International Journal of Science and Research (IJSR)
ISSN: 2319-7064
SJIF (2022): 7.942

High Performance and Scalable Microservices Architecture using Kubernetes

Balasirisha J[1], Mounika K[2], Mahim Saxena[3], Kishore SVSRK[4], Ravi Kumar MV[5]

[1]*balasirisha_j[at]nrsc.gov.in*
[2]*mounika_k[at]nrsc.gov.in*
[3]*mahim_saxena[at]nrsc.gov.in*
[4]*kishore_svsr[at]nrsc.gov in*
[5]*ravikumar_mv[at]nrsc.gov.in*

Abstract: ***Microservices have become increasingly popular over the past few years, and it's not quite a surprise that the Microservices application architecture continues to invade software design. Microservices' architecture is emphasized in the industry and has gained prominence for its dynamic and agile qualities in API management and execution of highly defined and discrete tasks. Microservices are more adaptable and less complicated to maintain over time. However, the solutions with Microservices are not without their challenges. One of the significant challenges is dealing with massive event ingestion scenarios. If the microservices are overwhelmed with the volume of incoming events, issues like contention, lack of stability, and performance issues can all arise. This is where the auto - scaling capabilities associated with Kubernetes become usable and effective. Based on the load experienced by the application, auto - scaling allows applications to add or remove computing resources. When the load is high, for the application to keep up with the load, additional resources are to be provisioned. When the load is low, resources are revoked, ensuring that no resources are idle. When implemented correctly, auto - scaling allows applications to use fewer resources while maintaining application performance. Among the most popular open - source tools for enabling microservices with automation, this paper describes the auto - scaling mechanism that delivers applications with better performance. In this paper, the need for the development of a scalable environment along with open - source monitoring and alert management tools that help proactive system management is proposed. This paper serves as a reference for implementing microservices architecture.***

Keywords: Microservices, Kubernetes, Autoscaling, Monitoring, Alert – Management

1. Introduction

The Microservices style of architecture develops complex application software from small, individual applications that communicate with each other using language - independent interfaces (APIs). Microservices break down complex tasks into smaller processes that work independently. Building an application using microservices increases the number of microservice instances during high load and reduces them during low load times.

Micro - services implement the service - oriented architecture model that allows application deployment in highly distributed patterns to provide flexibility, agility, and scale. Cloud - native platforms with containers and serverless deployments help realize the flexibility, agility, and scalability of the microservices architecture efficiently.

Microservices - based applications can be deployed within containers, which are completely virtual operating system environments that provide processes with isolation and dedicated access to underlying hardware resources. It ensures that issues in one microservice can't affect others. Containers, on the other hand, provide a more light weighted way to isolated execution of service instances. It is much more convenient to distribute the load, create highly - available deployments, and manage upgrades while easing development and team management.

However, the solutions with microservices have their own challenges. One of the significant challenges is dealing with massive event ingestion scenarios wherein thousands to millions of requests must be processed simultaneously, ensuring the health of the backend infrastructure. Issues like Contention, lack of stability, and performance issues can arise if the microservices are overwhelmed. This is where Kubernetes can be an effective solution to address these issues. Kubernetes is an open - source container orchestration 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 auto - scaling. It is also essential to know whether to scale up or down the infrastructure at any given time.

2. Literature Survey

Microservices are the most scalable way of developing software. But, a lot matters in choosing the correct way to deploy microservices: (i) processes or containers? (ii) Run on servers or use the cloud? (iii) Is Kubernetes required? When it comes to microservice architecture, there are a lot of options, and it is hard to know which is the best. It is crucial to define a target architecture before beginning to scale microservices; otherwise, the IT landscape may devolve into chaos and exhibit worse properties than the existing monolithic applications.

Microservice applications can run in many ways, with different tradeoffs and cost structures. What works for small applications spanning a few services will likely not suffice for large - scale systems. From simple to complex, here are the five ways of running microservices:

1) Single machine, multiple processes: buy or rent a server and run the microservices as processes.

2) Multiple machines, multiple processes: the obvious next step is adding more servers and distributing the load, offering more scalability and availability.
3) Containers: packaging the microservices inside a container makes it easier to deploy and run along with other services. It's also the first step towards Kubernetes.
4) Orchestrator: orchestrators such as Kubernetes or Nomad are complete platforms designed to run thousands of containers simultaneously.
5) Serverless: run code directly on the cloud.

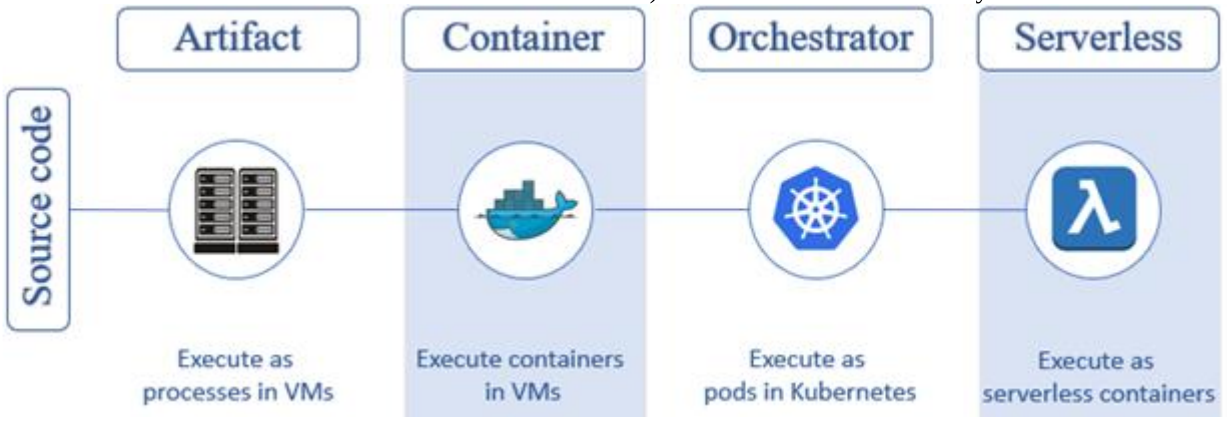
Artifact
Container
Orchestrator
Serverless
Source code
Execute as processes in VMs
Execute containers in VMs
Execute as pods in Kubernetes
Execute as serverless containers

Figure 1: Different ways to run Microservices

Orchestrators are platforms specialized in distributing container workloads over a group of servers. The most well - known orchestrator is Kubernetes.

Orchestrators provide, in addition to container management, extensive network features like routing, security, load balancing, and centralized logs — everything needed to run a microservice application. Kubernetes is the most popular option for organizations making heavy use of containers; it provides the benefits of load balancing, self - healing, and automated rollouts and rollbacks.

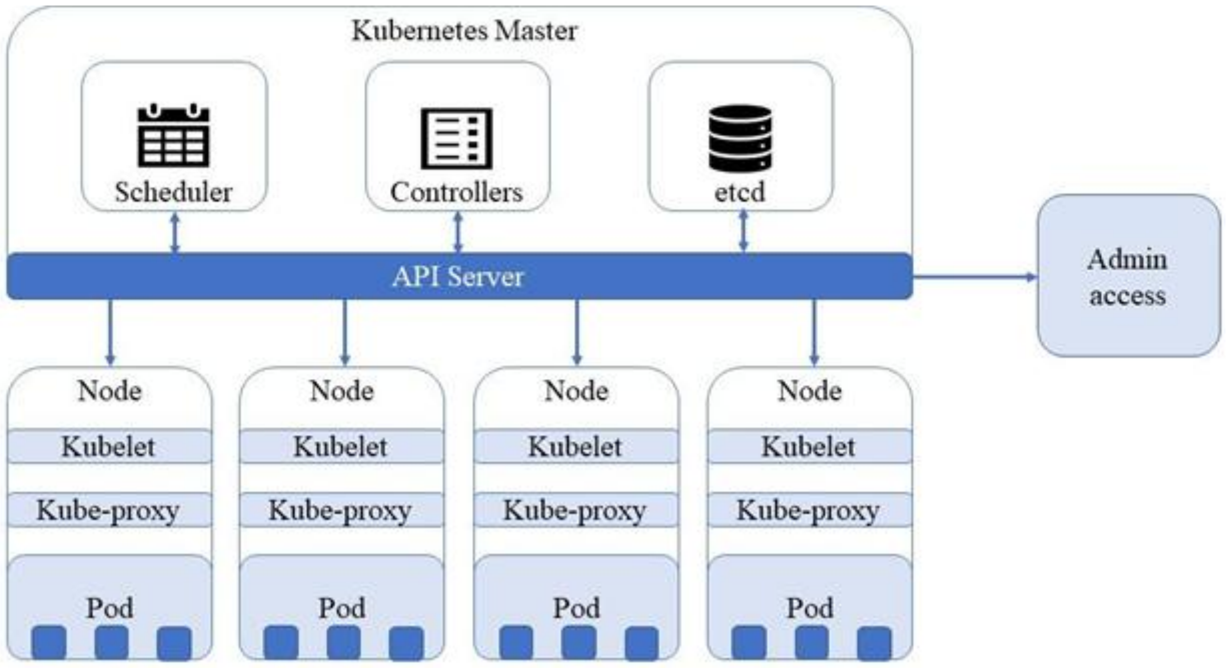

Figure 2: Kubernetes Architecture

Though microservices are deployed on containers, tools to monitor everything happening inside their clusters successfully are required. For this, the application has to expose metrics to track the load, and an agent is used to display the container/pod status on the dashboard. This paper discusses Prometheus – a service that intermittently polls a set of configured targets to fetch their metric values. And Grafana – a pre - configured dashboard that allows visualization of the most important metrics. After all, data in a graphic format is the quickest for us to understand and figure out what to do next. Grafana integrated directly with Prometheus, allows the building of helpful dashboards for microservice applications.

The focus here will be on the Horizontal Pod Autoscaling of Kubernetes, which is used to automatically scale the number of pods on deployments, replica - sets, stateful - sets or a set of them, based on observed usage of CPU, memory, or using custom metrics. To understand the deployment architecture, concepts like System Metrics, Custom Metrics, backpressure, etc. have to be studied and looked into.

System Metrics

The runtime environment generates metrics at all levels of abstraction of the computing node, from hardware metrics to application metrics. An auto - scaling system needs these metrics to measure and analyze application performance and check if SLA violations are occurring or are likely to happen in the future. Different auto - scaling systems look at different metrics, and the simplest auto - scalers look at low - level metrics such as CPU utilization. This type of auto - scaler is then usable for every application as every application generates CPU utilization metrics.

Custom Metrics

Custom metrics, also known as application - specific metrics, define and collects information which the built - in basic Monitoring metrics cannot. Such metrics can be captured using an API provided by a library to instrument the code, which can then be sent to a backend Monitoring application. In this deployment solution, the Arrival rate custom metric (number of records arriving per second) has been used for auto - scaling.

Backpressure

A pod in a microservice architecture can be defined as the smallest object representing a microservice in action. It usually contains one container but can be a group of more than one. 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, 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 system's capacity to process them, if it is not mitigated in time, it will impact the running processes.

Although Kubernetes dramatically simplifies the deployment of containerized applications, its multi - level architecture and multiple abstraction layers (e. g., pods, services) introduces new complexities to the daily tasks of application monitoring. Static monitoring solutions designed for standalone desktop applications are not suitable for Kubernetes because it is a distributed environment where numerous applications and services are spread across multiple cluster nodes. The platform requires monitoring tools that can dynamically capture container events and be tightly integrated with Kubernetes schedulers and controllers.

Capturing, Visualizing and Monitoring Backpressure

System operators need to monitor backpressure signals because they indicate a system is nearing its capacity limit. Leading indicators like response times and queue sizes are essential because they give you time to respond proactively before the system becomes unavailable. Many solutions exist for monitoring Kubernetes clusters, viz., Heapster, Prometheus, and several proprietary Application Performance Management (APM) vendors like Sysdig, Datadog, and Dynatrace. Efficient auto - discovery features and support for containers and Kubernetes make Prometheus a perfect choice for monitoring Kubernetes applications and cluster components. Prometheus is an open - source monitoring toolkit that polls a set of configured targets to fetch metric values intermittently. Prometheus can be configured to scrape metrics from applications, and once it has collected the data, it stores and indexes it in such a way that it can be queried in meaningful ways. Once a curated list of queries is compiled, it can be used to create dashboards using Grafana to render system metrics monitored by Prometheus. Grafana is a multi - platform analytics and visualization web application that acts as a single pane of glass for displaying all of the system's metric data. Prometheus collects rich metrics and provides a powerful querying language; Grafana transforms metrics into meaningful visualizations. Both are compatible with many, if not most, data source types. It is prevalent for DevOps teams to run Grafana on top of Prometheus.

Prometheus provides insights into Kubernetes cluster and containerized applications, but a separate tool is required to communicate when any problem occurs. Prometheus Alertmanager will send notifications to configured notification channels when the Kubernetes cluster meets pre - configured events and metrics rules in the Prometheus server. These rules can include sending critical messages when an unhealthy node is detected or warning when node resource consumption is reaching its limit. Alertmanager mechanism can be configured to send event notifications to a number of channels, including email, Slack, webhook, and other common platforms. Furthermore, Alertmanager offers the ability to group related alerts into a single notification and to suppress alerts triggered by problems for which notices have already been sent.

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 client - server level need to be controlled along with the systematic addition of new instances, which can be done using the below backpressure management.

Horizontal Pod Autoscaler (HPA)

The horizontal pod auto - scaler automatically adds or removes pods based on a particular metric. If the workload can be scaled, HPA responds to the resource requirements by increasing or decreasing the number of Pods. It ensures consistent performance irrespective of the situation, leading to cost - effective, qualitative results. A 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.

3. Methodology

System architecture

The architecture is based on a server - client type in which a server responds to more clients. The server and the client run web microservices, the communication protocol being HTTP and the data format being JSON. This architecture is useful in distributing and dynamically redistributing resources between clients. This architectural model is used to build large, complex, and scalable applications horizontally consisting of small, independent, and decoupled processes communicating with each other using application programming interfaces (API).

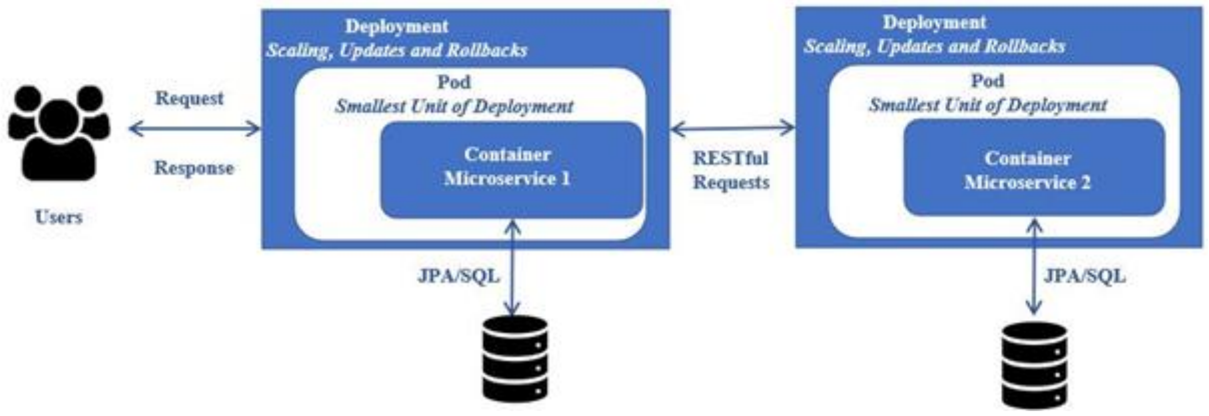

Figure 3: Deployment Architecture

The application architecture was built using the Java Spring Boot framework. The orchestrator service component ensures, on the one hand, the communication between the server and the clients, sending tasks from the server to the clients. On the other hand, it monitors the status of incoming requests.

The proposed system could be split into six different modules:

1) Deploying Microservices on containers and orchestrating them with Kubernetes: To run the containerized workloads and services in a self - contained execution environment, automation is the key process for managing and scheduling the applications. This includes management of Containers lifecycle, viz., provisioning, deployment, scaling, networking, and load balancing.
2) Creating a Custom Metrics API Service in Kubernetes: This allows accessing both system metrics and custom metrics by third - party monitoring adapters and pipelines. With custom metrics and horizontal autoscaler, resource metrics can be tracked and scale thresholds for the containers. Accordingly, the workloads can be adapted.
3) Prometheus – This is a service to pull custom metrics exposed from applications, stores them, and provides an easy way to query them which works seamlessly with Kubernetes. This tool scrapes the metrics, translates them to monitoring format, and pushes them to monitoring API. Metrics from pods/ clients are captured through HTTP requests. Services do not have to continuously send data - it is pulled by Prometheus server.
4) Configuring HPA (Horizontal Pod Autoscaling): HPA monitors the resource requests from the application workloads by querying the metrics. Custom metrics pipeline feeds metrics to HPA in Kubernetes. The target threshold value of HPA definition for the application workloads is compared against the average resource utilization. If the target threshold is reached, then HPA will scale up the resources for deployment to meet higher demands. If the target threshold is below the threshold, it will scale down the deployment.
5) Alert Manager - Alert manager is configured that handles all the alerting mechanisms for Prometheus metrics. Certain rules can be configured on custom metrics using Prometheus, which, if broken, will notify about the problem, and the end user can be notified with emails or other tools. Further, Alert manager offers the ability to group related alerts into a single notification and to suppress alerts triggered by problems for which notifications have already been sent.
6) Grafana - An open - source lightweight dashboard tool is configured that is integrated with Prometheus, which allows building helpful dashboards for running applications. This dashboard provides charts, graphs, and alerts for easier interpretation and understanding.

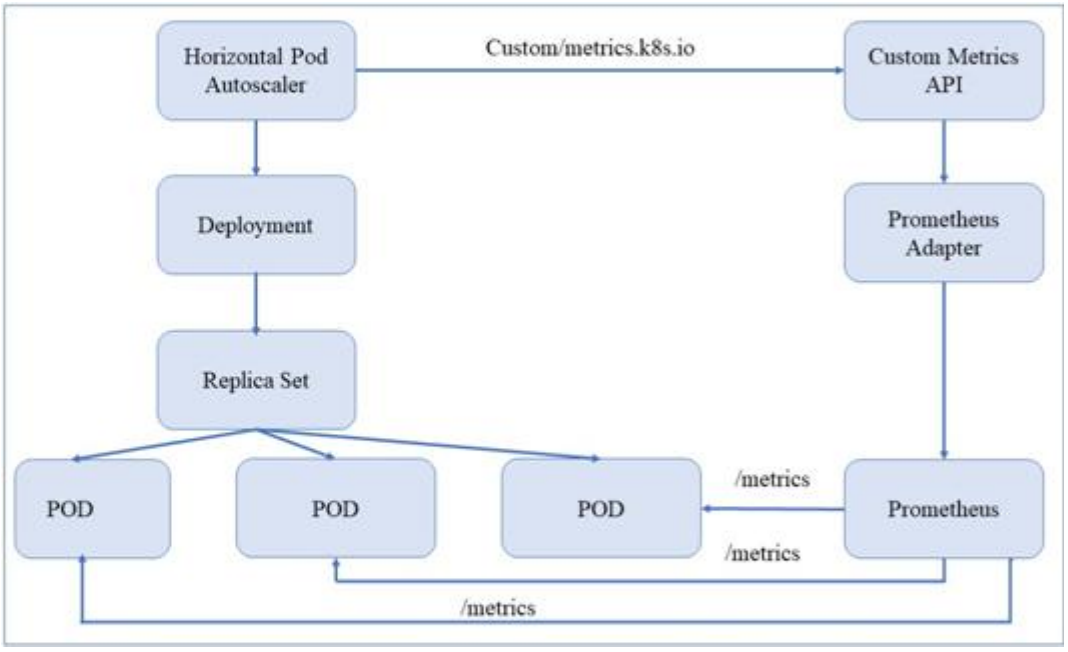

Figure 4: Monitoring System Design

4. Conclusion

This paper discusses various aspects of auto - scaling and, specifically, its application in Microservice architecture. With Kubernetes, the microservice architecture deployment can be automatically managed and scaled. Kubernetes offered features like self - healing, service exposure & load balancing, scalability, and zero downtime were experienced. With microservice architecture, a more manageable, independently deployable, and more reliable setup is established with benefits like development velocity and product quality thereby increasing the modularity aspect. Thus, Kubernetes integrated with microservice architecture provided a more resilient and fault - tolerant framework with high availability and scalability.

5. Future Work

In the current architecture, horizontal pod auto scaling was implemented to handle multiple requests of varying requirements. But the number of requests is never fixed, and the need for allocating the right size of the cluster takes high priority. This issue of traffic impacts the overall application. In such cases, the Cluster Autoscaler (CA) plays an essential part. Based on Kubernetes (K8S) scheduling statistics, which factor in various indicators such as the number of Pod instantiations, the Cluster Autoscaler optimizes the cluster size. When Horizontal Pod Autoscaling 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 fail as there will be no instance ready for the new Pod, thereby causing the exhaustion of the cluster. This project can be extended to use Cluster Autoscaler to provision a new compute unit/node and adds it to the cluster.

References

[1] Backpressure, <https: //flink. apache. org/2021/07/07/backpressure. html>
[2] Kubernetes, <https: //kubernetes. io/>
[3] Architecting Kubernetes clusters, <https: //learnk8s. io/kubernetes - autoscaling - strategies>
[4] Horizontal Pod Autoscaler, <https: //kubernetes. io/docs/tasks/run - application/horizontal - pod - autoscale/>
[5] Kubernetes HPA, <https: //towardsdatascience. com/kubernetes - hpa - with - custom - metrics - from - prometheus - 9ffc201991e>
[6] Prometheus, <https: //prometheus. io/>
[7] Prometheus - adapter, <https: //github. com/kubernetes - sigs/prometheus - adapter>
[8] Grafana On Kubernetes, <https: //devopscube. com/setup - grafana - kubernetes/>
[9] Monitoring A Spring Boot Application, <https: //tomgregory. com/monitoring - a - spring - boot - application - part - 4 - visualisation - and - graphing/>

Author Profile

Balasirisha J, working as Scientist/Engineer - 'SC' in Systems and Infrastructure Solutions Group at National Remote Sensing Centre, ISRO. Email: balasirisha_j[at]nrsc.gov.in

Mounika K, working as Scientist/Engineer - 'SE' in Systems and Infrastructure Solutions Group at National Remote Sensing Centre, ISRO. Email: mounika_k[at]nrsc.gov.in

Mahim Saxena, working as Scientist/Engineer - 'SC' in Systems and Infrastructure Solutions Group at National Remote Sensing Centre, ISRO. Email: mahim_saxena[at]nrsc.gov.in

Kishore SVSRK, working as Group Head, Systems and Infrastructure Solutions Group at National Remote Sensing Centre, ISRO with a working experience of more than 20 years. Email: kishore_svsr[at]nrsc.gov.in

Ravi Kumar MV, working as Deputy Director, Management Systems Area at National Remote Sensing Centre, ISRO with a working experience of more than 25 years. Email: ravikumar_mv[at]nrsc.gov.in