OOM for Production-Grade Deployments

Introduction

The ONAP Operations Manager provides a set of capabilities that facilitate Carrier Grade deployments of ONAP. ONAP deployments need to be capable of offering service while under adverse conditions typically with overall availability measured at five-nines or 99.999% uptime or about 5 minutes of downtime per year. This requirement might be strict for an orchestration system, but keep in mind that ONAP's closed loop control system could be providing monitoring a control for one or more critical VNFs that need to meet stringent up-time requirements as found in the TL 9000 Quality Management System Measurements Handbook.

The Road to High Availability

The progression of the ONAP project towards a fully Carrier Grade has started and will continue over the Beijing or possibly even subsequent releases. The steps along this progression are roughly as follows:

- Highly Available Kubernetes Deployments
- Reliable and Repeatable Deployment
- Health Monitoring
- Component Recoverability
- Centralized Logging
- Intra Component Clustering
- Pod Placement Rules
- ONAP S/W Upgrades & Rollbacks
- Geo-Redundant Deployments

For each of these steps the following sections describe the requirements in more detail and the technologies used to achieve it.

Highly Available Kubernetes Deployments

There is a high degree of variability possible in the deployment of Kubernetes. In some cases it may be installed and managed by hand, done with 3rd party tools like Rancher or even provided by a cloud provider like Microsoft Azure Container Service - Kubernetes has a description of the options here. Kubernetes provides guidance on creating deployments that may be suitable for carrier grade deployments of ONAP on their Building High-Availability Clusters wiki page.

Reliable and Repeatable Deployment

During the Amsterdam release OOM provided a set of capabilities to deploy some or all the ONAP components rapidly and efficiently as a cloud native application with the Kubernetes container orchestration system (note that DCAE is an exception here as DCAE provides its own orchestration system). Each of the components has a deployment specification that describes not only the containers and the container requirements but the relationships or dependencies between the containers. These dependencies dictate the order in-which the containers are started for the first time such that such dependencies are always met without arbitrary sleep times between container startups. For example, the SDC back-end container requires the Elastic-Search, Cassandra and Kibana containers within SDC to be ready and is also dependent on DMaaP (or the message-router) to be ready before becoming fully operational. Here is the deployment specification that describes these dependencies:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: sdc-be
  name: sdc-be
  namespace: "{{ .Values.nsPrefix }}-sdc"
spec:
  selector:
    matchLabels:
      app: sdc-be
  template:
    metadata:
      labels:
        app: sdc-be
      name: sdc-be
      annotations:
        pod.beta.kubernetes.io/init-containers: '[
          {
              "args": [
                  "--container-name",
                  "sdc-es",
                  "--container-name",
```

```
                "sdc-cs",
                "--container-name",
                "sdc-kb"
            ],
            "command": [
                "/root/ready.py"
            ],
            "env": [
                {
                    "name": "NAMESPACE",
                    "valueFrom": {
                        "fieldRef": {
                            "apiVersion": "v1",
                            "fieldPath": "metadata.namespace"
                        }
                    }
                }
            ],
            "image": "{{ .Values.image.readiness }}",
            "imagePullPolicy": "{{ .Values.pullPolicy }}",
            "name": "sdc-be-readiness"
        },
        {
            "args": [
                "--container-name",
                "dmaap"
            ],
            "command": [
                "/root/ready.py"
            ],
            "env": [
                {
                    "name": "NAMESPACE",
                    "value": "{{ .Values.nsPrefix }}-message-router"
                }
            ],
            "image": "{{ .Values.image.readiness }}",
            "imagePullPolicy": "{{ .Values.pullPolicy }}",
            "name": "sdc-dmaap-readiness"
        }
        ]'
spec:
  containers:
  - env:
    - name: ENVNAME
      value: AUTO
    - name: HOST_IP
      valueFrom:
        fieldRef:
          fieldPath: status.podIP
    image: {{ .Values.image.sdcBackend }}
    imagePullPolicy: {{ .Values.pullPolicy }}
    name: sdc-be
    volumeMounts:
    - mountPath: /usr/share/elasticsearch/data/
      name: sdc-sdc-es-es
    - mountPath: /root/chef-solo/environments/
      name: sdc-environments
    - mountPath: /etc/localtime
      name: sdc-localtime
      readOnly: true
    - mountPath: /var/lib/jetty/logs
      name: sdc-logs
    - mountPath: /var/log/onap
      name: sdc-logs-2
    - mountPath: /tmp/logback.xml
      name: sdc-logback
    lifecycle:
      postStart:
        exec:
          command: ["/bin/sh", "-c", "export LOG=wait_logback.log; touch $LOG; export SRC=/tmp/logback.xml;
```

```
export DST=/var/lib/jetty/config/catalog-be/; while [ ! -e $DST ]; do echo 'Waiting for $DST...' >> $LOG; sleep
5; done; sleep 2; /bin/cp -f $SRC $DST; echo 'Done' >> $LOG"]
        ports:
        - containerPort: 8443
        - containerPort: 8080
        readinessProbe:
          tcpSocket:
            port: 8443
          initialDelaySeconds: 5
          periodSeconds: 10
      - image: {{ .Values.image.filebeat }}
        imagePullPolicy: {{ .Values.pullPolicy }}
        name: filebeat-onap
        volumeMounts:
        - mountPath: /usr/share/filebeat/filebeat.yml
          name: filebeat-conf
        - mountPath: /var/log/onap
          name: sdc-logs-2
        - mountPath: /usr/share/filebeat/data
          name: sdc-data-filebeat
      volumes:
        - name: filebeat-conf
          hostPath:
            path: /dockerdata-nfs/{{ .Values.nsPrefix }}/log/filebeat/logback/filebeat.yml
        - name: sdc-logs-2
          emptyDir: {}
        - name: sdc-data-filebeat
          emptyDir: {}
        - name: sdc-logback
          hostPath:
            path: /dockerdata-nfs/{{ .Values.nsPrefix }}/log/sdc/be/logback.xml
        - name: sdc-sdc-es-es
          hostPath:
            path: /dockerdata-nfs/{{ .Values.nsPrefix }}/sdc/sdc-es/ES
        - name: sdc-environments
          hostPath:
            path: /dockerdata-nfs/{{ .Values.nsPrefix }}/sdc/environments
        - name: sdc-localtime
          hostPath:
            path:  /etc/localtime
        - name:  sdc-logs
          hostPath:
            path:  /dockerdata-nfs/{{ .Values.nsPrefix }}/sdc/logs
      imagePullSecrets:
      - name: "{{ .Values.nsPrefix }}-docker-registry-key"
```

Another feature that may assist in achieving a repeatable deployment in the presence of faults that may have reduced the capacity of the cloud is assigning priority to the containers such that mission critical components have the ability to evict less critical components. Kubernetes provides this capability with Pod Priority and Preemption.

Prior to having more advanced carrier grade features available, the ability to at least be able to re-deploy ONAP (or a subset of) reliably provides a level of confidence that should an outage occur the system can be brought back on-line predictably.

Backup and Restore

A critical factor in being able to recover from an ONAP outage is to ensure that critical state isn't lost after a failure. Much like ephemeral storage on VMs; any state information stored within a container will be lost once the container is restarted - containers are managed as Cattle, not Pets. To ensure that critical state information is retained after a failure, the OOM deployment specifications for the ONAP components use the Kubernetes concept of Persistent Volumes, an external storage facility that has its own lifecycle. The use of a persistent volume is specified in the ONAP deployment specifications. Here is an example from the sdnc db-deployment.yaml:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: sdnc-dbhost
  namespace: "{{ .Values.nsPrefix }}-sdnc"
spec:
  selector:
    matchLabels:
      app: sdnc-dbhost
  template:
    metadata:
      labels:
        app: sdnc-dbhost
      name: sdnc-dbhost
    spec:
      containers:
      - env:
        - name: MYSQL_ROOT_PASSWORD
          value: openECOMP1.0
        - name: MYSQL_ROOT_HOST
          value: '%'
        image: {{ .Values.image.mysqlServer }}
        imagePullPolicy: {{ .Values.pullPolicy }}
        name: sdnc-db-container
        volumeMounts:
        - mountPath: /etc/localtime
          name: localtime
          readOnly: true
        - mountPath: /var/lib/mysql
          name: sdnc-data
        ports:
        - containerPort: 3306
        readinessProbe:
          tcpSocket:
            port: 3306
          initialDelaySeconds: 5
          periodSeconds: 10
      volumes:
      - name: localtime
        hostPath:
          path: /etc/localtime
      - name: sdnc-data
        persistentVolumeClaim:
          claimName: sdnc-db
      imagePullSecrets:
      - name: "{{ .Values.nsPrefix }}-docker-registry-key"
```

At the bottom of the deployment specification is a list of the volumes, localtime and sdnc-data which uses an external persistentVolumeClaim of sdnc-db. This claim is specified as follows:

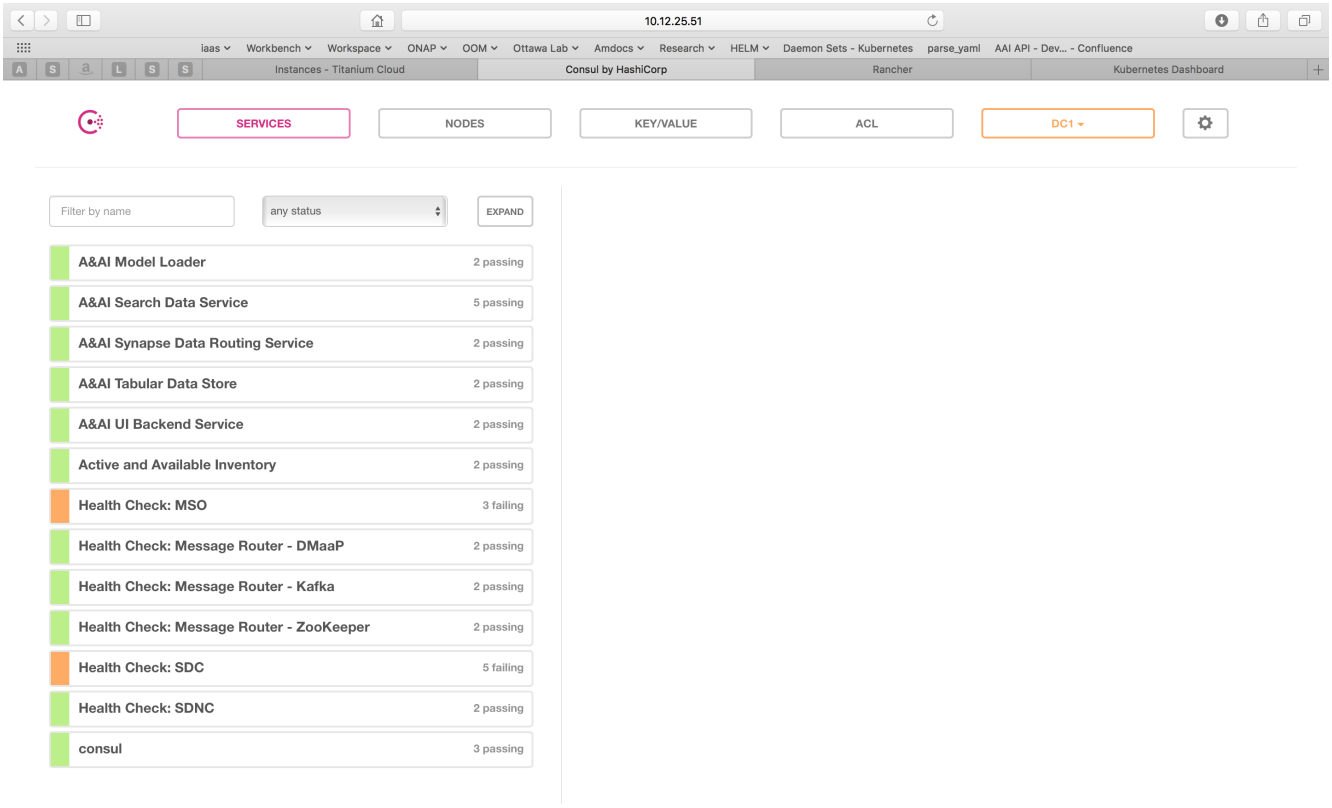
```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: "{{ .Values.nsPrefix }}-sdnc-db"
  namespace: "{{ .Values.nsPrefix }}-sdnc"
  labels:
    name: "{{ .Values.nsPrefix }}-sdnc-db"
spec:
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  hostPath:
    path: /dockerdata-nfs/{{ .Values.nsPrefix }}/sdnc/data
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: sdnc-db
  namespace: "{{ .Values.nsPrefix }}-sdnc"
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 2Gi
  selector:
    matchLabels:
      name: "{{ .Values.nsPrefix }}-sdnc-db"
```

Many different types of storage are supported by this capability such as: GCEPersistentDisk, AWSElasticBlockStore, AzureFile, AzureDisk, FC (Fibre Channel), FlexVolume, Flocker, NFS, iSCSI, RBD (Ceph Block Device), CephFS, Cinder (OpenStack block storage), Glusterfs, VsphereVolume, Quobyte Volumes, HostPath (Single node testing only), VMware Photon, Portworx Volumes, ScaleIO Volumes, and StorageOS.

As critical state is stored outside of the ONAP containers on a storage media specific to the cloud environment, specific instructions on how to backup and restore such storage is outside of the scope of ONAP.

Health Monitoring

All highly available systems include at least one facility to monitor the health of components within the system. Such health monitors are often used as inputs to distributed coordination systems (such as etcd, zookeeper, or consul) and monitoring systems (such as nagios or zabbix). Within ONAP Consul is the monitoring system of choice and deployed by OOM in two parts. A three-way, centralized Consul server cluster is deployed as a highly available monitor of all of the ONAP components. The Consul server provides a user interface that allows a user to graphically view the current health status of all of the ONAP components for which agents have been created - a sample from the ONAP Integration labs follows. Monitoring of ONAP components is configured in the agents within JSON files and stored in gerrit under the consul-agent-config.



Initially the Consul agents are using the same health monitoring facilities as the robot test infrastructure which are typically just validating that the end-point is reachable. Some health checks already support more advanced checking - such as validating that a database is able to create, update and delete an entry. Consul exposes an API that allows external agents to use the results of the health check, such as the Kubernetes "liveness" probes described below.

Component Recoverability

OOM deploys ONAP with Kubernetes defined by deployment specifications as mentioned earlier. These same deployment specifications are also used to implement automatic recoverability of ONAP components when individual components fail. Once ONAP is deployed, a "liveness" probe starts checking the health of the components after a specified startup time. These liveness probes can simply check that a port is available, that a built-in health check is reporting good health, or that the Consul health check is positive. Should a liveness probe indicate a failed container it will be restarted as described in the deployment specification. Should the deployment specification indicate that there are one or more dependencies to this container or component (for example a dependency on a database) the dependency will be satisfied before the container/component is restarted. This mechanism ensures that, after a failure, all of the ONAP components restart successfully. Note that, during the Amsterdam release, deployment specifications were created for all ONAP components but not all of these deployment specifications are restartable (idempotent). Further work is required during the Beijing release to ensure recoverability of all the ONAP components.

Centralized Logging

An important tool in achieving minimal downtime is the ability to rapidly diagnose problems and determine the root cause. The Logging Enhancements Project have been building a centralized log collection system based on the Elastic Stack and a Filebeat collector container that is instantiated alongside the containers for each of the ONAP components. Here is an example from the aai-traversal deployment specification:

```
    spec:
      containers:
      - name: aai-traversal
[...]
      - name: filebeat-onap-aai-traversal
        image: {{ .Values.image.filebeat }}
        imagePullPolicy: {{ .Values.pullPolicy }}
        volumeMounts:
        - mountPath: /usr/share/filebeat/filebeat.yml
          name: filebeat-conf
        - mountPath: /var/log/onap
          name: aai-traversal-logs
        - mountPath: /usr/share/filebeat/data
          name: aai-traversal-filebeat
```

Filebeat collects logs from within the namespace of each component and ships them to the centralized logging stack that was deployed by OOM with the other ONAP components. Users are able to point their web browsers to the Kibana component and see all of the raw logs as well as predefined dashboards that show the state of ONAP in real-time.

Intra Component Clustering

The OOM project is not responsible for creating highly available versions of all of the ONAP components, but does provide via Kubernetes many built in facilities to build clustered, highly available systems including: Services with load-balancers (including support for External Load Balancers), Ingress Resources, and Replica Sets. Some of the open-source projects that form the basis of ONAP components directly support clustered configurations like ODL with instructions on Setting Up Clustering or MariaDB Getting Started with MariaDB Galera Cluster. .

OOM uses the Kubernetes service abstraction to provide a consistent access point for each of the ONAP components, independent of the pod or container architecture of that component. For example, the SDN-C component may introduce OpenDaylight clustering as some point and change the number of pods in this component to three or more, but this change will be isolated from the other ONAP components by the service abstraction. A service can include a load balancer on its ingress to distribute traffic between the pods and even react to dynamic changes in the number of pods if they are part of a replica set. A replica set is a construct that is used to describe the desired state of the cluster. For example 'replicas: 3' indicates to Kubernetes that a cluster of 3 instances is the desired state. Should one of the members of the cluster fail, a new member will be automatically started to replace it.

Some of the ONAP components many need a more deterministic deployment; for example to enable intra-cluster communication. For these applications the component can be deployed as a Kubernetes StatefulSet which will maintain a persistent identifier for the pods and thus a stable network id for the pods. For example: the pod names might be web-0, web-1, web-{N-1} for N 'web' pods with corresponding DNS entries such that intra service communication is simple even if the pods are physically distributed across multiple nodes. An example of how these capabilities can be used is described in the Running Consul on Kubernetes tutorial.

The SDN-C Clustering on Kubernetes page describes a working example of many of these techniques working together.

Pod Placement Rules

OOM will use the rich set of Kubernetes node and pod affinity / anti-affinity rules to minimize the chance of a single failure resulting in a loss of ONAP service. Node affinity / anti-affinity is used to guide the Kubernetes orchestrator in the placement of pods on nodes (physical or virtual machines). For example:

- if a container used Intel DPDK technology the pod may state that it as affinity to an Intel processor based node, or
- geographical based node labels (such as the Kubernetes standard zone or region labels) may be used to ensure placement of a DCAE complex close to the VNFs generating high volumes of traffic thus minimizing networking cost. Specifically, if nodes were pre-assigned labels East and West, the pod deployment spec to distribute pods to these nodes would be:

```
nodeSelector:
    failure-domain.beta.kubernetes.io/region: {{ .Values.location }}
```

Where "location: West" is specified in the values.yaml file used to deploy one DCAE cluster and "location: East" is specified in a second values.yaml file (see OOM Configuration Management for more information about configuration files like the values.yaml file).

Node affinity can also be used to achieve geographic redundancy if pods are assigned to multiple failure domains. For more information refer to Assigning Pods to Nodes. Kubernetes has a comprehensive system called Taints and Tolerations that can be used to force the container orchestrator to repel pods from nodes based on static events (an administrator assigning a taint to a node) or dynamic events (such as a node becoming unreachable or running out of disk space). There are no plans to use taints or tolerations in the ONAP Beijing release.

Pod affinity / anti-affinity is the concept of creating a spacial relationship between pods when the Kubernetes orchestrator does assignment (both initially an in operation) to nodes as explained in Inter-pod affinity and anti-affinity. For example, one might choose to co-located all of the ONAP SDC containers on a single node as they are not critical runtime components and co-location minimizes overhead. On the other hand, one might choose to ensure that all of the containers in an ODL cluster (SDNC and APPC) are placed on separate nodes such that a node failure has minimal impact to the operation of the cluster. An example of how pod affinity / anti-affinity is shown below::

Pod Affinity / Anti-Affinity

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values:
            - S1
        topologyKey: failure-domain.beta.kubernetes.io/zone
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: security
              operator: In
              values:
              - S2
          topologyKey: kubernetes.io/hostname
  containers:
  - name: with-pod-affinity
    image: gcr.io/google_containers/pause:2.0
```

This example contains both podAffinity and podAntiAffinity rules, the first rule is is a must (requiredDuringSchedulingIgnoredDuringExecution) while the second will be met pending other considerations (preferredDuringSchedulingIgnoredDuringExecution).

ONAP S/W Upgrades & Rollbacks

Kubernetes has built-in capabilities to enable the upgrade of pods without causing a loss of the service being provided by that pod or pods (if configured as a cluster). As described in the OOM User Guide, ONAP components provide an abstracted 'service' end point with the pods or containers providing this service hidden from other ONAP components by a load balancer. This capability is used during upgrades to allow a pod with a new image to be added to the service before removing the pod with the old image. This 'make before break' capability ensures minimal downtime.

When upgrading a cluster a parameter controls the minimum size of the cluster during the upgrade while another parameter controls the maximum number of nodes in the cluster. For example, SNDC configured as a 3-way ODL cluster might require that during the upgrade no fewer than 2 pods are available at all times to provide service while no more than 5 pods are ever deployed across the two versions at any one time to avoid depleting the cluster of resources. In this scenario, the SDNC cluster would start with 3 old pods then Kubernetes may add a new pod (3 old, 1 new), delete one old (2 old, 1 new), add two new pods (2 old, 3 new) and finally delete the 2 old pods (3 new). During this sequence the constraints of the minimum of two pods and maximum of five would be maintained while providing service the whole time.

Initiation of an upgrade is triggered by changes in the deployment specifications. For example, if the image specified for one of the pods in the SDNC deployment specification were to change (i.e. point to a new Docker image in the nexus3 repository – commonly through the change of a deployment variable), the sequence of events described in the previous paragraph would be initiated.

Unfortunately, not all upgrades are successful. In recognition of this the lineup of pods within an ONAP deployment is tagged such that an administrator may force the ONAP deployment back to the previously tagged configuration or to a specific configuration, say to jump back two steps if an incompatibility between two ONAP components is discovered after the two individual upgrades succeeded. This rollback functionality gives the administrator confidence that in the unfortunate circumstance of a failed upgrade the system can be rapidly brought back to a known good state.

This process of rolling upgrades while under service is illustrated in this short YouTube video showing a Zero Downtime Upgrade of a web application while under a 10 million transaction per second load.

Many of the ONAP components contain their own databases which are used to record configuration or state information. The schemas of these databases may change from version to version in such a way that data stored within the database needs to be migrated between versions. If such a migration script is available it can be invoked during the upgrade (or rollback) by Container Lifecycle Hooks. Two such hooks are available, PostStart and PreStop, which containers can access by registering a handler against one or both. Note that it is the responsibility of the ONAP component owners to implement the hook handlers – which could be a shell script or a call to a specific container HTTP endpoint – following the guidelines listed on the Kubernetes site. Lifecycle hooks are not restricted to database migration or even upgrades but can be used anywhere specific operations need to be taken during lifecycle operations.

Note that although OOM uses Kubernetes facilities to minimize the effort required of the ONAP component owners to implement a successful rolling upgrade strategy there are other considerations that must be taken into consideration. For example, external APIs – both internal and external to ONAP – should be designed to gracefully accept transactions from a peer at a different software version to avoid deadlock situations. Embedded version codes in messages may facilitate such capabilities.

Geo-Redundant Deployments

As described in the Pod Placement Rules section, OOM enables the placement of specific pods into specific zones or regions thus providing protection from a single cluster failure. These placement rules can also be used to distribute specific resources, such as a DCAE cluster, close to the VNFs that DCAE is monitoring. To build such a distributed network operators will use Kubernetes Federation to link multiple clusters.

The Kubernetes federation control plane enables clusters that are geographically separated to function as a single deployment with DNS servers and load balancers distributing work across the clusters. Federation also enables hybrid clouds say with nodes being provided by a private OpenStack cluster and a Microsoft Azure cluster. Note that clusters can each scale to thousands of nodes so it is unlikely that capacity will be the sole reason for deploying ONAP within a federation of clusters.

List of Epics

The following list of JIRA Epics represent the development activities required to complete the OOM related carrier grade activities (to be confirmed):

Key	Summary	T	Description
OOM-109	Platform Centralized Logging		As an ONAP operator, I want to have centralized logging for each of onap component, so that I can emit standardized, machine-readable logging output. See: https://wiki.onap.org/display/DW/LOG+M1+Release+Planning **Acceptance Criteria** • ONAP components logs are centralized into a common repository • ONAP components logs are searchable • ONAP Transactions are logged and tracable across all ONAP components • ONAP components logs are standardized in order to facilitate the search • ONAP components logs are machine readable.

OOM-6	Automated platform deployment on Docker /Kubernetes		As an ONAP operator, I want to deploy ONAP into a fully containerized architecture so that I can easily deploy the platform on any infrastructure.
			
			Acceptance criteria
			
			• MVP components are deployed with K8S with a single click. MVP components are:
			  ◦ AAI
			  ◦ SO
			  ◦ Message-router
			  ◦ SDC
			  ◦ VID
			  ◦ SO (without TOSCA support)
			  ◦ SDNC
			  ◦ Robot
			  ◦ APPC
			  ◦ DCAE Gen1
			  ◦ DCAE Gen2 Controller (DCAE services themselves will be managed using HEAT)
			  ◦ Portal
			  ◦ VF-C
			  ◦ Multi-VIM
			  ◦ MSB
			  ◦ Policy
			  ◦ *Stretch: SO with TOSCA support*

OOM-10	Platform configuration management		As an ONAP operator, I want to centrally manage ONAP platform and components configurations so that I can enable multi-instance, scalable/resilient container platform deployment.
			
			New requirement 20180315 - verify that healthcheck is working for each retrofitted component
			
			Acceptance Criteria
			
			• All ONAP components are parameterized and fully configurable
			• Ability for operators to manage independent platform environments without any hardcoded parameters
			• Configurability includes at least, but not limited to:
			  ◦ Credentials, Secrets
			  ◦ Environment parameters: host names, node ports, IP addresses, URLs/Paths
			  ◦ Environment variables
			  ◦ Versions, Image names
			  ◦ Logging levels
			  ◦ Runtime parameters, e.g. JVM size
			  ◦ High-availability, resiliency, clustering parameters
			  ◦ Auto-Scaling, auto-healing policies

OOM-412	Handle branching in OOM configuration - 16 Nov		WE will need a story to parameterize the downloading of branched artifacts in branches - also see ~~OOM-411~~ (dup of ~~OOM-476~~) as well for the docker images
OOM-486	HELM upgrade from 2.3 to 2.8.0		update 20180124: we are good for 2.7.2 on Rackspace and AWS as well - vnc-portal working now (Note: rancher 1.6.10 upgraded - next test 1.6.12-14) Update: Rancher 1.6.14, Helm 2.8.0, Docker 17.03.2, Kubernetes 1.8.6, Kubectl 1.9.0 OK old..... Team, we need to do the 1.6.14 rancher and helm 2.8 client (upgrade rancher from 2.6) and cover off the ram and vnc-portal issues - will post links shortly We need to fix vnc-portal running on helm 2.6+ - see ----~~OOM-441~~ ---- Also raising priority by request as several teams run into this issue running helm 2.5+ One issue I covered off is the new docker image - it came out a couple months before the helm update - and also has the issue Full integration testing on multiple deployment environment will certify our move from Rancher 1.6.10/Helm 2.3 20180124: upgrade procedure for

```
AWS
ubuntu@ip-172-31-13-94:~$ wget http://storage.googleapis.com/kubernetes-helm
/helm-v2.7.2-linux-amd64.tar.gz
--2018-01-24 22:20:04--  http://storage.googleapis.com/kubernetes-helm/helm-
v2.7.2-linux-amd64.tar.gz
Resolving storage.googleapis.com (storage.googleapis.com)... 209.85.203.128,
2607:f8b0:4004:808::2010
Connecting to storage.googleapis.com (storage.googleapis.com)|209.85.203.128
|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 12166338 (12M) [application/x-tar]
Saving to: ‘helm-v2.7.2-linux-amd64.tar.gz’
helm-v2.7.2-linux-amd64.tar.gz           100%
[================================================================================
======>]  11.60M  7.27MB/s    in 1.6s
2018-01-24 22:20:06 (7.27 MB/s) - ‘helm-v2.7.2-linux-amd64.tar.gz’ saved
[12166338/12166338
ubuntu@ip-172-31-13-94:~$ tar -zxvf helm-v2.7.2-linux-amd64.tar.gz
linux-amd64/
linux-amd64/README.md
linux-amd64/LICENSE
linux-amd64/helm
ubuntu@ip-172-31-13-94:~$ sudo mv linux-amd64/helm /usr/local/bin/helm

ubuntu@ip-172-31-13-94:~$ helm version
Client: &version.Version{SemVer:"v2.7.2", GitCommit:"
8478fb4fc723885b155c924d1c8c410b7a9444e6", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.3.0", GitCommit:"
d83c245fc324117885ed83afc90ac74afed271b4", GitTreeState:"clean"}
ubuntu@ip-172-31-13-94:~$ helm init --upgrade
Creating /home/ubuntu/.helm
Creating /home/ubuntu/.helm/repository
Creating /home/ubuntu/.helm/repository/cache
Creating /home/ubuntu/.helm/repository/local
Creating /home/ubuntu/.helm/plugins
Creating /home/ubuntu/.helm/starters
Creating /home/ubuntu/.helm/cache/archive
Creating /home/ubuntu/.helm/repository/repositories.yaml
Adding stable repo with URL: https://kubernetes-charts.storage.googleapis.com
Adding local repo with URL: http://127.0.0.1:8879/charts
$HELM_HOME has been configured at /home/ubuntu/.helm.
Tiller (the Helm server-side component) has been upgraded to the current
version
Happy Helming!

ubuntu@ip-172-31-13-94:~$ helm version
Client: &version.Version{SemVer:"v2.7.2", GitCommit:"
8478fb4fc723885b155c924d1c8c410b7a9444e6", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.7.2", GitCommit:"
8478fb4fc723885b155c924d1c8c410b7a9444e6", GitTreeState:"clean"}
```

OOM-590	OOM Wiki documentation of deployment options		Raised at request on the 20180117 OOM meeting

Deliverables

WIKI deployment pages - eventually to copy to read the docs

6 issues