Evaluation of IBM Bluemix (PaaS) auto-scaling service for microservices.

Patric Lind
Abstract

Evaluation of IBM Bluemix (PaaS) auto-scaling service for microservices

Patric Lind

The company Sisyfos Digital AB host some of their solution on the cloud, and have expressed an interest in scaling their applications. The ability to scale applications can be an important factor to achieve high availability, and therefore led to questions such as in what extent is this possible and efficient for different applications.

Using a service for scaling on this cloud infrastructure, testing with multiple applications and varying incoming requests, the goal was evaluate benchmarks and identify bottlenecks for incoming requests with its response time.

The results show that depending on the application, scaling can be a valuable tool under the right prerequisites. Some of these prerequisites need to be evaluated beforehand, and the successfulness of the scaling is dependent on it. Under the right conditions, an almost linear result can be achieved when scaling between one and three instances. However, the most significant aspect of scaling is the identification of bottlenecks in the application.
Evaluation of IBM Bluemix (PaaS) auto-scaling service for microservices.

Patric Lind

Contents

1 Introduction 4

2 Background 4

2.1 Technology and tools ................................................. 4
2.1.1 JSON .......................................................... 4
2.1.2 NoSQL Database ................................................ 4
2.1.3 REST API ..................................................... 5
2.2 Cloud computing ...................................................... 5
2.2.1 Cloud Service models ......................................... 5
2.2.2 Cloud scalability ............................................... 5
2.3 Service Oriented Architecture ..................................... 6
2.3.1 Microservices .................................................. 6
2.3.2 Spring .......................................................... 6
2.3.3 Maven .......................................................... 7
2.4 Bluemix .............................................................. 7
2.4.1 Services and runtimes ....................................... 7
2.5 Related Work ....................................................... 8

3 Requirements - Challenges 9

4 System Design 10

4.1 Storage .............................................................. 10
4.2 Data Computation ................................................. 10
4.3 Dispatch handler .................................................. 11

5 Implementation 11

5.1 Motivation for tools and methods ................................ 11
5.1.1 Infrastructure .................................................. 11
5.1.2 Communication ............................................... 11
5.1.3 Database ...................................................... 11
5.2 Execution .......................................................... 12
5.2.1 Storage Microservice ....................................... 12
5.2.2 Computation Microservice .................................. 12
5.2.3 Dispatcher Microservice .................................... 13
5.2.4 Load testing ................................................... 13
5.2.5 Binding Bluemix services ................................... 14

6 Method evaluation and results 14

6.1 Testing .............................................................. 16
6.1.1 Test results .................................................... 16
6.1.2 Auto-scaling testing and results ......................... 20
6.2 Discussion ........................................................ 24
6.2.1 Threats to Validity .......................................... 25

7 Conclusion and Future Work 25

7.1 Future work ....................................................... 25
1 Introduction

Today cloud hosting has become a valuable solution for companies, where they can host their server and applications on clouds [1]. Cloud computing is often payable on demand in correlation with its usage and performance. This makes it beneficial for the customers who can avoid paying for idling resources. Depending on the cloud service, the customer can save time hosting their servers on a cloud when the cloud automatically handles the infrastructure of the platform.

From different companies different variations of cloud services are offered. The company for this thesis, Sisyfos Digital AB, uses Bluemix and for this report IBM Bluemix will be used for cloud computing [2]. Bluemix offers a so called Auto-scaling service for the applications running on their cloud. This service offers automatic resource scaling in regards to metrics of the application, to meet the customers demands. This service could be in great interest since it offers to save money, with no downtime. This thesis will evaluate how viable this auto-scaling service performs with a simple implementation of microservices, with the scaling focus on response time and throughput.

2 Background

2.1 Technology and tools

2.1.1 JSON

JSON (JavaScript Object Notation) is a text based, lightweight, language independent and human readable format for data interchange. Due to its characteristics it is suitable to data exchange between client and server and it is also available for a lot of programming languages [3].

```json
{
  "name": "Kalle",
  "id": 328282,
  "knownLanguages": [
    {
      "type": "English",
      "skill": "Novice"
    },
    {
      "type": "Swedish",
      "skill": "Fluent"
    }
  ],
  "location": {
    "country": "Sweden",
    "city": "Stockholm"
  }
}
```

Listing 1: A JSON object example

2.1.2 NoSQL Database

NoSQL were originally a combination of the two words "no" and "SQL" reffering to users who did not want to use a relational database. This term is often most used for describing a database that does not follow the principles of relational database management systems (RDBMS) [4]. In comparison with relational databases, NoSQL handles unstructured data such as e-mails, word-files or multimedia while still processing the data faster and scaling better. Three popular types of NoSQL databases are: Key-value stores, column-oriented databases and document-based stores [5].

Key-values NoSQL databases stores values which is indexed by an index called a key. A document-based NoSQL database will use a document as a value which is linked to the key, and this document may be of e.g. a JSON type.
2.1.3 REST API

Representational state transfer (REST), also called RESTful when implementing REST, is a Web architectural style that follows certain constraints for Web service communication. These constraints, in its core, includes being a layered client-server system which is stateless and cacheable [6]. However, the client and the server must agree on media through a uniform interface (URI). The URI enables identification between requests which empowers the system to act differently depending on e.g. the request URI or media type used.

For web services, an application programming interface (API) offers an interface of communication between the client and the server. REST API is the concept of applying the RESTful architectural style with an API. RESTful APIs most often use HTTP operations such as GET, POST, PUT and DELETE. One benefit of using a RESTful API is the cross-platform character since the requests need no information except the URI of the requests.

2.2 Cloud computing

Cloud computing is a model for an on-demand access service to share a remote pool of configured computing resources with the users having a minimal management and service interaction [7]. The users only pay for the resources that they use and, depending on the cloud computing service, will often avoid the need to maintain the infrastructure or resources.

Cloud computing have five characteristics: On-demand self-service, ubiquitous network access, resource pooling, rapid elasticity and measured service [8]. By an on-demand self-service a user will automatically have the resources that is required without any human interaction. Regarding the second characteristic the requirement is that the user can access these resource platforms from anywhere on anything from a tablet to a laptop. Resource pooling means that the user will share the available resources, and the resources are being pooled to meet all the users demands. With a measured service the users will only pay for the used resources. The rapid elasticity of the cloud will give the illusion of having infinite computing resources to the user, but the resources are elastically increased or decreased depending on the current usage.

2.2.1 Cloud Service models

Generally speaking, a cloud can offer three different services to its users; software as a service, platform as a service and infrastructure as service [9].

**SaaS:** A SaaS offers a software application to its clients, often accessible through a web browser. This is the highest level of the services offered, and the users have no control over the application, environment or hardware.

**PaaS:** For a PaaS the underlying hardware and infrastructure are supplied, such as virtual servers, environment, operating system and developer tools. The users will upload its application which will run on the PaaS and often, some configuration will be available to the user.

**IaaS:** The IaaS will only supply the hardware such as local storage, physical servers and the minimal software for it to run. It is then up to the user to configure and install the needed software, runtimes and its configurations.

2.2.2 Cloud scalability

The ability to handle increased or decreased usage demands is called scaling. Scaling is often used for cloud computing when users only pay for the resources used [1]. They are essentially two different kinds of scaling: horizontal scaling and vertical scaling.

**Vertical scaling:** Also called scaling up or down. This is when the application either increases or decreases the resources that are available. However, this requires a downtime due to the resources are being updated and replaced.

**Horizontal scaling:** Known as scaling in or out, and is when you increase or decrease the number of instances which have identical resources. By changing the number of active instances, there is no downtime. Due to having multiple instances of the same resources a so
called load balancer is often used. The load balancer monitors and distributes the incoming request between the instances [10].

Figure 1: Visual examples of horizontal and vertical scaling. The dotted boxes represent a new instance after being scaled, and the non-dotted represent a instance before being scaled.

2.3 Service Oriented Architecture

2.3.1 Microservices

Microservices are an implementation approach of service-oriented computing and is used for creating and breaking down the complexity of programs into modules. These modules aim for flexibility, independence, continuous delivery and scalability [11]. The modules, called services, aim at performing only a single function and nothing more. These services should be as independent as possible to the rest of the system. Microservices often use lightweight communication protocols e.g. REST or HTTP, but may use messaging protocols such as RabbitMQ. However, there are no standard communication protocol for microservices [12]. Using microservices offers the opportunity to scale individual parts of the system up or down depending on the microservice usage. The microservices can be distributed, since they act independently, but this introduces an increased system complexity. Since microservices can be distributed, microservice may be a suitable architecture used for clouds.

2.3.2 Spring

Spring framework is an open source extension to Java SE and at its core works with dependency injection (DI) and inversion of control (IoC) principles. All dependencies and relationships between methods are created by the so called IoC container. This IoC container handles the instantiation, the life cycle and destruction of all objects, which is injected into the main program [13]. All objects handled by the IoC container is called beans and all bean dependencies are bound by configuration metadata which the IoC container uses. The metadata takes form of a XML-file with its configuration where the dependency injection is specified, among other things. [14]. A dependency injection can take form in two ways: constructor-based dependency injection and setter-based dependency injection. Some of the benefits of using dependency injection is to achieve cleaner code while Spring handles the lookup for the dependencies and injections. Therefore the code is easier to maintain, more modularized and easier to test.

Since Spring is suitable for developing web application an extension of Spring is Spring Boot which makes everything much easier. Not only does Spring Boot help with the configuration of beans in the XML-file, it auto-generates it if the right annotations in the code is supplied. Spring Boot also contains an embedded Tomcat server, and offers production ready Spring application [15]. According to Redmonk, Spring is the domination framework for Java and Spring Boot is on an exponential rise [16].
2.3.3 Maven

Maven is a tool for building and managing Java applications. Maven uses its project object model (POM) and plugins to build projects [17]. A dependency is a project dependency on e.g. a JAR-file that the project requires to have in order to run. That dependency may either be an internal local dependency or an external dependency from e.g. a framework. The POM-file include, among other things, the dependencies that are needed to build the project. Maven resolves all dependencies in the project and builds it accordingly to its configuration in the POM-file. If a dependency is not found in a local repository, Maven will download it [18].

2.4 Bluemix

Bluemix is a platform as a service (PaaS) that is built upon the open source Cloud foundry. Bluemix enables its users to create, manage and deploy cloud applications without the expertise knowledge needed to install or configure them. [2]

Cloud Foundry abstracts the infrastructure that is needed to run a cloud, and still offers a variety of options. Cloud foundry offers support for many different development frameworks and application services for different clouds [19]. It also supplies buildpacks, which automatically provides framework and runtime support. This buildpack also configures the deployed application to make the application run and communicate with used services. If a buildpack is not available you can still use community buildpacks to support your application.

Since Bluemix is built upon Cloud Foundry a lot of functionality is inherited to Bluemix e.g. Cloud Foundry command line interface. This command line allows users to manage and create cloud application. In addition to the inherited functionality, Bluemix offers additional services, monitoring and management of the applications resources.

2.4.1 Services and runtimes

When an application is running on Bluemix, runtime environment variables about the application and the bound service credentials are available [20]. The environment variables include information such as disk space, memory and application specific system variables e.g. instance number. The service credentials often include information used for connecting to internal or external services. Bluemix has a variety of services and runtime environments but some examples are:

**Liberty for Java**: Applications running on Liberty for Java, is supplied by liberty-for-java buildpack which supplies the runtime Java EE [21]. If a Java application is deployed to Bluemix, the Liberty for Java buildpack is used when deployed as: standalone applications, server directory or packaged server. These different types of deploys offers the users to decide what to deploy, and at its minimum, a standalone web application can be supplied and Bluemix handles the rest. At the same time it offers high configuration options for users who want to configure its environment and server by deploying e.g. a Liberty packaged server. The buildpack also supplies a WebSphere Liberty application server which is developed for cloud usage [22].

**Cloudant NoSQL DB**: Is a document-oriented cloud database which stores documents in a JSON-format. It stores the document in database clusters, with a focus on scalability, high availability, and durability [23]. Requests to Cloudant are made through API HTTP requests.

**RabbitMQ**: RabbitMQ offer communication between application through asynchronous messages queues [24]. An application can send and receive messages by using queues, which other applications can subscribe to. The messages are stored safely in the queue until an application fetches the message.

**Auto-Scaling**: With this service Bluemix will automatically scale available resources to the application in or out [25]. Bluemix does this automatically through user configured parameters called policies. When a policy is triggered, namely when a condition is met, scaling activities take place. This includes both for scaling in or out. Bluemix does this by injecting an Auto-scaling agent into the runtime, which monitors the metrics and compares it with the policies. The agent then makes scaling requests accordingly to the policies [26].
Depending on the used runtime, different policies are available. When using Liberty for Java scaling may be triggered by heap usage, memory usage, throughput or response time. By using the Auto-scaling service, all of these metric types are being monitored. Each policy has a higher or lower limit for the metrics, when to scale in or out, but also other parameters exist such as: Statistic window, breach duration, scaling in and scaling out cooldown. All these parameters have the unit seconds. Statistic window specifies the length of the past time period for when data is valid, namely the metrics that exceed this time window are not recognized as valid. The breach duration expresses the time period that is needed to be exceeded for a scaling action to take place. Cooldown parameters express the cooldown after an action, before it can take place again. It is available to modify the minimum and maximum instance count, to limit the possibility of using too much resources.

The Bluemix router works like a load balancer and distributes the HTTP requests randomly among the active instances. 

One thing to note is that these services add a memory overhead for the bound application.

### 2.5 Related Work

Aad Versteden and Erika Pauwels published "State-of-the-art Web Applications using Microservices and Linked Data" [28]. This paper covers how microservices interact with each other when interchanging messages with Linked Data, built as web applications with popular JavaScript frameworks. The paper both includes internal and external communication and which microservice dependencies they get.

Amir Fazli et al. [29] published "The Effects of auto-scaling in Cloud Computing on Entrepreneurship". This work investigates the entrepreneurs decision to enter a new market, in regards to meeting the computational demands and only paying for resources which cloud computing and auto-scaling offers. The authors create a game theory model to examine the decision made for expanding their market.

Alessandro Papadopoulos et al. [30] published the journal article "PEAS: A Performance Evaluation Framework for Auto-Scaling Strategies in Cloud Applications" which addresses issues with proposed auto-scaling strategies for improving Quality of Service indicators. The issues are often linked to faulty testing, or very specific test conditions. The research group presents a Performance Evaluation framework used for auto-scaling testing which covers more general conditions and extensive testing.

Rehan Saleem published in his Master Thesis "Cloud Computing's Effect on Enterprises" in 2011 [31]. The thesis is focused on the data security and cost aspects, the gain and flaws, of cloud computing for enterprises. It also covers what size of the enterprise is the most suitable for using cloud computing.
3 Requirements - Challenges

In this thesis there will be an implementation of microservices which will be uploaded to IBM Bluemix and clients will be able to communicate to this service through Bluemix with a public REST API. The microservices will also communicate with each other through REST API and the saved data will be stored in Cloudant NoSQL.

The testing of the auto-scaling service will be evaluated primarily on the throughput and response time for the server to complete the request, in correlation to the amount of requests and the amount of instances running on the cloud.

- What were the difficulties with the implementation?
- What horizontal scaling performance can be achieved?
- How adaptable is the horizontal auto-scaling to the incoming requests?
- When is it relevant to use the auto-scaling service?
4 System Design

The system is based on microservice architecture and have three individual components: Storage, Data computation and Dispatch handler. Each microservice acts independently, except the Dispatch handler, and the internal communication between the services uses RESTful API calls. Each component follows the microservices architecture and are made for in and out horizontal scaling, to continuously match the demands of incoming requests.

The service this system offers is to GET, PUT, POST and DELETE values with an ID on the server. At the same time it does a data computation before returning the response as a JSON object to the client.

Figure 2 gives an overview over the system. The blue bar represents client requests and the orange bars represents load balancers which distributes the requests between the application instances. Each green box is a service, which may contain several instances of the same application.

4.1 Storage

The Storage microservice implementation a RESTful API, and offers the service to save values with an ID to the Cloudant NoSQL database. All values with their corresponding ID will be stored as a JSON object with its revision number.

It only handles incoming RESTful request and takes either ID or both ID and value number as input depending on the operation. The storage service will try to execute the operation against the database and regardless of successfulness it will return a JSON object with its parameters and the total time it took to complete the request.

4.2 Data Computation

Apart from handling all requests through RESTful, this service offers a simulation for data computation. It takes a JSON object as input, with the same format as Storage microservice application, it also does a data computation before returning the time it took to complete inside the JSON object.
4.3 Dispatch handler

Through RESTful request it extracts the needed ID and value number, in order to act like a proxy between the client and the Storage component. From the Storage response it redirects the JSON response to the Data computation component, before returning the new returned object to the client with the time each component took to complete its request.

5 Implementation

All applications were uploaded and hosted at the nearest Bluemix cloud, which is located in the United Kingdom. Applications and Bluemix services can be both configured and uploaded through either Bluemix web page, by using Bluemix command line interface (CLI) or through Cloud Foundry. Cloud Foundry was mostly used for creating, deploying and updating the application on the cloud, while the web page was mainly used for adding, configuring and binding the services to an application.

The plan was to create lightweight microservices to simulate a real service for a realistic evaluation of how suitable the auto-scaling service is for an application. However, the microservices could not be lightweight on Bluemix cloud, since each microservice needed to be deployed as an own application to be able to scale individually. Thus making each application having the need to include the used libraries, to be able to run as a web application.

A vital aspect for having a viable evaluation was to have a simple, consistent and realistic implementation of the microservices, to narrow down the variables affecting the performance of the microservices. Another aspect to its simplicity was due to the costs. The more complicated and resource consuming applications, the more it would cost to scale. An application often includes information to be saved, and computation on the data. Therefore three microservices was needed to simulate this application; Saving data on a database, data computation and the main dispatch handler for handling the requests between the microservices.

5.1 Motivation for tools and methods

5.1.1 Infrastructure

The first steps involved getting familiar with the different services and what options Bluemix could offer for developing microservices. The requirement was that Spring Boot was to be used for development. In order to create an application, one has to choose an underlying infrastructure and environment, Bluemix offers a few that are suitable for Java.

The motivation for choosing Liberty for Java was primary due to the fact that the company of the thesis, Sisyfos Digital AB, uses this infrastructure. Also, the Auto-scaling service offers more option for scaling metric for Liberty for Java, which is not available for other infrastructures.

5.1.2 Communication

Having the underlying infrastructure for the microservices, the next step was to evaluate and determine how the microservices should communicate with each other. Given the available option for Bluemix, and the options for communication between application in the Liberty for Java environment, it all came down to RabbitMQ or creating a callable API for each application. One could argue that RabbitMQ is more suitable for developing microservices that has an asynchronous request queue that handles high bursts of messages better, while an API is more responsive and the request will be processed synchronously.

The main reason for choosing an API over RabbitMQ was the pricing for using the service RabbitMQ on Bluemix. There were other reasons, as it would still be needed to have a public API for clients to connect to, even though RabbitMQ would be used internally between the microservices. It is also convenient to have a mutual API standard for all applications since the application needs a REST API for the clients to call when saving information on the database.

5.1.3 Database

When deciding between the different databases Bluemix had to offer, all databases that had a non-free pricing were excluded. The options got limited down to two databases, which had the database
models NoSQL and SQL respectively. The only information that would be stored on the database would be a key called ID, its value and time checkpoints therefore making a relational database seem overcomplicated for its usage. Cloudant, the NoSQL database, makes both handling and returning objects a lot simpler as the NoSQL database saves its information in JSON documents and need no parsing.

5.2 Execution

The first step includes getting familiar with Spring and its benefits for developing web applications. The extension of Spring, Spring Boot, simplifies the dependency injections and Spring configuration, making less code needed to be written, to be able to have a running web application. IntelliJ IDEA was helpful for generating the POM-file, which specifies the dependencies needed from external libraries. Maven then reads this POM-file, download the needed libraries and compile the package into a runnable web application. All these tools made building a web application very smooth with minimal configuration.

In the early stages of the thesis it became clear that all applications need to store the time taken for different stages, for each request. Since the Dispatcher applications response time, and throughput would be dependent on the other two web applications, these timings of each request would be crucial for determining bottlenecks and evaluating when to scale the different applications. These values would also be needed during the evaluation and during the creation of the result graphs. The easiest choice of how to store these different timings was in the JSON object that each application would return. If any errors occur in any service, e.g. a network error, the application will continue its process but with an empty JSON object with its timings and error status such as the HTTP status code. The application and Cloudant NoSQL communication is done by synchronous client-side HTTP requests, called RestTemplate. RestTemplate enforces RESTful principles and handles all connections when the application provide the URLs, headers and request body. The communication between all applications also use RestTemplate to standardize the RESTful principles.

5.2.1 Storage Microservice

The first microservice implemented was the Storage application, which is responsible of handling the storing and the fetching of the saved data. It is connected to the Bluemix service database Cloudant NoSQL. The Cloudant database is limited for the free version to 20 read and 5 writes a second, making this database the bottleneck for all microservices. The application has a simple implementation and acts like an interface to the database. Through REST API calls, the application parses the URL from the request along with the parameters and method, and forwards it to the database. Using the library Spring Cloud Connectors the application can still operate both locally and on the Bluemix Cloud Foundry, without any changes in the code. This library helped with extracting the needed credentials, to connect to the Cloudant NoSQL. These credentials were extracted from the Bluemix environment variables, and for local usage Spring retrieves the credentials from a local property file. When making requests to the Cloudant database for all operations except GET, the revision and id value must match. If not, the operation will fail and therefore all requests need to always start off with a GET-request to retrieve the metadata. If an error occurs, such as trying to delete an object that does not exist, the Storage application still will return a JSON with its timings but with some empty fields.

5.2.2 Computation Microservice

The Computation application was almost implemented in the same way as the Storage application, in the aspect of handling incoming requests. The main difference is that it does a computation simulation, which would represent a realistic computation used by an average application. The idea was to make a static computation taking around 320 milliseconds to complete, not to make the computation relative to each request. The reason for the static computation was to not interfere with the result evaluation, in correlation with the type of request, but to still make a difference when handling a lot of requests. Therefore a function for calculating factorials was implemented. To match a single computation to take the time 320 milliseconds, the factorial number was set to 25,500 after some testing. The time it took to complete the calculation is returned inside the JSON object to the caller.
5.2.3 Dispatcher Microservice

The dispatcher application was created last among the microservices, and acts like an interface for the clients. The application itself only acts in a similar way as a proxy server between the client and the other two microservices, by forwarding messages sent by the client. It also adds its own timings before returning the JSON object which it has received from the other services. The dispatcher has the same RESTful API as the storage application.

5.2.4 Load testing

A customized load tester was developed in order to customize the testing and extract the internal timing values from the application, which was used during the evaluation. The goal was to develop a tool for testing the microservices scalability and performance by executing a set schedule of requests.

The load tester takes the input of a schedule as a list. Each element in the list represents the amount of requests per second that the load tester will execute for a given time. Each request the load tester does gets executed in a asynchronous thread by a ThreadPoolExecutor which sends a HTTP request to the Dispatcher microservices API endpoint. The ThreadPoolExecutor provides an improved performance for executing a large number of asynchronous tasks and will either reuse idle threads or create a new thread for each task. Every task is a function call which executes the request against the endpoint, and the function will return a Future<> which is the result of the functions request. However the results from each thread will not automatically return due to its asynchronous nature, so after each completed task, it needs to be checked if any thread has completed its task, to fetch the result and mark the thread as done. If any thread has completed the request it saves the response.

The request follows the RESTful principles and variates the different REST operation by randomizing the operation as following: 70 percent chance for a GET request, and 10 percent each for POST, PUT and DELETE requests. The ID values used are also randomized but limited to a few hard coded variable IDs. After all requests has been made, all results will be extracted from the saved responses and exported to a spreadsheet. The spreadsheet contains the following information:

- **Request Time**: The time the request was made.
- **Request per second**: Specifies in what interval the request was sent with, in relation with other requests.
- **Sequence Number**: The sequence number sent.
- **Overall Time**: The time it took to get a response from the API endpoint, in milliseconds.
- **Overall Service Time**: The time it took for the application to respond to the request in milliseconds. This includes network delay.
- **Internal Service Time**: The time it took for the application to handle the request internally, in milliseconds. This excludes network delays.
- **Application instance**: The instance that handled the request.
- **HTTP Status Code**: The status code for the HTTP request that the was returned.
- **REST Operation**: The REST operation that was made from the Load tester.

*Includes all used microservices which is separated in different columns.
5.2.5 Binding Bluemix services

When using Bluemix services for Liberty for Java, it is vital to deploy the application as a packaged server. For some Bluemix services, a service agent needs to be injected into the Liberty container to be able to run and read data. With Spring Boot, if not packaged as a packaged server when deployed to Bluemix, the buildpack will not install Liberty for Java only Java EE and the application will run on Spring Boots embedded Tomcat container. When deployed as a packaged server the buildpack installs Liberty for Java and the application will run on the Liberty web container. Thus, the service agent will be able to get injected. Also, when deployed as a packaged server with its server configuration, it became more effective due to the application being launched much faster, which would increase the responsiveness of the scaling. This was not known and taken into consideration in the early stages of the thesis, and not detected until the latest stages of the implementation. Otherwise the adding of Bluemix services was easy and effortless.

6 Method evaluation and results

The evaluation was split into three parts: initial testing, instance testing and auto-scaling testing. Two testing schedules were developed for the use of testing each evaluation part. Both schedules had some warm up with low amounts of requests. Each test took about one hour to complete. The time needed for a single test was approximated because a lot of testing was needed, but at the same time the testing could not be too short due to the responsiveness of the auto-scaling policies. For example bursts with the duration below 3 minutes would not give the system a chance to scale, and therefore would be impractical to test.

The first schedule simply increased the request sent per minute by one, starting from one request per minute, and continued until the set maximum number of request is achieved. This schedule will be called *linear schedule*. The linear schedule was made for testing the maximum throughput, for different amounts of instances, and the number of instances needed to handle the amount of set requests, while the response was still acceptable. The following notation will be used in some cases for specifying the amount of Dispatcher and Computation instances. The first number specifies the amount of Dispatcher instances, and the second is the equivalent for Computation instances.

The second schedule is called *burst schedule* and was made for testing a viable scenario where the incoming requests comes in bursts. The idea was to simulate a schedule where the requests had a correlation with a typical work day for the average work person. Which means that the amount...
of interaction with a work system is different depending on the time of the day. The example was set to be low before and after the working hours, but also during lunch. The pressure on the system would be high during the morning and after lunch, and medium the rest of the working hours. All different transitions between low, medium and high have a short linear transition of requests.

![Figure 4: The burst schedule: The X-axis represents the time in the schema in minutes, and the Y-axis is the amount of requests per second.](image)

Due to the bottleneck of the Storage application, the limit of the amount of requests per second was set to 18. The application can handle more, but if a service falls behind with the current incoming requests, there must be an overhead for the storage service to recover if the requests are not diminishing. This is impossible if the maximum amount of incoming requests are set to 20 requests per second.

Due to this low bottleneck limit it became clear during the testing that the storage microservice would never have to scale, because of the fast response time by a simple database request. The storage service will therefore not have a scaling mechanic, but will not be removed from the system during testing.

All information received is either from the data extracted from the JSON that the web application returns, or through Bluemix auto-scaling metric statistic tab. The metric statistic tab shows the activity for the application for the last 30 minutes. There is no way to export the information shown in the metric statistic tab except through screenshots, and the information disappears after 30 minutes. Each application has 512 MB as maximum memory size for the following tests. Some pre-testing of the microservices gave the results that it would be sufficient with a maximum of three instances each for the microservices to handle the two schedules with an acceptable response.
6.1 Testing

The initial testing used both schedules for evaluating the standard performance of how the system, with different instances, should perform with auto-scaling service active. The thought was to have some standard values to compare the auto-scaling against; of how the scaling ideally should perform if there was instant startup of the instances when scaled and having the optimal scaling policies. These results would give information such as averages for different amounts of requests per second, both from the clients and applications perspective.

The linear schedule are only used for determining the maximum limits for different kinds of instances, while the burst schedule will be used to evaluate the auto scaling performance. All of the following testing was made between the 17th to 22nd February 2017.

6.1.1 Test results

Due to the knowledge that it would be sufficient with a maximum of three instances each on Dispatcher and Computation applications, the schedules were tested for each combination of a minimum amount of one instances on each application to a maximum of three. A total amount of nine tests were therefore made per schedule.

**Linear schedule**

![Figure 5](Image)

Figure 5: A diagram of the client side average response time for different instances. If a combination of instances are not responding within 3 seconds, the combination is removed from the graph. The notation of the instances is written in the number of instances for Dispatcher application first, with the number of Computation application instances after. For example 3-2 stands for three Dispatcher, and two Computation instances.

Figure 5 shows all averages of the response times for different kinds of requests with the linear schedule, each request per second had the duration of three minutes. The maximum response time in the graph is cut to a maximum of three seconds, but in some cases it could reach up to two minutes at which the request has a timeout and was discarded. This gave a hint of how many instances needed to be active to handle different kinds of requests. For example, when sending 20 requests per second, three instances of Computation and Dispatcher instances would be sufficient as well with two instances each. However the latter would have a slightly higher average response of 136 milliseconds.
The different average timings for the components of 5 requests/second, is shown in figure 6. These different components have a correlation with each other but still gives the knowledge of what component is causing an increased overall response. The oDispatcher is sometimes refereed to as Overall time, by being the time it took for a client to get a response from a request.

Figure 6: The different components average timings. The i stands for internal timings, the time taken for each applications internally, and the o stands for overall timing. That is the time it took to complete a request to that application.

For example: a request to the Storage application takes an average of 100 milliseconds to complete, but when the application receives the request the time taken is only around 25 milliseconds. All these shown in figure 6 values are the standard timing values the system as a whole should produce.

Figure 7 shows a comparison of these averages for one instance each of the Dispatcher and Computation application. The first value is when the client makes 7 requests per second, and all values are normal and similar to figure 6. However when sending 11 and 12 requests per second, the client has an increased average response with roughly 1530 and 6500 percent respectively, while the internal values for Computation and Dispatcher application does not increase in the same volumes.

Due to unknown reasons the client has a greatly increased response time in comparison with what the application perceives that the response time is. The server and the client might have different timing due to the heavy pressure on the application in some cases. So even though it is the clients response time matters the most, the internal timings of the applications mapped to the clients response time. In figure 8 the perspective of the applications components auto-scaling agent is shown. Although we can see the increasing response time from the application and auto scaling agent, it is not in proportion with the clients actual response time shown in figure 7.

Loosely mapping the clients response (figure 7) with the internal values shown in figure 8, gives a hint that the throughput has a limit for a single instance at around 10-12 requests/second, and if above around 10 requests/second the response starts to increase significantly. However no real conclusion from the response time can be set, except if it is above the average standard response of approximately 380 milliseconds, the response time for the client starts to grow significantly.

During these tests the Dispatcher application is also covered, but seemed pretty insignificant as long as the Computation application could keep up with the requests. Shown in figure 9 there is no real difference in overall response for the client, when different Dispatcher instances are active. This also is shown by Bluemix auto-scaling agent in figure 10 when the Dispatcher and Computation instances are 1-3. Figure 10 shows that the throughput from the Dispatcher application is still stable and is sufficient up to 20 requests/second.
Figure 7: Component comparison between the averages for different requests per second, when all applications have only one instance.

Figure 8: Computation response and throughput of one instance, from the auto scaling agents perspective. The colors mark in what interval the client sent the requests at.
**Burst schedule**

The burst schedule shows similar information as the linear schedule did. If the throughput exceeds 10 requests/second, the response will increase regardless of the amount of active Dispatcher instances. Therefore the standard comparison with auto-scaling will be done with the averages for one Dispatcher instance and three Computation instances shown in figure 11. However some of the testing had some unknown increased response time during 18 requests/second, which explains having a higher average for that interval, but never exceeding 3 seconds in average response time each.
6.1.2 Auto-scaling testing and results

Before starting any scaling, the task was to measure the startup time for an instance which will come in handy for setting the policies. The auto-scaling reported e.g. in one case that it launched an instance at the time 1:59:48. From the clients perspective, the first request that was handled by this instance was at 2:00:34. Comparing these two values for all started instances gave the average startup time of 46 seconds, but these values ranged between 30-70 seconds. By this we can expect that it will at least take 45 seconds for an instance to start responding to requests, after the auto-scaling agent has scaled up with another instance. All these values are on average, and there might be some time delay between the two different clocks on which the time is based, but it gives an approximation. An example of how the client sees its response time when the amount of Computation instances are scaled to two is shown in figure 12. At the time 12:29:30 the auto-scaling agent starts scaling up with one instance, and the instance is active at 12:30:21 (black line marked 1). That is after 50 seconds from when the agent scaled, the second instance helps with stabilizing the response time for the client. Around 12:31:34 the application has fully recovered and responds with a reasonable response time.

Figure 12: The response time for the client, for one instance of Dispatcher application, when Computation instances are scaled to two instances. The X-axis is in milliseconds and represents the time passed from the beginning of the graph. Each color represents a Computation instance. The 1 represents when the second instance starts being active, and the application starts to recover.
The next step was to experiment with the policies, to further evaluate how they are working in practice. When defining policies, the configuration factors that matters is statistic window, breach duration, scaling cooldown and the metric type and its values. Statistic window and breach duration are the two configuration values that the rest of the configuration is based upon. The smaller the breach duration is, the less time is needed to trigger the scaling if an average metric value is exceeded. In correlation with breach duration, the statistic window specifies the time range in which the averages for the metrics are calculated. The minimum value available is 30 seconds for both configuration. The theory was that both these policies needed to be set to minimum, due to the rapid changes of the requests. This was confirmed by the test results shown below, where all metrics are in seconds and all configuration not listed is the same for all the tests. The tests scale up and scale down metric are in relation with each other, the time shown in these cells are the time in comparison with the scaling for the cell above. For example, the Test2 started scaling up 17 seconds after Test1 did, but scaled up 33 seconds before Test3. The same logic applies to the scale down cells.

<table>
<thead>
<tr>
<th></th>
<th>Breach</th>
<th>Statistic Window</th>
<th>Scale up</th>
<th>Scale Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test1</td>
<td>30s</td>
<td>30s</td>
<td>0s</td>
<td>0s</td>
</tr>
<tr>
<td>Test2</td>
<td>30s</td>
<td>60s</td>
<td>17s</td>
<td>24s</td>
</tr>
<tr>
<td>Test3</td>
<td>30s</td>
<td>120s</td>
<td>33s</td>
<td>54s</td>
</tr>
<tr>
<td>Test4</td>
<td>60s</td>
<td>120s</td>
<td>58s</td>
<td>180s</td>
</tr>
</tbody>
</table>

When scaling up much earlier, the response time got significantly better average results, especially since the burst only lasts for 10 minutes before returning to medium or low amounts of requests. The cooldown for scaling in and out was set to two minutes, and was not tested to a great extent. Since the startup time for an instance was at average 45 seconds, the instance need some time to catch up and become steady.

**Response time testing**

From the information above, the optimal settings is used: the breach duration and statistic window is set to 30 seconds, and the cooldown for scaling in and out is set to 120 seconds. The average response time for the internal Computation application was used for determining when to scale the application up or down, found in linear testing. Therefore for when to scale up was set to 450 milliseconds or above, and to scale down when below 350 milliseconds. However when defining these upper and lower scaling conditions for response time, the thought arose that defining when to scale down based on response time is not a good idea. Its clear that if the response time is above the average, another instance may be needed. On the other hand if e.g. two instances are producing good response time, it does not necessary imply that the application should scale down one instance. This occurred in the first test, the scaling up went pretty smooth but the application scaled down when the optimal amount of instances were active.

![Throughput](image1)

![Response Time](image2)

**Figure 13:** The scaling for Computation with response time metric. The 1 stands for when scaling down, and 2 for scaling up.
Figure 13 shows the application scaling down to one instance when receiving 14 requests/second, even though it is not enough with only one instance. However, when having two instances active it produces good response time and therefore scales down, and then scales up again when it notices that it can’t handle the requests with one instance. No matter which policy configurations are active, and the metric type is set to response time, the amount of instances will never be stable even if the incoming requests are. Therefore scaling down on response time is a dead end. Scaling up works within reason, but it will only scale when the requests are getting out of hand.

Throughput testing

The policies for throughput testing is set as above, with the exception of using the throughput metrics instead of response time. By using the information gained from the linear schedule, the Computation throughput for a single instance can reach up to 12 requests/second. However with above 10 requests/second the response time for the client starts to increase. By setting the throughput to scale up at 80% of its maximum capacity when responding within reasonable time, each instance have an overhead to be able to react in time and respond to bursts. With an overhead the application will scale up when it senses an increased flow of requests, giving a new application time to start up before the burst and not during it. Since the metric type calculates the average throughput of all the instances, when scaling up and the throughput of a single instance is e.g. 9 requests/second, then when the other instance is active the average throughput will be 4.5 requests/second. Therefore the scaling down configuration need to be at maximum 50% of the scaling up throughput. Otherwise, if the incoming requests are above the scaling up threshold, it can still be below the scaling down threshold making the application scale up and down in until the incoming requests change. Therefore the scaling down throughput was set to 4 requests/second.

The results in comparison to the standard graph of the burst schedule for 1-3 is shown in figure 15. The scaling was overall good, it scaled up before the bursts arrived, but due to the cooldown of the scaling up it took some time to scale up again to match the requests. It did however recover quickly, but the overall average response time went up in some cases. The table below shows how long it took before the application got scaled up or down, in seconds. For example the system scaled up one instance 32 seconds after the client started sending 9 requests/second.

<table>
<thead>
<tr>
<th>Requests/second</th>
<th>Scale up</th>
<th>Scale Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>32s</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>63s</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>44s</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>43s</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>47s</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>55s</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>37</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>30s</td>
</tr>
</tbody>
</table>
Figure 15: The average response time comparison between the standard 1-3 (blue) and the scaling attempt (red).

Even though the scaling start up timings are similar for almost all intervals, the second wave of 14 and 18 requests/second is much higher than the first. One reason to why the 14 requests/second interval is so high is due the length of the time period, which is only 1 minute for that interval. The 18 requests/second interval continues for 10 minutes, which gives the system the time it needs to recover, and therefore the average response time is much lower. A graph over the recovery of the system is shown in figure 16, which explains why some averages are high. As seen below, the system recovers relatively fast, but an average response time alters somewhat with this impression. The total time it takes for the system to recover to the average response time is around 3 minutes. A lot more testing was done but scaling up on 8 requests/second and scaling down on 4, gave the best results for this scenario.

Figure 16: The overall response time for the second burst shown in figure 15. The 1. marks the end of 14 requests/second, and the start of 18 requests/second. The 2. marks the time for when the third instance is launched.
6.2 Discussion

Regarding horizontal scaling for an application, it all comes down to the pre-testing done and the startup time. If the application deployed as a packaged server, Bluemix will offer fast startup time, which allows the application to recover from bursts in matter of minutes with no human interaction. However I would argue that this scaling service is not suitable to microservices, since you will only scale the whole application and therefore need to have all the microservices as an own application. This is makes the microservices not lightweight and thus inefficient.

Comparing the two metric scaling policies response time and throughput, throughput is superior. With the needed pre-testing of the applications limits, the key is to scale out before the applications instances are running out of resources. This is not achievable with scaling in regards of its response time, and the client will experience some delay. The bigger problem is scaling down, the application will never know when to scale down, since the response time might be acceptable while having the optimal active instances. However with throughput scaling, the scaling can occur before the maximum amount of resources is being used and the client will experience a minimal delay. There is a breakpoint of how early you want to scale in correlation with how many resources that can be idle. No real conclusion can be drawn of what horizontal scaling that can be achieved. The reason for this is for the low bottleneck of the microservices, which offered no real scaling performance evaluation. An increase of one or two instances gave the impression that the scaling could achieve a nearly linear performance increase per instance, but no information outside this range can be gained.

Another aspect to use this service is to identify the bottlenecks and when you want to scale your application. There might be cases when scaling will not offer an increased performance. Also there are scenarios where you can use the auto-scaling service but not use the policies used in this thesis. An option for date and time scheduled scaling exist, and might be an better option than to scale depending on metrics, if an application experience a repeating schedule of requests. However having a scaling metric upon all application might be a good idea, even though only minimal testing is been done. It gives some kind of performance guarantee that e.g. an application will slowly scale if the usage has increased and no maintenance is done during a short time period during for example a weekend.

During these tests, the testing of the scaling should be longer to give the application more time to react and have a even more realistic scenario. In these tests, if scaled only when need to, the scaling adds a delay which have more impact on the results than if one test had been done during a whole day. Also having longer testing period for each test, the result averages become more stable, since the scaling time takes up less percentage of the total time of the tests. For example when the scaling activity takes roughly one minute and the burst has a duration of 10 minutes, 10% of the time the application is scaling one instance. During longer tests this scaling percentage time will be significantly reduced. Another note with longer tests is the configuration of advanced policies, such as breach duration, has a minimum of 30 seconds. In other words is the scaling configuration not very effective for short time scaling, and will have more success for longer test periods.

The different schedules also have a too quick transition between low amounts of traffic to high traffic. In 3 minutes the schedule can transition from low traffic, which one instance can easily handle, to high which is around 220% of what a single instance can handle. This can also be connected to having a bottleneck in the storage application, which limits the applications to be able to have a low amount of traffic. If the application could handle more requests, more scaling could be done and the scaling metrics would have a wider range of overhead of its measurement. Each requests/second takes roughly up 12% of the resources a single instance can handle. Thus making a "miscalculation" of one request may have a big difference if the application will scale or not.

The risks of have a set scenario of when to scale and test it, creates the risk of overfitting the scaling policies to a too narrow scenario. Often different scenarios need different scaling configuration, and making the policies too specific for a certain scenario may ultimately lead to the scaling works bad in the general cases except that certain scenario. Some overfitting could not be avoided due to the short time period of the tests, therefore the scaling only focuses on how suitable the scaling can become for only a single scenario. One can draw conclusion from this scenario of how suitable it is for other scenarios, depending on what could be customized and achieved. The scaling service for a general usage seem promising, but needs more research.
The original strategy of evaluation multiple applications with different scaling metrics would prove to be a bad approach. Since the Storage application would need no scaling, as well as the Dispatcher application, it only added complexity to the testing. It was unclear at the start of this thesis what would be the best scaling option for each application, but due to different bottlenecks only one application needed to really scale.

The implementation of the application and deploying it to Bluemix went fairly well. No real difficulty was encountered with the implementation, but with the testing. When testing the applications on Bluemix, some results for the same tests had a heavy variation in some cases. This made a lot of tests become corrupt and no real conclusion could be drawn. Therefore many tests had to be redone in order to have valid data, this caused the thesis to been drawn out and lack all fully covered testing.

6.2.1 Threats to Validity

Some results extracted from the tests may have be altered by network delay between the test computer and Bluemix, which is out of any parts reach. Also the test computer might have added some delay with the response time, due to the resources available to the computer was not optimal. However this resource shortage which may altered with the results, should in theory have the same effect on all the results.

Another aspect is the resource pooling on the cloud which may have altered the results. During the testing period, all tests could not be done at the same time and with resource pooling some results may have different results depending on the usage of other users applications. There was no modification on the garbage collector of the application which might have released some unused resources while the application was already under high pressure. This can both make the application get more resources, but also the time it took for the garbage collector may differ from test to test. Also when the garbage collector started freeing up resources matters, depending on how pressured the application was at that moment.

7 Conclusion and Future Work

Extracting performance results for evaluation from Bluemix would prove to be harder than expected. This may be the cause of the resource pooling of the clouds nature. However Bluemix auto-scaling service is a suitable option for low scaling applications with scaling on the metric throughput, if the bottlenecks are identified, the applications are implemented for scaling, and testing for the policies is conducted.

No real indication for how well the performance is enhanced for horizontal scaling can be obtained for more than three instances, but below four instances the performance for each added instance is linear to the performance of a single instance.

7.1 Future work

For a better evaluation of the performance of the microservices longer tests can be conducted. Removing the bottlenecks free the constraints for what horizontal scaling performance that can be achieved for a higher number of instances. There are also possibilities to evaluate the configuration policies more in detail, especially for the scaling metrics heap usage and memory usage, which was not covered in this thesis. Evaluation when combining policies are also available, e.g. scaling out on response time and scaling in on throughput.
References


[17] Apache Maven Project. What is maven?. [https://maven.apache.org/what-is-maven.html] [2017-02-28]


[23] IBM Bluemix Docs. Getting started with Cloudant. [2017-02-28]
https://console.ng.bluemix.net/docs/services/Cloudant/index.html

https://www.rabbitmq.com/features.html

https://console.ng.bluemix.net/docs/services/Auto-Scaling/index.html

[26] Sri, Lak; Vellanki, Bala; Willenborg, Ruth. Scaling applications in IBM Bluemix. [2017-02-28]

[27] Carew, David. Design, manage, and test multi-instance Bluemix applications. [2017-02-28]


http://dx.doi.org/10.1145/2930659