FaaS + K8s = faast!

Santhosh Kumar M
23D0369

Siva Prasad Reddy Garlapati
23M0747

GitHub: https://git.cse.iitb.ac.in/santhoshkm/dfaast.git

Abstract

The integration of Functions as a Service (FaaS) with Kubernetes (K8s) presents a promising approach to building scalable and efficient cloud-native applications. In this report, we detail the steps involved in setting up FaaS using Docker and Kubernetes on a private cloud infrastructure. We demonstrate the functional completeness, correctness of feature sets, and performance through autoscaling experiments.

1   Introduction

Cloud computing isn going through a paradigm shift towards more agile, scalable, and efficient architectures. This transformation is largely driven by the adoption of microservices and serverless computing models, which offer benefits such as reduced operational overhead, improved scalability, and increased resource utilization. At the forefront of this evolution are Functions as a Service (FaaS) and container orchestration platforms like Kubernetes (K8s), which provide the foundation for building and deploying modern cloud-native applications.

1.1   Functions as a Service (FaaS)

FaaS is a development paradigm where developers write smaller function and can deploy them without having to bother about the scalability and operational aspects. Unlike traditional monolithic applications, which require constant provisioning and management of infrastructure, FaaS abstracts the underlying infrastructure concerns, allowing developers to focus solely on writing code. This results in greater agility, as developers can rapidly develop, deploy, and scale individual functions without worrying about server provisioning or maintenance.

1.2 Kubernetes (K8s)

Kubernetes has emerged as the de facto standard for container orchestration, providing a robust platform for deploying, managing, and scaling containerized applications. With features such as automated scheduling, service discovery, and horizontal scaling, Kubernetes simplifies the complexities associated with deploying and managing containers at scale. By abstracting away the underlying infrastructure, Kubernetes enables developers to focus on building and deploying applications, while ensuring reliability, scalability, and flexibility.
Integration of FaaS with Kubernetes:

While FaaS and Kubernetes offer distinct advantages on their own, their integration presents a compelling proposition for building cloud-native applications. By combining the serverless computing model of FaaS with the container orchestration capabilities of Kubernetes, organizations can achieve the best of both worlds: the scalability and agility of serverless computing, coupled with the flexibility and control of containerized environments. This integration allows developers to leverage the benefits of serverless computing while retaining the ability to manage and orchestrate containers at scale, making it an ideal choice for modern cloud-native applications.
The integration of FaaS with Kubernetes offers several key benefits:

- Scalability: Kubernetes provides automated scaling capabilities, allowing FaaS functions to scale dynamically in response to fluctuations in workload, ensuring optimal resource utilization and cost efficiency.

- Flexibility: By running FaaS functions in containers orchestrated by Kubernetes, developers have the flexibility to deploy, manage, and scale applications using familiar tools and workflows, without being locked into a specific FaaS platform.

- Portability: Kubernetes abstracts away the underlying infrastructure, making it easier to deploy FaaS functions across different environments, including on-premises data centers, public clouds, and hybrid cloud environments.

- Operational Efficiency: Kubernetes simplifies the operational overhead associated with managing and orchestrating containers, allowing organizations to focus on developing and deploying applications, rather than managing infrastructure.

1.3 Objectives of the Project

The primary objectives of this project are as follows:

- To set up a FaaS platform using Docker containers.

- To deploy and integrate the FaaS platform with Kubernetes on a private cloud infrastructure.

- To demonstrate the functional completeness and correctness of feature sets of the integrated solution.

- To conduct experiments to demonstrate platform scalability with no operational overheads

2 Background and Motivation

2.1 Overview of Docker and Kubernetes

2.1.1 Docker

Docker is an open-source platform that enables developers to develop, deploy, and run applications in containers. Containers are lightweight, portable, and self-sufficient units that contain everything needed to run an application, including the code, runtime, libraries, and dependencies. Docker provides tools and services for building, distributing, and running containers efficiently. It uses a client-server architecture, where the Docker client interacts with the Docker daemon to manage containers and containerized applications. Docker has gained widespread adoption due to its ease of use, portability, and scalability.
Limitations/Challenges of Docker

- Resource Management: Docker lacks native support for advanced resource management features like auto-scaling and load balancing.

- Complexity: Managing and orchestrating multiple containers manually can be complex and time-consuming, especially in large-scale deployments.

- Compatibility: Ensuring compatibility between containers and different environments, such as development, testing, and production, can be challenging.

2.1.2 Kubernetes

Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It provides a robust set of features for managing containerized workloads across a cluster of machines, enabling developers to deploy applications with ease and scale them seamlessly as needed.

2.2 Functions as a Service (FaaS)

Functions as a Service (FaaS) is a cloud computing model that abstracts away server management and infrastructure concerns, allowing developers to focus solely on writing and deploying individual functions or pieces of code. FaaS platforms execute these functions in response to events or triggers, such as HTTP requests, database changes, or file uploads, without the need for developers to provision or manage servers. FaaS offers a serverless approach to building and

3

deploying applications, enabling rapid development, scalability, and cost efficiency.

Architecture of FaaS Platforms:

1 Execution Environment: FaaS platforms provide a runtime environment for executing functions, typically based on containerization technology such as Docker. Each function runs in its isolated environment, ensuring resource isolation and security.

2 Trigger Mechanism: Functions are triggered by events or triggers, which can originate from various sources such HTTP requests, message queues, timers, or external APIs. FaaS platforms provide integrations with these event sources to enable seamless triggering of functions.

3 Orchestration and Scaling: FaaS platforms automatically scale function instances based on demand, ensuring optimal resource utilization and cost efficiency. Auto-scaling policies can be configured based on metrics such as CPU utilization, request rate, or custom metrics.

4 Management and Monitoring: FaaS platforms offer management and monitoring capabilities for functions, allowing developers to deploy, monitor, and manage functions using web interfaces or command-line tools. Monitoring tools provide insights into function performance, resource usage, and error handling.

Key Components of FaaS Platforms:

1 Function: The fundamental unit of computation in FaaS platforms, representing a single piece of code or a function that performs a specific task. Functions are stateless and ephemeral, with no persistent state or local storage between invocations.

2 Trigger: Events or triggers that initiate the execution of functions. Triggers can include HTTP requests, database changes, file uploads, or timer-based events.

3 Runtime Environment: The execution environment in which functions are executed, typically based on containerization technology such as Docker. Runtime environments provide isolation, security, and resource management for executing functions.

4 Event Source: Sources of events or triggers that initiate function execution, such as HTTP endpoints, message queues, or cloud storage services. FaaS platforms provide integrations with various event sources to enable seamless triggering of functions.

Workflow of FaaS Execution:

1 Function Deployment: Developers write and deploy functions to the FaaS platform using tools provided by the platform or through command-line interfaces. Functions can be deployed as standalone units or as part of larger applications.

2 Event Triggering: Events or triggers initiate the execution of functions based on predefined criteria or conditions. FaaS platforms handle event routing and triggering, ensuring that functions are executed in response to the appropriate events.

3 Function Execution: Functions are executed in isolated runtime environments, with each function instance handling a single event or request. FaaS platforms manage the lifecycle of function instances, including provisioning, execution, and teardown.

4 Result Handling: Upon completion of function execution, results are returned to the caller or sent to downstream services for further processing. FaaS platforms handle result handling and error management, ensuring reliability and fault tolerance.

Benefits of Functions as a Service (FaaS):

1 Rapid Development: FaaS platforms enable rapid development and deployment of applications, allowing developers to focus on writing code without worrying about infrastructure concerns.

2 Scalability: FaaS platforms automatically scale function instances based on demand, ensuring applications can handle fluctuations in workload without manual intervention.

3 Cost Efficiency: FaaS platforms offer a pay-per-use billing model, charging users only for the resources consumed during function execution. This results in cost efficiency and eliminates the need for upfront investment in infrastructure.

4 Simplified Operations: FaaS platforms abstract away server management and operational tasks, reducing the operational overhead for managing and maintaining infrastructure.

2.3   Importance of Private Cloud Infrastructure:

Private cloud infrastructure refers to cloud computing resources that are dedicated to a single organization and are not shared with other organizations. Private clouds can be hosted on-premises or by third-party providers and offer several key benefits:

- Security and Compliance: Private clouds offer greater control and customization over security measures and compliance requirements compared to public clouds. Organizations can implement stringent security policies

and regulatory compliance standards to protect sensitive data and meet industry-specific regulations.

- Performance and Reliability: Private clouds provide dedicated resources and infrastructure, resulting in improved performance, reliability, and availability compared to shared public cloud environments. Organizations can tailor the infrastructure to meet their specific performance and reliability requirements, ensuring consistent performance for mission-critical applications.

- Data Sovereignty: Private clouds allow organizations to retain full control over their data and maintain data sovereignty. This is particularly important for organizations operating in highly regulated industries or regions with strict data privacy laws.

- Customization and Flexibility: Private clouds offer greater customization and flexibility compared to public clouds, allowing organizations to tailor the infrastructure to their unique requirements. Organizations can customize the network, storage, and compute resources to optimize performance and meet the needs of their applications.

Private cloud infrastructure provides organizations with greater control, security, performance, and customization compared to public cloud environments, making it an ideal choice for sensitive workloads and mission-critical applications.

3 Setup design and Implementation

3.1 Setup

We utilized a single physical machine for our setup, employing VirtualBox to create a virtual machine running Ubuntu 22.04. This virtual machine, along with the physical host, served as the two nodes necessary for running Docker containers and Kubernetes. The virtual machine was designated as the Kubernetes control plane, while the physical machine was configured as a worker node. Initially, we attempted to incorporate an additional virtual machine as a worker node. However, this setup proved to be unstable, frequently resulting in system hangs. As a result, we opted to proceed with only two nodes for testing purposes. The configuration of our demo setup is illustrated in Figure 1.

To ensure network connectivity, we configured the virtual machine with a network bridge linked to the host system, enabling it to seamlessly integrate into the lab network and obtain an IPv4 address from the Lab network's DHCP system.

For installation, we followed standard procedures using 'apt-get' to install Docker and Kubernetes on both nodes. On the control-plane node, 'kubeadm' was installed to facilitate Kubernetes administration operations. Subsequently, upon initializing the control plane, we utilized security certificates in the worker node
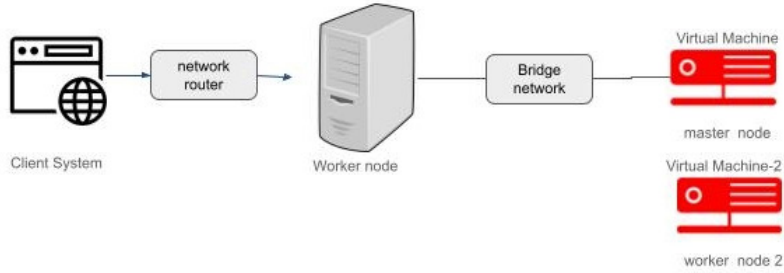
Figure 1: Demo environment setup

to join the Kubernetes cluster.
The steps followed for installation are given below

- 'setup.sh': Load required kernel modules (overlay and br _netfilter) on both the nodes.

- 'setup.sh': Configure the network for IP packet forwarding

- 'setup2.sh': Install Docker dependencies modules namely gnupg2 and ca certificates

- 'setup2.sh': Enable installation from Docker repository

- 'setup2.sh': Install containerd.io

- 'setup2.sh': Configure the container to start by default on reboot

- 'setup3.sh': Enable installation from Kubernetes repository

- 'setup3.sh': Install "kubelet, kubectl, and kubeadm" on both the nodes

- On the control-plane/master node, do the following:

  - 'setup-master.sh': Setup Kubernetes master node
  - 'setup-master.sh': Create the token to be used by the worker node to join the cluster

- On the worker node, join the cluster using the token from the master node.

3.2 Implementation Details

In this section, we delve into the technical aspects of implementing various capabilities for managing Functions as a Service (FaaS) within a Kubernetes environment. FaaS offers a serverless computing paradigm that enables developers to focus on writing and deploying individual functions without managing the underlying infrastructure. Leveraging Kubernetes as the orchestration platform provides scalability, reliability, and flexibility for deploying and managing

7

functions effectively. We outline the steps involved in deploying functions, pausing and resuming their execution, updating to newer versions, rolling back to older versions, adjusting resource limits, deleting functions, retrieving logs and resource usage metrics, and setting/getting replication counts. Additionally, we highlight key assumptions and considerations guiding the implementation process.

3.2.1 Capabilities Implemented

- 1. Build Docker - The program facilitates the seamless building of Docker images containing the function code. Using Docker, we encapsulate the function along with its dependencies into a containerized environment, ensuring consistency across different deployment environments.

- 2. Deploy Function - Deploying functions to Kubernetes is simplified through the program. It orchestrates the deployment process, ensuring that the function is correctly instantiated within the cluster and is accessible for execution.

- 3. Update Deployment - The ability to update function deployments is crucial for incorporating changes or new versions of the function code. Our program efficiently handles updates to existing deployments, ensuring smooth transitions without downtime.

- 4. Delete Deployment/Label - To manage resources effectively, the program offers functionalities to delete deployments based on specific labels or individual deployment names. This capability streamlines the removal of outdated or redundant deployments, maintaining cluster cleanliness.

- 5. Scale Deployment - Automatic scaling of deployments based on resource utilization is a key feature of Kubernetes. Our program leverages Horizontal Pod Autoscaling (HPA) to dynamically adjust the number of replicas based on CPU and memory usage metrics.

- 6. Expose Service - Services expose functions to external clients, allowing them to access the functionality provided by the deployed functions. Our program supports various service types, including ClusterIP, NodePort, LoadBalancer, and ExternalName, enabling flexible and secure access.

- 7. Get Function Logs - Logging is essential for monitoring the execution and performance of functions. Our program provides the capability to retrieve logs generated by function executions, facilitating debugging and performance analysis.

- 8. Get Resource Usage - Monitoring resource usage is critical for optimizing the performance and cost-effectiveness of function deployments. Our program retrieves resource usage metrics, including CPU and memory utilization, enabling administrators to identify potential bottlenecks and optimize resource allocation.

Figure 2 and 3 pictorially show the steps involved in function deployment and trigger operations.
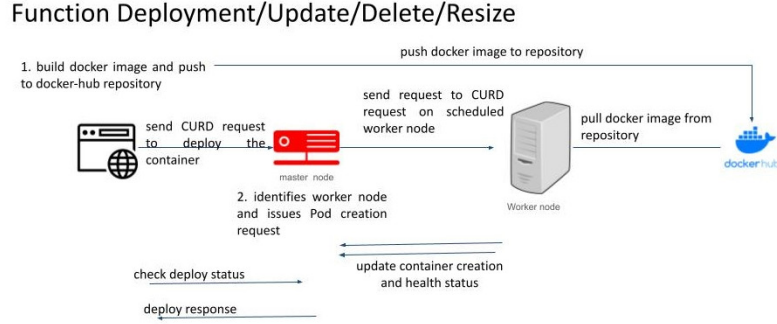

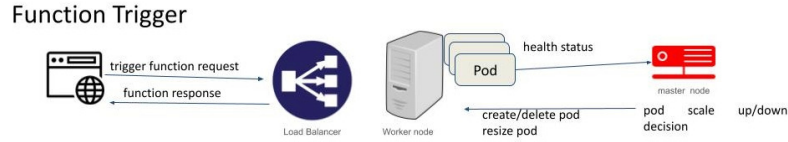Figure 2: Function Deployment

Figure 3: Function Trigger

Below are the assumptions/limitations of our implementation

- Namespace Deployment - All functions would be deployed to "default" namespace. We have not implemented this functionality.

- New Cluster - We assume Kubernetes cluster is pre-created for use.

- Single Cluster - We use only one cluster and have not implemented multiple cluster management.

- Node addition - We currently support anual addition of worker nodes.

- HTTP-Based Trigger - Only support HTTP-based triggers for invoking functions, excluding other trigger types (e.g., message queues, cron jobs).

- Label Grouping - we have not implemented label based grouping of various Kubernetes resources including functions.

4 Results and Discussion

In this section, we present the results of our implemented Python program for managing Functions as a Service (FaaS) on Kubernetes. The program encompasses various functionalities crucial for deploying, managing, and monitoring

functions within a Kubernetes cluster. We conducted comprehensive testing to evaluate the functional correctness and performance of each feature.

Testing and Evaluation

We conducted rigorous testing to validate the functionality and performance of our program. This included:

- Functional Correctness Testing: We verified that each functionality operates as intended, ensuring that deployments, updates, scaling, and deletions occur without errors.

- Performance Testing: We evaluated the program's performance under varying conditions, including scenarios where CPU and memory usage increased. We analyzed the program's ability to scale deployments dynamically in response to changing resource demands.

To test Autoscaling functionality, we deployed a multiprocess function that takes CPU percentage as argument via HTTP request, and spawns the thread to continously be in while loop at fixed interval. Similar algorithm was written to test the memory usage where we allocate memory based on the HTTP request parameter.

Results and Performance Metrics

Our testing revealed that the implemented functionalities operate as expected, providing a robust framework for managing FaaS deployments on Kubernetes. We observed:

- Functional Completeness: All planned functionalities were successfully implemented and demonstrated their intended behavior.

- Correctness: Deployments, updates, and deletions occurred without errors, validating the program's reliability and correctness.

- Performance Benchmarking: We generated performance metrics, including deployment latency, scaling response time, and resource utilization. Graphical representations of CPU and memory usage during load testing are presented below.

Figure 4 shows the auto scaling for different CPU workloads.

5 Conclusion

The Python program developed for managing FaaS deployments on Kubernetes proved to be effective and reliable. It offers comprehensive functionalities for deploying, updating, scaling, and monitoring functions within a Kubernetes environment. Through rigorous testing, we confirmed its functional completeness, correctness, and performance, underscoring its suitability for real-world FaaS applications.
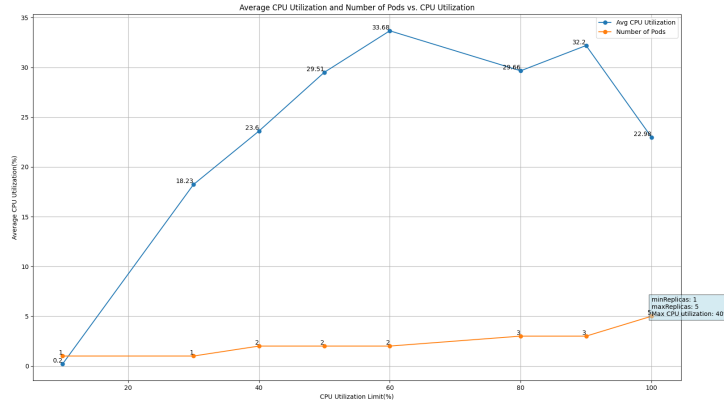
Figure 4: Auto Scaling in Action

5.1   Future Work

- Namespace Management: Implement support for deploying functions to multiple namespaces, allowing for better organization and isolation of resources within the Kubernetes cluster.

- Cluster Automation: Develop automated procedures for creating and provisioning new Kubernetes clusters, enabling seamless deployment and scaling of FaaS applications across different environments.

- Multi-Cluster Support: Extend the program to manage multiple Kubernetes clusters, enabling distributed deployments and enhancing fault tolerance and scalability.

- Node Automation: Implement automated node management capabilities, allowing for dynamic scaling of worker nodes based on resource demand and workload requirements.

- Trigger Diversification: Expand trigger support beyond HTTP-based triggers to include other event-driven mechanisms, such as message queues, scheduled jobs, or external events.

- Label-Based Grouping: Introduce label-based grouping of Kubernetes resources, enabling easier management and orchestration of related functions, deployments, and services.

By addressing these limitations and pursuing future enhancements, our program can evolve into a more versatile and comprehensive solution for managing FaaS deployments on Kubernetes, catering to a broader range of use cases and deployment scenarios.

References

[1] Docker desktop overview, 12 2021.

[2] Kubernetes . Overview, 09 2023.

[3] Debojeet Das. deep dive into containers (docker and k8s), 04 2024.