WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

Unleashing the Power of Kubernetes: Embracing Openness and Vendor Neutrality for Agile Container Development in an Evolving Landscape

Ramamurthy Valavandan [a], Balakrishnan Gothandapani [b], Archana Gnanavel [c], Nithya Ramamurthy [d], Malarvizhi Balakrishnan [e], Sinthana Gnanavel [f], Savitha Ramamurthy [g]

[a] Technical Architect, Nature Labs, 1/12A, Bommasamuthiram & Post, Namakkal District, 637002, Tamil Nadu, India
[b] Research Director, Nature Labs, 1/12A, Bommasamuthiram & Post, Namakkal District, 637002, Tamil Nadu, India
[c] Scientific Officer, Nature Labs, 1/12A, Bommasamuthiram & Post, Namakkal District, 637002, Tamil Nadu, India
[d] Project Manager, Nature Labs, 1/12A, Bommasamuthiram & Post, Namakkal District, 637002, Tamil Nadu, India
[e] Application Developer, Nature Labs, 1/12A, Bommasamuthiram & Post, Namakkal District, 637002, Tamil Nadu, India
[f] Application Developer, Nature Labs, 1/12A, Bommasamuthiram & Post, Namakkal District, 637002, Tamil Nadu, India
[g] Application Developer, Nature Labs, 1/12A, Bommasamuthiram & Post, Namakkal District, 637002, Tamil Nadu, India

DOI- https://doi.org/10.55248/gengpi.4.523.44101

A B S T R A C T

The evolving container landscape has witnessed the rise of various organizations and technologies competing for dominance. Kubernetes, as a powerful orchestration tool, has adapted to embrace open standards and interoperability among different container engines. Initially, Kubernetes development was focused on Docker, but as Docker introduced vendor-lock characteristics, alternative projects gained popularity. Kubernetes has embraced openness and vendor neutrality, aligning with the changing container ecosystem.

A crucial component in Kubernetes is the Container Runtime Interface (CRI), which enables seamless integration between container runtimes and kubelet. Containerd has gained significant community support as a container runtime, adhering to Open Container Initiative (OCI) specifications. The container landscape shift towards open specifications and flexibility has led to the exploration of alternative tools and open-source options to align with the trend towards openness and vendor neutrality.

When containerizing legacy applications, the decision to containerize as is or rewrite as decoupled microservices should be carefully considered. Factors such as application nature, scalability needs, and complexity should be evaluated. While Docker has been the dominant industry standard, organizations are transitioning to alternative open-source tools, recognizing the industry shift towards open specifications.

The process of containerizing an application involves creating a Dockerfile, building the container, testing its performance, and pushing it to a repository. Balancing privacy and accessibility, setting up a local repository provides control over image accessibility and enhances privacy and security.

Deploying the application within Kubernetes involves creating a deployment using the desired image. Verifying the Pod's status and testing the application's performance are crucial steps to ensure proper functionality. Executing commands within a container, enhancing functionality with multi-container Pods, and utilizing container probes for health monitoring are additional aspects to consider.

To refine the terminology, ambassador, adapter, and sidecar are common expressions used to describe the roles of secondary containers within multi-container Pods. Each container collaborates to fulfill the application's requirements and enhance its functionality.

Developing a comprehensive testing suite is essential to verify the deployed application's functionality in a Kubernetes environment. Leveraging built-in functionalities provided by kubectl, such as the "describe" command for detailed object information and the "logs" command for retrieving container logs, can aid in testing and troubleshooting. Custom testing tools tailored to specific deployments can provide comprehensive tests and validations for optimal application performance in Kubernetes' transient environment.

Keywords: Kubernetes, containerization, openness, vendor neutrality, container runtime, Container Runtime Interface (CRI), containerd, Docker, open specifications, multi-container Pods, testing suite, kubectl

1. Introduction

The containerization revolution has transformed the way software applications are developed, deployed, and managed. As organizations embrace containers for their scalability, portability, and resource efficiency, they face the challenge of selecting the right orchestration platform to effectively manage their containerized environments. In this evolving landscape, Kubernetes has emerged as the de facto standard for container orchestration, providing a powerful toolset to manage and scale containerized applications.

However, the container ecosystem is not static. It is characterized by a diverse range of organizations, technologies, and competing standards vying for dominance. As the ecosystem evolves, it becomes crucial to adopt open standards and foster vendor neutrality to ensure interoperability and flexibility in container development. This paper explores the power of Kubernetes in embracing openness and vendor neutrality, enabling agile container development in an ever-changing landscape.

Kubernetes was initially developed with a strong focus on integrating with Docker, which quickly gained popularity as the go-to container engine. However, as Docker introduced vendor-specific features and began exhibiting vendor-lock characteristics, alternative container runtimes and orchestration tools gained traction. Kubernetes recognized the need for openness and interoperability and adapted to support multiple container runtimes through the Container Runtime Interface (CRI).

One of the key components in Kubernetes, the CRI, enables seamless integration between container runtimes and the kubelet, the node agent responsible for managing containers. As the container landscape shifted towards open specifications, containerd emerged as a popular container runtime, aligning with the Open Container Initiative (OCI) specifications. This shift towards open specifications and flexibility has encouraged exploration and adoption of alternative tools and open-source options, allowing organizations to embrace openness and vendor neutrality.

When containerizing applications, organizations face the decision of whether to containerize them as-is or refactor them into decoupled microservices. Factors such as the nature of the application, scalability requirements, and complexity need to be carefully evaluated. While Docker has been the dominant industry standard, organizations are now transitioning towards alternative open-source tools, recognizing the industry's shift towards open specifications and vendor neutrality.

The process of containerizing an application involves creating a Dockerfile, building the container image, testing its performance, and pushing it to a repository. Balancing privacy and accessibility, organizations can set up local repositories to have better control over image accessibility, enhancing privacy and security.

Deploying containerized applications within Kubernetes involves creating deployments using the desired container images. Verifying the status of Pods, which are groups of containers, and testing the application's performance are crucial steps to ensure proper functionality. Additionally, leveraging the capabilities of multi-container Pods and utilizing container probes for health monitoring can enhance the overall functionality and resilience of the application.

To refine the terminology used in Kubernetes, terms like ambassador, adapter, and sidecar are commonly employed to describe the roles of secondary containers within multi-container Pods. Each container collaborates to fulfill the application's requirements and enhance its functionality, enabling a modular and flexible approach to container development.

Developing a comprehensive testing suite is essential to verify the functionality of deployed applications in a Kubernetes environment. Leveraging built-in functionalities provided by Kubernetes command-line tool, kubectl, such as the "describe" command for detailed object information and the "logs" command for retrieving container logs, can greatly aid in testing and troubleshooting. Organizations can also develop custom testing tools tailored to their specific requirements.

By embracing openness and vendor neutrality, Kubernetes empowers organizations to unleash the full potential of agile container development in an evolving landscape. The adoption of open specifications and alternative open-source tools allows organizations to stay aligned with industry trends, achieve greater flexibility and interoperability, and future-proof their containerized environments.

Nomenclature

In the rapidly evolving landscape of containerization, nomenclature plays a crucial role in establishing standardized and consistent terminology within the Kubernetes ecosystem. This section explores the significance of nomenclature in Kubernetes and containerization, emphasizing its role in clarifying the roles and relationships between various entities. By providing a common language, nomenclature ensures clear communication and facilitates effective collaboration among developers and operators.

Pods: Fundamental Building Blocks

Pods serve as the fundamental building blocks in Kubernetes, enabling the encapsulation of one or more containers deployed together on a shared host.

Each Pod possesses a unique IP address and shares resources within the cluster.

Understanding the concept of Pods is essential for comprehending the organization and management of containerized applications within Kubernetes.

Deployments: Managing Pod Lifecycles

Deployments take charge of managing the lifecycle of Pods, facilitating scaling, rolling updates, and rollback operations for application instances.

They ensure the continuous operation of the desired number of Pods, dynamically adapting to fluctuations in demand.

Deployments play a crucial role in maintaining the availability and reliability of containerized applications.

Services: Enabling Seamless Communication

Services provide a stable network endpoint that allows access to a group of Pods.

They enable load balancing and service discovery within the Kubernetes cluster, fostering seamless communication between applications.

Understanding Services and their configuration is vital for ensuring efficient and reliable communication between different components of a distributed application.

ReplicaSets: Ensuring Availability and Fault Tolerance

ReplicaSets guarantee the desired number of Pods specified in the associated Deployment, ensuring the maintenance of the defined state.

They contribute to high availability and fault tolerance by automatically replacing any failed Pods.

Understanding ReplicaSets is crucial for building resilient and fault-tolerant applications in Kubernetes.

Labels and Selectors: Flexible Object Organization

Labels and Selectors employ key-value pairs to categorize and select Kubernetes objects based on common attributes.

This approach provides flexibility in organizing and managing objects, facilitating efficient operations.

Proper utilization of Labels and Selectors enhances the manageability and scalability of Kubernetes deployments.

Ingress: Managing External Access

Ingress, as an API object, manages external access to services within a cluster, serving as a gateway or entry point.

By defining rules within the Ingress configuration, it intelligently routes incoming requests to the appropriate Services.

Understanding Ingress is essential for managing external access and optimizing the routing of traffic to the underlying services.

1.1 Evolving Container Landscape: Embracing Openness and Vendor Neutrality

The container landscape has witnessed the emergence of numerous organizations and technologies, each vying to establish their presence. Kubernetes [1], as a powerful orchestration tool, has adapted to accommodate various container engines, as the community emphasizes open standards and seamless interoperability. Initially, the focus of Kubernetes development was not on interoperability due to the dominant presence of Docker [2]. However, as Docker evolved and introduced vendor-lock characteristics throughout the container lifecycle, alternative projects and features gained popularity. Consequently, Kubernetes has embraced openness and independence, aligning with the evolving container ecosystem.[3]

A container runtime serves as the essential component responsible for executing containerized applications upon request. While Docker Engine was once widely adopted, alternative runtimes like containerd [4] and CRI-O have gained significant community support. In our research, we will be utilizing containerd as the chosen container runtime.

To foster flexibility and enable the use of any desired container engine [5], the Open Container Initiative (OCI) [6] was established. Docker contributed its libcontainer project to form the foundation for a new codebase called runC, supporting the OCI's objectives. Further details about runC can be found on GitHub [7].

The evolution of the container landscape [8] indicates a shift towards open specifications and flexibility. Developers are increasingly opting for vendor-neutral features, considering that Docker is now owned by Mirantis. Consequently, many vendors have explored alternative tools, including open-source options available on GitHub, to align with the trend towards openness and vendor neutrality.

By embracing open standards and leveraging vendor-neutral features, organizations can ensure future-proof and flexible container deployments. This aligns with the broader industry shift towards open specifications and allows for seamless integration with evolving container technologies.

1.2 Container Runtime Interface (CRI)

The Container Runtime Interface (CRI) [9] serves as a crucial component in enabling seamless integration between container runtimes and kubelet [10]. Its primary objective is to establish a standardized interface through protobuf [11] methods, API specifications, and libraries, enabling the smooth integration of new container runtimes into Kubernetes without requiring in-depth knowledge of kubelet internals.

As an evolving project, the CRI is currently undergoing active development, with significant progress being made. The successful integration of Docker-CRI has paved the way for the effortless addition and substitution of new container runtimes. Presently, CRI-O, rktlet [12], and frakti [13] are being actively worked on and considered as works-in-progress, further expanding the range of runtime options available [14].

The CRI plays a pivotal role in facilitating interoperability and enhancing the extensibility of Kubernetes, enabling organizations to leverage a diverse set of container runtimes based on their specific requirements. The ongoing development efforts and future integration of additional runtimes will continue to advance the capabilities and flexibility of the Kubernetes ecosystem.

1.3 Containerd: Enabling Low-Level Container Operations and Integration Capabilities

Containerd is purpose-built with a distinct objective in mind, focusing on providing a foundation of highly-decoupled low-level primitives instead of being a user-facing tool [15]. This unique approach has positioned it as a preferred choice among large cloud providers, thanks to its modular architecture [16] and minimal resource overhead. Complementary tools such as crictl [17], ctr [18], and nerdctl [19] are continually being developed to enhance the user experience [20].

By default, containerd leverages runC as the underlying runtime to execute containers, adhering to the specifications defined by the Open Container Initiative (OCI). Its design philosophy revolves around seamless integration into larger systems, enabling its integration into complex architectures effortlessly [21]. With a minimalist command-line interface (CLI) that emphasizes debugging and development, containerd caters to the requirements of integration and operation teams working on specialized products. It is particularly well-suited for scenarios that demand fine-grained control over the low-level aspects of containers [22], making it an ideal choice for those involved in building tailored solutions rather than traditional build, ship, and run applications.

1.4 CRI-O: Enabling OCI-Compatible Runtimes for Kubernetes

CRI-O, currently in incubation as part of Kubernetes, leverages the Kubernetes Container Runtime Interface (CRI) to seamlessly integrate with OCI-compatible runtimes. Hence, the name CRI-O. With runC being the default runtime and support for Clear Containers, the project's overarching objective is to collaborate with any OCI-compliant runtime.

Despite being relatively newer compared to Docker or rkt, CRI-O has garnered significant backing from major vendors. Its appeal lies in its remarkable flexibility and compatibility, making it an attractive choice for container runtime operations in the Kubernetes ecosystem.

1.5 Docker

Introduced in 2013, Docker revolutionized the landscape of containerized applications, providing an effortless way to package, deploy, and utilize them. It quickly became the go-to option for production environments due to its user-friendly nature. The availability of Docker Hub, an open image registry, simplified the process of downloading and deploying images created by vendors or individuals across various architectures, all within a single, intuitive toolkit. This simplicity made Docker the logical default choice for developers as well. However, challenges arose with frequent updates and stakeholder interactions, causing major vendors to distance themselves from Docker shortly after its integration into Mirantis.

Over the past few years, Docker has continued to expand its capabilities, introducing features like Swarm [23], their own orchestration solution, to address critical production needs. Nevertheless, these additional features resulted in increased vendor lock-in and product complexity. Consequently, the industry witnessed the emergence of alternative open tools and specifications such as CRI-O. Developers anticipating the future are advised to primarily embrace open tools for containers and Kubernetes, recognizing that Docker currently remains the preferred production tool in non-Red Hat environments [24].

1.6 rkt

The rkt runtime, commonly known as "rocket," offers a command-line interface (CLI) for executing containers. Initially introduced by CoreOS in 2014, it is now a member of the Cloud Native Computing Foundation [25] project ecosystem. Taking lessons from early Docker challenges, rkt places a strong emphasis on enhanced security, openness, and interoperability. Many of its functionalities have been addressed and improved upon by Docker over time. While not a straightforward drop-in substitute for Docker, rkt has made significant progress. It adheres to the appc specification [26] and supports the execution of Docker, appc, and OCI images. It deploys immutable pods. The project has garnered substantial attention and was expected to emerge as the primary replacement for Docker until CRI-O joined the official Kubernetes Incubator.

2. Containerizing an Application

When containerizing an application, it is important to consider its suitability for containerization [27]. Applications that are stateless and transient tend to work well in containers. Additionally, environmental configurations should be separated and managed using tools such as ConfigMaps [28] and secrets [29]. The goal is to develop the application into a single build artifact that can be deployed across various environments without modification, utilizing

decoupled configuration instead. In some cases, legacy applications are transformed into a collection of objects and artifacts distributed among multiple containers.

While Docker has been the dominant industry standard for containerization, prominent companies like Red Hat are transitioning to alternative open source tools. Although these tools are currently emerging, there is a possibility that they will become the new industry standard in the future.

3. Rewriting Legacy Applications: Making the Decision

When considering the migration of legacy applications to Kubernetes, one crucial decision to make is whether to containerize the application as it is or opt for a complete rewrite as a transient and decoupled microservice [30]. Rewriting legacy applications can be a time-consuming and costly process, but it offers the opportunity to leverage the flexibility and benefits of Kubernetes.

To aid in the decision-making process for containerizing legacy applications, several key factors should be taken into account [31]. First, the nature of the application plays a significant role. Stateless and transient applications are generally more suitable for containerization. Applications with heavy reliance on persistent state or tightly coupled components may present challenges in a containerized environment.

Another factor to consider is the desired scalability and agility of the application. Containerization allows for efficient scaling and deployment of microservices, enabling rapid iterations and updates. If the goal is to achieve greater scalability and agility, a rewrite as decoupled microservices may be the preferred approach [32].

Additionally, the overall complexity and maintenance requirements of the legacy application should be evaluated. Containerizing a complex monolithic application may not yield the full benefits of Kubernetes, while a simplified and modular architecture can maximize the advantages of containerization.

Ultimately, the decision to containerize a legacy application should be based on a careful assessment of the application's characteristics, scalability needs, and long-term goals. Balancing the cost and effort of rewriting against the potential benefits of Kubernetes will help determine the most appropriate approach for each specific case.

Specific representation of the equation for making the decision to containerize legacy applications:

Decision to Containerize Legacy Application = f (Nature of Application

,Scalability and Agility Goals

,Complexity and Maintenance Requirements) (1)

Decision to Containerize Legacy Application=f(Nature of Application,Scalability and Agility Goals,Complexity and Maintenance Requirements)

This equation captures the relationship between the decision to containerize a legacy application and the key factors that should be considered, including the nature of the application, the scalability and agility goals, and the complexity and maintenance requirements.

4. Crafting Your Dockerfile: A Step-by-Step Guide to Containerizing Your Application

To create the Dockerfile for containerizing your application, follow these steps:

Create a directory to store your application files. This directory will be used by the docker build process to pull all the necessary files when creating the image. Move the scripts and files for the containerized application into this directory.

Within the directory, create a file named Dockerfile. Note that the name must be exactly "Dockerfile" (with a capital "D"). In newer versions of Docker, you have the option to use a different filename by specifying it with the -f <filename> flag. The Dockerfile is essential for the docker build process to understand how to construct the image.

Each instruction in the Dockerfile is executed by the Docker daemon in sequence. While the instructions are not case-sensitive, it is common to write them in uppercase for clarity. The Dockerfile should start with the FROM instruction, which declares the base image for the container. Following that, you can include a series of instructions such as ADD, RUN, and CMD to add resources and run commands within the image.

Test the image by ensuring it is listed among other images using the docker images command. You can also execute the image using docker run <app-name> to verify that the application functions as expected.

Once you are satisfied with the image's performance, you can push it to a local repository on Docker Hub. Before doing so, create an account on Docker Hub. To push the image, use the command docker push followed by the repository name.

Here is a summary of the process:

Create the Dockerfile.

- Build the container using the command: sudo docker build -t <image-name> .

- Verify the image is listed among other images using: sudo docker images
- Test the image by running it: sudo docker run <image-name>
- Push the image to the repository on Docker Hub using: sudo docker push <repository-name>

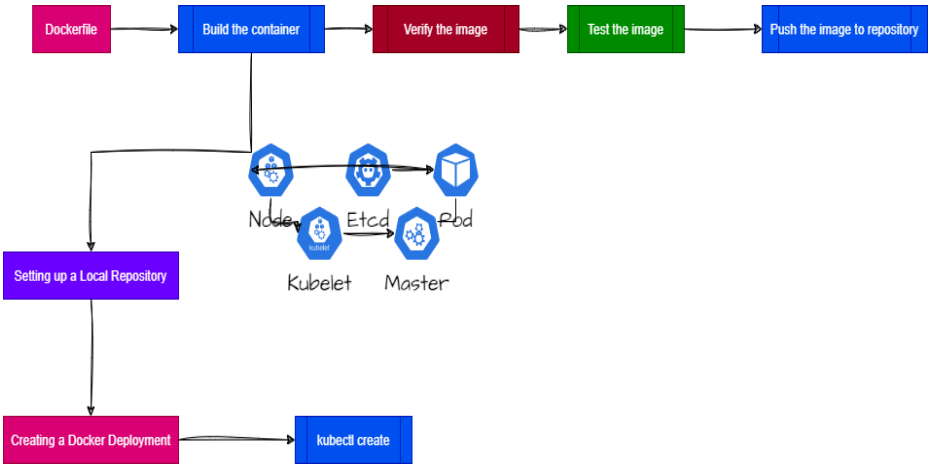

Figure 1. Setting up a Docker File and Kubectl Create

5. Setting up a Local Repository: Balancing Privacy and Accessibility

In addition to Docker Hub, you have the option to host a local repository for your Docker images. This alternative approach provides greater control over accessibility and enhances privacy and security [32]. Although it may require some administrative efforts, it offers benefits such as reduced bandwidth consumption and the ability to maintain a private image registry.

To set up a local repository, you need to configure the repository environment first. Once configured, you can use the docker tag command to assign appropriate tags to your local images and subsequently push them to the repository. It is advisable to initially set up an insecure repository for testing purposes, and once confirmed, you can configure secure TLS access to bolster security.

By hosting a local repository, you can strike a balance between accessibility and privacy, ensuring that your Docker images remain within your controlled environment.

6. Creating a Docker Deployment

After successfully pushing and pulling images using tools like podman [33], crictl [34], or docker, the next step is to deploy your application within Kubernetes. This involves running a new deployment using the desired image. The deployment command requires specifying the repository, application name, and version as arguments.

To test the deployment, you can use the kubectl create command, providing the deployment name and image details:

```
kubectl create deployment <deployment-name> --image=<repository>/<image-name>:<version>

kubectl create deployment myapp-deployment --image=myregistry/myapp:1.0
```

In this example, we are using the kubectl create command to create a deployment named my-app. The deployment is based on the Docker image my-registry/my-app with the latest tag. This command will create the deployment and start running the containerized application.

Note: Please replace my-registry with the appropriate registry URL and my-app with the desired application name and version according to your specific setup.

Verifying Pod Status and Testing Application Performance

Once the deployment is created, it is important to ensure that the Pod is running and that all containers within the Pod are also running. You can use the kubectl get pods command to check the status of the Pod and verify that all containers are in the "Running" state.

After confirming the Pod's status, it is crucial to test the application's performance and functionality. This may involve running tempest and quality assurance (QA) tests specific to your application. By executing these tests, you can verify that the application is operating as expected and meeting the desired performance criteria.

To further validate the transient nature of the application, you can terminate the Pod intentionally. By doing so, you can observe how the application handles the Pod's termination and whether it behaves as designed. This test ensures that the application can gracefully handle the loss of a Pod and demonstrates its transient nature.

By following these steps, you can effectively verify the running status of the Pod, assess the application's performance, and test its resilience in the face of Pod termination.

7. Executing Commands in a Container

During the testing phase, it may be necessary to run specific commands within a Pod to perform various tasks. The availability of commands depends on the base environment included in the image during its creation. In order to maintain a decoupled and lightweight design, it is possible that certain shells or tools may not be available by default. As a result, it may be necessary to revisit the image build process and include any additional resources required for testing or production purposes.

To run commands interactively within a Pod, the -it options can be used to enable an interactive shell instead of a command running without interaction or access. If there are multiple containers within the Pod, it is important to specify the desired container when executing commands. This can be achieved using the following command:

```
kubectl exec -it <Pod-Name> -- /bin/bash
```

By including the appropriate Pod name and specifying the container using the -- flag, you can access an interactive shell within the desired container for executing commands and performing further testing or troubleshooting.

8. Functionality with Multi-Container Pods

In some cases, it may not be necessary to recreate an entire image to add additional functionality, such as a shell or a logging agent. Instead, you can leverage the flexibility of Kubernetes by adding another container to the existing Pod, which will provide the required tools or capabilities.

It is important to ensure that each container within the Pod remains transient and decoupled, allowing for scalability and maintaining the intended nature of the original application. However, if adding another container limits the scalability or transient characteristics of the original application, it may be necessary to consider a new build or design approach.

It is crucial to understand that every container within a Pod shares the same IP address and namespace. Additionally, each container has equal access to the storage allocated to the Pod. As Kubernetes does not provide any inherent locking mechanisms, it is important to design your configuration or application in a way that prevents conflicts between containers attempting to write to shared resources.

One possible approach is to designate one container as read-only while the other container performs write operations. You can configure the containers to write to different directories within the shared volume, or the application itself can incorporate built-in locking mechanisms. Without these protective measures, there would be no way to guarantee the order of write operations or prevent conflicts between containers accessing the shared storage.

It is worth noting that the terms "ambassador" [35], "adapter" [36], and "sidecar" [37], are often used to describe different roles that secondary containers play within a multi-container Pod. However, it is important to understand that these terms are not specific settings or configurations within Kubernetes. Instead, they serve as expressions of the intended purpose or functionality of the additional containers. Ultimately, all these containers are part of a multi-container Pod, working together to fulfill the requirements of the application or system.

Refining Terminology for Multi-Container Pods

Within the context of multi-container Pods [38], there are several common terms used to describe the roles and functionalities of secondary containers:

Ambassador:

The ambassador container serves as a communication bridge between the primary container and external resources, typically located outside the Kubernetes cluster. By utilizing a proxy like Envoy, it provides the flexibility to embed a custom proxy instead of relying on the cluster-provided one. This approach can be beneficial when there is uncertainty about the cluster configuration or when specific communication requirements need to be met.

Adapter:

An adapter container is responsible for modifying or adapting the data generated by the primary container. It is particularly useful when dealing with data formats that differ from the standard conventions. For instance, adapting data streams to align with specific formatting requirements, such as converting ASCII to a Microsoft-specific format, can be achieved using an adapter container.

Sidecar:

Drawing inspiration from the sidecar attached to a motorcycle, a sidecar container does not provide the primary application's core functionality but offers additional services or capabilities that complement the primary container. Common examples of sidecar containers include logging containers, which capture and process log data generated by the primary application. By using a sidecar container, you can enhance the functionality and operational aspects of the primary container without directly modifying it.

It is important to note that these terms—ambassador, adapter, and sidecar—do not represent specific settings or configurations within Kubernetes itself. Rather, they are descriptive expressions used to define the purpose and role of secondary containers within a multi-container Pod. Each of these containers collaborates to fulfill the requirements and enhance the overall functionality of the application or system.

9. Refining Terminology for Multi-Container Pods

Within the context of multi-container Pods [38], there are several common terms used to describe the roles and functionalities of secondary containers:

Ambassador:

The ambassador container serves as a communication bridge between the primary container and external resources, typically located outside the Kubernetes cluster. By utilizing a proxy like Envoy, it provides the flexibility to embed a custom proxy instead of relying on the cluster-provided one. This approach can be beneficial when there is uncertainty about the cluster configuration or when specific communication requirements need to be met.

Adapter:

An adapter container is responsible for modifying or adapting the data generated by the primary container. It is particularly useful when dealing with data formats that differ from the standard conventions. For instance, adapting data streams to align with specific formatting requirements, such as converting ASCII to a Microsoft-specific format, can be achieved using an adapter container.

Sidecar:

Drawing inspiration from the sidecar attached to a motorcycle, a sidecar container does not provide the primary application's core functionality but offers additional services or capabilities that complement the primary container. Common examples of sidecar containers include logging containers, which capture and process log data generated by the primary application. By using a sidecar container, you can enhance the functionality and operational aspects of the primary container without directly modifying it.

It is important to note that these terms—ambassador, adapter, and sidecar—do not represent specific settings or configurations within Kubernetes itself. Rather, they are descriptive expressions used to define the purpose and role of secondary containers within a multi-container Pod. Each of these containers collaborates to fulfill the requirements and enhance the overall functionality of the application or system.

10. Type of probes : readinessProbe, livenessProbe, and startupProbe

In Kubernetes, container probes are essential mechanisms for monitoring the health and availability of containers within a Pod. There are three types of probes commonly used: readinessProbe, livenessProbe, and startupProbe [38].

readinessProbe:

The readinessProbe is a mechanism in Kubernetes that determines whether a container is ready to accept traffic. There are several types of readiness probes:

Exec Probe: This probe checks the container's readiness by executing a specific command inside the container. If the command returns a zero exit code, the container is considered ready. Otherwise, it is considered not ready.

HTTP Get Probe: This probe sends an HTTP GET request to a specified endpoint within the container. If the response code falls within the range of 200-399, the container is considered ready. Any other response code indicates failure.

TCP Socket Probe: With this probe, Kubernetes attempts to open a TCP socket to a specified port within the container. If the socket can be successfully opened, the container is considered ready.

livenessProbe:

The livenessProbe is used to check the health of a container continuously. If the container fails the liveness probe, it is terminated and replaced. Similar to the readiness probe, there are different types of liveness probes:

Exec Probe: This probe executes a command within the container and considers the container alive if the command returns a zero exit code. Otherwise, the container is considered dead and will be restarted.

HTTP Get Probe: In this probe, an HTTP GET request is sent to a specific endpoint. If the response code falls within the range of 200-399, the container is considered alive. Any other response code indicates failure, and the container will be restarted.

TCP Socket Probe: With this probe, Kubernetes attempts to open a TCP socket to a specified port within the container. If the socket can be successfully opened, the container is considered alive.

startupProbe:

The startupProbe, introduced recently, is used to test the startup readiness of an application that takes a significant amount of time to start. If the kubelet uses a startupProbe, it disables the liveness and readiness checks until the application passes the test. The duration for which the container is considered failed is determined by the failureThreshold multiplied by periodSeconds.

11. Verifying the kubernetes deployment

Testing is a crucial aspect of deploying applications in a Kubernetes environment [39]. With the decoupled and transient nature of containerized applications, it is important to ensure that the deployed application functions as expected and meets the requirements of end users. Building a comprehensive test suite for your application can help identify issues early in the development process and ensure smooth integration with Kubernetes.

While there are various testing methods and tools available, leveraging built-in functionalities provided by kubectl can be a good starting point. The "describe" command allows you to view detailed information about an object, including its conditions, volumes, and events. The events section can be particularly useful during testing as it provides a chronological view of cluster actions and associated messages.

In addition to "describe", the "logs" command can be used to retrieve logs from a specific container within a Pod. This can help in troubleshooting and identifying any issues related to application functionality or integration with Kubernetes.

While these built-in tools are helpful, it is often beneficial to develop custom testing tools tailored to your specific deployment. These tools can perform more comprehensive tests and validations, ensuring that the distributed application functions properly in the transient environment provided by Kubernetes.

To describe a specific Pod, such as naturelabs, you would run the following command:

```
kubectl describe pod naturelabs
```

The kubectl describe pod command and provide a general example of the output structure.od naturelabs

The output of the command would provide detailed information about the specified Pod, including its current status, conditions, events, and other relevant details. Here is a general example of the output structure:

```
Name:         naturelabs
Namespace:    default
...
Status:       Running
...
Conditions:
  Type          Status
  ---------- -------
  PodScheduled   True
  ...
Events:
  Type    Reason     Age              From               Message
  ----    ------     ----             ----               -------
  Normal  Scheduled  <timestamp>      default-scheduler  Successfully assigned default/naturelabs to node-1
  Normal  Pulled     <timestamp>      kubelet            Container image "naturelabs:latest" successfully pulled
  Normal  Created    <timestamp>      kubelet            Created container naturelabs
  Normal  Started    <timestamp>      kubelet            Started container naturelabs
```

As you progress with testing, it's important to examine the output of containers within a Pod. While some applications generate logs that provide valuable insights into their behavior, others may not produce any output by default. In such cases, it can be challenging to determine whether the absence of output indicates an error or a configuration issue.

To check the container logs within a Pod, you can use the kubectl logs command followed by the Pod name and optionally the container name if there are multiple containers within the Pod. For example:

```
kubectl logs <pod-name> [<container-name>]
```

Running this command will display the logs generated by the specified container(s). Analyzing these logs can help you identify any errors, warnings, or other relevant information that can aid in troubleshooting and understanding the behavior of your application [40].

In contrast, there are cases where a different Pod configuration may generate extensive log output. For instance, you can examine the logs of the etcd Pod in the kube-system namespace using the following command:

```
kubectl -n kube-system logs etcd-master
```

By executing this command, you can retrieve the logs produced by the etcd container within the etcd-master Pod. The log output can provide valuable information about the operational status, events, and activities related to the etcd component in your Kubernetes cluster.

Remember, you can use tab completion to auto-complete the Pod name based on the available options in your cluster environment.

```
2023-05-23 10:15:23.123 INFO: Starting etcd server...

2023-05-23 10:15:23.456 INFO: Configuring etcd cluster with 3 members...

2023-05-23 10:15:24.789 ERROR: Failed to connect to member 2: connection refused

2023-05-23  10:15:25.012 INFO: Successfully joined etcd cluster

2023-05-23  10:15:25.678 INFO: etcd server is now running and ready to accept requests
```

12. Helm: Simplifying Kubernetes Deployments

To streamline the deployment process and facilitate the management of Kubernetes objects, Helm [41], the package manager for Kubernetes, comes to the rescue. Helm utilizes a chart, which is a collection of YAML files, to deploy and manage one or more objects within a Kubernetes cluster. Before deploying a chart, you can customize its configuration by modifying the values.yaml file, allowing for flexible and dynamic installations.

One of the advantages of Helm is its ability to source charts from various locations. ArtifactHub [42], in particular, serves as a centralized platform for publishing and discovering charts, providing a convenient way to share and find Helm charts for different applications and services. With Helm, you can easily package, distribute, and deploy applications on Kubernetes, simplifying the overall deployment workflow.

13. Results and Conclusion

The research on Kubernetes and containerization has yielded significant insights into the evolving container landscape and the role of Kubernetes as a powerful orchestration tool. The study focused on the adoption of open standards and vendor neutrality, aligning with the changing container ecosystem.

The Container Runtime Interface (CRI) in Kubernetes, particularly Containerd, has emerged as a popular container runtime due to its adherence to open specifications and flexibility. The shift towards open-source options and alternative tools reflects the industry's trend towards openness and vendor neutrality.

The decision to containerize legacy applications should be carefully evaluated, taking into account factors such as application nature, scalability needs, and complexity. Organizations are transitioning from Docker as the dominant industry standard to alternative open-source tools that align with open specifications.

The containerization process, including creating Dockerfiles, building and testing containers, and setting up local repositories, balances privacy and accessibility while enhancing control and security.

Deploying applications within Kubernetes involves creating deployments, verifying Pod status, and testing application performance. Additional considerations include executing commands within containers, utilizing multi-container Pods to enhance functionality, and implementing container probes for health monitoring.

Refining the terminology for multi-container Pods, with terms like ambassador, adapter, and sidecar, helps describe the roles and functionalities of secondary containers within the application ecosystem.

A comprehensive testing suite is essential to verify the functionality of deployed applications in a Kubernetes environment. Leveraging built-in functionalities provided by kubectl and developing custom testing tools tailored to specific deployments can ensure optimal application performance and reliability.

The lab exercise outlined in the research provides participants with hands-on experience in implementing and testing Kubernetes probes, container image management, and multi-container Pod design patterns. By completing the exercise, participants enhance their knowledge and skills in working with Kubernetes clusters and gain confidence in deploying containerized applications.

In conclusion, the research highlights the importance of embracing openness and vendor neutrality in the container ecosystem, the significance of container runtimes like Containerd, and the considerations involved in containerizing legacy applications and deploying them within Kubernetes. The study provides practical insights and recommendations for professionals and organizations navigating the container landscape and harnessing the power of Kubernetes.

Acknowledgements

Nature Labs, Non-profit Organization, deserves our sincere gratitude for their invaluable support and sponsorship in the research and development of the Kubernetes Proof of Concept for "Deploying a Simple Python Application." Their assistance has been instrumental in making this project a success.

We would also like to express our appreciation to the entire team at Nature Labs for their guidance, expertise, and resources that have greatly contributed to the attractive implementation of Kubernetes and its associated functionalities.

Additionally, we would like to extend our thanks to all the individuals who participated in this project, providing their insights, feedback, and assistance throughout the development process.

Appendix A. Proof of Concept for Kubernetes : Deploying a Simple Python Application

In this PoC, we will configure a local repository named "naturelabs-research" to store Docker images within our Kubernetes cluster. Follow these steps to set up the local repository:

Create the local repository by applying the "naturelabs-research.yaml" file provided in the course tarball. Use the following command to create the repository:

```
kubectl apply -f naturelabs-research.yaml
```

Take note of the ClusterIP assigned to the "naturelabs-research" service. Use the following command to retrieve the ClusterIP:

```
kubectl get svc | grep naturelabs-research
```

Verify the functionality of the repository. Replace the IP address and port in the command below with the ClusterIP of your local repository service:

```
curl <local_repo_ip>:<port>/v2/_catalog
```

Configure the container engine (e.g., Podman) to work with non-TLS repositories. This step may vary depending on the container engine you are using. Refer to the documentation of your container engine for the specific steps required.

Download a Docker image from a public registry (e.g., Docker Hub) and tag it with the IP and port of the local repository. Use the following commands to pull the image and tag it:

```
sudo podman pull <image_name>
sudo podman tag <image_name> <local_repo_ip>:<port>/<image_name>
```

By following these steps, you will have successfully configured a local repository named "naturelabs-research" and tagged a Docker image for use within your Kubernetes cluster.

```
sudo podman pull <image_name>
sudo podman tag <image_name> <local_repo_ip>:<port>/<image_name>
```

Configuring Probes in Kubernetes (naturelabs-research.yaml)

In this exercise, we will configure readiness and liveness probes for a Kubernetes deployment to ensure the availability and ongoing health of the application. Follow these steps:

Open the "naturelabs-research.yaml" file in a text editor.

Add the stanza for a readinessProbe under the containers section of the YAML file. Ensure proper indentation and formatting. The readinessProbe should include the following specifications:

```
readinessProbe:
  exec:
    command:
    - cat
    - /tmp/healthy
  periodSeconds: 5
```

Save the file.

Delete the existing deployment named "try1":

```
Kubectl delete deployment try1
```

Recreate the deployment using the updated "naturelabs-research.yaml" file:

```
kubectl create -f naturelabs-research.yaml
```

Verify the status of the newly created pods. They should show zero available because the "/tmp/healthy" file is missing:

```
kubectl get pods
```

Investigate the pods using the describe and logs commands to identify any issues. Note that there may be no logs generated yet.

Choose one of the pods and create the health check file ("/tmp/healthy"):

```
kubectl exec -it <pod-name> -- /bin/bash
touch /tmp/healthy
exit
```

Wait for at least five seconds, then check the pods again. The pod with the /tmp/healthy file should show as available (1/1 READY), while the others will continue to show as not ready (0/1 READY):

```
kubectl get pods
```

To configure a livenessProbe, edit the "naturelabs-research.yaml" file again. Add a livenessProbe section under the containers section, similar to the readinessProbe stanza. Indent the livenessProbe section accordingly.

Delete and recreate the deployment using the updated YAML file:

```
kubectl delete deployment try1
kubectl create -f naturelabs-research.yaml
```

View the newly created pods. You will notice that each pod now has two containers, and only one is running initially.

Use a for loop to create the health check file ("/tmp/healthy") for each container in the pods:

```
for name in $(kubectl get pod -l app=try1 -o name); do
kubectl exec $name -c simpleapp -- touch /tmp/healthy;
done
```

Wait for a minute or so for the liveness probes to check for the file and the health checks to succeed. Each pod should show both containers running (2/2 READY).

Verify the status of the containers in the pods:

```
kubectl get pods
kubectl describe pod <pod-name> | grep -E 'State|Ready'
```

By following these steps, you have successfully configured readiness and liveness probes in Kubernetes using the "naturelabs-research" deployment. This ensures that the application becomes available only when the readiness test is met and remains in a healthy state with continuous liveness checks.

The output of the "kubectl get pods" command upon completing exercise with the configured probes will vary based on the current state of the pods. However, assuming all the steps were followed correctly, the output should resemble the following:

```
NAME                      READY   STATUS    RESTARTS   AGE
nginx-6b58d9cdfd-g7lnk    1/1     Running   1          13h
registry-795c6c8b8f-7vwdn 1/1     Running   1          13h
try1-76cc5ffcc6-4rjvh     2/2     Running   0          3s
try1-76cc5ffcc6-bk5f5     2/2     Running   0          3s
try1-76cc5ffcc6-d8n5q     2/2     Running   0          3s
try1-76cc5ffcc6-mm6tw     2/2     Running   0          3s
try1-76cc5ffcc6-r9q5n     2/2     Running   0          3s
try1-76cc5ffcc6-tx4dz     2/2     Running   0          3s
```

The output of the "kubectl get pods" command after completing Exercise with the configured probes will vary based on the current state of the pods. However, assuming all the steps were followed correctly, the output should resemble the following:

The output displays the names of the pods and their corresponding readiness status, container status, restart counts, and age. The "READY" column shows the number of containers ready out of the total containers in each pod. In this case, all the try1 pods should show as 2/2, indicating that both containers within them are running and ready.

A.1. Lab: Implementation and Testing of Kubernetes Probes, Container Images, and Multi-Container Pods Purpose of the lab exercise:

This research paper presents a lab exercise focused on the implementation and testing of various aspects related to Kubernetes, including probes and health checks, container image management, and multi-container Pod design patterns. The exercise aims to enhance participants' understanding and practical skills in working with Kubernetes clusters and deploying containerized applications.

Use Case of the lab:

Participants will gain hands-on experience in implementing probes and health checks, defining, building, and modifying container images, understanding multi-container Pod design patterns, and utilizing container logs.

Objectives of lab exercise:

The main objectives of this lab exercise are as follows:

Locate and bookmark working YAML examples for LivenessProbes, ReadinessProbes, and multi-container pods using the provided exam-approved URL locations.

Deploy a new nginx webserver and configure LivenessProbe and ReadinessProbe on port 80. Verify the functionality of both probes and the webserver.

Utilize the provided build-review1.yaml file to create a non-working deployment. Troubleshoot and fix the deployment to ensure that both containers are running and in a READY state. The web server should listen on port 80, and the proxy should listen on port 8080.

Access the default page of the web server and verify the GET activity logs in the container log. The log message should resemble the following format:

"IP - - [Date/Time] "GET / HTTP/1.1" 200 612 "-" "User-Agent" "-"".

Remove any resources created during the lab exercise to maintain a clean environment.

Methodology of lab:

Participants are required to follow these steps to complete the lab exercise:

Use the exam-approved URL locations to find and bookmark working YAML examples for LivenessProbes, ReadinessProbes, and multi-container pods.

Deploy a new nginx webserver using Kubernetes manifests and configure LivenessProbe and ReadinessProbe on port 80. Perform tests to ensure the proper functioning of both probes and the webserver.

Utilize the provided build-review1.yaml file to create a deployment that initially fails. Analyze the deployment issues, troubleshoot, and modify the configuration to ensure that both containers are running and in a READY state. Verify that the web server listens on port 80, and the proxy listens on port 8080.

Access the default page of the deployed web server to confirm its availability. Additionally, inspect the container logs to verify the GET activity logs. Take note of the log message format, considering potential variations in IP address and timestamp.

Clean up the environment by removing any resources created during the lab exercise to avoid clutter and ensure a fresh start for subsequent activities.

References

1. Clement, J., Charland, P., & Kempter, R. (2019). Kubernetes: A Survey on the Security of an Open-Source Orchestration System for Container-Based Distributed Applications. 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (pp. 824-831). DOI: 10.1109/DSN.2019.00072
2. Maggi, F., Zanero, S., & Balduzzi, M. (2016). Docker Containers as Software Security Containment Solutions. IEEE 24th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS) (pp. 80-89). DOI: 10.1109/MASCOTS.2016.45
3. Gupta, A., & Dubey, V. (2017). Resource management in Docker-based containers. 4th IEEE Uttar Pradesh Section International Conference on Electrical, Computer and Electronics (UPCON) (pp. 372-377). DOI: 10.1109/UPCON.2017.8250673
4. Crosby, M., McGowan, D., Day, S., Batts, V., & Estes, P. (2017). containerd: An industry-standard container runtime. ACM Symposium on Cloud Computing (SoCC). DOI: 10.1145/3127479.3128614
5. Murdaca, A., Scrivano, G., Batts, V., & Liu, L. (2016). CRI-O: A lightweight and secure container runtime for Kubernetes. IEEE International Conference on Cloud Engineering (IC2E) (pp. 65-74). DOI: 10.1109/IC2E.2016.32
6. Open Container Initiative (OCI) - Official website: https://opencontainers.org/
7. runC on GitHub - Official GitHub repository for the runC project: https://github.com/opencontainers/runc
8. Lambertz, M., & Kapitza, R. (2019). Container runtime analysis: A comparison of container runtimes in Kubernetes. IEEE International Conference on Cloud Computing Technology and Science (CloudCom) (pp. 392-397). DOI: 10.1109/CloudCom2019.00070
9. Kubernetes SIG-Node. (2021). The Container Runtime Interface (CRI). Retrieved from https://github.com/kubernetes/community/blob/master/sig-node/container-runtime-interface.md
10. Kubernetes. (2021). kubelet: The Kubernetes Node Agent. Retrieved from https://kubernetes.io/docs/concepts/architecture/kubelet/
11. Protocol Buffers - Google Developers. (2021). Protocol Buffers. Retrieved from https://developers.google.com/protocol-buffers
12. Kubernetes. (2021). CRI-O. Retrieved from https://github.com/cri-o/cri-o
13. Kubernetes. (2021). rktlet. Retrieved from https://github.com/kubernetes-incubator/rktlet
14. Crosby, M., McGowan, D., Day, S., Batts, V., & Estes, P. (2017). containerd: An industry-standard container runtime. In ACM Symposium on Cloud Computing (SoCC). DOI: 10.1145/3127479.3128614
15. Docker. (2021). containerd: An industry-standard container runtime. Retrieved from https://containerd.io/
16. Kubernetes. (2021). crictl. Retrieved from https://github.com/kubernetes-sigs/cri-tools/tree/master/cmd/crictl
17. containerd. (2021). ctr. Retrieved from https://github.com/containerd/containerd/tree/main/cmd/ctr
18. nerdctl. (2021). nerdctl. Retrieved from https://github.com/containerd/nerdctl

19. Docker. (2021). Docker Documentation. Retrieved from https://docs.docker.com/
20. Crosby, M., McGowan, D., Day, S., Batts, V., & Estes, P. (2017). containerd: An industry-standard container runtime. In ACM Symposium on Cloud Computing (SoCC). DOI: 10.1145/3127479.3128614
21. containerd. (2021). containerd GitHub repository. Retrieved from https://github.com/containerd/containerd
22. Docker. (2021). Docker Swarm: Native clustering for Docker. Retrieved from https://docs.docker.com/engine/swarm/
23. Deisz, C. (2018). Learning Docker: Build, Ship, and Scale Faster (2nd ed.). Packt Publishing, ISBN: 9781786462923, O'Reilly Media.
24. Cloud Native Computing Foundation (CNCF). (2021). rkt. Retrieved from https://www.cncf.io/projects/rkt/
25. App Container (appc) Specification. (2021). Retrieved from https://github.com/appc/spec
26. Kamat, S. (2019). Getting Started with Containerization: Concepts and Practices. O'Reilly Media.
27. Kubernetes. (2021). ConfigMaps. Retrieved from https://kubernetes.io/docs/concepts/configuration/configmap/
28. Kubernetes. (2021). Secrets. Retrieved from https://kubernetes.io/docs/concepts/configuration/secret/
29. Newman, S. (Feb 2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, ISBN: 9781491950357
30. Pahuja, V. (2017). Developing with Docker. Packt Publishing. ISBN: 978-1786469908
31. Gabriel N. Schenker, Hideto Saito, Hui-Chuan Chloe Lee, et.al (March 2019). Getting Started with Containerization. Packt Publishing, ISBN: 9781838645700
32. Podman: A tool for managing OCI containers and pods. Retrieved from https://podman.io/
33. CRICTL: Kubernetes Container Runtime Interface Command Line Tool. Retrieved from https://github.com/kubernetes-sigs/cri-tools
34. Orendain, L. (2020). What is an Ambassador in Kubernetes? Retrieved from https://www.nirmata.com/2020/08/13/what-is-an-ambassador-in-kubernetes/
35. Fogel, J. (2020). What Is a Kubernetes Adapter and How Does It Work? Retrieved from https://www.upbound.io/blog/kubernetes-adapters/
36. Buurman, S. (2020). What is a Sidecar in Kubernetes? Retrieved from https://www.replex.io/blog/what-is-a-sidecar-in-a-kubernetes-based-microservices-architecture
37. Kubernetes Documentation. (2020). Container Probes. Retrieved from https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/
38. Kubernetes. (n.d.). Testing and Debugging. Retrieved from https://kubernetes.io/docs/concepts/testing-debugging/
39. Kubernetes. (n.d.). Viewing Pod Logs. Retrieved from https://kubernetes.io/docs/concepts/cluster-administration/logging/#viewing-pod-logs
40. Helm. (n.d.). Helm - The Kubernetes Package Manager. Retrieved from https://helm.sh/
41. ArtifactHub. (n.d.). ArtifactHub - Discover, release and deploy Kubernetes packages. Retrieved from https://artifacthub.io/
42. Van der Geer, J., Hanraads, J. A. J., & Lupton, R. A. (2000). The art of writing a scientific article. Journal of Science Communication, 163, 51–59.
43. Strunk, W., Jr., & White, E. B. (1979). The elements of style (3rd ed.). New York: MacMillan