Migration of Legacy Applications to the Cloud

Pedro Nunes
Instituto Superior Técnico
Lisbon, Portugal
pedro.l.nunes@tecnico.ulisboa.pt

Prof. José Alves Marques
Instituto Superior Técnico
Lisbon, Portugal

ABSTRACT

Cloud computing has brought an opportunity for businesses to modernize their legacy applications. Migrating legacy applications to the cloud and being able to successfully leverage its characteristics has become a major objective for several companies whose systems rely on old technologies and perform poorly when compared to what the cloud has to offer. This project provides a general state-of-the-art regarding this topic and proposes some alternative cloud-oriented architectures for a specific legacy application - EDOCLINK. It describes the process of migrating certain components to the cloud as well as a performance comparison of both its versions.

Author Keywords
Cloud Computing; Legacy Application; Cloud Migration

INTRODUCTION

Cloud computing is a term often used as a keyword that represents a lot of different ideas which contributes to a general confusion on what it really means. As provided by NIST, the American National Institute of Standards and Technology, 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* [3]. This definition points out some key characteristics of cloud computing that are the reason for its growing popularity. It provides on-demand access to computing resources which means that even smaller scaled businesses are able to leverage these resources into their favor since the cost of such infrastructure is calculated in a pay-as-you-go methodology, i.e. it is proportional to the capabilities of the hardware and the amount of time in which it was effectively used. When deploying an application, these companies were often faced with a dilemma: they would either cut costs in order to build an infrastructure as small as possible to handle its current needs and face the risk of not being able to scale if the demand for its application increases or accommodate potentially heavier loads and face the risk of never actually having big enough demand that justifies the size of the infrastructure.

Cloud computing definition comprises both the applications provided over the Internet and the infrastructure allocated in the data centers that support them [1]. It allows services to be deployed on the Internet without the need to build or maintain the hardware required to run it. It also provides the ability to deploy a service in a given environment whose capabilities are not static i.e. the risk of building a huge infrastructure for an application that turns out to be a failure or, the opposite, building a small scale hardware that is only able to provision a small fraction of its users is eliminated by the elasticity of such resources. The hardware and software stored in the data centers to which these applications are deployed is called the *Cloud*. Companies are actively trying to leverage this platform into their favor [2] by either developing new software to specifically operate in the cloud or by migrating legacy *on-premise* applications to it.

This report focuses on the process of migrating a legacy application to the cloud. Current state-of-the-art will be presented containing some examples of methodologies and techniques developed to guide the migration process, some related technologies that are often compared to the cloud and the clarification of what is the application to be migrated and its current architecture. The next step is to propose some new architectural approaches suitable for the cloud, comparing them against each other and pick the most suitable for this specific case. The main objective is to achieve a functioning version of the application which utilizes cloud services and whose performance is improved when compared to the current version, measured by some metrics that will be discussed further.

This idea of combining computing resources isn't new. For example, Grid Computing is a model of distributed computing that allows a user to access network resources separated among several computing units whose capabilities are combined in order to achieve a higher processing power [3]. This model is related to cloud computing since it also makes use of several resources loosely distributed however, grid computing is often owned, managed and used within an organization to process a single task in opposition to cloud computing whose infrastructure is owned by a central cloud provider and may be in physically distant locations.

Besides its similarities to Grid Computing, Cloud Computing relies on the key concept of Virtualization which is a way to abstract the physical hardware and provide virtualized resources to an application [3]. This is often achieved by the introduction of a layer - the Hypervisor or Virtual Machine Manager - that is responsible for allowing multiple operating systems and the

applications that each of them is running to share the resources of the hardware structures [4]. Virtualization allows multiple applications to run on the same hardware whose resources are more efficiently utilized. This technology is one of the fundamentals of cloud computing since it provides the capability of assigning computing resources dynamically and on-demand to multiple applications.

Despite this being the most common definition of Virtualization, the term itself has gained a new meaning with the introduction of Container-based Virtualization. In opposition to VMs, containers do not get their own virtualized hardware; they use the hardware of the host system. Containers offer a level of abstraction from the environment in which the application is executed. Containers are shipped with the whole package of libraries and dependencies needed to run the application, which avoids dependency conflicts and separates different projects or applications easily [6]. They are extremely lightweight in comparison to VMs since they do not need to contain every component required to run the operating system. These characteristics have made container-based virtualization very popular in the last few years.

Cloud service providers, as are known the companies that provide services to run application in the cloud, leverage these virtualization technologies to offer computing resources to its users that are adjusted on-demand and billed in a pay-as-you-go method. However, IT organizations, as any other type of organizations, have very different needs and business models. Given this granularity, cloud providers offer services that are grouped in mainly 3 different groups, which are:

Infrastructure as a Service which is a type of cloud architecture in which the hardware/infrastructure is virtualized in form of VM's. Clients often choose this architecture because it is easier and cheaper to deploy the application in a remote infrastructure instead of having to buy, build and manage it entirely. Example: Amazon Web Services (AWS) although several available services in AWS are comparable to PaaS.

Platform as a Service which is a type of cloud architecture which introduces an additional level of abstraction compared to IaaS, providing a development environment or API allowing the customer to develop the desired application within the given environment. In this architecture, the provider hosts the application in its own infrastructure and makes it available through the internet. Example: Azure, Google App Engine.

Software as a Service which is a type of cloud architecture in which the provider is responsible for managing and controlling all the underlying infrastructure, providing a software in the form of an internet service. The client does not control any part of the stack since it is controlled by a third party. Typically, the end user accesses the software through a web browser or a simple terminal and updates and upgrades are managed by the provider. Example: iCloud, Microsoft Office.

Concerning cloud providers, there are 3 main players in the Cloud Computing scene: Amazon with AWS (Amazon Web Services), Google with Google Cloud and Microsoft with Azure. These 3 main providers offer a lot of similar features.

In the scope of this project, the main objective consists in deploying a cloud version of EDOCLINK. EDOCLINK is a document management platform which allows its users to control the lifecycle and tasks related to a document. It also allows the organization of a certain process and the tasks and documents related to such process.

At its current state, EDOCLINK´s technical architecture is composed by an Applicational Server, a Database and some satellite components responsible for specific tasks. A main installation of EDOCLINK in a client consists in deploying a physical server containing an instance of SQL Server as the database and the rest of the components. This type of deployment is limited by the capabilities of such server and adjusting its power to the changing needs of the organization requires manually adjusting its hardware.

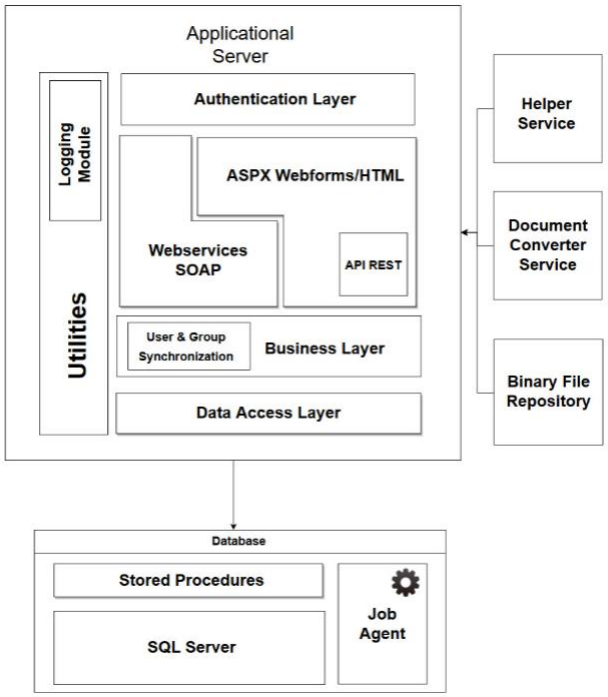


Figure 1 - EDOCLINK Logical Architecture

The figure above represents the current state of EDOCLINK's architecture. Starting from the satellite components, the Helper Service is responsible for handling alarms and background tasks periodically. It is deployed as a Windows service in the physical service and, according to the period configured, communicates with the Application Server via SOAP service. Firstly, it retrieves a list of pending alarms. According to the type of alarm retrieved, it contacts another SOAP webservice in the Applicational Server that is responsible for handling the execution of such alarm. These alarms can range from sending internal notifications to the user because a certain deadline is reached to sending e-mails.

The Binary File Repository is responsible for storing the documents introduced in EDOCLINK. In the current development version, this component can be configured to be filesystem based, meaning that the documents and files are stored in a certain directory within the server, or Azure based meaning that it is possible to use Azure Files or Azure Blobs, both Azure services capable of storing files and documents which can be accessed from anywhere.

Lastly, the Document Converter Service is responsible for converting documents to an image format. This component can also retrieve the total number of pages of a certain document. It is used to convert the first page of a document to an image and resize it to a thumbnail so that it can be showed in the user interface. When browsing to a certain document, the user is taken to a document visualizer. In this visualizer, the Document Converter Service is, in the first place, responsible for returning the total number of pages. Then, it converts the first page and presents it. It also immediately converts the second page, to have it ready once the user requests it.

Concerning the Applicational Server, it contains most of the functionalities of EDOCLINK The Business Layer holds most of the EDOCLINK's logic, being responsible for receiving requests from the user interface or from a SOAP Web Service, process them and, if needed, conduct them to the Data Access Layer that will be responsible for triggering stored procedures in the database that collect several information related to documents, users or distributions. The Authentication Layer aims at providing Single Sign-On capabilities to EDOCLINK by allowing users from a certain organization to log in EDOCLINK by using the same credentials in their own domain, using Active Directory. If requested, it is also capable of providing access to EDOCLINK via an external provider as Twitter or Facebook.

EDOCLINK sets a GUID – Globally Unique Identifier – to each object of its objects, such as users, documents or distributions. This GUID identifies such object uniquely and is used to request information regarding it.

EDOCLINK – New Architecture

Concerning the main objective of deploying a working version of EDOCLINK in the Cloud, it was necessary to modify its current architecture in order to maximize the benefits that it has to offer. Currently, EDOCLINK follows a monolithic architecture in which the core of its functionalities are concentrated in a single module, the Applicational Server. Although the migration of the entire application was the main objective, it ended up being too ambitious. In order to fit in the time constraints if this project, there was the need to decide which components should be the target of this migration. Given this, the option went to the satellite components, because of the isolated nature of their functionalities. The migration of such components to Cloud services will maintain their functionality while providing the opportunity to bring improvements in its scalability and some room to apply some changes to the way some modules are implemented.

After selecting the targets of the migration, there was the need to select the most appropriate way to do it. Starting by the Binary File Repository, as stated before, there was already the option to use an Azure service to store documents in the Cloud. So, in the version presented in this report, this component will only be deployed as an instance of Azure Blobs, that allows storing of large amounts of data accessed by several ways including a library provided by Azure to multiple programming languages. Despite the migration process being relatively straightforward, it was necessary to rearrange the way it communicated with the Document Converter Service which was the target of the most profound changes. In its actual architecture, the Document Converter Service holds a cache of the images

previously converted. This cache consists in a configurable directory which contains the documents whose conversions were requested previously as well as the images of each page as well. With the objective of improving the process of deployment of a new version of this module in the Cloud, it was decided to containerize it along with its libraries and dependencies. The custom container image is then stored in an Azure Container Registry, which is a service capable of storing custom container images which can be easily deployed by using the functionality Web App for Containers. This functionality allows the creation of a web application from a given container image using the Azure App Service, capable of deploying the container as a web application and providing several configurations as well as the possibility of scalability. The objective with this migration is to be able to scale to accommodate load peaks, by improving the number of active instances able to process a conversion or by increasing the capabilities of such service in terms of the computing power assigned to it. However, if the Document Converter Service consists in a single module responsible for handling the cache management and performing the actual conversion, the creation of a mechanism that would replicate the cache state to every active instance when scaling up would be necessary. To avoid this problem, it was decided that the functionalities of this module would be split in two: a sub-module responsible for caching and managing the access of a user to a document and another sub-module responsible for the conversion itself.

Therefore, an API that receives requests from the Applicational Server to convert a certain page of a document was created. This module, defined as DCSLogic from now on, receives request containing the GUID of the document to convert, the page requested and a flag defining the need to resize it into a thumbnail format. This request should also contain an authentication header consisting in a JSON Web Token. A JWT consists in string representing a set of claims related to the identification of the user requesting the conversion, in the form of its GUID. DCSLogic is then responsible for confirming the authenticity of such request by evaluating the expiration timestamp of the JWT as well as comparing its contents to its signature which was built using a secret string known to both parties.

After confirming the access, it will then look for the requested image in the cache and, if it does not exist, request it to the second sub-module responsible for the conversion itself, the DCSConverter. A more detailed explanation a single conversion lifecycle will be offered later.

Concerning the Helper Service and the need to periodically trigger its execution, the most suitable Azure service for this module is Azure Functions. It allows a certain functionality to be ran according to a certain trigger. It can be fired by the reception of a request by other application or by performing a change in a certain file stored in Azure Blobs or Azure Files. However, in this case, the trigger used was a Timer Trigger to which we can assign a CRON expression, which represents the amount of time between executions of such function.

The Applicational Server and the database were only deployed to a VM in Azure, having suffered no changes. The final EDOCLINK's architecture in this Cloud version is as follows:

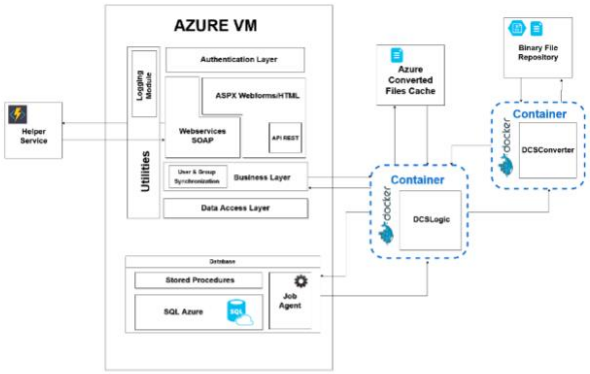
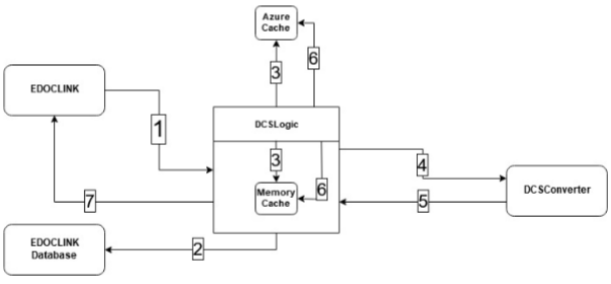


Figure 2 - EDOCLINK Cloud Architecture

The first step of this migration consisted in deploying a working version of EDOCLINK in a Azure VM. This VM holds the Applicational Server and the database as a SQL Server instance. At first, the hardware provided to this VM was not enough to provide a smooth user experience. In order to get closer to what resembles a production environment of EDOCLINK, it was necessary to increase the capabilities of such VM to the category F2S provided by Azure, which uses 2 virtual CPU cores and 4GB of RAM, as well as a data disk with 6400 IOPS. Every test executed in the scope of this project uses this VM configuration.

Concerning the migration of the Document Converter Service, the following figure represents its technical architecture as well as the lifecycle of a conversion:



1. The DCSLogic receives a conversion request from the Applicational Server containing the document GUID, a thumbnail flag and the page to convert as parameters. It

should also contain na authentication header containing the JWT with the user GUID.

2. Using the document and the user GUID, query the database to verify if such user can access the document.
3. In case the access is denied, respond to the request with a 401 HTTP status code. Else, verify is the selected page exists in the cache.
4. In case the selected page exists in cache, respond to the request with such image encoded in base-64. In case it doesn't, a new request is made to the DCSConverter.
5. The DCSConverter accesses the document and converts the desired page, returning it encoded in base-64.
6. The DCSLogic receives the response to the request from the DCSConverter and stores the image in the cache along with the parameters of the request such as page number and the thumbnail flag.
7. The DCSLogic responds to the original request with the requested image encoded in base-64.

One of the biggest changes applied to this module, apart from the decomposition in 2 smaller modules, was the cache. It is now composed by a smaller size, faster access cache and a bigger size but slower access cache as a file repository in Azure Files. The new version of the cache consists in a set of key-value pairs stored in memory. The key to each entry is built from the document GUID, the thumbnail flag and the page number. The value of each entry consists in an object containing the image encoded in base-64, two Booleans representing the existence of such image in memory or in the Azure Files repository and a string containing the URL to the image encoded in base-64 in the Azure Files repository. The maximum cache size as well as the maximum amount of time that a certain entry should be kept in cache is configurable. When it is reached, the memory cache instance automatically removes the least used entries and a callback is triggered. This callback was programmed to reintroduce the same exact entry in the cache but erasing the field containing the image in base-64. Since most of the memory space occupied by a given entry is used to store this field. Re-adding it to the cache will maintain the converted image in the Azure Files repository and the URL to access it. If such image is requested, the DCSLogic will transfer it from the Azure files repository and populate the field containing it in the memory cache again, updating the Boolean values. When a new image is converted, its base-64 encoded string is added to both the memory cache and the Azure Files repository. The only way for a given entry to reside only in the Azure Files repository is when the maximum amount of time is reached or the maximum cache size is surpassed and that entry is selected as one of the least accessed ones, removing it from the memory cache but keeping it in the persistent module of the cache.

Concerning the DCSConverter, the conversion process was kept the same, using Aspose which supports manipulation and conversion of several different file types. It receives requests containing the path in which the file is stored in the Azure Blobs repository, accesses it, converts it to jpeg and resizes it to a thumbnail format of necessary. This module is completely stateless, meaning that each request is handled completely separately from the next ones and the previous ones. This property will come in handy when scaling it horizontally i.e. increasing the number of instances of the DCSConverter able to process conversion requests.

In terms of the migration of the Helper Service, besides migrating the code as-is to an Azure Function, there were a couple of changes that had to be made. The current version of the Helper Service was built under a Provider model. This is a design pattern created by Microsoft which allows a certain component to have multiple implementations. At runtime, the adequate provider is used to perform a certain task. This model was used to create multiple implementations of the handler of an alarm so that, according to its type, the correct provider was used to handle it. With the changes that this component has suffered throughout the years, such model is not used anymore. In fact, the Helper Service is now responsible for periodically executing a SOAP web service call to retrieve the list of pending alarms and then, for each alarm, execute another SOAP web service call that handles the treatment of such alarm. This way, this model was removed from the Helper Service.

When deploying a new Azure Function, one must choose between version 1.x or version 2.x. Version 1.x only exists to provide compatibility with older applications, being in version 2.x where updates are constantly being made. However, for C# applications, version 1.x targets .NET Framework and version 2.x target .NET Core. Given the effort being made in the migration of these components, it was decided that the new Helper Service Azure Function should be as updated as possible and so, must use version 2.x. In the process of porting the codebase from .NET Framework to .NET Core, one of the libraries used by the Helper Service that was responsible for the web service calls was not supported in .NET Core which is a rather common problem in these types of migrations. This one in particular was solved by adding this responsibility to the Helper Service but it is important to expect this kind of issues.

The rest of the migration was very straightforward and there is now a working version of the Helper Service running as an Azure Function.

EVALUATION

In order to quantify how the migration of the components above impacted its performance, several tests were performed. The objective of these tests was to guarantee that the scalability on-demand offered by these Cloud services was being used to increase the performance of such components.

The evaluation processes focused on the Document Converter Service since it is the component that has more room to benefit from such Cloud characteristics. As presented above, the Document Converter Service is composed by 2 different sub-modules: the DCSLogic and the DCSConverter. Given the different perks of each sub-module the scalability strategy has to take them into account. Two types of scalability were used in this evaluation process: vertical and horizontal scalability. In horizontal scalability, the number of instances of the given component is increased and decreased based on the load introduced in it, In vertical scalability, the computing power i.e. the hardware used to host such component is switched to a more powerful one to assess if it impacts its performance.

Concerning the DCSLogic, horizontal scalability would imply the need to replicate the image cache through every active instance. In order to avoid this problem, scalability was not the main concern for this sub-module. The main objective here was to verify that the introduction of an image cache had translated in increased performance when receiving requests to a previously converted image. This test was run with a single instance of the DCSLogic and 10 active instances of the DCSConverter. 20 requests were concurrently made to convert a single page of a docx document with 1MB. In the following test, the cache was disabled so that a baseline could be drawn of what the system performs like in the absence of a cache.

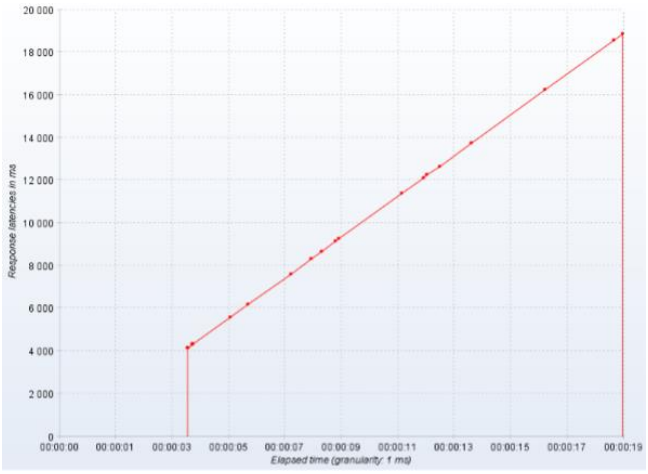


Figure 3 - 20 Requests with Disabled Cache

The figure above shows the results obtained. Despite having 10 different instances able to convert, the time spent in the DCSConverter is still very high compared to the total time of a request. The 20 requests took 19 seconds to be processed. In the following figure the exact same conditions were kept but with the cache enabled. The results were as follows:

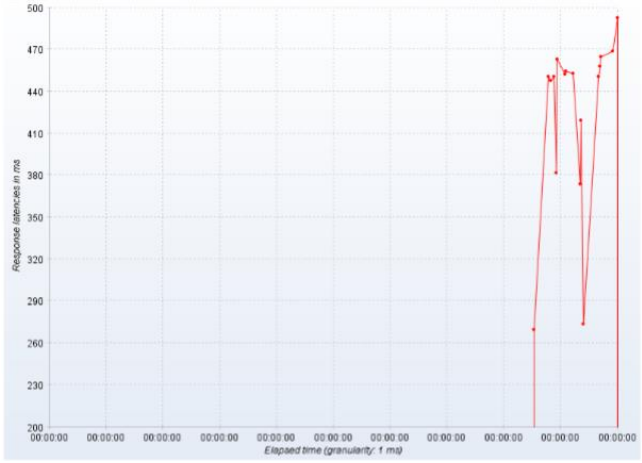


Figure 4 - 20 Requests with Enabled Cache

For the exact same test but with the cache enabled, the 20 requests were processed in less than a second. The introduction of the image cache resulted in a 95% faster response to each request.

Having concluded that, the focus turns now to the study of scalability in the heaviest part of the conversion process that happens in the DCSConverter. First concerning the horizontal scalability, the objective is to conclude if increasing the number of instances of the DCSConverter able to convert images has a toll on performance. To do this, 20 requests were concurrently made directly to the DCSConverter to convert a single page of a docx document

while only 1 instance was active. The results are expressed in the following figure:

Figure 5 - 20 Requests with 1 Instance

The total of 20 requests took 1 minute and 17 seconds to be processed. As there's only 1 active instance, the requests are processed consecutively, making the whole process way slower. The first 10 requests took around 40 seconds to be processed. This will be valuable when comparing with the next test performed. This next test replicates the exact same condition as the previous one, however, there are now 10 different active DCSConverter instances ready to process a request. The results are as follows:

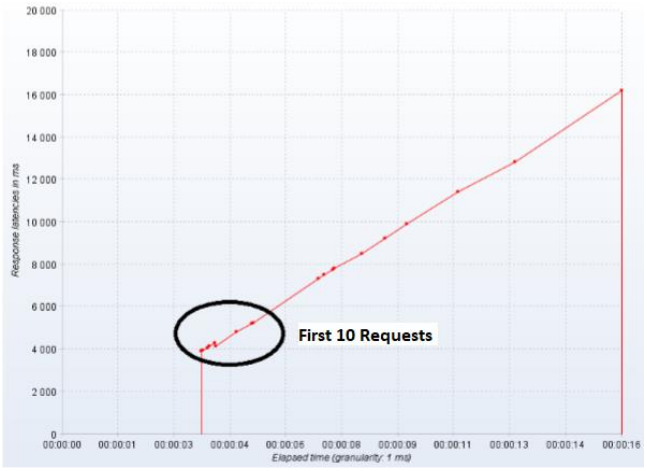


Figure 6 - 20 Requests with 10 Instances

In this case the differences are quite evident. The total of 20 requests were processed in a total of 16 seconds, in opposition to the 1 minute and 17 seconds that the same 20 requests took in the previous test. As a request is received by the DCSConverter, the load balancer existing in the Azure App Service where the DCSConverter is deployed is responsible for semding it to a free instance of the DCSConverter. This enables the processing of at most 10 different requests simultaneously. In this case, the first 10 requests took a total of 5.2 seconds to be processed with an average of 4.3 seconds. The similarities in these values show that the requests were processed concurrently by each of the 10 active instances. Moving from 1 to 10 active instances results in a performance increase of around 80% which is extremely significant.

Although the previous tests concluded that increasing the number of active instances impacts the performance significantly, having a fixed number of instances active at a given time does not go in line with the on-demand capabilities of the cloud. The objective is that the number of instances can automatically adjust to the load introduced in the application. To do this, the auto-scale capabilities of an Azure App Service were used. The auto-scale provided by Azure allows the number of active instances to scale up and down based on a certain metric. The metric is related to the usage of the hardware used to deploy it, like CPU usage or memory usage. This metric is calculated based on the results of the last 5 minutes. If such metric reaches the configured threshold, the auto-scale manager will increase the number of instances by a certain number previously configured. If the value of the configured metric drops below the threshold the number of instances will also be scaled down accordingly.

At first, the tests that were run using this auto-scale followed the same pattern as the previous ones: 20 concurrent requests to convert a single page of a 1MB docx document. However, the total execution time of these tests is not enough for the auto-scale manager to calculate that the metric threshold was reached, since it needs data from at least the last 5 minutes. The results obtained in these tests were very similar to the ones obtained in the test where there was only 1 active instance, meaning that the auto-scale was never being triggered.

To better compare the current version of the Document Converter Service with the one deployed on the Azure Cloud, making sure that the auto-scale capabilities were being used, it was necessary to increase the total number of requests made, so that the total time of the test enabled the triggering of the auto-scale configuration.

For the next tests, instead of a fixed number of 20 concurrent requests, a concurrent thread group was used. This thread group was configured so that at any given moment there are always 20 concurrent requests being processed. When one of the requests receives a response, another one is fired so that there is a constant rate of 20 requests at the same time for 30 minutes. The requests were of the same kind of the previous ones, a single page of a 1MB docx document. However, instead of targeting the DCSConverter itself, it follows the same path that a normal conversion request would. This test starts with the Applicational Server building the request to the DCSLogic, which has its cache disabled. Then, the DCSLogic routes the request to the DCSConverter that processes it using one of the available instances.

The auto-scale configuration used was based on the CPU usage being above 60%. This was due to the conversion process being very CPU intensive which means that when a certain instance is handling too many requests, its CPU usage spikes.

When this threshold is reached the number of instances is increased by 3, until a total of 10 instances. This means that at any given moment, the DCSConverter can have 1, 4, 7 or 10 active instances.

The following figure shows the results obtained for the first test in this scenario:

Figure 7 - Cloud Version Document Converter Service without Vertical Scaling

The evolution of the processing time of the requests shows that the auto-scale threshold was reached 3 times. There is the first leap from 1 to 4 instances at around the first minute, the leap from 4 to 7 at around the seventh minute and the final leap to 10 instances at around the thirteenth minute. As expected, the increasing number of DCSConverter instances able to process a conversion request results in a lower and lower average processing time. These results show that the use of the auto-scale capabilities provided by the Azure App Service do translate into a better performance.

In the second scenario for this test, the objective was to conclude if horizontal and vertical scaling capabilities could be combined in order to increase the performance even more. In this case, the exact same test configuration was kept with the difference that the vertical scaling category in which the DCSConverter was deployed was changed to one with double the processing power. Due to the conversion process being so CPU-heavy, the question is if doubling the CPU power translates into lower average conversion times.

The results are presented in the following figure:

Figure 8 - Cloud Version Document Converter Service with Vertical Scaling

As in the previous test scenario, the auto-scale threshold was once again reached at around the first minute, the eighth minute and the fifteenth minute. There is a clear distinction between the average conversion time with the different number of instances available. However, the average conversion time is more than cut in half when compared to the previous test. This is due to the fact that a higher processing power given to the App Service running the DCSConverter, makes the conversion process much faster which proves that besides scaling horizontally, the Document Converter Service performance is also able to scale vertically.

After observing the horizontal and vertical scaling capabilities offered by the new version of the Document Converter Service, there is the need to compare it with its current version. The exact same test was performed using the same 1MB DOCX document. A constant flow of 20 concurrent requests was kept and the performance results obtained from the current Document Converter Service are stated in the following figure.

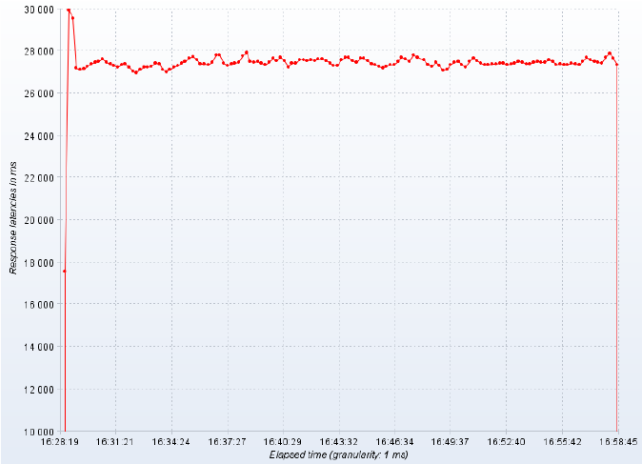


Figure 9 - Current Document Converter Service

As observed in the graph above, the current Document Converter Service is only able to reach an average value of 27 seconds in the processing time of the conversion requests. Its inability to scale according to the load introduced makes it extremely dependent on the VM capabilities in which it is deployed. The following table illustrates the comparison between the several tests performed in terms of number of requests processed and average processing time.

Table 1 - Document Converter Service Performance Comparison

	Total Requests Processed	Average Processing Time
Current Document Converter Service	1314	27432
Cloud Version without Vertical Scaling	3404	24000/18000/12000/7000
Cloud Version with Vertical Scaling	9007	11000/7000/4000/2900

The average processing times in the cloud version are separated between the several instance number available (1, 4, 7 and 10 instances). There is a clear difference in terms of the number of requests processed in the same 30 minutes, where the test without vertical scaling resulted in almost 3 times the number of requests and the test with vertical scaling resulted in almost 7 times the number of requests compared to the current version. In terms of the average processing time for 1 instance for the version without vertical scaling, although it is very similar to the current version, it progressively decreases while the number of instances increases.

PRICE COMPARISON

Despite the performance improvements, the migration of the Document Converter Service to the Cloud brought some overall cost increase. In the following table, the cost to execute the same 30 minutes load test in the current version is compared to the Cloud version:

Table 2 - Price Comparison

Version	Total Cost in Euros	Cost per Request Handled in Euros
Current EDOCLINK	0.0814	6.19×10^{-5}
Cloud Version without Vertical Scaling	0.459	1.34×10^{-4}
Cloud Version with Vertical Scaling	0.791	8.78×10^{-5}

Despite having a higher total price, the price per request of the Cloud version with vertical scaling is extremely similar to the current version. The number of requests handled in the cloud version was almost 7 times bigger than its current version for the same amount of time.

CONCLUSION

The existence of legacy application may become a problem for organizations which need to scale their needs. Migrating to the cloud provides on-demand scalability with flexibility to choose the amount of changes that need to be made to the application. The process of migrating some components of EDOCLINK to the Cloud brought performance improvements especially when under load. By deploying the Document Converter Service in the Cloud, its was able to scale vertically and horizontally. Despite the major improvements in terms of performance. It comes at the cost of some price increase when compared to the current version. However, the price analysis is of extreme importance to measure if the performance increases that it has brought are enough to justify the price increase. It is the responsibility of the organizations enrolling in the process of migrating their legacy applications to analyze the cost-performance of its Cloud version and decide if its is the path to follow.

REFERENCES

1. A View of Cloud Computing By Michael Armbrust, Armando Fox, Rean Gri_th, Anthony D. Joseph, Randy Katz,Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, Matei Zaharia Communications of theACM, April 2010, Vol. 53 No. 4, Pages 50-58 https://doi.org/10.1145/1721654.1721672
2. Jamshidi, Pooyan & Ahmad, Aakash & Pahl, Claus. (2014). Cloud Migration Research: A Systematic Review. IEEE Transactions on Cloud Computing. 1. 142 - 157. https://doi.org/10.1109/TCC.2013.10.
3. Zhang, Qi & Cheng, Lu & Boutaba, R. (2010). Cloud Computing: State-of-the-art and Research Challenges. Journal of Internet Services and Applications. 1. 7-18. https://doi.org/10.1007/s13174-010-0007-6.
4. Simon Crosby, Xensource and David Brown, Sun Microsystems, 2006/2007 https://doi.org/1542-7730/06/1200
5. Michael Eder, Hypervisor- vs. Container-based Virtualization, Seminars FI / IITM WS 15/16, Network Architectures and Services, July 2016 https://doi.org/10.2313/NET-2016-07-1 01