Beijing Scope

During the Beijing release the following capabilities are added or enhanced from the OOM capabilities available in the Amsterdam release:

- Deploy - with built-in component dependency management (including multiple clusters, federated deployments across sites, and anti-affinity rules)
- Configure - unified configuration across all ONAP components
- Monitor - real-time health monitoring feeding to a Consul UI and Kubernets
- Heal - failed ONAP containers are recreated automatically
- Scale - cluster ONAP services to enable seamless scaling
- Upgrade - change-out containers or configuration with little or no service impact
- Delete - cleanup individual containers or entire deployments

The following sections describe these areas in more detail.

Mike Elliott demonstrated many of these concepts at the virtual face-to-face meeting, shown here:

OOM Beijing Demo...Mike Elliott.mp4

In addition to the capabilities listed below note that the OOM deployment of ONAP in Beijing is now done within a single Kubernetes namespace where in Amsterdam a namespace was created for each of the ONAP components.

Deploy - with built-in component dependency management (including multiple clusters, federated deployments across sites, and anti-affinity rules)

The OOM team with assistance from the ONAP project teams, have built a comprehensive set of Kubernetes deployment specifications, yaml files very similar to TOSCA files, that describe the composition of each of the ONAP components and the relationship within and between components. These deployment specifications describe the desired state of an ONAP deployment and instruct the Kubernetes container manager as to how to maintain the deployment in this state. 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 - where ready implies the built-in "readiness" probes succeeded - before becoming fully operational.

When an initial deployment of ONAP is requested the current state of the system is NULL so ONAP is deployed by the Kubernetes manager as a set of Docker containers on one or more predetermined hosts. The hosts could be physical machines or virtual machines. When deploying on virtual machines the resulting system will be very similar to “Heat” based deployments, i.e. Docker containers running within a set of VMs, the primary difference being that the allocation of containers to VMs is done dynamically with OOM and statically with “Heat”.

Example SO deployment descriptor file shows SO's dependency on its mariadb component:

(M)SO deployment specification excerpt

```
..
kind: Deployment
metadata:
  name: mso
..
spec:
..

      initContainers:
      - command:
        - /root/ready.py
        args:
        - --container-name
        - mariadb
..
      containers:
      - command:
        - /tmp/start-jboss-server.sh
        image: {{ .Values.image.mso }}
        imagePullPolicy: {{ .Values.pullPolicy }}
        name: mso
..
```

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.yamlfile).

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
```

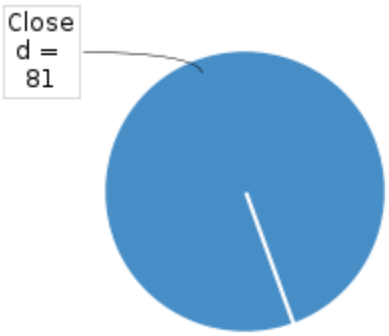
This example contains both podAffinity and podAntiAffinity rules, the first rule is is a must (requiredDuringSchedulingIgnoredDuringExecution) while the second will be met pending other considerations (preferredDuringSchedulingIgnoredDuringExecution).

Preemption

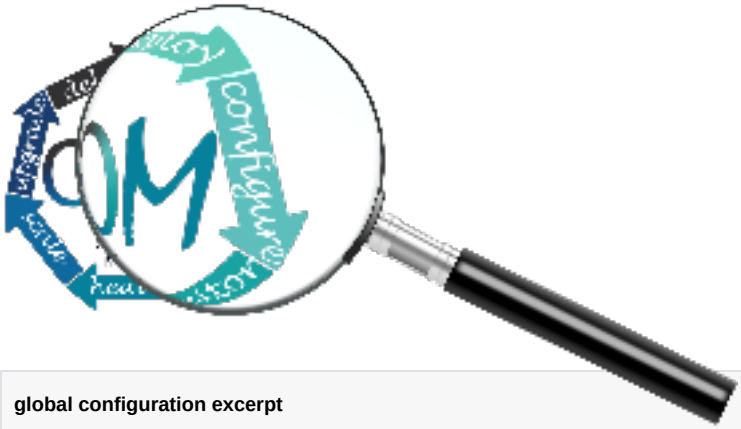
Another feature that may assist in achieving a repeatable deployment in the presence of faults that may have reduced the capacity of the cloud is assigning priority to the containers such that mission critical components have the ability to evict less critical components. Kubernetes provides this capability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

Prior to having more advanced production 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.

The overall Epic for OOM based deployment of ONAP is **OOM-6** - Getting issue details... STATUS with the following status:



During the Beijing release the 'initContainers' constructs were updated from the previous beta implementation to the current released syntax under the JIRA Story **OOM-406** - Getting issue details... STATUS



Configure - unified configuration across all ONAP components

Each project within ONAP has its own configuration data generally consisting of: environment variables, configuration files, and database initial values. Many technologies are used across the projects resulting in significant operational complexity and an inability to apply global parameters across the entire ONAP deployment. OOM solves this problem by introducing a common configuration technology, Helm charts, that provide a hierarchical configuration configuration with the ability to override values with higher level charts or command line options. For example, if one wishes to change the OpenStack instance oam_network_cidr and ensure that all ONAP components reflect this change, one could change the vnfDeployment/openstack/oam_network_cidr value in the global configuration file as shown below:

global configuration excerpt

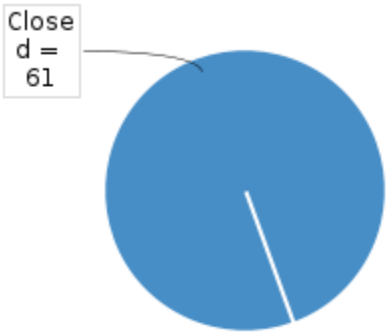
```
nsPrefix: onap
nodePortPrefix: 302
apps: consul msb mso message-router sdnc vid robot portal policy appc aai sdc dcaegen2 log cli multicloud
clamp vnfsdk aaf kube2msb
dataRootDir: /dockerdata-nfs

# docker repositories
repository:
  onap: nexus3.onap.org:10001
  oom: oomk8s
  aai: aaionap
  filebeat: docker.elastic.co

image:
  pullPolicy: Never

# vnf deployment environment
vnfDeployment:
  openstack:
    ubuntu_14_image: "Ubuntu_14.04.5_LTS"
    public_net_id: "e8f51956-00dd-4425-af36-045716781ffc"
    oam_network_id: "d4769dfb-c9e4-4f72-b3d6-1d18f4ac4ee6"
    oam_subnet_id: "191f7580-acf6-4c2b-8ec0-ba7d99b3bc4e"
    oam_network_cidr: "192.168.30.0/24"
    username: "vnf_user"
    api_key: "vnf_password"
    tenant_name: "vnfs"
    tenant_id: "47899782ed714295b1151681fdfd51f5"
    region: "RegionOne"
    keystone_url: "http://1.2.3.4:5000"
    flavour_medium: "m1.medium"
    service_tenant_name: "services"
  dmaap_topic: "AUTO"
  demo_artifacts_version: "1.1.0-SNAPSHOT"
```

The migration from the disparate configuration methodologies to Helm charts is tracked under the OOM-460 - Getting issue details... STATUS JIRA story. Within each of the projects a new configuration repository contains all of the project specific configuration artifacts. As changes are made within the project, it's the responsibility of the project team to make appropriate changes to the configuration data. The OOM-10 - Getting issue details... STATUS Epic tracks this work with the following status:



Monitor - real-time health monitoring feeding to a Consul UI and Kubernets

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). OOM provides two mechanims to monitor the real-time health of an ONAP deployment:

- a Consul GUI for a human operator or downstream monitoring systems and Kubernetes liveness probes that enable automatic healing of failed containers, and
- a set of liveness probes which feed into the Kubernetes manager which are described in the Heal section.

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,and
- a number of Consul agents.

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, here is an example from the AAI model loader:

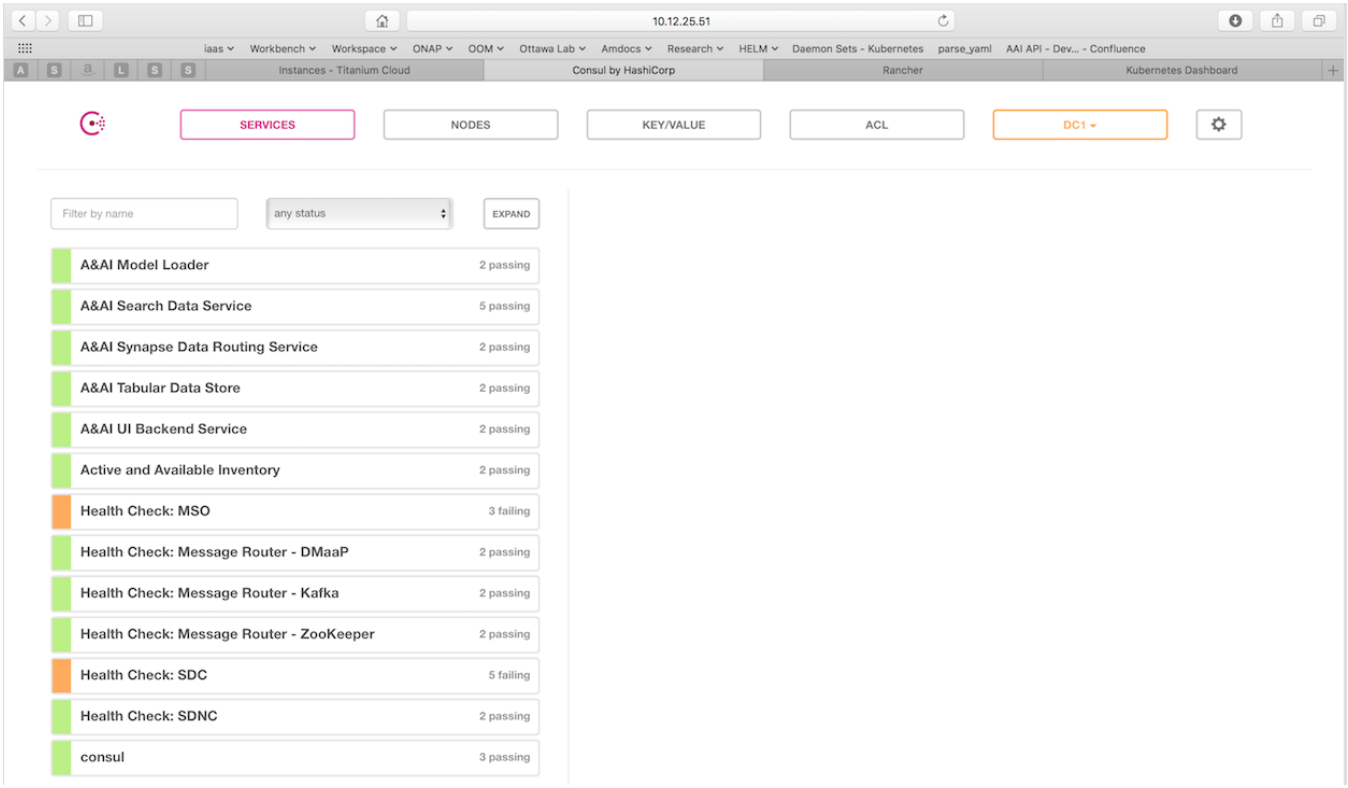
aai-model-loader-health.json

```
{
  "service": {
    "name": "A&AI Model Loader",
    "checks": [
      {
        "id": "model-loader-process",
        "name": "Model Loader Presence",
        "script": "/consul/config/scripts/model-loader-script.sh",
        "interval": "15s",
        "timeout": "1s"
      }
    ]
  }
}
```

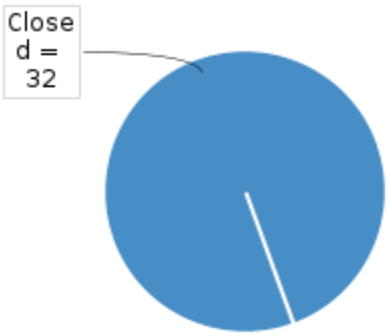
To see the real-time health of a deployment go to:

```
http://<kubernetes IP>:30270/ui/
```

where a GUI much like the following will be found:



The OOM-7 - Getting issue details... STATUS JIRA Epic tracks this work with the following status:



Heal - failed ONAP containers are recreated automatically

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 terminated and a replacement will be started in its place - containers are ephemeral. 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 replacement container /component is started. This mechanism ensures that, after a failure, all of the ONAP components restart successfully.

For example, to monitor the SDNC component has following liveness probe can be found in the SDNC DB deployment specification:

sdnc db liveness probe

```
livenessProbe:
  exec:
    command: ["mysqladmin", "ping"]
  initialDelaySeconds: 30
  periodSeconds: 10
  timeoutSeconds: 5
```

The 'initialDelaySeconds' control the period of time between the readiness probe succeeding and the liveness probe starting. 'periodSeconds' and 'timeoutSeconds' control the actual operation of the probe.

Note that containers are inherently ephemeral so the healing action destroys failed containers and any state information within it. To avoid a loss of state, a persistent volume should be used to store all data that needs to be persisted over the re-creation of a container. Persistent volumes have been created for the database components of each of the projects and the same technique can be used for all persistent state information.

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.

Scale - cluster ONAP services to enable seamless scaling

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.

Upgrade - change-out containers or configuration with little or no service impact

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.

OOM uses Helm K8S package manager to deploy ONAP components. Each component is arranged in a packaging format called a chart – a collection of files that describe a set of k8s resources. Helm allows for rolling upgrades of the ONAP component deployed. To upgrade a component Helm release you will need an updated Helm chart. The chart might have modified, deleted or added values, deployment yamls, and more.

To get the release name use:

```
helm ls
```

To easily upgrade the release use:

```
helm upgrade [RELEASE] [CHART]
```

To roll back to a previous release version use:

```
helm rollback [flags] [RELEASE] [REVISION]
```

for example, to upgrade the onap-mso helm release to the latest MSO container release v1.1.2:

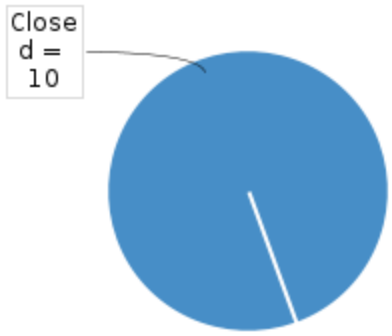
- Edit mso valus.yaml which is part of the chart

Change “mso: nexus3.onap.org:10001/openecomp/mso:v1.1.1” to

“mso: nexus3.onap.org:10001/openecomp/**mso:v1.1.2**”

- From the chart location run:
  helm upgrade onap-mso ./

The previous mso pod will be terminated and a new mso pod with an updated mso container will be created.



Delete - cleanup individual containers or entire deployments