eBook

Scaling and Auto-Scaling Strategies for Cloud-Native Applications

Fiorano®

DISCLAIMER

No AI was used in the production of this work

TABLE OF CONTENTS

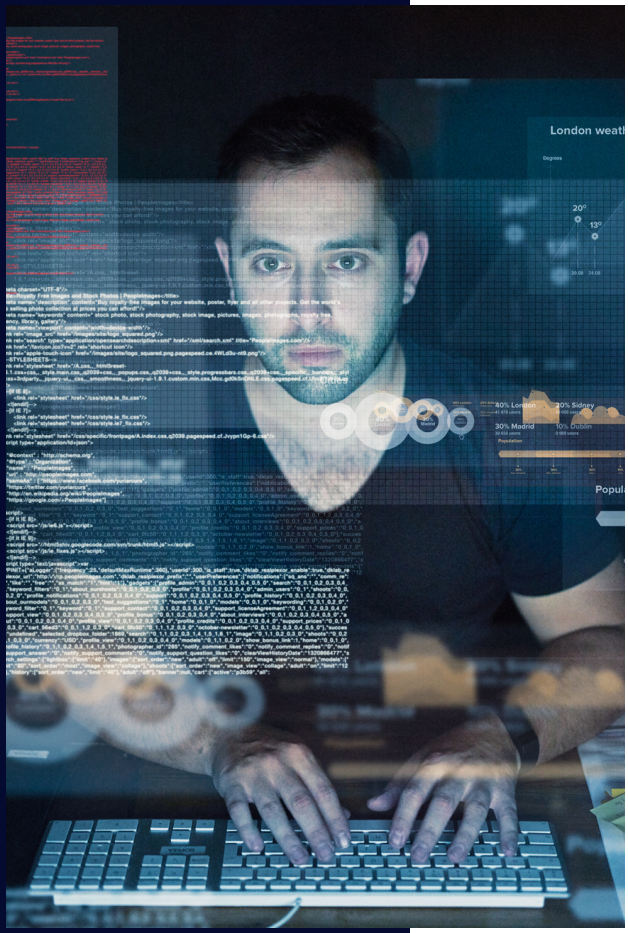
Audience:

- CIO, CTO
- IT Leaders
- Digital Business Leaders
- Digital Product Leaders
- Business Architects
- Application Architects

Objectives:

- Understand containerization vs virtualization
- Understand scaling vs auto scaling
- How to use containerization to auto scale cloud-native applications

Estimated read time:
8 mins



INTRODUCTION

The cloud-native model provides the framework or blueprint for developing applications explicitly designed for the cloud. This model is made up of microservices architecture, containerization, orchestration and automation, and load-balancing. It must be resilient and self-healing, scalable and elastic, as well as stateless and API-driven. Lastly, the cloud-native model aims to maximize cost efficiency, scalability, and flexibility, providing organizations with the ability to develop and deploy applications more rapidly and respond quickly to changing business requirements.

Undoubtedly, cloud computing and the computing-as-a-service (cloud-native) model are innovative, forward-looking technologies providing users with the option of dynamic, dependable, resilient computing, including aspects such as:

- An emphasis on Quality of service (QoS)
- Highly scalable and highly available software applications
- Cost-effective resources without the need to buy and maintain physical infrastructure

While the cloud-native model offers users almost unlimited resources such as CPUs, memory, persistent storage, and network at competitive prices, it must be effectively managed at an application level to fully leverage these resources and ensure optimal performance, reliability, and cost-effectiveness.

As described throughout this text, cloud-native applications are designed to take full advantage of the cloud infrastructure and servers. However, effectively managing them requires careful consideration of aspects such as scaling and auto-scaling, monitoring and alerting, performance optimization, security and compliance, and container orchestration.

While every one of these aspects is imperative to effective cloud-native application management, this article's core focus is scaling and auto-scaling strategies for cloud-native applications. The ability to scale a cloud-native application is based on the principles of containerization and virtualization as the cornerstones of cloud-native computing.

Let's continue this discussion by examining each concept and its fundamental differences, the difference between scaling and auto-scaling, the types of scaling (horizontal and vertical), and what the concept, Quality of Service (QoS) is before looking at several strategies used to implement scaling and auto-scaling cloud-native applications.

CONTAINERIZATION VERSUS VIRTUALIZATION

In summary, containerization and virtualization are technologies used to deploy and manage software applications in the cloud; however, they differ in isolation and resource utilization. In a sense, containerization is a form of virtualization, but it differs from traditional virtualization in several key aspects. Both technologies aim to create isolated environments for running applications but do it differently.

In traditional virtualization (or hardware virtualization), a hypervisor (software that creates and runs virtual machines) runs on top of the physical hardware and allows multiple VMs (virtual machines) to run on top of it. Each virtual machine emulates the underlying hardware and includes its own operating system and kernel. In other words, multiple operating systems can run on the same physical server, effectively creating separate virtualized environments.

On the other hand, containerization is a type of operating system-level virtualization. It doesn't simulate the entire physical machine, just its operating system. Therefore, containerization allows multiple containers to share the same host OS kernel, which provides isolated user spaces for each container.

In a traditional sense, containers package an application and its dependencies, libraries, and configurations needed to run the application. In the cloud-native context, applications comprise many different microservices, each encapsulating a single business function. Each microservice and its dependencies, libraries, and configurations are packaged in a single container. Each container encapsulates a single function, but all the containers share the host operating system kernel, resulting in a lower resource overhead than traditional virtual machines.

SCALING VERSUS AUTO-SCALING

Auto-scaling and scaling are related concepts in terms of managing resources to meet the demands of an application or system. They differ, however, in their approach and degree of automation.

Note:
In this article's context, scaling refers to adjusting the number of containers and the hardware resources needed by each container, such as CPU, RAM, networking, and disk space.

Manual Scaling V/S Auto-Scaling

DevOps teams manually scale these resources based on their current workload assessment. The team may decide to increase (or scale up) the number of containerized microservices containing a specific business function if an application, such as an eCommerce store's checkout process, experiences increased traffic. The team will scale back these containers when demand decreases to meet the lower demand.

On the other hand, auto-scaling is an automated process where the system adjusts its resources dynamically based on predefined rules or metrics. The goal of auto-scaling is to match the available resources with the current demand, allowing the system to scale up or down automatically as needed.

HORIZONTAL VERSUS VERTICAL SCALING

There are two types of scaling: horizontal scaling and vertical scaling. The most significant difference is that horizontal scaling adds more virtual machines, containers, or nodes to a system. In contrast, vertical scaling adds more resources (CPU, RAM, networking, and storage) to an existing container, machine, or node. In other words, horizontal scaling horizontally implies scaling out, whereas vertical scaling indicates scaling up.

In the containerization context, horizontal scaling adds additional containers to handle an increased workload, while vertical scaling adds more resources to a specific container to handle the extra workload.

QUALITY OF SERVICE (QOS)

The research article "*QoS-aware resource scheduling using whale optimization algorithm for microservice applications*" describes containerizing microservices as a "*structural approach where multiple small sets of services are composed and processed independently with lightweight communication mechanisms.*"

According to the authors of this paper, when deploying and maintaining a cloud-native application, it is mandatory to stick to the accompanying service level agreement (SLA) with an uptime usually of 99.999%. The best way to meet these SLA requirements is to implement what is known as a Quality of Service-aware algorithm to find the best resources when deploying containerized microservices.

Containerization and QoS are related in that containerization provides a level of isolation and resource allocation for containerized applications. For instance, you can define resource limits and QoS settings for different containers or groups of containers in container orchestration platforms such as Kubernetes.

A robust orchestration and management system (like Kubernetes) is essential to ensure the efficient deployment and management of these microservices.

Note:
Kubernetes is one of the most popular container orchestration platforms that can handle these tasks effectively.

USING CONTAINERIZATION TO SCALE AND AUTO-SCALE CLOUD-NATIVE APPLICATIONS

As described above, containerization forms the core of all cloud-native scaling (and auto-scaling) strategies. Therefore, let's dive into this topic by looking at the following use case.

You are the founder of a mental health MedTech start-up with an app that helps users focus on their mental and emotional well-being by offering evidence-based digital self-help as well as the ability to book virtual appointments with a psychologist. This app can be divided into the following microservices or independent services that communicate with each other to form the application:

Each microservice is packaged in its own container and exposes an API to communicate with the other microservices. The user ID is typically passed between APIs as it is the primary key or part of a foreign key for each table in the linked database.

- **User authentication service**
- **Digital self-help service**
- **Mood tracking and journaling service**
- **User progress tracking service**
- **Virtual appointment booking service**
- **Secure video conference service**
- **Community support service**
- **Reminder and notification service**
- **Integration with wearable devices service**
- **Payments and billing service**
- **Data privacy and security service**

A USER JOURNEY

Now that we have a list of services, let's look at a user journey to see how each microservice interacts with each other.

Note:
For this discussion, we will limit each microservice to one user.

The starting point: Log into the App

For ease of application, let's assume you are the user. And your starting point is a web-based GUI (graphical user interface) where you can log into the app.

Note:
This web page functions as the start-up's home page and the entry point for the application.

When you click the **Sign In** button, the user authentication service starts and authenticates you.

Record Your Mood in Real Time and Journal

Once you have logged into the app, the next step is to use the mood tracking and journaling service to log your mood and journal for the day. When you click this menu option, the container containing this microservice spins up, passing your user ID from the user authentication service.

The user authentication service has completed its job, so it spins down.

Book a Virtual Appointment

After journaling for the day, the next step is to book a virtual appointment with your psychologist and log out.

The microservice used here is the virtual appointment booking service.

Note:
When you log out of the app, the container management platform spins down all the containers containing the microservices you used.

Appointment day

The next time you log into the app is to attend your scheduled virtual appointment. The same sign-in process occurs as described above. The only difference is that you'll open the secure video conference service to chat virtually with your psychologist. Once finished, you'll log out again.

Scaling and Auto-Scaling Strategies

Now that we've looked at a simple user journey, the next step is to use this user journey to consider several scaling and auto-scaling strategies, ensuring each containerized microservice is deployed and scaled independently when required, using QoS metrics to meet the stated uptime in the accompanying SLA.

Note:
We cannot reiterate enough that scaling and auto-scaling strategies are critical, making sure each microservice can handle varying workloads efficiently and promoting flexibility, cost-effectiveness, and maintainability.

Here are several strategies for scaling and auto scaling each microservice independently:

HORIZONTAL SCALING

Consider implementing horizontal scaling for microservices that handle user interactions, such as authentication, digital self-help, mood tracking, and journaling. This strategy involves deploying multiple instances of the same microservices and using a load balancer to distribute incoming requests.

As stated above, each microservice can only service one user's requests in our use case. Therefore, the same number of container instances must be spun up for each additional user and vice versa. In other words, as the user base grows, you can add more instances to handle the increased traffic.

CONTAINER ORCHESTRATION

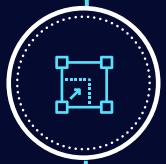
Use a container orchestration platform like Kubernetes to manage and automate the deployment, scaling, and management of your microservices. Kubernetes provides built-in auto-scaling capabilities, allowing you to set up rules to automatically scale the number of containers based on metrics such as CPU utilization or request rate.

AUTO-SCALING RULES

Set up auto-scaling rules based on performance metrics for each microservice. For example, if the virtual booking service experiences high demand during certain hours, configure the auto-scaling rule to increase the number of instances during these periods.

SERVERLESS FUNCTIONS

For microservices with intermittent or low-resource utilization, consider implementing them as serverless functions instead of containerized microservices. Serverless computing platforms automatically manage the scaling and provisioning of resources based on incoming requests, saving costs, and simplifying infrastructure management.



DATABASE SCALING

Ensure that the database used by your microservices is capable of scaling independently. Use cloud-based databases with auto-scaling capabilities to handle increasing storage and read/write operations as the user base grows. It is also a good idea to consider adding an instance of the database behind each microservice to prevent performance bottlenecks.

MICROSERVICE DECOMPOSITION

If a particular microservice becomes a performance bottleneck, consider decomposing it into smaller, more specialized microservices. This allows you to scale individual components independently and optimize resource allocation.

CONCLUSION

As seen throughout this text, implementing scaling and auto-scaling strategies is mandatory to ensure that your cloud-native application can effectively handle fluctuating workloads, ensure efficient resource utilization, and maintain the requisite QoS standards, resulting in a reliable, resilient, highly available, resource-effective, and cost-effective application.

Moreover, the practical use case describes how dividing your application into containerized microservices, you can develop, deploy, and scale each component independently, promoting flexibility, maintainability, and ease of development, ensuring that the application executes as optimally as possible.

ABOUT FIORANO

Enabling Change at the Speed of Thought

Fiorano is a cloud-native event-driven microservices platform that combines integration, low-code, and eiPaaS capabilities to build and deploy global hybrid multi-cloud applications.

By making business processes event driven, Fiorano helps enterprises achieve massive scalability, responsiveness, and increased productivity.

With Fiorano, companies can respond better in volatile markets and deliver exceptional customer and employee experiences.

Drop us a line:

info@fiorano.com

www.fiorano.com | Fiorano Software | @FioranoGlobal