GlobalLogic®

Optimizing Your

Microservices Architecture

With Auto-Scaling

Abstract

In this white paper, GlobalLogic takes a closer look at the method of elastic auto-scaling used in microservices architecture of your cloud operations. By examining the different auto-scaling methods involved in event-centric microservices architecture, we provide insights to the reader on their use for optimized processes. This white paper serves as a guide to any organization using cloud microservices architecture or planning on migrating to it soon.

Contents

Introduction

As cloud computing takes the main stage, various sectors ensure to incorporate this technology to continue improving their digital operations and deliver optimal results. With the stakeholders focusing on building effective applications, the use of microservices proved to be a huge step-up for their business. Unlike the traditional monolithic architecture that composed of a bulky application, microservices architecture provides a single application with smaller services which communicate with each other. It helps teams code better, use other components flexibly, and optimize resource and cost distribution in applications.

In a microservices architecture, each service faces a certain load. The difference in load requires a different number of instances to be created for each service and due to the dynamic nature of the requests, a real-time load distribution method needs to be deployed to allocate and use the resources optimally. This method is the essence of auto-scaling. It provides a dynamic way of allocating resources based on the load faced by each process in the microservices architecture. However, there are many ways of auto-scaling. With each having a set of advantages and disadvantages of their own, choosing the right scaling method depends on several factors such as traffic factor, backpressure, etc.

As an expert in cloud-based operations, GlobalLogic has worked on numerous cloud-based microservices architecture for leading businesses across the globe. With a dedicated team of professionals having adept knowledge of the subject, we explore the latest technologies and trends in the field of cloud computing and microservices.

Based on our expertise in this domain, we have researched the most popular open-source tools that help you auto-scale your microservices. By assessing these tools and various methods of elastic auto-scaling, we provide a guide to help readers select the best scaling process for their unique project. Along with information on concepts of backpressure and congestion, we have also provided details on distributed queuing systems such as Kafka for readers to get comprehensive insights on dealing with their microservices architecture to improve the project scalability and results.

Enhancing Performance of Event Driven Architecture with Auto-Scaling

Event Driven Architecture (EDA) is used when distributed systems need to be integrated for more efficient communications between them. One of the most practical examples of EDA is ecommerce sites; the distributed systems could be the product catalogue, checkout page and

orders tracking. When a new product is put in the cart it sets off a new event. The event is communicated to the checkout page which now reflects the product and its price. When the purchase is complete, the event is communicated to the orders tracking page and so on.

A variety of heterogeneous systems can be integrated using the EDA approach which makes them easier to scale, for example when there is an unusually high number of shoppers using the site at the same time. The EDA approach to microservices allows a more efficient flow of dynamic data which encourages taking action as soon as the event is generated, for optimized outcomes. On the front end, this means that a purchase is progressing smoothly in real-time.

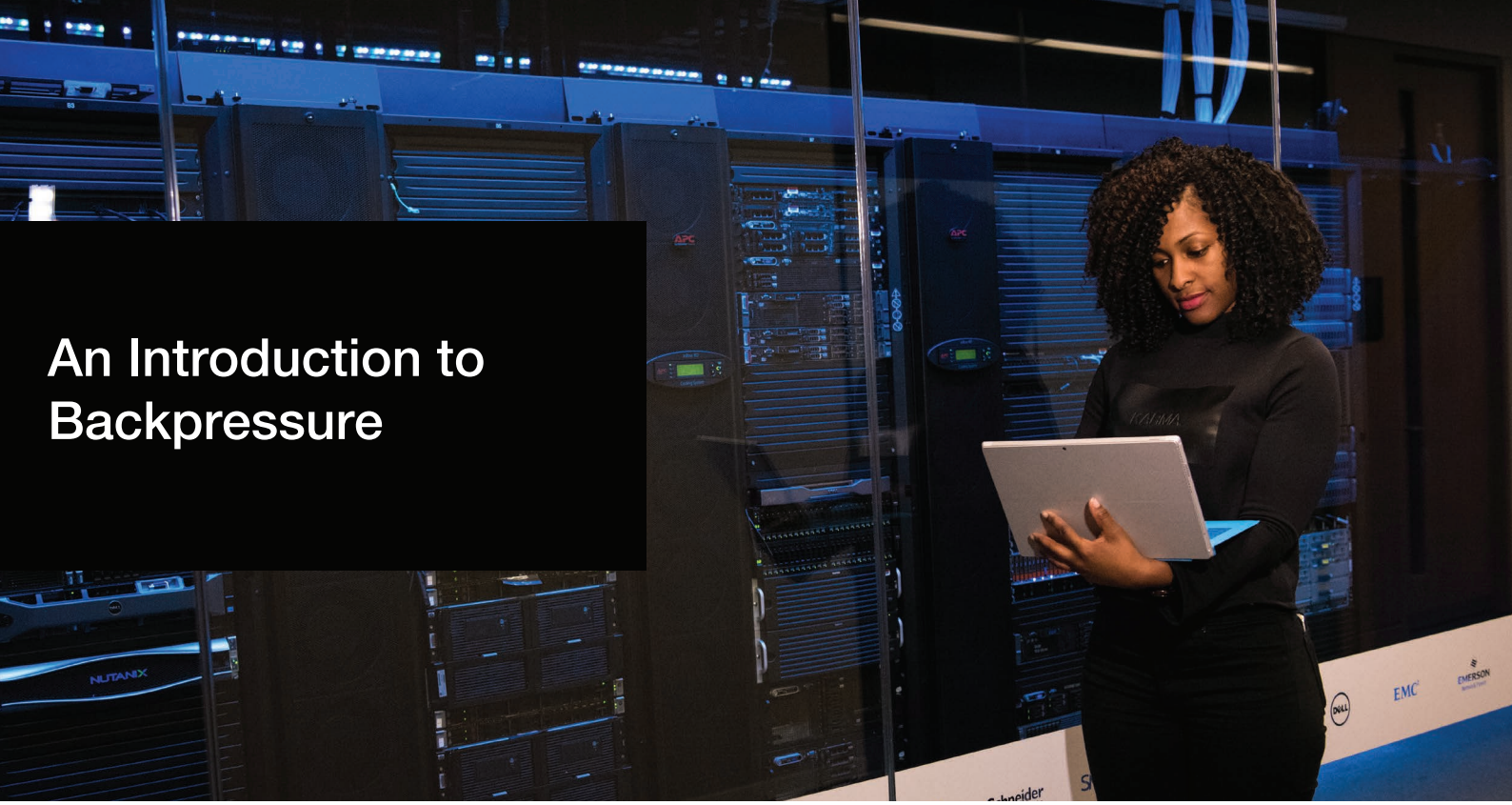
However, the EDA-based solutions with microservices are not without their challenges. One of the major challenges, is dealing with massive event ingestion scenarios. In cases like these, thousands to millions of events need to be processed at the same time while ensuring that that back-end infrastructure is not suffering. Contention, lack of stability, and performance issues can all arise if the microservices are overwhelmed with the volume of incoming events.

This is where the elastic auto-scaling capabilities associated with Kubernetes comes into play. Kubernetes is an open-source platform that helps in the management and discovery of containers of services that make up an application, and facilitates automation. The containers are grouped into logical units which are easy to scale using elastic auto-scaling.

However, having elastic capabilities sometimes is not enough. It is also important to know whether to scale up or down the infrastructure at any given point in time. Even when an instance is idle, it is charged for the time it is powered on. At times like this, the process switches into using the concept of shrinking, which detects the idle nodes and reduces the instances to prevent resources from being expended.

Our focus will be on Kafka, a typically stochastically distributed queuing system. Kafka is a stream-processing software that can not only build data pipelines to get data between systems or applications, but also build real-time streaming applications that can react or transform based on the data. The open-source software does this by running as a cluster on servers that span multiple datacenters.

To understand all this, we need to look at dynamic queues including concepts like traffic factor and backpressure. In the next chapter we will take a more detailed look at backpressure and congestion, which will help you understand how massive ingestions can be handled properly.



An Introduction to Backpressure

A Pod in a microservice architecture can be defined as the most basic and smallest object that represents your microservice in action. It usually contains one container but can be a group of more than one container as well. In elastic cloud computing, the ability of the system to function, especially under heavy-load scenarios is a crucial factor while increasing the Pod instances. It could be compromised due to extended memory usage of the Pod containers or event accumulation in case of EDA event hub, thereby leading to overall poor performance of the system.

In such scenarios, the system deliberately pushes back the overflowing requests to avoid an overload. This resistance to accept new events or decreased system responsiveness to deliver the desired outcome is termed as backpressure. Since the demand exceeds the capacity of the system to process them, if it is not mitigated in time, it will impact the running processes.

Understanding Backpressure Management Mechanism In Microservices

The preferable solution of developing in-built capacities in individual systems does not work in favor of large-scale service architectures thanks to the dynamic nature of the cloud and containers. To ensure stability and smooth functioning of processes, interactions at the producer-consumer level need to be controlled along with systematic addition of new instances, which can be done using backpressure management mechanisms. Typically, they choose one of the following strategies to handle the excessive demand faced by systems in a backpressure condition:

1. Throttling

To reduce the bandwidth of any system, one of the effective solutions is to reduce the rate of incoming flow of requests from producers. This is usually done to allow time for consumers to process pending requests before restoring its normal situation. Once this is achieved, producers can produce messages at their normal rates.

But on typical distributed queuing systems such as Kafka, the clusters do not have the ability to limit the number of consumers on the system. It is designed in a way that consumers can consume at a fast pace and producers can push large volumes of data quickly. Effective management on such systems can be done by allocating new partitions, compressing the size of the message, etc. Configuration properties of the two consumer classes offered by Kafka, written in Scala and Java can be modified to enhance the performance of the system. For example, the amount of data a server should return for a message of a particular size, the amount of data to be returned per partition, etc.

2. Horizontal Pod Autoscaler (HPA)

If the workload can be scaled, HPA responds to the resource requirements by increasing or decreasing the number of Pods. It ensures there is consistent performance irrespective of the situation, leading to cost-effective qualitative results. Few instances when HPA adds more Pods are when the memory threshold is exceeded, an increased rate of client requests per second is recorded, or while servicing external requests. Each workflow has a different HPA object, which regularly checks the pre-decided threshold of the metrics to accommodate changes at the earliest. While adding new consumers in the consumer groups is easy, it cannot cross the number of partitions. This increased number of consumers improves the throughput of the system and reduces the workload. A balance is created on all individual EC2 (Elastic Compute Cloud) nodes, slowly returning to stable condition. This is the recommended method to increase the number of instances for effectively scaling an EKS (Elastic Kubernetes Service) cluster.

3. Vertical Pod Autoscaler (VPA)

Unlike horizontal scaling, vertical scaling is done to increase the power capabilities of the system. For example, ramping up the storage capabilities of the CPU, RAM, etc. VPA automatically recommends the values for CPU limits, based on the dynamic incoming requests. Hence, this is usually implemented when the messages are processed in batches. Because when the demand increases, the size of the batch increases, pressurising resources like the CPU. In such a situation, VPA scales or enhances resources like the CPU and memory and slowly brings the system back to its stable condition. Since microservices architecture normally deals with stateless processing and not with internal states, vertical scaling might not be the ideal choice for them.

4. Cluster Autoscaler (CA)

While horizontal scaling scales the number of consumers, there is a threshold for each cluster which should not be crossed ideally. Once it is exceeded, efforts to mitigate backpressure will not result in a stable system as it indicates an exhaustion of resources on that particular cluster. During such situations, AWS EK CA increases the number of EC2 instances to the EKS cluster. The number of nods is adjusted as per the capabilities of the Pods to share the workload. The configurations are aligned as per the EC2 auto-scaling group, so that scaling is automatically done at the EC2 level when new Pods cannot be added to the system.

In the next chapter, we will talk more about detecting backpressure in AWS.

Detecting Backpressure in AWS

Amazon Web Services (AWS), is one of the most popular cloud computing platforms available today. The variety of services offered and the reliable support has made AWS a leader in this space. But like any other cloud service, AWS too experiences backpressure when the demand is high. In the earlier chapter, we shed some light on backpressure and its management mechanisms. In this chapter, we examine a few backpressure symptoms in respect to AWS specifically.

Increasing queue dept at Kafka Topics: This is also called the “eager producer, lazy consumer” scenario and is characterized by consumers not being able to process events in a timely manner. This can happen because of increased input throughput from the producer end, or due to some other existing performance issues.

Latency at event execution: This happens when events are getting pulled from the topic but the actual processing speed is not able to keep up at an equal pace creating a distortion in the flow. Latency can have many other causes which include issues with resources, locking conditions, bad coding, and even contention or bottlenecks at some back-end resource like database connection.

Contention at bare-metal resource level: This is caused by CPU usage spiking to 90-100% or physical memory usage spiking to 95-99%. Contention also happens when disk I/O transfer rates spike towards the maximum admitted rate. These are just some the typical scenarios, however, there could be other causes like competing virtual machines or an issue with a connected physical resource.

Traffic Factor Ratio

Traffic Factor Ratio (TFR) is defined as the coefficient between the number of events that arrive per unit of time and the number of events that are consumed per unit of time. The TFR is useful in determining the state of event-driven-led processing, specifically in the context of stochastically distributed queueing systems like Kafka. We will be talking more about the Traffic Factor Ratio in the next chapter.

In a typical steady system, the traffic ratio should be below a value of 1. If the value is higher than 1, the system is said to be in non-steady state, and we are in the presence of an ‘eager producer, lazy consumer’ scenario. When this indicator of backpressure is evident, it is a clear sign that a further Pod horizontal scaling is required.

Here's how symptoms of backpressure can be detected and solved in AWS:

At the Topic level: If TFR is greater than 1 at the Topic level then the solution is to scale out the consumer group. By expanding the EKS service by adding more Pod instances to a service. The additional Pod instances increase out throughput for the topic. For seamless scaling at the topic level, a TFR custom metric can be created on AWS CloudWatch, and further monitored in order to trigger the scaling of Pods inside the EKS service whenever the condition is true.

At the Pod level: Pod instances health can be monitored by using the K8S (Kubernetes) metrics server in terms of mean CPU and memory usage. When the K8S metrics server detects that the mean CPU/memory usage is going beyond the upper threshold, the Horizontal auto-scaling activates. Processing capacity is increased by scaling up the number of Pods for the duration that they are required. If, however, the mean CPU/memory usage drops below the predefined minimum threshold due to idle condition, the number of Pods can be further reduced, by shrinking the pool. This mechanism helps to maintain a Pod economy, by keeping only the number of Pods that are required at a given time in order to avoid either a CPU peak load or CPU idle state.

At the Cluster level: There might be cases where additional Pod instances are required but scaling cannot proceed further. This could be because of contention on the existing EC2 instances like CPU limits reached, not enough memory, file descriptors exhausted, etc. In such a case, scaling needs to take place at the Cluster level. For this to happen, an EKS Cluster Auto-scaler (CA) needs to detect the conditions and instruct the Auto-Scaling Group (ASG), to increase the cluster size. Similarly, when the CA detects some nodes are idle, it informs the ASG to reduce the cluster size accordingly to economize resources.

At the Microservices level: At a functional level, AWS CloudWatch can also monitor the microservices that are actually consuming events - i.e. an abstract or concrete microservice - in order to measure total average message processing time. Pre-defined thresholds can be set as the upper and lower limits which can trigger specific actions if these thresholds are reached/crossed. Latency in message processing can lead to event accumulation which in turn can lead to TFR > 1. However, at this level auto-scaling might not always be effective due to multiple factors that could cause the scenario. Simply scaling Pod instances in the consumer group or cluster won't make a difference if there is another underlying root cause. At the microservices level the most effective automatic action would be to alert a specific group to take appropriate actions, or even trigger an intelligent mechanism to mitigate the root cause.

Traffic Factor & Its Effect On Microservices

Understanding how a microservice can scale quantitively is determined by the traffic it can handle. This means, the number of queries per second a system can handle. Analysing this is crucial if you require the system to graciously respond to the overwhelming incoming requests without impacting its performance or in the worst-case scenario, crashing. In the context of microservice architectures, this is referred to as traffic factor. To decide on the action to be taken to stabilise the state, first the traffic factor ratio (TRF) is calculated. It is the ratio between the number of incoming events and consumed events at a given time.

If the TRF is below 1, the system is said to be in a steady state. The producer is not generating more requests than the consumer can consume. But if the TRF is above 1, the system is in a non-steady state, where the incoming events from the producer surpasses the events consumed by the consumer, also known as 'eager producer, lazy consumer' scenario.

TFR = Average number of bytes per second received from producers/ Number of bytes per second sent to consumers

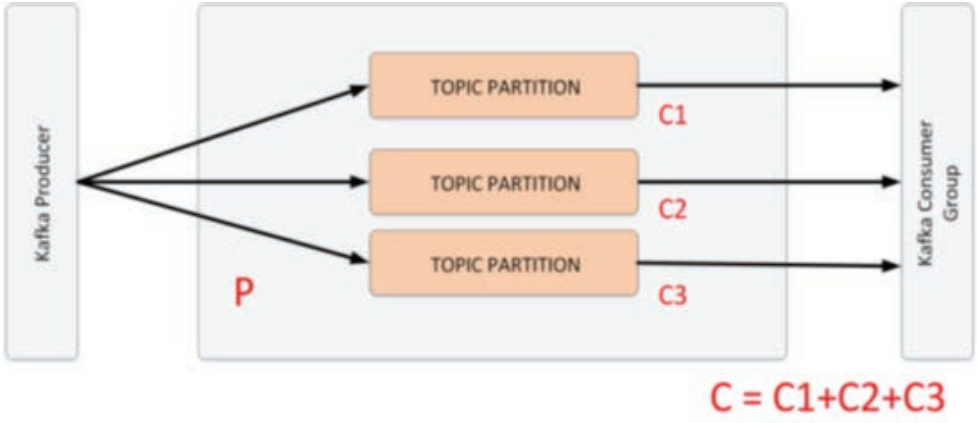
According to this diagram, TFR = P/ (C1+C2+C3)

If TFR > 1 the system is said to be in a non-steady state

If TFR <= 1 the system is said to be in a steady state

Situations where TFR could go above 1 are when the number of customer instances are not increased as per the traffic; number of topic partitions are not scaled; the clusters are not scaled, etc.

To tackle this issue in sophisticated microservice architectures, auto-scaling is done at the topic, Pod, cluster, or even the microservice level when required. Based on the pre-configured threshold value, automatic scaling or shrinking of the instances will take place. If the TFR crosses the pre-decided threshold, it indicates that the consumption rate needs to be increased for a better throughput and lesser workload. If the TFR is below the pre-decided threshold, it indicates that the idle customer instances can be removed to enhance efficiency of the system.



Horizontally Auto-Scaling EKS Pods Based on Traffic Factor

Now that we have discussed Traffic Factor and how to calculate TFR, we can move on to understanding how it is used in horizontal auto-scaling of EKS (Elastic Kubernetes Services) Pods in AWS. EKS was specially developed to be easily integrated with AWS. The service helps to manage scalability of the Kubernetes (K8S) control plan nodes including those involved in EDA-based solutions.

To determine whether to scale out or shrink, the system makes use of AWS MSK (Managed streaming for Apache Kafka) which automatically gathers Apache Kafka metrics and sends them to Amazon CloudWatch. This service is available by default on AWS.

In AWS MSK implementation, backpressure can consist of monitoring the Kafka topic's traffic factor in real-time; once traffic factor is higher than a pre-established threshold, automatic compensation is triggered which should take an action. That action results in scaling out the number of microservice consumer instances by means of an EKS Horizontal Pod Autoscaler (HPA).

Traffic factor for a specific Kafka topic can be calculated using two provided custom metrics as shown below.

The custom metrics are numerically represented and the TFR can be calculated by dividing the former by the latter. A pre-configured upper threshold is set depending on the requirement of the process. In most cases, the value is 1 but it can be higher if required. AWS CloudWatch can be configured to pull these values from the custom metrics. If the value of the TFR is higher than the upper threshold, auto-scaling out of the Kubernetes cluster can be triggered.

To ensure that auto-scaling takes place in the K8S cluster associated with the right consumer microservices Pod, the respective microservice needs to be deployed on to an AWS EKS infrastructure. This can be done by leveraging the Amazon CloudWatch Metrics Adapter for Kubernetes – another service developed for AWS.

Namespace	Metric Name	Dimensions	Description
AWS/Kafka	Bytes in Per Second	Topic	Average number of bytes per second received from producers.
AWS/Kafka	Bytes out Per Second	Topic	Average number of bytes per second sent to consumers.

How it works

When pre-configuring the upper threshold, if the value is set to 2, when producer throughput approaches twice the consumer throughput, scaling is triggered. In this scenario, scaling out happens at the consumer end to increase throughput and compensate for the increase at the producer end.

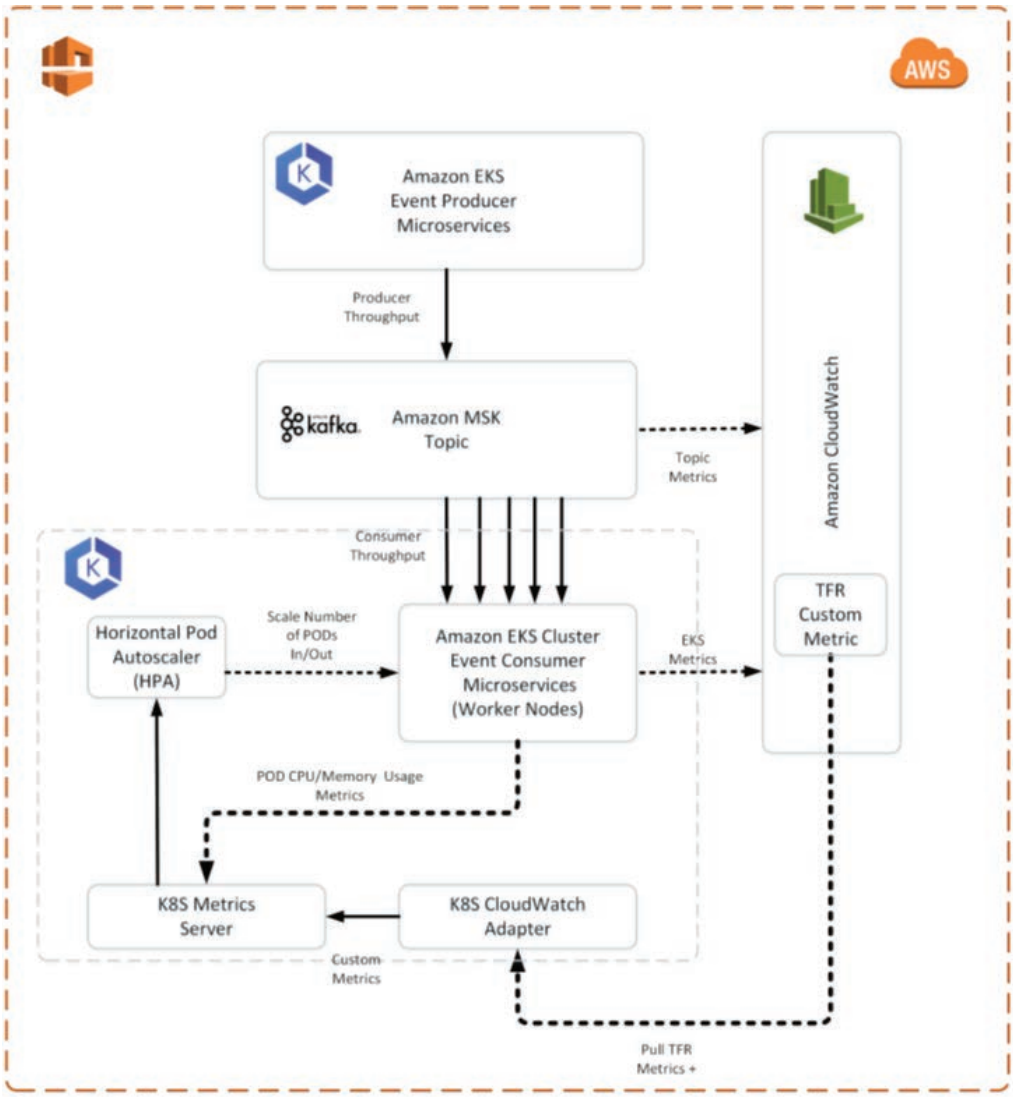
While one set of metrics triggers scaling out, setting a minimum threshold can help shrink the Kubernetes cluster. When the TFR falls below this pre-configured lower threshold, it is an indication of an over-sized consumer group which leads to idle condition. In this case scaling in or shrinking of the cluster will help conserve resources.

Setting both upper and lower threshold for traffic factor allows the dynamic auto-scaling of Kubernetes cluster based on demand. This type of scaling is termed as elastic auto-scaling of consumers.

The K8S Metrics Server also provides another factor that is used by HPA to trigger scaling of Pods. This metric measures the average Pod CPU/memory usage, which is monitored by default OOB (Out Of the Box) using the Pod CPU/memory Metrics. Once again, pre-established upper and lower thresholds can be set for the average CPU/memory usage. If the metrics show a value near the upper threshold, scaling out is triggered and the level of Pod parallelism (the number of Pods that can run in parallel and can coordinate among themselves), is increased. If it drops to the lower threshold, scaling in takes place and Pod instances are shrunk.

Backpressure Compensation Mechanism

While the whole system might appear to be complicated, AWS has developed a user-friendly platform in which implementation of horizontal auto-scaling of ESK Pods is a simplified process. The benefits of seamless processing and resource conservation make the implementation not only worth it but also necessary.



Representation of backpressure compensation mechanism for Pod autoscaling in an event-driven-led architecture with producer/consumer microservices.

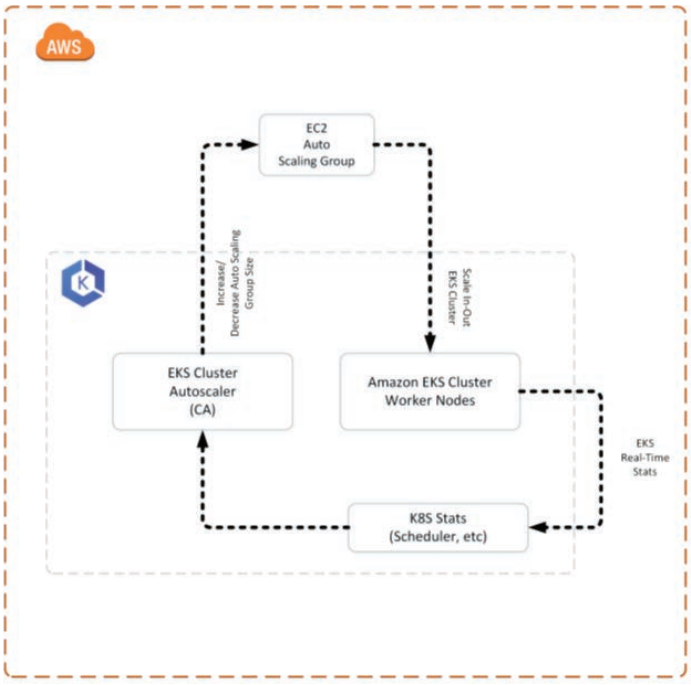
Horizontal Auto-scaling EKS Clusters with Cluster Autoscaler (CA)

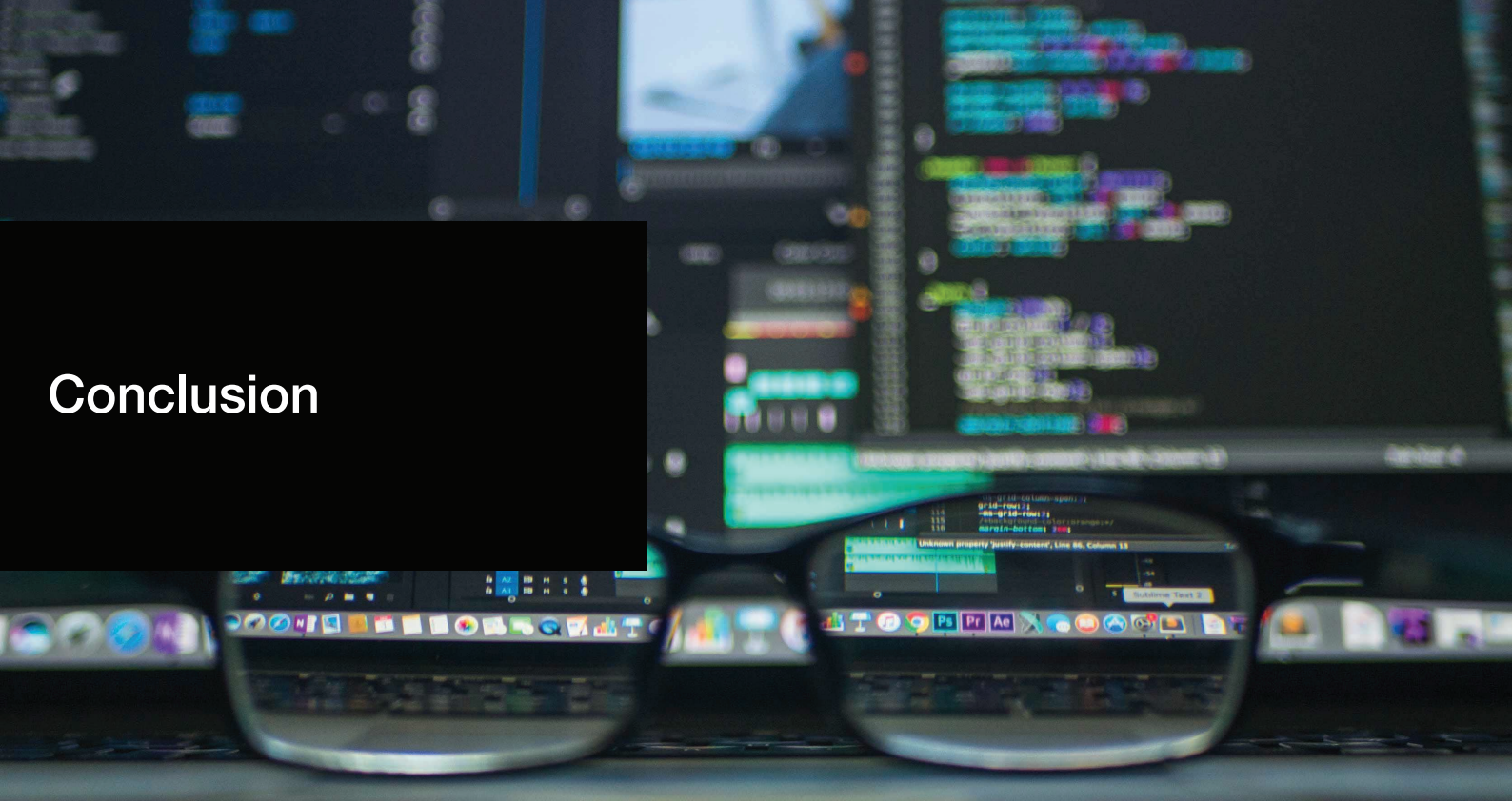
When deploying microservices, the Elastic Kubernetes Services (EKS) is one of the most preferred Kubernetes services. As the number of requests is never fixed, the need for allocating the right size of EKS cluster takes high priority. This issue of traffic impacts the overall application, however the EKS has a way to deal with this. The Cluster Autoscaler plays an important part in this. Based on the scheduling statistics of Kubernetes(K8S) which factor in various indicators such as number of Pod instantiations, the CA optimises the cluster size. When HPA decides to scale-out the number of Pod instances, the new Pods begin to be commissioned. While this is the ideal approach, if the process continues to increase the number of Pods, it can result in failure as there will be no EC2 instance ready for the new Pod, thereby causing the exhaustion of the EKS cluster.

Auto-scaling based on K8S scheduling stats

This results in wastage of resources, and since the process keeps recurring, the overall loss of resources can be high. In scenarios like this, CA intervenes to help the architecture by expanding the cluster and adding a new EC2 instance to the EKS cluster so as to turn the instantiation of the new Pod into a success. Similarly, when the EC2 instances register small or no-activity, the CA helps in optimizing the cluster. It shrinks the cluster, therefore removing the unused EC2 instance, and follows the process of Pod re-agrupation where the existing Pods are relocated to other nodes.

In a way, CA plays a crucial role in addressing the problems of backpressure and traffic factor. While these issues can be the cause of sudden influx or drop in the service requests, the use of Horizontal Auto-scaling CA has proven to be favourable for microservices.





Conclusion

In this white paper, we looked at the various aspects of elastic auto-scaling and specifically its application to AWS systems. We covered three main areas:

- When to scale: Scaling takes place when a system reaches a non-steady state or when there is any disproportionate activity between the producers and consumers. When backpressure is detected in the system, scaling is triggered.

- How to scale: We also examined the fact that scaling by itself is not enough. For a seamless process, the system needs to know whether to scale out or to scale in and at which end of the microservices architecture scaling needs to occur.

- Where to scale: the other aspect we looked at was the different levels where scaling could occur: Topic level, Pod level, Cluster level, and Microservices level. We delved in detail on how metrics like TFR (Traffic Factor Ratio) are used to detect backpressure/congestion and can be used to identify at which level scaling should occur.

The final bit we cover here, is how to know that the Elastic Auto-Scaling implemented is working for your system. As with any other scenario, to get a result we need to run tests. With auto-scaling, Iterative Performance Testing is highly recommended. It is a testing method which uses real loads coming from real-world peak-load scenarios to evaluate whether the theoretical configuration actually works.

Once the auto-scaling configuration is set, a load-test needs to be done to check if it behaves as it's supposed to. The testing should mimic real-world scenarios and match the same peak-load and no-load conditions. What we want to know is whether scaling out happens as expected when the load is high and scaling in when the load drops. Aside from checking the scaling capabilities of the platform, we also need to check the responsiveness of the system during the load-tests. Any signs of contention, latency, hot spots, bottlenecks, or other generic performance issues will require attention, introducing further small changes on the configuration as needed.

The next step in iterative performance testing after the load-test is a Performance Test. A performance test is carried out by increasing the load on the system over time, while evaluating the results and making any required minor adjustments. The iteration should be repeated whenever necessary to re-check performance until the objective peak load is reached.

About GlobalLogic

GlobalLogic is a leader in digital product engineering. We help our clients design and build innovative products, platforms, and digital experiences for the modern world. By integrating strategic design, complex engineering, and vertical industry expertise—we help our clients imagine what's possible and accelerate their transition into tomorrow's digital businesses.

Headquartered in Silicon Valley, GlobalLogic operates design studios and engineering centers around the world, extending our deep expertise to customers in the communications, automotive, healthcare, technology, media and entertainment, manufacturing, and semiconductor industries.

www.globallogic.com