AUTOMATION OF ORDERING AND PROVISIONING OF RESOURCES IN A WEB-HOSTING SERVICE ON CLOUD INFRASTRUCTURE

Emil Nylind
Department of Computing Science
Umeå University

This thesis is submitted for the degree of
Civilingenjörsexamen i teknisk datavetenskap
Master of Science in Computing Science and Engineering

May 2017
ACKNOWLEDGEMENTS

I would like to acknowledge Ahmed Ali-Eldin, my thesis supervisor, without whose guidance and advice this work could not have been completed.

I would also like to express my sincerest gratitude towards the department of ICT Services and System Development for welcoming me to perform my thesis there after supporting me in my studies and development for the last two years.
ABSTRACT

Despite the increasing sentiment in industry that automation is a powerful way to reduce cost in the paradigm of cloud computing, there is no established golden-path to approach it. We investigated an approach to performing automation in the constrained scenario of web hosting, with the purpose of producing a software artefact capable of automating the domain.

We establish specific criteria which we care about automating and then produce a software artefact which satisfies the criteria. We establish that our model and example of how this type of performing automation can we done, and that it works.

Further, we introduce an autonomous management strategy in the form of an auto-scaler built for Microsoft Azure Web Apps, for which our assumptions for all concerning real-world applications hold. We prove that the method works by building it, and performing an experiment verifying this fact.

We conclude on arguing that the automation is feasible, but in many cases can be a larger cost than expected. We argue also that the scaling often proposed in literature is technically unfeasible in real-world scenarios, and that our scaler, while simple, works for these scenarios.
TABLE OF CONTENTS

1 INTRODUCTION ..............................................................................................................1
  1.1 CLOUD COMPUTING ........................................................................................................1
    1.1.1 What is cloud computing? ..............................................................................................1
  1.2 MANAGING CLOUD INFRASTRUCTURE ........................................................................2
    1.2.1 Managing IaaS infrastructures .................................................................................2
    1.2.2 Managing PaaS infrastructures .................................................................................2
    1.2.3 Approaches ..................................................................................................................3

2 THE PROBLEM .................................................................................................................5
  2.1 WHAT IS THE PROBLEM? ...............................................................................................5
    2.1.1 Why is this an interesting problem? ............................................................................5
    2.1.2 Why is it considered unsolved? ..................................................................................6
  2.2 PROBLEM BREAKDOWN ...............................................................................................6
    2.2.1 Ordering .......................................................................................................................6
    2.2.2 Provisioning .................................................................................................................6
    2.2.3 Managing ....................................................................................................................6
  2.3 FORMALIZATION ..........................................................................................................7
    2.3.1 Technical challenges ....................................................................................................7
    2.3.2 Scientific challenges ....................................................................................................7

3 METHOD ............................................................................................................................9
  3.1 INTRODUCTION .............................................................................................................9
  3.2 SOFTWARE AGENT AUTOMATION .............................................................................9
  3.3 AUTOMATION SCRIPTS ...............................................................................................10
  3.4 OPTIMIZING THE COST IN THE VIRTUAL DATA CENTRE ...........................................10

4 AZURE OVERVIEW & AUTOMATION ..........................................................................11
  4.1 OVERVIEW OF THE AZURE CLOUD ..........................................................................11
    4.1.1 Resource groups ..........................................................................................................11
    4.1.2 Compute services ........................................................................................................11
    4.1.3 Storage services ........................................................................................................12
    4.1.4 Data management services ......................................................................................12
  4.2 AZURE AUTOMATION API .........................................................................................13

5 AUTOMATION ARTEFACT ............................................................................................15
  5.1 TECHNICAL CHALLENGES .........................................................................................15
    5.1.1 Ordering ......................................................................................................................15
    5.1.2 Provisioning ...............................................................................................................16
    5.1.3 Managing .....................................................................................................................17
  5.2 SCIENTIFIC CHALLENGES ..........................................................................................17
  5.3 TECHNICAL CONTRIBUTIONS ....................................................................................18
  5.4 SCIENTIFIC CONTRIBUTIONS ....................................................................................18

6 SCALING EXPERIMENT ....................................................................................................19
LIST OF FIGURES

FIGURE 1 IaaS vs PaaS vs SaaS as illustrated by Microsoft. [8] ................................................................. 2

FIGURE 2 Visualization of how resource groups, service plans, and web apps are associated. 12

FIGURE 3 Logical structure of Azure SQL resources .......................................................... 13

FIGURE 4 The three main components of the automation artefact ............................................ 15

FIGURE 5 Overview showing the flow of data in the artefact .................................................. 24
LIST OF TABLES

TABLE 1 RESULT OF SCALE OUT EXPERIMENT .............................................................................. 20
TABLE 2 RESULT OF SCALE IN EXPERIMENT ............................................................................. 21
NOMENCLATURE

Words

git A popular source code version-control software.

Acronyms / Abbreviations

IaaS “Infrastructure as a Service” – a type of cloud service level, dealing directly with infrastructure components such as virtual machines, virtual networks, storage disks, etc. [1]

LAMP “Linux, Apache, MySQL, and PHP” – a common software bundle for building websites/web applications. [2]

MVC “Model, View, Controller”, a design pattern for separating data from presentation. It does this by introducing a component between that data and the presentation called the controller. [3]

OS Operating system, the software that manages computer hardware and software resources.

PaaS “Platform as a Service” – a type of cloud service level, providing platform layer resources. The cloud service itself deals with scaling and the environment in which to run the customers application. [1]

SaaS “Software as a Service” – a type of cloud service level, referring to on demand applications provided over the internet. [1] Examples such as Google’s Gmail and Slack.

SDK “Software Development Kit”, a software bundle to assist with developing software for some service or platform.

VM Virtual machine, a simulated computer system commonly running software or running computations.

WAMP “Windows, Apache, MySQL, and PHP” – variation on the LAMP bundle, using Windows as the operating system instead of Linux. [4]
1 INTRODUCTION

In this chapter we introduce the base concepts of cloud computing. We also introduce two approaches to manage infrastructure on cloud computing resources.

1.1 Cloud computing

Similar to the concept of electricity where one need not care about the means of delivery and only pay for the consumption, cloud computing emerged as a means of using computing resources as you would a utility. As with the electricity and water delivered to one’s home, the computing resources are hidden in a similar manner – only requiring knowledge of how to use the desired application.

The term cloud computing refers to a collection of on-demand computing service types, viewing computing infrastructure as a “cloud” from which you can draw computing resources from anywhere, on demand [5].

1.1.1 What is cloud computing?

On this subject, there has been a lot of debate and it took a fair amount of time to arrive at an agreed-upon definition. An easily-digested summary of cloud computing was eloquently stated by Vaquero et al. They stated, “Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs” [6]. The National Institute of Standards and Technology in the United States hold the most commonly used definition, however. They list five properties which are essential for cloud computing. They are as follows: On-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service. [7]

In addition to computing resources and storage, cloud providers tend to offer software services to assist in developing, deploying, and managing software on their platform. The end goal of this is to allow customers to run their everyday IT infrastructure in their datacentres, for which they can be charged based on usage.

1.2 Managing cloud infrastructure

Cloud infrastructure allows the customer to off-load a lot of hardware and software management tasks on the servers of the cloud provider. How much of the management is provided by the cloud provider can be split into different groups. They each have their own benefits and drawbacks. The three most prevalent Cloud models are Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). [1]
In this IaaS model, a service provider provides virtualized components such as VMs and networks that can form the building blocks of a large-scale computing infrastructure. As a general rule of thumb, IaaS allows the customer to manage the applications, data, runtime, middleware, and operating system of their resources. PaaS refers to the cloud service in which the cloud provider handles the infrastructure for the customer. The customer only provides the applications and data. It often provides a Software Development Kit for the customer to use. SaaS refers to providing on-demand applications over the internet. For our purposes only IaaS and PaaS are of interest.

![IaaS vs PaaS vs SaaS](image)

**Figure 1 IaaS vs PaaS vs SaaS as illustrated by Microsoft. [8]**

1.2.1 Managing IaaS infrastructures

When managing IaaS infrastructure, the end user is placed in charge of managing almost everything themselves. This means they generally will need to build their own strategies for handling machine images, software updates, virtual networking. Systems management and monitoring solutions need to be deployed. Provisioning of new resources and decommissioning of old resources is up to the end user.

However, using IaaS services tend to give the end user much more flexibility, and migration to other platforms – including self-hosting – is still possible with some forethought and care when selecting the cloud features to use.

1.2.2 Managing PaaS infrastructures

When managing PaaS infrastructure, the end user needs to perform less management. The platform will supply the handling of operating system, software updates, and monitoring. The platform itself supplies the systems management and monitoring. The end user instead deals with only the applications which they run. Provisioning is up to the platform, and the user need only track which applications are running.

However, this also locks the user into the platform which they choose. The user is expected to follow certain guidelines and have reduced choice in the tools which are available to them.
1.2.3 Approaches

There is some variety in how one can approach the management of cloud resources. Each have their strengths and weaknesses. Each approach suits different needs.

**Manual approach:** The most straight-forward way to approach the management of cloud resources. Using the graphical user interface provided by the cloud vendor. As it stands all current cloud vendors publish a web-based interface to order resources. Using this interface one creates, manages, and removes the resources following ones needs. This is generally easy.

On the other hand, it requires human intervention at every step. Using human resources to administer this is also expensive, given that a professional with specific knowledge is required to perform this administration. It also does not scale easily or efficiently, as the amount of computational resources to administer increases, so also do the amount of human effort.

**Automation:** Instead of relying on the user interface as one would when working manually, the user attempts to find repetitive interactions with the cloud vendor. Once such an interaction is identified, the user can employ the management API provided by the cloud vendor to write a script or piece of software which performs the interaction automatically. By systematically doing this, the user will end up with scripts for all of their most common interactions with the vendor.

This allows the end user to create software processes for taking infrastructure management decisions automatically. Potentially very time saving, if a script or software artefact can perform some action it is likely faster than their human counterpart. Along with that, a well-written automation script is significantly less prone to human error. The interaction can then also potentially be performed by a less technically skilled person.

There is however a significant drawback. If a process is hard to automate, or the process changes often, it can be surprisingly expensive. Engineering time is expensive.
2 THE PROBLEM

In this chapter we introduce the problem, and then further detail what parts of the problem we will consider more carefully.

2.1 What is the problem?

The problem is to automate the creation and management of virtual environments – using a software approach – in a cloud providing web-hosting for traditional web applications or websites. The archetypical example of this is a so-called LAMP/WAMP-stack. This type of stack has begun to fall out of favour in recent times, however. Variations have been established, replacing for instance MySQL with another database management system. Some also replace the language PHP with more modern alternatives, such as Python or Ruby. Instead, many now favour cloud platform services, such as Microsoft’s Azure Web Apps.

The core issue is to identify a strategy which allows for a high degree of automation of the cloud work-task of ordering and provisioning such a bundle. That is ordering and creation, however we spoke of management as well. By management we intend to investigate the ability to optimize the run-time cost of the applications running on the cloud infrastructure operated by the software artefact. The hope it to find some method that can be employed to reduce the cost of running applications, while still catering primarily to real-world constraints.

The goal with this work is to construct an implementation for Microsoft Azure. Ideally, this work would be implemented for multiple or at least two cloud providers, but the resources available causes this to be unfeasible.

2.1.1 Why is this an interesting problem?

As a general rule in life, it is always desirable to achieve the same result or work while lowering resource usage. By solving this problem, even partially, valuable human resource spending is reduced. These resources can then instead be utilized elsewhere, generating other value. If we also solve the optimization problem outlined in 2.3 Formalization, Managing, then we also reduce the computational resources used. This is also generated value.

From the perspective of department of ICT Services and System Development (ITS), the company sponsoring this research, they wish to reduce the amount of human resources they require in order to operate the web-hosting services. Since they are not interested in generating a profit – they are not a for-profit organization – from the hosting, and their clients are other government funded agencies, lowering cost is a
win-win-win. ITS can focus on other tasks yielding more value, the government agencies can pay less since the administrative overhead is lowered, and the government wins by spending less overall.

2.1.2 Why is it considered unsolved?

As previously stated, this is currently an unsolved problem at the department of ITS. However, we believe that this can also form a useful case-study of how automation and management of cloud resources can be performed more generally. Further, the methods of cloud scaling currently circulating in research literature is often based on assumptions which do not hold for many real-world applications, which must deal with economic constraints and feasibility.

2.2 Problem breakdown

2.2.1 Ordering

In this work we aim to construct a software artefact which publishes a graphical user interface for placing order for cloud resources. These orders should link who placed the order with an identity, so that costs can be tracked and billed.

We envision this to be a web portal in which ITS’s customers can themselves log in and place an order. These orders can, for instance, be an SQL Database or a Web App. We should investigate if there is some resource for which ordering can be inferred, rather than explicit.

The interface should also present an ordering mechanism for decommissioning previously ordered resources.

2.2.2 Provisioning

This is the process of taking an order as established in Ordering and establishing the requested resource in the cloud. In essence, we want the server to interpret the placed order and through automation construct and order these resources in the cloud. It should then publish the information and credentials needed to access and use the resources to the web interface.

The process should then, of course, also ensure that the firewalls of the resources are configured to be accessible to the end-user. For instance resources should be be configured to be able to access local databases established at Umeå university.

Furthermore, some of the resources could potentially be optimized at the time of provisioning to be placed in the optimal location to reduce costs through, e.g. co-locating databases within an SQL server.

As with Ordering, it is our aim that this can automatically take decommissioning orders and remove the requested resources from the cloud.

2.2.3 Managing

We want to establish a method for tracking the usage of the resources for each customer so that they can be billed for their consumption.
We are also interested in how the system can automatically monitor all the currently provisioned resources, and if it can attempt automatic cost reduction in their resource allocation. We aim to construct a real-world type auto-scaler for the web apps, since most current research into the subject is based on assumptions which do not hold in our scenario.

The auto-scaler should follow a rules-based approach, as is the case is many modern scaling techniques [9] [10].

2.3 Formalization

In order to evaluate if the problem has been solved. We break each issue down into the following set of questions. These questions are what we set out to answer in developing the software artefact.

2.3.1 Technical challenges

The technical challenges which will be addressed are the logistics of writing a software artefact and its architecture capable of performing the desired automation. The automation requires the software to rely heavily on asynchronous communication. The asynchronous nature is a significant challenge in ensuring a consistent view of the system. The asynchronous communication is not reliable and must sometimes deal with partial success, failure, and retries.

2.3.2 Scientific challenges

<table>
<thead>
<tr>
<th>Ordering</th>
<th>How can one automate the ordering of web hosting? What requirements are there on the ordering process to allow automation of subsequent processes?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provisioning</td>
<td>How can we manage SQL databases, elastic database pools, web applications, and other resources during provisioning? What resources can be safely hidden from consumers?</td>
</tr>
<tr>
<td>Managing</td>
<td>How can we get this system to track how the resources across the cloud is allocated? Can we find some optimization strategy(s) which can be employed to use resources more efficiently? To what degree is the resulting system autonomous?</td>
</tr>
</tbody>
</table>

All applications which are considered are assumed to have monitor capabilities built-in to allow use of the state and load of the application. This hold true for all web applications using Azures provided technologies.
3 Method

In this chapter we introduce the approach we took in solving our problem.

3.1 Introduction

In order to answer the questions posed in the The Problem - Formalization section, we studied literature to understand the methods and strategies available to achieve automation in cloud infrastructure. We also constructed a software artefact which implements the desired automation outlined. From this, we studied the issues that arose when autonomous management of resources in the cloud is employed, and the patterns and difficulties in constructing such a solution.

This is called a build methodology [11]. The software artefact provides the automation desired by the end-user, and the development was intended to yield answers to the research questions and provide material for analysis in combination with existing literature.

3.2 Software agent automation

For our build methodology, we constructed a number of goals which we desired the system to automation. Those goals were as follows:

- Allow for ordering of the following resources with no human intervention
  - Microsoft Web App Service
  - SQL Database
  - SQL Server
  - SQL Elastic Database Pool
  - Cloud storage
- Allow for the publishing of relevant information concerning the resources, such as connection information and resource credentials.
- Allow of tracking of individual end-users resources usage automatically.
- Allow for the auto-scaling of web apps to allow a reduction in consumed resources during idle times.
- Allow the end-user to terminate and decommission their resources.
- Once we had established these goals, we implemented these features using common software tools and practices.
3.3 Automation scripts

In order to realize the goals, we investigated how the Azure Automation API operated, and spent time getting familiar with the process. This was done both through studying of documentation released by Microsoft, as well as information found on various technical sources. We also investigated what resources Azure offered, how they are related to each other, and how they meter use.

We spent time manually constructing the types of resources that were to be automated to gain an understanding of what the process of performing it with automation would entail. Once we had a good grasp of how the process functioned manually, a script was developed for each action for which automation was desired. We built such automation scripts for: Storage, SQL servers, SQL databases, SQL elastic pools, and web apps.

The scope of the automation was limited to the aforementioned resources, since we consider supporting all types of applications unfeasible given available resources.

3.4 Optimizing the cost in the virtual datacentre

Once we had an artefact capable of the level of automation of ordering and provisioning as desired, we investigated how to lower the costs of these resources. We did this by introducing an auto-scaler for the web app components, which the end-user could opt-in on enabling.

How to auto-scale a PaaS offering is not always straight-forward since many have some scaling properties already built in. We developed an approach which allowed us to scale the application, and to do so without causing service interruption.

We tested that our assumptions hold by showing that it works when implemented through the experiment in chapter 6. We did not engage the question of which scaling rules are more effective, and how the metrics used and their optimal threshold values affect the performance.
4 Azure Overview & Automation

In this chapter we briefly cover how the Azure cloud platform is structured, and introduce the components which are relevant to the work.

4.1 Overview of the Azure cloud

The Azure cloud platform is an offering by the software vendor Microsoft. It is a platform that allows the end-user to build, deploy, and manage applications in Microsoft-managed data centres. It offers both an Infrastructure as a Service platform, a Platform as a Service platform, and Software as a Platform. It started out primarily supporting Windows-based applications, but have now expanded to include Linux support as well.

Azure offers a range of services, which can be condensed into the categories Compute, Mobile services, Storage services, Data management, Messaging, Media services, CDN, Management, and Machine Learning.

Of these services, we will only have need to investigate small portions of their offering, falling in the categories of compute, storage services, and data management.

4.1.1 Resource groups

In Azure, you can create logical groups of related resources. This allows you to refer to them as a single entity, and keep track of the expenditures related to the resources in the group. A resource can only exist in a single resource group, making it a top-level logical unit. There are no limits to the amount of resources that can reside in a resource group.

4.1.2 Compute services

In their compute services, they offer virtual machines and app services along with other computation services. Of these, we will utilize the App service. The app service is Microsoft Azure’s platform as a service, offering a fully managed solution for web and mobile applications. They offer development support for .NET, Java, PHP, Node.js, and python.

A web app is executed on an App Service Plan. An app service plan is a pool of computing resources, which can be allotted to a single web app, or multiple. An illustration is shown in Figure 2.
App Service Plan: The app service plan is, as mentioned, a pool of computing resources. The service plan defines how large a pool of resources is, and what type of machines instances are utilized. A service plan can vary from the most basic plan, involving 1 instance with 1 core and 1.75 GB RAM, to their premium tier plans which allows up to 20 instances with 4 cores and 7 GB RAM, subject to availability.

4.1.3 Storage services

The storage services provide access to storing and accessing data on the cloud. It does this through REST and SDK APIs. To specify charges for these services, you establish a Storage Account, from which data storage costs are charged. The storage account provides a namespace for your storage resources, which then powers further storage services, such as the table storage service, blob service, queue service, and file service. Below is a short summary of the table service and blob service.

Table Service: The table service allows for storing structured text accessible by partition key and a primary key. It is a store for NoSQL-data in the cloud. It is constructed for fast data access.

Blob Service: Allows for a storage account to store unstructured data as blobs. This is allows storing binary data and media. Provides security mechanism to control access to data. [15]

4.1.4 Data management services

Under the data management services there are a large selection of database and data warehouse services. Of primary concern is the SQL Server, SQL Elastic Database Pool, and SQL Database resources.

SQL Server: This resource, despite its name, is not an actual SQL Server. Instead, it a logical service which acts as a central administrative point for many databases [16].

Elastic Database Pool: An elastic SQL-pool, which provides a configurable service level over a pool of SQL databases. An elastic database pool allows the databases to share capacity to make the best use of the capacity at hand. It allows databases with different loads to attain different levels of capacity. [17]

SQL Database: A relational database service, based on the Microsoft SQL Server engine. Often manages its resources through an elastic pool, but it is not a requirement. [18]
4.2 Azure automation API

The primary tool for automating Microsoft Azure today is the Microsoft Azure PowerShell cmdlets [19]. They are a collection of commands for interacting with the Azure platform via PowerShell.

The API is structured around a plethora of commands for querying, ordering, and modifying Azure resources. It uses the general Get/Set/Select paradigms present in other PowerShell libraries. For instance, to retrieve information regarding a web app you can use `Get-AzureRmWebApp -ResourceGroupName $val1 -Name $appName`, and similarly you can modify the web app with `Set-AzureRmWebApp`.

The API is extensive and is actively developed, adding more functionality and features often. Once a user has authenticated themselves there is very little which they can not perform with some combination of Azure PowerShell cmdlets. The API is generally well documented, although some commands lack proper descriptions.
5 AUTOMATION ARTEFACT

In this chapter we introduce the technical and scientific challenges which our automation strategy encountered, and present our technical and scientific contributions.

5.1 Technical challenges

Our primary technical challenge was to build a software artefact which actually worked and performed the automation tasks that were desired. This required significant engineering effort, and presented many engineering challenges.

We split the system into three abstract components, a view component from which the user can view the current state of the system and place orders, a provisioning component which performs work against Azure to manage the cloud resources, and a batch processing component to update the system model of the cloud resources on Azure with reported data from the provisioning component. We also have a common database which all parts may access.

The components interact as shown in Figure 4.

Figure 4 The three main components of the automation artefact.

5.1.1 Ordering

There were few technical challenges in the process of ordering, as it is an established process. The process can simply be reduced to constructing a form which is submitted and stored on a server. While some initial human intervention is a requirement to setup a user account for the customer – letting anyone, anywhere, arbitrarily order resources without authentication is a terrible idea –, the only other challenge is ensuring
that the form contains enough information to allow automation of all subsequent processes by the provisioner. What information is needed varies dependent upon what resource is being provisioned.

5.1.2 Provisioning
The primary challenges posed by provisioning were regarding communication with Azure and ensuring that errors and failure states were captured, and keeping the internal model of the currently provisioned resources in sync with the actual state. It was also challenging to get the communication flow from Azure back into the artefact for processing.

5.1.2.1 Job failure and retries
Any communication or request made to Azure can fail, at any time for almost any reason. This places very strict requirements on the scripts which perform this communication. The scripts must be re-entrant. That is, if they are interrupted part way through the procedure, they must be able to be safely called again and complete the work. This requires extensive checks against which resources exist on Azure and what their statuses are. This causes the scripts to run slower, but the system overall becomes a lot more robust.

If a script fails or reports an error, it can then safely be retried without risk of causing errors. In our artefact, we retry these scripts up to 10 times using an exponential back-off strategy.

5.1.2.2 Communication with Azure
Communication with Azure is a non-trivial issue. Since the Azure automation API is written for PowerShell, the artefact needs to run separate processes to perform its communication. This requires that cross-process communication take place. The PowerShell sub-process needs to send raw, unprocessed information back into the main process.

A script with PowerShell commands range from a run time of seconds to 20 minutes. This entails that the main process cannot await a response. This is the reason that the architectural style shown in Figure 4 was chosen. By using a batch processing style we can easily deal with the communication as if it was entirely asynchronous from the view of the artefact.

The PowerShell scripts are however also not Boolean successes or failures. They can be successful, or partially successful, or fail entirely, even after being retried multiple times. This means that when we perform our batch processing we need to consider the possible errors that can occur. In the case of critical failures, we may even need to run clean-up scripts in order to re-establish the consistency between its own internal model of the world and the actual reality.

5.1.2.3 Background jobs
We have already stated that communication with Azure is required to run in a background job due to their time-consuming nature. This means we were highly encouraged to use a library specifically for this. There are many libraries which provide fully featured background job processing, with retries, exponential back-off strategies, and recurrent jobs already implemented and perfected. This largely solved our issue with running background processing in an environment – our web application – which is generally hostile towards it.
For instance we use such a library to perform the recurrent job of processing incoming script results.

5.1.3 Managing
The technical challenges in managing the resources emerges from the cost-lowering goal. We investigated a method of lowering costs by introducing a custom auto-scaler for the web apps we allow the users to provision.

Our contribution is a system which allows easy, bounded scaling of a web app within Azure. It works on the concept of increasing/decreasing the amount of the computing resources available in the App Service Plan. It does this using a simple rule based engine, which when active creates a frequently recurrent background job which performs the following logic:

- Reads the current load of the web app from the Azure API, polling for at least 1 minute
  - If the metrics fail to be retrieved, finish without scaling.
- If the value is greater than set threshold
  - If current plan-level is not at capacity for workers
    - Add another worker instance to the pool
  - If current plan-level is at capacity for workers
    - Upgrade the plan to the next level
    - Add another worker to the pool
- If the value is less than set threshold
  - If current worker-level is greater than 1
    - Remove a worker instance from the pool
    - If new worker level fits in a lower service-plan level
      - Downgrade the service-plan level.
  - If current worker-level is equal to 1
    - Do nothing

5.2 Scientific challenges
The scaling methods developed for scaling in the cloud is seldom based in the reality of industry, which often operates in an entirely different domain of assumptions than academia. Our challenge is to create an auto-scaler which works based on the reality in an industry setting. The scaling methods presented in literature often assumes that there are no hard limits on how much computing resources that are available, and views the resources as an infinite pool. Hard limits are needed. It is not technically feasible to allow scaling to arbitrary levels, and it eventually becomes a plumbing problem rather than a scaling problem. If for no other reason, when it comes to scaling, it is sometimes better to allow the service to become unresponsive than scale out to such levels that the costs skyrocket.
Our scaler is based on assumptions which we know work in the industry environments in which it will be run, and we have been inspired by the ideas expressed by Chieu et al. in their work on dynamic scaling of web applications [9]. The full logic of the scaler is expressed in section 5.1.3, and the logic is evaluated on an on-going basis.

The main goal was to construct a simple system that works in practice, which we succeeded with. On the topic of scaling, for the state of the art of auto-scaling, Botran et al. [20] have compared more than 40 state of the art auto-scaling algorithm in their paper. It is an enlightening read, but out of scope for this system.

5.3 Technical contributions
We have constructed the software artefact, which the department ITS had need. It provides a model and example of how this type of automation can be implemented, and further we show that our approach works.

The software artefact is under consideration for being open-sourced for use by any other institution or entity which has a similar need to ITS. If permission is granted, the source and assets needed to run the system will be published.

5.4 Scientific contributions
Currently, most scaling methods and algorithms circulating in research literature is built on assumptions which does not hold for most real-world applications. We have built a simple, realistic model for performing scaling of small to medium web apps running on Azure.

We proved that our assumptions regarding the scaler hold for the tested real-world applications, and works for any arbitrary application hosted upon the Azure Web App platform.
6 SCALING EXPERIMENT

In this chapter we perform an experiment to establish that the auto-scaling performs as expected. We provide a detailed description of the experiment, its results, and then evaluate the result.

6.1 Experiment description

6.1.1 Experiment setup

In order to assess that the auto-scaler performed as expected we devised an experiment. The experiment was based on putting an Azure Web App under heavy load while having our auto-scaler turned on and inspecting the behaviour of the system. For this purpose, we wrote a small application which taxes the CPU of a worker instance to 100% of capacity. We could use any metric which can be described as a percentage on the interval $[0-1]$, but we only test CPU utilization in this experiment.

For the experiment we setup a web application with worker instances with one core. We set up the web app to scale to a maximum of five worker instances. This enables the artefact to perform scaling from the Basic Service Plan of one to three workers, up to the Standard Service Plan. The standard service plan has a maximum worker capacity of 10.

The web application was set to auto-scale through our software artefact, and the time between evaluating the auto-scale logic was set to five minutes. The metric polling time was set to one minute, polling for CPU utilization as a percentage. The web application was not exposed to any external traffic during the experiment.

The results and behaviour is noted manually by inspecting the output of the Azure Portal. Progress of the auto-scaler and the timing of scaling is noted by manually inspecting the web-interface provided by the background-job management library of the software artefact.

6.1.2 Scaling out

In order to test the ability of the auto-scaler to scale out, that is add more resources to the worker pool, the following process was performed.

The resource consumption script is set to run on the web app, and we make note of CPU utilization. We await that the auto-scaling logic is triggered, and then inspect its behaviour. We repeat this process for 6 iterations which should allow it to scale out fully and then remain there for two iterations. We note for each iteration whether it

(a) Scales out, adding 1 worker instance.
(b) Scales in, removing 1 worker instance.
6.1.3 Scaling in
In order to test the ability of the auto-scaler to scale in, that is remove resources from the worker pool, the following process was performed. This part continues on the state of the experiment described in 6.1.2.

The resource consumption script disabled on the web app, and we make note of CPU utilization. We await that the auto-scaling logic is triggered, and then inspect its behaviour. We repeat this process for 6 iterations which should allow it to scale in fully and then remain there for two iterations. We note for each iteration whether it

(a) Scales out, adding 1 worker instance.
(b) Scales in, removing 1 worker instance.
(c) Takes no action.

6.2 Result

6.2.1 Scale out
For each iteration, we recorded the CPU utilization prior to the auto-scaling logic executed. This CPU utilization is presented in the column CPU Utilization. We then recorded the action the auto-scaler took and present this data in the column Worker Instances. The CPU utilization of the first sample is recorded just after the resource consumption script has started.

<table>
<thead>
<tr>
<th>Sample #</th>
<th>CPU Utilization</th>
<th>Worker Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>99-100%</td>
<td>1, increased to 2.</td>
</tr>
<tr>
<td>2</td>
<td>99-100%</td>
<td>2, increased to 3.</td>
</tr>
<tr>
<td>3</td>
<td>98-100%</td>
<td>3, increased to 4.</td>
</tr>
<tr>
<td>4</td>
<td>98-100%</td>
<td>4, increased to 5.</td>
</tr>
<tr>
<td>5</td>
<td>99-100%</td>
<td>5, no action.</td>
</tr>
<tr>
<td>6</td>
<td>97-100%</td>
<td>5, no action.</td>
</tr>
</tbody>
</table>

Table 1 Result of scale out experiment

The experiment runtime was 22 minutes for samples #1 to #4. The total runtime for samples #1 to #6 was 32 minutes.
6.2.2 Scale in
For each iteration, we recorded the CPU utilization prior to the auto-scaling logic executed. This CPU utilization is presented in the column **CPU Utilization**. We then recorded the action the auto-scaler took and present this data in the column **Worker Instances**. The CPU utilization of the first sample is recorded just after the resource consumption script has been terminated.

<table>
<thead>
<tr>
<th>Sample #</th>
<th>CPU Utilization</th>
<th>Worker Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5% to 25%</td>
<td>5, decreased to 4.</td>
</tr>
<tr>
<td>2</td>
<td>0% to 5%</td>
<td>4, decreased to 3.</td>
</tr>
<tr>
<td>3</td>
<td>0% to 5%</td>
<td>3, decreased to 2.</td>
</tr>
<tr>
<td>4</td>
<td>0% to 5%</td>
<td>2, decreased to 1.</td>
</tr>
<tr>
<td>5</td>
<td>0% to 5%</td>
<td>1, no action.</td>
</tr>
<tr>
<td>6</td>
<td>0% to 5%</td>
<td>1, no action.</td>
</tr>
</tbody>
</table>

**Table 2 Result of scale in experiment**

The experiment runtime was 22 minutes for samples #1 to #4. The total runtime for samples #1 to #6 was 32 minutes.

6.3 Evaluation
From the results presented in section 6.2.1 we see that when the auto-scaler is enabled, and is under heavy load, the system automatically increases worker instances linearly until it reaches the maximum number of instances allowed as would be expected by the algorithm. We further note that it remains at the maximum of the bounded range if exposed to continued heavy CPU load. Therefore, we conclude that the scale out feature works as expected. What may be desirable to adapt is the rate at which the scaling logic is executed to allow faster scaling, since traffic spikes can be faster than the five minute setting can react. This should however be as simple as modifying a variable and is expected within the bounds of this experiment.

When the web app has the load removed the auto-scaler, as presented in section 6.2.2, automatically begins to decrease worker instances linearly until it reaches the minimum number of instances allowed as is expected. We further note that when remaining under low load it remains at the minimum bound. We therefore conclude that the scale in feature also works as expected.

We have no reason to believe that we have performed the experiment inaccurately or unfairly, and we trust the result. While the formal experiment was not repeated multiple times, the system was tested thoroughly during construction and performed similarly under those conditions.
This chapter describes the design of the software artefact. It covers which technologies are employed, the architecture, as well as other miscellaneous details.

It explains the development process since it is considered of value for the answers to the end goal of the work. We provide some analysis on the results drawn from the software artefact.

7.1 Software artefact design

7.1.1 Technologies

The software artefact is written in the programming language C#, utilizing the .NET framework. These tools were primarily chosen to ensure that ITS can maintain the software artefact with their current staff, if they should want to. The author had little experience with the technologies, but was assured they were similar in almost all respects to tools which the author was familiar with. The development was performed in Microsoft Visual Studio.

The reasoning behind using these tools is primarily to benefit from the experience of the developers working at ITS. Learning the fundamentals of the .NET framework was decided to be a necessary time-sink to ensure the artefact could be maintained. The arguments was that, by instead focusing on using well-known technologies that the ITS supervisor was comfortable with – rather than other technologies – would yield better assistance.

It did however instead prove to be a large loss in terms of productivity. It caused the need to study many new ecosystems, namely C#, .NET, and Azure. This soaked up a lot more time than estimated, and caused large delays in the engineering of the software artefact.

The interface is built using C# and the ASP.NET MVC framework. It employs the Hangfire framework for tracking and running background jobs, such as executing PowerShell scripts to modify resources on Azure. Running background jobs inside the MVC framework is discouraged and considered bad practice, since it can crash the entire process and implementing it manually is very error-prone. Hangfire is a ready-made solution to the problem. Hangfire is a background job management library, specifically designed for this kind of task. It includes the ability to schedule jobs, create recurring jobs, and automatically retry failed jobs.
7.1.2 System overview

The software artefact is composed of a web application which interacts with the Azure cloud by running PowerShell scripts asynchronously. The PowerShell script then acts upon Azure using Microsoft Azure Cmdlets.

![Figure 5 Overview showing the flow of data in the artefact.](image)

In Figure 5 we see that the flow of data is as follows:

1. The user submits some request (order) to the ASP MVC interface.
2. The ASP MVC application takes this order and submits a background job based on the order.
3. The background job is eventually fired by the Hangfire library, running some piece of PowerShell to affect Azure. When the background job detects that the PowerShell is finished, it submits the raw output of the script to the database.
4. A recurrent job in Hangfire starts a batch processor, which takes all results submitted by background jobs and processes their results. This involves the following:
   a. Parsing the script output into a useful form.
   b. Detecting what the result of the script ultimately was, such as if it was completely successful, or failed, or had no effect.
   c. From this information update the database with the current state on Azure.

In order to get all of these pieces to communicate properly and deal with credentials -- both to the MVC application and to Azure -- took significant engineering efforts. The result, however, is a system which is surprisingly robust. During testing it has not failed in any unexpected manner.

7.2 Initial setup

As previously mentioned, the view available to the user is implemented in the form of an ASP.NET MVC web application. This application allows for registering an account and linking an Azure subscription to said account. The user must also select a unique identifier which the system can tag their resources on Azure with. Upon having both a subscription and an identifier, the application performs some base setup on Azure. It creates a storage account, and it also creates three resource groups:

- `<identifier>-data`
Once this setup procedure has been performed, the user can start ordering resources.

7.3 Ordering
The ordering component is fairly simple. The interface exposes the ability to order SQL databases, and web apps. When an order is placed, a controller performs some basic book-keeping, such as controlling that the account has succeeded with the initial setup, and then submits information to the provisioning component about what was ordered.

7.4 Provisioning
The provisioning component is arguably the most complex part of the system. The provisioning component takes as input some information about the resource that has been ordered.

Once it has decomposed that information -- and enriched it with information from the database -- into the form needed by the PowerShell script managing the type of resource ordered, it submits a background job to Hangfire to execute the script.

Each script is written to be extremely fault tolerant, and to recover from most common errors. Each script writes all relevant information in an easy to parse format back to the Hangfire job. When the job is successful, the background job submits its raw-text result to a database table containing data to be processed by the system for model updates.

The system updates its internal model of what is provisioned on Azure by performing batch-processing on the output of all scripts submitted to the aforementioned database table. It does this by first parsing all relevant information out of its raw-text format into a script result type, and then submitting this object to a processor.

The processor contains logic for how to update the internal model of the system given the output of a script.

7.5 Managing
For the autonomous management part, as previously stated, we implemented a rules based scaling method which can be turned on for the web apps. This was implemented as a PowerShell script, which when ran either scales out, scales in, or does nothing dependent on the current load of the system suggested by the Azure provided metrics. [21] This script is then, when scaling is toggled on, submitted to Hangfire as a recurrent job with an interval of five minutes. The interval can be set to any time interval. This has the effect that every five minutes the system evaluates whether it should scale out (increasing the available computing resources) or if it is safe to scaling in (decreasing the available computing resources). Of course, the system can also be in a state of equilibrium, for which no scaling changes are desired.
By using this strategy we gain confidence that turning scaling on and off is as easy as setting up / cancelling the recurrent job. It also provides the stability and crash-resilience that an enterprise-grade solution such as Hangfire provides by retrying and rescheduling the scaling job.

This way of handling scaling is, as previously mentioned, inspired by the work of Chieu et al. [9]
8 CONCLUSIONS AND FURTHER WORK

In this chapter we draw conclusions on the work performed and provide answers to our research questions. We propose areas which may merit further work, and a recommendation to ITS regarding the software artefact.

8.1 Conclusions

During the course of this work, we have implemented an artefact which fulfils the requirements outlined in the problem formulation. It supports ordering all the outlined desired resources, and can provision them automatically without any human intervention. The artefact took significant engineering effort to get working.

In fact, the artefact took such an amount of engineering efforts to get working, that maintaining this artefact could be a resource drain. If further resources are to be automated, we believe that these would take just as much work to automate as these did. Such work would however benefit from the base architectural engineering which has been performed.

We have built a rules-based auto-scaler which works, and hold in our version of real-world conditions. This scaler reduces the amount of provisioned resources per web app, since the applications no longer need to be over-provisioned. This reduces the amount of consumed resources, and provides value to the user.

We also argue that the scaling often proposed in literature can be technically unfeasible, since a scaler can not be allowed to scale out indefinitely, and that doing so would eventually become a plumbing problem. On this issue we echo the common software engineering adage: Keep it simple, stupid.

8.1.1 Conclusions on research questions

8.1.1.1 Ordering

How can one automate the ordering of web hosting? By the way of our software artefact we can conclude, unsurprisingly, that ordering of web hosting can be done easily in a web application which submits form data and stores it in a database.

What requirements are there on the ordering process to allow automation of subsequent processes? We can only draw conclusions dependent on our experiences, and these will vary depending on the cloud platform which is used, and other factors. We found, however, that we required some initial setup of a user
account in the web application. The application needed to get a user identifier with which it could tag resources, and an Azure subscription id which it could link to the user account.

8.1.1.2 Provisioning

How can we manage SQL databases, elastic database pools, web applications, and other resources during provisioning? What resources can be safely hidden from consumers? We find that these questions are somewhat interrelated. Based on how one manages the resources mentioned, one can hide some of the others. As for which resources could be hidden in our approach, we found that the SQL Server, Storage account and Elastic database pools could easily be managed behind the scenes without need for concerning the customer. They did not need to be exposed to the user.

8.1.1.3 Managing

How can we get this system to track how the resources across the cloud is allocated? We found it effective to use user identifiers in combination with the Azure tagging system, which allows tagging resources with arbitrary information. We combined this with using Azure’s resource groups to bundle the user’s resources together. In addition to this, we also maintained a local model of the resources provisioned in the cloud which we constructed by analysing the results returned by the Azure API.

Can we find some optimization strategy(s) which can be employed to use resources more efficiently? Yes, as an example we have constructed a rules-based scaler which ensures that the web apps uses its resources more efficiently.

To what degree is the resulting system autonomous? Almost entirely, the system requires no human intervention on the part of ITS once a user account for a customer has been set up. However, the customer part of the interaction is still entirely manual.

8.2 Further work

There is much that could be further worked on. For instance, improving the metrics and threshold values for the scaling system would most likely yield even better results.

It would also be interesting to perform some research into the actual yield – that is how much effort or resource-costs were reduced – of automating workflows and processes. This could however also prove a difficult topic to investigate, given the heterogeneity of processes and their differing yields. I do however suspect that this work which we have performed here will prove a net negative of effort and not tractable for salaried work.

Another interesting cost optimization which we would like to see investigated is how to reduce the cost of extremely low-traffic web apps. For instance, could one find a method to clump low traffic applications onto a single service plan, and upon detecting a higher than normal demand split the application out onto its own service plan in order to scale it?
8.2.1 Recommendation to ITS

We would recommend evaluating whether the hereto committed engineering effort would have been beneficial, if it had been paid labour. That is, for one engineer working full time for a period of four to five months, is the amount of potential work saved worth that salary and the maintenance of this artefact?

If the answer is no, then we recommend not committing further resources to this project. In the end, if creating the automation costs more than the saved manual labour then it is not worth automating.

Currently, the artefact still requires some engineering effort to ensure that it can perform on enterprise levels of use. It also requires some further engineering to extract some information which is currently not collected from Azure.
References


References


