it is only available internally to some organization or selected group of organizations. Obvious differences from a cloud user's perspective are the location of data and management of physical resources on which the virtualization environment is built. *Hybrid* clouds are a combination of the above such that a private cloud is bridged to another private or public cloud. They remain functionally independent but the private clouds gain benefits in tolerance against hardware failure and resource exhaustion as workload (i.e. virtual machines) can be shifted elsewhere in case of a shortage of capacity. Such expansion of a private cloud is called cloudbursting [**?**]. When a private cloud is the actor in cloudbursting, it is considered functionally transparent to the users of the cloud. The cloud user has a single interface towards the cloud which handles bursting behind the scenes. Bursting may also be implemented outside any cloud infrastructure layer, closer to the application. In this case bursting is typically handled by an application controller component in charge of elastic scaling (elasticity controller).

Customers can benefit from clouds at different levels of service. The simplest case for a customer is using cloud deployed *software as a service (SaaS)* without having to consider any operative aspects of the software. Google Docs is an example in this category. Google operates the service supposedly deployed on their private cloud infrastructure and customers merely log in to the service and use it over the Internet. Moving down to the next level of service, customers can deploy their own applications to *platform as a service (PaaS)* clouds like Heroku, Microsoft Azure or Google App Engine. The service provided is a platform for applications with related application programming interfaces (APIs) and services for managing and monitoring the deployment. A PaaS cloud enables customers to focus on the application instead of infrastructure at the cost of losing control and ownership of it. One further level down, *infrastructure as a service (IaaS)* clouds enable customers to provision virtualized infrastructure resources (virtual machines, storage, network) to build their own infrastructure, platform and application. IaaS gives the most control on the deployment, but requires considerably more management compared to the other service levels.

Cloud service levels form a hierarchy with infrastructure at the bottom, a platform deployed on the infrastructure and software on the platform offered as a service to customers. A new service or application may be built by leveraging any of these service levels. For example, the Heroku PaaS platform uses Amazon's EC2 infrastructure and an application deployed on Heroku will then complete the stack. On the other hand, an application could simply be deployed on EC2, skipping the PaaS layer, if it was deemed beneficial to gain additional control of the stack down to virtual infrastructure. Starting at a lower abstraction level increases the responsibilities of the application or organization operating it to include infrastructure or platform management as well as managing the application.

Ultimately all deployment models and service levels are meant to provide scalable computing resources to customers. How to best benefit from scalability, which level if and what it actually means in each case is up to the customer.

2 Scalability

Scalability is one of the elusive -ilities in information systems, a quality attribute whose importance is clear as workloads a system must handle grow to surpass trivial or initially planned quantities. Yet it is hard to pin down exactly what scalability means in each discussion of it [17].

Computing resources are limited and eventually any system which grows in data or usage will saturate the resources available to it. The system may then also end up needlessly large or expensive in case resource requirements decrease afterwards. The resources in question may be e.g. processing capacity for computationally intensive systems or storage capacity for data intensive systems. Network capacity is a notable scalability point in distributed systems. Structural scalability concerns the internal design of a system and how the design lends itself to growth or shrinking of the system's data model or, for example, its deployment.

2.1 Dimensions

has multiple dimensions. Scaling is said to be *vertical* if the scaling point or points in question are internal to some system component. For example, the amount of RAM available to a specific server or its CPU speed is a vertical scaling point in terms of that server. A vertically scaled system remains logically equivalent in the process of scaling. Scaling this way is straightforward as software requires no changes to take advantage of further resources on a system. In contrast, adding more server instances, *horizontal* scaling, is a more coarse grained operation and requires software to be written specifically to leverage the multiple servers by running tasks in parallel [7].

elastic architectures chapter? Switching to a higher level of abstraction in system design changes the viewpoint from horizontal to vertical. Horizontal scaling of nodes in a server cluster or cloud can be considered vertical scaling from the viewpoint of the utility provided (e.g. processing capacity) by it to a higher level system using it as a component. This is how infrastructure as a service (IaaS) and platform as a service (PaaS) scaling viewpoints differ. Horizontal IaaS virtual machine scaling is vertical platform capacity scaling from the viewpoint of an application deployed on a PaaS cloud where the platform manages the infrastructure.

The vertical dimension gets exponentially more expensive as the system size increases. The scaled system needs to be changed to a more capable type of server as the need for more resources increases past what the current server can physically support. This scaling path eventually leads to mainframes and supercomputers. The limits for horizontal scaling, on the other hand, are traditionally in the domain of data centers and the amount of servers that can fit on their racks. Horizontal scaling has been very common in Internet architectures since the early days of the network, but recently virtualization has made vertical scaling an important option to consider as well. System configuration can be adjusted dynamically at runtime in a matter of seconds, which is faster than the minute or two time frame of horizontal scaling in a cloud. A combination of both scaling dimensions can be used to implement a fine grained scaling solution.

Cloud computing pushes both scaling dimensions past their traditional boundaries. Hybrid clouds and cloud interoperability make it possible to

scale out a system past the boundaries of data centers and cloud providers. The network becomes the limiting factor here as the communication between nodes in a system needs to be transmitted between clouds over the Internet.

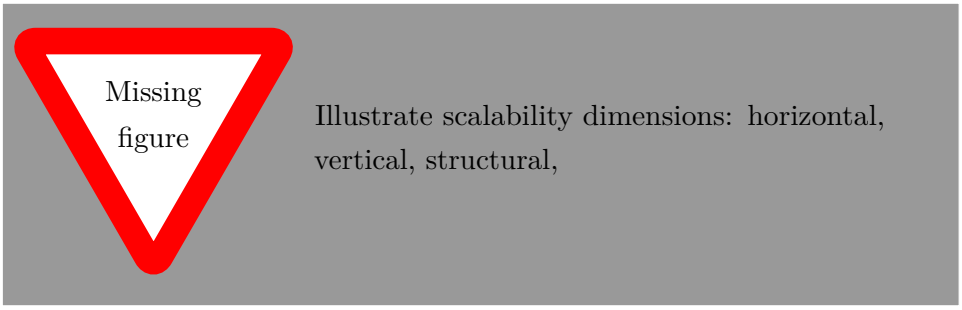
A third dimension to consider is *structural scalability* which has to do with the behavior of a piece of software as its data model, amount of data or amount of tasks to execute varies in size [7]. A requirement for scalable software is to be internally efficient in terms of the asymptotic time and space complexity of its algorithms [**?**] and additionally support parallel processing in terms of tasks and data [**?**].

Task parallelism is a feature of software systems which are capable of simultaneously executing multiple tasks on the same or different data. A serial program, in contrast, must proceed with a single task at a time. Task parallel software lends itself well to horizontal scaling as separate tasks can be executed on distributed nodes of a system. Vertical scaling, on the other hand, can be applied to adjust the performance of each task independently.

Data parallelism is the capability of a software system to perform the same operation in parallel to different instances of data. In distributed systems, a large computing task is typically split into multiple independent tasks which can be executed on separate server nodes simultaneously without communication between them. Results of the split tasks are then sent back to a controller which combines them and computes the final result. This was done in internet scale already back in 1999 with the Seti@Home project. Since then millions of ordinary computer users have donated CPU time to search for extraterrestrial intelligence by running an application which analyzes pieces of a large set of radio telescope data [24][1]. More recently Google and e.g. the open source distributed database Hadoop have made use of the MapReduce programming model for distributed data parallel computing.

mapreduce citation

Structural scalability is closely related to the horizontal scaling dimension. To take full advantage of horizontal scaling in, the application has to support parallel execution and minimize synchronization between the parallel threads of execution. Depending on the use case the parallelism can be either task or data based, but in both cases the notion of parallelism has to be built in to the application.



Illustrate scalability dimensions: horizontal, vertical, structural,

2.2 Tradeoffs

Scaling a system is not without its negatives. Vertical scaling gets expensive at an exponential rate when the system grows in available resources. Horizontal scaling increases complexity for coordination. Structural scaling requires algorithm and data model design to fit the chosen scaling mechanism.

Advances in computer science and technology have reduced the impact of these tradeoffs from what it used to be with older technology. Virtualization and dynamic provisioning of virtual machines have made it possible to use computing resources more efficiently in a modern data center. High capacity servers are not kept idle waiting for spikes in system load. Virtual resources can be allocated dynamically based on demand at any given time. Nevertheless, for vertical scaling, the cost is still the definitive tradeoff.

Scaling horizontally, a common tradeoff point is the need for communication between nodes in a distributed system. If such communication can be avoided or minimized, the software scales well; Adding servers does not cause excessive use of network bandwidth or decrease the benefit gained by adding each new server due to increased processing needed for keeping the system more complex state synchronized. For data parallel computations, the MapReducemodel enables this on a massive scale but requires algorithms to fit its mold with two distinct phases, *map* and *reduce*. The map phase distributes data to mapper nodes where it is processed. Intermediate results from the map phase are fed to reducer nodes for another round processing which ends with the final result. Both of the steps can be processed in parallel on distributed systems. The model clearly requires a specific approach to algorithm implementation in order to take advantage of parallel computation

cross ref when there is more on MapReduce in later chapter

and is only applicable to structurally similar problems which can be expressed in terms of the map and reduce functions.

Task parallel software can get congested due to synchronized access to common data. ACID (Atomicity, Consistency, Isolation, Durability) transactions are inherently serial in nature so a shared relational database, for example, quickly emerges as a bottleneck for scalability. To remedy this, databases can be scaled by applying various techniques such as sharding [**?**]. Need for ACID transactions should also be scrutinized. Many highly scalable systems make do with the BASE (Basically Available, Soft state, Eventually Consistent) consistency model in favor of the more strict ACID model.

cite BASE usage

When scaling down or in, the tradeoff is with ensuring performance and reliability while minimizing cost. When a deployment is at its minimum size, it's difficult to reliably react to increased load without false positives and try to keep the application responsive . Scaling up or out to accommodate the load will take some time, so the decision should be made early enough to keep the application responsive during the scaling activity. Ensuring performance, reliability and fault tolerance as required by e.g a service level agreement sets limits for the minimum system configuration. A capacity buffer of appropriate size has to be kept to allow time for scaling activities.

citation for difficulty of deciding when to scale up

At code level, in addition to the need for communication between horizontal nodes, tradeoffs are made between the asymptotic complexity of algorithms in terms of CPU time or data storage space needed for execution. System design principles are of key importance to minimize the impact of scalability tradeoffs.

Scalability should be considered in context. Discussing e.g. only the CPU capacity of a system is a moot point if the system becomes overly complex or expensive to maintain due to the increase in computational capacity. Designing a system with one scalability factor in mind may reduce scalability in terms of other factors. Tradeoffs like these are important to understand when designing systems. The relative importance of scalability factors can be derived from the requirements of the system in question. By going after the most important factors, the utility (change introduced by scaling which the stakeholders experience as a tangible benefit) [**?**] of the scaling effort is the highest.

Scalability can be analyzed as a multi-criteria optimization problemwhere,

given the priorities of the system in question, different scaling strategies will perform differently as the system grows. Multi-criteria analysis will help to choose the correct scaling factors from both technical and stakeholder benefit viewpoints as shown by Duboc in her work on the subject [14] Choosing the strategy with the most utility for the system's stakeholders should be the goal.

2.3 Bounds

analysis in the design phase of a system can save effort and costs during a system's lifetime as changes are easiest and cheapest to make in the beginning. Any real system will have its bounds set by its environment and stakeholders through requirements expressed in terms of functional and non-functional requirements. Some requirements are harder to meet than others and an understanding of the laws of scalability helps in managing expectations and succeeding in system implementation. This chapter presents basic laws of scalability to establish the limits within which scalability engineering takes place.

Typically a portion of any computation is not parallelizable. The size of this portion determines the lower bound in terms of execution time for a program according to Amdahl's law [**?**]. The law can be expressed as a function which gives the maximum speedup S that can be achieved with N nodes working in parallel,

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}},$$

where P is the proportion of the program that can be executed in parallel and conversely $(1-P)$ the serial proportion. As N tends to infinity, the speedup tends to $1/(1-P)$.

include a graph of Amdahl's law

For example, if a given computation has a serial part which is 10% of the complete computation, then the benefit of increasing parallelism for the remainder of the computation will tend towards zero as the number of parallel nodes increases. The upper bound for $S(N)$ in this case is 10. The computation can be sped up at most by a factor of 10 regardless of the amount of parallel processors introduced to the system. Before reaching this

theoretical limit, typically a practical limit for evenly dividing P into parallel tasks or data sets would be reached . This implies that software design level structural scalability is very important in order to keep the non-parallelizable code to a minimum.

cite practical limit of scaling in Amdahl's land

Amdahl's law underlines the importance of algorithmic optimization to maximize the speedup achievable with parallel processing. However, although the benefit of adding more parallel nodes tends to zero, the processing capacity of each of the nodes does not of course diminish in the process. In fact, the entire array of parallel nodes is idle for $1 - P$ percent of the execution. To make efficient use of a horizontally scalable system, the problem therefore needs to be of a nature which benefits from a large number of N. That is, the proportion of inherently serial code $1 - P$ needs to be minimized and the problem needs to be divisible to N or more parallel parts.

Dividing a fixed set of data or tasks can only be done in a limited number of practical ways. Amdahl's law assumes a static size and even division for data over N nodes and gives the maximum proportional speed but does not consider that more data or tasks can be processed in the same time. In practice, the benefit of parallel processing is larger given a problem with dynamic data or task set size. Having more data or tasks is key to being able to split them N ways. With virtualization, N can also be adjusted to fit the input size.

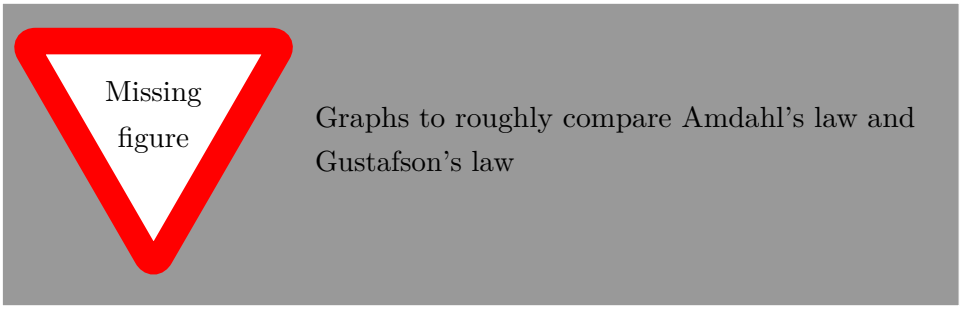
could also discuss efficiency of parallel processing, E = S/N.

Gustafson's law [**?**] shows that parallel processing is efficient given the right kind of problem. It assumes the serial fraction of computation $\alpha = 1 - P$ is static while the divisible amount of data or tasks grows evenly with the amount of parallel nodes N. The speedup according to the law is then

$$S(N) = N - \alpha(N - 1).$$

In practice α will also grow due to overhead caused by increased parallelism, but as long as the overhead is insignificant, Gustafson's law shows that scaling horizontally is efficient up to large numbers of N if the data or task size grows with the system.

In contrast to Amdah'ls law, Gustafson shows that parallel computing in a dynamic environment (data divided into parts equal in amount to that of computing nodes) scales very well.



Graphs to roughly compare Amdahl's law and Gustafson's law

Gustafson's law -> mapreduce

3 Scalability in cloud infrastructures

Public infrastructure clouds (IaaS clouds) make computing resources available to customers on a pay per use basis. Customers provision virtual servers, storage and networks from a pool of physical resources. Part of the allure of IaaS clouds is that the availability of further resources is made to seem infinite. Cloud service providers do set limits to the size of deployment under a single account, but those limits can be raised by separate agreement. The initial limits are there more to avert denial of service attacks than to safeguard against actual resource depletion.

It is apparent that clouds are massively scalable systems in terms of performance, reliability, cost, maintenance and a multitude of other quality attributes when the size of deployment of physical hardware on which the virtual resources are provisioned varies. Cloud computing takes distributed computing forward by increasing dynamism in the structure of distributed systems. Virtualization enables quick provisioning and deprovisioning of servers, storage and networks. Setting up a public cloud infrastructure for a new business can be accomplished in a matter of minutes or hours. The pay-per-use model enables quick responsiveness to change since adding servers to the environment does not come with a large up front cost. Similarly, when removing servers from the system, the released capacity will not necessarily go to waste. It becomes available to other users of the cloud.

This thesis focuses on the user level of IaaS clouds and the benefits attainable at that level for building scalable information systems. The underlying

physical implementation of a cloud and scalability therein is left mostly out of scope. Higher levels of the cloud service stack (PaaS, SaaS) are not discussed directly, but the elasticity measures presented in later chapters do apply to them as well.

In cloud context, the basic principles of scalability remain as discussed in chapter 2. Vertical scaling is achieved by adjusting the performance of existing virtual machines by changing the amount of available resources. This can imply relocating the virtual machine to a different physical host if the current host can't accommodate the scaled up VM [37]. In practice, vertical scaling is currently slower than it could be due to limitations on adjusting CPU and RAM dynamically at runtime [36]. Changing these parameters requires a restart and, for example on Amazon EC2, a newly provisioned VM instance will replace the old one. The process is heavy considering the gained benefit and as discussed above will grow exponentially expensive when resource demands increase.

Horizontal scaling is where clouds excel. Virtual machines are cloned as needed and load is balanced among them. Scaling the network, load balancers and other infrastructure tools like monitoring is needed when the system grows to surpass their capacity [36].

3.1 Rapid elasticity

Differentiate scaling and rapid elasticity "rubber band"

A cloud is said to be elastic [29] if the resources it provides can be provisioned and deprovisioned dynamically and automatically. This implies the necessity to monitor the cloud so that provisioning decisions can be made based on performance data. Provisioning must be automatic, i.e. decisions to scale out or scale in should be acted on without human intervention. This implies the need for cloud customers to access a programmatic interface with which cloud provisioning actions are carried out. The actions should resolve as fast as possible to enable constant matching of the size of deployment to service demand.

The benefit of elasticity is realized when the gap between demand and capacity can be kept as small as possible. When demand increases and more capacity is needed, rapid elasticity can enable the service to scale out quickly enough

so that no requests need to be refused. Scaling in rapidly when demand decreases means unneeded resources are kept reserved for a shorter time and consequently less money is wasted on unused capacity. The utilization rate of provisioned resources can be kept at a better level compared to a system that would prepare for demand spikes by overprovisioning resources which then end up being idle during non peak demand.

graph of demand vs capacity with steps smaller using elastic cloud vs hardware (Handbook of cloud computing)

.

3.2 Virtual machine lifecycle

Rapid elasticity is all about adjusting the size of a system by instantiating new virtual machines and terminating existing ones. This takes the VMs through a lifecycle. Optimizing this lifecycle is key to successful rapid elasticity.

The VMs go through a number of phases during the lifecycle. The high level phases from an application perspective are

- template preparation,
- instance configuration,
- instance start,
- instance contextualization,
- instance monitoring (running state) and
- instance termination.

In the template preparation phase, the virtual machine and its data is prepared up to a point from which it can be instantiated in the cloud. The template could be a basic installation of an operating system on virtual

hardware or further specialized for a specific purpose. The tradeoff between generic and specialized templates is the time it takes to configure and contextualize an instantiated generic VM for a specific purpose and, on the other hand, the effort needed to maintain specialized templates.

Instance configuration is the first phase on the way to instantiating a specific VM instance from the template. This phase may include steps like choosing the size of the VM instance i.e. how much memory and CPU capacity the instance will have. Network configuration is set at this phase as well as other virtual hardware configuration. Security settings such as SSH access keys are configured in this phase before VM is started up.

With the template chosen and configuration set, the VM instance is ready to be started. This phase is in the cloud provider's domain, but customers need to be able to monitor the progress in order to have up to date information on their deployment. Behind the scenes, the cloud provider chooses a physical server on which to allocate the VM instance and makes the necessary changes in their system to allocate portions of physical CPU, memory, storage and other resources to the VM.

When the start is done, the customer system will learn of the availability of the new instance via some reporting mechanism offered by the cloud provider. This is typically an API query over HTTP, i.e. a request-response cycle. An event mechanism whereby the cloud notifies the customer would be preferred to shorten feedback time or the need to busy loop querying the status, but scalability and security considerations on the cloud provider side may prevent such a scenario.

After starting up, the virtual machine needs to be contextualized for the dynamic runtime environment of the service it is part of. The VM could be added to a group of workers fetching work items from a queue or added to a load balanced cluster of application servers, for example. Monitoring and other infrastructure services are configured with runtime information at this point. To work around waiting time in a scenario where a controller component would connect to the new VM to perform contextualization tasks after it starts up, the virtual machine may be configured to pull its context from another server by executing a script at startup. Context may additionally be provided as a mountable block storage volume separate from the template. The Open Virtualization Format (OVF) standard advocates

the use of ISO CD images for this purpose [12]. Amazon and Eucalyptus among others provide a local network service for querying instance specific metadata over HTTP.

There has been a lot of research activity regarding the contextualization phase in the form of describing one-off solutions to accomplish a specific goal like joining instantiated VMs to a scientific computing cluster [22], standardizing an interface between VM instances and a configurator component to separate concerns of the VM internal implementation and deployment configuration by the inversion of control principle [25] and using this phase to carry out tasks related to a higher level service management approach [31] [23] [8].

After contextualization, the VM instance is in the running state. The VM carries out its tasks and reports its status as configured until, at some point in time, the VM will be shut down. The termination phase is where the VM should inform all related system components of its eventual termination so that the system as a whole can react to it by e.g. removing a load balancing setup or monitoring scope.

These phases need to be customizable so that cloud customers can add their own logic in them. Template preparation, configuration, contextualization and termination phases are the main customization points. Automation tools like Puppet [3] and Chef [2] exist to help system administrators carry out configuration tasks. Claudia [31] proposes a new abstraction layer on top of IaaS to enable more purposeful cloud service management including use of multiple cloud service providers.

3.3 Triggers and bounds - monitoring an elastic cloud deployment

Clouds have the capability to scale, but system specific logic is needed to make decisions on when and how to scale. Scaling decisions can be based on the business requirements set for the system. Good requirements are measurable and unambiguous. What is measurable depends on the monitoring capabilities of the cloud system. The monitoring subsystem needs to be customizable so that service specific metrics can be included in the data set and scaling logic. Separate monitoring tools like Ganglia [28] and Nagios [**?**] serve this purpose in hybrid or highly specialized configurations

because the cloud customer has full control over the monitoring subsystem and it can be used in private clouds as well across the whole hybrid. The tradeoff is having to maintain that part of the system as well.

Quality of monitoring data is important to make timely decisions. With large deployments, the amount of data can be large and analyzing it all can put load on the system. Data is typically aggregated from service tiers or groups of servers to reduce the amount of raw data that is to be processed by the monitoring subsystem. Another way to reduce monitoring load is to gather data at longer intervals. This quickly reduces the quality of the scaling metrics. Cloud systems aiming at just-in-time scalability already have to account for provisioning delays of tens of seconds or a few minutes. If the data on which scaling decisions are based is also a few minutes old, this makes the total reaction time sum up to e.g. 10 minutes. Balancing the monitoring overhead and scaling reaction time is an exercise needed to optimize each system.

The metrics used to make scaling decisions are typically related to performance or fault tolerance. CPU and network load and available storage capacity are straightforward metrics on a subsystem level as well as a heartbeat metric indicating the live status of each VM. System-wide and service specific metrics like requests handled per second, time spent on each service tier and the size of work queues are understandable by business stakeholders and therefore usable for concretely agreeing on and discussing system performance. Such metrics are typical for quantifying the quality of service (QoS) and are referred to in service level agreements (SLA) [4] with specific limits for the metrics that should not be crossed.

Operating a system has to be profitable or at least sustainable. Cost is often the upper bound for scaling a system in the cloud. Business stakeholders need to set limits above which the system is not allowed to scale based on cost. The lower bound is set by technical limitations of system architecture or business requirements on fault-tolerance and availability. Understanding the economics of IT systems deployed on clouds is a key success factor in the long run [33]. Cloud adoption in enterprises begun with simple cost saving goals but is moving towards enablement of lean enterprises capable of quick changes in business direction.

Find citation of shifting business goals with cloud

Clouds are a technology which levels the IT system playing field considerably

between startups and large corporations. With the pay-per-use model, large up front investments in computing infrastructure are not required to start a business, yet the scalability is available in case the service popularity explodes.

- Elasticity: when/how to scale
  - Infrastructure prerequisites
  - Lifecycle
  - Triggers
  - Business considerations
    - Elasticity window (min/max)

4 Elastic system architecture

Restructure this chapter to be about elasticity and going further into detail with concepts hierarchically below elasticity! Architecture as a subsection.

start with explanation of elasticity vs scalability

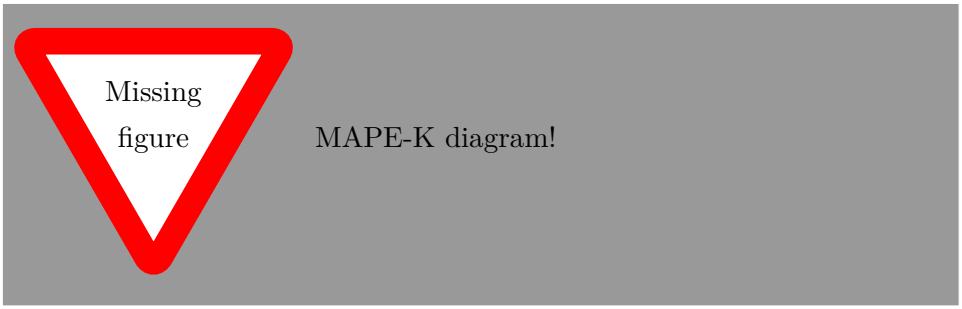
4.1 Elasticity as a controlled process

Response times to infrastructure provisioning requests in cloud services can be in the range of a few seconds. To effectively manage a system at such speeds it is essential that reactions to regularly occurring or anticipated events are built in to the system and automated.

Concepts from process control theory and autonomic computing can be applied to implement a cloud application system which knows its state and reacts to changes in it. An essential part of such a system is a controller component external to the application itself. The responsibilities of the *elasticity controller* are to monitor the system, analyse the metrics, plan corrective actions and execute them. This is known as a MAPE-K control loop [19][30] named after its phases (Monitor, Analyse, Plan, Execute, Knowledge). The knowledge in MAPE-K loops is shared data between the actual MAPE phases.

The cloud application deployment is monitored and the configuration is adjusted based on metrics reported by monitoring agents (pieces of software) attached to the application or its environment. This attachment can be

nonintrusive, where the agent is located outside the application and monitors external phenomena like network traffic or CPU load. Intrusive monitor attachment works by instrumenting the application or execution environment itself for monitoring (i.e. a Java application is instrumented using the java.lang.instrument API to monitor the internal workings of the JVM). Aspect oriented programming can be used to instrument at the application level to monitor metrics unique to the application or its business logic.



Monitoring data is analysed by the controller in the corresponding phase of the MAPE-K loop. The raw sensor data is turned into knowledge in this phase. The MAPE-K knowledge can be an advanced modeled abstraction of the system where the data is fed into or simply a group of variables reflecting the state of the monitored system now and the way it is changing over time.

Analysis of the system model may indicate that one or more criteria of acceptable system behavior are no longer met (reactive trigger) or some metric is about to exit its tolerated range (proactive trigger). Given such a situation, the controller will enter the plan phase with the purpose of creating a plan of action to bring the metric values back to or keep them in the tolerance zone. This plan can be based on a set of rules that govern the operation of the controller component or again a more elaborate model driven approach which approximates the behavior of the actual system.

The execution phase is where the controller interfaces with the application and its environment to carry out the actions decided in the planning phase. This phase relies on automation APIs available for the environment and the runtime configurability of the application.

The executed actions will cause changes in the behavior of the system which are then reported back to the controller in subsequent control loops.

4.2 Rules to satisfy requirements

The control loop needs metrics that are relevant to the system in question and bounds to specify acceptable value ranges for the metrics. Each metric and its acceptable range represent a *requirement* for the controller. Rules for controlling the system are created with the purpose of making sure the system will always meet these requirements.

Requirements expressed in terms of the implementation technology (system load, network traffic, etc.) are straightforward to set up for monitoring and further processing. If non-technical stakeholders like business decision makers are involved in the requirements elicitation, technical requirements may be difficult to communicate understandably. Therefore higher level requirements (e.g. cost per visit to a website, type of user activity, etc.) expressed in business terms may be the starting point of defining the elasticity requirements for a system.

To monitor and make scaling decisions based on metrics expressed in business terms, it is necessary to instrument the application code or monitor the state of the application's domain model (database). This kind of monitoring takes more effort compared to non-intrusive technical metrics since the monitoring has to be customized for the application. The choice of customization or relying on lower level metrics is a tradeoff one has to make when designing an elastic system.

A mapping from business requirements to technical requirements may be necessary to facilitate communication of the requirements from their source down to the implementation of the controller.

cite requirement mapping

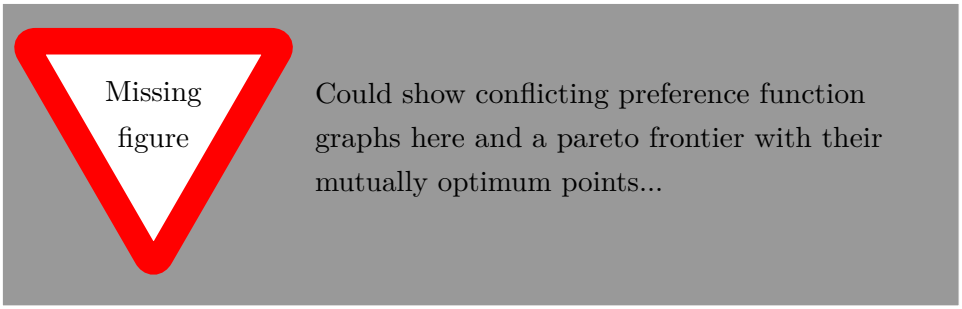
4.3 Multi-criteria decision analysis

Often the requirements given for the performance of a system conflict each other. If, for example, a system is optimized in terms of response time by adding more virtual machines to the deployment, cost rises too high to operate the system. Or if memory usage is minimized by writing data to disk, the performance may suffer to increased access time to data. The requirements may form a complex network of this kind of interdependencies. It quickly becomes difficult to specify simple rules for satisfying all the

requirements simultaneously.

Multi-criteria decision analysis is a method for finding an optimal decision considering conflicting criteria. It can be applied here to formalize the decision making under conflicting requirements.

cite multi criteria analysis



The multiple criteria are considered together by the use of a *utility function*

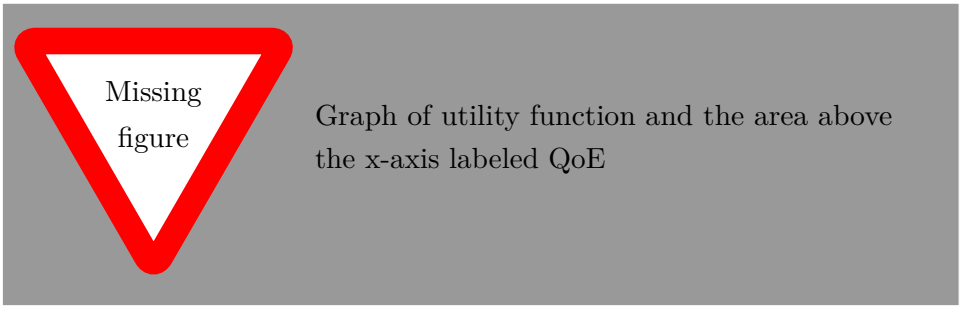
$$U(X) = \sum_{i=1}^{k} w_i P_i(X) \tag{1}$$

with a normalized range $U(X) \in [0, 1]$ in the domain of real numbers, where a value of 0 denotes the worst possible utility and 1 denotes that the system fully satisfies its combined requirements. X is a set of j parameters $\{x_1, \ldots, x_j\}$ which are needed to calculate the utility. Metric values and other knowledge of the system state are typical parameters. The utility function is a weighted sum of k *preference functions* $P_i(X)$ with $1 \leq i \leq k$. Each elasticity related requirement is defined as a preference function P_i with a normalized range $P_i(X) \in [0, 1]$, where a value of 0 denotes the worst possible preference for this requirement and 1 denotes that the requirement has been optimally fulfilled. The weights w_i represent the relative importance of each preference function to overall utility, with $\sum_{i=1}^{k} w_i = 1$.

4.4 Quality of elasticity

The utility function (1), given business-related preferences, measures the business utility of a system with regard to its performance metrics. Plotting the utility over time as the usage pattern changes shows how the system responds to these changes. A perfecty elastic system would adjust its capacity

to match or slightly surpass the required level for maximum utility. The aggregate measure of utility over time shows how well the system responds to changes, i.e. how well the system scales out and in as a response to changes in its environment.



Graph of utility function and the area above the x-axis labeled QoE

.

The *quality of elasticity* (QoE) for a system over time can be quantified as the integral of the utility function from some moment of time a to time b divided by the duration of the measurement $b - a$:

$$QoE = \frac{\int_a^b U(X)\ dx}{b-a} \tag{2}$$

The range for QoE is the same as that of the utility function, i.e. $QoE \in [0, 1]$ in the domain of real numbers.

For real systems the utility function is represented by monitored metric values gathered over time rather than a mathematical function. In this case the integral can be approximated by means of numerical analysis. The trapezoid method is used for this as the step from each data point to the next is linear and the method gives exact results in such a case .

cite trapezoid method

cite trapezoid exact for linear, calculus book?

The numerical trapezoid method version of the QoE formula is

$$\frac{\int_a^b U(X)\ dx}{b-a} \approx \frac{\frac{b-a}{2N}\sum_{k=1}^{N}(U(X_{k-1}) + U(X_k))}{b-a} = \frac{\frac{h}{2}\sum_{k=1}^{N}(U(X_{k-1}) + U(X_k))}{b-a} \tag{3}$$

where N is the amount of evenly spaced data point intervals and X_i with $i \in [0, N]$ is the set of monitored metric values for utility data point i.

The spacing of data points is denoted by $h = \frac{b-a}{N}$ which solves to 1 when the data is evenly spaced and available for each interval. Finally with all simplifications applied, quality of elasticity is calculated with the following formula:

$$QoE = \frac{\frac{1}{2}\sum_{k=1}^{N}(U(X_{k-1}) + U(X_k))}{b-a} \tag{4}$$

QoE is a measure of the quality of the elasticity controller's decision making and execution capability in the specific environment it is in. The behavior of the measured application and the strictness of elasticity related requirements given for the application influence QoE. The range of possible runtime QoE values has to be approximated or tested empirically by excercising the system in order to use QoE as a tool to reason about elastic performance. A threshold QoE value can be chosen so that whenever below that threshold, the elasticity controller will work to increase the system's utility. Normalizing the QoE value between this threshold and the maximum 1.0 then gives a "score" for the system's elasticity:

$$QoEScore = \frac{QoE - QoE_{min}}{QoE_{max} - QoE_{min}} \tag{5}$$

Table 1 lists notable factors of QoE and attributes them to system components based on the components' infuence on the factor. The factors are both technical and business related in nature and are discussed in the remainder of this chapter.

QoE Factor	**System Component**			
	Elasticity Controller	Cloud Provider	Application	Business modeling
Price - performance ratio	X	X	X	
Infrastructure pricing		X		
Billing granularity		X		
VM provisioning speed	X	X	X	
Metric data quality	X	X		
Decision making speed	X			
Correctness of scaling decisions	X			
Level of parallelization			X	
Utility preferences				X

Table 1: QoE factors and the system components which can affect them.

For good elasticity, the elasticity controller has to be fed with timely and

correct information on the status of the system. The controller's monitoring subsystem has to be able to deliver relevant metrics quickly. The data should be such that it can be reliably used to make scaling decisions. Data jitter can be a problem as well so often aggregate data is preferred so trends can be analysed. There is a clear tradeoff between how quickly the metrics show a trend change and how reliable that indication is. These issues related to *metric data quality* may need to be tuned specifically for each application.

With quality metric data, the next step is to react on results of data analysis quickly and correctly. These factors, namely *speed of decision making* and *correctness of scaling decisions* are up to the elasticity controller. Decision making speed can vary based on the the implementation of the controller. Scaling decisions can be made reactively when metrics pass their thresholds or predictively based on predictive algorithms . In simple cases the predictive algorithm could be as simple as specifying time ranges throughout the day and scaling out or in according to typical usage. Advanced algorithms do exist for more complicated usage patterns [20][34][32][9][38][13]. Predictive scaling is possible as long as there is some indicator in the metrics that can be used to decide that higher load is about to come. For web sites, sudden spikes of activity like click throughs from a social media discussion ("Slashdot effect") with a link to some normally low usage server are impossible to predict. In such a case, the performance is up to how quickly the spike of activity is identified and whether the reaction to it matches the size of the spike.

Scaling out horizontally is helpful if the application is structured to take advantage of it. The *level of parallelization* exhibited by the application and its algorithms decides whether the elastic scaling will actually help with the application's performance. The theory of scalability was discussed in detail in chapter 2.

Assuming the application is well parallelizable, the *price - performance ratio* then quantifies the performance received for a certain expenditure related to scalint out the deployment infrastructure. The ratio is mainly affected by the cloud provider as it sets the *infrastructure pricing* for its offering and specifies the kind of resources available. The elasticity controller and the application implementation also have a role considering the effective use of the infrastructure. The controller should choose the amount and

type of VM instances to fit the scenario. The elastic application should be implemented to use infrastructure resources in a way that matches the cloud provider's capabilities. Advanced optimizations in price - performance ratio may increase coupling of the application to the infrastructure specifics of the cloud provider. This can lead to increased effort required if the application is ever deployed elsewhere, i.e. cloud vendor lock-in.

Further related to price, *billing granularity* affects QoE [5][21][26][35]. The minimum price paid for a provisioned VM instance as well as the billing interval in the pay-per-use model has an effect on economical use of the infrastructure. Cost of instantiation or termination could discourage scaling out and increase the level of commitment to infrastructure. The billing interval also relates to commitment. The typical hourly billing interval with no extra cost for instantiation or termination means it is irrelevant whether an instance is running for a minute or just short of an hour. Effective use of infrastructure needs to reflect this characteristic set by the cloud provider.

VM provisioning speed is a multi-faceted QoE factor. It concerns the effectiveness of VM lifecycle phases (see chapter 3.2) from configuration through start and contextualization to termination. An effectively configured VM image will complete contextualization faster, so the choices made in the configuration phase affect provisioning speed. If a VM is provisioned with a baseline configuration of just the operating system, contextualization of the instance has to include everything from installing application dependencies to configuring them and announcing the availability of the new instance to the application infrastructure. Moving some idempotent tasks like software and operating system update installation from contextualization to configuration can decrease provisioning time as configuration is done once before and the configured VM template image is cloned for each VM instance to use.

Time spent in the starting and termination phases is up to the cloud provider as it is the provider's responsibility to reserve and release physical resources for the virtual machine. These phases are pure waiting time in terms of the elasticity controller and the application. Billing typically starts at the start of this phase, so cost of inefficiency is born by the cloud user . Similarly, billing stops at the completion of the VM stop or termination process, so again the user can only hope for fast completion.

cite billing start at instance start command receipt, not completion

Finally the ultimate bounds for QoE are set by *utility preferences*. The

preferences are modeled above as functions of metric data at a point in time. The definition of these functions determines the system's utility. The definitions of these functions need to be realistic and applicable to the deployment environment. If the definitions are unrealistic, the system may not exhibit any utility or quality of elasticity as defined here. The preferences may need to be defined together with technical and business stakeholders to arrive at workable results. Preferences typically consider cost, response time, throughput and usage of storage space, memory or other resources but could be anything that is measurable and for which measured trends can be mapped to elastic scaling operations. An elicitation method like the Architectural Tradeoff Analysis Method [10] can be used to elicit the preferences from stakeholders. The ATAM method finds quality attributes that are pivotal to the success of a system to its stakeholders, but appears to require a lot of effort if applied to its full extent.

4.5 Elastic application architecture

Successful deployment of applications and services on a cloud infrastructure requires a scalable application architecture. The cloud is not a silver bullet for scalable software. The software has to be built to take advantage of the environment.

Algorithms for parallel computation like MapReduce and architectural patterns like master/worker and cell based architecture are important building blocks of harnessing the processing power of a cloud service. The general rule is to design the architecture so as to have as few as possible shared components in a deployment. Shared components will end up being the bottlenecks when scaling out for two reasons; They get hit the hardest from multiple other system components when the system is under heavy load and they are often harder to scale themselves . Also the unavailability of a shared component can have far reaching effects on the whole application. Shared components quickly become single points of failure .

Cite shared component hard to scale

cite

Public cloud services are built on commodity hardware. The durability and serviceability of this kind of hardware is by no means in the same class as that of mainframe hardware. The way to cope with failure of relatively cheap hardware is to accept it happens and design for failure . Cloud resource

cite amazon best practices or oreilly patterns book if not an academic paper

pricing is based on low margins and high volume and best practices advocate "buying insurance" by adding redundancy at every step.

cite

Designing for failure implies automating infrastructure and application deployment and maintenance tasks as fully as possible. Automated recovery from server failures can be implemented quite simply by terminating a misbehaving server instance and provisioning another one to replace it. This is in fact one of the scenarios Amazon Web Services covers in their SLA. Notable omissions from the SLA are server instance uptime and performance. This underlines the design for failure philosophy of their cloud service offering.

cite amazon ec2 sla

Mention netflix chaos monkey wrt design for failure, maybe throw in the FAAS tech report [16]

- elasticity enables design for failure
- What kind of apps/architectures can benefit from elasticity
- Data intensive, how to benefit? (IO bound, memory bound)
- Computation intensive, how to benefit?
- Other categories? Mobile?

HASN'T THE ABOVE BEEN COVERED ALREADY?

- How does an application provide information on performance based on which it can be scaled?
  - Quality of elasticity
  - Scaling metrics
    - Requests per time period?
    - Depth of work queue?
    - Computational patterns?
    - Event broadcast mechanism?
    - Other way to relay information?
    - Aspect oriented? Cross-cutting concern
    - Latency
    - Costs (VM hours, ...)
    - SLA

5 Elastic scaling prototype setup

This chapter details the elasticity controller prototype that was implemented for this thesis as a tool to evaluate the controllability of scalable application deployment on a cloud platform. The aim of the prototype implementation was to

- find out what solutions exist to implement an elasticity controller without depending on a specific cloud infrastructure provider's elasticity services,
- explore the real world problems of making elastic scaling decisions and
- test the QoE concept in practice.

The prototype was designed to be cloud provider agnostic. The Amazon EC2 cloud platform is used here as the deployment platform, but all components are implemented using open source software with the intention of supporting any single cloud platform or multiple platforms at once. Deploying an application over multiple clouds serves to increase fault tolerance . On the other hand, not tying up the application to proprietary cloud provider APIs helps to avoid cloud vendor lock-in, which could make migration to other deployment platforms cost-prohibitive in the long run.

cite cross cloud fault tolerance

The technologies and software components used in the implementation of the controller as well as the simple controlled application and environment are presented in the following chapters.

5.1 Business application

The "business application" used for the tests is a simple Java servlet based Service Oriented Architecture (SOA) service implemented using the Spring framework, version 3.1.1. The application was originally started as a proof of concept application for an information technology consultancy company's internal capability development initiative and contains more code than is used in the context of this elasticity controller prototype test.

Upon receiving a HTTP POST request, the business service calculates a million random integers and returns the last one as HTTP response header.

The service is clearly useless for any real purpose and is simply written to represent a CPU bound highly parallelizable computation task. The application does not store user state, which makes it easy to spread the service requests among any number of servers running the same code. The deployment includes a load balancer node as a single point of entry. Requests first arrive at the load balancer which allocates them to any number of configured application servers using a simple round robin algorithm.

The application is deployed on the Amazon Elastic Compute Cloud (EC2) IaaS platform. All server nodes run the Ubuntu Linux operating system. The minimal configuration consists of one application server node (jetty) and a load balancer node (nginx). Further details of system components and their versions is given in table 2.

Scaling out for this application means

- provisioning more application server instances,
- contextualizing the instances to work as part of the application infrastructure and
- changing the infrastructure configuration to account for the newly provisioned instances.

Figure 1 shows the deployment diagram for the application along with elasticity controller components which will be discussed in further chapters.

The elasticity requirements for this application are related to response time and cost. These requirements conflict as reducing response time by adding more server instances will increase the cost.

The response time requirement states that the application must respond to requests within $r_{min} = 0.8$ to $r_{max} = 1.5$ seconds. The response time preference decreases linearly from the maximum value 1.0 down to zero as the response time grows from 0.8 to 1.5 seconds. The metrics used to calculate this preference are response time r and response time slope k. The response time preference function is defined as

$$P_r(r,k) = \begin{cases} 1, & \text{if } r < r_{min}. \\ 0, & \text{if } r > r_{max} \text{ or } k > k_{spike}. \\ \frac{r - r_{min}}{r_{max} - r_{min}}, & \text{otherwise.} \end{cases} \tag{6}$$

where k_{spike} is a threshold identifying a *spike*, a sudden increase in requests sent to the system. The spike threshold is defined as $200\frac{ms}{min}$, i.e. the system is in a spike situation if response time has increased increased 200 milliseconds per minute. The metric resolution here is 5 minutes.

The slope is defined as

$$k = \frac{r - r_5}{5} \tag{7}$$

where r_5 is the response time 5 minutes prior to the current time.

The response time is measured at the application server using an instrumented wrapper for the standard Jetty request handler. Network latencies are therefore unaccounted for.

The cost requirement is specified based on the cost of each service request. As a simplification, each VM instance is considered to cost 1 currency units per hour. A maximum cost of 20 currency units is given as the upper bound to limit scaling out indefinitely. The cost preference function is defined as a normalized linear mapping in the range $[0 \dots 1]$ from a minimum cost per request c_{min} to a maximum cost per request c_{max} of 1.0, i.e.

$$P_c(e,v,k,q,t_{vm}) = \begin{cases} 1, & \text{if } c < c_{min} \text{ or } v \leq 1 \text{ or } k > 50 \text{ or } q > 0. \\ 0, & \text{if } c > c_{max} \text{ and } v > 1. \\ \frac{c - c_{min}}{c_{max} - c_{min}}, & \text{otherwise.} \end{cases} \tag{8}$$

where e, v, k, q and t_{vm} denote current arrival rate of requests, amount of application server VM instances currently in use, response time slope, average queue length at the application servers and maximum average server throughput (requests per second per VM instance) measured when the average response time is less than r_{min}, respectively. c_{min} is further defined

as the cost per VM instance c_{vm} divided by t_{vm} , i.e.

$$c_{min} = \frac{c_{vm}}{t_{vm}} \tag{9}$$

where t_{vm} is measured only when $r < r_{min}$.

can the throughput metric be simplified, the response time rule does not make sense?

The application running on *m1.small* instances on Amazon EC2 will typically handle approximately 3 requests per second with a response time less than the r_{min} of 0.8 seconds. This makes c_{min} roughly equal to 0.3 currency units.

5.2 Elasticity controller

The prototype elasticity controller is distributed between multiple server nodes in the environment. The controller exists to answer three basic questions on elastic scaling, namely

- *when* to scale,
- *how much* and *in which direction* to scale and
- *how* to scale.

These questions map to the MAPE-K phases of analysis, planning and execution. Here the deployment is described in terms of the MAPE-K (see chapter 4.1).

Monitoring is implemented using the Ganglia monitoring system [28]. The application servers and the load balancer run the Ganglia monitoring daemon (*gmond*), which sends metric data to a separate monitoring server. The Ganglia monitoring server runs both the *gmond* and *gmetad* daemons together aggregate and store metric data from all monitored servers.

The *analysis* and *planning* phases are the responsibility of an elasticity controller application written specifically for this prototype. The controller queries the Ganglia server for system state metrics every 20 seconds and calculates the utility based on the preference functions defined for the business application. The controller first analyses the system utility. If utility is below a threshold of 0.5 , a decision is made to carry out a scaling operations. The controller then proceeds to the planning phase.

Remember to update final utility scaling threshold!

For the purpose of making a plan to scale the system either out or in, the controller application uses a scaling function $S(X)$ to indicate the needed direction and distance from optimum utility. The range of $S(X)$ is $[-1 \ldots 1]$. A value of -1 represents the maximum impulse to scale in while a value of 1 represents the maximum impulse to scale out. To factor in the direction of scaling, the preference functions are divided into two groups based on requiring either scaling out or scaling in to improve their value. The scaling function then takes the form

$$S(X) = \sum_{i=1}^{j} w_i^{in} P_i^{in}(X) + \sum_{i=1}^{k} w_i^{out} P_i^{out}(X). \tag{10}$$

where P_i^{in} denotes a preference which requires scaling in and P_i^{out} denotes a preference which requires scaling out to improve. The weights w_i^{in} and w_i^{out} are distributed among these two groups so that $\sum_{i=1}^{j} w_i^{in} = 1$ and $\sum_{i=1}^{k} w_i^{out} = 1$. In practice the prototype used here has one preference (P_r) for scaling out and one (P_c) for scaling in, so both their weights are 1.0.

In addition to the scaling function, the elasticity controller detects trends in response time by keeping track of the request response time in a 5-minute moving window. The slope k_r of response time is calculated based on the delta of the metric value 5 minutes ago and currently. If response time has grown more than 200 ms per minute in the last 5 minutes, i.e. $k_r > 200 \frac{ms}{min}$, the trend is considered a *spike*, a sudden large increase in activity.

With this information, the elasticity controller creates a scaling plan. Scaling is done in the direction indicated by the scaling function $S(X)$ and the amount of server instances to add or remove depends on the slope. With gradual growth, the amount of application server nodes is increased by half. If the response time slope indicates a spike is underway, the server instance count is multiplied by three.

For scaling in, the amount is decided with the use of the cost preference function. The cost preference is calculated with decreasing instance count until it reaches the maximum preference value. Any instances above the count which yields maximum cost preference are terminated.

With the above definition of the scaling function $S(X)$, it is possible that positive scale out preferences and negative scale in preferences cancel each

other out or interfere with each other at a time when the metrics clearly indicate scaling is needed in a specific direction. For this reason, the cost preference is considered to be at its maximum in the following situations:

- If the slope is larger than $50\frac{ms}{min}$. This indicates an increasing load trend.
- If there are requests queued up at the application server(s), i.e. all available processing threads are processing a request and more requests are waiting to be processed.
- If there is only one application server.

Similarly the response time function has one special case when it is set to minimum preference. This is when the response time slope indicates a spike. The preference function then indicates the need to scale out regardless of the current response time value.

The controller application delegates *execution* of the plan to an infrastructure automation tool called Chef. Chef is a cloud age tool which enables an *infrastructure as code* approach to IT operations. All installation and configuration is done with scripts written in Ruby. In Chef terminology, these are referred to as recipes within cookbooks. Chef manages the state of the deployment on a separate server which is a part of the infrastructure. Each server node managed by Chef runs the chef client daemon which periodically connects to the Chef server and configures the node to match the state received from the Chef server.

The MAPE-K *knowledge* is a shared concept between the controller application, Ganglia and the Chef server. The controller keeps track of trend averages of metrics it fetches from Ganglia and system contextualization is done based on system state information managed by Chef. The load balancer server runs *chef-client* every 30 seconds. This is how knowledge of application servers is updated at the load balancer. Similarly the address of the monitoring server is read from Chef every time a new node is configured (and periodically during the node's lifetime). Any change in system configuration due to scaling operations gets updated to live nodes in this way.

Although the monitoring system reports new values every 15 seconds, the moving average values for metrics take time to reflect a change in system

performance. Response time, for example, is measured as a biased histogram of 1028 data points. The histogram sample uses a forward-decaying algorithm which favors data from the past 5 minutes [11][**?**]. This metric type is good at evening out jitter in the values but takes time to react to significant changes as well. To allow time for the metrics to stabilize after a provisioning operation, the system has a 3-minute quiet period after each provisioning completes. No new provisioning operations are started during this time. This gives time for the metrics to stabilize for a new amount of virtual machines and e.g. data to start actually arriving from newly provisioned VMs, but also means the system is not responsive to utility changes during this time. Responsiveness is further hindered by the time it takes to provision instances, as the total quiet period consists of first waiting for new instances and then waiting for metrics to accommodate for them.

Components in the prototype environment		
Component	**Version**	**Purpose**
Jetty	8.1.7.v20120910	Web server & Java servlet container
Nginx	1.1.19	Load balancer & HTTP server
Chef	10.18.2	Infrastructure automation tool
Gatling	1.4.1	Load testing tool
Ganglia	3.4.0	Monitoring system
Ganglia Web	3.5.2	Monitoring dashboard user interface
Codahale Metrics	3.0.0-SNAPSHOT	Java library for collecting metrics from application server and application
Ubuntu Linux	12.04.1 LTS	Operating System on all nodes.

Table 2: Server applications and tools used in the prototype.

6 Test results

This chapter discusses the performance of the elasticity controller prototype and the test application in terms of utility preferences and the overall quality of elasticity for application deployment described in chapter 5. Two different testing scenarios are presented along with results of how the controller responded to the request load generated by the scenarios. Quality of elasticity is evaluated as a conclusion to the chapter.

6.1 Test scenarios

The elasticity controller was tested under two scenarios. Scenario 1 subjected the system to a gradually growing rate of requests while scenario 2 generated a sudden spike of requests. The scenarios were implemented using the Gatling load testing tool [**?**].

Gatling simulates users making requests to the application. Each simulated user sends equal requests, i.e. only one request type is used. The request handler calculates a million random integers and returns the last one. After receiving a response, the user waits for a random amount of time between 500 and 800 milliseconds before sending another request. This is repeated for one hour. The actual time depends on response times and ramp down periods.

The amount of users varies throughout the scenarios. The gradual growth scenario starts with six users and adds two more every 10 minutes. All users finish after one hour. The scenario steps with their timings for scenario 1 are:

- **00:00** - 6 users ramp up in 30 seconds.
- **00:10** - 2 users ramp up in 1 second.
- **00:20** - 2 users ramp up in 1 second.
- **00:30** - 2 users ramp up in 1 second.
- **00:40** - 2 users ramp up in 1 second.
- **00:50** - 2 users ramp up in 1 second.
- **01:00** - 10 users ramp down in 1 second. The first 6 users ramp down in 30 seconds.
- **01:00:30** - Finished ramping down users.

The spike scenario starts with three users for five minutes, then gradually adds six more users during a period of ten minutes and finally starts the spike of 30 users at 00:25. The spike nearly triples the user amount during one minute. The spike lasts for 20 minutes until 00:45. At 00:55 The six

user wave starts ramping down over the next 10 minutes, so that at 01:00, when the first 3 users ramp down, 3 users from the six user wave are still ramping down. Finally all users are ramped down at 01:05. The steps with their timings for scenario 2 are:

- **00:00** - 3 users ramp up in 5 seconds.
- **00:05** - 6 users ramp up in 600 seconds.
- **00:15** - 6 users fully ramped up.
- **00:25** - 30 users ramp up in 60 seconds (spike).
- **00:45** - 30 users ramp down in 60 seconds (spike ends).
- **00:55** - 6 users ramp down in 600 seconds.
- **01:00** - 3 users ramp down in 5 seconds.
- **01:05** - Finished ramping down users.

6.2 Results: Scenario 1

Scenario 1 exhibiting gradual growth in request rate was handled quite well by the elasticity controller. Starting off with 1 application server instance and 6 users, a reference throughput (t_{vm} used in cost preference calculation) rate of 3.4 requests per second per server was established at 00:07 as show in figure 13 on page 50. Utility (fig. 2 on page 47) stayed above the scaling trigger value of 0.7 until the first two users were added at 00:10. The controller took 6 minutes to react to the increased amount of users. At 00:16, utility fell below 0.7 due to an increase in response time (fig. 11 on page 50) to $1.2s$ and hence a drop in the corresponding preference function value (fig. 5 on page 48) to 0.4. The MAPE-K planning phase was triggered and concluded with a decision to provision one new VM instance as scaling utility (fig. 3 on page 47) was positive at 0.6 at the time and response time slope (fig. 8 on page 49) was under 200 indicating no spike was occurring.

Two aplication server instances were enough to satisfy utility for most of the test duration. Utility dropped below 0.7 briefly at 00:21 during the controller's post provisioning quiet period. This utility fluctuation is caused

by a drop in both response time and cost preference values (fig. 6 on page 48). The metrics for request rate and response time fluctuated down and up at this time due to the system adjusting to the new virtual machine. Averaged metric values are affected immediately when a new instance shows up in the monitoring system (averaged system wide values are calculated by dividing a sum of each node's metric value by the number of nodes). It takes 2-3 minutes after this to start receiving actual correct data from the new node and for 1-minute and 5-minute averaged metrics to react. The quiet period accommodates this fluctuation and prevented a hasty further scaling decision here.

The next significant change in utility was at 00:33. Request rate (fig. 7 on page 49) drops to 2 requests/s for no apparent reason but then recovers to 3 at 00:37. Response time reacts upward to nearly $1.2s$ a bit later at 00:36 returning to $0.8s$ at 00:40. This appears likely to be a slow down in the infrastructure as the request load did not change at the time. Response time reacts slower than request rate because it is calculated using a 5-minute median as opposed to a 1-minute mean for the request rate.

At 00:54, utility fell below 0.7 again. Response time had increased earlier and starting from 00:50 request rate fell down to 4 from 6. This was likely another infrastructure performance fluctuation as a comparable request rate change did not happen after 00:40 when 2 more users started making requests similar to 00:50. On the other hand, the effect of those additional users shows clearly in response time as it increases to 1 second after the user increase at 00:40. With the response time increase followed by the request rate drop, the elasticity controller made a further scale out decision to add 1 VM instance.

The added VM arrived late for this scenario. Provisioning started at 00:54 and continued until 00:59. The metrics show the VM coming properly online just before the one hour mark at which the Gatling load script terminates. The completion of VM provisioning is seen in the response time preference value as it increases quickly at the same time as the VM count metric (fig. 9 on page 49) increases. Then response time preference decreases as the new VM starts reporting response times and takes time to get up to full speed. The scenario ends at 01:00 and the 1 minute average request rate metric reacts a minute later to drop utility below 0.7 again. This triggers another scaling decision whereby 2 virtual machines are terminated, returning the

system to its original configuration of 1 application server instance.

The overall QoE value for this scenario is 0.86. The system performed relatively well in terms of quality of elasticity. Normalizing QoE between a minimum of 0.7 and maximum of 1.0 (eq. 5 on page 27)yields a "score" of 53% to the system given this elasticity scenario.

6.3 Results: Scenario 2

The second scenario tests the system's behavior under a load which spikes, i.e. suddenly increases from 9 users to 39 users. Reference throughput used for cost calculation goes up to 3.6 this time at 00:12 (fig. 25 on page 55). This difference has to be attributed to the cloud infrastructure as the virtual machine instances are different and hence also the placement on physical hardware at Amazon's data center is most likely different. The maximum reference throughput was reached later than in scenario 1 due to the difference in user amounts. Scenario 2 has 8 users making requests at 00:12 whereas scenario 1 reached its maximum throughput with 6 users. VM instance performance is clearly different between the scenario executions.

Utility (fig. 14 on page 51) was at maximum until the user amount reached 8. Response time had slowly increased from 00:08 with the gradual increase of users and lead to utility dropping below the 0.7 threshold at 00:16. Provisioning of one VM instance was started and the VM became operational at 00:20 returning utility above the threshold. The metrics jittered a lot during the quiet period again as they adjusted to the new instance. After stabilizing, utility increased back to maximum at 00:26, just in time to receive the spike.

The spike started at 00:25 with 30 additional users ramping up over the next 60 seconds. The system was clearly not ready to handle this kind of load as the application server request queues started filling up (fig. 24 on page 54) which did not happen at all in scenario 1 (fig. 12 on page 50). At 00:28 the response time preference (fig. 17 on page 52) quickly fell from 1.0 to 0.0. Response time was increasing but the slope metric was close to 0 as response time now and five minutes prior was nearly at the same level. Utility decreased below the threshold, but elasticity controller failed to detect the spike because of the low slope value. Due to this, provisioning of only one additional VM instance was started at this time.

The spike continued while the single VM was being provisioned. Its introduction to the load balancing array understandably did nothing to make this better and utility was flat at 0.5. Utility did not decrease further because the cost preference function was at maximum, i.e. the request rate (fig. 7 on page 49) was high enough to keep cost per request low. At 00:34, six minutes after the misfired single VM provisioning operation, the quiet period ended and the elasticity controller was able to add more VM instances. This time 9 more were added, as the controller is configured to triple the amount of VMs if it detects a spike.

One of the 9 VMs failed in the contextualization phase and at 00:40, 8 new application server instances started receiving requests. During the quiet period and even after it until 00:44, utility was below the 0.7 threshold, but no scaling operations were executed. The scaling utility metric (fig. 3 on page 47), which indicates whether to add or remove instances, was negative but not below 0.5, so scaling was not executed. At the same time both the response time and cost preferences were below 0.8, and they cancelled each other out in the calculation of scaling utility. The metric stabilization took longer than previously with 8 new VMs, so the post scaling quiet period failed to protect the system. However as the metrics influence scaling utility in opposite directions, a further scaling operation was avoided here right before the spike ended.

After the spike ended gradually between 00:45:00 and 00:45:30, the request rate metric began to show this at 00:46. Cost preference decreased and response time preference increased to maximum. The cost increase triggered removal of 8 VMs at 00:46:30. The termination caused a sudden drop in request rate which brought utility down to 0.5 due to rising cost per request per server. This was during the quiet period, however, and the metrics stabilized prior to the end of the period.

With the spike over and 3 VM instances left, the system ran at approximately 0.85 utility until 00:55. After this the 6 user "second wave" started ramping down over the next 10 minutes. Utility slowly decreased due to this until a bigger impulse was given by the ramp down of the 3-user "first wave" at 01:00. Metrics reacted to this at 01:00:30 and the controller brought the system down to its initial configuration of 1 application server instance. The remaining second wave users finished their rampdown at 01:05 with utility

already back at maximum since 01:02:30.

The overall QoE value for this scenario is 0.78. The system did not perform well with regard to the spike usage profile represented by this scenarion. The scaling threshold was set at 0.7. Normalizing QoE between a minimum of 0.7 and maximum of 1.0 (eq. 5 on page 27) yields a score of 27% to the system given this elasticity scenario.

6.4 Notes on quality of elasticity

The QoE scores obtained in the tests were 53% for scenario 1 and 27% for scenario 2. The controller is better at managing elasticity when the metrics change slowly like in scenario 1. Main reasons for this are the delays observed after a scaling decision execution was started. It took roughly 3-4 minutes from decision to the point where a new virtual machine was available to service requests and another 3 minutes for the quiet period during which metrics were allowed to stabilize for the new configuration. The elasticity controller prototype took a simple approach to running the MAPE-K loop; it was run sequentially, one at a time. the total time between two control loop executions could range from 20 seconds (no scaling decisions made) to 6-7 minutes when the scaling plan execution phase waits to receive control of new VM instances and contextualizes them. An improvement could be to run the control loop during provisioning and the quiet period and implement the possibility to cancel an operation if the metrics start to give an opposite indication while provisioning is in progress.

The metrics, which are polled every 15 seconds, are averaged for stability which makes the slower to react to changes. A 5-minute average for response time was used as it was readily available in the Codahale Metrics open source project, but a faster metric could have helped with the reaction time and allowed a shorter quiet period for the control loop. Similarly the response time slope metric was calculated over 5 minutes, which worked against spike detection in scenario 2.

The cost metric is rather superficial in these tests as the size of the deployment is small and hence small changes in user amount are percentually large. In real systems where this kind of calculation becomes relevant, there would be a business case backing it and a reasonable cost preference could be derived

from it. Issues like billing period granularity, which this prototype does not consider, then rise to a more important status. The controller would then have to keep track of billing cycles and other real time cost factors to properly optimize QoE in terms of cost.

The cloud infrastructure itself played a variable role in the QoE results. Particularly in scenario 1, the request rate dropped twice for no apparent reason caused by the scenario. As the effect of this was to only reduce utility within the threshold limit, these performance fluctuations reduced QoE with no reaction from the elasticity controller. In scenario 2, one of the total 15 VM instance provisioning operations failed. This is notable because they operations are automated and hence user error is eliminated as a possible cause. Many things can still fail on many levels, as Amazon runs the EC2 service on commodity hardware and the infrastructure is accessed over the Internet and provisioning relies on the availability of many other Internet services like the Ubuntu package repositories.

As the cloud provider business model is based on guarantees of quantity over absolute guarantees of quality, the design for failure approach can be used to mitigate this kind of risk. Given the pay per use pricing model, it is inexpensive to overprovision by a certain percentage as an insurance against failed VM delivery or performance fluctuations.

cite amazon design for failure

7 Conclusion

Summary and conclusion.

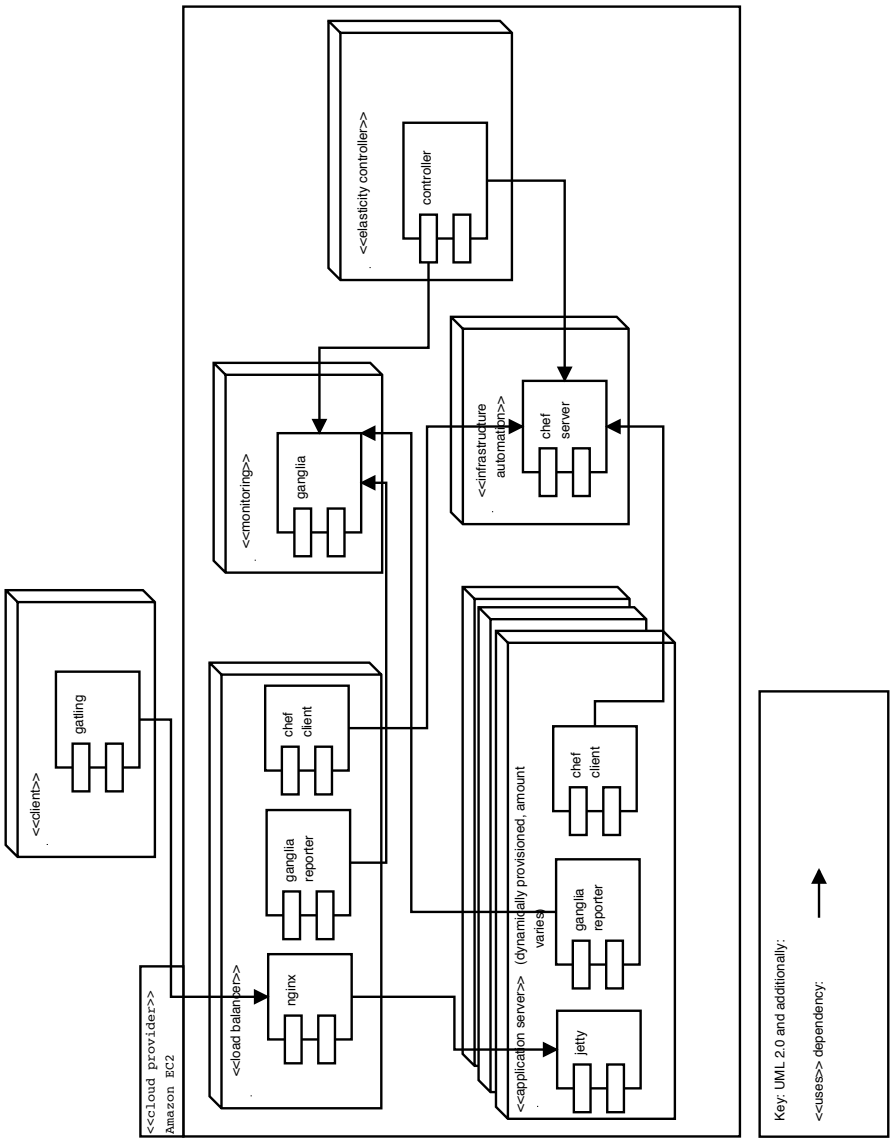


Figure 1: Prototype deployment diagram.

A Figures for test scenario 1

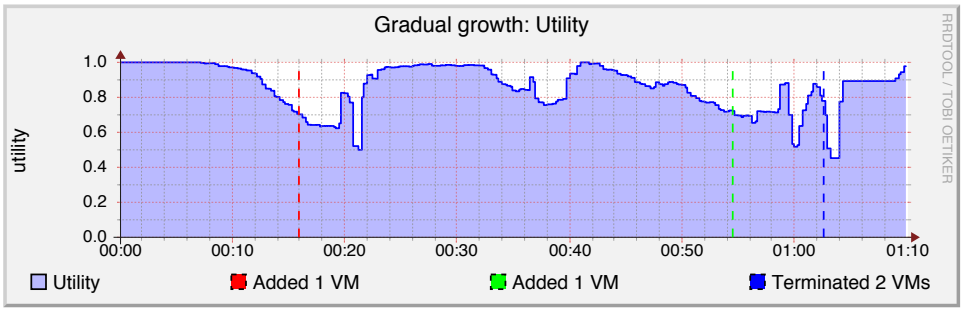


Figure 2: Utility during test scenario 1

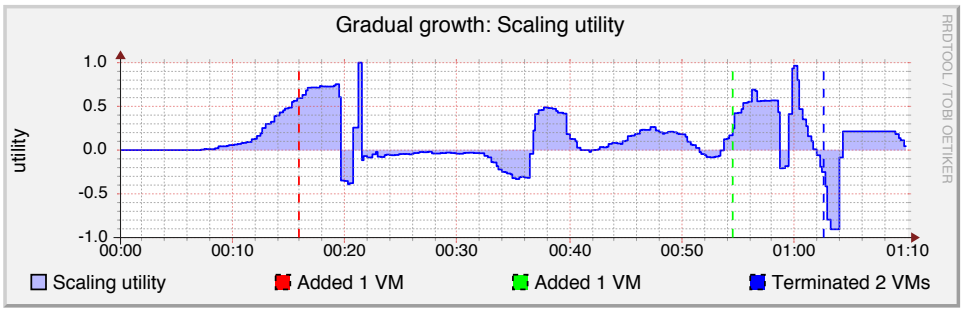


Figure 3: Scaling utility during test scenario 1

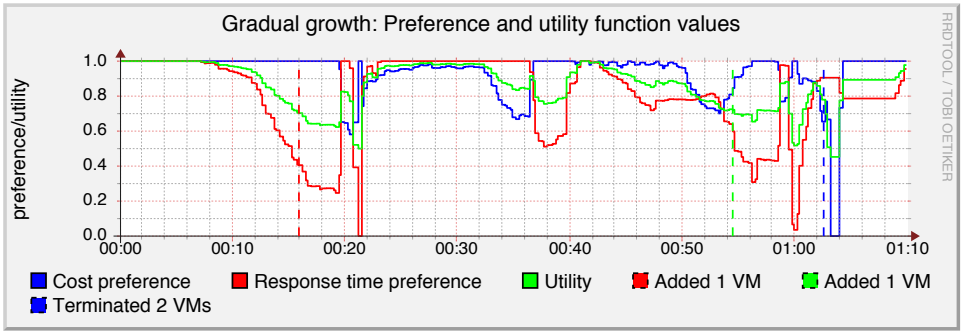


Figure 4: Combined preference function and utility values during test scenario 1

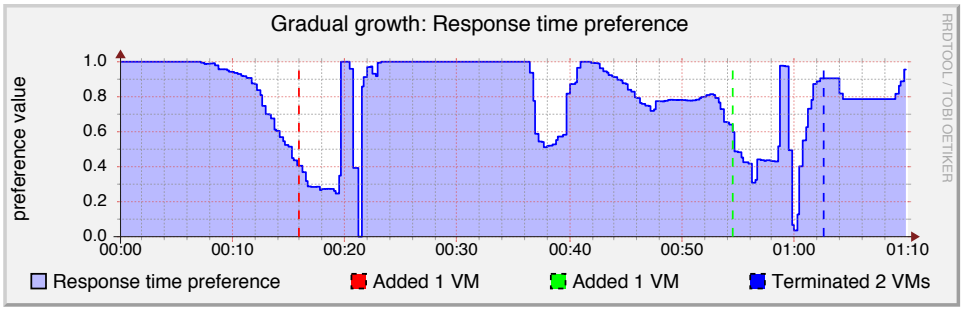


Figure 5: Response time preference function values during test scenario 1

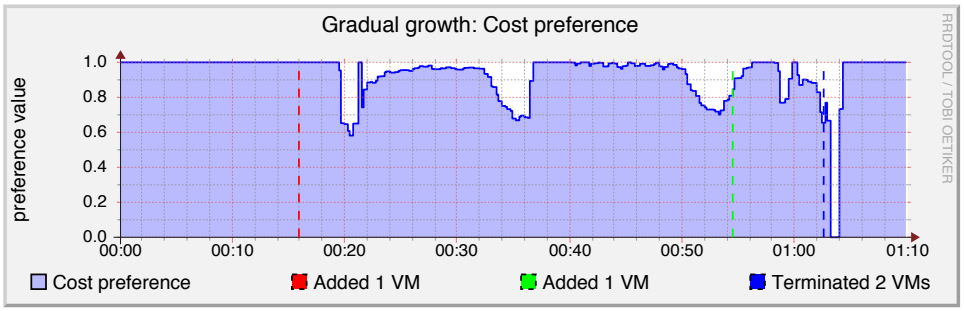


Figure 6: Cost preference function values during test scenario 1

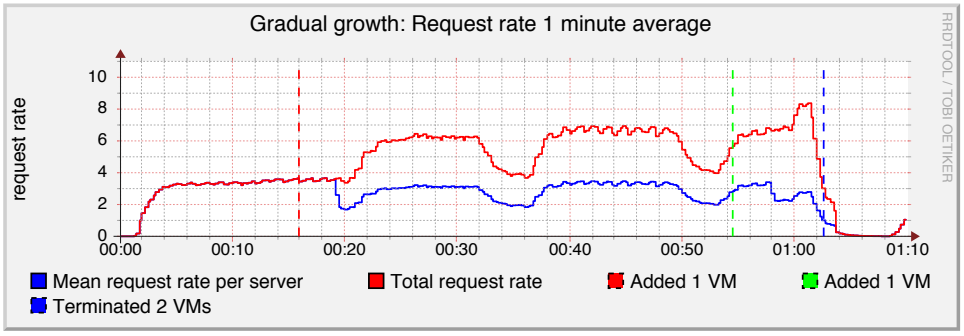


Figure 7: Request rate during test scenario 1

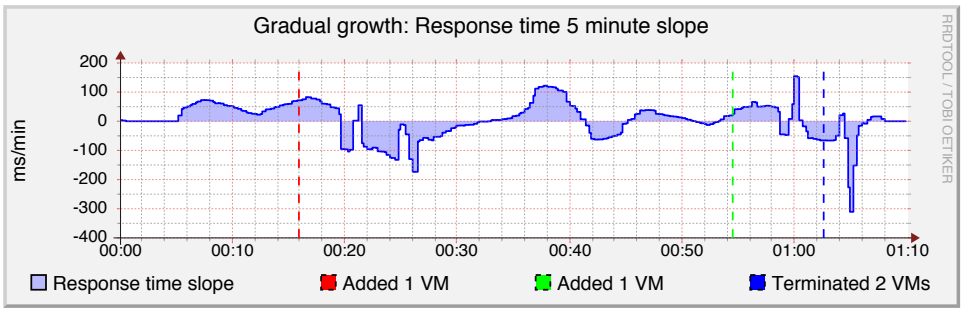


Figure 8: Response time slope during test scenario 1

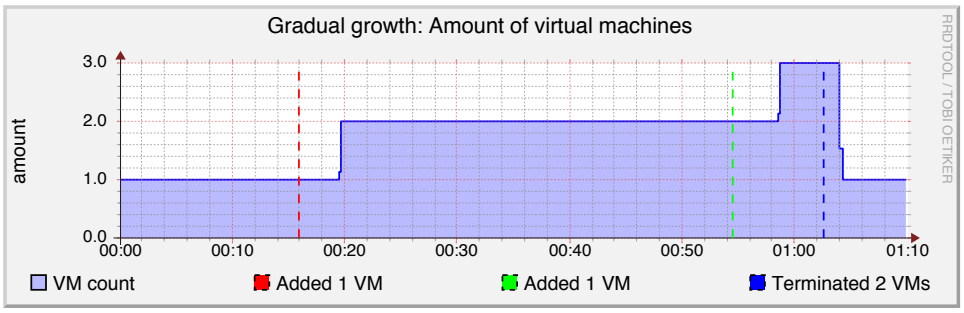


Figure 9: Scaling utility during test scenario 1

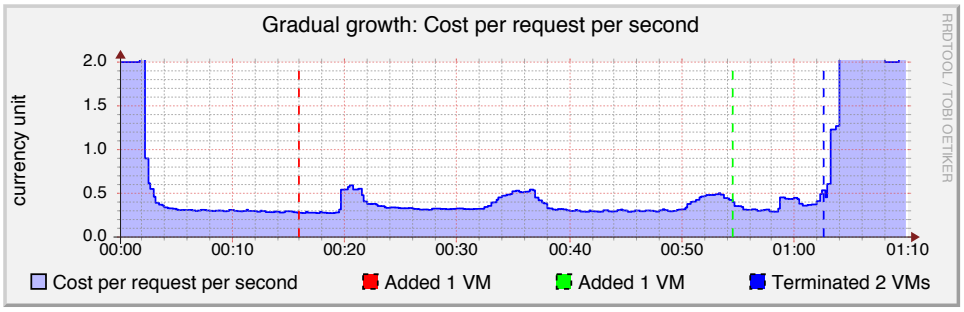


Figure 10: Cost per request per second during test scenario 1

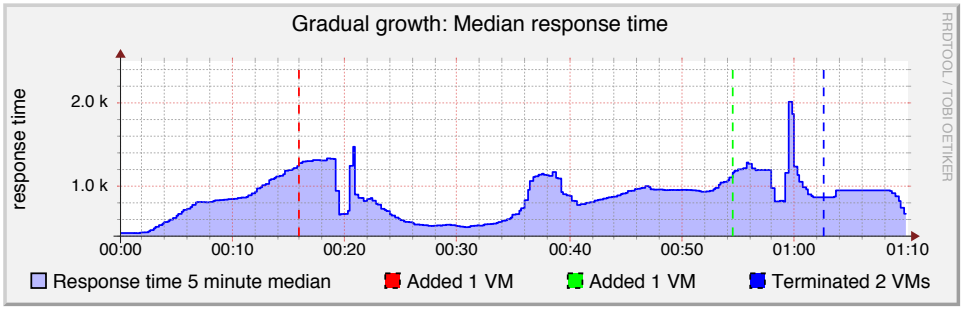


Figure 11: Response time during test scenario 1

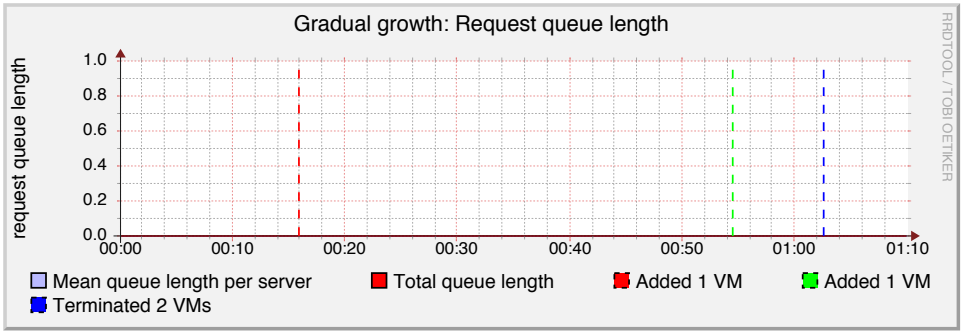


Figure 12: Application server request queue size during test scenario 1

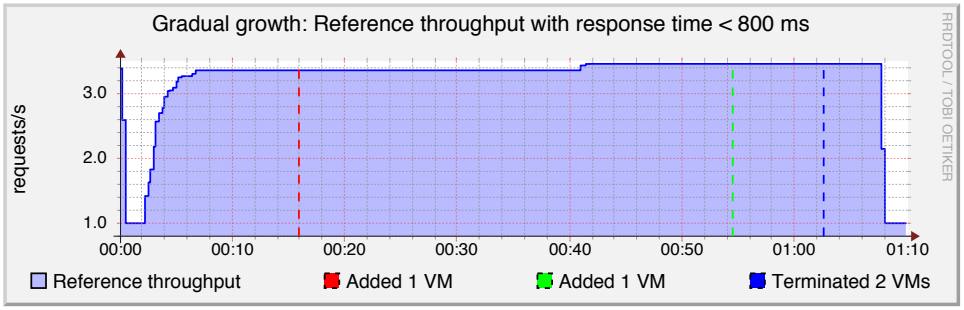


Figure 13: Reference throughput during test scenario 1

B Figures for test scenario 2

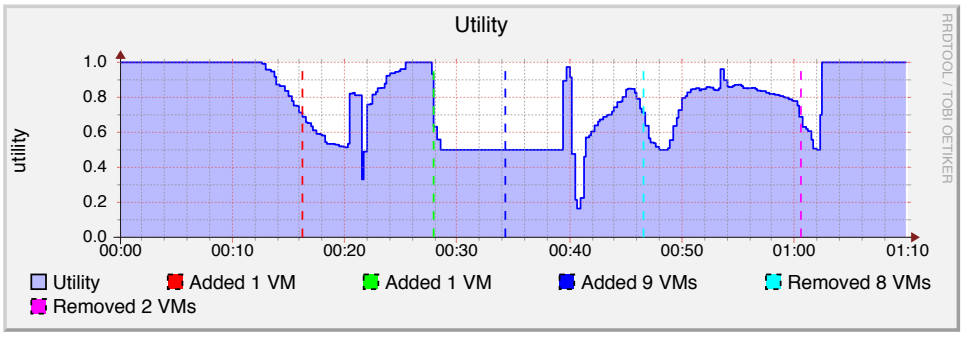


Figure 14: Utility during test scenario 2

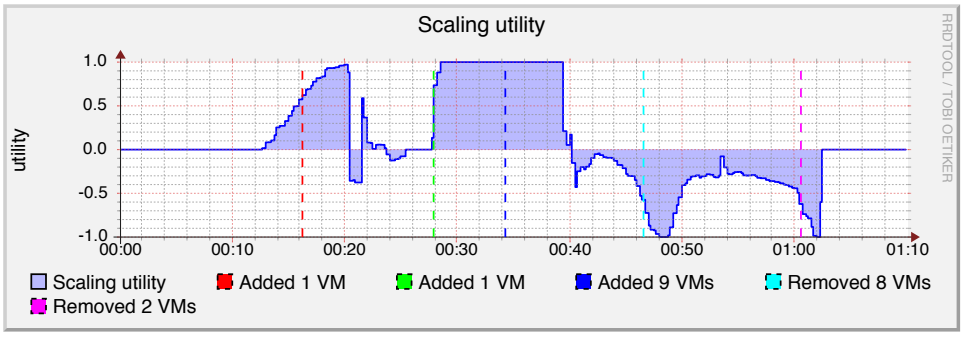


Figure 15: Scaling utility during test scenario 2

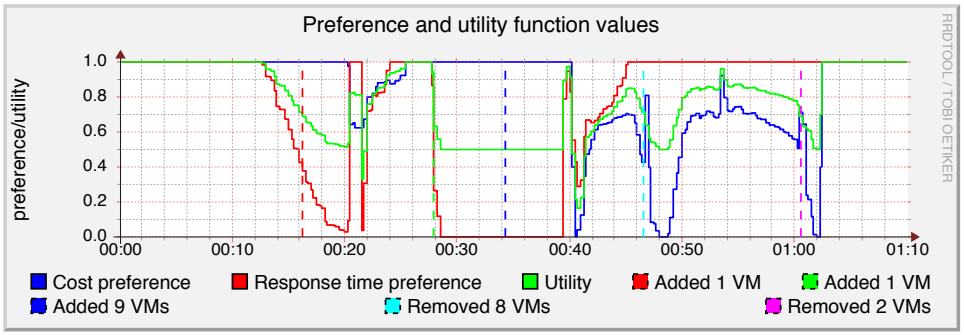


Figure 16: Combined preference function and utility values during test scenario 2

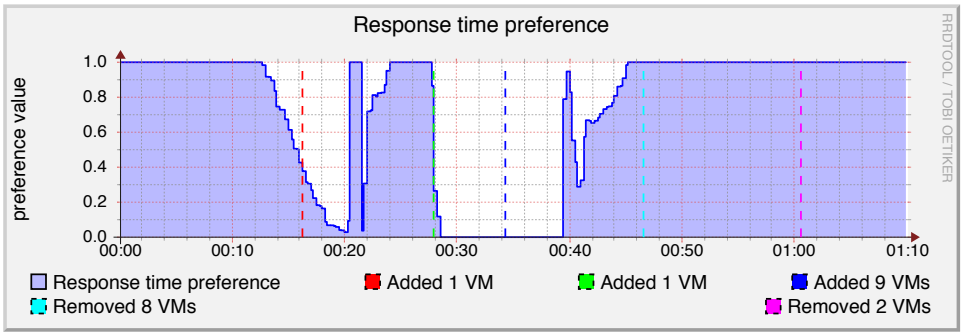


Figure 17: Response time preference function values during test scenario 2

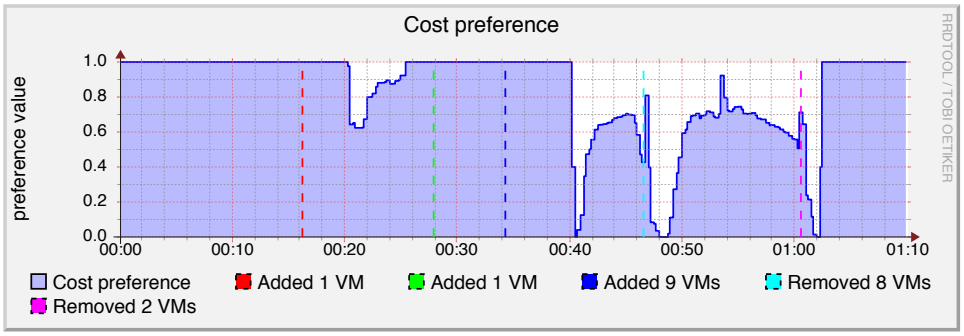


Figure 18: Cost preference function values during test scenario 2

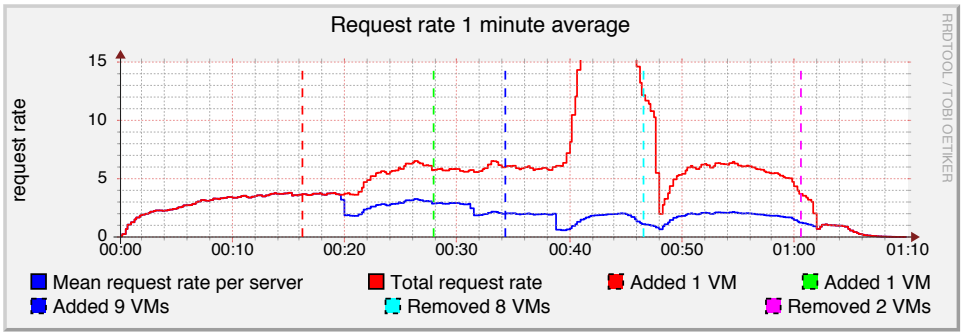


Figure 19: Request rate during test scenario 2

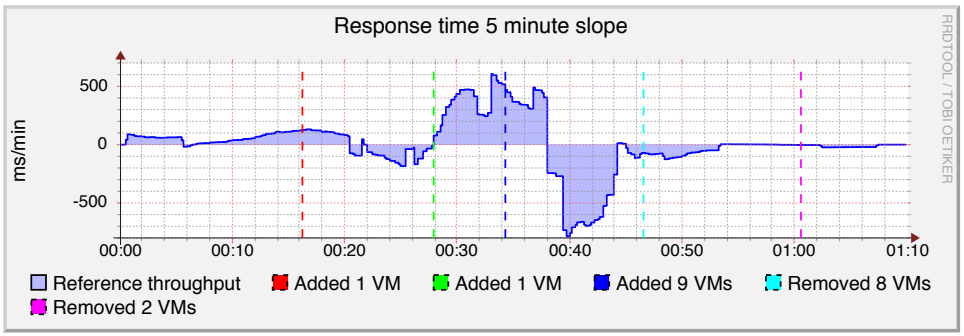


Figure 20: Response time slope during test scenario 2

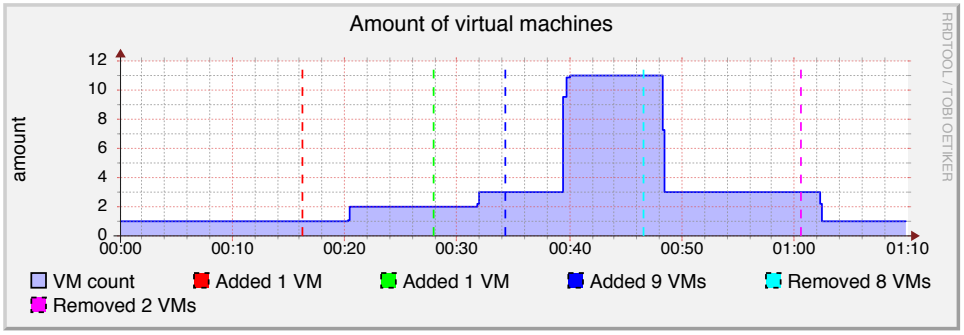


Figure 21: Scaling utility during test scenario 2

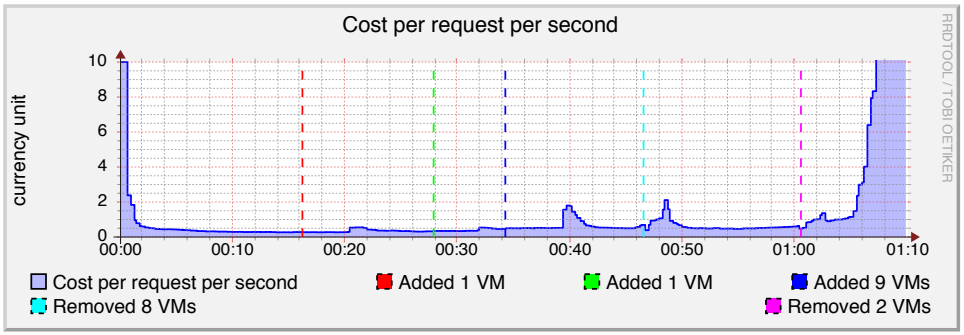


Figure 22: Cost per request per second during test scenario 2

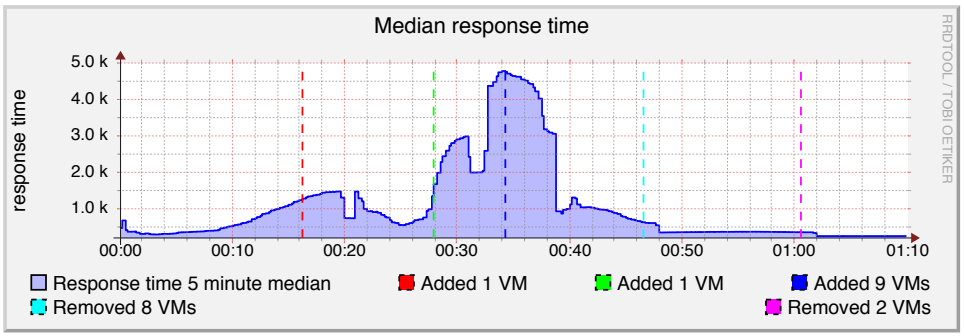


Figure 23: Response time during test scenario 2

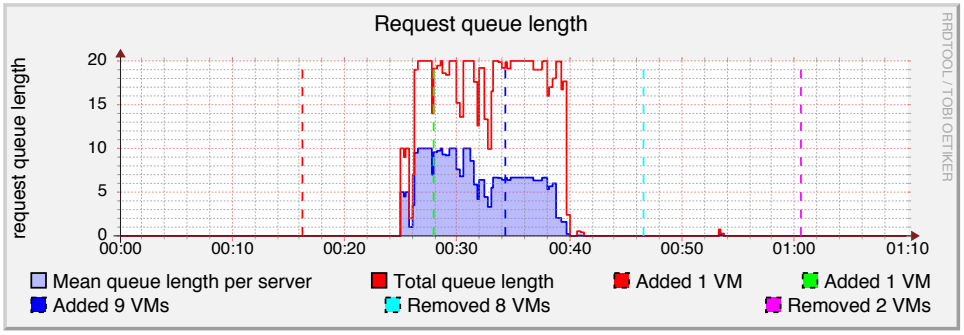


Figure 24: Application server request queue size during test scenario 2

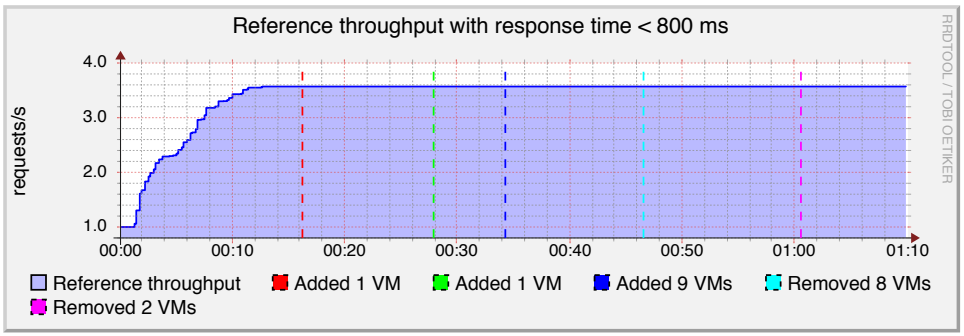


Figure 25: Reference throughput during test scenario 2

References

[1] *SETI@home website*. http://setiathome.berkeley.edu/, visited on 18/03/12.

[2] *OpsCode Chef website*, 2012. http://www.opscode.com/chef/, visited on 1st April, 2012.

[3] *Puppet Labs website*, 2012. http://puppetlabs.com/, visited on 1st April, 2012.

[4] Boloor, Keerthana, Chirkova, Rada, Salo, Tiia, and Viniotis, Yannis: *Management of SOA-Based Context-Aware Applications Hosted in a Distributed Cloud Subject to Percentile Constraints*. 2011 IEEE International Conference on Services Computing, pages 88–95, July 2011. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6009248.

[5] Brebner, Paul C.: *Is your cloud elastic enough?* In *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering - ICPE '12*, page 263, New York, New York, USA, 2012. ACM Press, ISBN 9781450312028. http://dl.acm.org/citation.cfm?doid=2188286.2188334.

[6] Buyya, Rajkumar, Yeo, Chee Shin, Venugopal, Srikumar, Broberg, James, and Brandic, Ivona: *Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility*. Future Generation Computer Systems, 25(6):599–616, June 2009, ISSN 0167739X. http://dl.acm.org/citation.cfm?id=1528937.1529211.

[7] Cáceres, Juan, Vaquero, L.M., Rodero-Merino, L., Polo, Álvaro, and Hierro, J.J.: *Service Scalability Over the Cloud*. In *Handbook of Cloud Computing*, pages 357–377. Springer, 2010. http://www.springerlink.com/index/MT877601J8700786.pdf.

[8] Chapman, Clovis, Emmerich, Wolfgang, Marquez, Fermin Galan, Clayman, Stuart, and Galis, Alex: *Elastic service definition in computational*

clouds. 2010 IEEE/IFIP Network Operations and Management Symposium Workshops, pages 327–334, 2010. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5486555.

[9] Chen, Xi, Chen, Haopeng, Zheng, Qing, Wang, Wenting, and Liu, Guodong: *Characterizing web application performance for maximizing service providers' profits in clouds*. 2011 International Conference on Cloud and Service Computing, pages 191–198, December 2011. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6138519.

[10] Clements, Paul, Kazman, Rick, and Klein, Mark: *The ATAM - A Method for Architecture Evaluation*. In *Evaluating Software Architectures: Methods and Case Studies*, chapter 3, pages 43–84. Addison Wesley, 2002, ISBN 978-0201704822.

[11] Cormode, G, Shkapenyuk, V, Srivastava, D, and Xu, Bojian: *Forward Decay: A Practical Time Decay Model for Streaming Systems*. In *Data Engineering, 2009. ICDE '09. IEEE 25th International Conference on*, pages 138–149, 2009.

[12] Crosby, S, Doyle, R, Gering, M, and Gionfriddo, M: *Open Virtualization Format Specification (OVF 1.1.0)*. Technical report, Distributed Management Task Force (DMTF), 2010. http://www.dmtf.org/standards/ovf.

[13] Cunha, Matheus, Mendonca, Nabor, and Sampaio, Americo: *Investigating the Impact of Deployment Configuration and User Demand on a Social Network Application in the Amazon EC2 Cloud*. 2011 IEEE Third International Conference on Cloud Computing Technology and Science, (section IV):746–751, November 2011. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6133224.

[14] Duboc, Leticia, Rosenblum, David, and Wicks, Tony: *A framework for characterization and analysis of software system scalability*. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07*, page 375, New

York, New York, USA, 2007. ACM Press, ISBN 9781595938114. http://portal.acm.org/citation.cfm?doid=1287624.1287679.

[15] Furht, Borko and Escalante, Armando (editors): *Handbook of Cloud Computing*. Springer, 2010, ISBN 978-1-4419-6523-3. http://www.springer.com/computer/communication+networks/book/978-1-4419-6523-3.

[16] Gunawi, HS, Do, T, and Hellerstein, JM: *Failure as a service (faas): A cloud service for large-scale, online failure drills*. Technical report, EECS Department, University of California, Berkeley, 2011. http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-87.pdf.

[17] Hill, Mark D: *What is scalability?* ACM SIGARCH Computer Architecture News, 18(4):18–21, December 1990, ISSN 01635964. http://portal.acm.org/citation.cfm?doid=121973.121975.

[18] Höfer, C. N. and Karagiannis, G.: *Cloud computing services: taxonomy and comparison*. Journal of Internet Services and Applications, pages 81–94, June 2011, ISSN 1867-4828. http://www.springerlink.com/index/10.1007/s13174-011-0027-x.

[19] Huebscher, Markus C. and McCann, Julie a.: *A survey of autonomic computing—degrees, models, and applications*. ACM Computing Surveys, 40(3):1–28, August 2008, ISSN 03600300. http://portal.acm.org/citation.cfm?doid=1380584.1380585.

[20] Iqbal, Waheed, Dailey, Matthew N., Carrera, David, and Janecek, Paul: *Adaptive resource provisioning for read intensive multi-tier applications in the cloud*. Future Generation Computer Systems, 27(6):871–879, June 2011, ISSN 0167739X. http://linkinghub.elsevier.com/retrieve/pii/S0167739X10002098.

[21] Islam, Sadeka, Lee, Kevin, Fekete, Alan, and Liu, Anna: *How a consumer can measure elasticity for cloud platforms*. Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering - ICPE '12, page 85, 2012. http://dl.acm.org/citation.cfm?doid=2188286.2188301.

[22] Kijsipongse, Ekasit and Vannarat, Sornthep: *Autonomic resource provisioning in rocks clusters using Eucalyptus cloud computing.* In *Proceedings of the International Conference on Management of Emergent Digital EcoSystems - MEDES '10*, page 61, New York, New York, USA, 2010. ACM Press, ISBN 9781450300476. http://portal.acm.org/citation.cfm?doid=1936254.1936265.

[23] Kirschnick, Johannes: *Toward an Architecture for the Automated Provisioning of Cloud Services.* IEEE Communications Magazine, (December):124–131, 2010.

[24] Korpela, E., Werthimer, D., Anderson, D., Cobb, J., and Leboisky, M.: *SETI@home-massively distributed computing for SETI.* Computing in Science & Engineering, 3(1):78–83, 2001, ISSN 15219615. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=895191.

[25] Liu, Huan: *Rapid application configuration in Amazon cloud using configurable virtual appliances.* In *Proceedings of the 2011 ACM Symposium on Applied Computing - SAC '11*, page 147, New York, New York, USA, March 2011. ACM Press, ISBN 9781450301138. http://dl.acm.org/citation.cfm?id=1982185.1982221.

[26] Mao, Ming and Humphrey, Marty: *Auto-scaling to minimize cost and meet application deadlines in cloud workflows.* In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*, page 1, New York, New York, USA, 2011. ACM Press, ISBN 9781450307710. http://dl.acm.org/citation.cfm?doid=2063384.2063449.

[27] Marshall, Paul, Keahey, Kate, and Freeman, Tim: *Elastic Site: Using Clouds to Elastically Extend Site Resources.* 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, pages 43–52, 2010. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5493493.

[28] Massie, Matthew L, Chun, Brent N, and Culler, David E: *The ganglia distributed monitoring system: design, implementation, and experience.* Parallel Computing, 30(7):817–840, July 2004, ISSN 01678191. http://linkinghub.elsevier.com/retrieve/pii/S0167819104000535.

[29] Mell, Peter and Grance, Timothy: *The NIST Definition of Cloud Computing*, 2011. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[30] Müller, H, Kienle, H, and Stege, U: *Autonomic Computing Now You See It, Now You Don't*. In *Software Engineering - International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures*, volume 5413, pages 32–54. Springer Berlin Heidelberg, 2009. http://www.nature.com/nature/journal/v435/n7046/full/4351165a.htmlhttp://www.springerlink.com/index/0k008550k285gq9v.pdf.

[31] Rodero-Merino, Luis, Vaquero, Luis M., Gil, Victor, Galán, Fermín, Fontán, Javier, Montero, Rubén S., and Llorente, Ignacio M.: *From infrastructure delivery to service management in clouds*. Future Generation Computer Systems, 26(8):1226–1240, October 2010, ISSN 0167739X. http://linkinghub.elsevier.com/retrieve/pii/S0167739X10000294.

[32] Roy, Nilabja, Dubey, Abhishek, and Gokhale, Aniruddha: *Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting*. 2011 IEEE 4th International Conference on Cloud Computing, pages 500–507, July 2011. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6008748.

[33] Suleiman, Basem, Sakr, Sherif, Jeffery, Ross, and Liu, Anna: *On understanding the economics and elasticity challenges of deploying business applications on public cloud infrastructure*. Journal of Internet Services and Applications, December 2011, ISSN 1867-4828. http://www.springerlink.com/index/10.1007/s13174-011-0050-y.

[34] Tan, Yongmin, Nguyen, Hiep, and Gu, Xiaohui: *PREPARE: Predictive Performance Anomaly Prevention for Virtualized Cloud Systems*. Proc. of ICDCS, (Vcl), 2012. http://www4.ncsu.edu/~ytan2/icdcs2012_tan.pdf.

[35] Van den Bossche, Ruben, Vanmechelen, Kurt, and Broeckhove, Jan: *Cost-Optimal Scheduling in Hybrid IaaS Clouds for Deadline Constrained Workloads*. 2010 IEEE 3rd International Conference on Cloud Comput-

ing, pages 228–235, July 2010. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5557990.

[36] Vaquero, Luis M., Rodero-Merino, Luis, and Buyya, Rajkumar: *Dynamically scaling applications in the cloud*. ACM SIGCOMM Computer Communication Review, 41(1):45, January 2011, ISSN 01464833. http://doi.acm.org/10.1145/1925861.1925869http://portal.acm.org/citation.cfm?doid=1925861.1925869.

[37] Verma, Akshat, Kumar, Gautam, and Koller, Ricardo: *The cost of reconfiguration in a cloud*. Proceedings of the 11th International Middleware Conference Industrial track on - Middleware Industrial Track '10, pages 11–16, 2010. http://portal.acm.org/citation.cfm?doid=1891719.1891721.

[38] Wilkes, John: *PRESS: PRedictive Elastic ReSource Scaling for cloud systems*. 2010 International Conference on Network and Service Management, pages 9–16, October 2010. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5691343.