Scale smart, not hard

Theoretical approaches and ideas to improve cloud computing platforms by data-driven machine learning algorithms

Jonas Miederer
matriculation number: 32723
Stuttgart Media University
Nobelstrasse 10
Stuttgart 70569, Germany
jm104@hdm-stuttgart.de

ABSTRACT

In this paper some theoretical ideas are presented that may improve the usage of cloud computing services. Many users feel overburdened or do not have sufficient knowledge to configure their cloud service setup correctly. Other users, especially large-scale customers, have to manage a large amount of services, which makes it hard to constantly maintain and adjust to current needs. According to the model of Google Cloud Platform, an exemplary architecture is described with its core components. This architecture represents the status quo, a design that is leveraged similarly by other other cloud providers like Amazon or Microsoft. Based on these components, new theoretical ideas are introduced, described and evaluated. The approaches are primary data-driven and based on machine learning techniques. By this means, an adaptive, optimal configured system can be created that helps the users to run their services more easily and with less management effort. Although not all approaches would be completely useful, some ideas have the potential to improve scalability, resilience, costs and user experience of running cloud services.

KEYWORDS

Cloud computing, Google Cloud Platform, Machine learning, adaptive algorithms, scalability, resilience, user experience

1 INTRODUCTION

In the last decade, a whole new market emerged with the objective to provide strong computing power for everyone. These so-called *cloud services* offer infrastructures (*Infrastructure as a service (IaaS)*), like virtualized computing resources, platforms (*Platform as a service (PaaS)*), providing cloud components to software, as well as software (*Software as a service (SaaS)*). The four large companies *Amazon, Microsoft, IBM* and *Google* control the market of cloud computing with a market share of more than 68% as measured by revenue [6]. Although Google is still placed behind its competitors with a market share of only 6%, which corresponds to only a sixth of Amazon's revenue, they exhibit the largest gain of all cloud infrastructure services, with a year-over-year growth of 162%. This extraordinary and rapid increase indicates, that Google is becoming a serious rival for the leaders Amazon and Microsoft.

Every provider tries to dominate this highly competitive market by outdoing its rivals. This works especially well by signing contracts with well-known large-scale customers in order to advertise their powerful and reliable systems and to signalize that they can handle high loads of data without failing. For example, Amazon's cloud service *amazon web services (AWS)* is used by Amazon itself, Airbnb, Comcast, Soundcloud, Foursquare and many more. But probably the most important customer is the video platform Netflix, which was accountable for more than 33% of up- and downstream traffic in North America during peak period in 2015 [8]. In contrast to that, Google's largest customer is the audio-equivalent, the music streaming platform *Spotify*. Other notable companies using the *Google Cloud Platform (GCP)* are *Disney, Coca Cola,* as well as Netflix, who use GCP as backup / replication service in case of an unexpected failure or data breach at AWS. Besides these external customers, Google also uses its GCP to host their own services like Search, YouTube, Gmail and Google Drive.

All of these companies using one of the cloud providers have common requirements regarding their applications, which led to the decision to host and run the services at external platforms instead of building their own infrastructure: First of all, the services have to be scalable. However, dealing with millions of visitors per day often lacks both know-how and money, especially at startups or services with high computational effort. Furthermore, it is easier, faster and often cheaper to add additional instances provided by a cloud service to an existing cluster instead of scaling up or out self-managed machines, once the demand is increasing.

Another important factor is resilience: Outages, downtimes or even failures should be avoided as much as possible, they can cause serious harm to the respective business. Nevertheless, external causes like earthquakes cannot be prevented, so absolute availability is hard to guarantee. In this respect the CTO of Amazon and co-architect of AWS, Werner Vogels, noticed that "everything fails, all the time" [9]. Instead of trying to build completely flawless systems, which is nearly impossible, one should assume that failures can occur anywhere and anytime. Therefore, cloud services need to provide high resilience, allowing the services to recover quickly and reliably.

However, this often causes further problems the operators of those services have to deal with: Especially for large-scale companies running sophisticated, complex or data-intensive services, the number of parameters required to build a reliable, secure and performant system becomes unmanageable. Each cloud provider offers different methods and techniques, each of them expecting a large number of configuration parameters. Moreover, this can be an obstacle for private customers with less technical background, too. They may be overwhelmed making all arrangements in order to smoothly run a simple service.

Cloud provider like Google contemplate to offer their customers the greatest possible flexibility. This is necessary in order to cover a wide range of user requirements and to be able to operate a large diversity of services, whether it is for a music-streaming platform or a cable company. The downside is that large-scale customers cannot invest the effort to constantly adapt the system parameters to the current needs and smaller customers often do not possess the know-how and expertise to fully understand and apply all of those parameters correctly.

This implies that the young industry of cloud computing is required to provide a simpler and more intuitive way for customers to manage their services. The most advanced approach would be a self-managing platform or system, which always finds its currently optimal parameters in order to run the services as reliable, available and fast as possible. This diminishes the configuration effort to a minimum, while even misconfiguration and a wrong choice of parameters can be prevented, increasing the overall system performance. This system should not only choose its optimal configurations, but should also adapt and manipulate them during runtime if necessary to react to the changing environment, demands and system load.

This work suggests and evaluates different proposals with the aim of a scalable and resilient self-managing system, driven by machine learning, based on Google's *Cloud Platform*. The second section of this paper describes the state of the art architecture and its core-components of a generic service running on GCP. The third section proposes some new ideas and techniques, how the current architecture could be improved and enhanced with machine learning in order to provide a better experience regarding scalability and resilience. The fourth section evaluates the approaches of the previous section concerning its feasibility and reasonableness. The last section covers the conclusion, summing up the insights and giving a brief outlook.

2 GOOGLE CLOUD PLATFORM

Google Cloud Platform (GCP) is the cloud service provided by Google and was initially released in 2011. GCP is not a single product or service, but consists of multiple IaaS and PaaS components, for example storage and database, networking, management and computing services. With a year-over-year growth of 162% is the fastest growing cloud service, which means that GCP is becoming more popular, while some of the largest companies like *Snapchat* and *Spotify* already decided to use Google's service.

Google itself advertises its services to be future-proof, which allows its customers to "grow from prototype to production to planet-scale, without having to think about capacity, reliability or performance" [5]. They identified four key requirements for their architecture in order to build scalable and resilient applications. Those requirements, its core concepts and components as well as advantages and disadvantages will be described in the following:

2.1 Region-based hosting

As mentioned earlier, it is nearly impossible to protect datacenters from external influences like natural disasters, hardware damages or data corruptions. Hence all cloud service providers take precautions to prevent loss of data and service outages, which harm both their own as well as their customer's business. Google and others try to counter this potential hazard by replication; GCP supplies their services at multiple places spread across the globe.

The central concept regarding availability and resilience is the idea of *regions* and *zones*: Google has currently defined six geographical regions (Western US, Central US, Eastern US, Western Europe, Eastern Asia-Pacific and Northeastern Asia-Pacific), which again contain one or more zones [4]. While components in the same zone can communicate with each other, the interaction of resources between multiple regions respectively between zones of different regions is not possible.

Therefore, in terms of availability and resilience, services should be hosted in more than one zone. Since the zones are independent, it is possible to keep the data or any service running at two zones in the same region. If an unexpected event occurs that leads to outages, the other zones will most likely not be affected. However, in order to provide a fast service, it should be located as close to the users as possible. Thus the operator can decide for the suitable region to host its service. If the primary userbase is located in Germany, the operator would probably decide for the Wester Europe region instead of the Central US region.

That way, GCP does not only provide resilience and availability, but can also decrease network latency independent from the operator's location.

2.2 Load balancing

Load balancing is essential for large-scale applications to distribute the total traffic. Assuming a simple service like hosting a website is publicly accessible. With only a few visitors, it is sufficient to leverage a single frontend server, delivering static files and communicating with the according backend server(s). This setup is very minimalistic, easy to deploy and to manage and there exists only one single endpoint for users. Nevertheless, this simplicity can rapidly lead to problems when the number of visitors is increasing.

The single endpoint constitutes a bottleneck, the frontend server has not enough capacity to serve all users at the same time, so that the service will not be available. As a solution, the operator can add additional parallel working servers, each of them representing an endpoint. Thus, the users have the choice which server to contact. Although this approach basically can solve the capacity limitations, it raises another problem: Each endpoint possess its own address, the single service is now available under multiple addresses. But for the users it is hard to remember multiple domain names or even IP addresses for each service. At this point, the load balancer is leveraged in front of the frontend servers. It again offers a single endpoint, so that each user request is accepted by the load balancer, but it does not actually handle it, but it redirects it to one of the frontend servers with low utilization. That way, the load balancer acts as reverse proxy, while it serves two main purposes: First of all, it is possible to increase scalability by scaling out (adding more machines) while maintaining a single endpoint. Additionally, availability is enhanced, since the load is split between multiple instances.

GCP offers multiple types of load balancing, the users has to decide which fits best for them. There exist the two primary types *HTTP(S) load balancing* and *Network load balancing*. To keep the latency as low as possible, the users can configure the HTTP(S) load balancer to work on a cross-region level. The load balancer automatically redirects the requests to the servers of the closest region. If that server is highly utilized, the request is redirected to the next closest region. Another possible HTTP(S) load balancer does not distribute the traffic by proximity but by the requested content type. The content-based load balancer redirects the requests according to its URL. Besides HTTP(S) load balancing, Network load balancing can be used to distribute the traffic between multiple instances. This type of load balancer can be used if the communication with the services is based on other protocols than HTTP(S). The drawback of this load balancer is, that it does not support cross-regional distribution without any major effort.

The load balancer uses so-called *Health Checks* in order to gather information about the current states of the servers. The health checker is a decoupled instance working independent of the load balancer. It regularly sends requests to the servers (polling) and waits for the responses. If a response is received, the server is assumed to be *healthy*, so that it can accept new connections that the load balancer will redirect to that server. If no response is received to a sequence of requests (e.g. no response for three consecutive requests), the server is rated as *unhealthy*, thus the load balancer does not pass new connections on that server, albeit existing connections to that server will be further processed. The health checker continues to poll those unhealthy instances, as soon as it responds to a number of consecutive checks, it is labeled as *healthy* again. [2]

2.3 Server Management

The main reasons to choose a cloud service are scalability, resilience, lower costs and lower managing effort. However, if a large-scale company like Spotify or Snapchat decides to run their services externally at GCP, Google has to make sure that their customers keep control of them. This can be quite a challenge, because these companies use many different services, it would be exceedingly complex and expensive to manage, monitor and manipulate each of them manually. For example the music streaming platform Spotify was actively running 810 (micro)services back in 2015 [1] and perhaps even more today.

For this reason, Google applied the concept of *instance groups*. In this manner, multiple instances can be pooled into certain groups, which greatly simplifies the management.

Two types of instance groups are available: The user can choose between managed or unmanaged instance groups.

In managed instance groups, completely identical instances are organized together. This has the advantage that they act like a single instance, changes and manipulations are applied to all instances in that group, so that the complexity dramatically decreases. By defining an instance template, every instance is created with the same properties and behaviors. As soon as an instance stops, crashes or is deleted, it is recreated instantly and automatically.

In addition to the aforementioned advantages, managed instance groups provide two more main benefits:

2.3.1 Health Checks: Health checks for managed instance groups work similar to the health checks for load balancing. In this way, the states of the instances within the group can be checked and measures can be taken to ensure that they are running properly. Although the health checker behaves the same way compared to load balancing, it is important to configure it more precise and detailed. Since the health checker is responsible for initiating the recreation of instances when they are unhealthy (after a crash, timeout, ...), this should only be performed if the instance cannot be recovered. This is because data saved within the instances will be lost after deleting and recreating the instance, so the data should generally be persisted in central places like *Google Cloud SQL* or *Google Cloud Storage* (see section 2.4).

2.3.2 Autoscaling: As mentioned earlier, scalability, low costs and low management effort are essential for applications. However, it can be quite hard to choose the optimal number of machines to run a specific service. If the operator undercalculates the amount of instances, the applications will not run smoothly or even fail, since the few instances are overloaded. If the operator overestimates the number of required instances on the other hand, he pays too much without any benefit, since the same work could be done by less machines, and he also increases the management effort unnecessarily. But even if he may have chosen the right number fitting his needs, the load will probably drop or increase in the future, requiring further adjustments. Especially when dealing with highly irregular traffic, this procedure can quickly become annoying and troublesome.

To help their customers by automatically choosing an appropriate number of instances within a managed instance group, GCP highly decreases managing effort and therefore lowers the costs. This feature is called *autoscaling*. To determine if additional instances should be added or removed, the autoscaler gathers information about the current utilization and takes a decision based on this data. The user can define a target value; if this value exceeded, one or more instances are added. If it is undershot, one or more instances are removed.

GCP provides multiple policies that define how the utilization of the current instances is measured.

The most straightforward policy is the average CPU utilization. During setup, the user defines the target value as the maximum permitted average CPU utilization. During service the autoscaler regularly polls all active instances of the instance group, requests their CPU utilization and averages all values.

Another policy is scaling based on *load balancing serving capacity*. This approach assumes that multiple instance groups are created and served via backend services. The backend services act as load balancer for those instance groups and splits traffic between them. The user can define a threshold value of either a maximum CPU utilization of the backend or a maximum amount of requests per second (RPS) handled by each instance group.

Another possible autoscaling policy is based on *Stackdriver Monitoring* metrics. Stackdriver[1] is a monitoring, logging and diagnostics service by Google. The user can either choose standard metrics or define custom metrics in order to gain information about the current utilization. The specific metrics can be chosen by the user, the only requirement is that the metric has to reflect the utilization of a single instance and adding or removing instances from the instance group directly affects that metric's value proportional.

If the user decides to make use of the autoscaling service, he has to select exactly one of the three strategies. Since the autoscaling feature only works for managed instance groups, it must be defined in the respective instance template.

Besides the managed instance groups, GCP offers unmanaged instance groups, too. Instances of this group do not necessarily have to be identical, the group can contain instances of all types. Although this type of instance groups provides a higher level of flexibility, it is missing most of the features supported by managed instance groups, e.g. autoscaling, instance templates, etc.

2.4 Data Persistence

GCP was built with the focus on independent regions to ensure a high level of resilience (see section 2.1). They defined multiple regions with each of them containing one or more zones. Therefore, resources hosted with GCP can be classified in three categories: Zonal resources can only be used by other resources in the same zone. There is no way to access these resources from other zones, not even from other zones within the same region. Analogous to that there are regional resources, too. They are available to all other resources in the same region, so they can be seen as cross-zonal resources. Although this hierarchical structure helps to logically organize the resources (e.g. instances are working per zone, there would be no reason to classify them as regional resource), some assets should be accessible for every other resource independent of its actual zone and region. This is covered by the global resources, for example images used to boot a new instance. However, these resources cannot be hosted on any of the customer's instances. As described earlier, instances are zonal resources that are only accessible within the same zone. For this reason, GCP offers storage and database services to replicate the data and make it globally available. In particular, this includes *Google Cloud Storage* for object/file storage and *Google Cloud SQL* for a global, fully-managed MySQL database.

3 POSSIBLE APPROACHES

In the previous section the fundamental functionality of Google Cloud Platform was described, enlarged upon the four core components *region-based hosting, load balancing, server management* and *data persistence*. Although the underlying principles are no novelty or innovations by Google, they provide their customers a large amount of configuration parameters that the users probably never heard of, do not fully understand their effects or simply rely on the default configuration in order to lower the effort of setting up the system. In many cases it may actually be sufficient to run the services with the predefined values since Google already optimized them as much as possible. Nevertheless, specifically for load-intensive and highly utilized systems, a detailed and customized configuration is inevitable.

But even if the system is perfectly configured, it is not enough to rely on this setup. System loads and requirements can change, in extreme cases within minutes or even seconds, so that these parameters have to be adjusted continuously. To detect and identify system failures, low capacities or other misconfigurations with impact on the flawless operation of the system in realtime, the user can leverage a monitoring tool in order to supervise its services. However, these tools cannot repair or fix the system with satisfying results, since it had to know about all possible configuration parameters provided by GCP.

To this end, the paper therefore suggests some new approaches based on machine learning, building on the four components described in section 2. On the one hand, these theoretical approaches shall support the customers of GCP by simplifying the management and reducing or even eliminating the permanent need for adjustments. On the other hand, they shall also better meet important requirements like scalability and resilience.

[1]see https://www.stackdriver.com

3.1 Global replication suggestion based on zonal hosting

Although region-based hosting already provides a high level of resilience and availability, the concept can be improved in various aspects. First of all, it is always important to host the services as close to the endusers as possible. This can decrease latency dramatically, improving the user experience, which again increases the probability of returning users.

Google offers the opportunity to deploy the services in different zones and different regions. If the operator of the service exactly knows where the majority of the traffic is coming from and that it is very unlikely to change in future, there is no need to dynamically change the configuration. Under these conditions it is sufficient to deploy the application once in the according regions. But this scenario is not always the case. Especially large-scale companies often have users all across the globe, while the demand varies considerably in certain ares. One example would be the gradual roll-out of a product to new markets. Until April 2013, Spotify was available in 20 countries, primary in Central Europe and North America. As of April 2013, the service was available in 8 more countries [7], so that the markets in Latin America, Asia and Eastern Europe suddenly grew strongly. This means that the operator had to deploy all of their services in new regions, create new instance groups manually (since they are zone specific resources) and configure them. Although this scenario does not happen that often, it can be shown that the process of setting up existing services in new zones can be cumbersome.

The solution for this problem could be an algorithm that is learning from all requests processed by GCP. Since supervised machine learning methods are better suited for this case than unsupervised techniques, enough data is required to train the algorithm. During training phase, the algorithm is fed with detailed metadata about the requests, for example the origin country or area, a timestamp, and other metadata like the destination service, the average RPS processed by that service (in order to estimate if it is a large or small service), the zones the service is currently deployed in, etc. That way, the algorithm could be trained in order to predict whether the service should additionally be deployed in other zones respectively regions that are not yet covered or if the deployment in some zones does not offer any added value and can therefore be removed. This has the advantage, that the whole process of supervising the system, deciding if the service should be deployed in other zones and the actual setup of that service is completely automated and does not require any form of user interaction. But the even greater benefit of this approach would be the self-adjusting characteristic: The algorithm is not rigid, since it constantly receives new input data that can be constantly incorporated to improve the performance of that algorithm. Thus it can react to temporal or regional effects (as described in the example above), which leads to a highly adaptable and flexible method of region-based hosting.

3.2 Adaptive load balancing

As soon as services are deployed in various regions and zones, it is reasonable to install a central load balancer to distribute the traffic as described above. Based on HTTP(S) load balancing, GCP offers the two types, the cross-region and the content-based load balancer. Although using a content-based solution makes the configuration easier, since all assets of the same type are grouped together, it is not the smartest balancing strategy. Assuming Spotify would adopt content-based load balancing: All static files like HTML or CSS files are served via one instance group, while the main content, the audio streams, are served by another instance group. Since the audio-streaming part of this setup produces much more overhead than just serving simple, small textfiles, this architecture is not really balanced although managed by a load balancer. Therefore, cross-region load balancers represent better approaches. Moreover, similar to the region-based hosting, it provides lower latencies, because all requests are served handled by the closest instance as possible, provided that it has enough capacity.

Under some circumstances, however, the load distribution policies may not be sufficient. The health checker provided by GCP is very minimalistic and basic and is working without any sophisticated algorithms in order to determine the health of the instances. It just polls the instances and labels them as *unhealthy* if no responses are returned. Thus the instances are classified binary, either declaring them healthy or unhealthy. A possible extension for this load balancing system could be an algorithm, that not only considers low latency, like in the case of cross-region load balancers, or high availability provided by health checkers, but also further aspects to increase the user experience and removing configuration barriers, while maintaining a high level of load balancing. The aim of that algorithm would be to predict the best possible server/instance for a user to request, since the closest server does not always need to be the server with the lowest latency and healthy instances do not automatically provide high capacities.

The following list gives an overview of possible metrics that could be used as input data in order to train the algorithm:

- **Instance-specific metrics:** This feature describes the general information about all available instances. While the traditional health checker just examines the binary state (responding / not responding), more detailed data would probably help to assess their current conditions. To give an example, the health checker could collect CPU utilization, memory usage, running processes etc. The more detailed and specific the data, the better the final prediction.
- **Accessibility/Availability:** Sometimes, not all instances are accessible / available at the same time. For example, GCP supports rolling updates, so that the instances are updated one after the other. If that update contains errors, it can be stopped halfway through, so that these affected instances are not available during rollback or recovering. In another

case, a whole instance group may be misconfigured, so that the instances cannot be recreated after a crash and therefore stay offline until the operator fixes the issue manually. This knowledge can be included in the training of the algorithm by extending the responsibilities of the health checker. It could regularly poll the instances, just as it does now, but instead of considering just the current state, it could raise statistics about each instance's availability. As a consequence, if one instance shows a below-average availability, it may not be considered as the best server, although it is very close to the user.

- **Responsibilities:** While some instance are created task-specific in order to run one single service, other instances may be designed to serve multiple purposes. This means that the capacity of the latter is not only depending on the requested service itself, but also on other tasks processed by the same instance. If the instance is confronted with multiple other heavy-duty tasks in parallel, it can provide less resources / capacity for the user-requested service as a logical consequence and therefore performs worse than single-purpose instances.
- **Processing time:** GCP provides a high level of flexibility with regard to custom architectural decisions. As a result, instances running the same services may not be designed the same way, which leads to different response times. For example, some instances may persist their data in a local database, while other instances use the centralized *Google Cloud SQL* service. Although the Google Cloud SQL service provides multiple benefits, it can be slower while processing the data due to additional remote network overhead, so that the local database can – in contrast to the remote DB – perform better. The load balancing algorithm may therefore take processing times, regardless of the distance or geographic location of the user, into account.
- **Additional factors:** Besides the features mentioned above, the algorithm may include more decisive and crucial factors in order to predict the best available server. For instance, it can consider the current daytime of the user as well as the servers in their respective regions. At night, most servers are less utilized and can process requests faster for this reason.

In this case an unsupervised algorithm should be chosen since the target values are not known during training. With the help of machine learning methods the algorithm receives the previously mentioned data and then learns a classification. After training phase, the algorithm can be leveraged in order to determine the best server that the user should request.

3.3 Central autoscaler manager

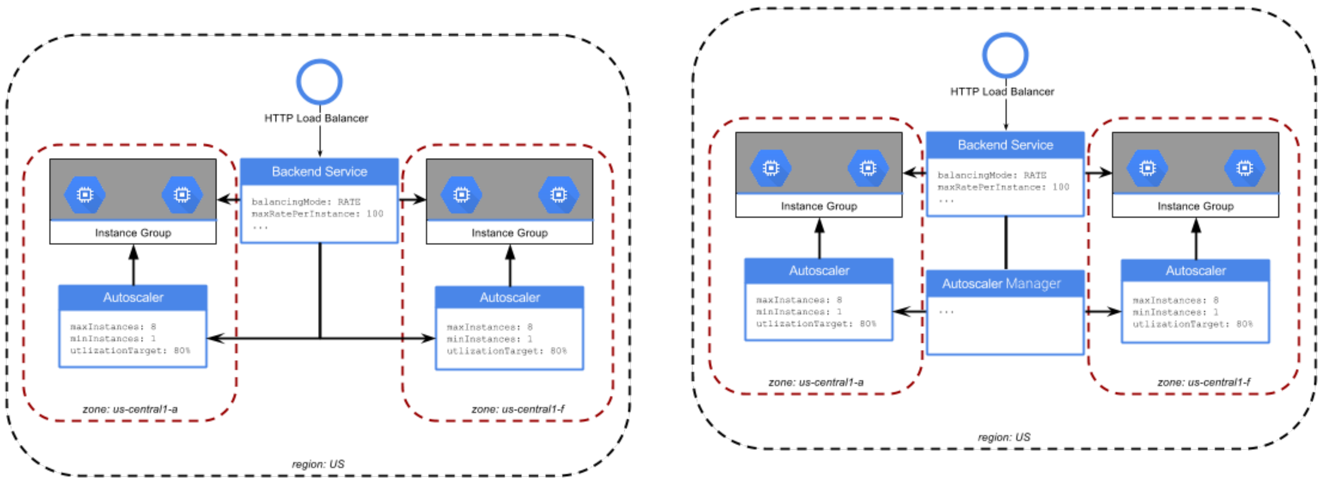
Autoscaling represents one of the most efficient techniques to achieve high scalability with low costs. It is the only system in the described architecture,too that automatically adjusts and changes the setup according to the current demand. Nevertheless, it also reveals certain weaknesses:

First of all, autoscalers are generally zonal resources. That means that they can only manage those instances which are within the same zone of one single region. However, this limitation can have effects on the complete setup: Since most operators tend to deploy their services in multiple areas and zones, they have no central control over one single entity, but have to manage and configure multiple autoscalers at the same time. One solution could be to reconsider the region- and zone-based hosting, so that there are only globally available instances without any hierarchies, each instance could communicate with each other instance. Although this concept of *global managed instance groups* would solve the problem of independent and separated autoscalers, resilience, clarity and overall manageability would decrease.

A better approach could be implemented by adding another centralized component on top of the current architecture. *Figure 1a* depicts the state-of-the-art system design by Google. It can be seen that the more zones the service is deployed to, the more autoscaler instances are necessary. Figure *1b* suggests an extension to this architecture by adding an additional component, called *Autoscaler Manager.* It represents a global-level service, which means that there is only one autoscaler manager per application (with increasing load or resilience concerns, there may probably work more than one instance in parallel, but the important point is that the number of autoscaler managers does not increase with the number of deployed zones).

This new concept has multiple advantages: Firstly, especially for large-scale companies the configuration effort dramatically decreases, since rules, configuration parameters, default values etc. can be deployed once to the autoscaler manger, which passes them to all registered zonal autoscalers. Secondly, the centralized control can help to build more stable systems. For instance, the autoscaler for one zone fails, while the traffic is heavily increasing. Based on the conventional architecture, the load balancer would still redirect the requests to the instances in that zone, because they are healthy until the whole capacity is exhausted. However, the additional autoscaler manager could detect the failure of each autoscaler and take appropriate measures in a timely manner. One option would be to take control and responsibility for that zone itself until the zonal autoscaler is available again. Another possibility consists in influencing the load balancer in order to protect that specific zone during the recovery of the autoscaler.

The greatest potential of this new approach can be achieved by another learning algorithm: Two use cases provide additional resilience as well as availability and scalability. One scenario is for example intelligent failure detection and prevention. Sometimes it can happen that an instance crashes and has to be recreated therefore. During recovery, this instance is not available, so that another instance is started in the meantime instead of waiting for it to reboot. If the whole system is very error-prone, a certain percentage of



```
balancingMode: RATE
maxRatePerInstance: 100
...
```

```
maxInstances: 8
minInstances: 1
utlizationTarget: 80%
```

(a) Conventional autoscaler architecture
Source: [3]

(b) Extended autoscaler architecture with an additional new autoscaler manager
Source: [3] (adapted)

Figure 1: Traditional and new autoscaler architecture

all instances is not available, which lowers the systems performance. The autoscaler manager may thus be trained so that it can detect future failures and create a new instance before it even fails. Another use case would be to predict the need for additional instances before they are actually required. Problems, that are often faced by smaller companies or startups, are server outages or missing capacities during suddenly increasing traffic. After TV appearances, commercials or other noticeable publicity, a large amount of users visits the website of that product/company, so that the utilization increases heavily within seconds. Although the autoscaler recognizes the demand for more instances, they have to be booted at first, which costs valuable time. However, a well trained algorithm could detect these patterns of an upcoming run on the servers and create new instances before they are needed. An additional benefit would be to not only predict the increasing demand, but also to calculate how many additional instances are enough to satisfy the needs.

Furthermore, the autoscaler manager could profit from such a learning algorithm by including more sophisticated metrics than just CPU usage or RPS as described in section 2.3.2. Additional information could be similar to those described for the load balancing approach.

4 EVALUATION

The previous section described several use cases and concepts in order to increase resilience and scalability while reducing costs and management effort for the users. These approaches are just theoretical ideas to show the opportunities and possibilities that more data-driven architectures in the context of Google Cloud Platform may provide.

The first proposal about region-based hosting including optimized deployment strategies reveals no major benefit for most of the operators running their services on GCP. Although large-scale companies distribute their application across several regions and even more zones, this is usually just a one-time job and therefore does not involve any major effort. The approach also introduced the idea of suggesting new zones to deploy primarily based on the incoming requests. Larger companies carefully examine and analyse the traffic anyway as a basis for their business decisions, so they usually do not require any supporting suggestions about the zones they should additionally deploy. Smaller companies and private customers, on the other hand, have no need to distribute their deployments across the globe, so it suffices to run the services in only two zones for resilience reasons. In summary, the global replication suggestion approach based on zonal hosting would be a-nice-have feature, but it would have little impact on existing deployment / replication strategies and behaviors.

The adaptive load balancer would probably more useful. The traditional load balancing service provided by GCP is based on a very simple health checker that just verifies whether the instances respond to the requests. They offer no opportunity to evaluate more advanced and meaningful metrics. The presented idea however explained how a self-adjusting and managing load balancer could work, that predicts the most suitable / best server for each user per session. Based on information about the system-state, historical data about metrics like availability etc. could help to learn an optimized strategy in order to distribute the traffic better. Multiple constraints are taken into account, for example the configuration effort for the customers would strongly decrease, since the algorithm learns the best strategy and finds the best parameters itself during training, so that no further adjustments should be necessary. For large-scale companies this could be really helpful, because load balancing is extremely important and a core-feature of scalable and stable applications. Thus a well trained algorithm could increase the overall performance of the system by attuning to the traffic and the available zones. Similarly to global replication suggestion, this approach is of small value for smaller customers, since traditional load balancing – if even necessary – is completely sufficient, the proposed extension would probably produce too much overhead.

The third concept, however, could provide benefits for every user segment, independent of their size. It fulfills various requirements, while being easy to setup and to maintain. It solves the problem of the limitations of zonal resources, which are independent and isolated from all other zones and regions. This means that all autoscalers can communicate via one single interface. The failure and load prediction features should contribute to a highly available, performant and resilient system.

5 CONCLUSION

These presented theoretical ideas show that cloud computing providers like Google Cloud Platform have many opportunities and use cases for *intelligent*, data-driven solutions. They can have advantages for both the users as well as the providers. The main benefit of these examples is the simple and intuitive usage. The users do not necessarily need extensive expert knowledge about cloud computing in general and its various configurations. They do not have to worry about possible wrong configurations or other specific details. Another advantage is the decision-making reliability: Since all request to any GCP-driven service can be used to train and improve the algorithms, a lot of training data is available. This means that the decisions made by those algorithms become more and more reliable and accurate. Therefore the chance of wrong configurations and decisions is lowered, the systems become more performant. Furthermore, these algorithms are capable of adapting to long- and midterm changes. By using these machine learning techniques, the configuration parameters do not have to be set fixed, but they can be adjusted automatically when the underlaying traffic patterns change for example. This flexibility helps the users to run perfectly fitted systems with minimal effort. The data-driven approaches are very scalable, too. Most machine learning based algorithms are designed in a way that they are capable of processing large datasets very fast. As soon as the algorithms are trained and ready for production, the calculation of the results comes down to inserting the parameters in an equation, which is almost as fast as a simple table lookup.

However, the presented ideas reveal some disadvantages, too: Decisions made by one of the algorithms are not always completely transparent and comprehensible. The underlaying algorithms are quite complex, so that the users may use the supporting and helping functionality, but cannot understand the way the algorithms make the decisions. This can lead to irritations and insecurities of the users again. Another drawback for the cloud platform providers is the additional overhead produced by learning, testing and applying the algorithms for the whole system. In the traditional architecture the users defined the parameters for the services, they were fixed and rarely adjusted. In the new approach the system constantly checks if any changes should be applied, probably with every request. This generates a big overhead and can even impact the system's performance.

In summary, these new data-driven approaches provide a wide range of new use cases and applications scenarios for the users and the providers, but they are quite hard to put into practice in order to build a perfectly working system with reliable predictions. Although Google is at the forefront of machine learning and artificial intelligence research, it would require major changes of the core architecture in order to apply those new theoretical ideas.

REFERENCES

[1] Kevin Goldsmith. 2015. Microservices @ Spotify. (2015). https://gotocon.com/dl/goto-berlin-2015/slides/KevinGoldsmith_MicroservicesSpotify.pdf

[2] Google. 2017. Adding Health Checks. (2017). https://cloud.google.com/compute/docs/load-balancing/health-checks

[3] Google. 2017. Building Scalable and Resilient Web Applications on Google Cloud Platform. (2017). https://cloud.google.com/solutions/scalable-and-resilient-apps

[4] Google. 2017. Regions and Zones. (2017). https://cloud.google.com/compute/docs/regions-zones/regions-zones

[5] Google. 2017. Why Google Cloud Platform? (2017). https://cloud.google.com/why-google/

[6] Synergy Research Group. 2016. Amazon Leads; Microsoft, IBM & Google Chase; Others Trail. (2016). https://www.srgresearch.com/articles/amazon-leads-microsoft-ibm-google-chase-others-trail

[7] Spotify Press. 2013. Spotify launches in 8 new markets today. (2013). https://press.spotify.com/fi/2013/04/16/spotify-launches-in-8-new-markets-today/

[8] Sandvine. 2015. Global Internet Phenomena - Latin America & North America. (2015). https://www.sandvine.com/downloads/general/global-internet-phenomena/2015/global-internet-phenomena-report-latin-america-and-north-america.pdf

[9] Matt Tavis. 2010. Architecting in the Cloud. (2010). https://s3.amazonaws.com/aws001/trailhead/TrailHead_ArchitectingInTheCloud.pdf