Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Zsolt Siklósi

DYNAMICALLY SCALABLE APPLICATIONS IN CLOUD ENVIRONMENT

ADVISOR

Zoltán Micskei

BUDAPEST, 2013

HALLGATÓI NYILATKOZAT

Alulírott **Siklósi Zsolt**, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2013. 01. 29.

.....

Siklósi Zsolt

Összefoglaló

A számítási felhő – vagy egyszerűen felhő – az információtechnológia legújabb generációja, napjaink legvirágzóbb kutatási-fejlesztési területe. A különféle célú, modern, felhőalapú online szolgáltatásoknak köszönhetően maga az elnevezés is beépült a köznyelvbe. Folyamatosan fejlődő iparág, amelyben csupán a közös vonások és gyakori szolgáltatásmodellek szabványosítottak. A felhő paradigmája újszerű fejlesztői és üzemeltetői látásmódot igényel, míg a végfelhasználók számára a korábbiakhoz képest kevesebb kötöttséggel jár. A számítási felhő virtuálisan végtelen, magas rendelkezésre állású, hibatűrő, megosztottan használt erőforráskészletekhez biztosít hozzáférést előfizetőinek használatalapú elszámolásban. A piacon kínált szolgáltatások skálája a teljesen virtualizált, többrétegű infrastruktúráktól az egyszerű, testre szabható munkafolyamat-támogató alkalmazásokig terjed. E piaci szegmensben a szolgáltatások gazdag tárházát felvonultató szolgáltató-óriások mellett kis, úttörő cégek is érvényesülhetnek speciális egyedi vagy hibrid megoldások nyújtásával. A még gyorsabb terjedés egyik – egyre inkább elmosódó gátját a felmerülő adatbiztonsági és titoktartási aggályok jelentik, melyekre a szolgáltatók szigorú biztonsági előírások, szabványok és garanciák bevezetésével próbálnak válaszolni. A felhőt egyedülálló pozitívumai, mint például a hatalmas kapacitás, platformfüggetlen elérhetőség, az adminisztratív teendők leegyszerűsödése, vagy az alacsony beruházási és kiszámítható operatív költségek vonzó alternatívává teszik a hagyományos üzemeltetési modellekkel szemben.

Diplomám fókuszában a felhő egyik figyelemre méltó tulajdonsága, a dinamikus skálázhatóság áll. A felhőben működő skálázódó infrastruktúrák, alkalmazások lehetővé teszik, hogy a kívánt szolgáltatásszint fenntartásához mindig elegendő erőforrás foglalódjon optimális kihasználtságot és költségeket eredményezve. A skálázás továbbá megteremti a bejövő terhelés tervezett vagy kiszámíthatatlan változásainak függvényében történő automatikus reakciók lehetőségét. A skálázható felhőalkalmazás tervezése és fejlesztése azonban a megszokottól eltérő megközelítést és technikákat igényel, újszerű tervezési aspektusokat vezet be.

Diplomámon keresztül az Olvasó mélyebb betekintést nyerhet a számítási felhő világába, annak elméleti hátterébe és gyakorlati hasznába. Külön fejezetek mutatják be a jelenlegi két domináns szereplő – Amazon Web Services, Windows Azure – alaptulajdonságait és nyújtott szolgáltatásait, különös tekintettel a skálázási megoldásaikra, azok technikai és konfigurációs részleteire. Az esettanulmányok fejezetei végigvezetnek két, skálázhatóságra tervezett felhőalkalmazás tervezési és megvalósítási folyamatán. A skálázási szolgáltatásokról – Amazon Auto-Scaling és Azure Autoscaling Application Block – összegyűjtött tudásanyagot a gyakorlatba átültetve működő dinamikus alkalmazásskálázást valósítottam meg. Az Olvasó lépésről lépésre követheti végig az ehhez szükséges konfigurációs és egyéb teendőket. Az esettanulmányokban külön részt kaptak a teljesítménymonitorozással és az optimalizációs célú beavatkozásokkal kapcsolatos tudnivalók, megoldások, eszközök, illetve ismertetek néhány mérési eredményt.

Célom, hogy diplomám részletes, átfogó képet adjon a számítási felhőről, azon belül is a dinamikus skálázásról gyakorlatias megközelítésben, amely referenciaként szolgálhat a felhővel foglalkozó alkalmazásfejlesztő vagy -üzemeltető szakembereknek.

Abstract

Cloud computing or simply the cloud is the next generation of information technology and probably the most thriving area of research and development these days. Even the word 'cloud' has blended into our colloquial terminology as nearly all modern – professional, commercial, business, social, entertainment – services available on the Internet utilize the powers of cloud computing. The field is continuously evolving; only the common characteristics and basic service types are defined. It constitutes a paradigm shift that requires a different attitude from development and maintenance experts but is transparent to end users. Cloud computing provides practically unlimited, highly reliable and resilient resource pools for shared usage among numerous consumers in a usage-based billing system. The offered service models range from entirely virtualized, multi-layer infrastructures to simple, customizable workflow support applications. Besides a few mammoth providers offering a wide selection of cloud services, small pioneers are coming forward successfully by providing unique, specialized or hybrid solutions. The only barriers of further spreading are the privacy and data security concerns; however these are constantly fading because of the strict guarantees and standards that providers adopt. The appealing characteristics of the cloud such as infinite capacity, elasticity, high availability from various client platforms, the ease of setup and administration or the reduced initial costs and steady, calculable operation costs make the cloud a strong alternative to traditional hosting.

The focus of this thesis is dynamic scaling, one of the most remarkable cloud features. An automatically scaled cloud infrastructure or cloud application ensures that the amount of reserved resources is always sufficient to keep up a certain service level while optimizing costs by avoiding over-provisioning. But scaling makes it possible to react automatically to demand changes without any human intervention. Scaling services can dynamically change the size of the cloudware based on pre-configured rules in response to load swings. It must be noted that designing and developing cloud applications built for scale require an unconventional approach and techniques.

My thesis intends to provide the reader with a thorough introduction into the world of cloud computing by way of describing both its theoretical background and practical usage. I present the main details and available services of the two currently dominant cloud providers, Amazon Web Services and Windows Azure, emphasizing their scaling solutions with an extensive technical documentation section. I lead the reader through the reflective design and implementation process of two different cloud applications that comply with the special cloud design principles. The knowledge gained of scaling services is then being put into practice by setting up and configuring dynamic scaling for the demonstration applications. I go through the necessary steps and actual tasks of configuration and maintenance of scaling solutions on two platforms. The performance monitoring sections intend to give an insight into the experiments and a deeper understanding of scaling behavior.

My goal is to create a mostly practical thesis on the paradigm of cloud computing through the described case studies that can serve as a reference on the two cloud platforms and the available scaling solutions from a developer and administrational aspect.

Table of Contents

1 Introduction

The cloud computing paradigm has become one of the most noticeable and interesting novelty in the IT world of our days. Although, it is based on existing technologies such as virtualization, it introduced an entirely new concept, and it requires an unusual point of view. Hosting applications and information in the cloud has many benefits compared to the time and resource consuming tasks of operating and maintaining an on-premises host, which also requires expertise. Moving to the cloud removes the burdens of administration, meanwhile provides a highly reliable, flexible environment. The cloud enables its users to reach optimal resource utilization by increasing or decreasing capacity according to current demand. With automatic scaling one can prevent the drawbacks of over-provisioning resources, and the system will also be able to handle unforeseeable spikes in demand. The change in capacity can be accomplished with little or no human intervention, the system scaling can happen automatically based on user-defined rules. Dynamic scaling provides the ability to respond immediately to changing external or internal circumstances and react automatically.

The aim of this thesis is to discover and analyze the characteristics of public and private cloud computing environments, and to explore the possibilities of dynamic scaling solutions. I will also dive into scalability, explore the available patterns and methods of designing applications optimized for working in the cloud. Using the gathered knowledge and practices, I designed and created two demonstrational applications for different cloud platforms. Development on these platforms was only possible after an extensive examination of the chosen platforms and the discovery of the provided services that are useful in common application design scenarios. Among the wide palette of various services, I especially focused on those connected to elasticity and scaling. I went through in high detail the available documentation of scaling services noting and comparing their capabilities. The scaling solutions that I created and applied to my cloud applications provide a simple but realistic example of how scaling works and the way of configuring and maintaining scaling services of well-known cloud platforms.

The next chapter introduces the reader to the commonly accepted and standardized definitions and distinctive features of cloud computing and shares some thoughts on the assessment of the cloud from different aspects. In the third chapter I approached scalability, one of the most appealing features of cloud computing, on a theoretical level. Possible methods of scaling and some noticeable application design and development patterns are discussed. In the fourth chapter I intended to describe the common services offered by the most notable cloud providers and delve deeper into the scalability features.

The detailed description of scaling solutions helps in understanding the following chapters in which I go through the design, implementation, and configurative tasks of the cloud application case studies on two platforms. This part is the most extensive because I tried to show layered pictures of the two scaling solutions through specifically designed cloud applications. I explain and demonstrate the majority of their features, enumerate their strengths and flaws while configuring then testing their capabilities. Each case study is closed with a section dedicated to

the practicalities and available tools of monitoring the cloud infrastructure. With these I intended to emphasize the importance of measuring application performance and scaling efficiency, and the fact that scaling and monitoring are inseparable.

My thesis tries to give a general knowledge of the paradigm of cloud computing and one of its most important characteristics, scalability. Also it methodically leads through the reader in the major public cloud platforms and their jungle of services allowing comparisons. The case study solutions thoroughly described give the reader an experience of developing cloud applications ready for dynamic scaling and the common configuration and operation concepts of scaling services.

2 Cloud computing definitions

This chapter intends to give the reader a short theoretical overview of the background of cloud technology. The cloud can be considered a new computing paradigm, the upcoming generation of IT evolution. Due to its modernity, it is continuously shaping and evolving. Cloud computing has become more of a buzzword than a definitive phrase due to the exuberance of marketing in IT industry. In the relevant articles and papers across the Internet one can find many contradictions and shallow, imprecise descriptions. I write down the industry standard definition of cloud computing laid down by NIST[1] along with some other fundamentals published by the Institute. Besides the definition I enumerate the many advantages of the technology in contrast with its drawbacks which still restrain the spread of clouds. Then I continue with the typology of cloud service models, I decipher the *aaS acronyms and illustrate the notable differences between the three main types. I also mention a few examples and significant providers found on the online market.

2.1 Cloud computing

2.1.1 NIST definition

> Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

The emphasis is on the on-demand attribute. The cloud should mean an ever-accessible, easily maintainable infrastructure that can be adapted to needs.

NIST completes this definition [1] with the following five essential characteristics of the cloud model:

- *On-demand self-service*

  This characteristic describes that the end users of the cloud are individually able to adjust the amount of resources to meet their need.

- *Broad network access*

  Services in the cloud should be available via standard networking for consumers using a wide range of client platforms.

- *Resource pooling*

  The provider houses a multi-tenant model for its clients that are meticulously isolated from each other. The consumers can require various physical and virtual resources from pools to meet their demands. This dynamic provisioning cannot affect the performance of the consumers' allocated environment.

[1] National Institute of Standards and Technology, http://www.nist.gov

- *Rapid elasticity*

  Computing clouds should be flexible and instantly adaptive so to meet demands. This characteristic introduces scalability of the cloud which will be thoroughly described in a later chapter.

- *Measured service*

  Resource usage of the cloud infrastructure should be monitored in high detail. Providers need to apply an automated measurement and analysis system which makes optimization and usage reporting possible.

Besides three service models that are discussed in the next part, NIST standardizes four cloud deployment models based on the public availability of the cloud infrastructure:

- *Private cloud*

  As its name suggests, a private cloud is used exclusively by an organization. Only the appointed consumers can access its services, or contracted third party companies that are responsible for the technical operation and management. A private cloud infrastructure may exist on or off premises of its owner.

- *Community cloud*

  A cloud deployment of this type is used by a specific community of individual and corporate members who have shared interests or goals.

- *Public cloud*

  The cloud infrastructure is deployed to be openly accessible for the general public. A business, academic, or government organization owns and manages it.

- *Hybrid cloud*

  This infrastructure is a composition of the aforementioned deployment models. Its components remain unique entities, but they are technologically bound together.

2.1.2 Pros and cons of cloud computing

The cloud is an entirely virtualized environment usually off premises of its owners who are able to reach its services through public networks. This way of storing and transmitting valuable private information raises concerns about security and privacy [2]. Looking at these aspects, cloud computing has many advantages and disadvantages, also they hold a great perspective of future improvements.

The strongest advantage of cloud computing may be that utilizing the power offered by a cloud infrastructure does not require that diverse administrative knowledge as opposed to traditional on-premise server hosting. Cloud platforms simplify management tasks by providing rich but relatively easy to use interfaces where administrators can perform regular tasks around their cloudware. Another advantage is related to costs and the ratio of utilization. The expenses of investment into buying and installing high-quality server hardware far exceed the initial costs of subscribing to the cloud and forming a complex virtual infrastructure. The operative expenses

may be higher but cloud platforms bill on a pay-what-you-use basis. Scalable resource provision makes it possible to reserve only as much capacity as the actual circumstances demand. This ensures optimal resource utilization with optimum operational costs.

Regarding the safety and reliability concerns it can be stated that cloud providers have very strict technological standards and methods with substantial financial, technical and human resources to ensure security for their clients. These providers constantly guarantee information security according to state-of-art technology and privacy laws. Their reputation is fragile which they must maintain with their best effort. Another important feature supporting cloud technology is availability. Cloud computing environments are built to endure, designed to be fault tolerant. Providers offer and adhere to SLAs with availability close to 100%. They spare no resources to make their services reliable by bringing up highly redundant systems complete with rigorous backup and recovery policies. Availability is considered to be global, consumers may equally access cloud services from every corner around the globe.

Most of the disadvantages of cloud technology are rooted in its complex structure and public accessibility. Providing highly reliable, always online, dynamically scalable sets of resources to many consumers concurrently requires high environmental complexity. The contrast between a simple server housing common services for an organization, and a cloud environment is blinding. Deeper complexity always comes with deeper security issues that must be taken care of. The vast publicly available surfaces, containing a myriad of services comprise many possibilities for attackers.

Cloud computing holds opportunities never seen before, however one can find its backsides as well. It is a thriving sector of today's IT industry, therefore it is continuously reshaping, improving.

2.2 Cloud service models

Three main service deployment models can be differentiated. These differ in the scope of control of the cloud environment that is made up of five conceptual layers (from bottom to top):

- *Facility: this layer includes all the orchestration and physical environment used when housing such an infrastructure.*
- *Hardware: contains all the physical components that build up the whole system.*

These two bottom layers are entirely controlled and maintained by the provider. Above these the logical layers of the infrastructure can be found:

- *Virtualized infrastructure: it consists of the lower software elements, virtualization hypervisors, cloud controllers, virtual machines, and other virtualized resources.*
- *Platform architecture: contains a programming platform complete with installed libraries, utilities, middleware and other software components which enable running, developing and deploying applications.*
- *Application: the top layer represents a fully functioning, utilized application specifically designed for working in a cloud environment.*

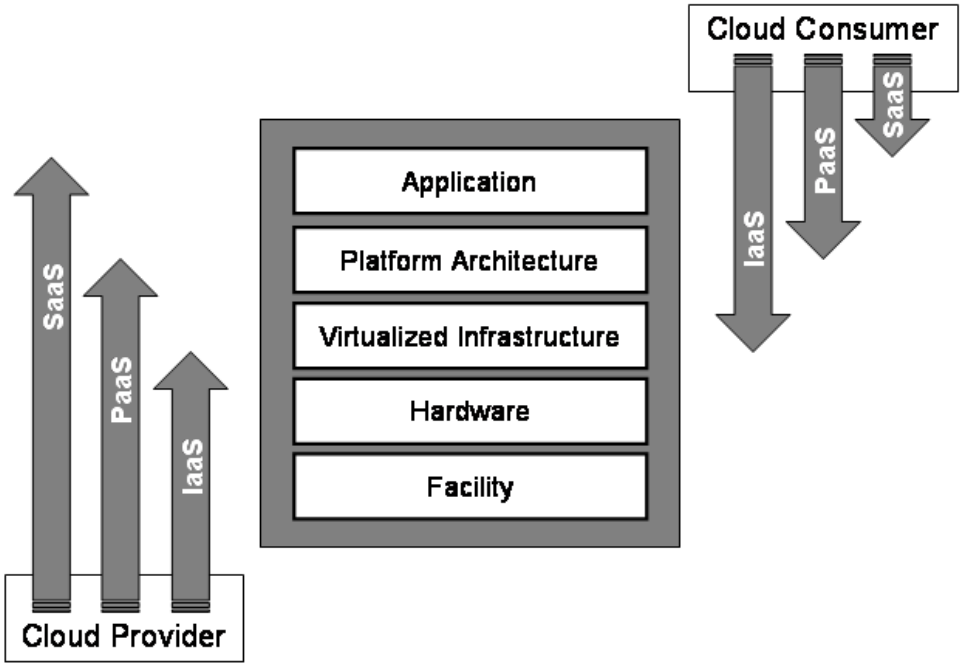

Figure 2-1. - Cloud deployment models (source: [1])

2.2.1 Infrastructure-as-a-Service (IaaS)

This model of service delivers a complete computing infrastructure, which can be configured, scaled, utilized on demand of its user. By subscribing to an IaaS cloud, an organization can avoid the costs and difficulties of purchasing, housing and maintaining various hardware and software components of an infrastructure. Instead, the organization gains full control of an entirely virtualized environment including virtual machines, storage solutions, firewalls and other network elements that can be adapted to meet their needs in every detail. The main benefits of using IaaS are flexibility and usage based billing.

The market leading [3] IaaS provider is Amazon Web Services[2]. I dedicate an entire chapter to outline its enormous capabilities and many services. AWS provides so substantial a service that several smaller providers offer their own, customized cloud systems based on the Amazon cloud. Other significant members of the IT market, such as HP, IBM also offer IaaS services. There are other notable providers worth to mention, like Rackspace or SoftLayer.

2.2.2 Platform-as-a-Service (PaaS)

A cloud built by PaaS model delivers a computing platform including the operating system, a fully configurable application execution and development environment, persistent data storage solutions, and many other built-in components that ease the process of developing, deploying, hosting applications in the cloud. Using a PaaS type cloud in different stages of software development reduces the infrastructure expenses as well as removes the burdens of managing the complex hardware and software layers beneath the application, lets developers focus on the application itself.

[2] http://aws.amazon.com/

There is a great abundance of PaaS providers on the market. I would highlight and will demonstrate the capabilities of Microsoft's Windows Azure[3] offering the rich soil of the .NET platform, and Google's App Engine[4], which enables cost-effective hosting of Java or Python based applications. There are many smaller companies or even start-ups noteworthy offering their custom cloud-based services, such as GigaSpaces, Force.com, Engine Yard, WorkXpress.

2.2.3 Software-as-a-Service (SaaS)

SaaS cloud providers offer ready-to-use cloud applications for their consumers who access its features via intranet or the Internet. It allows clients to use business software services, such as accounting, CRM, ERP and several other types for a monthly fee instead of having to design, develop and host it on their own. Cloud applications are hosted in secure, reliable and highly scalable environments to be able to adapt to real-time demands and ensure a steady performance. This underlying complexity is entirely transparent for service consumers whose role expands only to administer, configure and customize the provided software to maximize the efficiency of its usage.

Most of the SaaS providers base their services upon existing IaaS or PaaS solutions as these provide a good foundation to build highly elastic and complex applications on. The most well known SaaS applications are being used by everyday users. Google's mail service and Documents application are accessible to the general public but organizations may subscribe to them for internal use. It is not possible to highlight a few SaaS services for their palette is so broad in any type of applications.

[3] http://www.windowsazure.com/

[4] http://appengine.google.com/

3 Methods and patterns of scaling

In this chapter I give a deeper theoretical overview of scalability and the possibilities it holds emphasizing the currently available scaling techniques and their characteristics. The most important software design considerations and principles that should be adopted to enhance application scalability are gathered in a separate part.

3.1 Scalability

It is hidden in its name, scalability is an ability of a system. The proportions and capacity of a scalable hardware or software system can be adjusted so that it can handle the growth of load and perform its work optimized to performance and costs [4]. One cannot find a generally accepted, formalized definition of scalability; it is mostly determined by the features of the given system. Scalability is not exclusively an IT phenomenon, it can also be found in electric engineering, and even in the business world.

There are two distinct ways of scaling of systems [5]:

- *Vertical scaling*

  When a single system has the ability of vertical scaling, it means that the amount of its own resources can be adjusted in order to increase or decrease the capacity of the system unit. Virtualization technology enables to add or withdraw system hardware resources without physical efforts.

- *Horizontal scaling*

  The system is said to be scaled horizontally or scaled out, when instead of extending a single node, one or more individual nodes are added to the system. Such a distributed system holds much greater overall capacity and fault tolerance but it also has disadvantages attached.

It is possible to measure scalability with the scalability factor, which declares what percentage of the added resources - such as CPU, memory, networking hardware, etc. - can be actually utilized. Such components tend to have an operational or administrative overhead. Especially in distributed systems where the work of nodes must be precisely coordinated, this administrational overhead can increase by adding more components or nodes to the system. In such cases the global scalability factor of the system decreases with the growing number of added components.

Scalability might be classified based on the value of the scalability factor (SF). The theoretical optimal value would be SF = 1. This denotes linear scalability, when the overall capacity of the system linearly increases with adding more components to it. In most cases this cannot be achieved and the value of the SF stays below 1. This is called sub-linear scalability. The most pessimistic and also rare case is called negative scaling (SF < 0), when the architecture of the system does not enable scaling, therefore adding more components results in drastic degradation of the overall capacity.

3.2 Scaling in the cloud

In cloud computing scalability is achieved by horizontal scaling. The elementary computing units are made up of virtualized resources. The entirely virtualized cloud environment enables systems running in it to scale out by attaching and utilizing more computing units. This makes the cloud system elastic, as its proportions are continuously changing in response to internal or external circumstances.

Basically there are two methods for scaling cloud systems, scaling manually by the administrator or scaling automatically based on pre-defined rules. Obviously, manual scaling is of not much use, since it lacks dynamism, the ability to immediately respond to changes. In general, automatic scaling rules state conditions, which when met trigger actions. Conditions can be formed using different measures or metrics that describe the momentary state of the system from many angles. Such metrics are CPU utilization, memory or I/O usage, besides most cloud providers make it possible to create custom metrics. Complex rules can be assembled with the help of various arithmetic and logic functions applied to the metrics as operands.

Cloud providers have built-in and also configurable mechanisms for evaluating rule conditions. The administrator of the software system hosted in the cloud has the possibility to set the rule evaluation period or the action that needs to be performed when rule conditions are met. The action usually means positive or negative scaling out, attaching or removing components from the system. During the testing phase of automatic scaling, the administrator can request the cloud to send notifications via e-mail instead of actual scaling operations.

The scaling of a system may not exclusively mean changing the number of attached computing units, for there are other key components that have to be taken into account. These components, such as load balancers have high influence on the scaling potential of the system. Load balancers are responsible for spreading incoming requests equally across computing units. Most load balancing solutions include the ability of monitoring computing units attached underneath them, therefore the current load of computing units is taken into consideration when the load distributing algorithm routes incoming requests. With the growth of incoming connections, the pressure increases on the load balancer. Since it forms a bottleneck of the whole system below it, it is of paramount importance to ensure its operability.

3.2.1 Scaling the infrastructure

Clouds built on the IaaS service model regard single virtual machines as the elementary computing unit. Common IaaS providers do not enable to treat applications running on several computing units as a whole entity, the dependencies and functional relationships among units cannot be declared. As it will be elaborately described in a later section, cloud applications must be designed with due foresight to make them able to function in such an environment.

Scaling complex, multi-tiered applications is difficult with a VM-based approach, since the scaling conditions are evaluated based on the monitored performance of single VMs, regardless of the overall state of the system and its tiers. Vaquero et al. in [6] introduce a so called elasticity controller, implemented as a higher abstraction layer, which could coordinate scaling on separate tier levels or global application level. Scaling would be based on rules and performance

data would be collected from sensors woven into the application. This is a theoretical method that helps to overcome the limitations of the VM-based scaling mechanisms of IaaS clouds.

3.2.2 Scaling the platform

PaaS clouds offer a ready-to-use execution environment along with other useful accessories of a software platform. Such platforms have two basic layers, the container in which the applications are running and the database that provides persistent data storage. The containers provide an isolated environment for deploying and running application components, including several available service APIs and middleware functions, such as lifecycle management or messaging.

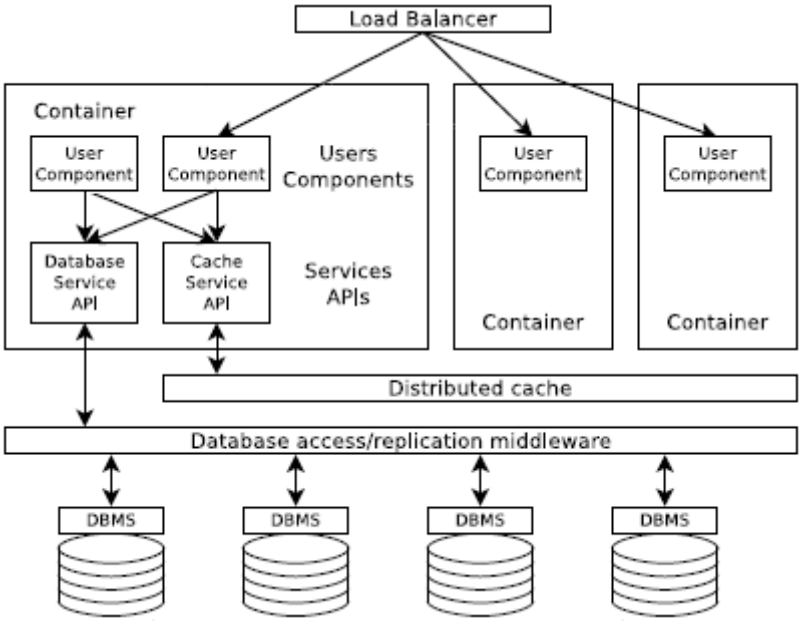

Figure 3-1. - Schematic view of a PaaS system (source: [6])

The container and database layers and the possible connections between them are present on Figure 3-1. The application components running in containers are registered in the load balancer which represents a gateway to the whole application. A distributed caching service is part of the displayed system, which can affect the application performance in a positive way by storing and supplying the most frequently used application data. The caching tier might get a key role in case of heavily data oriented systems. Beneath the application components lays the database layer containing a cluster of databases. An abstracted middleware layer brokers the data access operations across this cluster. Different replication methods can be applied to make the cluster and the stored information highly available and fault tolerant.

Scaling might appear both the container and the database layers in case of the growth of incoming connections. One approach to scale the container layer horizontally is to enable multi-tenant containers that can host and run more than one application components. This method requires strict security measures that ensure total isolation of components that might be owned and operated by different consumers of the platform service. Another solution is to place each component of a user into the same container. Either way scaling means the process of adding containers that can house several new instances of user application components, or releasing

existing containers in case of scaling down. Scaling operations shall be totally automated, entirely coordinated and administered by the platform. This volatile lifecycle of containers demands unconventional component design considerations.

Database scalability is such a vast topic that entire chapters could be dedicated to its background and the different solutions and practices. With cloud computing so called NoSQL databases are gaining importance, which offer high availability and a better potential of scalability than the traditional relational databases. The common advantages of the various NoSQL database systems that they are easy and very economic to cluster, they are designed to operate with as little human intervention as possible, and they have immensely flexible data models. However, this appealing behavior comes with drawbacks, such as the lack of deep business analytic functions, or that in some systems built-in replication mechanisms does not guarantee data consistency, thereby developers cannot implement ACID compliant transactions.

Traditional relational database management systems are also available, which stand on firm ground regarding database clustering, and the existing replication techniques provide data consistency and high availability. However, the replication mechanisms fail to keep up performance with expanding load, mostly because of conflicts occurring about data locked by running transactions. Housing and operating such clusters require more costly hardware infrastructure and trained administrative personnel. Relational database systems exist that make scaling possible. They introduce a middleware layer between the data and its consumers, the applications, which may reach the data through a proxy. This proxy tunnels requests to the middleware, which redirects them to the database. The middleware also takes care of coordinating replication and load balancing.

Database partitioning allows better scalability, higher availability and reliability. Many patterns have evolved from the simple master-slave architecture to sharding. Each has its own advantages and drawbacks. Sharding is an important paradigm of database scalability; its new concept uses a shared-nothing approach. The database is split up by some dimension or other method to smaller individual databases, and these shards can be spread across distributed database servers [8]. The main advantage of this method is that smaller databases are easier to manage, provide better performance, while they cost less.

Analyzing the theoretical background and the available solutions could be more exhaustive, however examining the data-tier scalability in detail does not fall under the scope of this work.

3.3 Cloud application design considerations

As the previous parts shed some light on the architecture of clouds, it is now obvious that designing and developing cloud applications demand unaccustomed approaches. Many useful articles can be found [9], which besides showing a way to adapt traditional concepts to the cloud, introduce wholly new aspects brought in by cloud computing features.

The cloud fundamentally provides a scalable environment, a great degree of elasticity, therefore the most important property of a cloud application is scalability. An application is said to be scalable when the proportion of its resources can be adjusted to reach optimum performance meanwhile keeping up operational and cost efficiency.

In the following section I will describe the most important architectural patterns and design practices suggested by the significant cloud providers. These concepts contain many parallel thoughts with basic SOA principles and ground rules of building highly available web applications.

Component decoupling

Cloud applications also take advantage of the ground rule of SOA that states to create loosely coupled components within the software system, which exist as individual entities. Decoupling components comes with many benefits, such as allowing reusability, supporting interoperability between systems, or reaching higher fault tolerance. When designing an application, tight dependencies amongst components need to be eliminated, so the failure or temporary decreased throughput of a component would not affect the performance of the whole application. Decoupling isolates components residing on different application layers. Component interactions should be asynchronous and happen through the public interface of the component. This approach includes another SOA principle, which says it is advantageous to separate the interface containing available component services from the implementation.

The most common way to achieve asynchronous interaction among application components is using messaging queues. Built-in queue services of cloud providers offer concurrent manipulations, high availability and resilience, moreover a well-documented service API for cloud applications. Messaging queues usually does not support transactional reads, hence instead of in-order, exactly-once delivery of messages, only at-least-once is guaranteed. Transactional message reads would require locking mechanisms, which are hard to implement in an elastic cloud environment without significant loss in performance. Further advantage of using queues is that the component which enqueues a message has no knowledge of the component that processes the message. This allows to exist more processing components that can work parallel on different messages, thereby increasing the overall performance. The ability to add or remove components to adjust the capacity of the system is the key point of scalability.

Running more instances of the same component enables parallelization to a great extent. In the application design phase the processes should be exhaustively analyzed and designed with care to utilize the powerful possibilities of parallel execution.

Preparing for ephemeral component lifecycle and statelessness

Application components can be considered volatile as their lifetime is not fixed. On the contrary, component instances might run for a variable amount of time, they can be started or stopped due to scaling operations, hardware or software failures or automated decisions of the cloud management system. The application logic inside a component should be designed and implemented to be aware of this unpredictable running time. To achieve this, components need to be as stateless as possible; they should not store stateful information about sessions. In a cloud application, where several instances of functionally equal components can run, it cannot be guaranteed that the user will always be directed to the same instance on which her session started. Though there are scenarios when statefulness is inevitable, so common cloud platforms usually provide methods for developing such components. In this case, the load balancers and the cloud management environment must be aware of stateful components to ensure users are

routed to the appropriate component, therefore sessions shall stay intact. To provide better scalability and keep statefulness as well, state replication solutions can be utilized.

Stateful user context data should never be kept in the memory of the specific instance, it should always be stored to and retrieved from persistent data storage. Thus when an instance handling a specific user session crashes or it is being terminated, another working instance of the same component can take over its place and keep the session alive. Storing and reading session states require a large number of data storage operations, but cloud storage solutions are designed to handle and serve concurrent queries.

Persistent dynamic and static data storage

Data used by an application can be divided to two basic classes; dynamic data is the information that is frequently used for processes and computations, therefore always changes. The other class, static data includes a variation of constantly stored files with a varying size range, which might need frequent, concurrent accessing, but their content seldom changes. Handling dynamic and static data demands different approaches to be able to reach optimum performance.

For dynamic data cloud providers offer a wide palette of available solutions from a simple binary bucket to relational databases with transactions and replication. The type of the data determines which service should be used. To speed up application processes, especially the read-intensive ones, a distributed caching solution might be adapted. Data caching components can exist within the application or in a separate tier of components depending on the size and data need of the application. When sizing the caching component instance, one must be aware that caching uses less CPU but it demands a large available memory to be able to serve requests rapidly. Available caching solutions allow to have multiple cache server instances configured to distribute cached data amongst themselves, thus guaranteeing high availability and fault tolerance of data. Time-to-live (TTL) values of cached objects shall be determined cautiously to prevent unnecessary memory allocation.

In the cloud content delivery networks (CDN) offers the most flexible and efficient way for storing static data and making it globally accessible for application users [13]. Be it multimedia files of a web application or a thin client needed to access services of the cloud application, such static data is best stored in CDN. These files might be large and need to be delivered very frequently to users, so they do not fit into common cloud storage. Moving static data files to CDNs takes off the burden of resource consumption and network traffic that serving these files would take. CDNs are designed to provide highly available storage and vast bandwidth access independent of the consumer's geographic location.

Security aspects

Services hosted in the cloud are only accessible through the public network, so securing application processes and data is a very important design consideration. It is advisable to allow connections only through SSL and to apply data ciphering methods when transmitting sensitive information. Providers offer built-in identity management services which ease user authentication and authorization. Such user management solutions can be seamlessly integrated into cloud applications.

4 Overview of leading cloud platforms

Thanks to the rapid spread of cloud computing, a colorful palette of cloud services can be found on the online market. I do not intend to present the whole scale of available products, so I chose the three most significant providers which might be considered as the flagships of cloud computing. Their strong fundamentals and rich features make them outstanding, and there can be found several smaller providers, which they form the base of.

4.1 Amazon Web Services

Amazon has the largest IaaS based cloud environment. They provide a great variety of services in connection with the cloud, such as EC2, S3, EBS, CloudWatch, and this enumeration could be much longer. They offer solutions to many scenarios in all the common fields of information technology.

4.1.1 Common features and services

The number one service of AWS is the Elastic Compute Cloud (EC2), which provides a highly flexible and scalable computing cloud for its consumers. It has a well-documented web service API coming with available configuration and administrative tools that enable a straightforward and dynamic cloud management. In EC2 virtual machine instances are regarded as the elementary computing units. These instances can be launched out of hundreds of readily available Amazon Machine Images (AMIs [31]) containing different operating systems and installed software or a custom AMI created and preconfigured by the user. Amazon offers a pay-per-use pricing model based on the computing hours of running instances. The owners of the instances get full control over the virtual machines. The lifecycle of an instance lasts from launching to termination, and between these two, it can be rebooted any number of times.

The current state and utilization of instances can be observed on the web-based AWS Management Console. Instance performance and resource usage can be monitored in great detail with the CloudWatch service, which collects and displays values of built-in or even user custom performance counters. Statistics and various graphs can be generated from the metrics. CloudWatch enables to create alarms based on instance metrics. Alarms can send notifications via Amazon's Simple Notification Service (SNS), which provides an efficient and flexible publish-subscribe messaging model for sending notifications from the cloud. Activated alarms might also trigger auto-scaling actions that will be unfolded in the next section.

EC2 instances must belong to one or more security groups, which are practically virtual software firewalls keeping the gate of the instances' network communications. By default a security group rejects all incoming connections, the user has to open the necessary ports manually. Amazon provides Elastic Load Balancer (ELB) for equal distribution of incoming HTTP or TCP connections across multiple EC2 instances. ELB is truly elastic for it automatically adjusts its own capacity according to the current load. Instance health is periodically monitored by the ELB and its distribution algorithm also takes the current health status of the instances into

consideration. ELB supports secure communications and as a gateway to all incoming connections, it takes over the resource and time consuming work of message decryption.

For persistent data storage Amazon gives its users many possibilities. Elastic Block Storage (EBS) provides highly reliable volumes that can be used as boot partitions of EC2 instances holding AMIs or directly attached to running instances as standard block devices. EBS volumes are the optimal targets for storing data off-instances thus making it independent of instance lifecycle. As the CloudWatch service expands to all cloud features, it covers EBS too by providing metrics of the status and I/O operations of volumes.

Point-in-time, consistent, incremental volume snapshots can be created of EBS volumes, then persisted to Simple Storage Service (S3) to achieve higher durability. S3 provides a highly reliable and secure cloud storage solution suitable for any use case. Objects identified by keys are stored in buckets, for their manipulation S3 offers standard REST and SOAP interfaces, and for distribution it provides HTTP, BitTorrent or custom protocols. S3 has many encryption mechanisms on client-side and server-side alike that allows users to cipher sensitive information before or during uploading to the cloud and provide automatic decryption upon data retrieval.

Amazon hosts a CDN service as well, called CloudFront, which provides an easy and efficient way to store and deliver static or streaming data. CloudFront is integrated with other AWS services; it webs the whole world to enable geography-aware, high bandwidth content delivery.

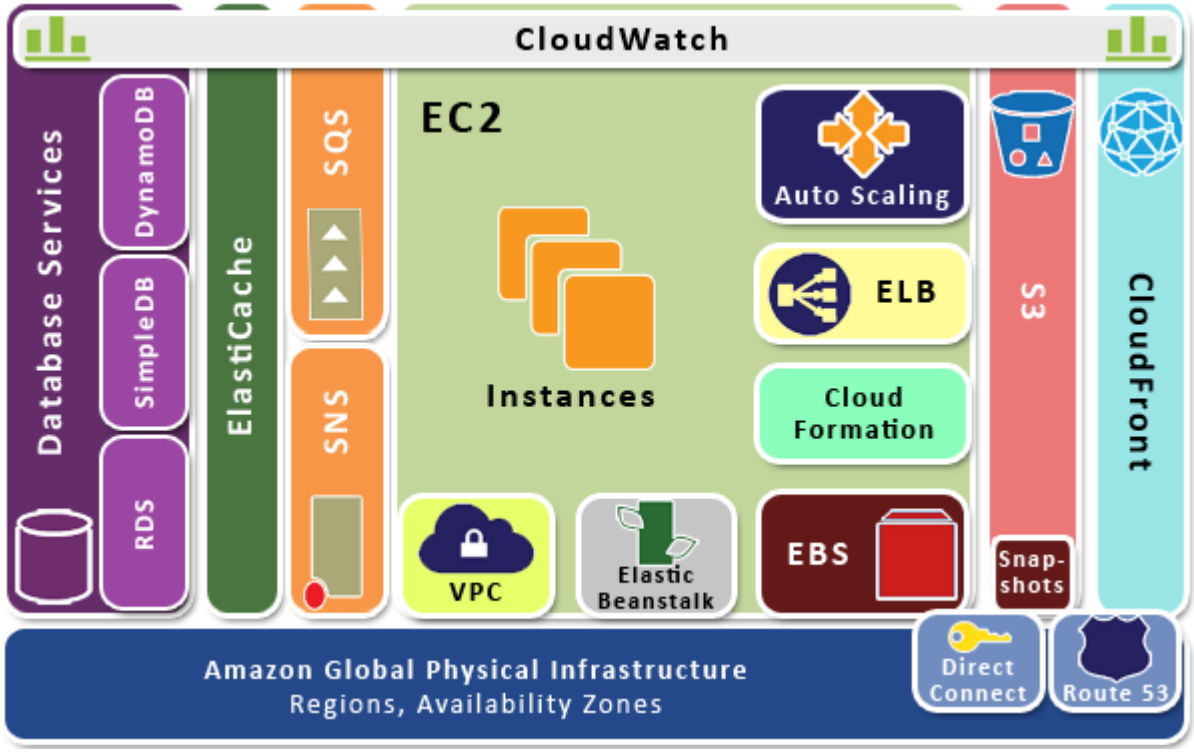

Figure 4-1. - Amazon Web Services architectural overview

Amazon's dynamic data handling and database solutions cover relational database management (RDS service), NoSQL services (SimpleDB, DynamoDB) and a distributed caching service (ElastiCache). To improve cloud application performance, ElastiCache enables to set up a cache cluster containing several nodes. ElastiCache service is protocol-compliant with the market leading caching product, memcached, so existing implementations and tools can be seamlessly integrated with ElastiCache. Amazon RDS provides a relational cloud database system based on MySQL or Oracle platform chosen by the user. With RDS special database instances can be launched in which a familiar database environment is accessible even for remote manipulation with conventional database management tools. RDS ensures data security with automated backups, user invoked snapshots, and replication mechanisms for MySQL databases. RDS allows vertical scaling on-the-fly, provisioning more resources at user request. SimpleDB offers a simple NoSQL database service for creating and storing multiple data sets, querying and retrieving data based on user-defined attributes. For larger datasets, when consistency and performance are essential, it is suggested to use DynamoDB, which is designed to host powerful database services with a high degree of automation and scalability.

Amazon's enormous infrastructure requires immense network capabilities to be able to serve the great number of user infrastructures and applications hosted within them. Amazon spreads services globally currently using eight regions, three in the USA, one in the EU, two in East Asia, one in South America, and one exclusively reserved for the US Government. Regions are divided to a certain number of Availability Zones (AZs) depending on the density of consumers in the particular geographic region. AZs are housed on physically distinct, independent infrastructures, and they are built to be highly reliable and resistant to all kinds of hardware or software failure. AZs are independent in themselves, so an AZ struck by occasional failures or natural disasters do not affect the performance of other AZs.

AWS includes several services regarding networking. Their scalable naming system service is called Route 53. The Virtual Private Cloud (VPC) service allows users to create a private, isolated network topology within the Amazon cloud and gives full control of network configuration including IP addresses, public and private subnets or routing. Placing databases or background applications in a separate backend virtual network that is not accessible over the Internet, further increases security. VPC has the ability to integrate with existing on-premises Virtual Private Networks, therefore it is possible for a company to use the cloud as an extension of their local infrastructure. AWS Direct Connect Service makes it possible to establish dedicated network connections between the local and the cloud infrastructure. Direct Connect guarantees a consistent, high network performance and security.

Amazon also launched a PaaS cloud service, called Elastic Beanstalk, which opens the possibility for PHP and Java developers to easily deploy and manage cloud applications that are able to utilize all the other AWS services. Elastic Beanstalk takes care of capacity provisioning, load balancing, auto-scaling and application monitoring with the least human intervention, but still gives developers full control of environment configuration and deployment procedures. Amazon provides a development toolkit for Eclipse, which includes the AWS SDK for Java and the necessary tools for deployment and debugging.

CloudFormation service enables developers and system administrators to describe, then launch complete hardware and software stacks along with all the related resources. It works based on declarative templates that contain the resources along with their parameters and the dependencies amongst them. CloudFormation supports controlled and consistent versioning and updating of these cloud stacks. Amazon offers several readily available stack templates that cover common scenarios.

Amazon offers several other services, which go beyond the scope of this work, but as for all the discussed services, elaborate developer and administration guides can be found on the official Amazon AWS website[5].

4.1.2 Auto Scaling service

Auto Scaling makes it possible to scale a group of EC2 instances dynamically up or down according to the current load of the fleet [14]. By using Auto Scaling, system administrators can prepare the hosted application for demand spikes, so its capacity adapts instead of crushing under the pressure. On the other hand, Auto Scaling helps in keeping the costs low by scaling down - withdrawing running but unused resources when demand decreases. Scheduled scaling is also possible if the application usage is roughly predictable. Unfortunately, Auto Scaling service does not yet have a web-based graphical interface inside the AWS Management Console, its features are available using the Auto Scaling Command Line Tools [30].

The core concept of Auto Scaling is that EC2 instances can be categorized into Auto Scaling Groups (ASGs). Instances grouped together share similar characteristics, they might be hosting the same application stack, so they can be treated the same. The main parameters of an ASG that need to be provided upon creation are the following:

- *Name: a unique name of the group used for identification.*
- *Availability Zones: a list of AZs where the instances of the ASG can be launched.*
- *Launch Configuration: a launch configuration needs to be created beforehand, which contains several attributes of the instances to be launched during scale up operations. The important parameters of launch configurations will be discussed later.*
- *Load balancers: a list of ELBs can be enumerated, under which the instances will belong.*
- *Desired capacity: the number of instances that should be running in the ASG.*
- *Minimum size: the minimum number of running instances.*
- *Maximum size: the maximum number of running instances.*
- *Default cooldown: the amount of time, in seconds, that need to pass after a scaling operation before allowing another scaling activity. As its name states, this cooldown period allows the scaled instance fleet to normalize its performance after scaling up or down. During cooldown, Auto Scaling rules are not being evaluated, this period allows the effects of a scaling operation become visible in the performance metrics. By setting the appropriate period length, one can prevent unnecessarily repeated and oscillating scaling activities. Repeated scaling may occur when the cooldown period is too short. After scaling up due to an increase in demand the group needs some time to launch new instances that can take over some of the load. If the next rule*

[5] http://aws.amazon.com/

evaluation comes too early, the good impact of the capacity growth might not be noticeable on the metrics values, so the scaler will initiate another scale up activity. This flow of events applies to the scale down operations as well. Setting the cooldown period too long causes that the group loses its capability of instantaneous reaction to the actual demand.

- *Health check type: with this property, it can be determined that the instance health status coming from the EC2 environment or from the Elastic Load Balancer should be taken into account when monitoring instance health across the ASG.*
- *Health check grace period: this amount of seconds needs to pass when an instance is launched before its health state monitoring is started.*

One of the most important parameters of an Auto Scaling Group is the Launch Configuration (LC) that holds all the required attributes necessary to launch new instances. In a custom Launch Configuration, the following information can be provided:

- *Name: a unique name of the LC.*
- *Instance type: the API name*[6] *of the type of which new instances should be launched. The instance type defines the amount of virtual resources allocated for the instance, such as memory, CPU, storage, or the allowed I/O performance.*
- *Image identifier: the unique ID of the Amazon Machine Image (AMI) that should be used for instance creation. As it was discussed earlier, available community or user custom AMIs refer to virtual images containing the operating system and the pre-installed software stack.*
- *Key name: the identifying name of a security key pair that belongs to the user subscription; it ensures authenticity for newly created instances.*
- *Security Groups: a list of Security Groups with which to associate created instances.*
- *User data: data provided in this parameter will be available to the launched instances. This data can hold configuration values or even a script, which will be executed on startup of the instance's operating system. User data scripts come very handy when implementing automated configuration management solutions like the one described in a later chapter.*

Once an ASG with the proper Launch Configuration is created, the definition of Auto Scaling triggers becomes possible for the group. A trigger combines two AWS features; it is made up of a CloudWatch alarm set to a metric and an Auto Scaling policy that defines the Auto Scaling activity need to be taken when the alarm threshold is crossed. Triggers usually stand in symmetric pairs, one for scaling up and another for scaling down based on the values of a given CloudWatch metric.

CloudWatch alarms can be set up to track the value of a metric. An alarm has three states: *OK*, *Alarm*, *InsufficientData*. It is possible to register actions to all alarm state transitions. When defining an alarm, one must provide the following parameters to customize its details:

- *Name: a unique name for the alarm.*
- *Metric name: the name of the chosen metric that should be monitored. There is a great selection of metrics provided by CloudWatch to choose from. Aggregated metrics, such as CPU utilization, disk read-write operations, network usage of an entire Auto Scaling Group is also*

[6] http://www.ec2instances.info/

available, but it is possible to track the performance of a single instance. Besides EC2 metrics, CloudWatch offers performance counters for ELBs, RDS database operations, EBS volume usage, SQS queue length, and a few other features.

- *Namespace: the namespace of the metric associated with the alarm (possible values: EC2, ELB, RDS, EBS, etc).*
- *Dimensions: this optional parameter defines the list of names or identifiers of the entities of which metric should be tracked. It can hold the identifier of an instance, a name of an ASG, etc.*
- *Unit: the basic unit for the alarm's metric.*
- *Statistic: the statistic function applied to metric values. Available statistics: SampleCount, Average, Sum, Minimum, Maximum.*
- *Threshold: the value against which the metric statistic is compared.*
- *Comparison operator: the arithmetic relation operator for comparing the metric statistic to the threshold.* $Statistic(\ Metric\ )\ <Comparison\ operator>\ Threshold$
- *Period: the time in seconds defining how long the statistic to be applied for the metric.*
- *Evaluation periods: the number of periods over which the statistic and the threshold need to be compared.*
- *Alarm actions: the list of actions to take when the alarm sets off. Two action types are supported: invoking an Auto Scaling policy and publishing to an SNS topic. Actions are identified by the unique Amazon Resource Number (ARN).*

CloudWatch alarms are integrated into the web Management Console, which makes alarm creation simple and straightforward.

Auto Scaling policies contain instructions about how to respond to alarms. Policies can be created with the Command Line Tools along with the following necessary parameters:

- *Name: the descriptive name of the Auto Scaling policy.*
- *Auto Scaling Group name: the name of the ASG to which the policy belongs.*
- *Adjustment type: specifies the type of the desired adjustment whether it is a change in capacity, or a fixed exact capacity. Change in capacity can be numeric or percental.*
- *Scaling adjustment: the integer value of the adjustment. The adjustment type parameter determines of its interpretation. It can be a positive or negative number or a percentage.*
- *Cooldown: the applicable cooldown period length for the current policy.*

Besides policy-based scaling, Auto Scaling enables administrators to create scheduled group update actions that can set the size and desired capacity of an ASG. Such scheduled actions can be specified with a time interval and a recurrence parameter passed in standard Unix cron syntax format.

Auto Scaling can be used for maintaining a desired level of capacity to ensure a stable performance. Maintaining the current capacity level is achieved by health monitoring. Health status updates can be received from the EC2 or an ELB, moreover it is possible to modify instance health status by a proper API method call. The ability of setting instance health status manually enables integration with other health monitoring solutions. When an instance gets marked unhealthy, it is immediately scheduled for replacement.

4.2 Windows Azure

With Azure, Microsoft tries to break the dominance of Amazon in the IaaS sector. Although it offers less infrastructure services than its opponent, it provides a stronger PaaS computing service that rests on the firm base of the .NET Framework. The ease of development and the built-in cloud service components available in the SDK make Azure the dominant amongst PaaS providers. This thesis focuses on the PaaS features of Azure.

Developing for Azure does not require special skills for regular .NET programmers, they just have to adapt to the unconventional but well-documented environment of the cloud and learn the capabilities of Azure services. Azure supports all the .NET Framework languages plus it enables deploying and running applications written in Java, PHP or Node.js. Cloud applications are being deployed then executed in virtual machines running Windows Server operating system. The Azure programming model [10] emphasizes the importance of the following three areas.

- *Administration*

  In contrast to the IaaS service model, Azure is built to handle most of the administrative tasks, such as installing updates and security patches to the operating system and different software. By dealing with administration automatically, Azure reduces the effort and cost of maintaining the application environment.

- *Availability*

  The Azure platform makes it possible for an application to be highly reliable, to have nearly zero downtime. In non-cloud environments, applications might become unavailable due to software or hardware failures, application or OS upgrades, and many other reasons. A cloud application hosted in Azure can avoid these downtimes, because of redundancy.

- *Scalability*

  Applications running on the Azure platform are scalable by design. The Azure programming model applies all the cloud design considerations to enable developing applications with high scalability.

4.2.1 Common features and services

An Azure application running on the compute platform is built from one or more roles. A role is a separable logical part of an application that has a well-defined task. The role contains the relevant code and configuration information needed for attending the particular task. Azure offers two basic role types that cover the majority of cloud application use cases. A web role can contain the Internet-facing part of an application that provides an interactive web interface for the clients. Web roles are hosted in Microsoft's Internet Information Services (IIS). Worker roles provide more general functionality, they may contain any kind of code that runs processes in the background. The descriptions and configuration information of the roles belonging to the application need to be specified in a service definition file. The roles are being run in the cloud in multiple role instances. Azure suggests launching at least two instances of each role to guarantee the availability of the role's functionality.

Behind the scenes Azure's Fabric Controller (FC) pulls the strings by transparently managing the virtualized resources and VMs of role instances. Upon starting the roles, the FC automatically launches the configured number and type of virtual machines as instances, and starts running the given role's executable. When the role is successfully initiated and begins running, the FC starts monitoring the role instances. If an instance suffers any kind of software failure due to an application exception or operating system error, or hardware failure in the cloud environment, the FC immediately replaces the unhealthy instance with a new one to maintain the configured capacity. To evade severe consequences of hardware failures, the FC places different instances of the same role into different fault domains. A coherent set of hardware resources makes up a fault domain, that share a common single point of failure. Running the role instances in separate fault domains increases reliability and fault tolerance of the entire application.

The Fabric Controller is also responsible for installing patches and upgrades of the operating system and other software, such as the .NET framework or the IIS server on the instance virtual machines. Some of these upgrading operations require rebooting the virtual machine. To avoid application downtime caused by instance updates, the FC puts role instances into different update domains. With the usage of update domains, it also becomes possible to deploy a new version of the application with no downtime. The FC controls the application update process, it goes through the update domains one by one, stops the instances running in that domain, deploys the new code version, then starts new instances based on the updated role. By proceeding so, the FC guarantees that a situation when no instances of a role are running cannot come up.

For storing data persistently, outside the instance virtual machines, Azure offers three data management cloud services: Blobs, Tables and SQL Azure. Blobs are ideal for storing large, unstructured binary data. Azure Drives service makes it possible to attach Blobs containing a Windows file system to instance virtual machines as mounted disks. Tables provide a NoSQL data storage solution for storing and fast accessing sizable amounts of typed data organized into key-identified groups. The name can be misleading, because Tables are not part of a relational database, they simply allow applications to store persistent and typed information. Complex SQL queries and table join operations are not available; the stored data can only be logically grouped and retrieved by the group's identifier. Blobs and Tables ensure high availability and fault tolerance as stored data is automatically, transparently for the user being replicated across three different locations inside the Azure cloud.

SQL Azure is the cloud version of the traditional SQL Server relational database management system [16] bundled with such features as transaction management, consistent data access concurrency, the T-SQL query language, etc. A SQL Azure database can be accessed through many well-known data management interfaces and administered by the conventional tools. SQL Azure is a PaaS database service in itself, because it takes care of managing the underlying hardware infrastructure and keeping the SQL Server up-to-date. It hides administrative functions from users to prevent unnecessary interference. The cloud version of the SQL Server Reporting Services is called SQL Azure Reporting, which allows creating in-depth business analytics reports on the stored data. SQL Azure offers the Data Sync service for synchronizing

SQL Azure and on-premises, non-cloud databases. SQL Azure automatically takes care of database replication to increase database reliability and data consistency.

Azure provides two integrated services for message-based interactions between two or more role instances. Queues are the simplest way of asynchronous communication, they work the usual way. A role instance puts a message into the queue and another takes it out and processes the message. Using queues enhances the application's ability to scale by decoupling the components. A message can be processed by any role instance, without the publisher of the message having any knowledge of the message's afterlife. The Azure Service Bus provides a consistent and predictable queue service and besides that a more general flow of interactions among instances or even applications residing outside the cloud. Service Bus follows the publish-subscribe messaging model by providing topics through which message publishers and its consumers, the subscribers can continue one-to-many communication. Service Bus makes it possible for non-cloud WCF-based applications to register endpoints in order to publish services that cloud or non-cloud clients can directly invoke. The detailed differences in capabilities and features of the simple Queue and the Service Bus are beyond the scope of this work (for in-depth comparison see [18]).

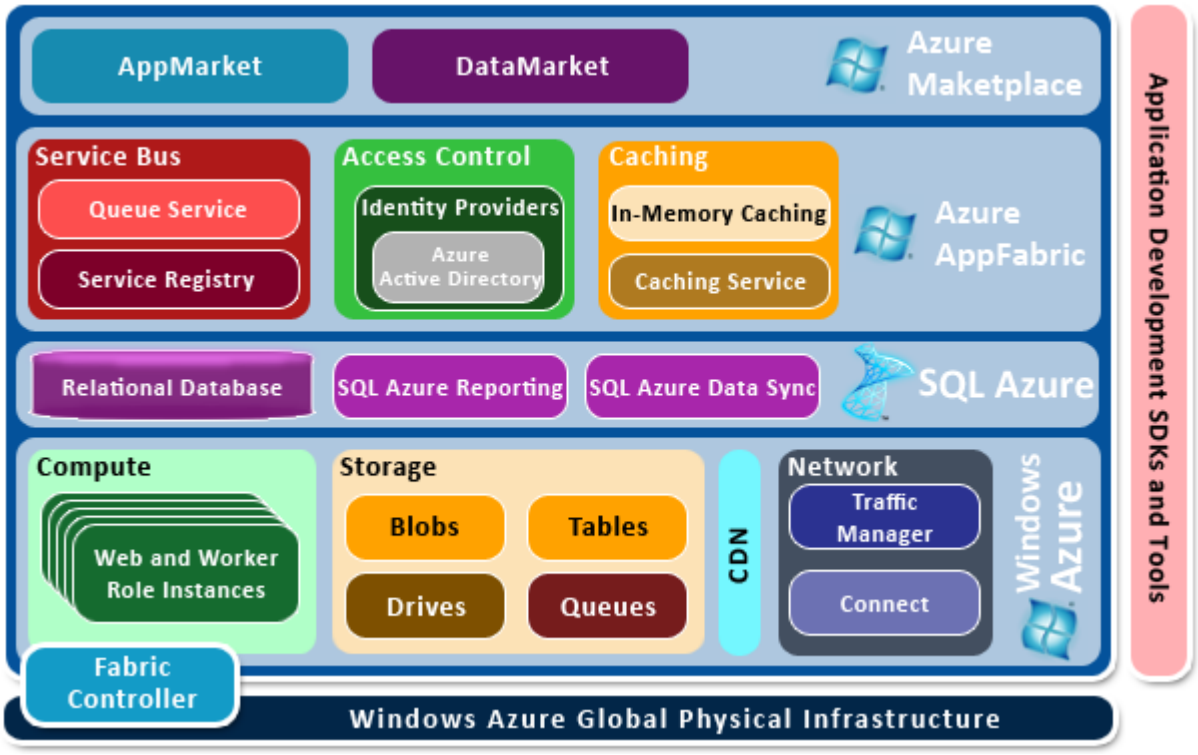

Figure 4-2. - Windows Azure architectural overview

Blobs storing static information that needs to be accessed often and with high bandwidth can be copied to Azure's CDN. Azure automatically moves the copy of the blob's content to the closest location possible to the data consumers.

Azure offers in-memory caching to improve application performance, so role instances can have a local cache for storing the most frequently accessed data. Caching service writes the contents

of local caches to the central distributed cache system inside the platform. Web roles running an ASP.NET application may use this way of caching to store session data.

Cloud applications can use a very flexible and easily integrable service regarding access control and identity management. For Windows-based corporate environments, Azure Active Directory provides a solid foundation for storing and managing users, but the Azure platform allows integration with other well-known Identity Providers (IdP), such as Windows Live ID, Google or Facebook accounts. The Access Control Service (ACS) as a mediator, retrieves and validates the various incoming IdP tokens, then translates them to a common format. With ACS, the administrators of cloud applications are able to define rules for transforming IdP tokens to the common Access Control token format.

Windows Azure has six datacenters running today, two in the US, two in Europe, two in Asia. For cloud applications distributed globally, it is possible to run role instances in many regions. Azure's Traffic Manager service is responsible for routing incoming connections to the appropriate instances based on rules defined by the application administrator. Azure's Connect service makes it possible to establish secure IPsec connections between cloud and on-premises infrastructures.

Azure introduces Marketplace for publishing Azure applications as a SaaS service and for making datasets available to the public. Potential customers can search the AppMarket and the DataMarket and purchase or subscribe to available applications or datasets, such as demographic, financial or legal information vaults.

Azure provides an SDK and many deployment tools to allow application developers to utilize the wide service palette described above. The necessary libraries and tools for developing, simulating and deploying Azure applications can be easily integrated with Visual Studio 2010 and Eclipse as well.

4.2.2 Autoscaling Application Block (WASABi)

WASABi is included in the Enterprise Library 5.0 Integration Pack for Windows Azure along with other useful application blocks, tools and PowerShell cmdlets. It allows defining how Azure applications should respond to varying load by horizontal scaling.

WASABi is designed to encapsulate the logic for automatic scaling, so it requires minimal changes to the scaled application. Another important principle of the Autoscaling Application Block, that its configuration and behavioral properties are stored in external declarative files, which can be modified without having to restart or redeploy the Block or the target application. WASABi can be hosted inside the scaled cloud application as a separate role, or in an individual application in the cloud or on-premises. WASABi provides many possibilities for extending its functionality [19].

WASABi provides the two following techniques with which application capacity can be adjusted to the current demand. These are not mutually exclusive; they can both be applied in hybrid auto-scaling solutions.

- *Instance scaling*

  WASABi can dynamically increase or decrease the number of running role instances. Using automatic instance scaling helps avoiding under-provisioning during traffic growths and over-provisioning when demand is low, therefore optimizes the application's performance and costs. It is advisable to define the maximum number of concurrently running instances to prevent launching too many instances due to application or configuration errors or unpredictable load deviances, and the minimum number to make sure that the desired capacity is available at all times. In case of horizontal scaling, the launch time of a new role instance must be taken into account when configuring the Autoscaling Application Block.

- *Throttling*

  WASABi provides this technique for applications that are harder to scale. Throttling allows developers to define operation modes of their applications appropriate to certain load levels or daily periods. When the demand is low or in the middle of the night the application might change to a mode that allows longer, resource intensive processing tasks or backup operations. Under heavy load or the predicted rush hours the application might be set to another mode when only the critical functions are available to keep up steady performance. Throttling can be used as an efficient complementary service of instance scaling. If a demand spike occurs, launching new instances can take some time, meanwhile the application functionality can be degraded by changing the mode of operation.

Rules and their associated actions determine the scaling behavior of an application. WASABi has two basic rule types:

- *Constraint rules*

  With constraint rules the lower and upper bound of the number of running role instances can be restricted. Constraint rules have precedence over reactive rules, and the priority among more constraint rules is determined by their rank attribute. Rule rank is an integer value, the higher number means higher priority at rule evaluation. Constraint rules are also beneficial for applications that have a predictable usage pattern in time, so WASABi allows assigning a time interval to each constraint rule indicating what time of day the rule should be in effect.

- *Reactive rules*

  As the name suggests, reactive rules lets applications to react to demand changes real-time. Reactive rules trigger actions when an aggregated value of a metric exceeds a threshold. WASABi supports the following measures to use as metrics: built-in performance counter values, Azure Queue length, role instance count and other custom metrics published by the application itself. Rule operands are the metric or counter values aggregated by a specific statistic function over a certain period of time. Reactive rules are usually paired, one for scaling up and another for scaling down based on the operand of the same metric. Reactive rules can trigger one or more of the following actions:

- instance scaling of the rule's target role,
- sending notifications,
- switching operation mode of a throttled application,
- applying modifications in a service configuration file,
- executing custom actions.

Among constraint rules, the assigned rank decides which to apply, but in case of reactive rules, WASABi resolves conflicting actions by an automatic reconciliation mechanism (for further details on this topic see [21]).

Using performance counter values in reactive rule operands require some modifications in the target role's code. The role must expose the necessary built-in or custom performance counter data to a special diagnostics table in the Azure storage service, so that the data collection process of the Autoscaling Application Block can access it.

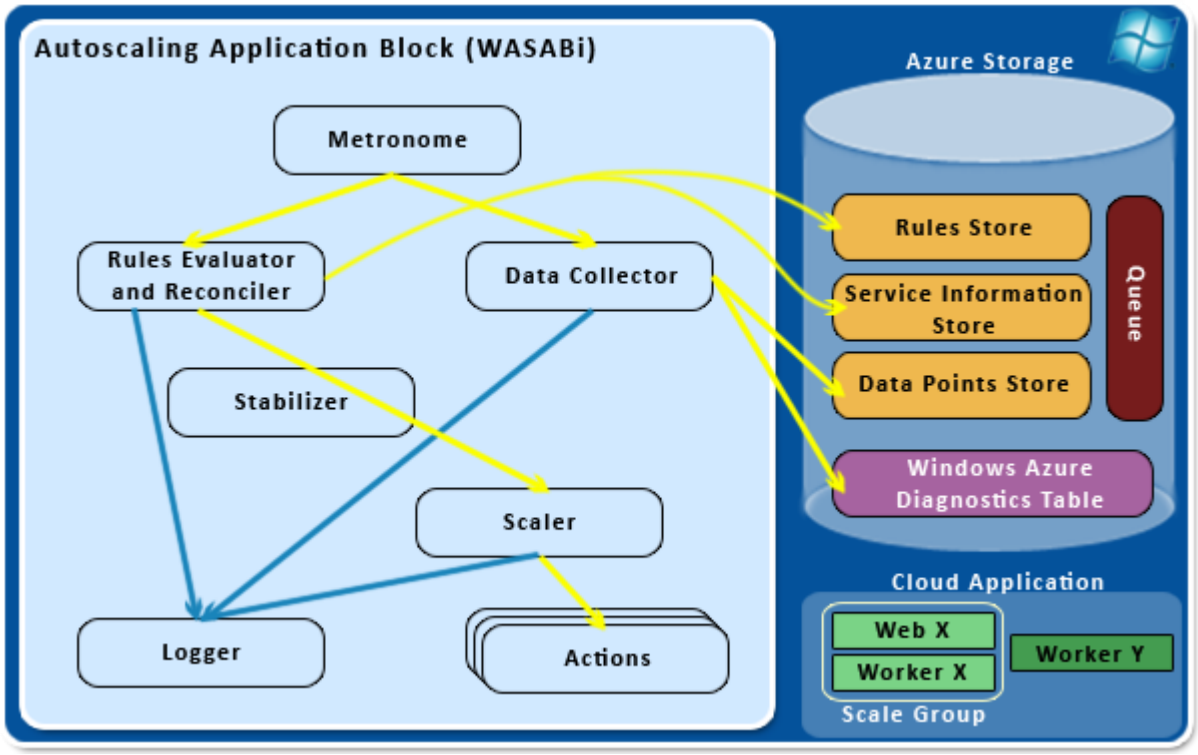

Figure 4-3. - WASABi architecture

WASABi stores all the auto-scaling rules in a dedicated Rules Store in the form of an XML file. The Rules Store can be located in a directory local to the WASABi component, or in a storage Blob in the cloud.

WASABi requires another XML file located in the Service Information Store, which contains all the necessary information on the application, such as roles, Azure subscription identifiers, storage account details, queue names, security certificate thumbprints, scale group definitions, or configuration parameters of the Stabilizer. Proper configuration information for these XML files will be revealed in a later chapter.

Roles with logical cohesion between them can be grouped into scale groups. Scale groups may stand as the targets of scaling rules, so the grouped roles are handled together. Scale groups allow declaring the importance of grouped roles relative to each other by assigning ratios to the roles. The roles that need larger scaling might get a higher ratio value. When a group is scaled, the instance number increment multiplied by the role's ratio gives the resulting number of instances after the scaling operation.

The previously mentioned Stabilizer is a WASABi component that suspends scaling activities during the preset cooldown period to give the application time to utilize the newly started instances therefore preventing repeated scales or abnormal oscillations. It takes time to the Azure platform to launch or terminate instances, so some time needs to pass while the effects of capacity change show on performance counter values. The Stabilizer makes sure that no scaling actions may take place during the cooldown phase, so unnecessary repeated scaling operations cannot happen. The Stabilizer can be configured with setting the cooldown period for individual roles and both scaling up and down operations within the service information descriptor XML.

4.3 Google App Engine

Google App Engine (GAE) is Google's PaaS type cloud platform [26], which allows users to deploy, run and maintain applications written in Java (or other JVM-based, like JS, Ruby), Python or Go. Google has a vast global infrastructure and an undeniable experience in hosting highly available and scalable applications thanks to their many widespread SaaS services, like GMail, GDocs, GDrive, etc. Unlike the previous two providers, GAE does not give out the possibilities to configure scalability rules and actions; application scaling is entirely automated and handled by the platform.

4.3.1 Common features and services

GAE applications run in isolated, secure environments, so-called Sandboxes, which are independent of the underlying hardware and operating system. To achieve high reliability and other appealing qualities, Sandbox has many restrictions compared to traditional execution environments. Applications residing in Sandboxes are not allowed to write to the file system of the instance they are running on, all file operations need to happen through the GAE Datastore, which will be discussed shortly. Another limitation of Sandboxes is that their network interfaces are closed to the public, applications running inside them can only receive HTTP or HTTPS requests on standard ports; and outgoing communications are only allowed through available GAE services, such as URL Fetch. Applications cannot run constantly, they are being executed in response to web requests, queued or scheduled tasks, and they must return a result within a certain time.

Web requests have a strict 60 seconds response time limit, so they are not applicable for time and resource consuming background operations, which commonly emerge in most software architectures. To add the ability to handle such processes, GAE introduces the Task, which has a less strict 10 minutes execution time limit. Task execution might be initiated on schedule controlled and monitored by the Cron service. GAE offers two different types, Push and Pull Task Queues [28] for lining up Tasks in the background created by incoming web requests or while

performing other Tasks. Push Task Queues can only be used within the platform, they provide an efficient and easily implementable solution. Pull Task Queues allow task consumers coming from outside the platform as well. Task consumers can lease tasks from the Pull Queue and are responsible for deleting them after execution. The GAE platform orchestrates automatic scaling based on queue lengths.

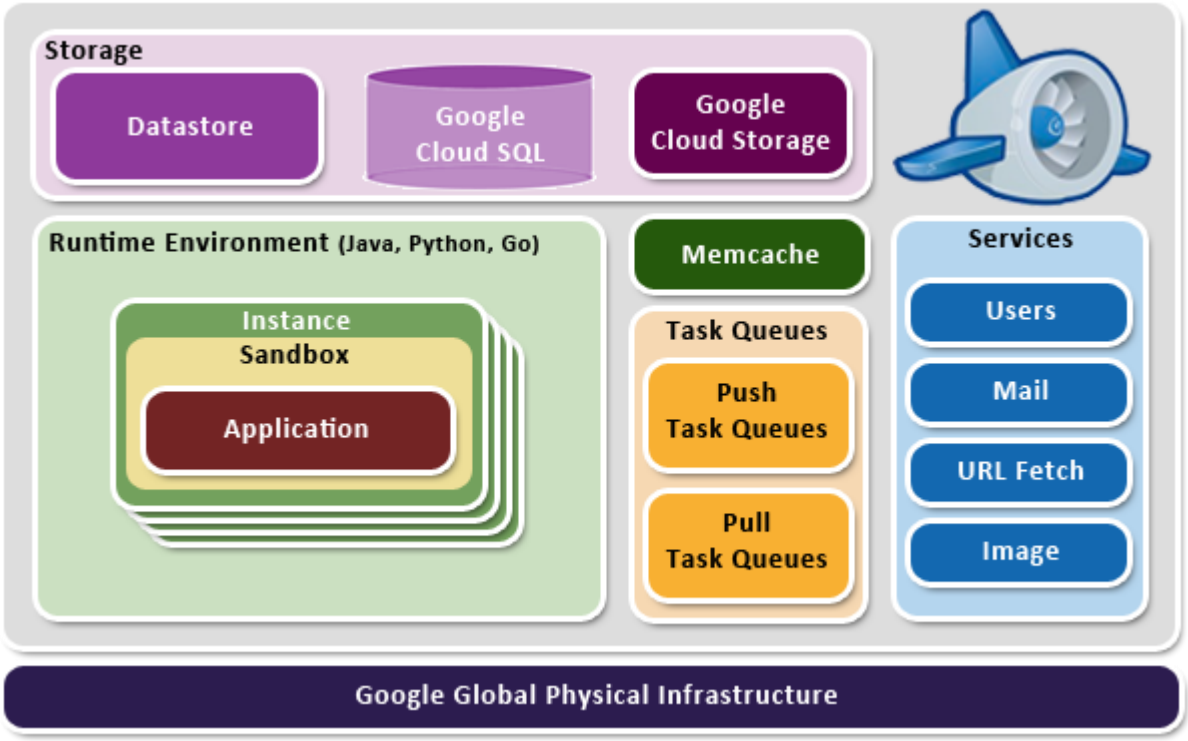

Figure 4-4. - Google App Engine architectural overview

GAE offers many possibilities regarding data storage. The GAE Datastore offers a distributed NoSQL database solution for storing and accessing objects attributed with sets of typed propertie. The Datastore provides a simple query interface, through which applications can assemble basic queries with filtering and sorting based on the property values of stored objects. The Datastore uses a replication mechanism to ensure high reliability and performance. Atomic transactions are also supported using optimistic concurrency control, and the Datastore guarantees data consistency. The Datastore is accessible via the standard JDO/JPA interfaces in GAE's Java Runtime Environment. GAE offers a fully managed, MySQL-based relational database system, called Google Cloud SQL. Besides these, Google provides Cloud Storage for storing and accessing static data in a fast and secure way.

Google gives cloud application developers a rich caching API in the form of the Memcache service. Memcache provides a high performance in-memory key-value cache that is available for all the instances. By storing frequently accessed data in the cache, the application need not turn to the Datastore or the Cloud SQL database, therefore its overall speed increases.

Applications built on the GAE platform can utilize many services using standard APIs. To simplify user management, applications can integrate the Google Account service. The Users API lets developers differentiate authenticated users and create applications that provide different

functionality based on user roles. With using Mail service, cloud applications can send individual or bulk e-mails with the assistance of the GMail infrastructure. GAE offers the URL Fetch service for outgoing network communication of applications. Using URL Fetch, applications are able to invoke remote web services or access resources outside the platform. GAE provides the Image service for automated image file manipulation and processing from application code.

5 Experimenting with dynamic scaling

The elaborately described scaling solutions of the Amazon and the Azure platform are being put in motion in the case studies of this chapter. The design process of scalable cloud applications, implementational and deployment details, configuring the scaling services, running benchmarks will be unfolded in high detail. I also highlight the importance of monitoring and the available solutions.

5.1 Scaling of web servers with Amazon Auto Scaling

I designed and implemented a scalable scenario for the IaaS platform of Amazon Web Services using the common web application stack that includes Apache, PHP and MySQL. I developed the PrimeFinder PHP application that offers a simple web interface for searching prime numbers in a user entered interval. The scalable part of the application is the web tier made up of a number of instances running web servers. These are grouped into an Auto Scaling Group that is placed beneath an Elastic Load Balancer that distributes the incoming HTTP connections. Empty Linux instances are being launched when the load increases, so they need to be properly orchestrated to be able to handle requests. Amazon offers a feature to automatically bootstrap the freshly launched instances. For further automated installations and configuration, I chose *Chef*, which is a centrally managed configuration management system. By using Chef, it becomes possible to deploy and configure web servers without any user intervention based on simply editable rule sets, called recipes. The load balanced Auto Scaling Group of web servers dynamically adjusts its own size according to the current density of requests, the launched instances automatically get prepared to work by bootstrap and Chef configuration.

5.1.1 Designing the PrimeFinder application

I designed this simple PHP application especially to demonstrate scalability. It uses the computing capacity of the web server to find all prime numbers between an interval supplied by the user. This method is not common; the resource intensive logic should be delegated to a separate application tier, the business logic tier, which should be dedicated to this task and should receive the commands from the web tier through some message-oriented middleware solution, such as a simple queue. For simplicity, I decided to implement the business logic tier within the web tier, which will be scalable, and extend the application with an additional database tier containing a single database server instance.

The database server is running MySQL, which holds the data content necessary to the user account management subsystem of the application that is responsible for secure storage of passwords, user authentication and registration. The only table of the database consists of three columns: a numeric *UserID* field that also acts as primary key, the *Username*, and the *Password* field, which stores the MD5 hash fingerprint of the user's password. The database connection parameters - address, user, password, default database name - are placed in a separate PHP file, which allows modifications independent from the application. The growth of load might affect the database server, but the scalability of the database tier could be achieved by applying MySQL clustering techniques, which fall beyond the scope of this thesis.

The prime finding algorithm is very primitive for its main purpose is being CPU intensive. It goes through each number of the given interval and determines whether it is a prime by trying to find a divisor other than one or the number itself. The application is measuring the elapsed time during the prime search.

5.1.2 Configuring the necessary Web Services

The case study was realized on a free usage tier Amazon AWS account, which offers several hundreds of instance hours and the utilization of most services, such as storage (S3, EBS), Auto Scaling, Elastic Load Balancing, CloudWatch, etc. free of charge. The services are configurable via the interactive web-based Management Console or downloadable command-line toolkits.

Nearly all services are accessible via the Management Console, but Auto Scaling does not have a web interface, therefore it can only be set up using the Auto Scaling Command Line Tool[7]. All service API tools require Java Runtime Environment installed on the management workstation. Remote API calls are signed with an X.509 certificate or the customer's Secret Access Key. To further enhance security and guarantee confidentiality, the calls can be encrypted in transit using SSL. Before start using the API tools, it is advisable to verify the identity of the software publisher and the integrity of the tools to avoid security breaches. All API tools are relying on the following environment variables, which must be defined properly:

- *EC2_HOME: contains the path to the API tools. Additionally, the bin subfolder of the tools' shall be added to the global PATH variable in order to be able to access the tools executables from any working directory.*
- *EC2_CERT and EC2_PRIVATE_KEY: the necessary certificate and key file can be generated and downloaded from the AWS website. After exporting their paths as environment variables, these files are used for user authentication and signing of remote API calls.*
- *EC2_URL: optionally, this variable can hold the address of the service endpoint of the region, where the user's cloudware is running.*

To determine whether the settings are correct, the ec2-describe-instances command of the EC2 API Tools can be called, which lists the instances belonging to the user's account along with their attributes, such as the machine image (AMI), public DNS name, current status, etc. The EC2 API command line toolkit consists of several commands useful for handling and managing instances (for a complete list of available commands, refer to [30]). Launching an instance can be done using the following parameterized command:

```
ec2-run-instances    "ami-8d5069f9"    -t "t1.micro"
                     --region "eu-west-1"    --availability-zone "eu-west-1a"
                     -g "default"    -k "mykeypair"
```

The first parameter states the unique identifier of the virtual machine image that should be run on the instance. Amazon offers several hundreds of AMIs that can be launched instantly, the majority of them are available for free, created by the community, but there are a few images published by official providers. The AMI identifier determines the architecture (32 or 64-bit)

[7] http://aws.amazon.com/developertools/2535

and the operating system. It is important to notice that the same image can have different identifiers in different regions. For a filterable list of available AMIs, see [31]. The -t option stands for the instance type. Amazon EC2 offers 13 types with varying number of compute units, amount of memory, size of instance storage, I/O performance, and of course cost. The second row of parameters state the region and the availability zone, where the instance will run. The -g option specifies the name of the Security Group to which the instance belongs. Security Groups are like virtual packet filtering firewalls with simple rules. By default, it completely isolates the instance, disallowing any incoming or outgoing network connections. The user can create rules on the Management Console, which allow traffic through specific ports. In my scenario, I formed two Security Groups, one for the single database server communicating on port 3306, and another for the web tier instances with port 80 open for HTTP connections. The -k option refers to a key pair in the cloud belonging to the user account, which can be used for authenticating direct requests to the instance, such as an SSH connection.

The AMI I chose for all tier instances is a 32-bit *Ubuntu Oneiric 11.10* stored on the Elastic Block Storage (EBS). If an instance is launched based on this image, the empty OS gets ready to use within a minute. The status can be observed real-time on the Management Console. I manually installed the necessary packages, started the services, and configured the MySQL server on a single instance. Obviously, this cannot be done in the case of web tier instances for they must be deployed automatically when the Auto Scaling service launches them. Amazon EC2 offers the user data script feature with which the user can pass a bootstrap shell script to the freshly launched instance that executes right after boot-up. One way of automation would be to create a complex script, which installs packages and deals with all the configuration tasks. Writing this script is strenuous; furthermore, it is non-repeatable in case something fails. This is why I decided to use a configuration management system, Chef.

5.1.3 Brief introduction to Chef [32]

"Chef is an open-source systems integration framework built specifically for automating the cloud" - says the official Chef homepage[8]. Chef allows administrators to create a centralized configuration management system for their infrastructure and enables high levels of flexibility and automation. Chef is also platform-independent, the assembled configuration sets work the same way regardless of the host's architecture or operating system.

The architecture of Chef distinguishes three types of hosts in the infrastructure:

- *Server: the Chef Server forms the central storage base for configuration data. It stores information necessary to configure and deploy nodes. Chef Server can be installed separately and comes with a web-based management interface, but Opscode, the creator of Chef offers a Hosted Chef service. In case of Hosted Chef, the central data repository and the server functions are housed by the reliable and scalable infrastructure of Opscode. Hosted Chef can be considered a PaaS-like service, as it runs in the cloud and takes off the burdens of Chef Server administration and maintenance.*

[8] http://www.opscode.com/chef/

- *Node: the hosts being configured by Chef are called nodes. The chef-client application is running on the nodes, which is responsible for interacting with the Chef Server.*
- *Workstation: a host for managing the Chef infrastructure, typically the workstation of the system administrator. It stores a local copy of the configuration data repository and has the functionality - in the form of the tool, knife - necessary to send commands to the Server. Using knife, it is also possible to connect to nodes via SSH from the workstation.*

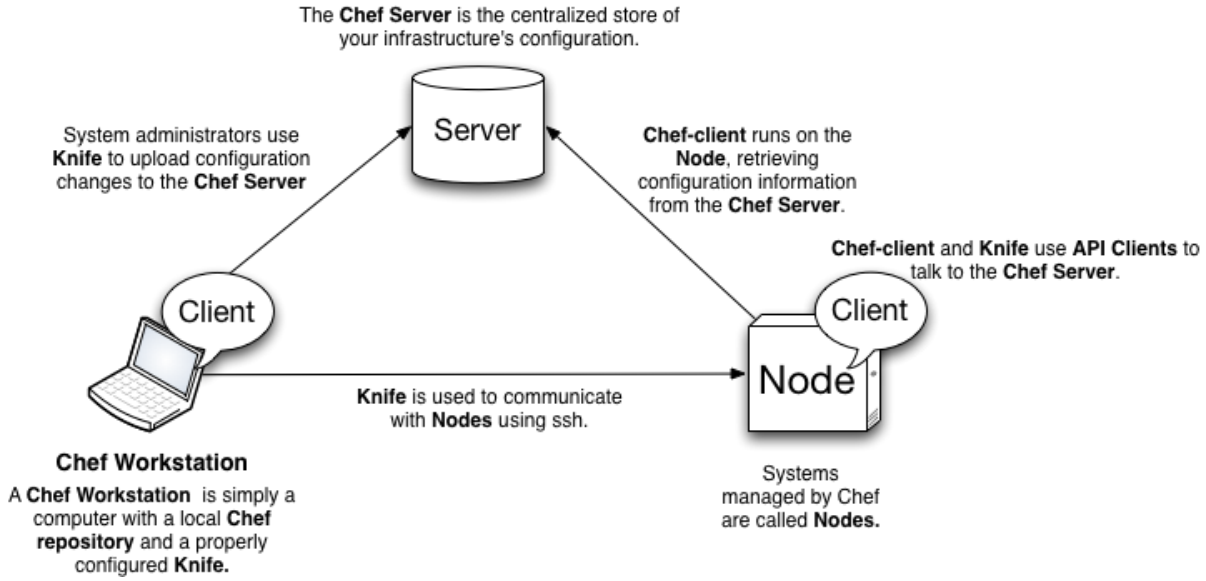

Figure 5-1. - Chef architecture and common interactions between hosts (source: [32])

Node configuration is laid down in its run list and attributes. Attributes provide a nested key-value pair collection of configuration information. The run list defines the ordered list of installation and configuration steps that need to take place on a node. These steps are grouped into so called *recipes*. Recipes are written in Ruby using the Domain Specific Language of Chef and describe a logically coherent part of node configuration. Recipes among other resources are collected into *cookbooks*. A cookbook encapsulates all the resources needed for automated configuration and deployment. Attributes can be defined in the cookbook with default values, which may be overridden at different nodes with node-specific values. Cookbooks also contain necessary templates, files, and libraries, which hold additional information required for the configuration.

In scalable scenarios, it is normal to have more nodes with similar functionality and settings. For grouping these nodes, Chef introduces roles. A role contains a run list with the desired recipes and cookbooks that should be executed and attributes with which the configuration processes can be further customized.

The path of execution taken during the run of Chef client is called a chef run, which creates, registers, authenticates the node, gets the appointed run list, loads and compiles the required resources, then configures the node. Chef runs are idempotent, which means that multiple chef runs will result in an identical node state.

Chef in itself is a deep topic and offers many more features, but the above mentioned concepts are enough to create a simple automated infrastructure.

5.1.4 Custom cookbooks and bootstrapping with Chef

I signed up for Hosted Chef, because it spared me from the tedious work of installing and operating a private Chef Server, besides it offers a responsive web user interface for managing roles, nodes, cookbooks, recipes.

I prepared a local host appointed to be the workstation. I installed Ruby and other dependencies along with Git for easier handling of repositories. I used Git to create a skeleton Chef repository. I downloaded the necessary key files from Hosted Chef and used knife to configure the environment and authenticate the workstation towards the Server.

Opscode Chef community[9] makes several professionally assembled and continuously maintained cookbooks publicly available, such as the Apache cookbook, which I intended to use. Knife assists the administrator in checking out these community cookbooks to the workstation repository then uploading them to the Chef Server. However, the Apache cookbook was lacking the necessary mods for enabling PHP and MySQL, so these alterations needed to be administered manually. The Apache cookbook comes with multiple recipes, but I only needed the default one installing the Apache core service and another that installs the PHP5 extension. The latter needed modification to describe that the PHP5 needs to support MySQL. To achieve this, I inserted the following blocks into the mod_php5 recipe:

```
package "php5-mysql" do
    action :install
end
package "libapache2-mod-auth-mysql" do
    action :install
end
```

I already decided that the target nodes will run Ubuntu OS, therefore I omitted to find out and expand the recipe further with the method of installing these features on other platforms.

After preparing the cookbooks responsible for setting up the web server environment, a custom cookbook had to be written, which deploys and configures the PrimeFinder application on nodes. This cookbook is made up of a single recipe, an attribute file, a customizable template file, and a static file. The attribute file defines the usable attributes and holds their default values. I defined the web root directory and the database connection parameters as configurable by attributes, and supplied default values:

```
default[:primefinderapp][:wwwroot] = "/var/www"
default[:primefinderapp][:db_address] =
                            "ec2-46-137-53-255.eu-west-1.compute.amazonaws.com"
default[:primefinderapp][:db_user] = "primedbadmin"
default[:primefinderapp][:db_password] = "primedbadminpassword"
default[:primefinderapp][:db_database] = "primedb"
```

[9] http://community.opscode.com/

The `dbconn.php` file of the PrimeFinder application is converted to a template file, in order to be able to use attribute values. During a chef run, the declarative tags will be substituted with the current attribute values, and the file will be rendered as a valid PHP file.

```
<?php
    $db_address = "<%= node[:primefinderapp][:db_address] %>";
    $db_user = "<%= node[:primefinderapp][:db_user] %>";
    $db_password = "<%= node[:primefinderapp][:db_password] %>";
    $db_database = "<%= node[:primefinderapp][:db_database] %>";
?>
```

The `index.php` containing the whole application logic is inlaid in the cookbook as a static file, because after being placed in the proper folder next to the `dbconn.php`, it can do its duty without any configuration. The only recipe in the `primefinderapp` cookbook simply copies the two PHP files to the location supplied in the `wwwroot` attribute and sets their ownership and access rights.

```
template "dbconn.php" do
    path "#{node[:primefinderapp][:wwwroot]}/dbconn.php"
    source "dbconn.php.erb"
    owner "root"
    group "root"
    mode "0755"
end

cookbook_file "#{node[:primefinderapp][:wwwroot]}/index.php" do
    source "index.php"
    owner "root"
    group "root"
    mode "0755"
end
```

I performed several manual tests to make sure that the automated configuration works flawless. Then I used and improved a solution described in [33] to create a custom user data script that will be passed to the newly launched instances and executed at their startup. This script is longer, so instead of displaying it, I will outline the activities it performs.

- *In the first step, it installs Ruby packages necessary to run the Chef client.*
- *Then it installs the Chef client using Ruby gems.*
- *It writes the configuration file of the Chef client with the necessary parameters of connecting to the Chef server, such as the Chef server URL and the validation key needed for authentication.*
- *It writes a bootstrap file containing the desired role name hardcoded, which will be used by the Chef client.*
- *Finally, it initiates a Chef run, which uses the previously created bootstrap file.*

The web server instances launched by the Auto Scaling service will be fed with this startup script, which performing the chef run, registers the instance as a node of web role, downloads the cookbooks, and installs and configures the node without any human intervention, so that it will be a fully operational web server running the PrimeFinder application.

5.1.5 Configuring the Auto Scaling service

Once the automatic deployment of web servers was tested and the database server was running, it was time to set up Auto Scaling. First, I created an Elastic Load Balancer (PrimeELB), which is configured to spread the HTTP traffic equally across the instances belonging to it. Besides load balancing, it offers a simple health monitoring feature, which uses periodic pinging as health check. If a monitored instance fails to respond within the defined timeout interval, it is marked unhealthy.

As mentioned before, Auto Scaling service does not yet have a web-based interface within the AWS Management Console, so for configuring scaling behavior, one must download the Auto Scaling Command Line Tool, which contains the necessary commands and accessories for creating and managing Auto Scaling Groups.

Before forming the Auto Scaling Group for web servers, a custom launch configuration (LCPrime) must be created, which contains the information necessary to launch instances. The following command can be used to supply this information:

```
as-create-launch-config LCPrime
    --image-id "ami-8d5069f9"    --instance-type "t1.micro"
    --region "eu-west-1"    --group "sgwebservers"    -monitoring-disabled
    --key "mykeypair"    --user-data-file "user-data-script"
```

The first parameter states the AMI, which shall form the base of new instances. These should belong to the Security Group of web servers, which allows network communication through port 80. The last parameter specifies the custom user data script that was discussed previously.

Using the LCPrime launch configuration, it is now possible to create an Auto Scaling Group called ASGPrime:

```
as-create-auto-scaling-group ASGPrime
    --launch-configuration LCPrime
    --region eu-west-1    --availability-zones eu-west-1a
    --min-size 1    --max-size 4
    --load-balancers PrimeELB    --health-check-type EC2
```

This command will create an ASG in the appropriate region with a limited number of instances, and puts the group's instances under the supervision of the load balancer. The `as-update-auto-scaling-group` command can be used for further modifications and fine tuning of advanced features, such as the cool-down period.

By issuing the ASG creation command, a new instance will be immediately launched, as the minimum size of 1 is a group requirement. The next step is to create Auto Scaling policies defining scaling activities that will be triggered by CloudWatch alarms.

```
as-put-scaling-policy ScaleUpPolicy
    --auto-scaling-group ASGPrime
    --type ChangeInCapacity    --adjustment=1
    --cooldown 300    --region eu-west-1
```

```
as-put-scaling-policy ScaleDownPolicy
    --auto-scaling-group ASGPrime
    --type ChangeInCapacity    --adjustment=-1
    --cooldown 300    --region eu-west-1
```

These commands add a pair of symmetric policies to the ASG, which define a certain change in capacity. A 5-minute cool-down phase is set after each scaling activity to allow the system to adjust itself to the capacity change. The only thing left is to set up CloudWatch alarms, which will invoke these policies when the value of a certain metric exceeds or falls below a threshold. CloudWatch alarms may be created via the online Management Console or using parameterized commands bundled in the CloudWatch Command Line Tools[10].

```
mon-put-metric-alarm HighCPUAlarm
    --metric-name CPUUtilization    --namespace "AWS/EC2"
    --comparison-operator GreaterThanThreshold    --threshold 80
    --period 60    --statistic Average    --evaluation-periods 1
    --alarm-actions "[unique ARN of ScaleUpPolicy]"
    --dimensions "AutoScalingGroupName=ASGPrime"    --region eu-west-1
mon-put-metric-alarm LowCPUAlarm
    --metric-name CPUUtilization    --namespace "AWS/EC2"
    --comparison-operator LessThanThreshold    --threshold 20
    --period 60    --statistic Average    --evaluation-periods 1
    --alarm-actions "[unique ARN of ScaleDownPolicy]"
    --dimensions "AutoScalingGroupName=ASGPrime"    --region eu-west-1
```

These commands create two alarms, which can be observed at the Management Console as well. The first alarm will go off when the average CPU utilization of the ASG instances over one minute exceeds 80 percent, and it will trigger the policy that issues a scaling up activity. Similarly, the LowCPUAlarm invokes the scaling down policy when the instances are idle, their average CPU usage is less than 20 percent. Policies can be specified with their unique identifier, the Amazon-specific ARN[11].

To summarize this case study, I displayed the most important services and components on an overview diagram, which helps understand the relations and cooperation among the entities.

[10] http://aws.amazon.com/developertools/2534

[11] ARN - Amazon Resource Number

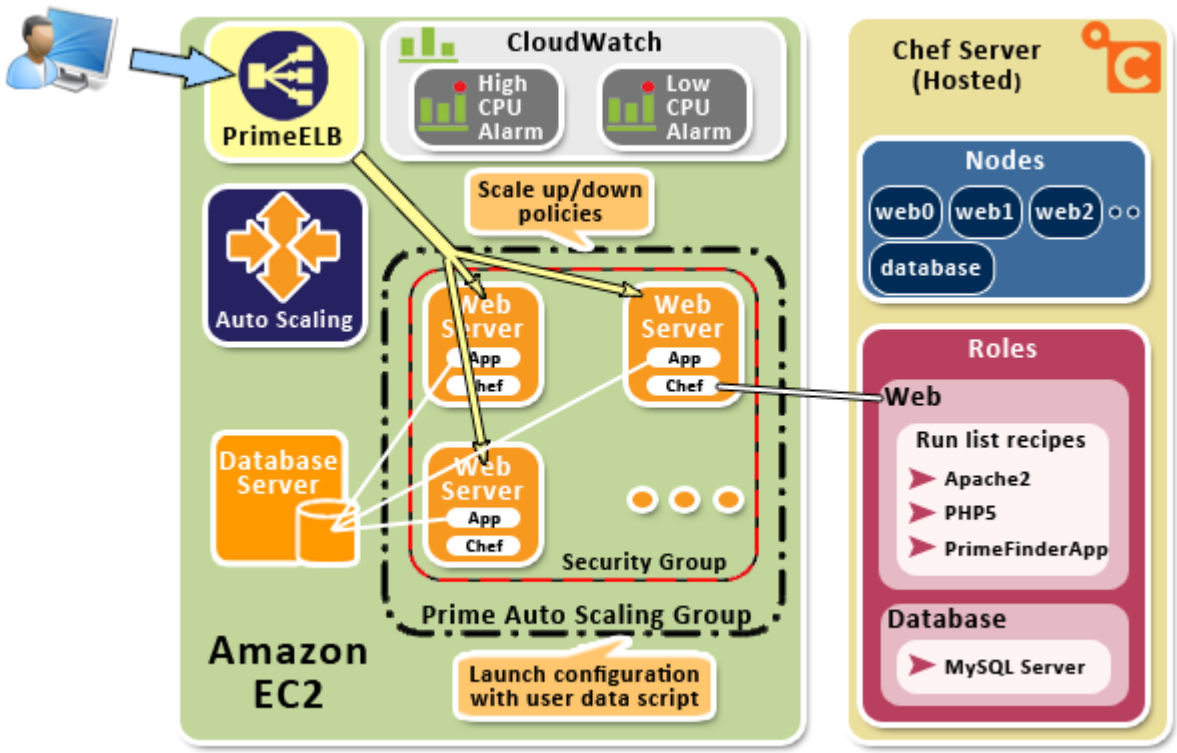

Figure 5-2. - Amazon Auto Scaling case study scenario

5.2 Monitoring the AWS infrastructure

5.2.1 Thoughts on monitoring

One of the most important aspects and challenges of cloud computing is monitoring. Cloud providers bill their clients based on the amount of consumed resources, such as computation time, used memory, persistent storage or network bandwidth. Therefore, it is essential for both parties to have a reliable and robust cloud monitoring solution. In order to be able to calculate statistics based on real-time performance data, all hardware or software components of the provided services must include a built-in, native monitoring subsystem, which publishes metrics to the cloud in a standardized method under any circumstances. Besides reliability, the monitoring solution shall offer a rich administrative and management surface, which enables complex querying, notifications, trend analysis, prognostication, and intervention to some level.

Monitoring solutions also need to be transparent for the user and entirely automated. It cannot have a negative impact on the overall performance. Since it forms the base of finances, it is highly important to exclude any possibilities of data tampering or forgery. It is also a helpful feature of monitoring if it provides visualizations of complex performance data. The human eye reads graphs more efficiently than large series of data. A monitoring solution cannot be complete without a configurable notification subsystem. Administrators do not have the capacity to constantly observe monitoring dashboards and graphs but they require to be instantly notified about unexpected deviations, operation malfunctions, or component health problems via a plausible communication channel.

5.2.2 Amazon CloudWatch

Amazon AWS provides CloudWatch to cope with these tasks. As it can be observed on the previous AWS architecture diagram (Figure 4-1), CloudWatch is spread throughout several of the other services so it can monitor the real-time performance of their components. CloudWatch enables the detailed monitoring of EC2 instances, persistent storage, relational and NoSQL databases, notification, queue and caching services, virtual networking components (e.g. load balancers) and Auto Scaling Groups.

There are three ways to access CloudWatch: it is integrated into the web-based Management Console; its features can also be utilized and automated with the Command Line Tools [15] or the AWS SDK. CloudWatch offers a clean user interface built in the AWS Management Console but it does not include the full functionality of CloudWatch. The web UI allows querying all the available metrics or performance counters that belong to the AWS services being used. Another important type of entity besides metrics is the *alarm*. Alarms can be defined with a certain threshold on a specific metric. The alarm gets triggered when the statistic value – such as average or maximum – of the chosen metric over a period compared to the threshold with the given comparison operator evaluates to true. The alarm metric is specified with its namespace, dimensions and name, which are to be discussed in more detail.

Alarms have three states, each of which can be assigned with a different action that will be triggered upon transitioning into the given state. There are two main types of actions: alarms may invoke Auto Scaling policies (such as scaling operations) or send out messages through Amazon Simple Notification Service (SNS).

CloudWatch comes with numerous built-in metrics that could not be handled without proper organization and structure, so metrics are classified into namespaces and dimensions. Namespaces are hierarchical, textual and they hold the service that the metric belongs to. Built-in metric namespaces always begin with "AWS/" prefix, which states that the metric resides on root level. Dimensions create subcategories within namespaces. For example, dimensions separate a distinct CPU utilization metric for each EC2 instance and another aggregated metric for each Auto Scaling Group. Instance and volume identifiers are the most common dimensions, which makes it possible to monitor the performance of various instances independently. As the plural notes, dimensions can be made up of more than one classifier. This feature enables the creation of multi-dimensional metrics that introduce the possibility of further categorization.

Clients can choose from two options regarding performance monitoring of EC2 instances: basic and detailed. They only differ in granularity; the default basic option provides performance metric updates at a 5-minute frequency, while detailed refreshes metric values in every minute though it is not free of charge.

Amazon understood that built-in metrics are not enough to meet the requirements of more complex infrastructural scenarios so they designed CloudWatch to be flexible to use custom, *user-created metrics*. Cloud applications running on EC2 instances – or even out of the cloud – can publish custom performance counter data points using the AWS SDK that is available to all common software development platforms. By publishing custom metrics, application developers can reveal the current operations of the software components' inner mechanics and make them

observable. Similarly to built-in metrics, custom ones are described with a namespace and a set of dimensions with arbitrary string values. The unit of the metric can also be specified from a predefined collection.

Custom metrics are useful if their values are nearly continuous, their values are updated periodically. The actual performance values need to be published at a given frequency. Using the CloudWatch Command Line Tools, this can be achieved with repeatedly run, scheduled shell scripts. Cloud applications must include the preconfigured AWS SDK component to gain capability of using the CloudWatch API, through which the web service requests and responses take place. By embedding the AWS SDK into the cloud application, it becomes possible to upload custom, internal performance measurement data that is entirely application-specific and not accessible from the runtime cloud environment. These custom metrics do not differ from the built-in ones; they can be queried and displayed graphically the same way or have CloudWatch alarms defined on them. Custom metrics enable developers to integrate application monitoring with the service that provides monitoring for the environment that runs the cloud application.

5.2.3 AWSWatch

The previous section shed some light on the capabilities of Amazon CloudWatch. Its web-based user interface within the Management Console is easy to use but it lacks several important features. It is capable of drawing accurate graphs of the values of the selected metric over the configured time interval, but neither the graph, nor the dataset it is based on is exportable. This deficiency prevents any further, deeper, more complex data analysis that could happen outside CloudWatch. It is also impossible to display more than one metric on the same graph, which could be used to detect underlying dependencies or correlations between performance fluctuations. Performance values can be queried with a properly parametered command available in the Command Line Tools or the AWS SDK, which gives a tabular output listing the data points of the metric in question.

I created a simple tool, called *AWSWatch* with a user-friendly graphical interface to avoid the tiring process of issuing the query command and make its output digestible. I designed it to give a comfortable interface and set of controls for selecting the metric and defining the query parameters. The output is exportable into CSV, which may form the base of further analysis.

This stand-alone application is made up of two components: the Windows Forms user interface and a class library containing the AWS SDK for .NET and the web service calls. The class library provides custom classes - `InstanceInfo`, `MetricInfo`, `MetricStatisticInfo` - that represent AWS-specific objects using types that are independent from the SDK. This design follows object-oriented principles and enables portability. For example, the definition of the type representing a metric independently from the AWS SDK looks like this:

```
public class MetricInfo
{
    public string Namespace;
    public Dictionary<string, string> Dimensions;
    public string Name;
}
```

I would also display the class definition of the `MetricStatisticInfo`, an instance of which represents a single data point of a given metric:

```
public class MetricStatisticInfo
{
    public Dictionary<MetricStatisticValue, double> Values;
    public double SampleCount;
    public DateTime Timestamp;
    public string Unit;
}

public enum MetricStatisticValue
{
    Average, Maximum, Minimum, Sum
}
```

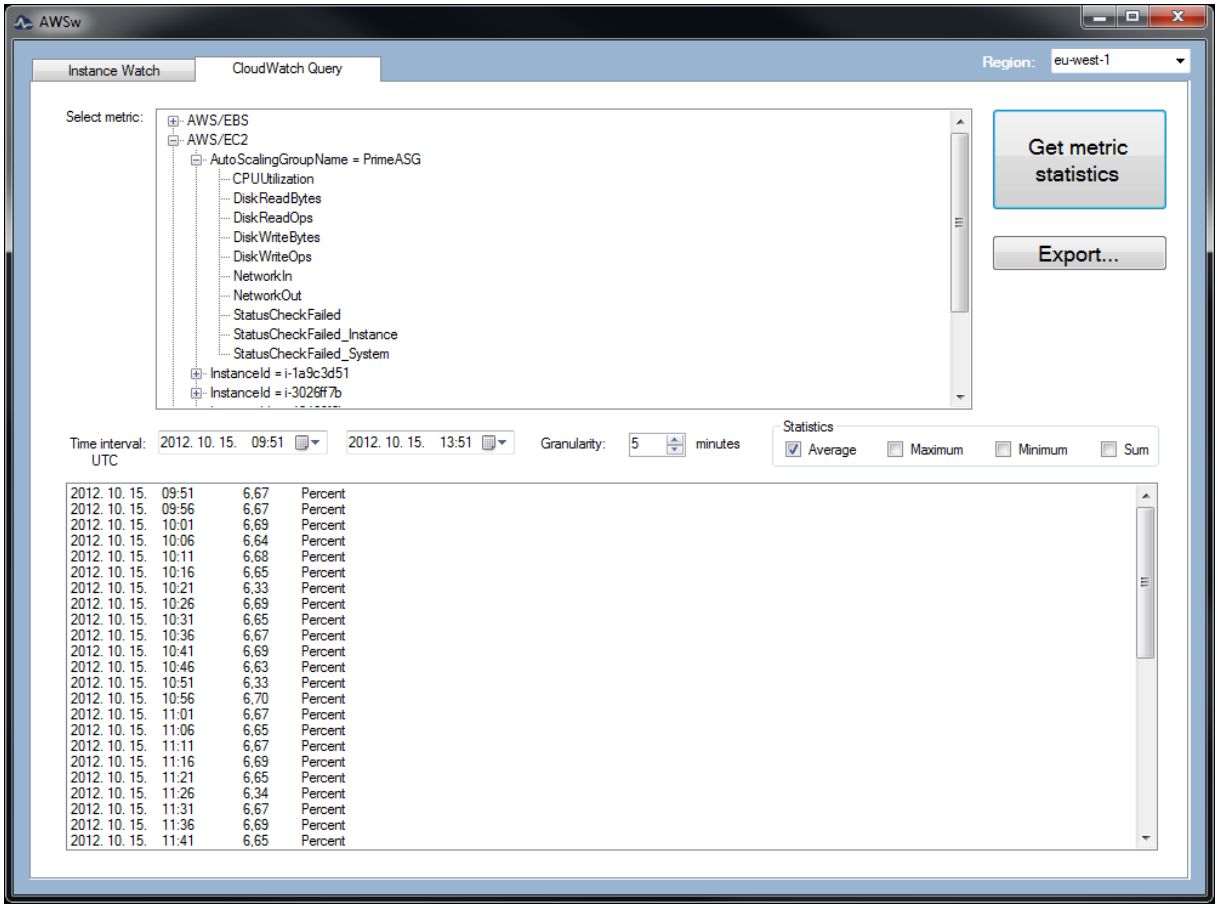

Figure 5-3. - AWSWatch CloudWatch Query interface

Web service calls using the AWS SDK follow the same logical pattern, which can be summarized in the following few steps:

- *Instantiation of the service client*

  The client object of the chosen service will act as a proxy and channel request-response communication between the caller and the cloud. From a developer aspect, this client object implements the `IDisposable` interface, so resource deallocation takes place automatically.

- *Creation and configuration of a request object*

  Besides client object types, the AWS SDK provides paired classes for requests and corresponding responses. A request type contains read-write properties representing filters associated with that type of filter. Upon instantiation of a request object, these filter properties assume their default values. The request conditions may be refined through the setters of the filter properties.

- *Issuing the request through the client*

  The service client exposes methods that are responsible for issuing the preconfigured web service request. These methods are the same as the commands available in the Command Line Tools. The request methods return a response object.

- *Enumerating response results*

  On successful request, the command-specific response object contains the query results in an enumerable collection. Be it on any platform, the AWS SDK comes with a thorough API reference documentation that helps in discovering the structure of request and response objects alike.

To illustrate a common service request to the AWS, I show here the code of querying the list of available metrics in a specific region with the help of AWS SDK for .NET:

```
var metrics = new List<MetricInfo>();
using (AmazonCloudWatchClient cwClient = new AmazonCloudWatchClient("eu-west-1"))
{
    var req = new ListMetricsRequest();
    var res = cwClient.ListMetrics(req);
    foreach (var metric in res.ListMetricsResult.Metrics)
    {
        var dims = new Dictionary<string, string>();
        foreach (var dimension in metric.Dimensions)
        {
            dims.Add(dimension.Name, dimension.Value);
        }
        metrics.Add(new MetricInfo {
                            Namespace = metric.Namespace,
                            Dimensions = dims,
                            Name = metric.MetricName
                        });
    }
}
```

In the above code snippet the previously described steps can be observed. Metrics are collected into a list named metrics in the form of custom, AWS-independent `MetricInfo` objects. The metric selector *TreeView* control is then populated with metric info after a complex grouping and sorting procedure, which creates a hierarchy out of the plain array of `MetricInfo` objects based on namespaces and dimensions that determine categories and subcategories.

The process of requesting the statistic values of the selected metric over the user-specified period is more complex. The filter properties of the request object - such as metric identification, time interval, etc. - are set to values specified on the user interface. The results, in this case the metric data points, are inserted into a listbox in a formatted string.

The other basic feature of the AWSWatch application is instance monitoring, which is capable of querying running instances along with their states periodically with a user-specified frequency, then export the log into CSV. This export might then be used for drawing line diagrams visualizing the change of instance count over time, which is practical when monitoring a scaling infrastructure. Querying instances is implemented using the AWS SDK the same way as shown previously, but to prevent the UI from becoming unresponsive during query cycles, the query logic is delegated to a background thread that fills a listbox with the periodic results. The querying process can be stopped any time and the result set can be exported to CSV.

I created AWSWatch solely for demonstrational purposes and it proved to be a practical tool for extracting monitoring data from CloudWatch. Furthermore, it sheds some light on the capabilities and the usage of a few of the available services of AWS SDK for .NET. There are many possibilities of improvement, which has not been implemented. For example in the current version the AWS credentials are hardcoded into the program. A new feature of asking the access keys from the user at first start then storing them ciphered would make the application more usable. It would also come in handy to have searching or filtering possibility of the hierarchical metric list that can grow large in case of bigger instance fleets. The web UI of CloudWatch offers the feature of drawing graphs of a single metric but occasionally it could be useful to display the values of multiple metrics over a specified time range on the same diagram. AWSWatch could be improved with advanced graph creating and exporting features.

I performed and logged several prime finding operations executed with various lower and upper bounds in different time of day during which I used AWSWatch to continuously monitor the instance count of the Auto Scaling Group and the relevant metrics, such as CPU utilization and a custom metric called Progress that will be introduced in the following section. I would like to demonstrate a simple horizontal scaling operation invoked by an alarm that is triggered by a long running prime finding process that consumes all the CPU time of an instance. This double diagram is entirely based on the measured metric data fetched with the help of AWSWatch.

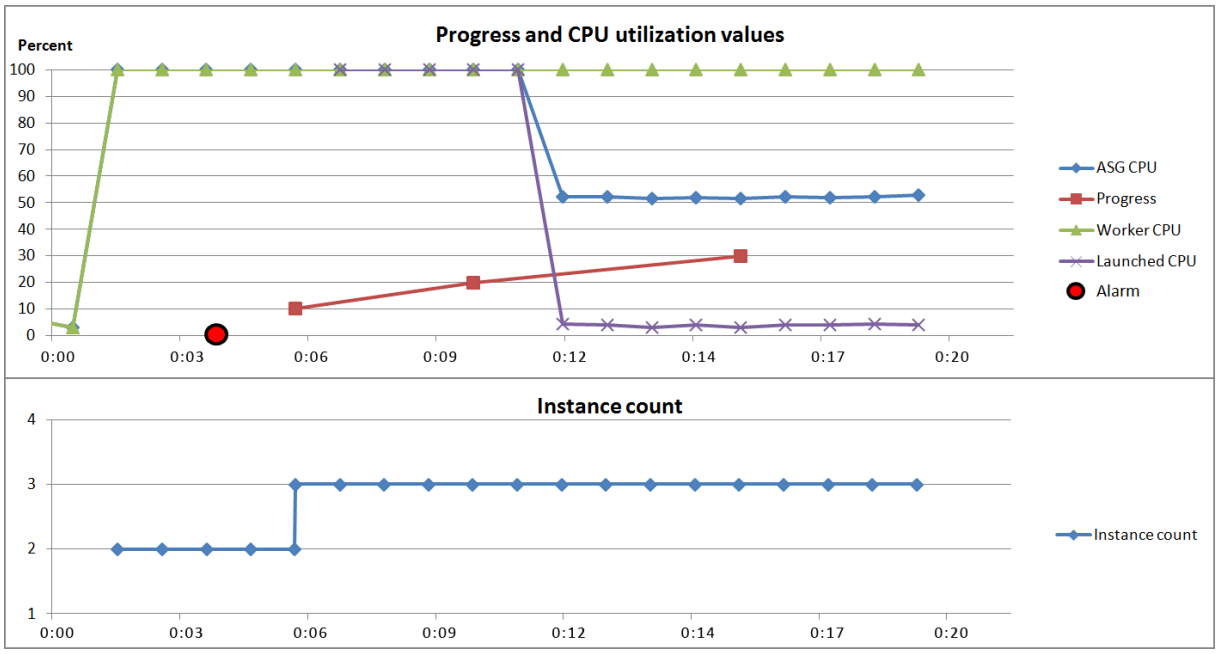

Figure 5-4. - Diagram of an alarm-triggered scaling operation

The lower diagram shows the change of instance count over the time duration of the experiment. At the beginning of the demonstration two instances are running: a constantly running stand-alone instance hosting the database and a single web instance within the Auto Scaling Group (ASG) that receives a user initiated prime finding request. Immediately, the instance starts the algorithm as it shows in its CPU utilization, which runs up to 100%. The red line displays the current progress that is published by the application through a custom metric to be described in the next section. An alarm is defined on the value of the CPU utilization aggregated over the ASG: it gets triggered when the minimum instance CPU load stays over 80% for three consecutive 1-minute long periods (marked with a red dot on the time axis of the upper diagram).

The alarm invokes the scaling action associated to it, which expands the ASG by one. The purple line marks the CPU utilization of the newly launched instance. It can be observed that it took about 3 minutes from the alarm to start up the instance, then it needed an additional 5 minutes to initialize and perform a Chef run that makes the initially empty instance fully capable of handling requests. After this, the CPU of the new instance frees up and it shows in the ASG's aggregated average CPU load too as it drops to 50%. Meanwhile, the prime finding operation is still in its early stage as its progress could not reach 40% during the 20-minute long experiment.

I also intend to show an example of unnecessary scaling. The following diagram depicts a scenario where the CPU-based alarm gets triggered but the prime finding process finishes before the launched instance could get down to work.

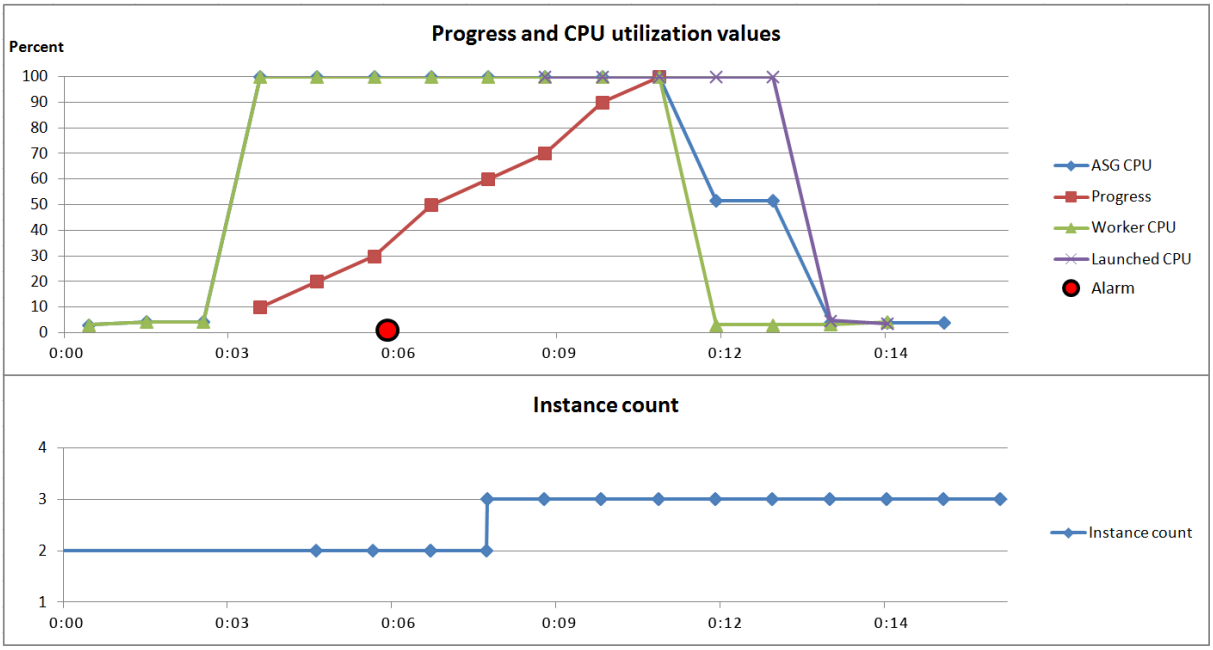

Figure 5-5. - Diagram of a premature horizontal scaling operation

It can be stated that the scaling out displayed on the graph above was in vain because the running operation finished before the freshly launched instance could become fully functional. Upon finishing initialization and auto-configuration, the other instance has already finished its task so there would be two idle instances and soon a down-scaling action would be invoked by a low CPU alarm. The reason for this behavior can be found in an improperly parametered alarm and the lack of knowledge on the operation's progress. Such oscillations can be prevented if the time needed for launching a new instance and the inner state of the application are taken into consideration.

5.2.4 Using custom metrics

The PrimeFinder application can be considered as a black box from the point of view of its cloud environment, there is no information on its internal operations. It just consumes all the CPU time of an instance for an unpredictable amount of time, which can be observed in the values of the built-in CPU utilization metric.

The aforementioned custom metrics make it possible for a cloud application to publish application-specific performance counters to CloudWatch. For experimental purposes I created a custom metric called Progress, the values of which report the actual progress of the current prime finding task. I modified the prime seeker algorithm so it can calculate the progress based on the currently checked number and the interval bounds. To avoid overloading the metric with too many values, which might cause a degradation of performance, I implemented the algorithm to publish only every 10th percent.

To add the capability of publishing metrics to the PrimeFinder application, the AWS SDK for PHP needed to be installed beside it, which exposes the necessary objects and methods for reaching the cloud. The SDK can easily be installed using *git*. This is achieved with a custom Chef cookbook, which automatically downloads the SDK and rewrites its configuration file containing

the AWS credentials. The SDK comes with many folders and files but all of its features are available through the `sdk.class.php`, which acts as a facade, so only this file needs to be included in an application.

The following code of metric publishing is separated into a function that instantiates a CloudWatch object then assembles the special array structure required for uploading a metric value.

```
$cw = new AmazonCloudWatch();
$cw->set_region(AmazonCloudWatch::REGION_EU_W1);
$response = $cw->put_metric_data('Custom/Prime', array(
    array(
        'MetricName' => 'Progress',
        'Dimensions' => array(
            array('Name' => 'InstanceId', 'Value' => $instanceId)),
        'Value' => $value,
        'Unit' => 'Percent'
    )));
```

It is important to specify the region before putting the metric data otherwise it gets uploaded to the default region (US-East-1). It should also be noted that the dimension of the custom progress metric is the identifier of the instance that runs the operation because concurrently running instances of an Auto Scaling Group might be running several prime finding processes in parallel.

Storing the current progress of the running prime finding operation proves to be a metric that is easy to calculate and understand. Also the progress values can be visualized on graphs. It is also possible to create an alarm based on the progress. Say if the progress of the current operation stays below 30% for five minutes, the prime finding is not likely to finish for another ten minutes, so a scaling action shall be triggered to launch a new computing instance that will take over incoming requests. The problem with this approach is that it neglects occasional performance drops of the cloud environment that are especially common in the AWS free usage tier. Furthermore, this method does not reflect that time necessary to reach the next percentage milestone increases as progress advances to the upper bound of the interval.

Taking the drawbacks of progress percentage into consideration, another type of metric was to be introduced. I added the functionality to the PrimeFinder to calculate an estimated remaining time of the current operation using simple arithmetic calculations with the previously elapsed time deltas as operands. The calculated estimation will give a rough approximation of the remaining running time. The method keeps track of elapsed time between checkpoints and weighs the estimated remaining time with an average of the ratios of the previous timeframes. The approximations given by this calculation can be significantly different from the real values. Accuracy could be further improved using more advanced mathematical techniques such as linear or logarithmic regression but these are not part of this thesis. A few measurements were made but the estimations were found to be imprecise because of the inaccurate estimating and the unpredictable performance deviations of the AWS free account.

5.2.5 Conclusions of AWS performance monitoring

During the described case study the free usage tier of AWS was used, which provides a good test environment: most of the services can be utilized free of charge up to a specified monthly usage limit. All resource consumption that exceeds the limit will be automatically charged to the user's credit card. Several monitoring scenarios were performed repeatedly and their results showed that the overall computing performance of the AWS free tier is rather unpredictable and unreliable. AWS operates with automatic management functions that monitor the resource usage of free instances. If significant, prolonged peaks are detected, the priority of further processes on the instances belonging to the account will be lowered. Therefore upcoming operations will run slower because instances receive a smaller share of the CPU resource pool. Surely, non-free instances do not suffer such discrimination. Cloud applications that require stable performance to meet a certain SLA might be deployed to dedicated instances.

Although, it is clear that even the free tier is usable for building dynamically scaling infrastructures as all the necessary services are available through the Management Console and the Software Development Kits. Proper scaling configuration requires detailed performance data that may be collected via CloudWatch or a third party tool such as AWSWatch, which uses the API of the AWS services. After performing and observing numerous experiments, I found that thorough analysis helps finding the proper settings and configuration values for automatic scaling.

It is also evident that embedding into the cloud application the feature of publishing custom metrics makes scaling much more efficient as the cloud runtime environment gains knowledge of the operation internals and such information may form the base of more accurate scaling decisions. I came to find that AWS provides an easy way of creating and storing custom metrics, which then can be handled the same as metrics built in the platform.

5.3 Using WASABi to auto-scale a multi-tier application in Azure

To test the capabilities of WASABi, I designed a multi-tiered application, which can be stressed at a desired level. The resource usage of the application shall depend on the supplied user input to enable human intervention in load. I decided to implement a simple cryptographic application, which provides an MD5 hash decoding service using brute force guessing. This manner of decryption causes that the necessary time to decode depends on the input hash.

5.3.1 Designing the MD5 Hash Decoder application

The task can be broken up to two components or roles in Azure terminology. The user interface of the service, which is an ASP.NET web site, is placed in a web role. It runs a simple web page that allows users to input the hash thumbprints that they wish to decode. The web site also gives feedback of the background processes by displaying a formatted list of previously decoded hashes with their corresponding decrypted values.

The hash decoding logic is placed in a separate component, a worker type role. The input hashes from the web site should be somehow made accessible for this background worker role. Direct method invocation between roles is not possible in cloud environments because running instances have little knowledge of each other, and it would also break the basic principle of component decoupling. Therefore, data exchange between the web frontend and the background logic roles is done through a queue, which stores the hashes that are waiting to be processed.

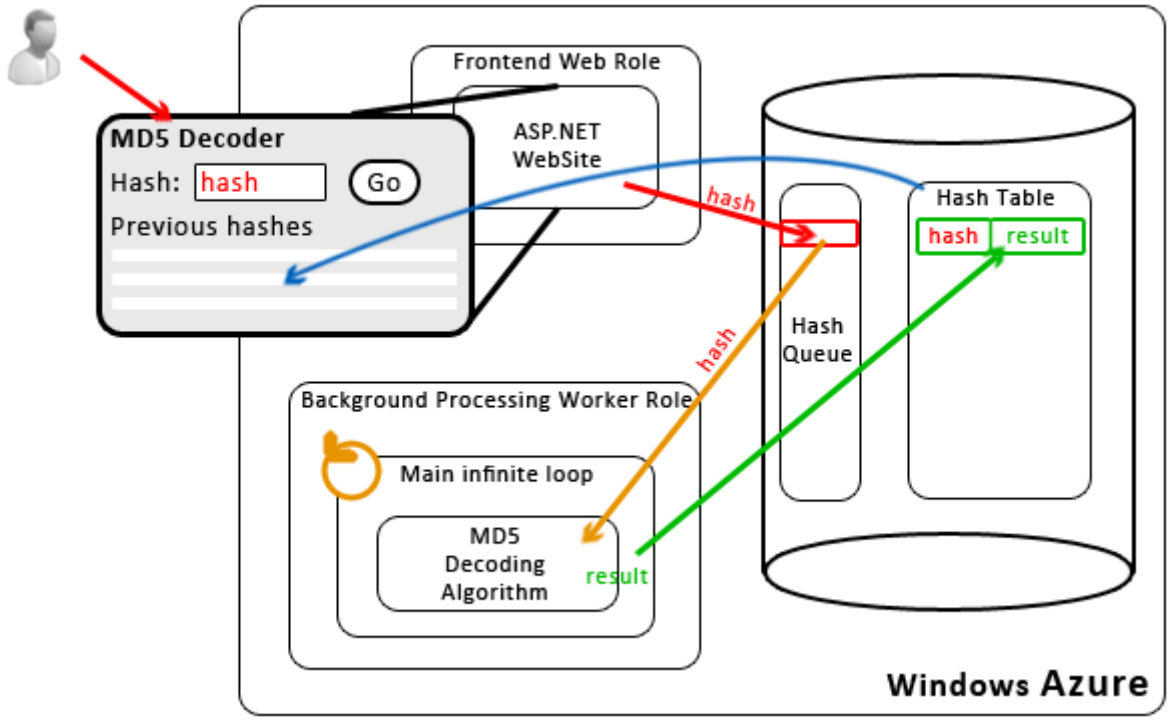

Figure 5-6. - Blueprint of the MD5 Hash Decoder Application

The Frontend Web Role providing the web user interface of the application has two responsibilities; it allows users to request decoding of inputted valid MD5 hashes and it displays a list of previously decoded hashes along with their hashed values. When initiating the decoding, it simply places the hash into the queue. The hash-value pairs that have already been successfully decoded plus the time taken by the decryption are stored as entities in a schemaless table residing in the persistent storage space of Azure. To give users a feedback of the decoding processes and results, the web interface includes a list that enumerates these entities.

An infinite loop works in the core of the Background Processing Worker Role. In the loop, it periodically turns to the queue and asks for a new message, if exists. The received message contains the hash to be decoded. To avoid unnecessary decryption processes, the role first checks whether the hash has already been decoded, by querying the table. The records of the table containing the hash-value pairs are identified by the MD5 hash itself. If the query returns a record, the role discards that hash and proceeds with polling the queue. In the other, more frequent case, when the inputted hash is not found in the table, the role commences the decoding by calling a function containing the brute force algorithm. This function is encapsulated in a separate class along with some configuration properties. The worker role also measures the time elapsed during the decoding process. This duration is then being stored in the table beside the hash-value pairs.

Guessing is done by hashing an ever-changing string and comparing the MD5 fingerprint to the inputted hash. The current string is varied algorithmically by incrementing the last character and lengthening character by character, making the entire search space exponential in size. Since brute force guessing consumes big amounts of time and resources, I made the guesser class configurable. To limit the possibilities, I restricted the length and character set of the string. This obviously degrades the functionality because the decoder will only be able to reverse hashes of string values that contain characters between the preset upper and lower ASCII-code bounds and of those with a limited length. By applying these configurable limits, I managed to shrink the search space. For instance, if the limits are set to enable values shorter than 7 lowercase alphabetical characters, the number of possible combinations is *26^6*. Brute force allows making the decoding time dependant of the inputted hash. For example, if I the MD5 hash of a single '*a*' character is given as input, then the decryption is instantly done, but with using more characters from the end of the alphabet, the decoding time can be dynamically adjusted according to experimental needs.

The decoding might finish by finding the hashed value or reaching the end of the search space. If the hash of the current guess equals to the inputted hash, then the guess is getting stored in the table along with its hash and the duration of the decoding process. But when the guessing reaches the final combination of the search space ('*zzzzzz*' in case of 6 lowercase letters preset), it returns that the hash cannot be decoded. Either way, the message received from the queue gets deleted to avoid being processed again.

5.3.2 Scalability potential of the application

The MD5 Hash Decoder is built to fulfill the design requirements of cloud applications. Its functionality is broken up to logically coherent components, which are loosely coupled and co-operating using a queue service. The incoming load on the Frontend Web Role is negligible comparing to the Background Processing Worker Role, which runs the decoding processes, therefore scaling must be targeted on this worker role. Multiple instances of a role might be running concurrently, in case there are many hashes waiting in the queue. When the load is light, one worker instance may be enough, but if its computing capacity gets used up by a long-running decoding process, many messages might line up in the queue representing incoming user requests. Scaling can be based on the length of the queue. When the number of awaiting messages exceeds a certain limit over a given time, another worker instance shall be launched that will pick up the arrived messages. Scaling up can be repeated many times until the queue gets shorter. When the queue length falls below an acceptable value, one or more idle worker instances should be terminated.

Using the queue makes it possible to process and decode the incoming hashes concurrently by several running worker instances that are identical in their functionality. Running instances are capable of independently retrieving and processing messages, and with scaling, parallel decoding processes can run. In an unforeseeable case of an instance crash in the middle of decoding a hash due to some internal or external reason, it cannot finish the process properly and fails to delete the message from the queue. Azure queues like any other common cloud storage queue services provide a feature called *message visibility*, which comes handy in such situations. Visibility means, that the message when pulled from the queue is not removed permanently, it is only set invisible therefore making it unreachable for other consumers. The message is only deleted on receiving an explicit removal request from the entity that consumed it. If this request does not arrive within the configured visibility time, the message reappears in the queue. This feature helps to prevent lost messages due to processing failures. Using the queue service, the application becomes scalable and highly fault tolerant.

To implement the scalable cloud application, the following main tasks need to be done:

- *Create the Azure service with 3 roles, configure the storage access and create the Hash Queue and Hash Table.*
- *Configure the Autoscaling Application Block through the rules and service information XML files and the App.config of the hosting role.*
- *Implement the Background Processing Worker Role with the hash decoding logic and the Frontend Web Role. Both need access to the storage Queue and Table.*

5.3.3 Configuring the Autoscaling Application Block – Flavoring with WASABi

Application configuration

The Autoscaling Application Block might be placed in a separate worker role of the cloud application, in an independent cloud service, or a locally hosted Windows application. Either way, the configuration data of the Block can be found in the App.config file. The Enterprise

Library contains an easy-to-use, visual editor for manipulating the App.config, which frees us from the strenuous task of XML editing. However, it is advisable to check the effects of modification in the XML and get familiar with the elements and their attributes. The Block parses the settings essential for its operation from the `autoscalingConfiguration` section of the App.config listed below.

```
<autoscalingConfiguration dataPointsStoreAccount="[StorageAccountConnectionString]"
    ruleEvaluationRate="00:01:00"
    rulesStoreName="Blob Rules Store"
    serviceInformationStoreName="Blob Service Information Store">
    <rulesStores>
        <add name="Blob Rules Store" type="[...]BlobXmlFileRulesStore"
             blobContainerName="autoscalingcontainer" blobName="rules.xml"
             storageAccount="[StorageAccountConnectionString]"
             monitoringRate="00:05:00"
             certificateStoreLocation="CurrentUser" />
    </rulesStores>
    <serviceInformationStores>
        <add name="Blob Service Information Store"
             type="[...]BlobXmlFileServiceInformationStore"
             blobContainerName="autoscalingcontainer"
             blobName="services.xml"
             storageAccount="[StorageAccountConnectionString]"
             monitoringRate="00:05:00"
             certificateStoreLocation="CurrentUser" />
    </serviceInformationStores>
</autoscalingConfiguration>
```

The most important properties here specify where the files containing the rules and service information are located. In my scenario, I chose to store them in the Azure Storage space in the form of XML-typed blobs, because blob storage provides a highly available, secure location for these important files. Rules and service information XML files might be placed local to the Block, in the file system, but WASABi also gives the possibility to create and use a custom store.

App.config contains the settings regarding diagnostics and logging. WASABi publishes two diagnostic event sources, one for general notifications about rule evaluation and scaling events and another for notifications on service information or rule descriptor changes. Journaling of these messages may take place into a common Windows Azure Diagnostics table in the storage space or custom logging solutions may be attached to the Block (for more information on custom logging see [24]).

I find it important to emphasize the role of management certificates played in all interaction with and within the Azure environment. A unique certificate is created for each Azure subscription and it is also possible to upload and use custom enterprise certificates. Cryptographic keys embedded in these certificates are being used for signing various requests to the cloud infrastructure, such as application deployment, configuration changes or submitting scaling operations. The certificates need to be uploaded on the management portal then it becomes possible to assign them to different hosted cloud applications. One must not omit specifying the name and fingerprint of the associated management certificate in the properties

of the application roles, otherwise incorrect certificate settings lead to unauthorized thereby failing service requests.

In the following section, I will elaborately show and explain the configuration contents of rules and service information XMLs. All rules are targeted to the Background Processing Worker Role.

Constraint and reactive rules

First of all, a constraint rule restricts the maximum number of concurrently running instances of the worker role and by defining a lower bound it ensures that at least one instance is always running to keep up a minimum level of operation.

```
<rule name="Default" rank="1" description="Constraint rule" enabled="true">
  <actions>
    <range min="1" max="4" target="BackgroundProcessingRole" />
  </actions>
</rule>
```

Before laying down reactive rules, one must specify the operands that these rules will be applied on. In the operands section I defined a queue length type operand, which represents the maximum length of the given queue over the last five minutes. It is worth to mention that 5 minutes is the shortest time span WASABi supports.

```
<queueLength alias="HashQueue_Length_Max_5min"
             queue="hashqueue" aggregate="Max" timespan="00:05:00" />
```

I defined a pair of scaling rules based on this operand, one for scaling the Background Processing Role up, another for scaling it down depending on the recent queue length:

```
<rule name="ScaleUpLongQueue" enabled="true">
  <when>
    <greater operand="HashQueue_Length_Min_5min" than="5" />
  </when>
  <actions>
    <scale target="BackgroundProcessingRole" by="1" />
  </actions>
</rule>
<rule name="ScaleDownShortQueue" enabled="true">
  <when>
    <less operand="HashQueue_Length_Max_5min" than="5" />
  </when>
  <actions>
    <scale target="BackgroundProcessingRole" by="-1" />
  </actions>
</rule>
```

These rules ensure that when the number of incoming tasks waiting in the queue rises above 5, a new instance of the processing role is launched. When the concurrently running role instances finished the decoding tasks, the queue shortens, and the idle instances will be shut down.

The operands are referenced in comparator expression elements coming from the following set: *greater, greaterOrEqual, less, lessOrEqual, equals*.

As mentioned beforehand, besides queue length, it is possible to create rules based on performance counter values. For this, the chosen performance counter needs to be defined as an operand along with the source of values, time span to watch, and aggregate function to apply on the diagnostic data:

```
<performanceCounter alias="BackgroundProcessingRole_CPU_Avg_5min"
                    performanceCounterName="\Processor(_Total)\% Processor Time"
                    source="BackgroundProcessingRole"
                    timespan="00:05:00"
                    aggregate="Average" />
```

In order to be able to utilize the performance counter values, one must properly configure the given role and place the appropriate code into the role's initialization function to make performance data available in the cloud. I will elaborately discuss this method in the following section. Using this operand, scaling rules can be assembled based on the performance counter values and thresholds. I defined a pair of rules using the CPU usage percent as the operand, which trigger scaling actions when the metric value exceeds 80 or falls below 20.

```
<rule name="ScaleUpHighCPU" enabled="true">
  <when>
    <greaterOrEqual operand="BackgroundProcessingRole_CPU_Avg_5min" than="80" />
  </when>
  <actions>
    <scale target="BackgroundProcessingRole" by="1" />
  </actions>
</rule>
<rule name="ScaleDownLowCPU" enabled="true">
  <when>
    <less operand="BackgroundProcessingRole_CPU_Avg_5min" than="80" />
  </when>
  <actions>
    <scale target="BackgroundProcessingRole" by="-1" />
  </actions>
</rule>
```

Real-life scenarios may require that more than one metric value shall be taken into consideration in scaling decisions. The Block's rule engine supports associating multiple operands to reactive rules and grouping them logically to create complex conditional expressions. One or more of the previously described comparator expressions might be nested inside the following grouping tags:

- *Any: the rule evaluates to true when one or more of the grouped condition stands.*
- *All: requires all the grouped conditions to be true.*
- *Not: negates the conditions grouped beneath it.*

Service information

The other XML file holding the service information contains the details of the Azure subscription of the user, the name of the service and its roles, and the storage connection string in the following structure:

```
<serviceModel>
  <subscriptions>
    <subscription name="[...]" subscriptionId="[...]"
                  certificateStoreLocation="CurrentUser"
                  certificateStoreName="My"
                  certificateThumbprint="[...]">
      <services>
        <service dnsPrefix="[Public DNS name of the hosted service]"
                 slot="[Staging|Production]"
                 scalingMode="[Scale|Notify|ScaleAndNotify]">
          <roles>
            <role alias="BackgroundProcessingRole"
                  roleName="BackgroundProcessingRole"
                  wadStorageAccountName="AzureStorage" />
          </roles>
        </service>
      </services>
      <storageAccounts>
        <storageAccount alias="AzureStorage"
                        connectionString="[StorageAccountConnectionString]">
          <queues>
            <queue alias="hashqueue" queueName="hashqueue" />
          </queues>
        </storageAccount>
      </storageAccounts>
    </subscription>
  </subscriptions>
  <stabilizer scaleUpCooldown="00:12:00"
              scaleDownCooldown="00:12:00" />
</serviceModel>
```

The first section contains the identification of the Azure subscription and points out the certificate that can be used for authenticating management requests towards the cloud. Here must be specified the hosted service of the subscription with its public DNS name and slot type. Azure provides two slots for a hosted service, the *Staging* for deploying and testing new versions of the cloud application and the *Production* for hosting the live version of the application. The latter is associated with the public DNS name of the service ending in *cloudapp.net*. Promoting a deployed application from Staging slot to Production can be achieved in a matter of seconds. Deploying a new version directly to the Production slot amounts to a nearly 15-minute downtime of the live application, which can be avoided by uploading it to Staging then migrating to Production.

The service information XML contains the scaling mode of the target service, which determines the Block's behavior. For experimental purposes, WASABi can be told to send email notifications instead of submitting scaling actions when circumstances trigger a scaling event. The service's scalable role name and the access to the storage holding performance data must also be specified in this file. Different service roles might be grouped together into scale groups, which can be defined in a separate section. Scale groups might contain roles with functional or performance dependencies among them that make it necessary to expand in unison thereby avoiding overdriving possible bottlenecks.

The Block's Stabilizer object can also be configured in this file by supplying the cool-down periods that need to pass after the execution of any scaling operation. Choosing these intervals properly prevents premature, oscillatory scaling actions. The approximately 5 to 8-minute launching time must be taken into consideration during while new scaling actions should be blocked.

After performing these steps of configuration file editing, the auto-scaling Block is ready to work; only the following lines of code should be inserted into the entry point of the host application, be it a cloud service role or an on-premise application:

```
Autoscaler autoscaler =
    EnterpriseLibraryContainer.Current.GetInstance<Autoscaler>();
autoscaler.Start();
```

5.3.4 Implementation details

In this section, I would like to highlight a few essential, Azure-specific fragments of the cloud application's code. To make the application user-friendly, I placed a table on the front page of the site, which displays a table of previously decoded hashes. Technically, this is an ASP.NET *GridView* with a dynamically set data source displaying the contents of the hash table residing in the Azure storage. The logic necessary to reach this table is separated in a facade helper class named `TableConnector` which takes care of establishing storage connection and provides methods for querying or updating the table.

The `HashValueEntity` class helps mapping table entities to runtime objects, makes entity attributes accessible via object properties. A `HashValueEntity` is made up of the mandatory *PartitionKey* and *RowKey* fields necessary for unique identification and indexing plus 3 custom fields: the inputted MD5 hash, its decoded value, and the elapsed time during decoding. The RowKey can be considered the primary key of the table meanwhile the PartitionKey supports a basic categorization of entities that makes query execution significantly quicker. In this case I chose the hash to be the identifier because MD5 collisions are extremely rare due to the one-way characteristic of the hash function.

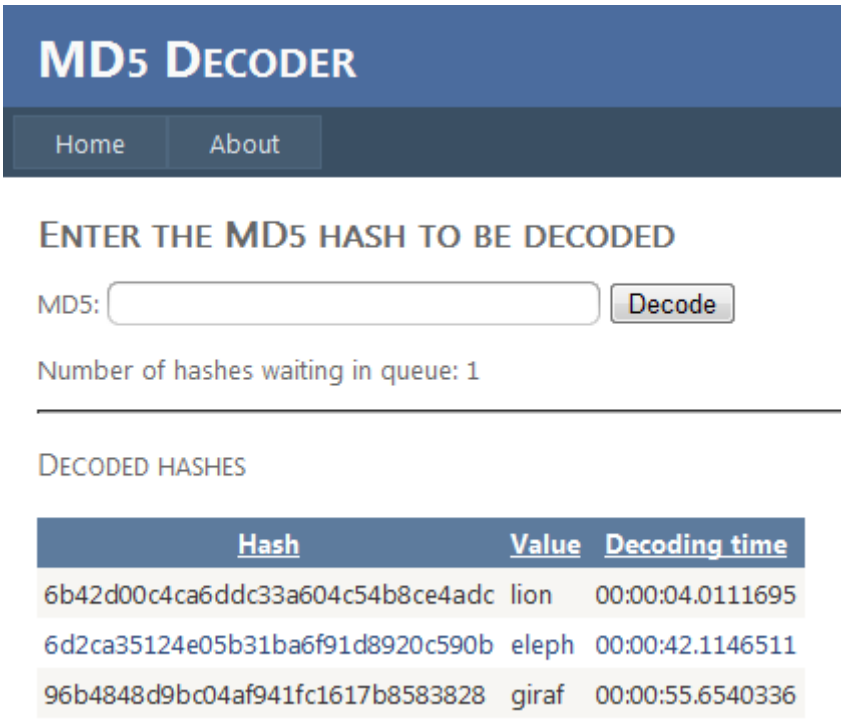

Figure 5-7. - Web user interface of the MD5 application

The GridView's data source is retrieved runtime via a method of `TableConnector`, which is made up of the following LinQ query fetching the table:

```
CloudTableQuery<HashValueEntity> hashQuery =
    (from entity in TableConnector._serviceContext
                    .CreateQuery<HashValueEntity>(TableConnector.CloudTableName)
     /* where entity.RowKey == hash */
     select entity).AsTableServiceQuery();
```

The private static `_serviceContext` member of the `TableConnector` class represents the context through which it is possible to send queries to the Azure storage table. The `TableConnector` does not expose direct context access; instead it encapsulates and publishes methods that return certain result sets. The commented line illustrates a query variant with a conditional clause that is used in the Background Processing Role, when the instance, before starting the time consuming decoding process looks up the inputted hash in the table whether it has already been decrypted.

I implemented the usage of a storage queue for containing the user inputs. An alternative could be the Service Bus Queue but the simplicity of holding 32-character long hash strings does not require such advanced features like ordering guarantee or transaction support. Following the same method as for the table, I implemented a helper class called `QueueConnector`, which gives access to the queue through message adding and removal methods while hiding the particulars of storage queue requests or connection establishment.

The main loop of the Background Processing Role frequently turns to the queue and receives the contained message if there is any:

```
while (true)
{
    var message = QueueConnector.GetMessage();
    if (message != null)
    {
        //looking up the hash in the table, then start decoding if needed
    }
    else { Thread.Sleep(QUERYFREQUENCY); //wait
}
```

The `QueueConnector.GetMessage` function invokes the actual message receiving method submitted to the cloud storage through a service context proxy object. Message retrieval needs to be supplied with a `TimeSpan` parameter, which determines the message visibility timeout, the amount of time after which the message reappears in the queue if it was not deleted meanwhile. I set two hours of visibility, which is ample to run the limited brute force guessing algorithm to completion.

The Frontend Web Role is also responsible for pushing the inputted hashes into the queue and displaying an approximate queue length. Both features utilize the same `QueueConnector` class for creating a context.

Another important part when using auto-scaling is to properly set up diagnostics. Detailed configuration suited to the given scenario is possible through a built-in class called `DiagnosticMonitorConfiguration`. With its help one can thoroughly configure the event and application logs along with the collection options of the desired performance counters. Proper configuration ensures that the events, trace information and performance metric values are periodically transferred from the role instance to dedicated tables in the storage space called WAD[12] Tables. I present the proper way of setting the diagnostic engine to monitor the values of the CPU time performance counter at a 30 seconds sampling rate:

```
DiagnosticMonitorConfiguration config =
    DiagnosticMonitor.GetDefaultInitialConfiguration();

config.PerformanceCounters.DataSources.Add(
    new PerformanceCounterConfiguration()
    {
        CounterSpecifier = @"\Processor(_Total)\% Processor Time",
        SampleRate = TimeSpan.FromSeconds(30)
    });
config.PerformanceCounters.ScheduledTransferPeriod = TimeSpan.Minutes(1);
```

The configuration can hold many more performance counters even custom ones created by the application developer. Inserting and populating custom metrics is rather cumbersome, and it falls beyond the scope of this work. For an overview and a full list of available platform built-in counters see [29]. It is also important to add system and application level event logs as data

[12] Windows Azure Diagnostics

source to the diagnostic information collector by properly setting the `DiagnosticMonitorConfiguration`.

My personal experience was that the diagnostic infrastructure needs some time to initialize after instance launch. This can be achieved from code by suspending the executing thread for a few seconds. Without this delay, the diagnostic instructions raise exceptions right after instance startup, which push the instance into an unstable state from which it cannot recover.

As a conclusion of the project description, here stands an overview diagram of the entire cloud application along with its modules and Azure storage entities.

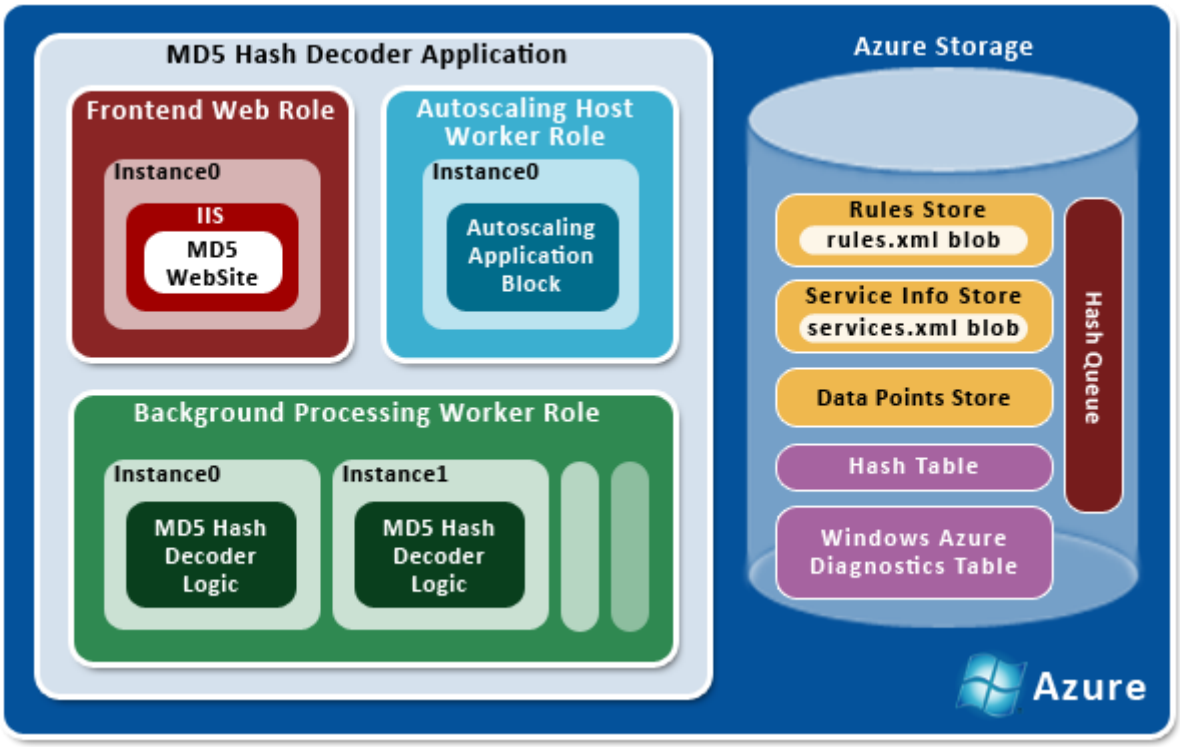

Figure 5-8. - WASABi case study architecture on Windows Azure

5.3.5 Monitoring the scaled Azure application

Azure Diagnostics automatically creates and fills tables containing monitoring information in the storage space of the configured storage account. Diagnostic data is separated in these tables:

- *WADLogsTable*

  This table stores trace information coming from running role instances, serves as the "console" of the remotely running application. It comes very handy in debugging. Developers may implement exception handling using trace messages for logging emerging exceptions. A table entity representing a log event is described with the timestamp, the identifier of the source role instance and the message among other miscellaneous attributes.

- *WADPerformanceCountersTable*

  It stores all the metric data points published from role instances. Similarly to Logs table, entities are attributed with a timestamp, the source instance identifier, and the particular performance counter name and its current value. This table provides the data mine for the data collector service of the Autoscaling Application Block. The collector sweeps through this table to harvest different statistical and aggregated values calculated from its content.

- *WADWindowsEventLogsTable*

  The components of .NET-based hosted services are running in virtual machines that have Windows operating system installed. All the informative or failure event messages generated by Windows are published in this table.

- *AutoscalerDatapoints*

  It is created and populated with data point entities by the Autoscaling Application Block's data collector object. Data points come from built-in or custom sources defined by rule operands and hold descriptive attributes as source identifier, current value, etc.

The previously described tables are similar to user-created tables, accessing them does not require more than the Storage API available in the Azure SDK. To extract the rich log and performance information, developers can implement simple data access and transform solutions for further analysis or export to other journaling or support systems.

Third-party software tools are also available to access the storage tables, such as Cerebrata's Azure Diagnostics Manager[13], which offers many features to visualize and analyze the gathered diagnostic data. Cerebrata also offers a solution for managing storage space and contained entities in the form of Cloud Storage Studio[14]. These applications are intended for commercial usage but free tools are also available on the Internet.

5.3.6 Implementing a custom rule operand for WASABi

In the Amazon case study I pointed out the benefits of using custom metrics to scaling. A properly implemented internal metric makes it possible to track the operations running under the hood. WASABi supports performance counters and queue length as operands of reactive rules. The performance data sets measured by the cloud environment give a detailed picture of the instance condition and actual resource consumption but only from an external point of view relative to the running application component. Queue lengths represent the number of incoming process requests not yet served but hold no information on why must they wait or the estimated remaining time. Under uncommon circumstances or unexpected failures these metrics may be misleading and the rules based on them set off unnecessary scaling actions. Dynamic scaling could be more efficient when internal factors are included in decisions. The Autoscaling

[13] http://www.cerebrata.com/Products/AzureDiagnosticsManager/

[14] http://www.cerebrata.com/Products/CloudStorageStudio/

Application Block enables to implement and use custom rule operands that serve as application internal metrics.

Similarly to the Amazon case study application, I decided to feature the MD5 decoder with the custom metric of progress meaning the current advance of the guessing algorithm in the bounded search space. This metric I implemented only as a proof of concept because logically it cannot serve as a reactive rule operand since the completion of the process is unpredictable. It cannot be determined of an incoming hash that it would take seconds or hours to decrypt, or that its hashed value is not even to be found in the entire search space. Progress though provides a useful performance counter for measuring the computing speed of instances.

In the following part, I intend to describe the implementation of the custom progress metrics step by step as it is rarely documented in the WASABi-related online technical content [25].

I needed to modify the application structure previously presented because testing, fault handling, quick fixing gets easier when the Autoscaling Application Block component is hosted locally in a console-based application instead of being deployed to the cloud in a separate role. Also, the assembly of the component defining and publishing the custom operand must be referenced by the Block. The component in question is the Background Processing Role in this application, which publishes the progress metric. Tying it with a reference to the Block would have created a dependence that I chose best to avoid. Instead, I implemented a solution in which the Background Processing Role pushes progress information into a separate storage table, and an individual component representing the custom operand local to the Block consumes this data and transforms it for the Block to process. This component realized as a class library named `ProgressOperandLib` is responsible for defining the operand and supplying values for the Block.

Besides placing the new library next to it, I also reconfigured the Block to use local file store for rules and service information XMLs and log events to the console instead of the cloud storage table. These arrangements make debugging WASABi exceptions and altering rules much easier.

The following two classes of `ProgressOperandLib` hold the necessary functionality to expose the custom progress operand:

- *ProgressParameterElement*

  The way of deserialization of the custom operand must be specified in a separate class representing the custom operand element. It is derived from `DataPointsParameterElement` type and decorated with an XML root attribute:

  ```
  [XmlRoot(ElementName = "progressOperand",
           Namespace = "http://Custom/ProgressOperand")]
  ```

  A few members and the `GetCollectorsFactory` method need to be overridden. This method instantiates the corresponding `DataPointsCollector` with the initial values of mandatory data point attributes.

- *ProgressDataPointsCollector*

  This class represents the object responsible for collecting data from custom operand elements. It shall implement the `IDataPointsCollector` interface and override its `Collect`

method that is invoked by the Block and returns a list of data points containing fields such as name, type and actual value of the data point, the name of source, or the sampling rate. I implemented in a separate function the logic that queries the latest entry of the progress table within the given time span of the sampling through a `TableConnector` helper class described previously. If no entity was returned, the Block will not receive a new data point.

After creating this library holding the custom operand, it should be referenced in the – now locally running – WASABi component. Furthermore, in the rules store section of the Block's App.config, the custom operand provider component needs to be registered as an extension with its assembly name in the following way:

```
<rulesStores>
  <add name="Local File Rules Store" type="[...]LocalXmlFileRulesStore"
       filename="rules.xml" certificateStoreLocation="CurrentUser">
    <extensionAssemblies>
      <add name="ProgressOperandLib" />
    </extensionAssemblies>
  </add>
</rulesStores>
```

Now the progress operand can be declared since the schema definition (XSD) of the auto-scaling rules XML file allows specifying custom operand elements with mandatory and optional attributes. The following snippet shows the section of the rules XML defining the operand and a reactive rule based on it. But as I mentioned before, progress is not a suitable counter to base rules on as it refers to searching the entire space. It cannot be determined or even estimated whether or when the decryption will finish.

```
<rules xmlns="[...]" enabled="true">
  <constraintRules>...</constraintRules>
  <reactiveRules>
    <rule name="ProgressScaleUp" enabled="true">
      <when>
        <less operand="Progress" than="25" />
      </when>
      <actions>
        <scale target="BackgroundProcessingRole" by="1" />
      </actions>
    </rule>
  </reactiveRules>
  <operands>
    <progressOperand xmlns="http://Custom/ProgressOperand"
                     alias="Progress"
                     aggregate="Max"
                     timespan="00:05:00" />
  </operands>
</rules>
```

After performing these steps, the correctness of the setup can be checked by viewing the dataset of the AutoscalerDatapoints table. If progress entities show up, it means that the Block successfully consumes the metric data provided by the custom operand source and manages to evaluate reactive rules based on progress.

Next, I intend to present the measurement results of a straightforward, longer scaling test case on a diagram visualizing all the essential monitored performance counter values.

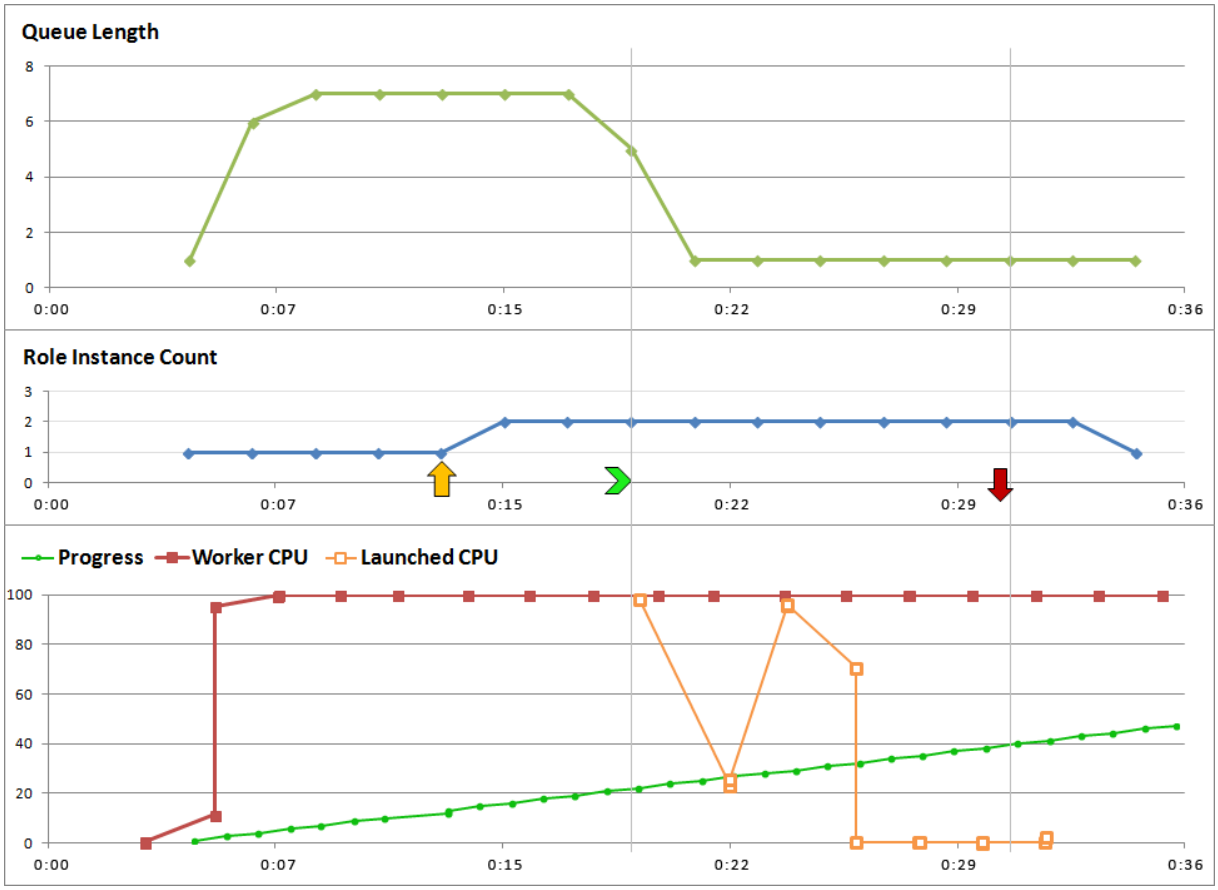

Figure 5-9. - Azure scaling test case measurement diagram

I tried to illustrate scaling with an example close to real life, in which a long-running process keeps the single running instance occupied meanwhile new requests are waiting in the queue. On the uppermost graph it is visible that a request gets pushed into the initially empty queue at 5 minutes from the start of monitoring. The single instance of Background Processing Role running instantly picks up the message from the queue and starts decoding the hash with brute force which consumes all its CPU time as it can be observed on the bottom graph. Even though the message is set to invisible in the queue by its consumer, it is still included in the message count due to the inaccuracy or flaw of the storage queue API; this is why queue length does not drop to zero. The progress data series on the lower diagram show that the decryption is rather slow but steady.

In this case I set up reactive rules for WASABi to scale the target role up or down when the queue length exceeds 5 or stays below 2 for seven minutes. I also bound the instance count between one and four, and configured the stabilizer to prevent unnecessary repeated scaling actions within a 15-minutes cool-down period. At approximately the twelfth minute, when the newly received elements in the queue have been waiting for 7 minutes, WASABi issues a scale-up operation – marked with the orange up arrow. The graph shows that it takes nearly 7 minutes for the platform to launch and set up a new instance. The instance entry point is marked with the green arrow pointing right. Just after its start, the new instance publishes performance counter values and diagnostic information, meanwhile it queries the queue and starts processing the messages. The brief decryption processes are visible on the orange graph that visualizes the CPU utilization of the launched instance. During all these, the first instance is constantly kept busy with decoding the hash first arrived.

After emptying the queue, the new instance becomes idle, its CPU usage descends to zero. The Block perceives that the queue shortened below the specified limit and as a result of rule evaluation it submits a down-scaling operation on the target role marked with the red arrow. Terminating the instance takes significantly less time than launching. It may be advisable to keep the secondary instance running while the other performs the long decryption. This could be achieved with a longer cool-down period after scaling down or by defining a reactive rule based on the CPU utilization performance counter operand aggregated over the group which only allows removing instances if the metric value stays below a small percentage value. If instant serving of requests has a priority over expenses, it is best to have multiple instances running all the time using a constraint rule with the desired minimum instance count.

5.3.7 Conclusions of scaling with WASABi

As I tried to illustrate with the case study, the Autoscaling Application Block is a flexible and powerful plug-in for Azure applications. While scaling an application running on the IaaS platform of Amazon requires mainly administrative tasks performed with the help of command line tools or the Management Console, attaching and configuring the WASABi component to an Azure application demands developer skills. This is not necessarily an advantage because dynamic scaling needs to be optimized by regularly revising its operations and adjusting the configuration accordingly. It is admitted that dynamic scaling – mainly in the introduction period – needs an administrative supervision which enables intervention in case of unwanted behavior or extraordinary external circumstances.

Among the few solutions available on the market, a custom administrative application could be created that would allow rule editing during operation. It is possible because WASABi re-parses the rules from the configured rules store at a given frequency.

I used a 3-months trial subscription during this case study, which has little limitations: nearly all the services can be fully utilized. Azure provides the new Management Portal that offers a highly responsive and simplified interface for administration. The obsolete, Silverlight-based previous portal is also available though. The main concept of Azure that it does not require special developer skills to create a cloud service was justified. Attaching WASABi for scaling is also relatively easy but there are many pitfalls during configuration.

As it was presented, monitoring the application and harvesting diagnostic data can be performed with professional tools or custom solutions. The desired scalability behavior can only be achieved through iterated analysis of performance measurements and feeding back the results into rule modification. At the early stage it is inevitable to observe and sample the incoming load pressing the application. Then after analyzing the data and recognizing trends, the scaling rule set should be modified according to the custom requirements to reach an optimized behavior.

In comparison to the Amazon case study, during the experiments I found that the trial subscription comes with a stable performance; the hosted services do not seem to have a lower priority in resource provisioning. I came to this conclusion when observing the custom progress metric, which had the same steady course in each test runs.

6 Discussion and conclusion

This final chapter tries to throw new light upon dynamic scaling and cloud computing reflecting the collected knowledge and gained experience during development and testing processes of the case study applications. The first part reevaluates the appealing characteristics of the cloud and scalability, and presents a brief feature comparison of the scaling solutions of the two platforms. The final conclusions drawn, achieved goals, and possible fields of further research are described in the second part.

6.1 Discussion

With my thesis I intended to give the reader a general overview of the field of cloud computing, its technological background and the abundant services of major providers. The benefits of cloud computing shines through but its general usage still raises a few concerns. Many corporations are afraid switching to cloud infrastructure mainly because it is remote, off premises therefore considered unsecure and unreliable. Companies are working hard to keep their sensitive information safe, so they are not willing to upload it somewhere to the cloud. On the other side, cloud providers are doing their best to ensure and certify the high levels of security in all layers of their infrastructure. Providers offer establishing secure, encrypted communication links between on-premises networks and the cloud environment, so it becomes possible to create hybrid solutions, where the sensitive information never leaves the company's perimeter and the advantages of applications running in the cloud can still be utilized.

For smaller companies the cloud holds great opportunities. Instead of buying, installing, configuring a complete hardware and software infrastructure, which requires money, time and expertise, they are able to subscribe to different cloud services that include a readily available, highly flexible environment. Establishing a complex system on-premise is costly, but it also requires continuous administration and different skills to operate and maintain all of its components. The cloud removes the strain of most administrative tasks so the experts can focus on more important things: service configuration and capacity planning. It became clear that moving to the cloud also demands IT experts, but their activities differ from planning and building a system, they must have a deep understanding of the chosen cloud platform to be able to get the most out of its capabilities. Cloud experts own a unique set of skills by knowing the methodology and best practices of moving existing architectures to the cloud. They must work in tight cooperation with the administrators and developers of the system because it is inevitable to make changes in order to reach higher compatibility with the cloud platform. However flexible cloud services may be, there still remain scenarios when moving to the cloud is impossible because of the characteristics of the infrastructure. It is a modern attitude though, to move as many corporate services to the cloud as possible because it helps reaching optimal cost and resource efficiency.

It is also possible to subscribe to SaaS services, which offer a variety of cloud software covering many areas of business processes and management. If the company does not employ complex enterprise resource planning systems or other custom, heavy business software, it would only require simple solutions supporting workflow or document management then it is a great option to choose from the wide palette of available SaaS services. Available software services offer a fully configurable interface accessible remotely from many client platforms, which gains importance with the unstoppable spread of smartphones and tablets. There is a strong connection between the thriving of cloud computing and handheld devices. Such devices have a limited amount of resources but they can reach the cloud, which has practically infinite capacity. The lion's share of the background work of user activities is processed in cloud systems.

The ever increasing number of Internet users and the growing penetration of modern handheld devices added to the unpredictable, stochastic swings of network traffic put a lot of stress to Internet services and applications available to the public. Therefore scalability became highly important. In the previous chapters I tried to shed light on the concept of scaling and emphasize the essential characteristics of existing scaling solutions. The available scaling services are built along the same concepts but they provide different scope and ways of configuration. It is worth to mention that scaling operations are fast but not instant, so the danger of crushing under a sudden peak of demand still stands. Configuring the rules of scaling requires experience in running the target application and foresight to create rules that keep up optimum performance. It also takes time to test the scaling configuration to avoid mistakes before applying the rules in real-time execution environment.

Amazon and Azure as greatest opponents try to gain advantage by offering a greater service palette combined with professional customer support that may include custom solutions devised by cloud architects. This way it becomes possible for companies with weaker IT competence to move their infrastructure – or a part of it – to the cloud by following elaborate migration plans created by the architects. Many provider independent consulting companies also offer such services.

The presented case studies fundamentally differ. PrimeFinder was a compact application built to run in IaaS environment, and scaling required a management automation framework – Chef – to take care of bootstrapping and configuring the new instances launched by the Auto-Scaling service. On the other hand, the .NET-based MD5 Decoder was being deployed and executed in Azure's PaaS compute service, the instance administration tasks were entirely took over by the platform. But scaling solutions and other utilized services have many in common so I would like to share a few comparative thoughts on them. Both Amazon and Azure offer a free trial subscription in which nearly all services can be used to a certain limit, however, Amazon provides a whole year as opposed to the three months of Azure. Both provide a continuously improving and expanding web Management Console where common administrative tasks can be performed without having deep skills. I experienced that both portals are very responsive and user-friendly. The portals show real-time, customizable diagrams of instance performance monitoring. Amazon's basic monitoring works at a 5-minute sampling which fails to log shorter events. Azure performance diagrams are not more reliable. I found that monitoring data collection had better to be delegated to API queries or specialized third-party tools.

Amazon showed better performance at launching and terminating instances but it is due to that the case study application was running on initially empty virtual machines while Azure needed not only to launch Windows-based VMs but deploy the component assemblies. Scaling configuration and rule definition is not that simple on either platform: on Amazon it is only possible through command line tools reaching the API; in Azure it requires editing XML files. It would make the process easier to perform if Amazon embedded the features of the Auto-Scaling service into the MC. WASABi configuration would be safer and easier if a tool was developed for editing rules and basic settings. Scaling actions are typically quick; according to my observations it takes under a minute for the controller to submit a scaling request upon an alarm or positive rule evaluation. Custom metric publishing from the cloud application is easier on Amazon platform as it only requires the inclusion of the AWS SDK holding the CloudWatch API commands.

Dynamic scaling can be considered a double-edged sword. In cases when the configuration is correct and rules are adjusted to a more or less predictable incoming traffic containing expectable peaks and lulls, scaling does prove its strength. Resource provisioning takes place with a dynamism that ensures that the cloud fleet size is optimal for serving the load all the time. Human intervention is only required under extreme conditions or premeditated operations. On the other hand, imprecise configuration leads to unnecessary, oscillating scaling actions which may have a negative effect on overall performance and do the opposite of cost optimization. Bottom line is that introducing dynamic scaling requires effort, analysis, and foresight in the initial phase and regular monitoring, evaluation and intervention further on.

6.2 Conclusion

The design and development procedures of the two scalable cloud applications gave me an insight of the unconventional methodology and concepts of cloud development. I tried to choose examples that show similarities to real-world, multi-tiered applications with paying heightened attention to the previously described cloud design considerations. After designing and implementing the applications I started an extensive research on Amazon Auto-Scaling and Azure WASABi by looking up both official and unofficial, experimental content sources. I managed to use the gained knowledge in practice by proper configuration of scaling. In the thesis I recorded a guide-like description of the tested scaling services.

The thesis emphasizes the importance and usefulness of embedding the feature of custom metric publishing into the cloud application and how it could improve the preciseness of scaling decisions. Practical as it may be, the methods and implementation hints are not that widespread in the available contents. I tried to collect and clarify the theorems, then supported my findings with packaging working examples of custom metrics in the case study applications.

The reader may get a hands-on experience and a deeper understanding of the case studies from the attached sources that contain all the created application code, configuration files of scaling and the cloud environment, and step-by-step guides of deployment and trial of scaling.

At the end of the case studies I elaborately described the possibilities of monitoring the dynamically scaled applications. I performed several measurements with various setups using

self-made or third-party tools. The results of performance monitoring were analyzed and used for further improving scaling behavior. I created a few complex graphs for displaying scaling events and performance data on the same diagram, which help visualize the underlying causalities between metric values and scaling actions while giving a notion about time proportions.

For the future I targeted to examine the available private cloud solutions that are being used in large business environments. I would like to explore the existing mechanisms for scaling and compare them to the scaling services of public cloud platforms. I plan to use the gathered knowledge to create a homegrown scaling solution for private clouds.

Another potential field of research is the scalability of other layers of the common application stack, such as the data, caching, network layers. In the future I plan to examine the theoretical aspects of database scaling which is a rather extensive and popular area nowadays. I would like to discover the existing solutions and compare the different methods by experimentation.

I hope that my thesis will be a useful reading for those who are getting to know the cloud phenomenon and it will provide guidance for software developers and administrators to create and maintain applications built for scale.

List of figures

References

[1] National Institute of Standards and Technology: The NIST Definition of Cloud Computing (Special Publication 800-145, 2011)

[2] National Institute of Standards and Technology: Guidelines on Security and Privacy in Public Cloud Computing (Special Publication 800-144, 2011)

[3] Christine Burns: 10 most powerful IaaS companies (2012), http://www.networkworld.com/supp/2012/enterprise2/040912-ecs-iaas-companies-257611.html

[4] Mark D. Hill: What is scalability? (SIGARCH Comput. Archit. News 18, 4, December 1990, 18-21, DOI=10.1145/121973.121975)

[5] Royans Tharakan: What is scalability? (2007), http://www.royans.net/arch/what-is-scalability/

[6] Luis M. Vaquero, Luis Rodero-Merino, Rajkumar Buyya: Dynamically Scaling Applications in the Cloud (ACM SIGCOMM Vol. 41, 2011, DOI=10.1145/1925861.1925869)

[7] Guy Harrison: 10 things you should know about NoSQL databases (2010, techrepublic.com), http://www.techrepublic.com/blog/10things/10-things-you-should-know-about-nosql-databases/1772

[8] Database Sharding (dbShards Whitepaper, CodeFutures), http://www.codefutures.com/database-sharding/

[9] Jinesh Varia: Architecting for the Cloud - Best Practices (2011, Amazon Web Services Whitepaper)

[10] David Chappell: The Windows Azure Programming Model (Whitepaper, 2010), http://www.davidchappell.com/writing/white_papers/The_Windows_Azure_Programming_Model_1.0--Chappell.pdf

[11] Brian Adler: Building Scalable Applications in the Cloud, Reference Architecture and Best Practices (RightScale Whitepaper, 2011), http://www.rightscale.com/info_center/white-papers/RightScale_White_Paper_Building_Scalable_Applications.pdf

[12] Matt Tavis: Web Application Hosting in the AWS Cloud - Best Practices (2010, Amazon Web Services Whitepaper), http://media.amazonwebservices.com/AWS_Web_Hosting_Best_Practices.pdf

[13] Wikipedia, Content Delivery Network, http://en.wikipedia.org/wiki/Content_delivery_network

[14] AWS Auto Scaling Developer Guide, http://docs.amazonwebservices.com/AutoScaling/latest/DeveloperGuide/

[15] AWS CloudWatch API Reference, http://docs.amazonwebservices.com/AmazonCloudWatch/latest/APIReference/

[16] David Chappell: Introducing the Windows Azure Platform (Whitepaper, 2010), http://www.davidchappell.com/writing/white_papers/Introducing_the_Windows_Azure_Platform,_v1.4--Chappell.pdf

[17] Understanding Windows Azure, http://www.windowsazure.com/en-us/develop/net/fundamentals/intro-to-windows-azure/

[18] Windows Azure Queues and Windows Azure Service Bus Queues - Compared and Contrasted, MSDN, http://msdn.microsoft.com/en-us/library/hh767287(v=vs.103).aspx

[19] The Design of the Autoscaling Application Block, MSDN, http://msdn.microsoft.com/en-us/library/hh680880(v=pandp.50).aspx

[20] Autoscaling and Windows Azure, MSDN, http://msdn.microsoft.com/en-us/library/hh680945(v=pandp.50).aspx

[21] Understanding Rule Ranks and Reconciliation, MSDN, http://msdn.microsoft.com/en-us/library/hh680923(v=pandp.50).aspx

[22] How to Use the Autoscaling Application Block, Azure Tutorial Guide, http://www.windowsazure.com/en-us/develop/net/how-to-guides/autoscaling/

[23] Paul Bouwer: Autoscaling Azure with WASABi (Case study, 2012), http://blog.paulbouwer.com/2012/08/21/autoscaling-azure-with-wasabi-part-1/

[24] Creating a custom logger for WASABi, MSDN, http://msdn.microsoft.com/en-us/library/hh680926(v=pandp.50).aspx

[25] Creating a custom operand for WASABi, MSDN, http://msdn.microsoft.com/en-us/library/hh680912(v=pandp.50).aspx

[26] What Is Google App Engine?, Google Developers, https://developers.google.com/appengine/docs/whatisgoogleappengine

[27] Storing Data, Google Developers, https://developers.google.com/appengine/docs/java/datastore

[28] The Task Queue Java API, Google Developers, https://developers.google.com/appengine/docs/java/taskqueue

[29] Overview of creating and using performance counters, MSDN, http://msdn.microsoft.com/en-us/library/windowsazure/hh411520.aspx

[30] Amazon AWS EC2 API Tools reference, AWS documentation http://docs.amazonwebservices.com/AWSEC2/latest/CommandLineReference/command-reference.html

[31] Amazon Machine Images, https://aws.amazon.com/amis

[32] Opscode Chef architecture introduction,
http://wiki.opscode.com/display/chef/Architecture+Introduction

[33] Edward Sargisson: Keeping an Amazon EC2 instance up with Chef and Auto Scaling (Trailhunger.com, Case study 2011),
http://www.trailhunger.com/blog/technical/2011/05/28/keeping-an-amazon-elastic-compute-cloud-ec2-instance-up-with-chef-and-auto-scaling/

Appendix A

Folder structure of the attachment:

[Scaling_CaseStudy_Amazon]	
|-> [AWSWatch] |	The self-made monitoring tool for AWS based on .NET SDK of CloudWatch.
|-> [bin]	Executable files for trying AWSWatch.
|-> [source]	Contains the complete Visual Studio solution.
|-> [Chef-Cookbooks] | | | |	This folder holds all the necessary community and custom made Chef cookbooks for performing the experiment according to the guide. Some of them needs configuration.
|-> [apache2] |	Community cookbook for installing Apache2, contains some modifications.
|-> [awssdk]	This cookbook installs the AWS SDK for PHP.
|-> [curl]	
|-> [openssl]	
|-> [primefinderapp]	Custom cookbook for deploying the scalable PrimeFinder application.
|-> [Measurement-Results]	Performance measurement data sets.
|-> [Source]	PHP files also included in PrimeFinder cookbook.
|-> [UserDataScript]	Script for instance bootstrapping by Chef.
[Scaling_CaseStudy_Azure]	
|-> [AutoscalerConsole]	Solution of locally running host of WASABi along with properly set XML configuration files.
|-> [IcetronicsAzure] |-> [AutoscalingHostWorkerRole] |-> [BackgroundProcessingRole] |-> [FrontendWebRole2] |-> [IcetronicsAzure]	The Visual Studio solution of the scalable MD5 Decoder application (under a different name) complete with the separate projects of the cloud service roles and deployment settings.
|-> [Measurement-Results]	Performance measurement data sets with detailed logs created by the Autoscaling Application Block during operation.

Appendix B

Amazon Web Services - PrimeFinder scalable cloud application deployment guide

Preparations and creating database node

1. Subscribe to AWS Free with a newly created account.
    - http://aws.amazon.com/free
2. Log in to the AWS Management Console (MC) and take a few preparatory steps.
    - https://console.aws.amazon.com/ec2/
    - Create a new Security Group called PrimeSG that will allow web and MySQL connections among instances. Add the following inbound rules: SSH (22), HTTP (80), MySQL (3306).
    - Create a Key Pair named PrimeKP and download its private key (.pem) file. Change the permissions of the downloaded key file to 700, thereby limiting access to it.
3. On the interface of the MC, launch a new instance, which will serve as the database node in the infrastructure.
    - Choose a Linux-based built-in AMI (the newest Ubuntu is chosen for this guide).
    - Place the instance in the PrimeSG Security Group.
    - Wait for the instance to launch and make a note of its public DNS name through which it is accessible over the Internet.
4. Connect to the database instance through SSH with the following command:

```
ssh ec2-x-x-x-x.eu-west-1.compute.amazonaws.com -l ubuntu -i ~/PrimeKP.pem
```

Configuring the database node

5. Install MySQL Server on the database instance via its terminal, make note of the chosen root password.

```
sudo apt-get install mysql-server
```

6. Change the MySQL Server bind address in the configuration file to the public DNS name of the node:

```
sudo nano /etc/mysql/my.cnf
 >> bind address = ec2-x-x-x-x.eu-west-1.compute.amazonaws.com
```

7. Enter the MySQL command interface as root:

```
mysql -u root -p
```

    - Create the necessary database and table.

```
CREATE DATABASE primedb
USE primedb
CREATE TABLE users (
    userid INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(100),
    password VARCHAR(100));
```

    - From the MySQL console, grant privileges to root to allow remote connections:

```
GRANT ALL ON *.* TO 'root'@'%' IDENTIFIED BY 'rootpassword';
```

Setting up Chef for the scalable infrastructure

8. Sign up for a free Hosted Chef account.
    - http://www.opscode.com/hosted-chef/
9. Prepare your workstation for remote Chef management by following the steps below.
    - Install Ruby,the necessary dependencies, and Chef:

```
sudo apt-get install ruby ruby-dev libopenssl-ruby rdoc ri irb build-essential wget ssl-cert curl
sudo gem install chef --no-ri --no-rdoc
```

    - Create a local Chef repository:

```
cd ~
curl https://nodeload.github.com/opscode/chef-repo/tarball/master >
     chef-repo.tgz
tar xvf chef-repo.tgz
mv -i opscode-chef-repo-a3bec38/ chef-repo
```

    - Create configuration folder and save public key files from Opscode Chef Management Console (username and organization key files and knife.rb configuration file):

```
mkdir -p ~/chef-repo/.chef
cp USERNAME.pem ~/chef-repo/.chef
cp ORGANIZATION-validator.pem ~/chef-repo/.chef
cp knife.rb ~/chef-repo/.chef
```

    - Launch automatic configuration, then confirm if it is working properly:

```
knife configure
knife client list
```

10. Prepare the Apache cookbook necessary for the future web nodes.
    - Download the prepared Apache cookbook from Opscode Community:

```
knife cookbook site download apache2
```

    - Add the following two blocks to the appropriate section of the mod_php5 recipe of the Apache cookbook in the local repository:

```
package "php5-mysql" do
    action :install
end
package "libapache2-mod-auth-mysql" do
    action :install
end
```

    - Upload the edited Apache cookbook to Hosted Chef server with knife:

```
knife cookbook upload apache2
```

11. Create a custom cookbook that installs curl necessary for using the AWS SDK.
    - Create the skeleton of the curl cookbook with knife:

```
knife cookbook create curl
```

    - Delete all but the recipes directory and paste the following Ruby-blocks into the default recipe (default.rb):

```
case node[:platform]
when "debian", "ubuntu"
  package "curl" do
    action :install
  end
```

```
  package "libcurl3" do
    action :install
  end

  package "libcurl3-dev" do
    action :install
  end

  package "php5-curl" do
    action :install
  end
end

service "apache2" do
  case node[:platform]
  when "debian","ubuntu"
    service_name "apache2"
    restart_command "/usr/sbin/invoke-rc.d apache2 restart && sleep 1"
    reload_command "/usr/sbin/invoke-rc.d apache2 reload && sleep 1"
  end
  supports value_for_platform(
    "debian" => { "4.0" => [ :restart, :reload ], "default" => [ :restart,
:reload, :status ] },
    "ubuntu" => { "default" => [ :restart, :reload, :status ] }
  )
  action :enable
end

service "apache2" do
  action :restart
end
```

Note: this recipe will only work on Ubuntu/Debian platform that is used in this guide. It also contains code necessary for restarting Apache after performing package installation.

- Upload the cookbook to hosted Chef:

```
knife cookbook upload curl
```

12. Create a custom cookbook that deploys and configures the AWS SDK for PHP.
    - Define custom attributes for authentication and fill in the values.

```
default[:awssdk][:installdir]   = "/var/www/awssdk"
default[:awssdk][:key]          = "AWS Access Key"
default[:awssdk][:secretkey]    = "AWS Secret Access Key"
```

Note: these access keys can be found after logging in to AWS under Security Credentials.

- Under the templates/default folder, create the config.inc.php.erb template file with the placeholders, which will be copied as the configuration file required by the AWS SDK :

```
<?php if (!class_exists('CFRuntime')) die('No direct access allowed.');
CFCredentials::set(array(
	'development' => array(
		'key' => '<%= node[:awssdk][:key] %>',
		'secret' => '<%= node[:awssdk][:secretkey] %>',
		'default_cache_config' => '',
		'certificate_authority' => false
	), '@default' => 'development'
));
```

- Create the default recipe with the following content:

```
case node[:platform]
when "debian", "ubuntu"
  package "git" do
    action :install
  end
end
directory "#{node[:awssdk][:installdir]}" do
	owner "root"
	mode "0755"
end
bash "git_clone_awssdk" do
	cwd "#{node[:awssdk][:installdir]}"
	code <<-EOH
	git clone git://github.com/amazonwebservices/aws-sdk-for-php.git .
	EOH
end
template "config.inc.php" do
	path "#{node[:awssdk][:installdir]}/config.inc.php"
	source "config.inc.php.erb"
	owner "root"
	group "root"
	mode "0755"
end
```

Note: these blocks are necessary to install git that will clone the AWS SDK to the configured directory.

```
knife cookbook upload awssdk
```

13. Create and prepare a custom cookbook containing the PrimeFinder application.
    - Create the cookbook in the local repository with knife:

```
knife cookbook create primefinderapp
```

    - Add custom attributes to primefinderapp/attributes/default.rb, such as the public DNS name of the database node EC2 instance and the configured MySQL credentials:

```
default[:primefinderapp][:wwwroot]     = "/var/www"
default[:primefinderapp][:db_address]  =
    "ec2-x-x-x-x.eu-west-1.compute.amazonaws.com"
default[:primefinderapp][:db_user]     = "root"
default[:primefinderapp][:db_password] = "rootpassword"
default[:primefinderapp][:db_database] = "primedb"
```

- Copy the PrimeFinder index.php under primefinderapp/files/default folder.
- Copy the PrimeFinder dbconn.php under primefinderapp/templates/default folder, rename it to dbconn.php.erb and replace its content with the following:

```
<?php
	$db_address  = "<%= node[:primefinderapp][:db_address] %>";
	$db_user     = "<%= node[:primefinderapp][:db_user] %>";
	$db_password = "<%= node[:primefinderapp][:db_password] %>";
	$db_database = "<%= node[:primefinderapp][:db_database] %>";
?>
```

- Insert the following blocks into the cookbook's default recipe under primefinderapp/recipes:

```
template "dbconn.php" do
	path "#{node[:primefinderapp][:wwwroot]}/dbconn.php"
	source "dbconn.php.erb"
	owner "root"
	group "root"
	mode "0755"
end

cookbook_file "#{node[:primefinderapp][:wwwroot]}/index.php" do
	source "index.php"
	owner "root"
	group "root"
	mode "0755"
end

cookbook_file "#{node[:primefinderapp][:wwwroot]}/cw-put-metric.php" do
	source "cw-put-metric.php"
	owner "root"
	group "root"
	mode "0755"
end

cookbook_file "#{node[:primefinderapp][:wwwroot]}/ec2-instances.php" do
	source "ec2-instances.php"
	owner "root"
	group "root"
	mode "0755"
end
```

- Finally, upload the prepared cookbook to Hosted Chef using knife:

```
knife cookbook upload primefinderapp
```

14. Log in to Chef Management Console, check if the cookbooks are present, then create a new role called web and specify its run list: drag and drop the following recipes among the available to the default run list: apache2, apache2::mod_php5, curl, awssdk, primefinderapp.

Creating the bootstrap file for dynamically created web nodes

15. Paste the following script content into an empty file and save it named bootscript.
    - Replace <ORG_NAME> placeholders with your own organization name in Chef.
    - Insert your public key file content to the last section.

```
#!/bin/bash
exec > >
  (tee /var/log/user-data.log|logger -t user-data -s 2>/dev/console) 2>&1
apt-get update
APT_GET="env DEBIAN_FRONTEND=noninteractive
            DEBIAN_PRIORITY=critical apt-get -q"
$APT_GET -y remove ruby1.8*
$APT_GET -y install ruby1.9.1 ruby1.9.1-dev libruby1.9.1
$APT_GET -y install build-essential
ln -sf gem1.9.1 /usr/bin/gem
gem install --no-rdoc --no-ri chef
ln -sf ruby1.9.1 /usr/bin/ruby
mkdir -p /var/log/chef
mkdir -p /var/backups/chef
mkdir -p /var/run/chef
mkdir -p /var/cache/chef
mkdir -p /var/lib/chef
mkdir /etc/chef
ln -s /var/lib/gems/1.9.1/bin/chef-client /usr/bin/chef-client
cat - >/etc/chef/bootstrap.json <<EOF
{
"run_list": [ "role[web]" ],
"default_attributes": { },
"override_attributes": { }
}
EOF
cat - >/etc/chef/client.rb <<EOF
log_level          :info
log_location       "/var/log/chef/client.log"
ssl_verify_mode    :verify_none
validation_client_name "<ORG_NAME>-validator"
validation_key         "/etc/chef/validation.pem"
client_key             "/etc/chef/client.pem"
chef_server_url    "https://api.opscode.com/organizations/<ORG_NAME>"
file_cache_path    "/var/cache/chef"
file_backup_path   "/var/backups/chef"
pid_file           "/var/run/chef/client.pid"
node_name           "`curl http://169.254.169.254/latest/meta-data/instance-
id`"
Chef::Log::Formatter.show_time = true
EOF
```

```
cat - >/etc/chef/validation.pem <<EOF
-----BEGIN RSA PRIVATE KEY-----
<...PUBLIC KEY FILE CONTENT...>
-----END RSA PRIVATE KEY-----
EOF
chef-client -j /etc/chef/bootstrap.json
```

Configuring AWS Auto Scaling service

16. Log in to AWS MC and create a Load Balancer named PrimeELB with the following settings:
    - Load Balancer Protocol: HTTP (port 80)
    - Ping settings: protocol - HTTP, port - 80, path - "/"
    - Advanced settings may remain default.
17. Install AWS Auto Scaling and CloudWatch Command Line Tools to workstation.
    - Auto Scaling: http://aws.amazon.com/developertools/2535
    - CloudWatch: http://aws.amazon.com/developertools/2534
    - Follow the instructions of the readme files, set the proper environment variables.
    - Authentication may take place using a credential file containing AWS keys or with a X.509 certificate and the corresponding private key file. The readme files provide thorough guides for setting up authentication.
    - Test the configuration by issuing this command:

```
as-describe-auto-scaling-groups
```

18. Create a properly parametered custom Launch Configuration with the following command:

```
as-create-launch-config PrimeLC
    --image-id ami-8d5069f9    --instance-type t1.micro
    --region eu-west-1    --group PrimeSG    -monitoring-disabled
    --key PrimeKP    --user-data-file bootscript
```

    Note: it is advisable to select the AMI of the most up-to-date Ubuntu Linux operating system instead of the one used in this launch configuration.

19. Create an Auto Scaling Group with the following options:

```
as-create-auto-scaling-group PrimeASG
    --launch-configuration PrimeLC
    --region eu-west-1    --availability-zones eu-west-1a
    --min-size 1    --max-size 4
    --load-balancers PrimeELB    --health-check-type EC2
```

    - After issuing this command, the Auto Scaling Group will instantaneously launch a new web instance to reach the specified minimum size.
    - The following command can be used to deprovision all running web nodes:

```
as-update-auto-scaling-group PrimeASG
    --region eu-west-1    --availability-zones eu-west-1a
    --min-size 0    --max-size 0
```

20. Create the policy pair defining the scaling actions:

```
as-put-scaling-policy ScaleUpPolicy
    --auto-scaling-group PrimeASG
    --type ChangeInCapacity    --adjustment=1
    --cooldown 300    --region eu-west-1
```

```
as-put-scaling-policy ScaleDownPolicy
    --auto-scaling-group PrimeASG
    --type ChangeInCapacity    --adjustment=-1
    --cooldown 300    --region eu-west-1
```

21. With the CloudWatch API command, create the alarms based on the CPU utilization metric:

```
mon-put-metric-alarm HighCPUAlarm
    --metric-name CPUUtilization    --namespace "AWS/EC2"
    --comparison-operator GreaterThanThreshold    --threshold 80
    --period 60    --statistic Minimum    --evaluation-periods 3
    --alarm-actions "[unique ARN of ScaleUpPolicy]"
    --dimensions "AutoScalingGroupName=PrimeASG"    --region eu-west-1

mon-put-metric-alarm LowCPUAlarm
    --metric-name CPUUtilization    --namespace "AWS/EC2"
    --comparison-operator LessThanThreshold    --threshold 20
    --period 60    --statistic Maximum    --evaluation-periods 3
    --alarm-actions "[unique ARN of ScaleDownPolicy]"
    --dimensions "AutoScalingGroupName=PrimeASG"    --region eu-west-1
```

22. Open the DNS name of the PrimeELB load balancer in a browser, register a new user then start finding prime numbers. This operation will consume most of the instance resources and the node will become unresponsive. Meanwhile, check the alarm and watch the scaling action take place by itself. The load balancer will redirect the incoming requests to the newly launched instance.