appviewX

Install and Upgrade - Technical Whitepaper Kubernetes Adoption Guide

Version: 2021.1.0

Copyright AppViewX, Inc.

Copyright © 2021 AppViewX, Inc. All Rights Reserved.

This document may not be copied, disclosed, transferred, or modified without the prior written consent of AppViewX, Inc. While all content is believed to be correct at the time of publication, it is provided as general-purpose information. The content is subject to change without notice and is provided “as is” and with no expressed or implied warranties whatsoever, including, but not limited to, a warranty for accuracy made by AppViewX. The software described in this document is provided under written license only, contains valuable trade secrets and proprietary information, and is protected by the copyright laws of the United States and other countries. Unauthorized use of software or its documentation can result in civil damages and criminal prosecution.

Trademarks

The trademarks, logos, and service marks displayed in this manual are the property of AppViewX or other third parties. Users are not permitted to use these marks without the prior written consent of AppViewX or such third party which may own the mark.

External Reference Links

This product includes software developed by the CentOS Project (www.centos.org).

This product includes software developed by Red Hat, Inc. (www.redhat.com).

This product includes software developed by VMware, Inc. (www.vmware.com).

All other trademarks mentioned in this document are the property of their respective owners.

Contact Information

AppViewX, Inc.

222 Broadway, FL 19

New York, NY 10038

Email: info@appviewx.com

Web: www.appviewx.com

Contents

Chapter 1. Overview **5**

Chapter 2. Transition to Kubernetes **6**

Transition to Kubernetes 6

Auto scaling 6

Resiliency 6

Security 6

Chapter 3. How did we achieve it? **7**

Architecture 7

Zero Trust Network 8

Service Mesh 10

Autoscaling Based on Custom Metrics 10

Cluster Management Best Practices 10

Tools 11

Chapter 4. Complex Use Cases **12**

Handling Graceful Shutdown of Pods 12

Datacenter Based Routing 13

Chapter 5. Outcome **14**

Chapter 6. References **15**

Copyright © 2021 [AppViewX, Inc]. All Rights Reserved.

Preface

Revision History

Revision	Description	Date
1.0	Initial release of document for Release 2021.1.0	Sept 2021

About this Guide

This guide outlines the kubernetes adoption guide.

Text Conventions

The following text conventions are used in this document:

Convention	Description
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`codeblock`	Indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Chapter 1: Overview

As a company, we always ventured into new innovative ideas for our product and the technical stack that we choose to delight and serve the customer better. In order to deliver fast with quality, the product was implemented based on microservice architecture. The customer infrastructure boundary between on-premise and cloud is growing thin day by day. To cater to the on-premise, hybrid model or ever-growing cloud and to better make use of the resources and security at the forefront, we embraced the Kubernetes to orchestrate our containerized workloads.

Copyright © 2021 [AppViewX, Inc]. All Rights Reserved.

Chapter 2: Transition to Kubernetes

- Transition to Kubernetes

Transition to Kubernetes

The main motivation behind adoption of Kubernetes is to make better use of the resources, resiliency to the services and achieve higher security by adopting a zero trust network model.

- Auto scaling
- Resiliency
- Security

Auto scaling

AppViewX services can have a custom throttling capability based on pre-configured memory configuration per API. This will enable AppViewX services to utilize (scale up) resources optimally as the demand surges and scale down when not in use. This will help to horizontally scale the vendor components on demand and optimize the resource usage.

Resiliency

There is no guarantee that the services will run without any interruption and they are bound to failure. Kubernetes keeps deployments healthy by restarting containers that have failed, killing and replacing unresponsive containers based on health checks. This helps to mitigate the common pain point of the application upkeep process.

Security

AppViewX architecture is designed around the concept of zero trust network model to enforce tighter security within the Kubernetes cluster. This means no one is trusted by default and required verification to gain access to the services.

Copyright © 2021 [AppViewX, Inc]. All Rights Reserved.

Chapter 3: How did we achieve it?

- Architecture
- Zero Trust Network
- Service Mesh
- Autoscaling Based on Custom Metrics
- Cluster Management Best Practices
- Tools

Architecture

AppViewX is designed based on microservice architecture, making it easier to move to containerized workloads and the containers being orchestrated using Kubernetes. The following diagram depicts the deployment architecture:

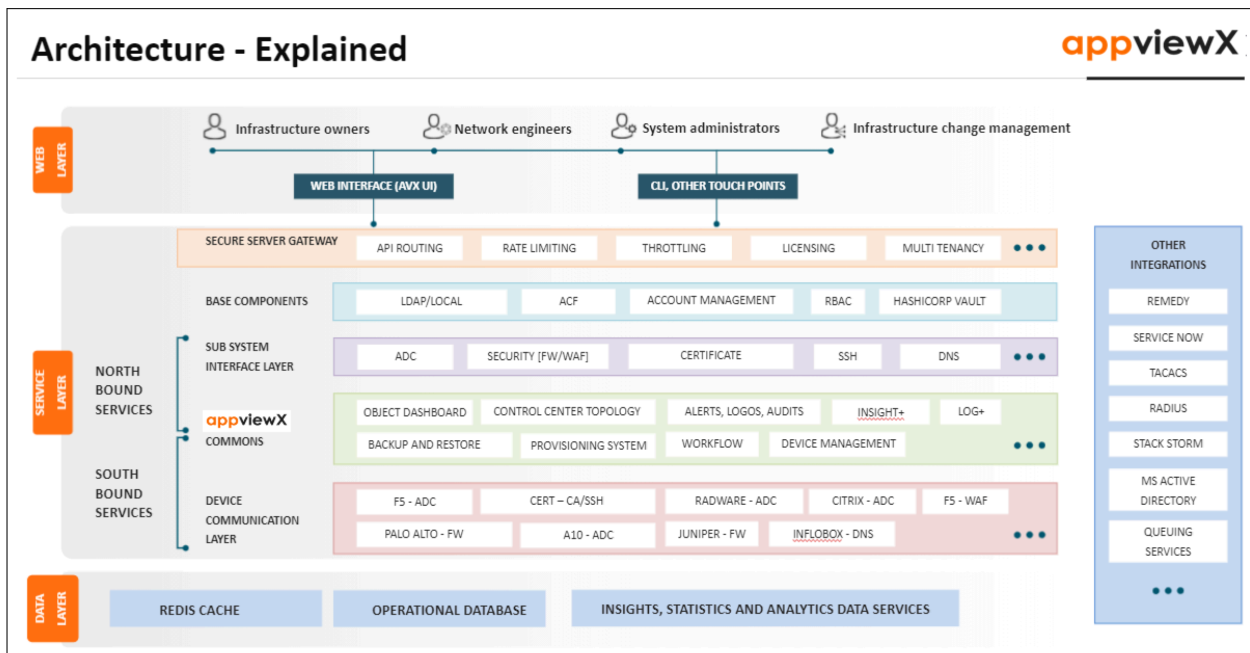

Figure 1 Deployment Architecture

In the diagram:

Copyright © 2021 [AppViewX, Inc]. All Rights Reserved.

- **Presentation/ Web Layer** - houses the AppViewX user interface related files and interacts with the service layer
- **Service Layer** - contains the Northbound and Southbound services that can be further classified into:
  - **Business Layer**:
    - Houses AppViewX specific business logic
    - Interacts with the Data layer for persisting the input data
  - **Device Communication Layer**:
    - Low code
    - Stateless layer
    - Routes communication to the respective vendor through APIs or SSH
    - Houses vendor specific business logic
- **Data Layer**:
  - Houses data persistence and retrieval logic
  - Redis caching is available

Zero Trust Network

AppViewX architecture is designed around the concept of **zero trust network** model. Zero trust network, refers to security concepts and threat model that no longer assumes that actors, systems or services operating from within the security perimeter should be automatically trusted, and instead must verify anything and everything trying to connect to its systems before granting access.

We use Calico as CNI for Kubernetes networking. The zero trust network model is enforced using Calico's network policy. By default, the network policy is applied across AppViewX components to enforce zero trust network policy. The policies are configured to allow traffic from only the intended source and rest will be rejected.

The following diagram depicts the traffic flow between the components:

Copyright © 2021 [AppViewX, Inc]. All Rights Reserved.

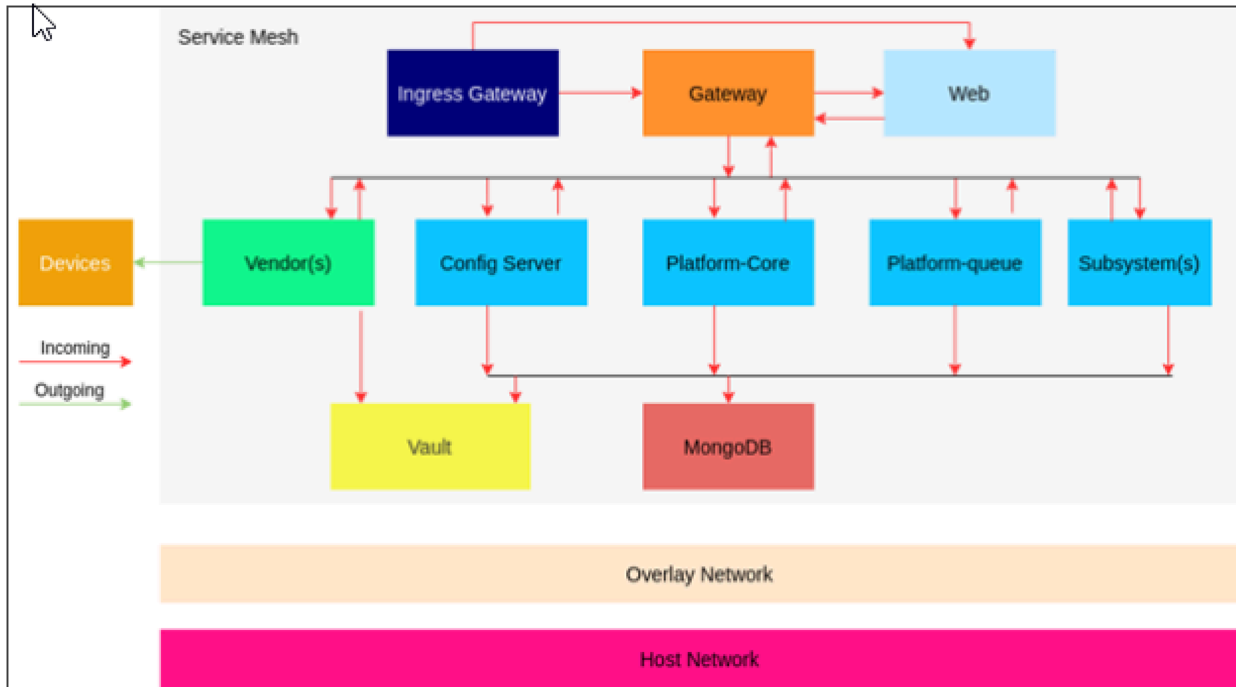

Figure 2 Zero trust network model

The table depicts the **ingress network policy** that has been configured between components to achieve zero trust.

	Platform Core	Platform Queue	Config Server	Service 1	Service 2	Gateway	Web
Database							
Vault							
Platform Core							
Platform Queue							
Config Server							
Service 1							
Service 2							

- **Green** - Ingress traffic (Incoming) to the component allowed
- **Red** - Ingress traffic (Incoming) to the component denied

Copyright © 2021 [AppViewX, Inc]. All Rights Reserved.

Service Mesh

The service mesh is typically implemented as a scalable set of network proxies deployed alongside application code (a pattern sometimes called a sidecar). These proxies handle the communication between the microservices and also act as a point at which the service mesh features can be introduced. The proxies comprise the service mesh's data plane, and are controlled as a whole by its control plane.

We leverage ISTIO's service mesh capabilities in the following areas of our deployment:

- To secure the service-to-service communication at the network layer. By default, **mTLS** is enabled and pod-to-pod traffic is completely encrypted.
- **Ingress controller** for routing traffic to the cluster
- Auto scaling of vendor pod using ISTIO traffic metrics

Autoscaling Based on Custom Metrics

AppViewX services can have a custom throttling capability based on pre-configured memory configuration per API. This is configured by adding the flag SLA-STRATEGY and setting its value to “memory”. This will throttle the incoming requests based on active execution and when the limit breaches, http status code 429 will be returned. This status will be captured by HPA and more pods will be scaled as per resource availability.

This will enable AppViewX services to utilize (scale up) resources optimally as the demand surges and scale down when not in use. To put it in context, during midnight config fetch, the vendor components are in demand and will be scaled up to occupy all the available resources (CPU and Memory). This will help to horizontally scale the vendor components on demand.

Cluster Management Best Practices

As part of Kubernetes adoption, we think these are some of the best practices to be followed in cluster management:

- Run only control plane components in the master node and avoid scheduling other pods in the master node.
- Enable TLS mode for docker service.
- Provision multiple master across DC (if applicable) to achieve high-availability for control plane components.

Copyright © 2021 [AppViewX, Inc]. All Rights Reserved.

- Use storage class for persistent volume provisioning.
- Monitor disk usage to avoid eviction of pods.
- Monitor ‘use of closed network connection’ error in kubelet logs to keep the node healthy. If you encounter this error then this might require restart of Kubelet service.
- If Calico is used as CNI and IP-in-IP is enabled, ensure that ip-ip protocol is enabled across worker node for communication.

Tools

It is imperative to choose tools that complement the Kubernetes and are also reliable and widely used in the industry. Therefore, we selected:

- **Kubeadm**: For cluster provisioning and management
- **Jenkins**: An automation server used as a continuous integration server
- **ISTIO**: Leveraging service mesh capabilities
- **Prometheus**: To scrape metrics from containerized workloads and store them in a time series database.
- **Grafana**: For visualizing Kubernetes cluster and monitoring the application metrics using dashboard.
- **ELK (Elasticsearch, Logstash, Kibana)**: Centralized log monitoring
- **Filebeat**: For harvesting application log files

Copyright © 2021 [AppViewX, Inc]. All Rights Reserved.

Chapter 4: Complex Use Cases

- Handling Graceful Shutdown of Pods
- Datacenter Based Routing

Handling Graceful Shutdown of Pods

A pod is a group of one or more containers, with shared storage and network resources. A pod's content is always co-located and co-scheduled and run in a shared context. Pods are **ephemeral**. They are not designed to run forever, and when a Pod is terminated it cannot be brought back. In general, Pods do not disappear until they are deleted by a user or by a controller.

Containers can be terminated at any time, due to auto scaling policy, deletion of pod or deployment or when rolling out updates. We have HPA defined based on custom metrics to auto scale vendor pods. During mid-night config fetch, vendor pods are in demand and the gateway will return 429 response code. The HPA is configured to monitor the 429 metrics and scale the pods to meet the demand. The HPA will keep polling for the 429 metrics and if there is no more breach in the configured threshold, the HPA will automatically scale down the pods. Since vendor pods take care of heavy lifting tasks, there might still be some task in-progress when the scale down is initiated by HPA. Though, Kubernetes will remove the terminating pods from its endpoint which means there will not be any further request sent to terminating pods but what happens to the unfinished tasks that are part of the terminating pod.

Whenever the pod is in terminating state, a **SIGTERM** is sent to the main process in each of the containers in the pod and a **termination grace period** is set, which defaults to **30 seconds**. Any cleanup activity has to be done within this period and we need to gracefully shutdown the pods. In our case, tasks in progress can last beyond 30 seconds and this would result in abrupt shutdown of the process and the business function would not have been completed.

To overcome this and to gracefully shutdown the process, we had to carry out following steps for the terminating pods:

- Set the **terminationGracePeriodSeconds=4800** for the containers.
- Create a preStop hook for the pod to send the TERM signal for the process. The process will capture the TERM signal and wait for all running threads to complete and does the graceful shutdown.
- The process will also notify the **prune daemonset** on graceful shutdown.
- The **prune daemonset** will watch for graceful shutdown messages and terminate the pods before the termination grace period expires. This is to avoid the pod waiting for the termination grace period to expire which will unnecessarily occupy resources.

Copyright © 2021 [AppViewX, Inc]. All Rights Reserved.

Based on the above strategy we were successfully able to manage the graceful shutdown of the process.

Datacenter Based Routing

As we cater to enterprise needs, **high-availability** of our application is mandatory and it's no more good to have features. We need high-availability for AppViewX microservices and MongoDB. The application is based on microservice architecture and the calls to the services are routed via gateway component. The gateway component supports a concept called **strict datacenter based routing**. When the gateway component is enabled with strict datacenter based routing mode, then the calls to a particular service has to be routed to the datacenter where it's provisioned.

To achieve high-availability of application and strict datacenter routing, we used Kubernetes **namespace** object as a solution. Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called **namespaces**. A Kubernetes namespace provides the scope for Pods, Services, and Deployments in the cluster. Users interacting with one namespace do not see the content in another namespace.

We used namespace and pod **affinity/anti-affinity** to spin the pods in the configured datacenter. The gateway routing table is configured with the fully-qualified domain of the service running in different datacenter with their respective namespace as DNS. This way the gateway component was able to talk to the service hosted in particular datacenter when strict data routing is enabled.

Copyright © 2021 [AppViewX, Inc]. All Rights Reserved.

Chapter 5: Outcome

We reaped a lot of benefits by adoption of Kubernetes with service mesh capabilities in terms of scale, performance and immutable infrastructure.

- **Optimal resource usage**:

We were able to make better use of the resource using custom metrics and scaling the pods when in demand.

- **Security**:

With service mesh and zero trust network model we were able to achieve higher and tightened security around the perimeter of the cluster.

- **Zero downtime:**

We were able to achieve zero downtime for upgrading AppViewX microservices.

- **Managed CI/CD**:

Better management of deployment code through modularity. Refined the way we build, package, deploy and manage applications.

Copyright © 2021 [AppViewX, Inc]. All Rights Reserved.

Chapter 6: References

1. https://kubernetes.io/docs/home/ - Kubernetes Object and cluster management
2. https://docs.projectcalico.org/about/about-calico - Calico overview and working
3. https://octetz.com/docs/2020/2020-10-01-calico-routing-modes/ - To understand calico routing
4. https://docs.projectcalico.org/reference/resources/networkpolicy#selector - Calico network policy
5. https://istio.io/latest/docs/concepts/ - ISTIO overview and concepts

Copyright © 2021 [AppViewX, Inc]. All Rights Reserved.