Agile Lifecycle Manager
Version 1.3

Installation, Administration and User Guide

14 December 2018

IBM

Agile Lifecycle Manager
Version 1.3

Installation, Administration and User Guide

14 December 2018

IBM

Note

Before using this information and the product it supports, read the information in "Notices" on page 163.

This edition applies to Version 1.2 of IBM® Agile Lifecycle Manager (product number 5737-E91) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Tables

Preface

This PDF document contains topics from the Knowledge Center in a printable format.

About this release

Agile Lifecycle Manager (Version 1.3) can now be deployed on IBM Cloud Private.

The IBM Agile Lifecycle Manager Installation, Administration and User Guide has been updated to version 1.3.:

ICP components
The ICP version of Agile Lifecycle Manager consists of a number of services packaged as Helm charts.

Requirements
Hardware and software requirements differ from those of the on-premise version.

Deployment
Agile Lifecycle Manager components and services run within containers, and communication between these containers is managed and orchestrated using IBM Cloud Private running on a Kubernetes cluster.

Installation
The installation process for Agile Lifecycle Manager 1.3 on ICP differs from the on-premise version.

Logging into the ICP UI
You log onto the Agile Lifecycle Manager ICP UI using the a logon URL constructed from the master node hostname and port number.

Related information:

How to download Agile Lifecycle Manager

IBM Agile Lifecycle Manager Version 1.3 Release Notes

Chapter 1. Product overview

IBM Agile Lifecycle Manager provides users with a toolkit to manage the lifecycle of both virtual and physical network services. This includes the design, test, deployment, monitoring and healing of services.

Attention:

Agile Lifecycle Manager is now also available as an IBM Cloud Private (ICP) version. The ICP deployment and configuration of Agile Lifecycle Manager differs significantly from the on-premise version. See the ICP deployment topic for an overview, and the .

Benefits

Using Agile Lifecycle Manager you can design and integrate external resources into virtual production environments and then automate the management of end-to-end lifecycle processes. This approach is known as 'network function virtualization'. This section elaborates on the benefits of this approach and the key functionality offered by Agile Lifecycle Manager, and also provides you with case study material.

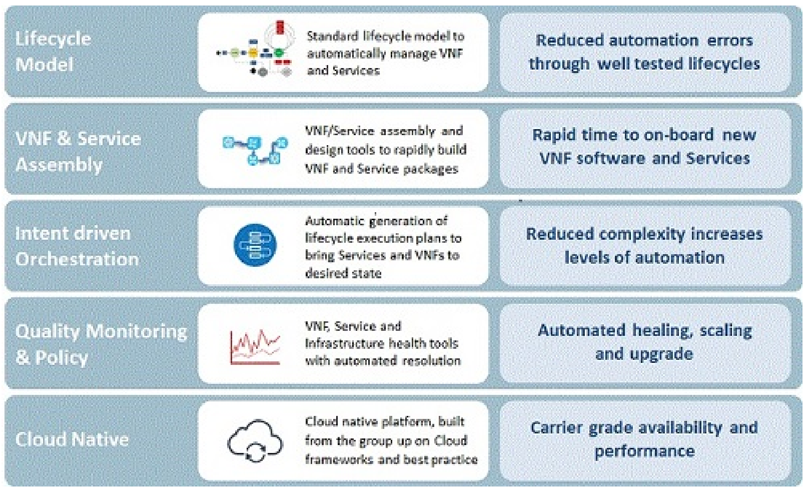
Benefits of network function virtualization (NFV)

NFV brings a significant operational paradigm shift for service providers. Today’s physical network appliances require highly manual processes to manage their end-to-end lifecycle. Testing, installation, configuration, and problem management of network appliances all revolve around manual activities that often require a physical truck roll or a human to run each lifecycle process.

NFV’s software paradigm promises fully automated lifecycle processes for bringing network services into production and maintaining them thereafter. Virtual network functions (VNF) allow a much simpler set of lifecycle tasks enabling near full automation of the creation and healing of virtual services, far more than is possible with their physical counterparts.

There are vast business opportunities associated with NFV transformation including new revenue streams, improved customer experience and reductions in both operational and capital expenditure. However, a fully automated lifecycle solution for NFV comes with additional complexities.

IBM Agile Lifecycle Manager is a comprehensive services design, testing and automated deployment platform addressing the challenges and complexities of the NFV paradigm. It delivers an end-to-end automated service lifecycle solution from initial design to production, as depicted in the following figure:

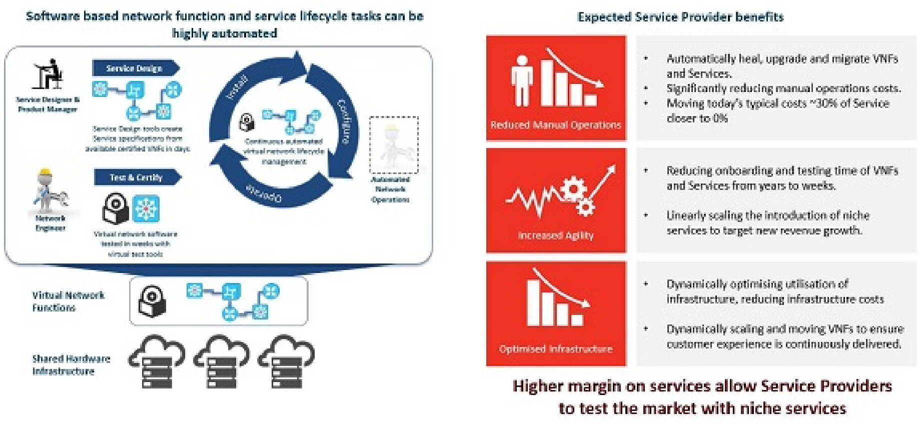


Lifecycle Model	Standard lifecycle model to automatically manage VNF and Services	Reduced automation errors through well tested lifecycles
VNF & Service Assembly	VNF/Service assembly and design tools to rapidly build VNF and Service packages	Rapid time to on-board new VNF software and Services
Intent driven Orchestration	Automatic generation of lifecycle execution plans to bring Services and VNFs to desired state	Reduced complexity increases levels of automation
Quality Monitoring & Policy	VNF, Service and Infrastructure health tools with automated resolution	Automated healing, scaling and upgrade
Cloud Native	Cloud native platform, built from the group up on Cloud frameworks and best practice	Carrier grade availability and performance

The key differentiating features of Agile Lifecycle Manager depicted in this figure include:

- A common way of handling resources through a unified lifecycle model
- Support for quick resources and assembly on-boarding
- Intent-driven lifecycle management
- Quality management and policy modules
- Cloud-native solution

The benefits of using Agile Lifecycle Manager to deliver NFV are illustrated in the following figure:


Software based network function and service lifecycle tasks can be highly automated
Service Designer & Product Manager
Service Design
Service Design tools create Service specifications from available certified VNFs in days
Network Engineer
Test & Certify
Virtual network software tested in weeks with virtual test tools
Install
Configure
Operate
Continuous automated virtual network lifecycle management
Automated Network Operations
Virtual Network Functions
Shared Hardware Infrastructure
Expected Service Provider benefits
Reduced Manual Operations
Increased Agility
Optimised Infrastructure
Higher margin on services allow Service Providers to test the market with niche services

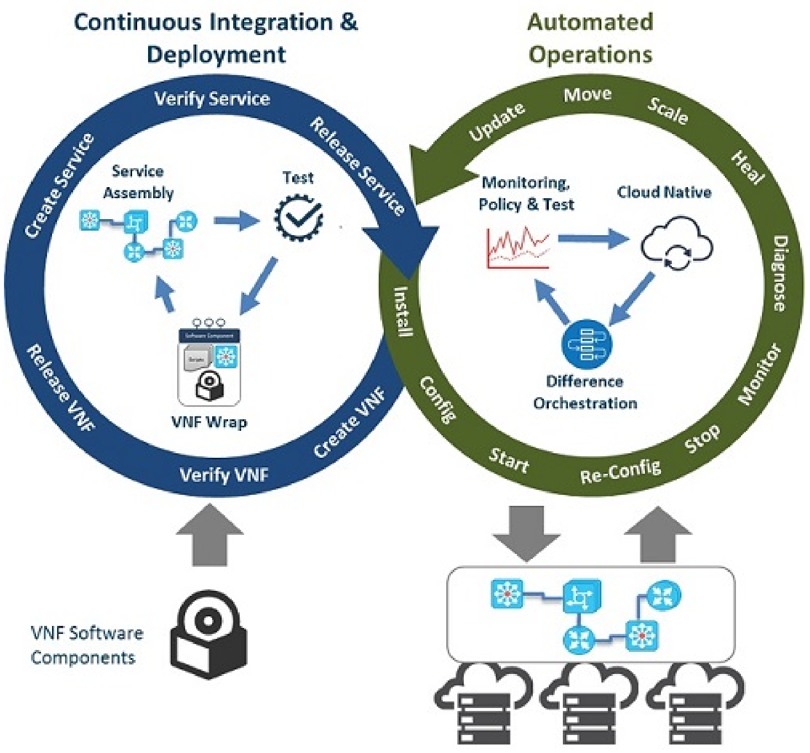
Reduced Manual Operations	• Automatically heal, upgrade and migrate VNFs and Services. • Significantly reducing manual operations costs. • Moving today's typical costs ~30% of Service closer to 0%
Increased Agility	• Reducing onboarding and testing time of VNFs and Services from years to weeks. • Linearly scaling the introduction of niche services to target new revenue growth.
Optimised Infrastructure	• Dynamically optimising utilisation of infrastructure, reducing infrastructure costs • Dynamically scaling and moving VNFs to ensure customer experience is continuously delivered.

Lifecycle management

To achieve NFV's promised levels of automation, Agile Lifecycle Manager provides a complete DevOps toolchain that manages the end-to-end lifecycle of virtual network services, from release management of VNF software packages to the continuous orchestration and running of VNFs and service instances.

Third party VNF software must be wrapped in a well-tested standard lifecycle interface, and service bundles of multi-vendor VNFs must be tested for interoperability and performance to ensure that there are no errors. This ensures that, once in production, services and VNFs can be constantly created, configured, updated, scaled, healed, and migrated without manual intervention.

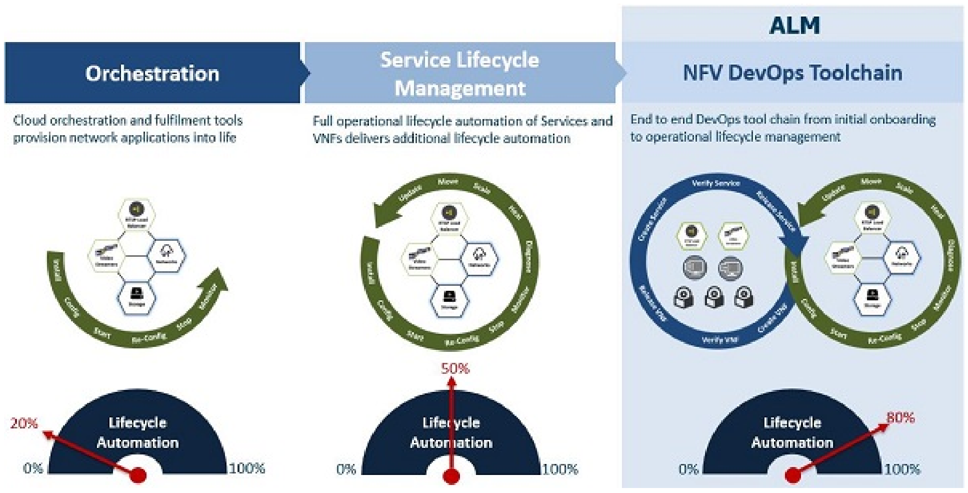
The complete lifecycle management is shown in the following figure:



VNF and services onboarding requires a comprehensive release management strategy, a suite of tools and a lifecycle integration framework to accommodate the variety of third party VNF software package formats. Continuous in-life orchestration also requires a new approach to modelling and managing the complexity of in-life network function lifecycle management. To achieve the levels of automation required, a much simpler and standardized approach is required to implement all foreseen lifecycle transitions.

Advantages of an end-to-end DevOps deployment model

The advantages of adopting an end-to-end DevOps deployment model are illustrated in the following figure:


ALM
Orchestration
Service Lifecycle Management
NFV DevOps Toolchain
Cloud orchestration and fulfilment tools provision network applications into life
Full operational lifecycle automation of Services and VNFs delivers additional lifecycle automation
End to end DevOps tool chain from initial onboarding to operational lifecycle management
Lifecycle Automation
20%
0%
100%
50%
80%

Deployment scenarios

You can find a deployment example in the "Functionality" on page 6 section, which depicts an example scenario of a video streaming service.

Case study: Heal operation using Agile Lifecycle Manager

In the following 'closed loop heal' case study, a server no longer receives IP packets. An alarm is raised, the event is evaluated, and a Heal event is triggered and executed by Agile Lifecycle Manager. The solution addressing this case study is comprised of the following components (some of which are bundled as IBM Netcool Operations Insight):

- IBM Tivoli Netcool/OMNIbus and Web GUI
- IBM Tivoli Netcool/Impact
- IBM Netcool Agile Service Manager
- IBM Agile Lifecycle Manager

Heal solution process, step-by-step

Note: This case study refers to the 'operations' and 'external OSS' actors, and their interaction. These are defined in the deployment overview here and here.

1. Tivoli Netcool/OMNIbus (acting as **external OSS**) receives an alarm that a server has stopped receiving IP packets.
2. This alarm then triggers a Tivoli Netcool/Impact policy.
3. As part of the service design and assembly onboarding, Netcool Agile Service Manager reconciles the Agile Lifecycle Manager assemblies with OpenStack virtual machines.
4. Tivoli Netcool/Impact (acting as **operations**) now has enough knowledge to trigger an Agile Lifecycle Manager Heal request.
5. Agile Lifecycle Manager transitions the 'broken' server through the following lifecycles:
   a. STOP
   b. START
   c. INTEGRITY
6. Netcool Agile Service Manager gets notified by Agile Lifecycle Manager that a lifecycle event has occurred on the 'broken' server.
7. This triggers the Netcool Agile Service Manager observer that raised the alarm to rerun.
8. Rerunning the observer clears the Tivoli Netcool/OMNIbus alarm, confirming that the server is receiving packets again, and 'healing' was successful.

Related information:

IBM Netcool Operations Insight

IBM Tivoli Netcool/OMNIbus and Web GUI

IBM Tivoli Netcool/Impact

IBM Netcool Agile Service Manager

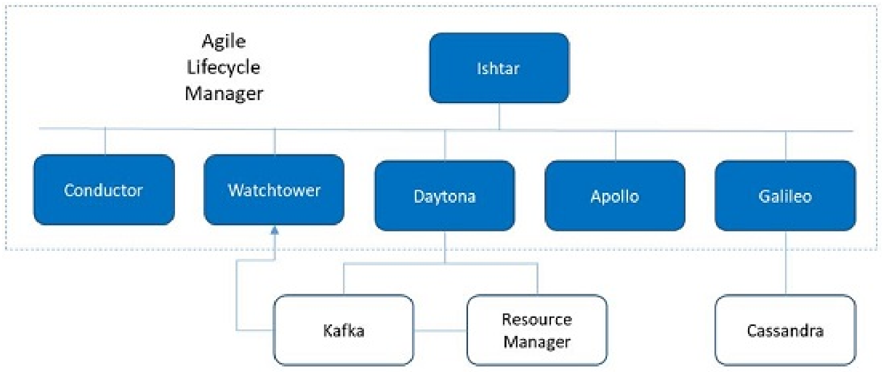
Knowledge center links

Architecture

This topic provides an overview of the Agile Lifecycle Manager architecture.

Basic architecture

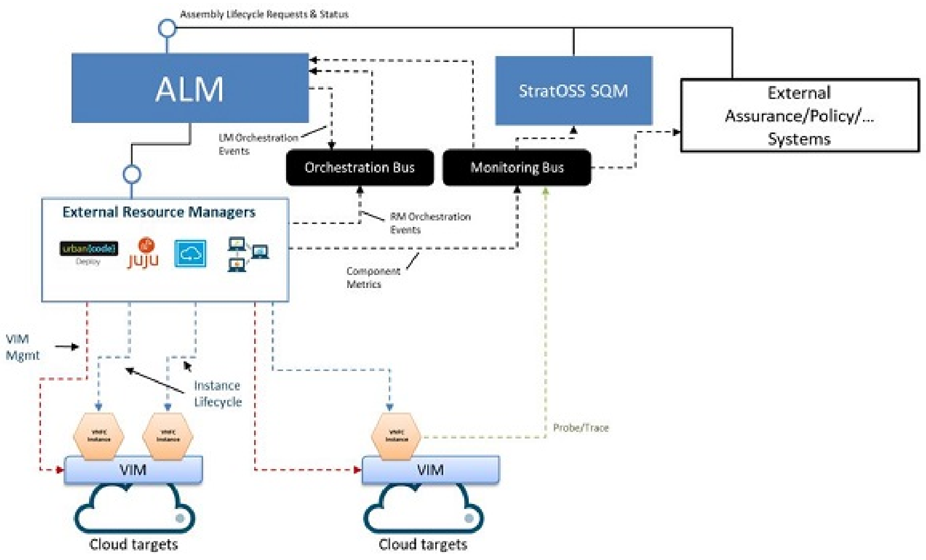
The basic Agile Lifecycle Manager architecture is depicted in the following figure:



Data flow

Agile Lifecycle Manager receives requests through its north bound API to put an assembly instance representing a VNF or service into an intended state, such as 'active'. Agile Lifecycle Manager in turn orchestrates all external resource managers through their northbound APIs to configure their managed resource instances accordingly. Throughout this orchestration period Agile Lifecycle Manager and each resource manager publish orchestration status events to a Kafka topic for each assembly and resource instance state change.

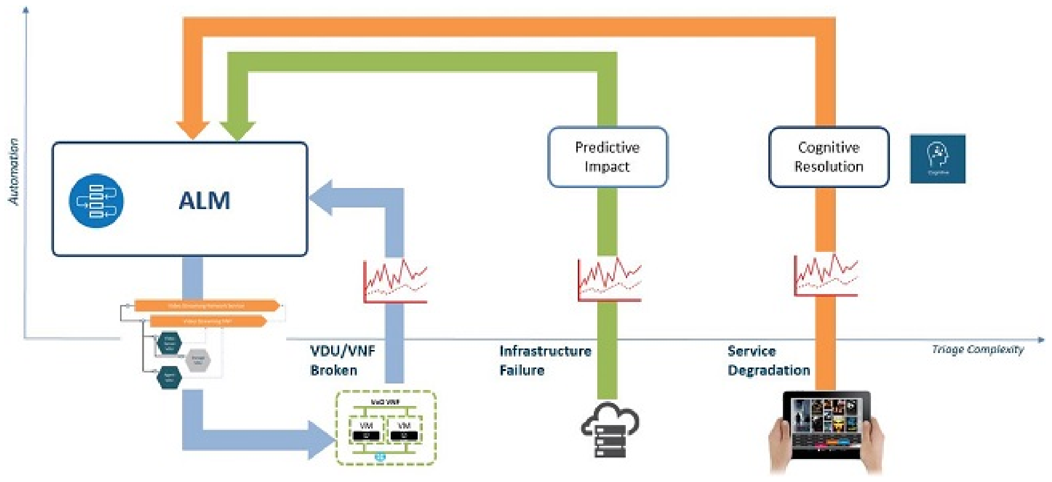
This process is depicted in the following figure:



Resource managers are responsible for orchestrating virtual infrastructure managers (VIM) to control cloud infrastructure compute, storage and network resources in support of their resource instances' standard lifecycles.

Extending Agile Lifecycle Manager

In addition to published orchestration events, performance and quality metrics may be published to a dedicated monitoring Kafka topic. These are consumed by automatic scaling and healing policies configured in Agile Lifecycle Manager. Agile Lifecycle Manager can be integrated with various external systems responsible for monitoring different aspects of an end-to-end service. These external systems, such as SQM or other assurance and analytics systems, can thus be extended to perform the following types of healing using the Agile Lifecycle Manager northbound API:



Broken virtual resources
: Resource instances that are not performing within acceptable levels are identified as broken and healed by Agile Lifecycle Manager.

Infrastructure failure
: The impact on resource instances by a physical infrastructure is reported and appropriate healing or migration is performed by the Agile Lifecycle Manager.

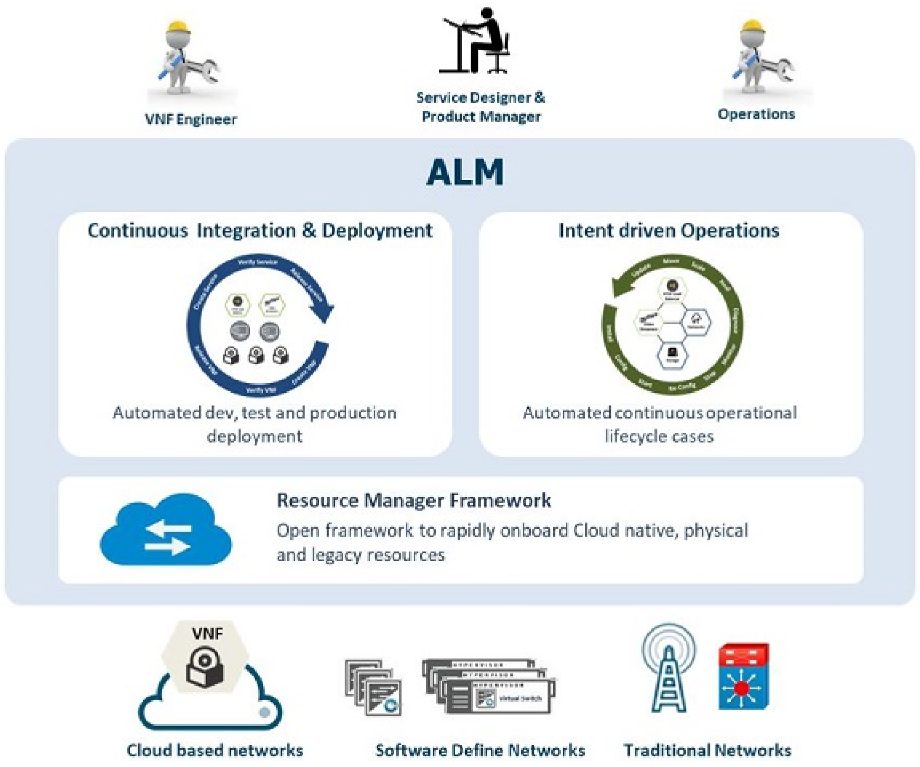
Service degradation
: Reduced customer experience or service quality can trigger scaling events.

Functionality

Agile Lifecycle Manager provides continuous integration and deployment of resources, intent-driven operations to automate lifecycle processes, and an open framework.

Overview

The following figure depicts the main functional capabilities of Agile Lifecycle Manager.

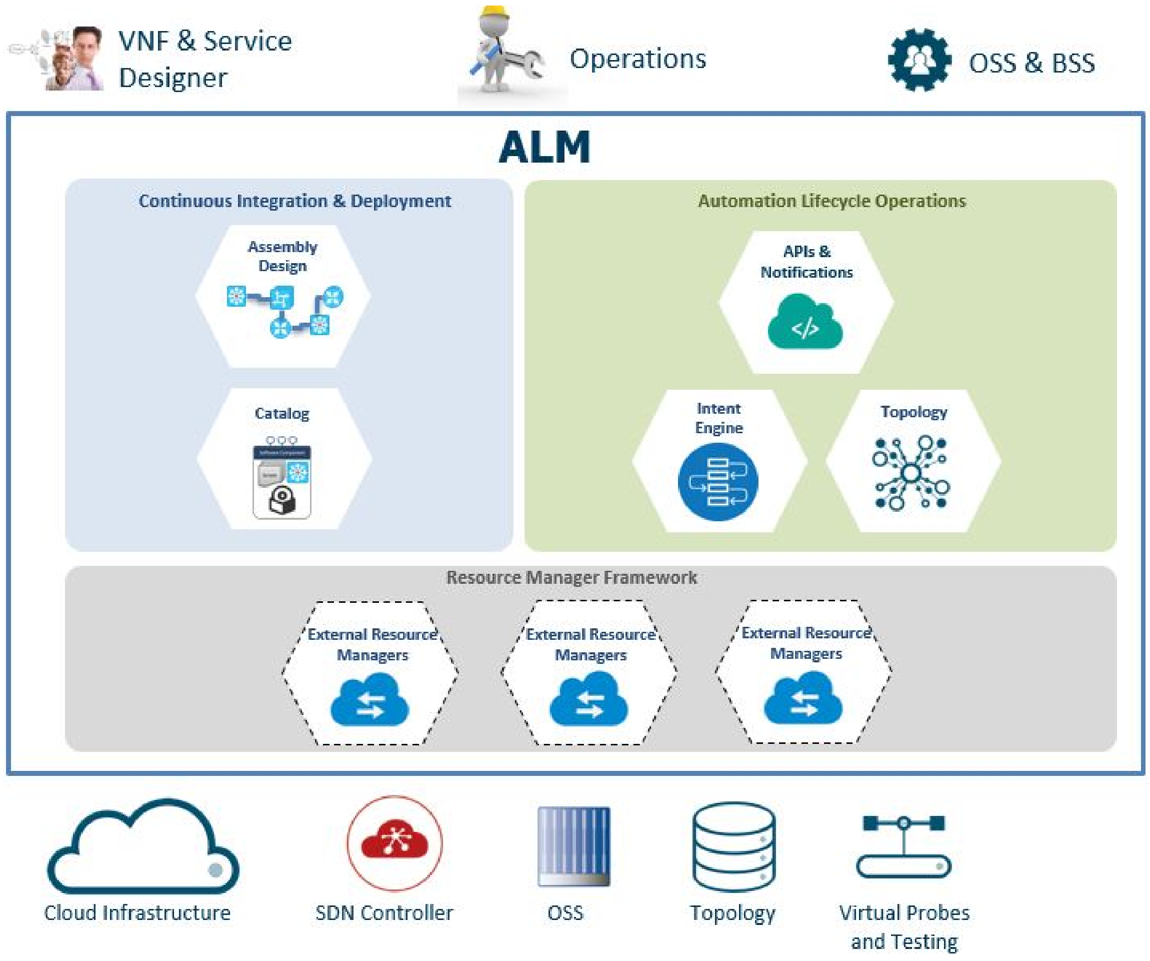

VNF Engineer
Service Designer & Product Manager
Operations
ALM
Continuous Integration & Deployment
Automated dev, test and production deployment
Intent driven Operations
Automated continuous operational lifecycle cases
Resource Manager Framework
Open framework to rapidly onboard Cloud native, physical and legacy resources
VNF
Cloud based networks
Software Define Networks
Traditional Networks

Continuous integration and deployment
: Provides rapid design and onboarding of external resources into production services.

Intent-driven operations
: Enable automated management of the end-to-end service lifecycle processes.

Resource Manager framework
: Makes possible open integration of external virtual and physical resources to be assembled with others into complete services.

These capabilities are broken down into the software functions depicted in the following figure.

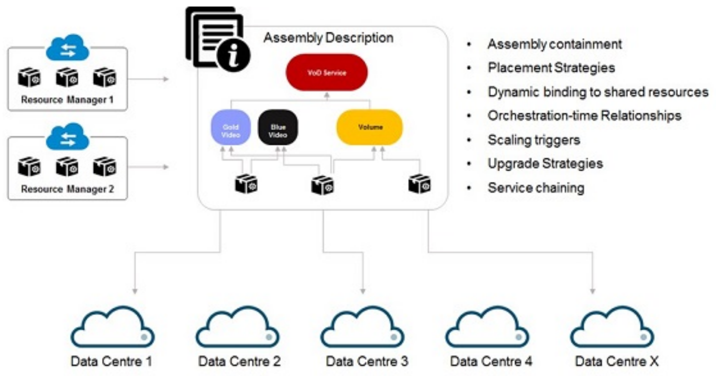

VNF & Service Designer
Operations
OSS & BSS
ALM
Continuous Integration & Deployment
Assembly Design
Catalog
Automation Lifecycle Operations
APIs & Notifications
Intent Engine
Topology
Resource Manager Framework
External Resource Managers
External Resource Managers
External Resource Managers
Cloud Infrastructure
SDN Controller
OSS
Topology
Virtual Probes and Testing

Continuous integration and deployment

In order to deliver an assembled end-to-end service, the assembly design function provides a set of tools that enable the rapid description of complex bundles of external resources and their combined operational processes.

Assembly descriptors are written in YAML and stored in the Catalog.

The following figure illustrates the assembly descriptor attributes.

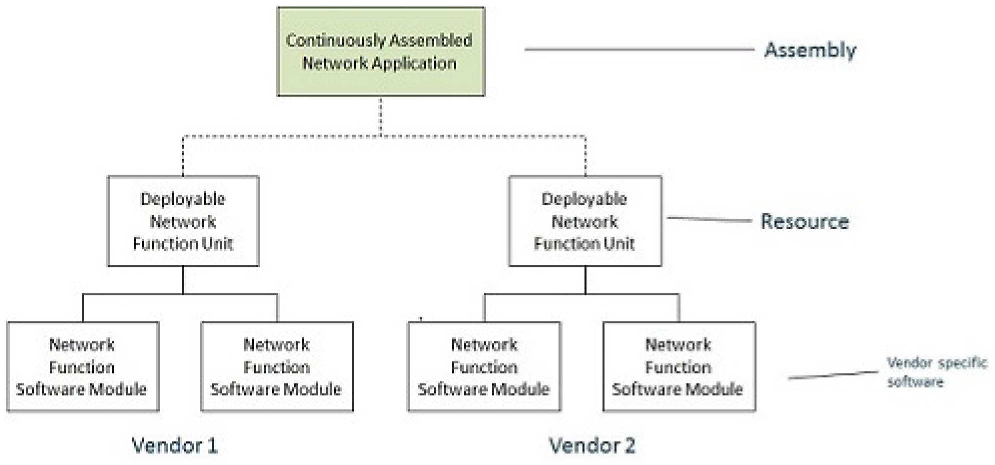


Descriptors specify the deployment and operational information required to allow the Intent Engine to instantiate a set of external resources, in the right location and with their inter-dependencies in place. Assembly descriptors contain attributes, versions, associated external resources and their relationships and deployment locations.

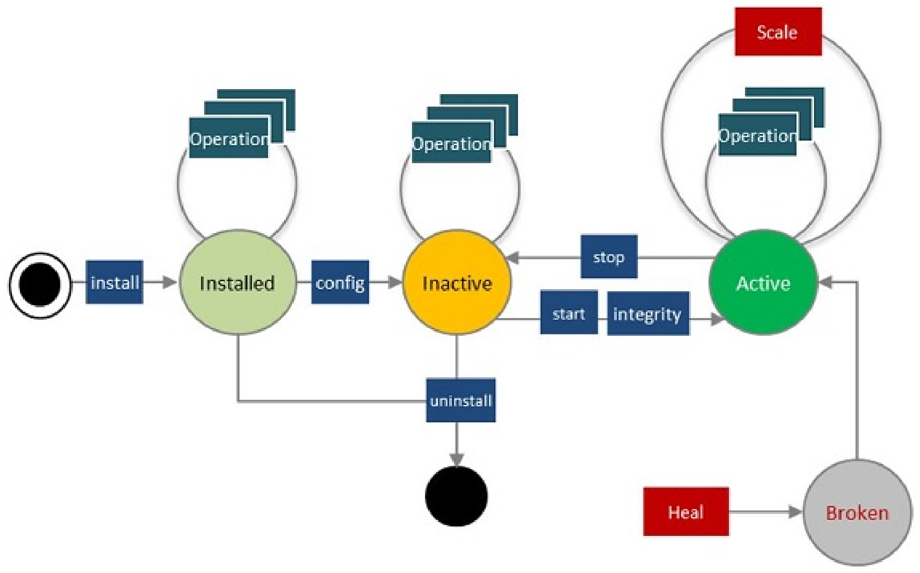
For more information on YAML specifications, see the following topic: “Assembly descriptor YAML specifications” on page 147

Resource Managers expose resource descriptors to Agile Lifecycle Manager. These resource descriptors represent deployable units of software whose lifecycles can be manipulated by Agile Lifecycle Manager. Resource Managers in turn are responsible for exposing a lifecycle interface for each resource instance it manages and managing any infrastructure required to support these resources instances.

The following figure shows the main entities that Agile Lifecycle Manager employs to model external resources that are assembled into compound services. Typically, software modules are compiled by an independent software vendor into external functional units that can be deployed independently and assembled into an application or service at a later stage. Each external deployable unit is considered a **resource** which has its own standard lifecycle. These resources are continuously assembled and re-assembled into logical applications or services that can span multiple data centers. These are represented in Agile Lifecycle Manager as **assemblies**.


Continuously Assembled Network Application
Assembly
Deployable Network Function Unit
Deployable Network Function Unit
Resource
Network Function Software Module
Network Function Software Module
Network Function Software Module
Network Function Software Module
Vendor specific software
Vendor 1
Vendor 2

Assemblies are composed of resources or other assemblies, and all resources and assemblies must support the same standardized lifecycle model as depicted in the following figure, to allow Agile Lifecycle Manager to dynamically manipulate resources into higher order services.



Each resource and assembly must support a standard lifecycle, which includes the following:

Standard lifecycle transitions

Each of the dark blue transitions in the preceding figure represents a software executable that is intended to bring its resource or assembly instance from one state to another.

The exception to this is the dark blue **integrity** transition, which provides a basic test primitive that is called periodically or explicitly after the start transition to ensure the resource is in an operational state.

In-life operations

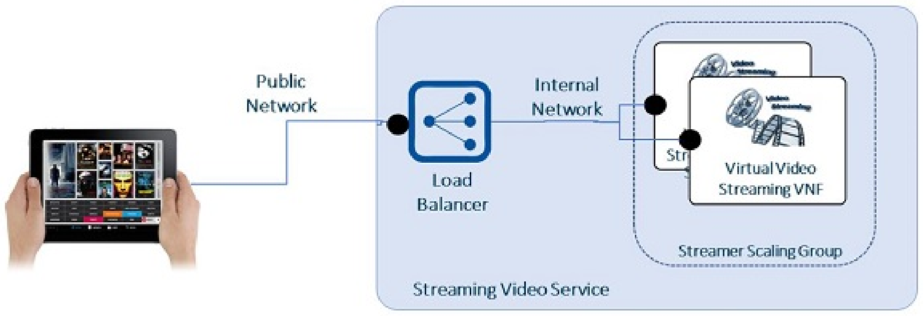
In addition to standard transitions, resources can optionally provide ad-hoc software executables that directly represent a specific resource use case, for example adding a user.

Also, in-life operations can be provided as the implementation of one end of a relationship.

Opinionated patterns

Set sequences of transitions are run to accommodate special scenarios such as scaling an assembly, or healing a resource that is in a state of error.

All resources must implement each standard lifecycle transition, and also (optionally) in-life operations. The intent engine coordinates all assembly and resources lifecycle transitions automatically along with any opinionated patterns, as required, and as depicted in the following figure of an example scenario. This figure depicts a video streaming service that includes several resources assembled into a deployment model.



In this example service, several external resources provided by different software or infrastructure providers are assembled:

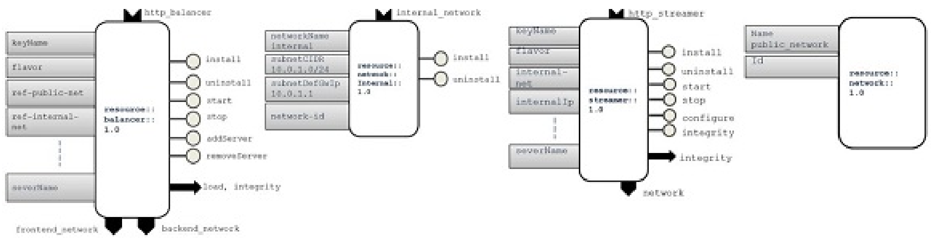
Load balancer resource
Manages the distribution of video traffic across several video streaming instances.

Video streaming VNF resource
Streams video traffic.

Internal network resource
Internal network to connect service resources.

Public network resource
External network connecting end users to the service.

The assembly descriptor that models the end-to-end service composes resource descriptors, which in turn describe their individual lifecycle aspects.



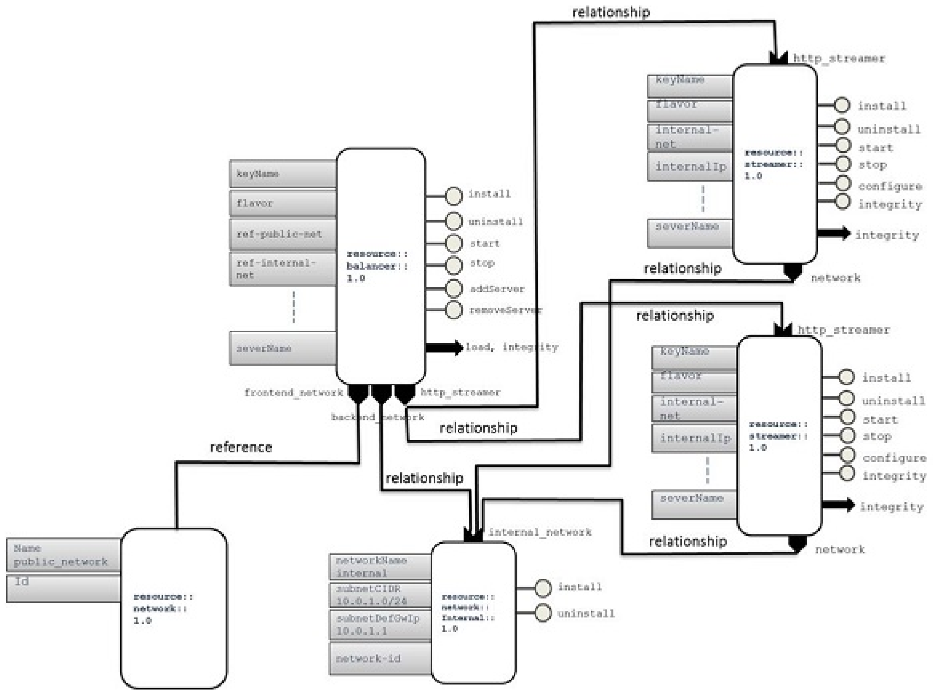
Each assembly and resource descriptor can model the following:

Required properties
Expected properties required by the assembly or resource.

Each can have defaults and read-only properties that allow resource instances to provide instance specific data, such as IP addresses.

Lifecycle actions
List all the standard transitions supported.

In-life operations
List of ad-hoc operations with individual properties for each.



In addition to the common descriptions listed above, assemblies can also model the following:

Composition
Group of assembly or resource children types that are included in this assembly type.

Property dependencies
Assemblies or resources can wait for properties on other resources or assemblies to be populated.

Relationships
Operational relationships between children assemblies or resources.

These relationships can be instigated on specific states of its endpoints and calls in life operations to execute.

References
References to assemblies or resource instances outside this assembly can be declared.

Assembly and resource descriptors are described in more details in the following topic: "Assembly descriptor YAML specifications" on page 147

The Catalog:

Continuous integration and deployment depends on the Catalog to manage the details of all types of descriptors, such as the assembly descriptors and resource descriptors, which are stored in the Catalog. Descriptors are used by the Operation Support System (OSS) to create new applications or services. Also, when the intent engine is asked to instantiate an assembly, it will request the descriptor from the Catalog.

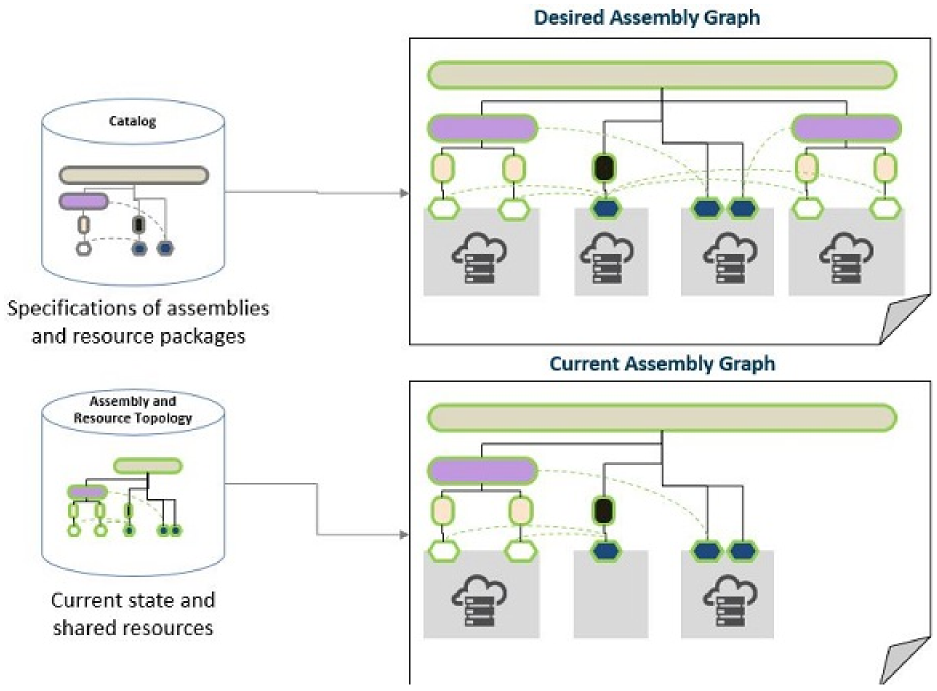
Automated lifecycle operations

API and notifications: Agile Lifecycle Manager applications adopt the microservices architecture so they communicate via APIs. Each microservice has a well-defined API representing atomic functionality. The components also communicate via notifications published to and consumed from the service bus (Kafka). Whenever the state of an assembly (or component) changes, the system publishes related event data onto the bus that can be consumed by other modules.

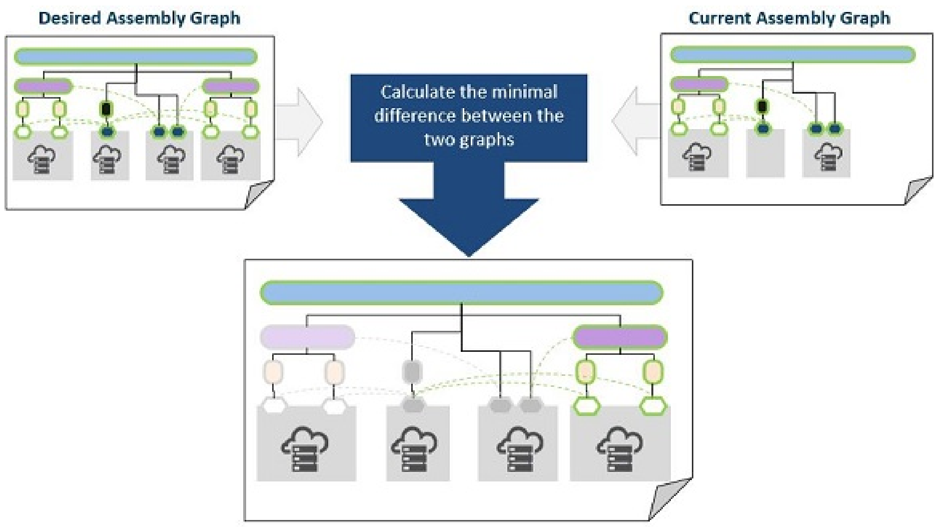
Tip: You can also use Kafka outside the Docker containers.

Agile Lifecycle Manager also interworks southbound and northbound via well-defined APIs. (See the Reference section for more information on API specifications.)

Intent engine: The Intent Engine is the functional entity that takes assembly and resource descriptors from the catalog and auto-generates the processes required to manage the complete lifecycle of a service and all of its constituent parts and their relationships to each other.


Desired Assembly Graph
Catalog
Specifications of assemblies and resource packages
Current Assembly Graph
Assembly and Resource Topology
Current state and shared resources

The intent engine generates process execution plans after receiving requests through the published API. The intent engine retrieves the assembly descriptor from the catalog and builds a complete graph of the desired state for the entire service and resources, resolving shared resources and placement strategies.



The desired graph is enriched with information about existing assembly or resource instances and updated to reflect the changes required to move the current service graph to the desired service graph. These changes form the basis for an execution plan that coordinates the lifecycle of those new and existing resources involved in the service graph.

The intent engine instructs, step by step, the Resource Manager(s) via API to execute the plan. The intent engine stores all assembly related changes in the topology.

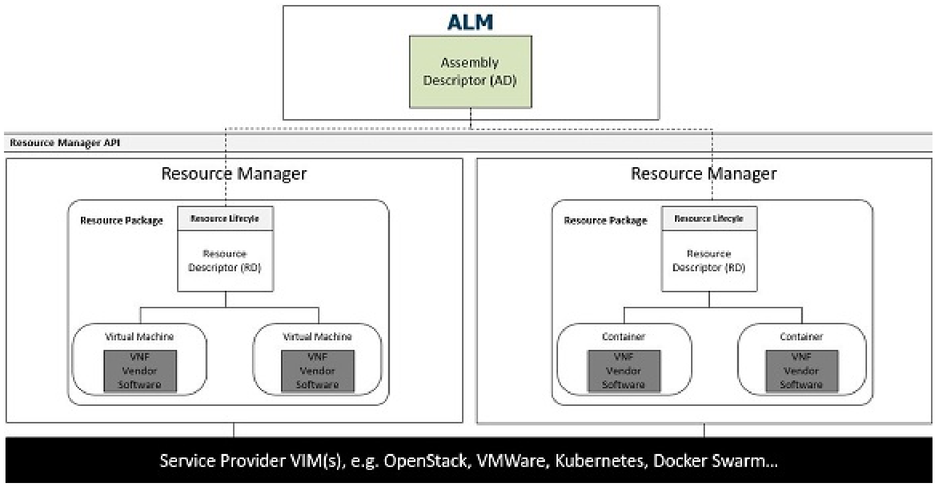
The intent engine interacts with the Resource Manager via API for discovering, configuring and manipulating resources.

The intent engine has several opinionated patterns to support the healing of broken resources:
- A resource put into the broken state on receipt of a Heal request is progressed through the stop, start, and integrity transitions to attempt to return it to the active state.
- If Heal is unsuccessful, the resource is left in the state prior to the failed transition and the Heal request returns as failed.

Topology: The Topology function stores details of assembly and resource instances. When the intent engine initiates an action on an assembly instance the details of the request will be stored in the topology. Any details that are part of the output from the operation will also be stored by the intent engine. The topology stores the history of all changes made to an assembly instance. The topology also manages the state of a service with regards to the requests.

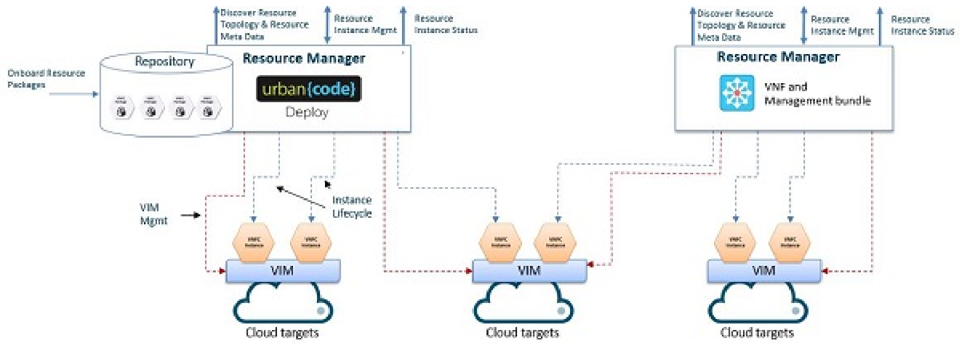
Resource Manager framework

The Resource Manager framework provides an open set of tools to allow VNF vendors to wrap their software in a standardized lifecycle that can be manipulated by Agile Lifecycle Manager. Proprietary VNF managers or general-purpose software managers, such as Canonical's JuJu Charms, IBM's Urbancode, or RedHat's Ansible, can be integrated to allow the virtual or physical resources they manage to be discovered and manipulated by Agile Lifecycle Manager.



Resource Managers adhere to an API that allows Agile Lifecycle Manager and other external systems to discover the data center topology supported by the resource manager instance and the resource types it supports. The API also provides the ability to manipulate and monitor the state and health of each resource instance.

The following figure depicts Agile Lifecycle Manager's ability to manipulate resources from many resource managers. Each resource manager manages the lifecycle of several virtual or physical devices and the underlying virtual or physical infrastructure.



Resource managers must abide by the following use cases and requirements:

- Multiple Resource Managers can manage resource instances on the same VIM(s)
- Resource Managers can manage one or more VIMs
- Component instances are managed by a single Resource Manager
- VNF types are registered with Resource Manager types
- Resource Manager instances are registered to work with one or more VIMs
- Instances of VNF types can be deployed to multiple VIMs by the registered Resource Manager VIMs
- Multiple Resource Managers can manage VNF types on a single VIM

The Resource Manager framework includes the following artifacts to support the above use cases and requirements:

Swagger API definition and specification
    Rest and Kafka API semantics and messages

Resource descriptor specification
Descriptor specification

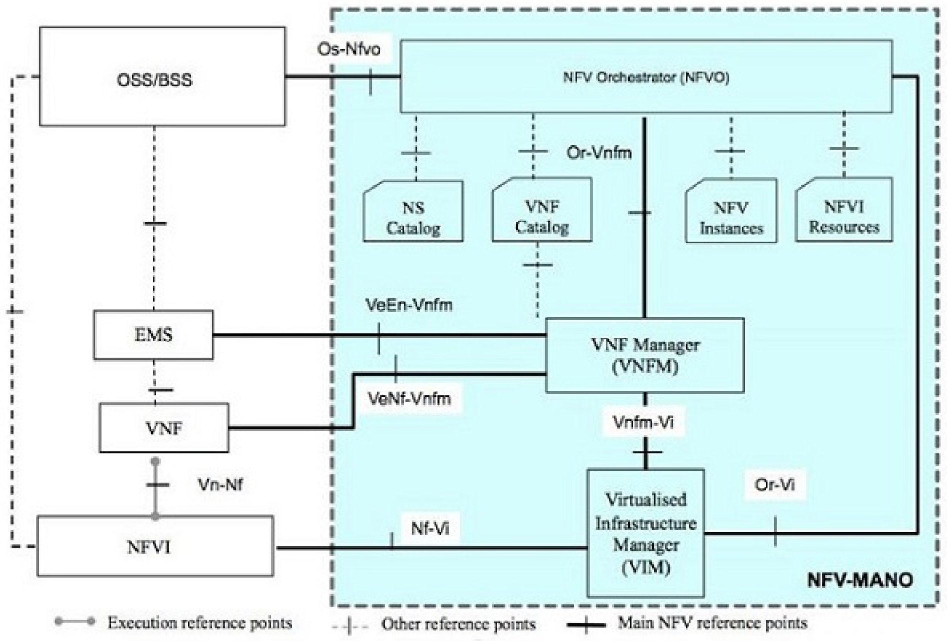
Resource archive format
Standard and portable packaging format for bundling software, lifecycle and operation scripts, descriptors and resource manager configuration

API drivers
Integrations for popular software management systems

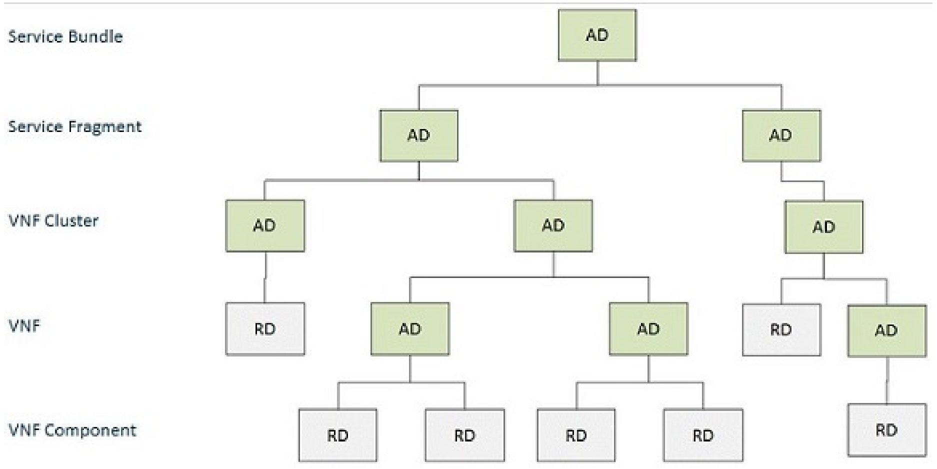
Resource Manager API: The Resource Manager API is responsible for defining the interactions between a lifecycle manager and the resource managers used to manage resources within virtual (or physical) infrastructures.

Mapping to industry standards

In recent years the industry has been actively generating standards for NFV. NFV specifications are published on a regular basis by various industry forums. Those include ATIS, Broadband Forum, ETSI NFV ISG, IETF, ONF and others. In parallel, open source projects have been established to accelerate NFV adoption. The most recent and notable initiative within the orchestration area is the Open Network Automation Platform (ONAP) that is joining two projects: The Enhanced Control, Orchestration, Management and Policy (ECOMP) and the Open Orchestrator (Open-O). IBM closely monitors relevant forums and partners with key industry contributing members.



ETSI has introduced a number of key concepts that provide a language for describing an NFV environment. The following figure shows a mapping of Agile Lifecyle Manager concepts to ETSI definitions. ETSI terminology/concepts are shown on the left of the figure mapped to IBM's core concepts, that is, Assembly Descriptor (AD) and Resource Descriptor (RD).



How a VNF vendor has engineered their software will determine how many resource descriptors it presents to Agile Lifecyle Manager. For example, a native Cloud style VNF implementation could provide many Resource Descriptors representing micro-services that are assembled into VNF components, which are in turn assembled into VNFs. Conversely, a VNF vendor may provide a single resource representing a complete VNF function. These resource descriptors in any case are re-assembled into an architecture specific to the service providers' environment and service design, once again by layering assembly descriptors and relationships.

Related concepts:

"Assembly descriptor YAML specifications" on page 147
This section describes the assembly descriptors that are used by Agile Lifecycle Manager.

"Lifecycle Manager API" on page 93
The Lifecycle Manager API is responsible for interactions with the operations available from Agile Lifecycle Manager. This section covers the definition of the Lifecycle Manager API and the specification of the messages sent across this interface.

Chapter 7, "Reference," on page 89
Use the following reference information to enhance your understanding of the Agile Lifecycle Manager APIs and YAML specifications.

Related reference:

"Asynchronous state change events" on page 108
Agile Lifecycle Manager will emit events when the state of an assembly or its components changes. Messages that are sent asynchronously are put onto a Kafka bus. The exact topics can be configured. These are emitted in response to Intent Requests causing the state of the Assembly Instance, or its associated components, to change. In the event of a failure to change state, an event will also be emitted.

Related information:

IBM overview of microservices

IBM Cloud Private components

The ICP version of Agile Lifecycle Manager consists of a number of services packaged as Helm charts.

Agile Lifecycle Manager download packages

The IBM Cloud Private version of Agile Lifecycle Manager consist of the following components:

Table 1. Core Agile Lifecycle Manager components for ICP

Package	**Details**
Daytona	Service responsible for the orchestration of lifecycle requests on service instances
Galileo	Service responsible for storing assembly topologies
Conductor	Management service containing configuration server, service registry, etc
Apollo	Repository service for service specifications. Contains both components from resource managers and services created through the designer.
Ishtar	North-bound public API into the Agile Lifecycle Manager
Watchtower	Service responsible for managing the health and policies of resources and assemblies
Nimrod	Service supporting the Agile Lifecycle Manager Graphical User Interface (GUI)
Relay	A secure internal proxy server to resolve inter-domain API calls.
Talledega	A service that stores Daytona process information.

Table 2. StatefulSet objects for Agile Lifecycle Manager on ICP

Package	**Details**
Cassandra	A distributed and robust database that is scalable while maintaining high performance.
ElasticSearch	A distributed search and analytics engine that is scalable and reliable.
Kafka	A message bus that efficiently consolidates data from multiple sources.
Zookeeper	A robust, distributed and scalable synchronization service.

Glossary

Refer to the following list of terms and definitions to learn about important Agile Lifecycle Manager concepts.

Agile Lifecycle Manager terminology

assembly
An assembly is a definition of a service and may comprise of one "resource (or component)" on page 22, (or more than one resource), and/or other assemblies.

It is defined in an assembly descriptor and can be instantiated as an assembly instance.

assembly descriptor (or descriptor)
An assembly descriptor is a computer-readable definition of an assembly implemented as a YAML file.

assembly designer (or service designer)
An actor or end user role designing services using Agile Lifecycle Manager. An assembly designer takes informal service design artifacts defined by service designers and translates them to a set of formal computer-readable descriptors that model the target service.

assembly instance
Instantiation of an assembly descriptor and all the composed resources or assemblies.

capability
Capabilities is a section of an assembly descriptor or a resource descriptor defining what functions the resources or assemblies are implementing.

catalog (or assembly catalog)
The repository within Agile Lifecycle Manager storing published assembly descriptors and resources descriptors.

cloud
The cloud is a common term referring to accessing computer, information technology (IT), and software applications through a network connection, often by accessing data centers using wide area networking (WAN) or internet connectivity.

CSAR (or archive)
Cloud service archive (CSAR) describes a format used for describing resource packages.

CSAR specification is a part of OASIS TOSCA.

deployment location
Deployment location is a facility where resources can be deployed while they are instantiated.

In various contexts deployment locations are referred to as data centre, project (OpenStack), or availability zone (OpenStack).

descriptor
See "assembly descriptor (or descriptor) "

forwarding graph
See service chain.

intent engine (or engine)
The entity responsible for generating the assembly deployment plan and instructing, step by step, resource managers to execute the plan.

Kafka Apache Kafka is a distributed streaming platform.

Tip: Streams of records are stored in Kafka in categories called 'topics'.

See the related links for more information.

lifecycle event (or event)
Agile Lifecycle Manager published intent and status change event onto a Kafka topic.

lifecycle state (or state)
A lifecycle state defines the state of a specific resource instance or assembly instance.

Examples of lifecycle states include: Installed, Inactive, Active, Broken, and Failed.

Changes from one lifecycle state to another are lifecycle transitions.

lifecycle transition (or transition)
A lifecycle transition is a process aiming to change the lifecycle state of an assembly or resource.

Lifecycle transitions are initiated through the Agile Lifecycle Manager API, orchestrated by Agile Lifecycle Manager and executed by the underlying resource managers.

Examples of lifecycle transitions include: install, configure, start, stop, integrity, and uninstall.

microservice
Microservices are a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services.

The benefit of decomposing an application into different smaller services is that it improves modularity and makes the application easier to understand, develop and test.

It also parallelizes development by enabling small autonomous teams to develop, deploy and scale their respective services independently.

migration
Migration is one of the opinionated patterns aiming to migrate a deployed NVF from a location to another.

monitoring metrics
Performance or health metrics published by resource managers and/or resources onto a Kafka topic.

network
A type of resource.

network function virtualization (NFV)
The design and integration of external resources into virtual production environments, which can then automate the management of end-to-end lifecycle processes.

Also see virtual network functions.

OASIS
OASIS is a non-profit consortium that drives the development, convergence and adoption of open standards for the global information society.

onboarding

Onboarding is the act of adding a resource manager to Agile Lifecycle Manager. It lets Agile Lifecycle Manager know that the resource manager exists, and it imports the descriptors of all the resource types managed by the resource manager. It also gathers the information about the deployment locations that the resource manager uses.

operations
'Operations' is a section of an assembly descriptor or a resource descriptor that defines sets of operations, which can be called to enable relationships to be created between resources and/or assemblies.

opinionated patterns
The group of lifecycle transitions to achieve a particular task.

Examples of tasks include: heal, reconfigure, and upgrade.

policies
'Policies' is a section of an assembly descriptor or a resource descriptor containing the set of policies that are used to manage the assembly or resource instances.

property
'Properties' is a section of an assembly descriptor or a resource descriptor containing the properties that belong to the resource or assembly descriptors.

These include the full set of properties that are required to orchestrate them through to the active state.

These can be understood as the 'context' for the management of the item during its lifecycle.

quality monitoring
Quality Monitoring is a process to monitor the health of deployed resources and NFV infrastructure and to test, monitor and evaluate the end-to-end service performance.

reference
'Reference' is a section of assembly descriptor or resource descriptor.

When the Agile Lifecycle Manager has already instantiated an assembly it is possible for another assembly to share the instance by referencing it within the references section.

The references section can also refer to existing objects that may have been created outside the Agile Lifecycle Manager.

relationship
'Relationships' is a section of assembly descriptor or resource descriptor.

Relationships define how the descriptors link requirements to capabilities.

A relationship has source-capabilities and target-requirements as parts of its description.

requirement
'Requirements' is a section of an assembly descriptor or a resource descriptor explaining what functions the resources or assemblies need before they can work successfully.

resource (or component)
A piece of software that can be automatically deployed in a virtual environment, and that supports key lifecycle states including install, configure, start, stop, and uninstall.

resource descriptor
The list of resource attributes and properties written in YAML.

resource health (or component health)
Resource health is a microservice within Agile Lifecycle Manager responsible for monitoring health-related messages and initiating recovering actions related to deployed resources.

For example, the resource health may send a 'heal' message to the intent engine if a certain event indicating health issues is detected.

resource instance
A resource instance represents the logical grouping of infrastructure being managed by an external resource manager.

resource manager
The entity instructing resources.

For example, IBM UrbanCode.

resource manager record

Agile Lifecycle Manager maintains a record of each resource manager it can use to create and manage resources. When a new resource manager record is created, the resource types managed by that resource manager instance are read into Agile Lifecycle Manager via the resource manager's API.

resource package
Resource package is described as a CSAR archive.

This is the bundle of everything needed for a resource that is loaded into a resource manager.

scale Scale is one of the opinionated patterns aiming to increase or decrease the amount of deployed resources of a specific type.

service chain
Instantiated as relationships in assembly descriptors.

TOSCA
Topology and orchestration specification for cloud applications (TOSCA) is a standard defined by OASIS.

topology (or instance inventory)
The repository storing key state information related to assembly and resource instances and topology of the deployment locations.

virtual infrastructure manager
The entity controlling the cloud infrastructure compute, storage and network resources.

For example, OpenStack.

virtual network functions (VNF)

Virtual network functions (VNF) allow a much simpler set of lifecycle tasks enabling near full automation of the creation and healing of virtual services, far more than is possible with their physical counterparts.

Also see network function virtualization.

Related information:

Kafka documentation (web link)

Chapter 2. Planning

This section helps you to plan your installation and use of Agile Lifecycle Manager by listing the minimum software and hardware requirements for both on-premise and IBM Cloud Private versions of Agile Lifecycle Manager.

Hardware requirements

This section lists the minimum hardware requirements for a deployment of Agile Lifecycle Manager.

Your minimum hardware requirements are determined by the needs of the components of your specific solution. The requirements listed here focus on what you need to deploy Agile Lifecycle Manager.

Table 3. **IBM Cloud Private** *Agile Lifecycle Manager hardware requirements*

Requirement	**Setting**
CPU	16 cores
Memory	32Gb
Disk	1Tb

Software requirements

This section lists the minimum software requirements for a deployment of Agile Lifecycle Manager.

The IBM Cloud Private version of Agile Lifecycle Manager has the following software requirements.

Table 4. **IBM Cloud Private** *Agile Lifecycle Manager software requirements*

Requirement	**Details**
Operating system	IBM Cloud Private Version 3.1.0 or later
Database cluster	Cassandra cluster
Messaging	Kafka messaging solution

IBM Cloud Private deployment overview

You can install Agile Lifecycle Manager within a private cloud, using IBM Cloud Private. This topic describes the default container architecture of an ICP deployment.

Agile Lifecycle Manager components and services run within containers, and communication between these containers is managed and orchestrated using IBM Cloud Private running on a Kubernetes cluster.

IBM Cloud Private (ICP)
: The IBM Cloud Private system on which the Kubernetes cluster is deployed.

Managing the cluster

The cluster is made up of a minimum number of virtual machines deployed as a master node (which include management, proxy and boot functions), and three worker nodes within the cluster (on which Kubernetes pods and containers, known as workloads, are deployed).

The components of Agile Lifecycle Manager are automatically deployed as pods running within the cluster.

You connect to the master node using the IBM Cloud Private GUI or the Kubernetes command line interface.

You connect to Agile Lifecycle Manager using a Web browser. The Agile Lifecycle Manager URL consists of the master node hostname and the port number.

Tip: You can create namespaces within your cluster. This enables you to have multiple independent installations within the cluster, each running in a separate namespace.

Storage

A persistent volume provides storage on demand to the various containers in the cluster.

The shared storage must be implemented using local storage.

Chapter 3. Installation

Agile Lifecycle Manager is now also available as an IBM Cloud Private (ICP) version. The ICP deployment and configuration of Agile Lifecycle Manager differs significantly from the on-premise version.

Installing and configuring on ICP

Agile Lifecycle Manager is distributed as a self-contained package delivered as a Helm chart. To install Agile Lifecycle Manager, you complete the required pre-installation tasks, then perform the installation.

Before you install (ICP)

Before installing Agile Lifecycle Manager on ICP, you perform a number of pre-installation checks and tasks.

Set up Kubernetes hardware architecture
Ensure that the Kubernetes hardware architecture on which IBM Cloud Private is installed is amd64.

Configure ports
Ensure all required ports are available for ICP installation.

See the following Knowledge Center page for more information: https://www.ibm.com/support/knowledgecenter/en/SSBS6K_3.1.0/supported_system_config/required_ports.html

Install IBM Cloud Private
Install IBM Cloud Private Version 3.1.0 or later.

Set the Enable sub-chart resource requests field to `false` in the IBM Cloud Private GUI during installation.

See the IBM Cloud Private Version 3.1.0 Knowledge Center: https://www.ibm.com/support/knowledgecenter/en/SSBS6K_3.1.0/kc_welcome_containers.html

Set up virtual machines
Ensure you have four virtual machines in your Kubernetes cluster:

- three worker nodes
- one master/management/proxy/boot node

Each node requires the minimum of 16 CPUs and 32 GB of memory.

Install client utilities
Ensure you have the Kubernetes and Helm client utilities installed on your master node.

Preparing your cluster

Perform these steps to prepare an Agile Lifecycle Manager cluster on IBM Cloud Private.

About this task

Follow the prerequisite steps outlined in the table to prepare a cluster for Agile Lifecycle Manager installation.

Table 5. Steps to prepare an Agile Lifecycle Manager cluster on ICP. This table outlines the steps to prepare an IBM Cloud Private cluster for Agile Lifecycle Manager installation.

Item	Action	More information
1	To support the cluster, provision at least four virtual machines: • 1 master/management/proxy/boot node • 3 worker nodes	For generic ICP requirements, see the following Knowledge Center topic: https://www.ibm.com/support/knowledgecenter/SSBS6K_3.1.0/supported_system_config/hardware_reqs.html
2	Download and install ICP.	For the download, see the IBM Support Download Document: For the installation, see the following section in the IBM Cloud Private Knowledge Center: https://www.ibm.com/support/knowledgecenter/en/SSBS6K_3.1.0/installing/installing.html
3	Install and configure the Kubernetes command line interface `kubectl` to enable command-line access to the cluster.	See https://www.ibm.com/support/knowledgecenter/SSBS6K_3.1.0/manage_cluster/cfc_cli.html
4	Install the ICP command line interface to enable command-line management of the cluster.	See https://www.ibm.com/support/knowledgecenter/SSBS6K_3.1.0/manage_cluster/install_cli.html
5	Familiarize yourself with the command-line interfaces that you will need to perform the installation and communicate with the cluster.	You can find more information at the following locations: • Helm CLI commands: Helm documentation: Commands • ICP CLI commands: ICP Version 3.1.0 documentation: CLI command reference • Kubernetes CLI commands: Kubernetes documentation: Overview
6	Use the `default` namespace in the cluster for your installation. **Optional**: If you want multiple independent installations within the cluster, then create custom namespaces within your cluster. Run each installation in a separate namespace. **Note:** Additional diskspace and worker nodes are required to support multiple installations.	See https://www.ibm.com/support/knowledgecenter/SSBS6K_3.1.0/user_management/create_project.html

What to do next

Load the Agile Lifecycle Manager archive into ICP.

Loading the Agile Lifecycle Manager archive into ICP

Run these commands to load the archive into ICP.

Before you begin

Before you perform this task make sure you have met the following prerequisites:

- You must have downloaded the eAssembly from IBM Passport Advantage. For more information, see the IBM Support Download Document.
- If you loaded any previous versions, then you must uninstall that version, including all images and Helm charts before loading the current archive into ICP. To do this, follow the "Uninstalling Agile Lifecycle Manager (ICP)" on page 33 instructions.

Procedure

1. Log into the cluster master node on ICP.
   a. Issue the following command:
      ```
      bx pr login --skip-ssl-validation -a https://IP-address:8443
      ```
      Where *IP-address* is the IP address of the master node.
   b. When prompted, specify the username and password for the master node. By default these are as follows:
      - Username: `admin`
      - Password: `admin`
2. Configure your cluster.
   ```
   bx pr cluster-config cluster-name
   ```
   Where *cluster-name* is the name of your cluster, as configured in "Preparing your cluster" on page 28. By default, the cluster name is `mycluster`.
3. Point the Kubernetes command line client `kubectl` at your cluster using the following command. Use this command syntax if you are installing in the default namespace, which is called `default`:
   ```
   kubectl config set-context cluster-name
   ```
   Where *cluster-name* is the name of your cluster, as configured in "Preparing your cluster" on page 28. By default, the cluster name is `mycluster`

   Note: If you are installing in a custom namespace, then you must also specify the name of the custom namespace, using the following syntax:
   ```
   kubectl config set-context cluster-name --namespace custom-namespace
   ```
   Where *custom-namespace* is the name of your custom namespace, as configured in "Preparing your cluster" on page 28..
4. Log into the Docker repository.
   ```
   docker login cluster-name.icp:8500
   ```
   Where *cluster-name* is the name of your cluster, as configured in "Preparing your cluster" on page 28. By default, the cluster name is `mycluster`

   When prompted, specify the username and password. By default these are as follows:
   - Username: `admin`
   - Password: `admin`

5. Unset the proxy connection. The proxy connection enables you to connect to the Internet using a proxy server. You must unset this for ICP to be able to work.

```
unset http_proxy
unset HTTP_PROXY
```

6. Load the Agile Lifecycle Manager archive into ICP.

```
bx pr load-ppa-archive --archive <alm_partnumber>.tar.gz
```

This command can take up to an hour to. The exact amount of time depends on your system resources.

Note: If you are installing in a cluster with a non-standard name or in a custom namespace, then you must also specify the non-standard cluster name and name of the custom namespace, using the following syntax. The square brackets must not be included; they are shown only to indicate that these parameters are optional:

```
bx pr load-ppa-archive --archive <alm_partnumber>.tar.gz [--cluster cluster-name]
 [--namespace custom-namespace]
```

Where the following parameters only need to be specified if you are using a non-default cluster or namespace:

- *cluster-name* is the name of your cluster, as configured in "Preparing your cluster" on page 28. By default, the cluster name is `mycluster`.
- *custom-namespace* is the name of your custom namespace, as configured in "Preparing your cluster" on page 28..

Provision storage for Agile Lifecycle Manager on ICP

The following procedure describes how to provision the storage required by Agile Lifecycle Manager on IBM Cloud Private.

About this task

To provision storage, you prepare your worker nodes, create a PersistentVolume yaml file, apply the PersistentVolume configuration, and then check the status of the PersistentVolume.

Restriction: As dynamic provisioning is not supported, you use local storage volumes for the Agile Lifecycle Manager StatefulSet applications.

nodeAffinity
The nodeAffinity configuration defines which worker node the storage is available on.

path The path configuration must be created before the volume can be used.

capacity
The capacity configuration needs to meet or exceed the storage requirements.

claimRef
The claimRef configuration needs to match the Agile Lifecycle Manager volume claims created.

Ensure that the namespace and release names are considered.

Procedure

1. Create the storage paths on the worker nodes. Run the following script **on each of your worker nodes** to create the required directories:

   ```
   for svc in cassandra kafka elasticsearch zookeeper ; do
     for num in 0 1 2; do
       mkdir -p /opt/ibm/alm/backup/$svc-$num;
       mkdir -p /opt/ibm/alm/data/$svc-$num;
       mkdir -p /opt/ibm/alm/logs/$svc-$num;
     done;
   done
   ```

2. Create a yaml file to define the PersistentVolume values. Change the following values:

   WORKER1
   Change this value to the IP address of the first worker node.

   WORKER2
   Change to the IP address of the second worker node.

   WORKER3
   Change to the IP address of the third worker node.

   RELEASENAME
   Change this value to the release name to be used when installing the Agile Lifecycle Manager chart.

   NAMESPACE
   Change this value to the namespace into which Agile Lifecycle Manager is installed.

   Example configuration file:

3. Apply the PersistentVolume configuration settings, as in the following example:

   ```
   $ kubectl apply -f my-alm-pv-config.yaml
   ```

   Where `my-alm-pv-config` is the name given to the configuration file created previously.

4. Check the availability of the storage volumes using the following script:

   ```
   $ kubectl get pv -l release=alm
   ```

   Where `alm` is the value previously defined for RELEASENAME. The system output will be similar to the following:

   ```
   NAME                      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM                                 STORAGECLASS    REASON    AGE
   alm-data-cassandra-0      50Gi       RWO            Retain           Available   default/data-alm-cassandra-0          local-storage             17h
   alm-data-elasticsearch-0  75Gi       RWO            Retain           Available   default/data-alm-elasticsearch-0      local-storage             17h
   alm-data-kafka-0          15Gi       RWO            Retain           Available   default/data-alm-kafka-0              local-storage             17h
   alm-data-zookeeper-0      5Gi        RWO            Retain           Available   default/data-alm-zookeeper-0          local-storage             17h
   ```

What to do next

You now install Agile Lifecycle Manager on ICP.

Installing Agile Lifecycle Manager on ICP

This topic describes how to install Agile Lifecycle Manager into IBM Cloud Private.

Before you begin

Ensure that your IBM Cloud Private cluster has Internet access.

If you are installing into a non-default namespace, ensure that the user you are deploying with has adequate privileges to perform the installation.

Note: If you are installing into a default namespace, your user profile will automatically have the correct privileges to perform the installation.

About this task

The following procedure describes how to edit the installation configuration file, and then how to install Agile Lifecycle Manager from the command line.

Procedure

1. Add the internal ICP Helm repository to the Helm configuration. This process is described in the following topic of the IBM Cloud Private Knowledge Center: https://www.ibm.com/support/knowledgecenter/en/SSBS6K_3.1.0/app_center/add_int_helm_repo_to_cli.html
2. Define the Agile Lifecycle Manager installation configuration file, for example, `alm-config.yaml`. **Example yaml file:**

```
global:
  image:
    repository: mycluster.icp:8500/default      # adjust if you importALM into
a different namespace
  ingress:
    api:
      enabled: true
    admin:
      enabled: false
    domain: ""
    tlsSecret: ""
  persistence:
    enabled: true
    useDynamicProvisioning: true
    storageClassName:
    storageSize:
      cassandradata: 50Gi
      kafkadata: 15Gi
      zookeeperdata: 5Gi
      elasticdata: 75Gi
  environmentSize: "size0"
```

3. Deploy the Agile Lifecycle Manager installation from the command line. The following installation script example assumes an IBM Cloud Private Helm repository named *`icp-repo`* and a configuration file called *`alm-config.yaml`*.

```
helm install icp-repo/ibm-netcool-alm-prod --name alm --values alm-config.yaml --tls
```

4. Verify that the installation is up and running.

   Example

```
kubectl get pod -l release=alm
```

   Where `alm` is the Helm release name for Agile Lifecycle Manager. The system should return information indicating that the pods have a status of `Running`.

   Example

```
helm list --tls
```

   This command returns a list of Helm deployments.

What to do next

You login to the Agile Lifecycle Manager ICP installation using a URL of the following format:

```
https://hostname_of_master_node:port_number/ibm/console
```

Where:

- *hostname_of_master_node* is the hostname or IP address of the master node of the Operations Management on IBM Cloud Private cluster.
- *port_number* is the port number of the application to which you want to log onto.

Uninstalling Agile Lifecycle Manager (ICP)

Uninstall Agile Lifecycle Manager by performing the following steps.

About this task

This procedure uninstalls the deployed version of Agile Lifecycle Manager, but does not remove the load images or charts.

Note: When you uninstall a release, any unused docker images that were used as part of the installation remain on the master node and on a local repository on one or more worker nodes. From time to time, Kubernetes removes unused images as part of its garbage collection process. For more information, see Configuring kubelet Garbage Collection.

Procedure

1. Run the following Helm command to determine which releases of Agile Lifecycle Manager are installed.
   ```
   helm ls --tls
   ```
2. Delete each release identified in the previous step by running the following Helm command against each release name in turn.
   ```
   helm delete --purge --tls release_name
   ```
   Where *release_name* is the name of one of the releases identified in the previous step.
3. Repeat the previous step until all of the releases are deleted.
4. Clean up storage:

   Example
   ```
   kubectl delete pvc -l release=alm
   ```
   Where `alm` is the Helm release name for Agile Lifecycle Manager.
5. Clean up remaining cron job objects and their related pods. After Agile Lifecycle Manager has been uninstalled, orphaned job objects and related pods may remain on the system. Remove these as in the following example:
   ```
   kubectl delete job -l release=alm --cascade
   ```
   Where `alm` is the Helm release name for Agile Lifecycle Manager.

Chapter 4. Using the UI

This section describes the Agile Lifecycle Manager User Interface, and the tasks it allows you to perform.

Related concepts:

Chapter 5, "Getting started (using the APIs)," on page 47
Agile Lifecycle Manager provides both a graphical UI and an HTTP API allowing the creation and administration of assemblies. This section describes a set of basic scenarios to get started using the APIs.

UI functionality

The Agile Lifecycle Manager UI is comprised of three separate tools, a descriptor editor, an operations console, and a resource manager controller.

Descriptor Editor (Editor view)

The Descriptor Editor allows users to access all descriptors, both onboarded resource descriptors and assembly descriptors stored in the Agile Lifecycle Manager catalog.

As resource descriptors are onboarded from resource managers owning and managing the actual resources, in the Agile Lifecycle Manager Descriptor Editor they can only be browsed and no changes can be made to them.

For assembly descriptors Descriptor Editor provides a wide set of tools from browsing to editing, uploading, and downloading assembly descriptors.

With Descriptor Editor you can perform the following tasks

- Browse descriptors
- Modify existing assembly descriptors
- Create a new assembly descriptor
- Upload a new assembly descriptor (including validation)
- Download a copy of an existing assembly descriptor to the local filesystem
- Remove assembly descriptors

Operations Console (Assemblies view)

The Operations Console allows user to operate and manage services, modeled as assemblies, through the full lifecycle from initial provisioning through lifetime operations all the way to their end of life.

With the Operations Console you can:

- Browse existing assembly instances
- Create new assembly instances
- Request lifecycle transitions on assembly instances
- Initiate automated healing of broken service components
- Scale In and Scale Out service components
- Uninstall Services

Resource Manager Controller (Resource Managers view)

The Resource Manager Controller enables users to manage connected resource managers and onboard new resource managers to Agile Lifecycle Manager.

Onboarding a resource manager makes Agile Lifecycle Manager aware of the underlying resource manager and onboards the associated resource descriptors to Agile Lifecycle Manager. Once the resource types are onboarded the corresponding resource descriptors can be viewed on Descriptor Editor and used as components in assembly descriptors. Already onboared resource managers can be refreshed to update Agile Lifecycle Manager with possible changes on resource manager configuration, or their managed resource types.

With the Resource Manager Controller you can perform the following operations on resource managers:

- Browse onboarded resource managers
- Refresh onboarded resource managers
- Introduce new resource managers to Agile Lifecycle Manager
- Remove obsolete resource managers from Agile Lifecycle Manager

Tip:

Calls to the Agile Lifecycle Manager API are made using either REST or RPC mechanisms, and each call returns an HTTP status code. You can find more detailed information on the API HTTP status code strategy here: "API HTTP status codes reference" on page 89

Logging into the ICP UI

You discover the Agile Lifecycle Manager ICP logon URL from the master node hostname and port number.

Before you begin

Ensure that the hostname in your URL resolves to an IP address. You can do this by querying a DNS server on your network, or by configuring the `/etc/hosts` file on your client machine. For example, in the example URL provided below, ensure that the hostname `netcool.master21.mycluster.icp` resolves to an IP address.

Procedure

1. Retrieve the hostnames.

   ```
   kubectl get ingress
   ```

   A Kubernetes ingress is a collection of rules that can be configured to give services externally-reachable URLs. Run the `kubectl get ingress` command to retrieve the hostname allocated by the Kubernetes Ingress controller to satisfy each Ingress. By default this command retrieves data similar to the following:

   ```
   NAME              HOSTS                          ADDRESS      PORTS     AGE
   master21-almgui   alm.master21.mycluster.icp     IP_address   80, 443   1d
   ```
2. Ensure that the hostname in your URL resolves to an IP address. You can do this using one of the following methods:
   - Configure your `/etc/hosts` file with the hostname and IP address in the output of the previous step.
   - Query a DNS server on your network.

3. Construct the URL using the data in the `HOSTS` column. You can then copy and paste this URL directly into your browser.

   ```
   https://hostname/ibm/console
   ```

   Where *hostname* is the name of the Agile Lifecycle Manager host that you want to log into. The hostname is made up of three elements:
   - Component name; for example: `alm`.
   - Release name; for example: `master21`.
   - Cluster name; for example: `mycluster.icp`.

   The resultant URL would be:

   ```
   https://alm.master21.mycluster.icp/ibm/console
   ```

   Note: Your hostname will differ from this example.

Managing assembly descriptors

You use the UI Editor to manage assemblies, for example editing assembly descriptors or creating new ones, importing or exporting them, or removing them from use.

Before you begin

To be able to edit assembly descriptors you should have opened the Agile Lifecycle Manager user interface and selected Editor-view from the top menu bar.

About this task

You use Editor to perform the following tasks:

Open an existing assembly descriptor
: You can open an existing assembly descriptor stored in the Agile Lifecycle Manager topology to view or modify it. Multiple descriptors can be open in parallel on separate browser windows.

Create a new assembly descriptor
: You can create a new assembly descriptor by writing a valid descriptor in the 'Editor' and saving it.

Change an assembly descriptor
: You can save an open assembly descriptor to the Agile Lifecycle Manager catalog after editing and replace the existing one in the Agile Lifecycle Manager catalog. Before saving a consistency check is performed. You will be prompted in case of invalid YAML format or when trying to change or save a resource type without sufficient permissions.

Save an assembly descriptor with a new name or version
: You can save an open assembly descriptor to the Agile Lifecycle Manager catalog with a new name or version number. Before saving a consistency check is performed. You will be prompted in case of already existing assembly name, invalid yaml-format.

Upload assembly descriptor from local file system
: You can upload an assembly descriptor from the local file system as a yaml-file. Before opening the uploaded yaml-file a consistency check is performed to verify the correctness of the yaml-format. Error messages will

be displayed if the uploaded file is not valid yaml. Wherever possible detailed information will be provided indicating where in the uploaded file the problem lies.

Download assembly descriptor into local file system
: You can download an open assembly descriptor to the local file system and select the target file location.

Duplicate an assembly descriptor
: You can duplicate an opened assembly descriptor. The version number of the duplicate assembly descriptor will be auto incremented while the name will stay the same.

Remove an assembly descriptor
: You are able to remove an existing assembly descriptor from the Agile Lifecycle Manager catalog

Procedure

Open an existing assembly descriptor

1. Find the assembly descriptor to open. All descriptors, both resources and assemblies existing in the Agile Lifecycle Manager catalog are listed on the left side of the Editor-view with associated action buttons.
2. Open the assembly descriptor.

   Once the right descriptor is located from the descriptor list it can be opened either to the same browser window or to a new tab. You can open a descriptor to the present window simply by clicking the corresponding section in the descriptor list.

   The selected item will be highlighted with white background and the descriptor yaml-content is presented on the right side of the descriptor list.

   You can open a descriptor to a new browser tab by clicking the action button associated with the descriptor.

Create a new assembly descriptor

3. Select to create a new assembly descriptor.

   You can create a new assembly descriptor by clicking the **New** button above the descriptor list.The Create Assembly Descriptor dialog box is opened. In the dialog box fill in the **Name**, **Version** and **Description** for the new assembly. Once these are defined click save to insert the new descriptor to the Agile Lifecycle Manager catalog. As the result a new descriptor with the given name is added to the descriptor list.
4. Edit the descriptor Open the new descriptor as described in Open an existing assembly descriptor. The new descriptor is now opened on the right side of the Editor-view and can be freely edited. By default the descriptor is created with template structure using the name, version and description given in the previous step and commented structure of a valid descriptor to ease the definition.
5. Save the new assembly descriptor

   When the descriptor is defined the newly edited descriptor can be saved by clicking the **Save** button located in the upper right corner of the text editor.

   When saving the descriptor you will be prompted about successful saving of the descriptor or an error condition in case the YAML-format is not correctly defined.

Change an assembly descriptor

6. Find the assembly descriptor to be changed. All descriptors are listed on the left side of the Editor-view.
7. Edit the assembly descriptor

   Make applicable changes to the descriptor by editing the YAML-descriptor opened in the text editor.
8. Save the changed assembly descriptor

   When the descriptor is defined the newly edited descriptor can be saved by clicking the **Save** button located in the upper right corner of the text editor.

   When saving the descriptor you will be prompted about successful saving of the descriptor or an error condition in case the YAML-format is not correctly defined.

Save an assembly descriptor with a new name or version

9. Open an assembly descriptor to be renamed or upgraded
10. Edit the assembly descriptor

    If you want to rename the assembly descriptor, edit the assembly name in the descriptor. (For example: `"name: assembly::name::1.0" -> "name: assembly::newname::1.0"`)

    If you want to upgrade the assembly to a higher version number, edit the version number of the assembly in the descriptor. (For example: `"name: assembly::name::1.0" -> "name: assembly::name::1.1"`)

    You are also able to change both the name and the version.
11. Save the renamed and/or upgraded assembly descriptor

    When the descriptor is defined the newly edited descriptor can be saved by clicking the **Save** button located in the upper right corner of the text editor.

    When saving the descriptor you will be prompted about successful saving of the descriptor or an error condition in case the YAML-format is not correctly defined.

    The newly saved version will replace the original in the Agile Lifecycle Manager catalog.

Upload assembly descriptor from local file system

An assembly descriptor is imported into the Agile Lifecycle Manager catalog by uploading a YAML file from the local file system. As part of the upload process certain consistency checks will be made and only if these pass will the new assembly descriptor appear in the Agile Lifecycle Manager catalog.

12. Start the assembly descriptor upload procedure

    You start the process of uploading an assembly descriptor from the local file system by clicking the **Upload** button above the descriptor list.
13. Select the assembly descriptor file

    An **Upload Assembly Descriptor** dialog box is opened. In the dialog box either drag and drop the file to the assigned target area or click the target area to open a file explorer to select the assembly descriptor file.

    The file being uploaded must have been saved as an YAML-file with the corresponding extension (.yml, .yaml).
14. Upload the assembly descriptor file

    After the file is selected click **Save** in the bottom of the dialog box to upload the assembly descriptor into the Agile Lifecycle Manager catalog.

    You will be informed of the status of the upload. When the upload is successful, a new descriptor item will appear in the descriptor list in the Editor. If the upload fails the consistency checks, error messages will be displayed indicating the source of the problem.

Download assembly descriptor to the local file system

15. Start the assembly descriptor download process

    Find and open the assembly descriptor you wish to download in the editor as described in Open an existing assembly descriptor.

16. Download the assembly descriptor

    When the descriptor is selected it can be downloaded by clicking the **Download** action button associated with the descriptor in the descriptor list.

    The downloaded assembly descriptor will be saved as a plain-text YAML file.

    A handle to access the downloaded descriptor is shown in the bottom left corner of the editor. The downloaded descriptor file can be accessed or opened in a local application by clicking the handle.

Duplicate an assembly descriptor

17. Select the assembly to be duplicated All descriptors, both resources and assemblies, existing in the Agile Lifecycle Manager catalog are listed on the left side of the Editor-view with associated action buttons.

18. Duplicate the selected assembly descriptor

    When the descriptor is selected it can be duplicated by clicking the **Duplicate** action button associated to the descriptor in the descriptor list.

Remove an assembly descriptor

19. Select the assembly to be removed. All descriptors, both resources and assemblies, existing in the Agile Lifecycle Manager catalog are listed on the left side of the Editor-view with associated action buttons.

20. Remove the assembly

    When the descriptor is selected and opened it can be removed from the Agile Lifecycle Manager catalog by clicking the **Remove** action button in the upper right corner of the text editor.

Operating assemblies

You use the UI Assemblies tool to create new assembly instances, or view existing ones and monitor any related activity. You can also use it to request lifecycle transitions or opinionated patterns on an assembly.

Before you begin

To be able to edit assembly descriptors you should have opened the Agile Lifecycle Manager user interface and selected Assemblies view from the top menu bar.

About this task

You use the Assembly view to perform the following tasks:

View existing assembly instances
: You can view the existing assembly instances managed by Agile Lifecycle Manager and the associated status.

Create a new assembly instance
: You can request a creation of a new instance of an assembly.

Request an assembly lifecycle transition
: You can request an intent on an existing assembly instance to transition to a new lifecycle state.

Request an opinionated pattern

You can request an opinionated pattern to be performed on an existing assembly instance. This include requests for Scale, Heal, Update, or Upgrade an assembly. Agile Lifecycle Manager will automatically resolve the required individual lifecycle transitions to achieve the requested target status and execute them subsequently.

Monitor process activity

You can monitor the process activity related to existing assembly instances including the progress of the lifecycle transition and opinionated pattern requests you have initiated.

Procedure

View an existing assembly instances

1. Browse the existing assembly instances Existing assembly instance can be seen on the starting page of the Assemblies view. Each assembly instance is represented by a card on the main page. Each card shows the Assembly name, Descriptor name, and the last action performed on the assembly. Action can be either a lifecycle state transition or a pattern like heal or scale.

Create a new assembly instance

2. Select to create a new assembly instance.

   A new assembly instance can be created by clicking the **Add** symbol on the top menu bar. Once the symbol is clicked a dialog is opened to fill in necessary details to perform the transaction.
3. Fill in assembly and transaction details. In the opened dialog you need to give following details:
   - Name for the new assembly instance. The defined name will be used as a unique, within Agile Lifecycle Manager, identifier of the assembly.
   - Descriptor, the descriptor is selected from a list of available descriptors that corresponds to the available descriptors existing in Agile Lifecycle Manager catalog.
   - The Target State for the lifecycle transition performed during the creation process. Available states include: Installed, Inactive, and Active. Agile Lifecycle Manager will resolve the necessary unitary transitions required to create and move the new assembly to the desired target state. For example if 'Active' is selected, Agile Lifecycle Manager will install the assembly and related components, configure, start, and perform integrity test on them.

   Click **Next** to move to the next phase of the definition process.
4. Enter the properties.

   In the opened dialog you need to give values to all properties defined in the selected descriptor. Some of the properties might have preset values defined in the descriptor. These values can be overridden by changing the corresponding value. All empty values must be given a value to create the assembly instance.

   Once the required details are filled in, click **Next** to move to the next phase of the definition process.
5. Verify and accept the changes.

   The final state of the definition presents you the assembly details and defined property values for verification.

   Once the information is verified to be correct click **Complete** to initiate the assembly creation process.

As a result the assembly is created and moved to the target state. New assembly instance can be seen as a new card on the starting page of the Assemblies view.

At any point of the process you can cancel the operation by clicking **Cancel**, or return to previous step by clicking **Previous**.

Request a lifecycle transition on an assembly

6. Select the target assembly instance

   You can select the target assembly by searching the assembly instance from the starting page of the Assemblies view and clicking the corresponding card.

   As the result a new page is opened showing details of the selected assembly. The view contains three parts:

   - Top bar presenting assembly status and lifecycle transition controls.
   - Process containment section showing action history of the related processes or assembly's component structure depending on the process containment selection.
   - Relationships section visualizing relations of the selected component.

7. Initiate a lifecycle transition on the selected assembly.

   Lifecycle transitions can be requested on the selected assembly instance by selecting a new target state for the assembly from the top bar. Only allowed states can be selected from the drop-down list.

   After a desired target state has been selected from the drop-down list click **Apply** next to the state selection to initiate the transition.

   All unitary actions performed and their status are visualized in the process section of the current page. Once the transition is completed the state of the assembly is changed and shown also on the starting page.

Request an opinionated pattern on an assembly

8. Select the target assembly instance You can select the target assembly by searching the assembly instance from the starting page of the Assemblies view and clicking the corresponding card.

9. Find the target component.

   You can browse the component structure of the selected assembly by selecting Containment view from the process containment section. Once Containment is selected the section presents the hierarchical component structure of the assembly. By default only the root level is shown. You can extend the view to show lower level components by clicking the component box in the view.

   Different patterns can be applied to different types of components. The component type is shown as a symbol next to each component. Also the applicable patterns depend on the type of the component.

   The target component can be selected from the hierarchy by laying over the mouse cursor on the corresponding item in the view.

10. Initiate an opinionated pattern on a component.

    Once the cursor is over a component that has applicable patterns available, a **spanner** symbol appears next to the component name. Available patterns can be seen by clicking the symbol.

    A new dialog box is opened giving options to cancel the operation of initiate any of the available patterns on the selected component. A pattern can be initiated by clicking the corresponding button in the dialog box.

    Running a pattern will result to a sequence of actions run on the assembly. The flow of the unitary operations run on different components can be viewed by selecting the Process view from the process containment section.

Monitor process activity

11. Select the target assembly instance You can select the target assembly by searching the assembly instance from the starting page of the Assemblies view and clicking the corresponding card.
12. Browse the process activity related to the selected assembly instance.

    You can browse the history of process activity related to the selected assembly by selecting the Process view from the process containment section. Once Process is selected the section presents the history of actions run on the components of the assembly.

    The process view is updated continuously according to preformed operations, for example when you initiate a lifecycle transition on the assembly or opinionated pattern on any of the related components.

Managing resource managers

You use the UI Resource Managers view to browse, refresh or remove existing resource managers, or add new ones.

Before you begin

To be able to edit assembly descriptors you should have opened the Agile Lifecycle Manager user interface and selected the Resource Manager view from the top menu bar.

About this task

You use Editor to perform the following tasks:

Add a new resource manager
: You can introduce a new resource manager to Agile Lifecycle Manager and onboard the resource types managed by the resource manager.

Browse existing resource managers
: You can browse the information related to existing resource managers.

Refresh a resource manager
: You can refresh a resource manager to Agile Lifecycle Manager and refresh the information about resource types managed by the resource manager.

Remove a resource manager
: You can remove an existing resource manager from Agile Lifecycle Manager. After removing a resource manager, Agile Lifecycle Manager is not able to manage any associated resource types.

Procedure

Add a new resource manager

1. Select to add a new resource manager.

   You can add a new resource manager by clicking the **Add new RM** button on top of the list showing the existing resource managers.
2. Fill in the resource manager details.

   To define the new resource manager you need to type in the name, type, and URL of the new resource manager.
   - **Name** is the unique resource manager name used as the identification of the resource manager.
   - **Type** is an Agile Lifecycle Manager internal attribute to categorize and separate different types of resource managers from each other.

- **URL** defines the actual location of the resource manager where it is deployed.

   Once the required information is filled in click the **Add new RM** button to initiate the onboarding process. As the result of successful onboarding a new resource manager is added to the list of shown resource managers.

Browse existing resource managers

3. View the list of existing resource managers.

   Existing resource managers and associated key attributes are shown on the front page when opening the Resource Managers view. Each resource manager record is associated with the following set of action buttons on the right side of the resource manager details section:
   - Refresh
   - Remove
   - View details
4. See the details of associated deployment locations.

   You can view the associated deployment locations where a resource manager is able to instantiate resources by clicking the **View details** action button associated with the resource manager.

Refresh a resource manager

5. You can refresh an existing resource manager and associated resource type descriptors by clicking the **Refresh** action button associated with the resource manager.

Remove a resource manager

6. You can remove an existing resource manager by clicking the **Remove** action button associated with the resource manager.

Upgrading an assembly instance

You can upgrade an instance to a new type, or change its property values, or both. This topic describes the assembly instance upgrade scenarios and limitations.

Before you begin

The assembly to be upgraded should be instantiated in the Agile Lifecycle Manager topology and must be in the 'active' state.

About this task

You upgrade an assembly instance by changing an active assembly instance from its current type and set of properties to a new type and/or new property values. The type of a component instance is determined by

```
[assembly|resource]::<type name>::<version>
```

The following assembly upgrade scenarios are supported.

- If the name of a property in the original and new type are the same, then they are assumed to be the same and can be mapped from the original to new properties.
- If a property value is changed in a component, then the component will be re-installed with the new value.
- If there is a new relationship between components of the new assembly type, the relationship is created. This may mean that a component must be transitioned to the correct states to create the new relationship.

- If a relationship between components is removed from the upgraded assembly, the relationship is deleted.
- If a property value of a relationship changes, then the relationship is deleted and re-created. This may mean that a component must be transitioned to the correct states to create the new relationship.
- If a component identified by name and type is not in the new assembly, it is uninstalled.
- If a component identified by name and type is not in the original assembly, it is created and transitioned to the active state.
- If an assembly's properties are changed, only the resources impacted are changed, resources that are not impacted remain unchanged. That is, if after an upgrade a resource has the same name, type and property values, then it will not be transitioned during the upgrade, but rather remain in the active state, unless a transition was triggered by a relationship change.
- If a component's descriptor changes in any way, it is expected that the type will have changed, that is, that there is a new type name and/or version, and the component will be re-installed.
- If a reference to an external component is removed from the assembly, then any relationships referring to it will be deleted.
- If a reference to an external component is added to the assembly, then any relationships referring to it will be created.
- The size of a cluster before an upgrade is maintained after the upgrade. [Question: what about the cluster sizing properties]
-

Assembly Upgrade limitations

Changing cluster property values to 'initial-quantity', 'minimum-nodes', 'maximum-nodes' or 'scaling-increment' is not supported.

Changes to policy and metric property values is not supported.

Chapter 5. Getting started (using the APIs)

Agile Lifecycle Manager provides both a graphical UI and an HTTP API allowing the creation and administration of assemblies. This section describes a set of basic scenarios to get started using the APIs.

Related concepts:

Chapter 4, "Using the UI," on page 35
This section describes the Agile Lifecycle Manager User Interface, and the tasks it allows you to perform.

"Lifecycle Manager API" on page 93
The Lifecycle Manager API is responsible for interactions with the operations available from Agile Lifecycle Manager. This section covers the definition of the Lifecycle Manager API and the specification of the messages sent across this interface.

"Assembly descriptor YAML specifications" on page 147
This section describes the assembly descriptors that are used by Agile Lifecycle Manager.

Related reference:

"Resource managers" on page 105
This topic describes the Resource Managers API specifications for the lifecycle management API. See the "Resource Manager API" on page 121 section for resource manager API specifications.

Configuration reference

This topic provides you with an overview of the Agile Lifecycle Manager services settings you need to know when configuring the solution for your own environment, such as port numbers, Swagger URLs, and API details.

API HTTP calls

Calls to the Agile Lifecycle Manager API are made using either REST or RPC mechanisms, and each call returns an HTTP status code. You can find more detailed information on the API HTTP status code strategy here: "API HTTP status codes reference" on page 89

Microservices ports and Swagger URLs

The following table shows the default ports and Swagger URLs for the Agile Lifecycle Manager microservices.

Table 6. Agile Lifecycle Manager microservices ports and Swagger URLs

Service	Port	Swagger URL	Notes
Daytona	8281	http://docker-host:8281/swagger-ui.html	Port **must not** be exposed through firewall

Table 6. Agile Lifecycle Manager microservices ports and Swagger URLs (continued)

Service	Port	Swagger URL	Notes
Ishtar	8280	http://docker-host:8280/swagger-ui.html	Port **must** be exposed through firewall **Note:** Ishtar Swagger also gives access to the public APIs of Galileo (from a drop-down list on the Swagger UI).
Nimrod	8290	http://docker-host:8290/	**Note:** The entry point if the UI is started. The port **must** be exposed through the firewall.

Service REST API

The Service REST API endpoints are available at the following URLs, and may be used as directed during a product support request:

Runtime metrics
: http://docker-host:port/management/metrics

Configuration properties
: http://docker-host:port/management/env

Kafka

To use Kafka outside the Docker containers, you set the environment variable **`KAFKA_ADVERTISED_HOST_NAME`** to the IP address of your Docker host, and then use that IP address when referencing Kafka in your scripts and software. The following Kafka topics are exposed by Agile Lifecycle Manager.

alm__processStateChange
: Process state change events

alm__stateChange
: State change events

alm__integrity
: Resource integrity metric messages aimed at Watchtower.

alm__load
: Resource integrity metric messages aimed at Watchtower.

alm__descriptorChange
: Indicates that a resource manager has updated its resource descriptors.

Note: There are two underscores (__) in the Kafka topics.

Runtime directories

At runtime, Agile Lifecycle Manager uses the following directories.

var_alm/config-repo
: Agile Lifecycle Manager Conductor configuration directory (Git repository)

var_alm/logs
: Log files for all the services

var_alm/cassandra
: Host-mounted Cassandra volume

Related reference:

"Asynchronous state change events" on page 108
Agile Lifecycle Manager will emit events when the state of an assembly or its components changes. Messages that are sent asynchronously are put onto a Kafka bus. The exact topics can be configured. These are emitted in response to Intent Requests causing the state of the Assembly Instance, or its associated components, to change. In the event of a failure to change state, an event will also be emitted.

Creating an assembly instance

You create a new assembly instance when you need to deploy a new service described in an assembly descriptor.

Before you begin

Agile Lifecycle Manager must be installed, with all included resources and test assemblies deployed to the catalog.

About this task

A new instance of an assembly is created by using the API for Daytona (Orchestrator) service. You can find more detailed information on the Daytona API, its methods, and associated attributes in the "Lifecycle Manager API" on page 93 reference section.

This task installs a new instance of a `t_bta` assembly called `test_1`, and then configures and starts it.

This example uses the basic test assembly `assembly::t_bta::1.0`. This assembly is composed of two resources named A and B, both of type `resource::t_simple::1.0`. It references three external resource instances, two networks of type `resource::openstack_neutron_network::1.0` and one image of type `resource::openstack_glance_image::1.0`. It has one relationship from A to B that is created when A and B are active. Resource B is in a cluster which on installation includes a single instance of B.

Procedure

1. Identify the assembly properties requiring a value when creating a new assembly instance. To do so, explore the corresponding assembly descriptor (in this example `assembly::t_bta::1.0`). Retrieve the descriptor from the Agile Lifecycle Manager catalog by running the following query on the Apollo API:

   ```
   GET /api/catalog/descriptors/assembly::t_bta::1.0
   ```

   The response to this query displays the descriptor. A sample extract is shown here. A full assembly descriptor sample can be viewed in the following topic: "Sample assembly descriptor" on page 71

   ```
   name: assembly::t_bta::1.0
   description: Assembly comprised of "components\\t_simple.yml"
   properties:
     data:
       default: "data"
   ```

```
    type: string
    description: 'parameter passed'
  output:
    description: an example output parameter
    type: string
    read-only: true
  deploymentLocation:
    type: string
    description: name of openstack project to deploy network
    default: admin@local
...
```

The purpose of the 'properties' section in the API request is to give values to required assembly properties. The 'properties' section in the API request must set the value of any properties from the 'properties' section of the assembly descriptor that don't have a default value. You can override any default values. In the following example steps the default value of 'deploymentLocation' is changed.

2. Initiate a **createAssembly** event from the Daytona API. Use the swagger-url for the Daytona service to create a new assembly instance using the following POST command:

```
POST /api/intent/createAssembly
{
  "assemblyName": "test_1",
  "descriptorName": "assembly::t_bta::1.0",
  "intendedState": "Inactive",
 properties":{
"deploymentLocation":"admin@local"}
}
```

In this case, the `test_1` assembly instance (`assemblyName`) does not exist, and so Agile Lifecycle Manager will attempt to install, configure and then start this new instance.

Note: Use the 'properties' section to define any assembly properties that are as yet undefined, or to override any already defined default values. In this example a value of `admin@local` is set for the deploymentLocation property.

3. If `test_1` has been successfully created, Agile Lifecycle Manager will return a `Response Code 201`, as in this example:

```
Response code 201

{
  "location": "http://10.220.217.161:8280/api/processes/
5a65ce87-4637-401e-868b-e20ec254fd35",
  "date": "Mon, 11 Sep 2017 11:54:20 GMT",
  "server": "ALM Ishtar/1.1.0-SNAPSHOT",
  "transfer-encoding": "chunked",
  "x-application-context": "ishtar:prod,swagger:8280",
  "content-type": null
}
```

The process identifier in this response is `5a65ce87-4637-401e-868b-e20ec254fd35`. As soon as the new assembly instance is created, it can be referred to by name. that is, `test_1`.

4. To check progress of this request, copy the process identifier from the response into the **`eventId`** parameter of the following GET command (`GET /api/topology/assemblies{id}`):

```
GET /api/topology/assemblies/5a65ce87-4637-401e-868b-e20ec254fd35
```

The `processState` in the following sample response indicates that the request has completed, while `intendedState` indicates that it is as yet inactive.

```
{
  "processId": "5a65ce87-4637-401e-868b-e20ec254fd35",
  "assemblyId": " ef4ea879-e313-4e3b-ad11-6a0a7e7544eb",
  "assemblyName": "test_1",
  "assemblyDescriptorName": "assembly::t_bta::1.0",
  "intentType": "CreateAssembly",
  "intent": {
    "assemblyName": "test_1",
    "descriptorName": "assembly::t_bta::1.0",
    "intendedState": "Inactive"
  },
  "processState": "Completed",
  "processStartedAt": "2017-09-14T07:34:55.569Z",
  "processFinishedAt": "2017-09-14T07:34:58.806Z"
}
```

Note: If the request had not completed it would have been in the `InProgress` state. If there had been an issue, it would have been in the `Failed` state, in which case there would have been a `requestStateReason` property with text describing the failure.

5. Initiate a **Start** event from the Daytona API to start the new assembly instance, that is, move it from an inactive to an active state. To start the new assembly instance, use the swagger-url for the Daytona service, using the following POST command:

```
POST /api/intent/changeAssemblyState
{
  "assemblyName": "test_1",
  "intendedState": "Active"
}
```

   Agile Lifecycle Manager executes Configure and then Start transitions, which move the state to 'active'.

6. To verify the status of the assembly instance, check it again:

```
GET /api/topology/assemblies/5a65ce87-4637-401e-868b-e20ec254fd35
```

   The 'intendedState' in the sample response indicates that the Start event has completed successfully.

```
{
  "processId": "5a65ce87-4637-401e-868b-e20ec254fd35",
  "assemblyId": " ef4ea879-e313-4e3b-ad11-6a0a7e7544eb",
  "assemblyName": "test_1",
  "assemblyDescriptorName": "assembly::t_bta::1.0",
  "intentType": "CreateAssembly",
  "intent": {
    "assemblyName": "test_1",
    "descriptorName": "assembly::t_bta::1.0",
    "intendedState": "Active",
...
```

Results

A new assembly instance with the name 'test_1' exists in an 'Active' state, and configured according to the rules defined in assembly descriptor 'assembly::t_bta::1.0'.

Exploring an assembly instance

This topic describes how to use Agile Lifecycle Manager to see the assembly structure, and (optionally) the associated history of lifecycle transitions of a previously created assembly, in this case test_1.

Before you begin

You must create the example assembly instance called `test_1`, which is described in "Creating an assembly instance" on page 49.

About this task

To see the assembly structure and the history of lifecycle transitions, you explore the topology of the assembly instance by using the API for Galileo (the gateway service). The Galileo API, its methods and associated attributes are explained in more detail in the "Lifecycle Manager API" on page 93 reference section.

Procedure

1. Identify the name of the assembly instance from its `assemblyName` field, in this case `test_1`.
2. Query the assembly topology through the Galileo API using the following GET command:

   ```
   GET /api/topology/assemblies?numEvents=<event numbers>&name=<assembly instance name>
   ```

   To see only the structure of the assembly, set the value of `numEvents` to zero (`0`). If you set it to a value greater than zero, the specified number of Assembly Lifecycle Request events that have occurred on the assembly and their associated State Change Events is displayed, starting with the latest.

   The following example depicts the GET request to obtain the topology of test_1, with no event history.

   ```
   GET /api/topology/assemblies?numEvents=0&name=test_1
   ```

   This query results in the following response:

   ```
   {
     "type": "Assembly",
     "id": "ef4ea879-e313-4e3b-ad11-6a0a7e7544eb",
     "name": "test_1",
     "state": "Active",
     "descriptorName": "assembly::t_bta::1.0",
     "properties": [
       {
         "name": "numOfServers",
         "value": "1"
       },
       {
         "name": "data",
         "value": "data"
       },
       {
         "name": "deploymentLocation",
         "value": "admin@local"
       },
       {
         "name": "resourceManager",
         "value": "test-rm"
       }
     ],
     "createdAt": "2017-09-12T06:38:43.365+0000",
     "lastModifiedAt": "2017-09-12T06:38:45.514+0000",
   ```

```
  "children": [
    {
      "type": "Component",
      "id": "c903c6db-9a79-4248-b62b-fd11ec01efe0",
      "name": "test_1__A",
      "externalId": "72857624-ccb4-4cf2-8ebf-0ab05e67a5a5",
      "state": "Active",
      "descriptorName": "resource::t_simple::1.0",
      "properties": [
        {
          "name": "data",
          "value": "data"
...
  ]
}
```

This sample response displays the identity, status and structure of the assembly instance. The assembly instance has two child instances, `test_1__A` and `test_1__B__1`. The names are generated from the assembly name `test_1` and the resource name from the `t_bta` descriptor. Resource `B` is in a scaling group, and each member of a scaling group is numbered. The relationship between the two resources can also be seen, as well as the references to resources used from the `deploymentLocation` property.

Healing a component

A heal request targets a component (a resource that is 'broken') and attempts to return it to an 'Active' state.

Before you begin

- You must create the example assembly instance with an assemblyName of `test_1`, as described in "Creating an assembly instance" on page 49.
- To heal the component of an assembly, the assembly to which the component belongs must be in an Active state.

About this task

- Agile Lifecycle Manager accepts the request to heal without performing any checks first.
- Heal is a pattern that calls 'Stop', 'Start', and then 'Integrity' on the component.
- If 'Integrity' is successful, then the heal is successful.
- The assembly containing the broken component must be in the 'active' state to call heal.
- The request to heal includes the ID of the component in the assembly instance to be healed.

Procedure

1. To obtain the ID of the component in the assembly instance to be healed, you can query the assembly topology through the Galileo API using the following GET command:

   Note: You identify the component to be healed by the following combination: The name or ID of the assembly, **plus** the name or ID of the component.

   ```
   GET /api/topology/assemblies?numEvents=0&name=test_1
   ```

   This query will return assembly topology information including the `id` in the `children` section, as depicted in the following sample extract:

```
...
  "children": [
    {
      "type": "Component",
      "id": "c903c6db-9a79-4248-b62b-fd11ec01efe0",
      "name": "test_1__A",
      ...
```

2. To initiate a heal pattern from the Daytona API, run the following POST command from the swagger-url of the Daytona service. You can use the name or ID as identifier. In the following examples, the ID or name obtained in the previous step can be used to define the `componentId` value, which targets the component to be healed. Any of the following examples will initiate a heal pattern.

   Assembly name and component name

   ```
   POST /api/intent/healAssembly
   {
     "assemblyName": "test_1",
     "brokenComponentName": " test_1__A"
   }
   ```

   Assembly name and component ID

   ```
   POST /api/intent/healAssembly
   {
     "assemblyName": "test_1",
     "brokenComponentId": "c903c6db-9a79-4248-b62b-fd11ec01efe0"
   }
   ```

   Assembly ID and component name

   ```
   POST /api/intent/healAssembly
   {
     "assemblyId": "ef4ea879-e313-4e3b-ad11-6a0a7e7544eb",
     "brokenComponentName": "test_1__A"
   }
   ```

   Assembly ID and component ID

   ```
   POST /api/intent/healAssembly
   {
     "assemblyId": "ef4ea879-e313-4e3b-ad11-6a0a7e7544eb",
     "brokenComponentId": "c903c6db-9a79-4248-b62b-fd11ec01efe0"
   }
   ```

   Agile Lifecycle Manager initiates the heal pattern, which cycles through 'Stop', 'Start', and 'Integrity'.

3. To check the status of the previously 'broken' component after healing, you query the assembly topology through the Galileo API using the following GET command, with numEvents set to 5 in order to see the sequence of heal events that occurred.

   ```
   GET /api/topology/assemblies?numEvents=5&name=test_1
   ```

Results

If the healing has been successful, the depicted state transitions will move from 'Broken' to 'Inactive' to 'Active'. A state of 'Active' connotes a healthy, in this case healed, component, as depicted in the following example.

Example

```
{
  "eventId": "d557dfcb-609b-40d3-9f87-c9cc8568ccc1",
  "rootAssemblyInstanceId": "acbd4cb1-1ec2-41f6-a2c7-693653291e5a",
  "rootAssemblyInstanceName": "test_1",
  "resourceInstanceId": "a4e6a96a-db96-4ef6-ac59-4e87c7efb5d2",
```

```
  "resourceInstanceName": "test_1__A",
  "resourceManager": "test-rm",
  "deploymentLocation": "admin@local",
  "externalId": "6d427f14-34a5-48f9-9575-2ec0b9ede2df",
  "eventCreatedAt": "2017-08-23T11:16:16.637Z",
  "previousState": "Broken",
  "newState": "Inactive",
  "successful": true,
  "changeStartedAt": "2017-08-23T11:16:16.546Z",
  "changeFinishedAt": "2017-08-23T11:16:16.637Z",
  "eventType": "StateChangeEvent"
}

{
  "eventId": "1786a172-7027-4223-943e-edc5673ad5bc",
  "rootAssemblyInstanceId": "acbd4cb1-1ec2-41f6-a2c7-693653291e5a",
  "rootAssemblyInstanceName": "test_1",
  "resourceInstanceId": "a4e6a96a-db96-4ef6-ac59-4e87c7efb5d2",
  "resourceInstanceName": "test_1__A",
  "resourceManager": "test-rm",
  "deploymentLocation": "admin@local",
  "externalId": "6d427f14-34a5-48f9-9575-2ec0b9ede2df",
  "eventCreatedAt": "2017-08-23T11:16:16.867Z",
  "previousState": "Inactive",
  "newState": "Active",
  "successful": true,
  "changeStartedAt": "2017-08-23T11:16:16.678Z",
  "changeFinishedAt": "2017-08-23T11:16:16.867Z",
  "eventType": "StateChangeEvent"
}

{
  "eventId": "70061a12-d42f-42c4-8c70-8ca9391ce37f",
  "eventCreatedAt": "2017-08-23T11:16:16.327Z",
  "assemblyInstanceId": "acbd4cb1-1ec2-41f6-a2c7-693653291e5a",
  "assemblyInstanceName": "test_1",
  "assemblyDescriptorName": "assembly::t_bta::1.0",
  "action": "Heal",
  "requestState": "Completed",
  "requestStartedAt": "2017-08-23T11:16:16.327Z",
  "requestFinishedAt": "2017-08-23T11:16:17.066Z",
  "properties": {
    "data": "data",
    "deploymentLocation": "admin@local",
    "numOfServers": "1",
    "resourceManager": "test-rm"
  },
  "eventType": "AssemblyLifecycleRequest"
}
```

Scaling a component

Scaling the component of an assembly is a pattern that will either add or remove a component instance from a scaling group (including all relationships).

Before you begin

- You must create the example assembly instance called `test_1`, which is described in “Creating an assembly instance” on page 49.
- The assembly to which the component belongs must be instantiated in the Agile Lifecycle Manager topology, and be in the 'Active state'.
- A cluster definition for the component itself must exist in the assembly descriptor, in which the minimum and maximum size of the scaling group and the default increment when scaling out or in are also defined.

You can find more detailed information on how to define clusters in assembly descriptors in the “Assembly descriptor YAML specifications” on page 147 reference section.

About this task

Assembly descriptors are in the Agile Lifecycle Manager catalog, which you can view using the process described in following topic: “Exploring an assembly descriptor” on page 60 For this task you use component `B`, which is part of the `t_bta` assembly (`assembly::t_bta::1.0`).

The cluster definition for component B is depicted in the following sample:

```
...
composition:
  A:
    type: resource::t_simple::1.0
    quantity: '1'
...
  B:
    type: resource::t_simple::1.0
    cluster:
      initial-quantity: '${numOfServers}'
      minimum-nodes: 1
      maximum-nodes: 4
      scaling-increment: 1
    properties:
 ...
```

Procedure

1. To identify the name of the component of the `test_1` assembly to be scaled, you can query the assembly topology through the Galileo API using the following GET command:

   ```
   GET /api/topology/assemblies?numEvents=0&name=test_1
   ```

   This query will return assembly topology information including the descriptorName, as depicted in the following sample:

   ```
   "type": "Assembly"
     "id": "bf649336-c8c5-49d9-9f4e-60567fe54135",
     "name": "test_1",
     "state": "Active",
     "descriptorName": "assembly::t_bta::1.0",
     "properties": [
       {
   ...
   ```

2. Retrieve the descriptor from the Agile Lifecycle Manager catalog by running the following query on the Apollo API using the descriptorName obtained in the previous step:

   ```
   GET /api/catalog/descriptors/assembly::t_bta::1.0
   ```

   The scaling group definition for B is shown in the following sample.

   ```
   ...
   composition:
     A:
       type: resource::t_simple::1.0
       quantity: '1'
   ...
     B:
       type: resource::t_simple::1.0
       cluster:
         initial-quantity: '${numOfServers}'
   ```

```
      minimum-nodes: 1
      maximum-nodes: 4
      scaling-increment: 1
    properties:
 ...
```

Here one instance of B is created when an instance of the assembly is created. By scaling Out or In, the amount of B instances can be changed between one and four. As the increment is defined as one, each scale out or scale in pattern will increase or decrease the amount of B instances by one.

3. To scaleOut, that is to add another instance of resource B to test_1, use the swagger-url for the Daytona service, using the following POST command: You can use either the assemblyName or assemblyId to initiate the 'Scale Out' pattern.

   ```
   POST /api/intent/scaleOutAssembly
   {
     "assemblyName": "test_1",
     "clusterName": "B"
   }
   ```

   Or:

   ```
   POST /api/intent/scaleOutAssembly
   {
     "assemblyId": "ef4ea879-e313-4e3b-ad11-6a0a7e7544eb",
     "clusterName": "B"
   }
   ```

   The clusterName identifies the group of resources to be scaled (B), increasing the amount of B instances by one unless the maximum scaling group size (in this case four) has been reached.

4. To scaleIn, that is to reduce the instances of resource B, use the following POST command: You can use either the assemblyName or assemblyId to initiate the 'Scale In' pattern.

   ```
   POST /api/intent/scaleInAssembly
   {
     "assemblyName": "test_1",
     "clusterName": "B"
   }
   ```

   Or:

   ```
   POST /api/intent/scaleInAssembly
   {
     "assemblyId": "ef4ea879-e313-4e3b-ad11-6a0a7e7544eb",
     "clusterName": "B"
   }
   ```

   Here the group of B resources are decreased by one, unless the minimum scaling group size (in this case one) has been reached.

5. To check the status of the assembly after scaling, query the assembly topology using the following GET command, with numEvents set to 5 in order to see the sequence of scaling events.

   ```
   GET /api/topology/assemblies?numEvents=5&name=test_1
   ```

   The assembly topology will depict all the instances of component B and the relationships towards the new resource instances.

Results

When viewing the topology after running ScaleOut, a new instance of B will be depicted called `test_1__B__2`, as well as a new relationship between it and the existing `test_1__A` instance.

Uninstalling an assembly instance

To uninstall an existing assembly instance from the Agile Lifecycle Manager topology, and the corresponding resources from the applicable resource managers, you use the swagger-url for the Daytona service.

Before you begin

Before you can uninstall an assembly instance, you must identify it.

For this example uninstall scenario, you must first create the example assembly instance called `test_1`, which is described in "Creating an assembly instance" on page 49.

About this task

You can find more detailed information on the Daytona API, its methods, and associated attributes in the "Lifecycle Manager API" on page 93 reference section.

Procedure

1. Identify the name of the assembly instance from its `assemblyName` field, in this case `test_1`.
2. To uninstall the assembly instance, run the following POST command from the swagger-url of the Daytona service: You can use either the assemblyName or assemblyId to uninstall an assembly instance.

   ```
   POST /api/intent/deleteAssembly
   {
   "assemblyName":"test_1"
   }
   ```

   Or:

   ```
   POST /api/intent/deleteAssembly
   {
   "assemblyId":"ef4ea879-e313-4e3b-ad11-6a0a7e7544eb"
   }
   ```

   A successful uninstall will result in a system response similar to the following example:

   ```
   {  "x-application-context": "ishtar:prod,swagger:8280",  "date": "Mon, 27 Nov 2017 15:54:03 GMT",
   "location": "http://9.20.65.179:8280/api/swagger/daytona/api/processes/86524eed-532e-47ad-aa8a-
   1d8e9ab0aae3",
   "transfer-encoding": "chunked",  "server": "ALM Ishtar/1.1.2.181",  "content-type": null}
   ```

Results

The uninstall process results in the removal of the test_1 assembly instance, as well as any corresponding resources from the applicable resource managers.

What to do next

You can double-check that the assembly instance has been successfully uninstalled by querying the assembly topology by name, using the following GET command:

```
GET /api/topology/assemblies?numEvents=0&name=test_1
```

If the uninstall process was successful, a response code of `404 (NOT FOUND)` will be returned, indicating that the assembly instance has been removed from Agile Lifecycle Manager.

Browsing assembly descriptors

This task allows you to browse all descriptors existing in the Agile Lifecycle Manager catalog. The descriptors include both the onboarded resource descriptors and assembly descriptors created in Agile Lifecycle Manager.

Before you begin

Agile Lifecycle Manager must be installed, with any included resources and test assemblies deployed to the catalog.

Remember: Resources must exist (for example, must have been onboarded) before you can browse descriptors.

Procedure

Query the existing descriptors in the Agile Lifecycle Manager catalog. The list of resource and assembly descriptors in the Agile Lifecycle Manager catalog can be viewed by running the following query on the Apollo API:

```
GET /api/catalog/descriptors
```

The response to this query lists the names, descriptions and references to all existing descriptors in the catalog, as depicted in the following sample response:

```
[
  {
    "name": "resource::t_simple::1.0",
    "description": "resource for  t_simple",
    "links": [
      {
        "rel": "self",
        "href": "http://10.220.217.175:8280/api/catalog/descriptors/resource::t_simple::1.0"
      }
    ]
  },
...

...
  {
    "name": "assembly::t_bta::1.0",
    "description": "Basic Test Assembly",
    "links": [
      {
        "rel": "self",
        "href": "http://10.220.217.175:8280/api/catalog/descriptors/assembly::t_bta::1.0"
      }
    ]
  }
]
```

Results

You now have reference information about the resource and assembly descriptors that are in the Agile Lifecycle Manager catalog.

Exploring an assembly descriptor

This task allows you to investigate the full contents of a specific resource or assembly descriptor in the Agile Lifecycle Manager catalog.

Before you begin

To explore an assembly descriptor, it must have been created in Agile Lifecycle Manager, or created in the Agile Lifecycle Manager catalog during the onboarding of a resource descriptor.

Procedure

Retrieve the descriptor from the Agile Lifecycle Manager catalog by running the following query on the Apollo API: The assembly descriptor explored in this example is `assembly::example::1.0`

```
GET /api/catalog/descriptors/assembly::t_bta::1.0
```

The response to this query displays the descriptor. A sample extract is shown here. The full assembly descriptor of this assembly can be viewed in the following topic: "Sample assembly descriptor" on page 71

```
name: assembly::t_bta::1.0
description: Basic Test Assembly
properties:
  data:
    default: "data"
    type: string
    description: 'parameter passed'
...
composition:
  A:
    type: resource::t_simple::1.0
...

  B:
    type: resource::t_simple::1.0
...
references:
  internal-network:
...
relationships:
  third-relationship:
    source-capabilities:
    - A.capability-3
    target-requirements:
    - B.requirement-3
...
```

Results

This task allows you to view a specific resource or assembly descriptor in the Agile Lifecycle Manager catalog.

Creating a new assembly descriptor

This task creates a new assembly descriptor and inserts it in the Agile Lifecycle Manager topology.

Before you begin

Agile Lifecycle Manager must be installed, with all included resources and test assemblies deployed to the catalog.

About this task

An example assembly named `assembly::example::1.0` is used in this task.

Procedure

1. Insert a new assembly descriptor into the Agile Lifecycle Manager catalog by running the following request on the Apollo API. The content of the assembly descriptor must be in YAML format. The full assembly descriptor of this assembly is in the following topic: "Sample assembly descriptor" on page 71

   ```
   POST /api/catalog/descriptors

   {
   DESCRIPTOR OF THE NEW ASSEMBLY IN YAML FORMAT
   }
   ```
2. Verify that the new descriptor exists in the Agile Lifecycle Manager catalog. The list of resource and assembly descriptors in the Agile Lifecycle Manager catalog can be viewed by running the following query on the Apollo API:

   ```
   GET /api/catalog/descriptors
   ```

   The response to this query lists the new descriptor.

Results

After completing this task, the newly created assembly descriptor exists in the Agile Lifecycle Manager topology and can be instantiated and managed by Agile Lifecycle Manager.

Updating an assembly descriptor

This task updates an existing assembly descriptor in the Agile Lifecycle Manager catalog.

Before you begin

An assembly descriptor must exist in the Agile Lifecycle Manager catalog.

About this task

This task changes an existing descriptor in the Agile Lifecycle Manager catalog without changing the version number of the descriptor.

Tip: This procedure is mainly intended for service designers who want to change a descriptor during the development process.

Procedure

1. Change the assembly descriptor as required.
2. Replace the previous assembly descriptor version in the Agile Lifecycle Manager catalog by running the following request on the Apollo API. The assembly descriptor replaced is this example is `assembly::example::1.0`. The full assembly descriptor of this assembly is in the following topic: "Sample assembly descriptor" on page 71 The content of the assembly descriptor must be in YAML format.

   ```
   PUT /api/catalog/descriptors/assembly::example::1.0

   {
   CHANGED DESCRIPTOR OF AN EXISTING ASSEMBLY IN YAML FORMAT
   }
   ```
3. Verify that the descriptor updates have been made. Retrieve the descriptor from the Agile Lifecycle Manager catalog by running the following query on the Apollo API:

   ```
   GET /api/catalog/descriptors/assembly::example::1.0
   ```

   The response to this query displays the updated descriptor.

Results

After completing this task the `assembly::example::1.0` descriptor in the Agile Lifecycle Manager catalog is updated according to the YAML given as input in step 2.

Removing an assembly descriptor

This task removes an existing assembly descriptor from the Agile Lifecycle Manager catalog.

Before you begin

An assembly descriptor must exist in the Agile Lifecycle Manager catalog before you can remove it.

About this task

After removing a descriptor from the Agile Lifecycle Manager catalog it is not possible to create new instances of it. All existing instances of the assembly will remain and are not deleted. However, it is recommended that you not remove an assembly descriptor while there are existing instances of it in the Agile Lifecycle Manager topology.

Tip: Assembly instances can be deleted by running the "Uninstalling an assembly instance" on page 58 task.

Procedure

1. Remove an assembly descriptor from the Agile Lifecycle Manager catalog by running the following request on the Apollo API. The assembly descriptor deleted in this example is `assembly::example::1.0`

   ```
   DELETE /api/catalog/descriptors/assembly::example::1.0
   ```
2. Verify that the descriptor has been deleted. Attempt to retrieve the descriptor from the Agile Lifecycle Manager catalog by running the following query on the Apollo API:

```
GET /api/catalog/descriptors
```

The response to this query should not list the descriptor anymore.

Results

After completing this task the descriptor has been removed from the Agile Lifecycle Manager catalog and can no longed be viewed or managed by Agile Lifecycle Manager.

Upgrading an assembly instance

Once an assembly has been created it can be upgraded. An assembly instance can be upgraded to a new type, or have changed property values, or both new type and property values. This topic describes how to upgrade an assembly instance, with examples of how to change types and add a component.

Before you begin

The assembly to be upgraded should be instantiated in the ALM topology and must be in the 'active' state.

For the following example, you must have created the example assembly instance called `test_1`, as described in “Creating an assembly instance” on page 49.

About this task

This task upgrades a property of the 'test_1' assembly instance of the type 't_single::1.0' to the type 't_single::1.1', which also has an additional component named 'B'.

Supported assembly upgrade scenarios

Definition of an assembly upgrade

Change an active assembly instance from its current type and set of properties to a new type and/or new property values.

The 'type' of a component instance is determined by `[assembly|resource]::<type name>::<version>`

If the name of a property in the original and new type are the same, then they are assumed to be the same and can be mapped from the original to new properties.

If a property value is changed in a component, then the component will be re-installed with the new value.

If there is a new relationship between components of the new assembly type, the relationship is created. This may mean that a component must be transitioned to the correct states to create the new relationship.

If a relationship between components is removed from the upgraded assembly, the relationship is deleted.

If a property value of a relationship changes, then the relationship is deleted and re-created. This may mean that a component must be transitioned to the correct states to create the new relationship.

If a component, identified by name and type is not in the new assembly it is uninstalled.

If a component, identified by name and type is not in the original assembly it is created and transitioned to the active state.

If an assembly’s properties are changed, only the resources impacted are changed, resources that are not impacted and are unchanged. That is, if after an upgrade, a resource has the same name, type and property values, then it will not be transitioned during the upgrade; it will remain in the active state, unless a transition was triggered by a relationship change.

If a component's descriptor changes in any way, it is expected that the type will have changed (that is, there is a new type name and or version) and the component will be re-installed.

If a reference to an external component is removed from the assembly then any relationships referring to it will be deleted.

If a reference to an external component is added to the assembly then any relationships referring to it will be created.

The size of a cluster before an upgrade is maintained after the upgrade.

The current limitations on an Assembly Upgrade are:

- Changing cluster property values to 'initial-quantity', 'minimum-nodes', 'maximum-nodes' or 'scaling-increment' is not currently supported by the Assembly Upgrade pattern.
- Changes to policy and metric property values is not supported by the Assembly Upgrade pattern.

Procedure

1. Obtain a copy of `t_simple::1.0` by browsing the catalog. Run the following query on the Apollo API to find it:
   ```
   GET /api/catalog/descriptors
   ```
2. Change the version to `t_simple::1.1`
3. Add a new resource 'B' to the composition table by copying resource A and changing it as depicted in the following example:
   ```
   B:
       type: resource::t_simple::1.1
       quantity: '1'
       properties:
         referenced-internal-network:
           value: ${internal-network.id}
         reference-public-network:
           value: ${public-network.id}
         image:
           value: ${xenial-image.id}
         key_name:
           value: "ACCANTO_TEST_KEY"
         data:
           value: ${data}
         output:
           value: "B_output"
         deploymentLocation:
           value: ${deploymentLocation}
         resourceManager:
           value: ${resourceManager}
   ```
4. Create a new assembly descriptor in the Agile Lifecycle Manager catalog called `assembly::t_single::1.1` by running the following request on the Apollo API.
   ```
   POST /api/catalog/descriptors/assembly::t_single::1.1
   ```
5. Query the assembly topology for the assembly instance 'test_1' by using the following API request on the Ishtar service.

```
GET /api/topology/assemblies?numEvents=0&name=test_1
```

The assembly type for `test_1` will be `assembly::t_single::1.0`, and there will not be a component of type 'B'.

6. Upgrade of assembly instance.

```
POST /api/intent/upgradeAssembly
{
  "assemblyName": "test_1",
  "descriptorName": "assembly::t_single::1.1"
}
```

7. Verify the updated assembly instance 'test_1' by using the following API request on the Ishtar service.

```
GET /api/topology/assemblies?numEvents=0&name=test_1
```

The assembly type for `test_1` will be `assembly::t_single::1.1`, and there will be a new component of type 'B'.

List all onboarded resource managers

Use this task to see all resource managers that have been onboarded to Agile Lifecycle Manager.

Before you begin

Agile Lifecycle Manager must be installed, with all included resources and test assemblies deployed to the catalog.

One or more resource managers must have been onboarded.

Remember:

Onboarding is the act of adding a resource manager to Agile Lifecycle Manager. It lets Agile Lifecycle Manager know that the resource manager exists, and it imports the descriptors of all the resource types managed by the resource manager. It also gathers the information about the deployment locations that the resource manager uses.

About this task

Agile Lifecycle Manager maintains a record of each resource manager it can use to create and manage resources. When a new resource manager record is created, the resource types managed by that resource manager instance are read into Agile Lifecycle Manager via the resource manager's API.

Procedure

Query the list of onboarded resource managers by running the following query on the Ishtar API:

```
GET /api/resource-managers
```

The response to the query returns the name, type and url of each resource manager present.

What to do next

You now know which resource managers have been onboarded, and can use their identifiers to access them.

Exploring an onboarded resource manager

You can explore the status of any onboarded resource managers using the API for Ishtar (Gateway).

Before you begin

To confirm a resource manager has been onboarded, you need to know the name that was used to create it in Agile Lifecycle Manager, in this case 'test-rm'.

Remember:

Onboarding is the act of adding a resource manager to Agile Lifecycle Manager. It lets Agile Lifecycle Manager know that the resource manager exists, and it imports the descriptors of all the resource types managed by the resource manager. It also gathers the information about the deployment locations that the resource manager uses.

About this task

Agile Lifecycle Manager uses resource managers to deploy new resource instances and execute transitions and operations on them. A resource manager manages instances of resource types in a number of resource locations where resource instances can be created.

Agile Lifecycle Manager maintains a record of each resource manager it can use to create and manage resources. When a new resource manager record is created, the resource types managed by that resource manager instance are read into Agile Lifecycle Manager via the resource manager's API.

In Agile Lifecycle Manager, the resource types are stored in `var_alm/catalog/resources` as descriptor files (see the “Assembly descriptor YAML specifications” on page 147 topic for more details). The deployment locations that a resource manager instance can deploy resources into are also onboarded. Each deployment location is considered local to a resource manager.

You can find more detailed information on the Ishtar API, its methods, and associated attributes in the “Lifecycle Manager API” on page 93 reference section.

Procedure

1. Ensure you have the correct resource manager name. The name of the resource manager is defined when the resource manager record is created, as described in “Creating a new resource manager record” on page 67.
2. Use the following GET command on the Ishstar API to confirm that a resource manager has been onboarded:

   ```
   GET /api/resource-managers/<your-rm>
   ```

   If the resource manager exists, the following response will be displayed.

```
{
  "name": "<your-rm>",
  "type": "default",
  "url": "http://<your-rm>:8295/api/resource-manager"
}
```

As <*your-rm*> is the name of a resource manager instance, the response body comprises the instance name and the type of the resource manager (in this case `default`). The type of the resource manager is local to Agile Lifecycle Manager to allow the user to describe resource manager instances that are of the same type.

The `url` property contains the URL of the resource manager that is used by Agile Lifecycle Manager to make requests to the resource manager instance. The `hostname:port` (<*your-rm*>:8295) must be reachable from the Agile Lifecycle Manager host. If using the Swagger API, the `{id}` parameter in the URL is set to the 'name' value (<*your-rm*>).

Tip: If the resource manager has not been onboarded, then the response code to the GET will be `404 – NOT FOUND`

What to do next

To perform another check of successful onboarding, you can look in `var_alm/catalog/resources` and check that any resource descriptors that the new resource manager supports have been onboarded. To view descriptors, follow the steps described in "Browsing assembly descriptors" on page 59.

Creating a new resource manager record

You create a new resource manager record in Agile Lifecycle Manager in order to make the system aware of the resource manager, and enable it to access the resources it manages.

Before you begin

This assumes a resource manager has been installed, configured and is ready to be used by Agile Lifecycle Manager at a given URL, which in this example is `http://<rm-ip-address>:<rm-api-port>/api/resource-manager`

Remember: A resource manager record, which is created in this task, is not the same as an actual resource manager.

Agile Lifecycle Manager maintains a record of each resource manager it can use to create and manage resources. When a new resource manager record is created, the resource types managed by that resource manager instance are read into Agile Lifecycle Manager via the resource manager's API.

About this task

To create a new record of a resource manager in order to make Agile Lifecycle Manager aware of its existence, you must give it a unique name, in this example 'ucd'.

Tip: 'Type' is a name local to Agile Lifecycle Manager and can be used to describe the type of resource manager.

You can find more detailed information on the Ishtar API, its methods, and associated attributes in the "Lifecycle Manager API" on page 93 reference section.

Procedure

1. To create a new resource manager record, use the following POST command on the Ishstar API:

   ```
   POST /api/resource-managers
   {
     "name": "ucd",
     "type": "ucd",
     "url": "http://<rm-ip-address>:<rm-api-port>/api/resource-manager"
   }
   ```

   If the response code to the POST is `201 – Created`, then the record has been created within Agile Lifecycle Manager. If the response code is anything other, then a problem has been encountered.
2. Use the following GET command on the Ishtar API to confirm the status of the new 'ucd' resource manager record.

   ```
   GET /api/resource-managers/ucd
   ```

   If the record has been created, the following response will be displayed:

   ```
   {
     "name": "ucd",
     "type": "ucd",
     "url": "http://<rm-ip-address>:<rm-api-port>/api/resource-manager"
   }
   ```

   In addition, the response body of the request returns information on the onboarded resource types and possible deployment locations, as depicted in the following example:

   ```
   {
     "resourceManagerOperation": "ADD",
     "deploymentLocations": {
       "test@local2": {
         "operation": "ADD",
         "success": true
       },
       "admin@local": {
         "operation": "ADD",
         "success": true
       },
       "admin@local2": {
         "operation": "ADD",
         "success": true
       }
     },
     "resourceTypes": {
       "resource::<your-rm>::1.0": {
         "operation": "ADD",
   ...
   ```

Results

A new resource manager record has been created in Agile Lifecycle Manager, which is now aware of the resource manager it references, and is able to access the resources it manages.

What to do next

If you attempt to create another record with the same name, a `409` error code will be returned.

You can obtain more information on the newly created resource manager by following the steps in "Exploring an onboarded resource manager" on page 66.

Updating a resource manager

Once a record for a resource manager instance has been created, you can perform an update to re-onboard the resource types.

Before you begin

The resource manager must have been onboarded before you can update it.

About this task

You can find more detailed information on the Ishtar API, its methods, and associated attributes in the "Lifecycle Manager API" on page 93 reference section.

Tip: If a resource descriptor with the same name already exists, it is not overridden. If a resource descriptor has been updated, but has the same name, it should be manually deleted from the catalog before trying to onboard the new version.

Procedure

1. To update an existing resource manager instance, use the following PUT command on the Ishstar API. The following example updates the record for '*<your-rm>*' by checking the original onboarding location of the resource manager for new or updated resources. If using the swagger API, the **`{id}`** parameter in the URL is set to the name value, in this case '*<your-rm>*'.
   ```
   PUT /api/resource-managers/test-rm
   {
     "name": "<your-rm>",
     "type": "<your-rm>",
     "url": "http://<your-rm>:8295/api/resource-manager"
   }
   ```
2. Use the following GET command on the Ishstar API to check the status of the resource manager following the update:
   ```
   GET /api/resource-managers/<your-rm>
   ```

   In addition, the response body of the request returns information on the onboarded resource types and possible deployment locations, as depicted in the following example:
   ```
   {
     "resourceManagerOperation": "ADD",
     "deploymentLocations": {
       "test@local2": {
         "operation": "ADD",
         "success": true
       },
       "admin@local": {
         "operation": "ADD",
         "success": true
       },
       "admin@local2": {
   ```

```
          "operation": "ADD",
          "success": true
        }
      },
      "resourceTypes": {
        "resource::<your-rm>::1.0": {
          "operation": "ADD",
  ...
```

Results

The resources for the updated resource manager are updated in the Agile Lifecycle Manager catalog.

Deleting a resource manager record

You can delete the record of a resource manager instance within Agile Lifecycle Manager.

Before you begin

The resource manager must have been onboarded before you can delete it.

About this task

After you delete the record of a resource manager instance, resources can no longer be created by that resource manager, and any resources already created can no longer be managed by that resource manager.

Note: The deployment locations for the resource manager are deleted. However, the resource descriptors in `var_alm/calalog/resources` are not deleted, and must be managed manually. See the following topic for more information on removing resource descriptors: "Removing an assembly descriptor" on page 62

Procedure

To delete a resource manager record, use the following DELETE command on the Ishstar API. The following example deletes the record for <*your-rm*>. If using the swagger API, the **`{id}`** parameter in the URL is set to the name value, in this case '<*your-rm*>'.

```
DELETE /api/resource-managers/<your-rm>
```

Successful deletion of <*your-rm*> is indicated by a `204` response code (Resource Manager was deleted).

What to do next

You can double-check that the <`your-rm`> has been deleted by running the GET command. If the response code to the GET is `404 – NOT FOUND`, then the DELETE action was successful.

Sample assembly descriptor

This reference topic contains the full assembly descriptor of the example assembly (`assembly::example::1.0`) used in the Getting Started section.

`assembly::example::1.0` sample descriptor

```
name: assembly::example::1.0
description: Assembly comprised of "components\\t_simple.yml"
properties:
  data:
    default: "data"
    type: string
    description: 'parameter passed'
  output:
    description: an example output parameter
    type: string
    read-only: true
  deploymentLocation:
    type: string
    description: name of openstack project to deploy network
    default: admin@local
  resourceManager:
    type: string
    description: name of the resource manager
    default:test-rm
composition:
  A:
    type: resource::t_simple::1.0
    quantity: '1'
    properties:
      referenced-internal-network:
        value: ${internal-network.id}
      reference-public-network:
        value: ${public-network.id}
      image:
        value: ${xenial-image.id}
      key_name:
        value: "ACCANTO_TEST_KEY"
      data:
        value: ${data}
      output:
        value: ${output}
      deploymentLocation:
        value: ${deploymentLocation}
      resourceManager:
        value: ${resourceManager}
references:
  internal-network:
    type: resource::openstack_neutron_network::1.0
    properties:
      deploymentLocation:
        value: ${deploymentLocation}
      resourceManager:
        value: ${resourceManager}
      name:
        type: string
        value: VIDEO
  public-network:
    type: resource::openstack_neutron_network::1.0
    properties:
      deploymentLocation:
        value: ${deploymentLocation}
      resourceManager:
        value: ${resourceManager}
      name:
        type: string
        value: public
  xenial-image:
    type: resource::openstack_glance_image::1.0
    properties:
      deploymentLocation:
        value: ${deploymentLocation}
      resourceManager:
        value: ${resourceManager}
      name:
        type: string
        value: xenial
```

Chapter 6. Administration

Use the following topics to understand administration tasks, such as viewing system logs, or adjusting global timeout limits for resource managers.

Monitoring system health

Docker containers have built-in health monitoring, which you can use to check if a service is still available.

Performing a health check

See the service REST API endpoints.

Managing the service logs

You can view logs for all Agile Lifecycle Manager services. You can also change the default logging behavior, such as the number of days logs are stored before they are deleted, the maximum storage space allocated, and the logging levels.

Before you begin

By default, logging rotates every day, keeping at most seven days with a maximum limit of 1GB of files. This means that with the five services that currently comprise Agile Lifecycle Manager at least 5GB of space must be available. If you increase the limit, ensure you have sufficient storage space available for your logs.

About this task

Logs are found in the `<install dir>/var_alm/logs` directory where `<install dir>` is the directory where Agile Lifecycle Manager is installed.

Remember: The default installation directory is `/opt/IBM/netcool`

Log files are named according to the Agile Lifecycle Manager component name, plus the date. New logs are created daily at midnight.

The supported logging levels are:
- TRACE
- DEBUG
- INFO
- WARN
- ERROR

To change the number of days logs are capped, or the maximum size of logs, the logging configuration must be updated. This can be done for all services, or per service.

Procedure

Change logging options for all services

1. To change the logging options for all services, edit the `var_alm/config-repo/application.yml` file as in the following example.

   Tip: An example line is in the application.yml file and is commented out. When you uncomment it, it should look like the following sample:
   ```
   logging:
     config: /var/alm/logback-spring.yml
   ```
2. Commit the change to Git.
3. Rename the example logback-sprint.yml file in the `var_alm/examples` directory, and move it to the `var_alm` directory.
4. Change the values for maxHistory and totalSizeCap in the renamed logback-sprint.yml file as in the following example:
   ```
   <!-- This part can be changed -->
      <!-- keep 7 days' worth of history capped at 1GB total size -->
         <maxHistory>7</maxHistory>
         <totalSizeCap>1GB</totalSizeCap>
      </rollingPolicy>
   <!-- End of part that can be changed -->
   ```

Change logging options for a specific service

5. Change the relevant yaml file in the config-repo directory to add a logging entry pointing to the log file.

   Note: The log file path is as used in the container, which is `/var/alm` and **not** `/var_alm/`.
6. Include the logback-spring.yml for the micro service in the log directory under `/var_alm/` so that it will be mounted within the container.

Change log levels

7. To change log levels, you POST to the `management/loggers` URL at runtime. The following example changes the log level for the Daytona service to DEBUG:
   ```
   POST http://<docker-host>:8281/management/loggers/com.accantosystems
   {
    "configuredLevel": "DEBUG"
   }
   ```
   - TRACE
   - DEBUG
   - INFO
   - WARN
   - ERROR

   Any data recorded at a log level above INFO, such as DEBUG or TRACE, **does not** appear in the console output, but only in the log file; that is in `var_alm/log/*`.
8. To check the result of your POST, perform a GET, as in the following example:
   ```
   GET http://<docker-host>:8281/management/loggers/com.accantosystems
   {
     "configuredLevel": "DEBUG",
     "effectiveLevel": "DEBUG"
   }
   ```

   The `configuredLevel` must be provided when changing the log level, and in this example is the new logging level to be set, while `effectiveLevel` is the current active logging level.

Setting timeout limits for resource managers

A request to a resource manager from Agile Lifecycle Manager times out if the request does not complete within the set timeout period. You can adjust the default limits for this period.

About this task

Timeout limits are a protection mechanism to prevent Agile Lifecycle Manager intents from hanging indefinitely. Timeout is measured by Agile Lifecycle Manager from the moment it receives a successful response from a resource manager.

The default timeout is 900 seconds (that is, 15 minutes). The timeout can be set in the following file by adding `default-timeout-duration` to the `resource-manager` section:

`<`*`distribution dir`*`>/var_alm/config_repo/daytona.yml`

To view or edit the `daytona.yml` file, use an appropriate text editor.

Procedure

1. Edit the `daytona.yml` file, as in the following example. In this example, the default timeout value is changed to 16 minutes (960 seconds).
   ```
   alm:
     daytona:
       resource-manager:
         default-timeout-duration: 960
   ```
2. Commit the configuration changes. In the `<`*`distribution dir`*`>/var_alm/config_repo` directory:
   ```
   git add daytona.yml
   git commit -m "changed global resource manager request timeout"
   ```
3. Restart Daytona. In the `<`*`distribution dir`*`>` directory:
   ```
   docker-compose -f alm-docker-compose.yml restart daytona
   ```

Results

The default timeout limits for resource managers has been changed.

Enabling HTTPS support (for the Nimrod service)

The Nimrod service is accessed through HTTP by default. This topic describes how you can enable HTTPS support instead.

About this task

Procedure

1. Create a keystore and self-signed certificate using the Java keytool. In the following example, the command is run from the `<`*`distribution_dir`*`>/var_alm` directory inside the installation directory:
   ```
   keytool -genkey -alias alm -storetype PKCS12 -keyalg RSA -keysize 2048
   -keystore keystore.p12 -validity 3650
   ```
2. Add the following properties to the nimrod.yml file in the`<`*`distribution_dir`*`>/var_alm/config-repo` directory. The password **must** match the one used in the previous step.

```
server:
  ssl:
    keyStore: file:/var/alm/keystore.p12
    keyStorePassword: pass1234
    keyStoreType: PKCS12
    keyAlias: alm
```

3. Commit the changes to the config-repo and restart the Nimrod service, as in the following example:

   ```
   git add nimrod.yml
   git commit -m "enabling https support"
   ```

4. In the <*distribution dir*>, restart Nimrod:

   ```
   docker-compose -f alm-docker-compose.yml restart nimrod
   ```

Results

Nimrod will now be accessible via HTTPS instead of HTTP.

Note: You may receive a warning about not being able to verify the certificate.

Ensuring Log files are not owned by the root user

When Agile Lifecycle Manager containers write to host-mounted Docker volumes (such as `var_alm/logs`) on a Linux host the files will, by default, be owned by root. To avoid this, it is possible to use a Docker-mandated approach that utilizes Linux uid remapping to map container uids and gids (user and group ids) to host uids and gids in a predictable manner.

About this task

Although this procedure will work on any Linux system with equivalent commands, the following steps assume an Ubuntu system is being used.

Procedure

1. Add the following to your `/etc/docker/daemon.json` file. Create the file if it does not exist:

   ```
   {
     "userns-remap": "USER"
   }
   ```

   Where USER is a user on your host system that will be configured to have access to any files written by the Docker containers. This configures the Docker daemon to apply uid and gid re-mapping between Docker containers and the host system.

2. Restart your Docker daemon:

   ```
   sudo /etc/init.d/docker restart
   ```

3. Add the following to the `/etc/subuid` file:

   ```
   USER:[USER uid]:1
   USER:165536:65536
   ```

   Substituting your chosen user for 'USER' and the uid of 'USER' for [USER uid]. This will map uid 0 (root) in your containers on to uid [USER uid] on the host (this means that files written by the microservices running inside the containers as root will be owned by USER), and uids 1 and upwards in your containers to the range starting at 165536. These uids on your host will be accessible by USER. You may choose whichever value you like here, as long as it does not overlap with other ranges in the same file.

4. Add the following to the `/etc/subuid` file:

   ```
   USER:165536:65536
   ```

   Substitute your chosen user for 'USER'. The start range value (165536) must be the same as in the previous step.

   Tip:

   If you want the root group inside the containers to map onto a specific group on your host, add the following instead:

   ```
   USER:[gid of group]:1
   USER:165536:65536
   ```

   You can find the official Docker documentation at the following site: https://docs.docker.com/engine/security/userns-remap/

Ensuring support for accented characters

To support accented characters correctly, it is imperative that the Java JVM being used to run Agile Lifecycle Manager is configured to support multi-byte characters.

Procedure

1. You configure the Java JVM being used to run Agile Lifecycle Manager to support multi-byte characters via the 'environment' section of the Docker compose `alm-docker-compose.yml` file as in the following example: The key settings are highlighted in bold text.

   ```
   ...
   environment:
    - spring.cloud.config.server.git.uri=file://var/alm/config-repo
    - eureka.instance.hostname=conductor
    - LOG_FOLDER=/var/alm/logs
    - spring.main.banner-mode=off
    - LANG=C.UTF-8
   ...
   ```
2. To check your JVM language settings you can use this command:

   ```
   docker exec daytona locale
   ```

   If the system output response is anything other than the values below then your Java JVM is incorrectly configured:

   ```
   LANG=C.UTF-8
   LC_CTYPE="C.UTF-8"
   LC_NUMERIC="C.UTF-8"
   LC_TIME="C.UTF-8"
   LC_COLLATE="C.UTF-8"
   LC_MONETARY="C.UTF-8"
   LC_MESSAGES="C.UTF-8"
   LC_PAPER="C.UTF-8"
   LC_NAME="C.UTF-8"
   LC_ADDRESS="C.UTF-8"
   LC_TELEPHONE="C.UTF-8"
   LC_MEASUREMENT="C.UTF-8"
   LC_IDENTIFICATION="C.UTF-8"
   ```

Authentication

The authentication mechanism in use in Agile Lifecycle Manager is OAuth2 combined with LDAP for user credential verification. Below is an explanation of some of the concepts and how authentication should be used in Agile Lifecycle Manager.

Client credentials

The resources of Agile Lifecycle Manager are protected by an authentication layer in place on the Gateway, Ishtar. In order to gain access to any of these protected resources, an authorisation process must take place. As a minimum, this would require providing some valid client credentials (client Id and secret) which would normally be deemed as sufficient security for any calling system into Agile Lifecycle Manager. Additionally, it is possible to require further user credentials (username and password), which would be administered on a per-user basis and should exist in the configured LDAP database. The Agile Lifecycle Manager UI (Nimrod) would be protected as such, where the Nimrod service itself provides the client credentials and the end user provides the additional user credentials.

The level of authorisation required is dictated by the Grant Type which must be specified when creating client credentials.

Bearer token

The OAuth2 mechanism makes use of Bearer tokens for authorisation. Once a client authenticates they are provided with a bearer token which can be used to authorise any subsequent interactions up until this token expires. It may then be possible to refresh this Bearer token (using an additional refresh token) to regain access after expiry of the Bearer token without needing to resupply all user credentials.

OAuth2 token lifetime configuration

When creating client credentials, there are 2 configurable values with these system defaults that control the lifetime of the OAuth2 tokens:

```
accessTokenValidity: 1200 #20 minutes
refreshTokenValidity: 30600 #8.5 hours
```

The unit for these values is seconds: 20 x 60 = 1200.

With these values, the behaviour will be that after 20 minutes, the system will refresh the Bearer token with a new one. At this point LDAP is checked to verify the user is still active. This happens every 20 minutes for 8.5 hours, after which the user will be forced to re-enter login credentials.

These default values are specified in the Ishtar configuration YAML file:

```
alm:
    ishtar:
        security:
            defaultAccessTokenValidity: 1200 #20 minutes
            defaultRefreshTokenValidity: 30600 #8.5 hours
```

Note: Changing these values will only affect the creation of new clients using the bootstrap process and will not affect any existing clients in the system. Intended usage of the system for changing these timeout values would be that client credentials (including these values) would be modified using the Client Credentials REST API as described in the following section:"Creating further client credentials" on page 80

Grant types

The following grant types are supported in Agile Lifecycle Manager:

Table 7. Agile Lifecycle Manager grant types

Grant type	Description
client_credentials	Authorisation requires only a valid **Client Id** and **Secret**. Recommended for calling systems.
password	Authorisation requires a valid **Client Id** and **Secret** and additionally valid **User Credentials**. Recommended for a UI providing human interaction.
refresh_token	This grant type can be used in addition to the password grant type and indicates that a caller or user can re-authenticate themselves using a Refresh Token without having to re-provide any user credentials.

Bootstrapping client credentials

In order to use the system, including to generate client credentials, there will need to exist some client credentials to authenticate against. There is a bootstrap process which can be used to initially load any clients into the system.

By default, when Ishtar initialises, it looks for a file with the path of: `/var/alm/bootstrap/client-credentials-bootstrap.yml`.

This path can be overridden in the application config by specifying a filepath under the property `alm.ishtar.security.clientCredentialsBootstrapFile`. The contents of the file should look similar to the following example:

```
clientCredentials:
    - clientId: DefaultClient
      clientSecret: DefaultClient
      grantTypes: client_credentials
    - clientId: DefaultClient2
      grantTypes: password, refresh_token
```

This would generate the client `DefaultClient` with the specified password and the grant type of `client_credentials`, and a client called `DefaultClient2` with a system generated password and the grant type of `password` and `refresh_token`.

The generated password for `DefaultClient2` will be output into the startup logs for Ishtar once only when the user is first generated. The clients will be created with default expiry times for the bearer and refresh tokens.

This file (if existing) will be processed on startup of Ishtar and the clients created. The file will then be deleted automatically. If the file specifies a client ID for an existing client then the startup of Ishtar will fail (this is a deliberate mechanism to avoid overwriting existing clients).

Creating further client credentials

In order to generate Client Credentials, a REST endpoint is provided on Ishtar. This endpoint is protected, so authentication is required to access it (an initial client must be boot-strapped into the system).

```
POST http://<docker-host>:8280/api/credentials
```

It expects a JSON payload in the following format:

```
{
    "clientId": "AlmClient",
    "clientSecret": "BE6fJ02mJuhz37GA",
    "scope": [
        "all"
    ],
    "authorisedGrantTypes": [
        "password",
        "refresh_token"
    ],
    "accessTokenValidity": 300,
    "refreshTokenValidity": 500
}
```

This means that the user's credentials will be checked against the LDAP server every 300 seconds (5 minutes) and the will be forced to logon every 10 minutes.

Care should be taken when choosing the values for the token lifetimes as they cannot be changed easily via the LDAP server. Particular care should be taken when choosing the value of the `refreshTokenValidity` value as forcing a user to logon every 10 minutes could lead to service complaints.

Note: Agile Lifecycle Manager doesn't yet support different types of scope. It is recommended for now that a scope of `all` is specified. Similarly, it is possible to update existing clients using a similar request to the following endpoint:

```
PUT http://<docker-host>:8280/api/credentials/<client-id>
```

Authorizing

Once client credentials are setup, it is possible to authorise using them. This example demonstrates how to authenticate given a client with a grant type of 'client_credentials'. To acquire a certificate, the client must post to the auth endpoint below:

```
POST http://<docker-host>:8280/oauth/token
```

The request must contain the client credentials in the header. This is concatenation of the id:secret, Base64 encoded, prefixed with 'Basic' under a header key of 'Authorization' (this is a standard OAuth2 mechanism).

The request body should be `x-www-form-urlencoded` and include this key/value:

```
grant_type=client_credentials
```

This will return a response similar to this:

```
{
    "access_token": "fdf8e754-1abe-42ae-b064-7969b05788ca",
    "token_type": "bearer",
    "expires_in": 1199,
    "scope": "all"
}
```

User credentials in LDAP

If the client has a grant type of 'password', then an additional authentication step of verifying user credentials will take place. These credentials are expected to belong in LDAP. An out-of-the box installation of Open LDAP is provided which Agile Lifecycle Manager is configured to point to.

Agile Lifecycle Manager can be configured to use an existing LDAP database. This would involve modifying the following config in the `var_alm\config-repo\ishtar.yml` file:

```
alm:
    ishtar:
        security:
            ldap:
                url: ldap://alm-openldap:389
                base: dc=alm,dc=com
                managerDn: cn=admin,${alm.ishtar.security.ldap.base}
                managerPassword: almadmin
                userSearchBase: ou=people
                userSearchFilter: uid={0}
                passwordAttribute: userPassword
```

Tip: Agile Lifecycle Manager provides a mechanism for initially loading users into the OpenLDAP database on initial creation, but beyond this provides no mechanism for managing users. There are many available LDAP clients which can be used for such purposes. One such client is the free Windows software LDAP Admin. See the following site for more information: http://www.ldapadmin.org/

See "Provided OpenLDAP LDAP server" on page 82 for the details of the configuration to be used with any LDAP client software to allow it to connect to the default OpenLDAP server that comes as standard with Agile Lifecycle Manager.

Secret and Password Encryption

By default, any client secrets stored in Agile Lifecycle Manager will be encoded with the BCrypt algorithm before they are stored in the local database. This is handled seamlessly and has no apparent difference to any calling systems.

Additionally, BCrypt encoding is the default option for LDAP passwords. In this case this is more apparent, as there is an expectation that any passwords stored in LDAP are encoded with BCrypt.

Alternatively, it is possible to use Agile Lifecycle Manager with an LDAP database that stores passwords in plain text, by changing the following application property in the `var_alm\config-repo\ishtar.yml` file:

```
alm.ishtar.security.ldap.passwordEncoding: PLAIN
```

Audit logging

The Agile Lifecycle Manager Gateway (Ishtar) maintains an audit log of all authentication attempts and all API requests that come through it. By default, these logs are stored as files and kept in the `{LOG_FOLDER}/security-audit directory` directory.

Configuring the audit log

Audit Logging of both authentication and API requests are enabled by default.

It can be disabled by setting the audit properties in the `/var_alm/config-repo/ishtar.yml` file.

Disabling only the logging of all the API requests is possible using the `includeHttpRequests` property:

```
alm:
  ishtar:
    security:
      audit:
        # Set to 'false' to disable ALL audit logs
        enabled: true
        # Set to 'false' to disable only audit logs of API requests
        includeHttpRequests: true
```

Audit log format

The format of the audit log is configured using an alternative Appender called SECURITY_AUDIT in the `Logback xml` configuration file. By default, it is configured in Ishtar using the following settings:

```
<appender name="SECURITY_AUDIT" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
      <!-- Daily Rollover or when file reaches 1GB -->
            <fileNamePattern>${LOG_FOLDER:-.}/security-audit/${eureka.instance.instanceId:
-#project.artifactId#}-security-audit.%d{yyyy-MM-dd}.%i.log.gz</fileNamePattern>
            <maxFileSize>1GB</maxFileSize>
        </rollingPolicy>
        <encoder>
            <charset>utf-8</charset>
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} ${LOG_LEVEL_PATTERN:-%5p}
--- %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}</pattern>
        </encoder>
    </appender>
```

Provided OpenLDAP LDAP server

The Docker Agile Lifecycle Manager package includes an OpenLDAP container and is the default LDAP server used for user management.

When running Agile Lifecycle Manager for the first time there are no users in the provided LDAP server. Initial users may be added by modifying the `/var_alm/ldap/initial_users.ldif` file.

Note: This file is only used to create users on the first start-up of the OpenLDAP container. To re-use this file the persistence volume of the container must be removed before the next start-up.

A user entry must take the format of:

```
dn: uid=Admin,ou=people,{{ LDAP_BASE_DN }}
changetype: add
objectClass: person
objectClass: uidObject
cn: Admin
sn: Admin
uid: Admin
userPassword: $2a$10$MzDrvf/9rsuzDRDkpvb1M.yfv0Vc2O.p3LUegU8AszlRwaBnIQ03W
```

The Agile Lifecycle Manager installation includes a script (`generate-ldap-user.sh`) to help generate the correct entries. To create further users any LDAP client can be used, or the LDAP protocol may be used directly.

One such LDAP client is the free Windows software LDAP Admin. See the following site for more information: http://www.ldapadmin.org

Use the following example to configure an LDAP client for anyone using the out-of-the-box Agile Lifecycle Manager OpenLDAP service with the default Agile Lifecycle Manager configuration:

```
Host: <docker-ip>
Port: 389
Base: dc=alm,dc=com
Username: cn=admin,dc=alm,dc=com
Password: almadmin
```

Example alm-docker-compose.yml file

This topic contains an example Docker compose file (alm-docker-compose.yml) for reference only.

Important: Use the following example Docker compose file for reference purposes only.

```
version: '3'
services:
  alm-cassandra:
     container_name: "alm-cassandra"
     image: cassandra:3.10
     ports:
       - "9042:9042"
       - "9160:9160"
     networks:
       - alm
     volumes:
       - cassandradata1:/var/lib/cassandra/data
     restart: always
     environment:
       # important: broadcast address must be set to the Docker hostname
       - CASSANDRA_BROADCAST_ADDRESS=alm-cassandra
       - CASSANDRA_CLUSTER_NAME=alm-cassandra-cluster
       - CASSANDRA_SEEDS=alm-cassandra
       - CASSANDRA_REMOTE_CONNECTION=true
       - CASSANDRA_START_RPC=true
  zookeeper:
    container_name: "zookeeper"
    image: wurstmeister/zookeeper
    restart: always
    networks:
      - alm
    ports:
      - "2181:2181"
  kafka:
    container_name: "kafka"
    image: wurstmeister/kafka:latest
    restart: always
    networks:
      - alm
    ports:
      - "9092:9092"
```

```
⌂
    environment:
      KAFKA_HOST_NAME: kafka
      KAFKA_PORT: 9092
      KAFKA_ADVERTISED_HOST_NAME: ${KAFKA_ADVERTISED_HOST_NAME:-kafka}
      KAFKA_ADVERTISED_PORT: 9092
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_CREATE_TOPICS: "alm__health:1:1:compact,
alm__descriptorChange:1:1:compact,alm__processStateChange:1:1,alm__stateChange:1:1,alm__taskUpdate:1:1"
  conductor:
    container_name: "conductor"
    hostname: "conductor"
    build:
      context: ./conductor
      dockerfile: Dockerfile
      args:
        JDK_IMAGE: ${JDK_IMAGE:-openjdk}
        JDK_VERSION: ${JDK_VERSION:-8u121-jre}
    restart: always
    networks:
      - alm
    volumes:
      - "./var_alm:/var/alm"
    ports:
     - "8761:8761"
    environment:
     - spring.cloud.config.server.git.uri=file://var/alm/config-repo
     - eureka.instance.hostname=conductor
     - LOG_FOLDER=/var/alm/logs
     - spring.main.banner-mode=off
     - LANG=C.UTF-8
  watchtower:
    container_name: "watchtower"
    hostname: "watchtower"
    build:
      context: ./watchtower
      dockerfile: Dockerfile
      args:
        JDK_IMAGE: ${JDK_IMAGE:-openjdk}
        JDK_VERSION: ${JDK_VERSION:-8u121-jre}
    restart: always
    networks:
      - alm
    volumes:
      - "./var_alm:/var/alm"
⌂
    depends_on:
      - kafka
      - zookeeper
    links:
      - kafka
      - zookeeper
    ports:
      - "8284:8284"
    environment:
     - spring.cloud.config.uri=http://admin:admin@conductor:8761/config
     - spring.cloud.config.failFast=true
     - eureka.client.serviceUrl.defaultZone=http://admin:admin@conductor:8761/eureka/
     # this is needed so that Watchtower registers the correct hostname/IP
address to the Conductor/Eureka
     - eureka.instance.ipAddress=watchtower
     - LOG_FOLDER=/var/alm/logs
     - LANG=C.UTF-8
  relay:
    container_name: "relay"
    hostname: "relay"
    build:
      context: ./relay
      dockerfile: Dockerfile
      args:
        JDK_IMAGE: ${JDK_IMAGE:-openjdk}
        JDK_VERSION: ${JDK_VERSION:-8u121-jre}
    restart: always
    volumes:
      - "./var_alm:/var/alm"
    networks:
      - alm
    depends_on:
      - kafka
    links:
      - kafka
    ports:
      - "8285:8285"
⌂
    environment:
     - spring.cloud.config.uri=http://admin:admin@conductor:8761/config
     - spring.cloud.config.failFast=true
     - eureka.client.serviceUrl.defaultZone=http://admin:admin@conductor:8761/eureka/
     # this is needed so that Relay registers the correct hostname/IP
address to the Conductor/Eureka
     - eureka.instance.ipAddress=relay
     - LOG_FOLDER=/var/alm/logs
     - spring.main.banner-mode=off
     - LANG=C.UTF-8
  simple-rm:
    container_name: "simple-rm"
    hostname: "simple-rm"
    build:
      context: ./simple-rm
      dockerfile: Dockerfile
      args:
        JDK_IMAGE: ${JDK_IMAGE:-openjdk}
        JDK_VERSION: ${JDK_VERSION:-8u121-jre}
    restart: always
    volumes:
      - "./var_alm:/var/alm"
      - "./simple-rm-data:/data/simple-rm"
      - simplermstorage:/data/simple-rm-storage
    networks:
      - alm
    depends_on:
      - kafka
      - relay
      - ishtar
    links:
      - kafka
      - relay
      - ishtar
    environment:
     # these should not be modified
     - alm.simplerm.config.name=test-rm
     - LOG_FOLDER=/var/alm/logs
     - alm.simplerm.directory=/data/simple-rm/test-rm
     #used in commands.sh
     - alm_simplerm_directory=/data/simple-rm/test-rm
     - alm.simplerm.storage=/data/simple-rm-storage
     - spring.main.banner-mode=off
     - LANG=C.UTF-8
⌂
    ports:
      - "8295:8295"
  simple-rm-register:
    container_name: "simple-rm-register"
    build:
      context: ./simple-rm-register
      dockerfile: Dockerfile
    networks:
      - alm
    volumes:
      - "./var_alm:/var/alm"
      - "./simple-rm-data:/data/simple-rm"
    depends_on:
      - ishtar
      - simple-rm
    links:
      - ishtar
      - simple-rm
    environment:
     - LANG=C.UTF-8
  apollo:
    container_name: "apollo"
    hostname: "apollo"
    build:
      context: ./apollo
      dockerfile: Dockerfile
      args:
        JDK_IMAGE: ${JDK_IMAGE:-openjdk}
        JDK_VERSION: ${JDK_VERSION:-8u121-jre}
    restart: always
    networks:
      - alm
    ports:
     - "8282:8282"
    depends_on:
      - "conductor"
      - "kafka"
      - "alm-cassandra"
    volumes:
      - "./var_alm:/var/alm"
⌂
    environment:
     - spring.cloud.config.uri=http://admin:admin@conductor:8761/config
     - spring.cloud.config.failFast=true
     - eureka.client.serviceUrl.defaultZone=http://admin:admin@conductor:8761/eureka/
     # this is needed so that Apollo registers the correct hostname/IP
address to the Conductor/Eureka
     - eureka.instance.ipAddress=apollo
     - LOG_FOLDER=/var/alm/logs
     - spring.main.banner-mode=off
     - LANG=C.UTF-8
  galileo:
    container_name: "galileo"
    hostname: "galileo"
    build:
      context: ./galileo
      dockerfile: Dockerfile
      args:
        JDK_IMAGE: ${JDK_IMAGE:-openjdk}
        JDK_VERSION: ${JDK_VERSION:-8u121-jre}
    restart: always
    networks:
      - alm
    volumes:
      - "./var_alm:/var/alm"
    environment:
     - spring.cloud.config.uri=http://admin:admin@conductor:8761/config
     - spring.cloud.config.failFast=true
     - eureka.client.serviceUrl.defaultZone=http://admin:admin@conductor:8761/eureka/
     # this is needed so that Galileo registers the correct hostname/IP
address to the Conductor/Eureka
     - eureka.instance.ipAddress=galileo
     - LOG_FOLDER=/var/alm/logs
     - spring.main.banner-mode=off
     - LANG=C.UTF-8
    ports:
     - "8283:8283"
    depends_on:
      - "conductor"
      - "alm-cassandra"
⌂
  daytona:
    container_name: "daytona"
    hostname: "daytona"
    build:
      context: ./daytona
      dockerfile: Dockerfile
      args:
        JDK_IMAGE: ${JDK_IMAGE:-openjdk}
        JDK_VERSION: ${JDK_VERSION:-8u121-jre}
    restart: always
    networks:
      - alm
    volumes:
      - "./var_alm:/var/alm"
    ports:
     - "8281:8281"
    depends_on:
      - "conductor"
      - "simple-rm"
    environment:
     - spring.cloud.config.uri=http://admin:admin@conductor:8761/config
     - spring.cloud.config.failFast=true
     - eureka.client.serviceUrl.defaultZone=http://admin:admin@conductor:8761/eureka/
     # this is needed so that Daytona registers the correct hostname/IP
address to the Conductor/Eureka
     - eureka.instance.ipAddress=daytona
     - LOG_FOLDER=/var/alm/logs
     - spring.main.banner-mode=off
     - LANG=C.UTF-8
  ishtar:
    container_name: "ishtar"
    hostname: "ishtar"
    build:
      context: ./ishtar
      dockerfile: Dockerfile
      args:
        JDK_IMAGE: ${JDK_IMAGE:-openjdk}
        JDK_VERSION: ${JDK_VERSION:-8u121-jre}
    restart: always
    networks:
      - alm
    volumes:
      - "./var_alm:/var/alm"
⌂
    depends_on:
      - conductor
    environment:
     - spring.cloud.config.uri=http://admin:admin@conductor:8761/config
     - spring.cloud.config.failFast=true
     - eureka.client.serviceUrl.defaultZone=http://admin:admin@conductor:8761/eureka/
     # this is needed so that Ishtar registers the correct hostname/IP
address to the Conductor/Eureka
     - eureka.instance.ipAddress=ishtar
     - LOG_FOLDER=/var/alm/logs
     - spring.main.banner-mode=off
     - LANG=C.UTF-8
    ports:
      - "8280:8280"
  nimrod:
    container_name: "nimrod"
    hostname: "nimrod"
    build:
      context: ./nimrod
      dockerfile: Dockerfile
      args:
        JDK_IMAGE: ${JDK_IMAGE:-openjdk}
        JDK_VERSION: ${JDK_VERSION:-8u121-jre}
    networks:
      - alm
    volumes:
      - "./var_alm:/var/alm"
    ports:
     - "8290:8290"
    depends_on:
      - galileo
      - apollo
    environment:
     - spring.cloud.config.uri=http://admin:admin@conductor:8761/config
     - spring.cloud.config.failFast=true
     - eureka.client.serviceUrl.defaultZone=http://admin:admin@conductor:8761/eureka/
     # this is needed so that Nimrod registers the correct hostname/IP
address to the Conductor/Eureka
     - eureka.instance.ipAddress=nimrod
     - LOG_FOLDER=/var/alm/logs
     - spring.main.banner-mode=off
     - LANG=C.UTF-8
⌂
volumes:
  cassandradata1:
    driver: local
  simplermstorage:
    driver: local
networks:
  alm:
    driver: bridge
```

Chapter 7. Reference

Use the following reference information to enhance your understanding of the Agile Lifecycle Manager APIs and YAML specifications.

Restriction: Code samples provided in this section may contain references to test data and other example text not relevant to your own scenario.

API HTTP status codes reference

Agile Lifecycle Manager provides both a graphical UI and an HTTP API allowing the creation and administration of assemblies. HTTP status codes strategy and response messages are described here.

REST and RPC mechanisms

Agile Lifecycle Manager consists of micro-services that use HTTP as the transport mechanism for requests and responses. A combination of REST interfaces and RPC interfaces are used.

RPC Used to submits intents to Agile Lifecycle Manager

REST Used by all other interfaces

Note: Due to the use of combined RTC and REST interfaces, the RTC IETF4 RFC 7231 specifications as documented at the following site are not strictly adhered to: https://tools.ietf.org/html/rfc7231

Supported methods

Each API service endpoint can implement a different set of HTTP request methods. The following methods are supported within Agile Lifecycle Manager:

GET The GET method requests a representation of the specified resource. Requests using GET only retrieve data and have no other effect.

PUT The PUT method requests that the enclosed entity be stored under the supplied URI. If the URI refers to an existing resource, it is modified; if the URI does not point to an existing resource, then the request will be rejected.

POST The POST method requests that the server accept the entity enclosed in the request as a new instance of the resource identified by the URI.

DELETE
The DELETE method simply removes the specified resource (if it exists).

Note: The Agile Lifecycle Manager API does not support API versioning, but remains backwards-compatible as far as possible.

Response calls

Every call to the Agile Lifecycle Manager API returns an HTTP status code.

If the request was unsuccessful, a JSON5 or YAML6 formatted response will be returned in the response body. The response return format depends on the format the of the original API intention. For example if an API supports YAML and you requested YAML, then the response call will be YAML. Mostly, responses will be JSON.

If the Agile Lifecycle Manager request was **successful**, then an HTTP 2xx status code will be returned. Different API calls will potentially return HTTP 2xx status codes. The details of the status codes returned can be found in the individual micro-service Swagger documentation or API guides.

If the request was **unsuccessful**, then for all but HTTP 404 errors the response will be in one of two forms, either generated by Agile Lifecycle Manager, or by some other process (rarely).

Errors generated by Agile Lifecycle Manager

These errors are generated when Agile Lifecycle Manager detects an error associated with an API request, and returns output containing information about the error.

More than one set of error details may be included. These will show the errors generated by components internal to Agile Lifecycle Manager and passed back through the layers of requests that were made.

Note: If you contact IBM support, include all the information contained in these Agile Lifecycle Manager error reports.

Response field	Meaning
url	The Service endpoint that was requested.
localizedMessage	The translated or localized error message that can be presented back to the end user making the request.
details	Further details on the issue presented in a structured format. This is an optional field so will not always be present. It is intended to give further technical details on the source of the error, and so may not be relevant to the user making the request. Often it contains internal error information from the Agile Lifecycle Manager components that have handled or detected the error. When available this information should be included in any support requests.

Example of a simple error response:

```
{
   "url": /api/resource-manager/923227664489862,
   "localizedMessage": "A FATAL ALM Driver error has occurred: Unknown Resource Manager 923227664489862",
   "details": {}
}
```

Example of a complex error response:

```
{
   "url": /api/resource-manager/configuration/923227664489862,
   "localizedMessage": "A FATAL ALM Driver error has occurred: Unknown Resource Manager 923227664489862",
   "details": {
      "responseHttpStatus": 500,
      "errorStatus": "FATAL",
      "responseData": { "url": "http://galileo:8283/api/topology/resource-managers/923227664489862";,
      "localizedMessage": "Unknown Resource Manager 923227664489862",
            "details": {}
          }
        }
      }
```

Errors *not* generated by Agile Lifecycle Manager

On rare occasions an error may occur for which no Agile Lifecycle Manager error code exists, resulting in a generic response containing the following fields:

Response field	Meaning
timestamp	The date and time to error occurred
status	The HTTP status code returned
error	A textual description of the HTTP status code returned
exception	The internal exception type that was raised
message	The actual error reported
path	If it was an API call or web service request that failed, then this field will identify what was being requested

Example:

```
{
   "timestamp": "2017-08-10T15:28:19.586+0000",
   "status": 500,
   "error": "Internal Server Error",
   "exception": "org.springframework.web.client.ResourceAccessException",
   "message": "I/O error on GET request for \"http://9.20.64.abc:8295/api/resource-manager/configuration\":
9.20.64.abc; nested exception is java.net.UnknownHostException: 9.20.64.abc",
   "path": "/api/resource-managers"
}
```

HTTP status code use

Within the header of the HTTP response received in reply to an HTTP request there is a three-digit decimal status code. The first digit of the status code specifies one of five standard classes of responses. The distinction between 4xx (client) and 5xx (server) errors makes integration easier.

1xx 1xx informational responses are not used explicitly by Agile Lifecycle Manager.

2xx 2xx response codes indicate success. This means that the action requested by the client was received, understood, accepted, and processed successfully.

Different Agile Lifecycle Manager components return different HTTP 2xx status codes dependent on the nature of the request made. To find out which HTTP 2xx status codes will be returned, see the individual micro-service Swagger documentation or the related API documentation.

3xx 3xx redirection responses are not used explicitly by Agile Lifecycle Manager.

4xx 4xx errors are client-based errors. These errors usually occur when a service exists and a successful connection has been established with an API endpoint, but the request does not contain all of the information it requires to 'understand' the request.

400 errors indicate an invalid request, which means either that mandatory parameters are missing, or that syntactically invalid parameter values have been detected (for example an expected URL being text only).

404 errors indicate that a requested API service cannot be found, or that a requested entity cannot be found.

More detailed 4xx status code information can be found in the following section: "4xx status codes"

5xx 5xx errors are server-based errors. These errors usually occur when a request was syntactically correct with all the required parameters, but Agile Lifecycle Manager was unable to carry out the action.

The HTTP response body will contain reasons for the failure.

Tip: The most usual case will be a simple HTTP 500 status code backed up with a JSON payload in the response body. This payload will contain a human-readable error message summarizing the problem and a section containing further technical details if available.

4xx status codes

These error codes are associated with client or user errors, but can also be caused by another system sending requests to Agile Lifecycle Manager.

To understand how the different HTTP 4xx status codes are used we need to separate the definition of a Service Endpoint from the data it is being asked to use. The service endpoint in a request is the full path to the service required **excluding** any variable parameters in the URL. For example, in the following request the service endpoint is indicated in **bold**.

`http:// <hostname>:<Port>/api/resource-manager/configuration``/9.20.64.abc`

- All POST requests will send their data in the body of the HTTP request.
- All GET and DELETE requests will send their data as variable URL parameters.
- All PUT requests will send their data as variable URL parameters and/or as data in the body of the HTTP request.

The handling of HTTP 400 and 404 status codes varies depending on what type of request was made. Generally, the 4xx HTTP Status codes will be produced by the Agile Lifecycle Manager functionality concerned with validating and verifying the incoming requests.

Table 8. Summary of how Agile Lifecycle Manager uses the HTTP 400 and 404 status codes

Type	URL invalid	URL (with variables)*	Invalid content	Entity does not exist
REST	404	404	400	n/a
RPC	404	n/a (no URLs pointing to entity)	400	400

* refers to an entity that does not exists

REST requests

An HTTP 404 error will be returned if the entire requested service URL, including any variable parameters, does not exist. This could be for one of the following reasons:

- The service endpoint requested does not exist or is not available (check your host name and ports).

- The requested resource, as identified by the URL variable parameters, does not exist.

The HTTP 400 status code (Bad Request) is used by the REST-style requests if the POST or PUT contains bad data. For example, if Agile Lifecycle Manager is given a Descriptor that it does not recognize.

RPC requests
An HTTP 404 error will be returned if the requested service endpoint does not exist.

An HTTP 400 error will be returned if there is incorrect content in the request body sent to the service. endpoint.

HTTP 409 error status:
The HTTP 409 status code (Conflict) indicates that the request could not be processed because of conflict in the request, such as the requested resource is not in the expected state, or the result of processing the request would create a conflict within the resource.

Examples of where an HTTP 409 status code would be used within Agile Lifecycle Manager are:

- Data constraint violations
- Concurrent modification exceptions.

Related concepts:

"Lifecycle Manager API"
The Lifecycle Manager API is responsible for interactions with the operations available from Agile Lifecycle Manager. This section covers the definition of the Lifecycle Manager API and the specification of the messages sent across this interface.

"Resource Manager API" on page 121
The Resource Manager API is responsible for defining the interactions between a lifecycle manager and the resource managers that are used to manage resources within virtual (or physical) infrastructures.

"Resource descriptor YAML specifications" on page 138
This section describes the descriptors that are used by Agile Lifecycle Manager.

"Assembly descriptor YAML specifications" on page 147
This section describes the assembly descriptors that are used by Agile Lifecycle Manager.

Lifecycle Manager API

The Lifecycle Manager API is responsible for interactions with the operations available from Agile Lifecycle Manager. This section covers the definition of the Lifecycle Manager API and the specification of the messages sent across this interface.

Related concepts:

Chapter 5, "Getting started (using the APIs)," on page 47
Agile Lifecycle Manager provides both a graphical UI and an HTTP API allowing the creation and administration of assemblies. This section describes a set of basic scenarios to get started using the APIs.

Interface architecture

This topic describes the API interaction principles, lists the possible HTTP error response codes, and outlines the four API sections.

API interaction principles

All of the message descriptions are in JSON format and should be submitted with the HTTP content-type header of `application/json`.

All Dates will conform to the ISO-8601 standard.

API definition conventions:

A table is associated with each example that explains the fields it contains, including the name of the field, a brief description, and whether the field is mandatory. Whether a field is required or not is based on the context of the examples. The underlying API definition may mark a field as optional, but in some contexts, the fields must be supplied.

Any field names in italics are examples only.

Possible HTTP error response codes

The following table lists the HTTP response codes that can be returned in various error scenarios. Any client should expect that any API call can return these codes under exceptional circumstances.

Table 9. HTTP error response codes

Code	Description
400 - Bad Request	The request contained invalid information. This may be an incorrect field, invalid value or an inconsistent state on a dependent resource. The HTTP response body should contain a JSON message with further details of the specific issue.
404 - Not Found	The requested resource or endpoint could not be found.
409 - Conflict	Agile Lifecycle Manager has been unable to process the request due to a conflict produced by some of the information supplied. For example, due to attempting to create two resource managers with the same name.
500 - Internal Server Error	An internal error has occurred whilst fulfilling the request. The HTTP response body should contain a JSON message with further details. In some situations, it may be necessary for a system administrator to consult the logs for further information.
502 - Bad Gateway	A remote system has failed to respond correctly causing this request to fail. The HTTP response body should contain a JSON message with further details. In some situations, it may be necessary for a system administrator to consult the logs for further information.
503 - Service Unavailable	Agile Lifecycle Manager is unable to process this request at this time. The request should not be retried until the underlying problem is resolved. The HTTP response body should contain a JSON message with further details. In some situations, it may be necessary for a system administrator to consult the logs for further information.

API sections

The API is divided into four sections. These provide access to the different functionality provided by Agile Lifecycle Manager. This API is expected to be called by External OSS systems to perform both fulfillment tasks and also some fault management tasks.

Managing assemblies
Agile Lifecycle Manager allows the creation of assemblies. These allow services to be created by Agile Lifecycle Manager, which will interact with resource managers to create and manage virtual resources that need to be provided for the service to work.

This part of the API includes two endpoints; the first allows the external OSS system to request a transition against an assembly instance, including a request for a new assembly instance. The second allows the external OSS to poll Agile Lifecycle Manager to find out the state of the request.

Asynchronous events
In response to Assembly Orchestration requests, Agile Lifecycle Manager will also place on a Kafka bus messages that describe the key events that occur during the processing of the request and also a message to indicate when the processing has been completed. It is recommended that the external OSS use this mechanism to check the state of requests rather than using the polling interface defined in the previous section.

Assembly topology
External OSS may need details of the assembly instances and of the components that it is comprised. The topology contains a hierarchy of the components of an assembly. A component may be either a resource or an assembly. It is also possible to request details of the events used during the assemblies life.

Resource manager handling
Resource Managers are responsible for managing the actual resources that are needed for a service to work. Agile Lifecycle Manager needs to be told which resources managers it will interface to. This process is known as 'Onboarding a Resource Manager'. The Lifecycle Manager API provides a set of calls that allows a new resource manager instance to be onboarded to Agile Lifecycle Manager, and also removed from Agile Lifecycle Manager. When a resource manager is onboarded the set of resource types and locations that they manage will be extracted using the Resource Manager API. The API also provides an endpoint that will make Agile Lifecycle Manager request the associated resource manager for a set of updated resources. Any existing resources will remain unchanged when this update occurs.

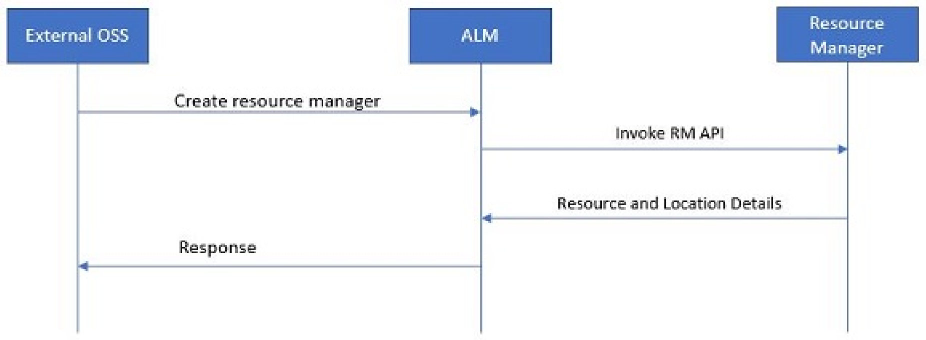
Scenarios

This section describes the lifecycle manager scenarios.

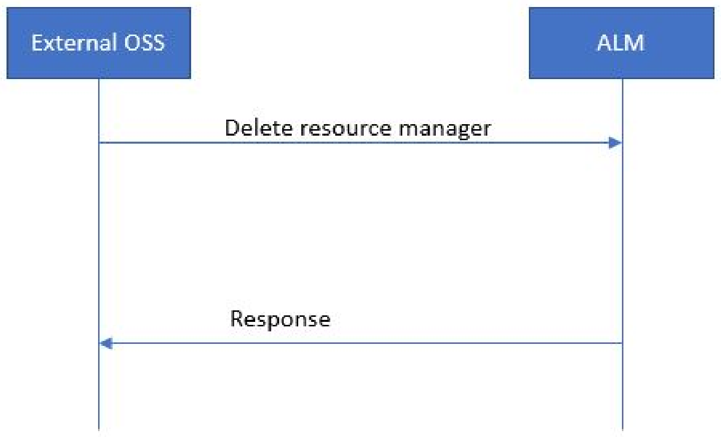
Resource manager handling

When a resource manager is onboarded, Agile Lifecycle Manager invokes a set of calls to the resource manager detailed in Resource Manager API.

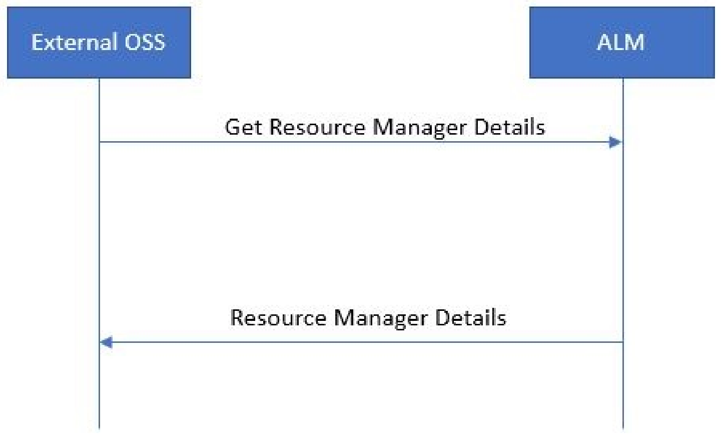
Onboard resource manager



Delete resource manager

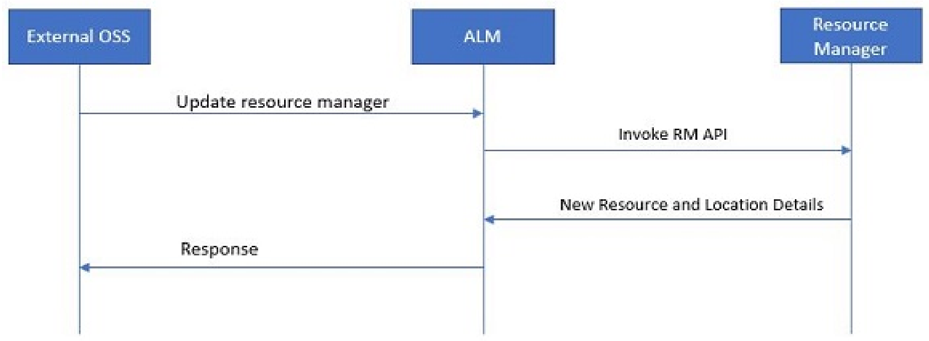


Get resource manager details

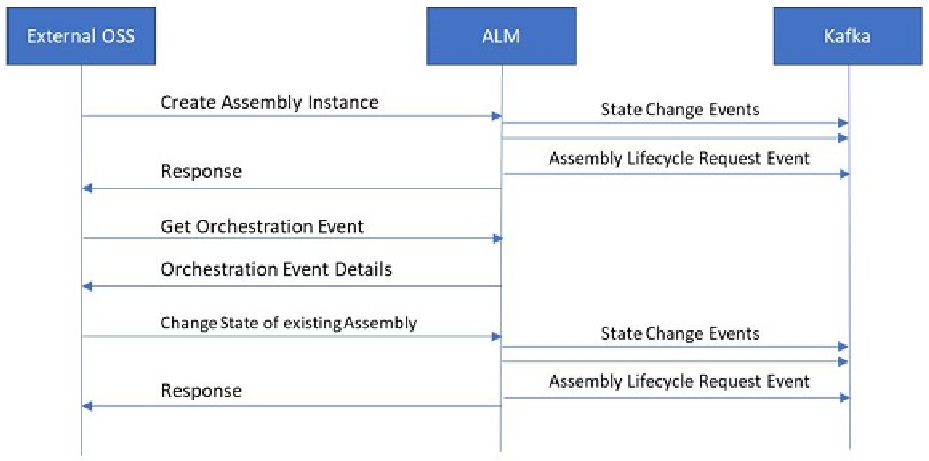


When a resource manager is updated, it returns the details of the resources and locations, and Agile Lifecycle Manager will store any new details, but not remove any existing details.

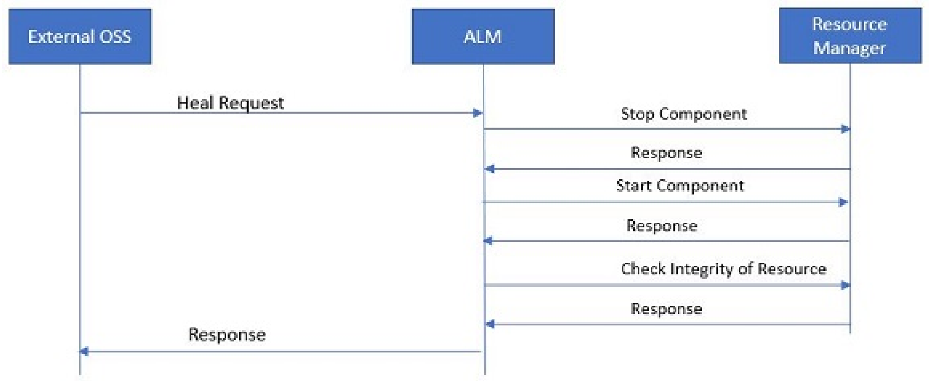
Update resource manager



Assembly creation and state transition


External OSS
ALM
Kafka
Create Assembly Instance
State Change Events
Assembly Lifecycle Request Event
Response
Get Orchestration Event
Orchestration Event Details
Change State of existing Assembly
State Change Events
Assembly Lifecycle Request Event
Response

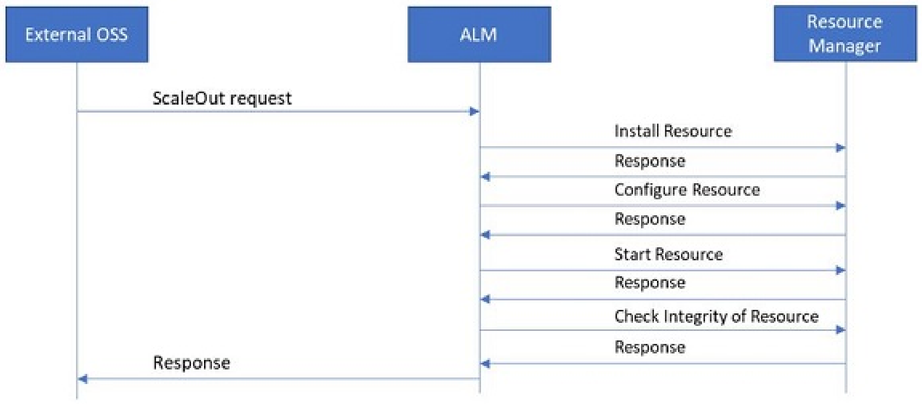
Heal



Note: Asynchronous events are not depicted.

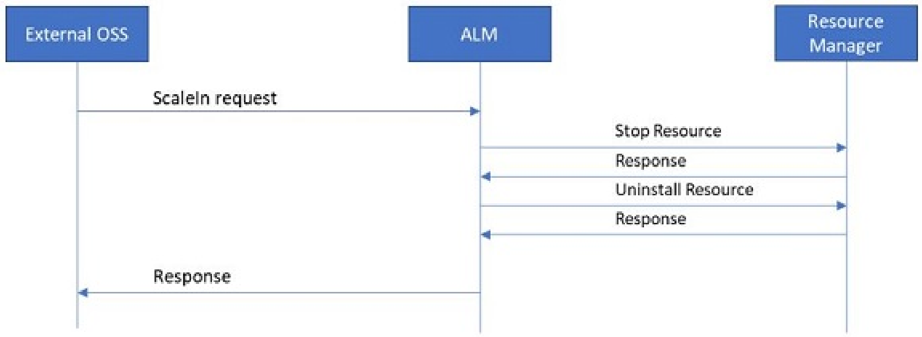
Scaling

Scale out



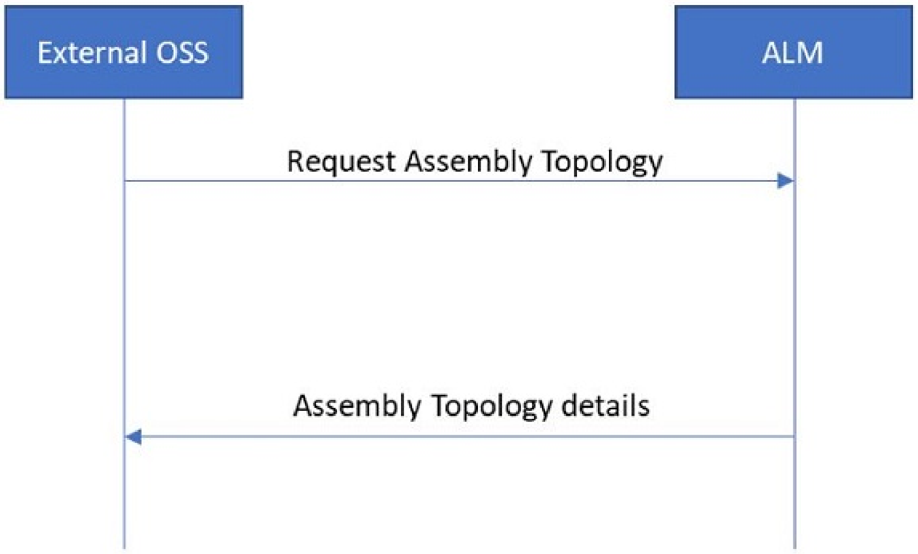
Note: Asynchronous events are not depicted.

Scale in



Note: Asynchronous events are not depicted.

Topology requests



Managing assemblies

This topic lists the assembly API calls, which are based on the state model for the lifecycle manager.

API definition conventions:

A table is associated with each example that explains the fields it contains, including the name of the field, a brief description, and whether the field is mandatory. Whether a field is required or not is based on the context of the examples. The underlying API definition may mark a field as optional, but in some contexts, the fields must be supplied.

Any field names in italics are examples only.

Standard response header

Each of the API calls returns a response header in the following format when the calls are successful.

Example response header

```
{
  "location": "http://192.168.99.100:8280/api/processes/9d63c16e-6685-4e7b-9123-81c196f99536",
  "date": "Fri, 08 Sep 2017 09:12:38 GMT",
  "server": "ALM Ishtar/1.1.0-SNAPSHOT",
  "transfer-encoding": "chunked",
  "x-application-context": "ishtar:prod,swagger:8280",
  "content-type": null
}
```

The 'location' URL in bold allows the caller to find out the state of the process that has been requested.

Process controller

This call allows the requestor of an intent to check the status of the associated process.

Request format

Aspect	Value
Endpoint URL	/api/processes/{id}
HTTP method	GET

The URL in the response header 'location' field is the URL that the requester must use to find the state of the process.

```
http://192.168.99.100:8280/api/processes/9d63c16e-6685-4e7b-9123-81c196f99536
```

Create assembly

Creates a new instance of an assembly based on the given descriptor and the properties.

Request format

Aspect	Value
Endpoint URL	/api/intent/createAssembly
HTTP method	POST

Example requests

Create assembly instance

```
{
  "assemblyName": "WED_102",
  "descriptorName": "assembly::t_single::1.0",
  "intendedState": "Active",
  "properties": {
      "data": "exampleValue",
      "deploymentLocation": "admin@local"
  }
}
```

Table 10. Fields to be used when creating a new assembly instance

Field	Description	Mandatory
assemblyName	The name of the assembly instance. Must be unique, must not start with a number, contain a space and must not contain double underscore character combination.	yes
descriptorName	The name of the assembly descriptor as defined at the head of the descriptor file	yes
intendedState	The state the assembly instance will be transitioned to once the assembly has been created. Allowed values are 'Installed', 'Inactive', 'Active'.	yes
properties	A set of tuples that match the required properties for the assembly instance. The list of actual properties is dependent upon the assembly descriptor.	yes (if there are required properties within the descriptor)

Response format

Aspect	Value
Return Code	201 CREATED

Change assembly state

Request format

Aspect	Value
Endpoint URL	/api/intent/changeAssemblyState
HTTP method	POST

Example requests

Change state of existing assembly using name

```
{
  "assemblyName": "WED_102",
  "intendedState": "Inactive"
}
```

Change state of existing assembly using ID

```
{
  "assemblyId": "1c3bd18a-05e9-4f49-b510-0e4785b2f0ae",
  "intendedState": "Inactive"
}
```

Request parameters

Table 11. Fields to be used when changing the state of an existing assembly

Field	Description	Mandatory
assemblyName	The name of the assembly instance. Must be unique, must not start with a number, contain a space and must not contain double underscore character combination.	yes (if assemblyId is not supplied)
assemblyId	The id of the assembly instance. This can be retrieved using the GetAssembly calls	yes (if assemblyName not supplied)
intendedState	The state which the assembly instance will be transitioned to once the assembly has been created. Allowed values are 'Installed', 'Inactive', 'Active'.	yes

Response format

Aspect	Value
Return Code	201 CREATED

Delete assembly

Requests to delete an assembly instance.

Request format

Aspect	Value
Endpoint URL	/api/intent/deleteAssembly
HTTP method	POST

Example requests

Delete assembly instance using name

```
{
  "assemblyName": "WED_102"
}
```

Delete assembly instance using ID

```
{
  "assemblyId": "1c3bd18a-05e9-4f49-b510-0e4785b2f0ae"
}
```

Request parameters

Table 12. Fields to be used when deleting an assembly

Field	Description	Mandatory
assemblyName	The name of the assembly instance. Must be unique, must not start with a number, contain a space and must not contain double underscore character combination.	yes (if assemblyId is not supplied)
assemblyId	The id of the assembly instance. This can be retrieved using the GetAssembly calls	yes (if assemblyName not supplied)

Response format

Aspect	Value
Return Code	201 CREATED

Heal components

This allows an assembly instances to be healed in the event that their resources are broken.

Request format

Aspect	Value
Endpoint URL	/api/intent/healAssembly
HTTP method	POST

Example requests

Heal assembly component using names

```
{
  "assemblyName": "WED_102",
  "brokenComponentName": "WED_102__t_single"
}
```

Heal assembly component using IDs

```
{
  "assemblyId": "5fd27c1e-403c-402b-a033-fef0940974d5",
  "brokenComponentId": "15a07604-377d-4fa2-955f-2a379560c24d"
}
```

Heal assembly component using a mixture of names and IDs

```
{
  "assemblyName": "WED_102",
  "brokenComponentId": "15a07604-377d-4fa2-955f-2a379560c24d"
}
```

Request parameters

Table 13. Fields to be used when healing a resource

Field	Description	Mandatory
assemblyName	The name of the assembly instance. Must be unique, must not start with a number, contain a space and must not contain double underscore character combination.	yes (if assemblyId is not supplied)
assemblyId	The id of the assembly instance. This can be retrieved using the GetAssembly calls	yes (if assemblyName not supplied)
brokenComponentId	This is the id of the component within the assembly instance. This can be found by using the GetAssembly calls.	yes (if brokenComponentName is not used)
brokenComponentName	The name of the component within the assembly instance. It can be found using the GetAssembly calls.	yes (if brokenComponentId is not used)

Table 13. Fields to be used when healing a resource (continued)

Field	Description	Mandatory
brokenComponentMetricKey	This is the ID of the component within the assembly instance (currently using the same ID as brokenComponentId). This can be found by using the GetAssembly calls.	Yes (if brokenComponentId or brokenComponentName is not used)

Response format

Aspect	Value
Return Code	201 CREATED

Scale components

This allows scalable components of an assembly to be scaled in or out.

Request format

Aspect	Value
Endpoint URL	/api/intent/scaleInAssembly
Endpoint URL	/api/intent/scaleOutAssembly
HTTP method	POST

Example request

Scale a cluster by name (depends on the endpoint as to whether it is 'In' or 'Out'

```
{
  "assemblyName": "WED_102",
  "clusterName": "storage_cluster"
}
```

Scale a cluster by ID (depends on the endpoint as to whether it is 'In' or 'Out'

```
{
  "assemblyId": "5fd27c1e-403c-402b-a033-fef0940974d5",
  "clusterName": "storage_cluster"
}
```

Request parameters

Table 14. Fields to be used when scaling components

Field	Description	Mandatory
assemblyName	The name of the assembly instance. Must be unique, must not start with a number, contain a space and must not contain double underscore character combination.	yes (if assemblyId is not supplied)
assemblyId	The id of the assembly instance. This can be retrieved using the GetAssembly calls	yes (if assemblyName not supplied)

Table 14. Fields to be used when scaling components (continued)

Field	Description	Mandatory
clusterName	The name of the cluster to be scaled. This is the name defined in the assembly descriptor for the cluster.	yes

Response format

Aspect	Value
Return Code	201 CREATED

Upgrade assembly

This upgrades an assembly, which means changing the descriptor that is associated with an assembly instance. This may cause the state of the assembly components to change while the upgrade is achieved.

Request format

Aspect	Value
Endpoint URL	/api/intent/upgradeAssembly
HTTP method	POST

Example request

Upgrade a cluster by name

```
{
  "assemblyName": "WED_102",
  "descriptorName": "assembly::t_single::2.0",
  "properties": {
      "data": "exampleValue",
      "deploymentLocation": "demo@local"
  }
}
```

Upgrade a cluster by ID

```
{
  "assemblyId": "5fd27c1e-403c-402b-a033-fef0940974d5",
  "descriptorName": "assembly::t_single::2.0",
  "properties": {
      "data": "exampleValue",
      "deploymentLocation": "demo@local"
  }
}
```

Request parameters

Table 15. Fields to be used when upgrading an assembly

Field	Description	Mandatory
assemblyName	The name of the assembly instance. Must be unique, must not start with a number, contain a space and must not contain double underscore character combination.	yes (if assemblyId is not supplied)
assemblyId	The id of the assembly instance. This can be retrieved using the GetAssembly calls.	yes (if assemblyName not supplied)

Table 15. Fields to be used when upgrading an assembly (continued)

Field	Description	Mandatory
descriptorName	The name of the assembly descriptor as defined at the head of the descriptor file.	yes
Properties	A set of tuples that match the required properties for the assembly instance. The list of actual properties is dependent upon the assembly descriptor.	Yes (depending upon descriptor requirements)

Response format

Aspect	Value
Return Code	201 CREATED

Resource managers

This topic describes the Resource Managers API specifications for the lifecycle management API. See the "Resource Manager API" on page 121 section for resource manager API specifications.

API definition conventions:

A table is associated with each example that explains the fields it contains, including the name of the field, a brief description, and whether the field is mandatory. Whether a field is required or not is based on the context of the examples. The underlying API definition may mark a field as optional, but in some contexts, the fields must be supplied.

Any field names in italics are examples only.

Create Resource Manager

Creates a record of a Resource Manager within Agile Lifecycle Manager and begins the onboarding process. When this request is placed Agile Lifecycle Manager will register the resource manager, and then it will request details of all the resource types that the resource manager is able to handle. This may take many seconds.

Request format

Aspect	Value
Endpoint URL	/api/resource-managers
HTTP method	POST

Example requests

```
{
  "name": "test",
  "type": "test-rm",
  "url": "http://localhost:8295/api/resource-manager"
}
```

Table 16. Create resource manager request fields

Field	Description	Mandatory
name	The name by which the resource manager instance is to be known by in the ALM	yes

Table 16. Create resource manager request fields (continued)

Field	Description	Mandatory
type	The type of resource manager that is being onboarded. This is a string supplied by those managing the resource managers. It is suggested that the same value be used for all resource managers that support the same set of resources	no
url	The URL where the resource manager interface can be found by the ALM	yes

Response format

Aspect	Value
Response Code	201 CREATED

Get Resource Manager

Gets the information about a Resource Manager within Agile Lifecycle Manager. The ID in the request is the unique name of the resource manager as defined by the 'name' field in Create Resource Manager.

Request format

Aspect	Value
Endpoint URL	/api/resource-managers/{id}
HTTP method	GET

Example response

```
{
  "name": "test",
  "type": "test-rm",
  "url": "http://localhost:8295/api/resource-manager"
}
```

Table 17. Get resource manager response fields

Field	Description	Mandatory
name	The name by which the resource manager instance is to be known by in the ALM	yes
type	The type of resource manager that is being onboarded. This is a string supplied by those managing the resource managers. It is suggested that the same value be used for all resource managers that support the same set of resources.	no
url	The URL where the resource manager interface can be found by the ALM.	yes

Response format

Aspect	Value
Response Code	200 OK

Update Resource Manager

Updates a record of a Resource Manager within the Agile Lifecycle Manager and begins the onboarding process.

Request format

Aspect	Value
Endpoint URL	/api/resource-managers/{id}
HTTP method	PUT

Example request

```
{
  "name": "test",
  "type": "test-rm",
  "url": "http://localhost:8295/api/resource-manager"
}
```

Table 18. Update resource manager request fields

Field	Description	Mandatory
name	The name by which the resource manager instance is to be known by in the ALM	yes
type	The type of resource manager that is being onboarded. This is a string supplied by those managing the resource managers. It is suggested that the same value be used for all resource managers that support the same set of resources.	no
url	The URL where the resource manager interface can be found by the ALM.	yes

Response format

Aspect	Value
Response code	200 OK

Delete Resource Manager

Deletes the record of a Resource Manager within the Agile Lifecycle Manager.

Request format

Aspect	Value
Endpoint URL	/api/resource-managers/{id}
HTTP method	DELETE

Response format

Aspect	Value
Response code	200 OK

Related concepts:

Chapter 5, “Getting started (using the APIs),” on page 47
Agile Lifecycle Manager provides both a graphical UI and an HTTP API allowing the creation and administration of assemblies. This section describes a set of basic scenarios to get started using the APIs.

Asynchronous state change events

Agile Lifecycle Manager will emit events when the state of an assembly or its components changes. Messages that are sent asynchronously are put onto a Kafka bus. The exact topics can be configured. These are emitted in response to Intent Requests causing the state of the Assembly Instance, or its associated components, to change. In the event of a failure to change state, an event will also be emitted.

API definition conventions:

A table is associated with each example that explains the fields it contains, including the name of the field, a brief description, and whether the field is mandatory. Whether a field is required or not is based on the context of the examples. The underlying API definition may mark a field as optional, but in some contexts, the fields must be supplied.

Any field names in italics are examples only.

ProcessStateChangeEvent

These events are associated with the process that performs the Intent request. These are sent out at the Start of the processing and when completed or failed.

The following example shows the first message sent when an intent has been received by Agile Lifecycle Manager. The processId will be used in all subsequent state change events. These happen when the process changes state.

Example of an initial state change event

```
{
   "processId": "e29b86a8-ca75-413b-921b-c4b895996c12",
   "assemblyId": "e4c198d1-2dbb-4557-baae-b5891fa258cf",
   "assemblyName": "example2",
   "assemblyDescriptorName": "assembly::t_single::1.0",
   "intentType": "CreateAssembly",
   "intent": {
      "assemblyName": "example2",
      "descriptorName": "assembly::t_single::1.0",
      "intendedState": "Active",
      "properties": {
         "data": "example data",
         "deplomentLocation": "admin@local"
      }
   },
   "processState": "In Progress",
   "processStartedAt": "2017-09-14T13:06:17.499Z",
   "eventId": "e08039e7-7efd-4b7e-86a3-9d39c48c2dd7",
   "eventCreatedAt": "2017-09-14T13:06:17.499Z",
   "eventType": "ProcessStateChangeEvent"
}
```

Example of the final message on successful completion of the Intent

```
{
   "processId": "e29b86a8-ca75-413b-921b-c4b895996c12",
   "assemblyId": "e4c198d1-2dbb-4557-baae-b5891fa258cf",
   "assemblyName": "example2",
   "assemblyDescriptorName": "assembly::t_single::1.0",
   "intentType": "CreateAssembly",
   "intent": {
      "assemblyName": "example2",
      "descriptorName": "assembly::t_single::1.0",
      "intendedState": "Active",
      "properties": {
         "data": "example data",
         "deplomentLocation": "admin@local"
      }
   },
   "processState": "Completed",
   "processStartedAt": "2017-09-14T13:06:17.499Z",
   "processFinishedAt": "2017-09-14T13:06:19.648Z",
   "eventId": "405cf14c-752e-4030-8a4e-6706e04b50f5",
   "eventCreatedAt": "2017-09-14T13:06:19.649Z",
   "eventType": "ProcessStateChangeEvent"
}
```

Example of the final message indicating Intent Failed

```
     {
        "processId": "8c922ec6-7589-4ba8-8b4e-d7841b9a9654",
        "assemblyId": "42ecbe49-0069-41f5-ac38-595b090c3d65",
        "assemblyName": "example3",
        "assemblyDescriptorName": "assembly::t_single::1.0",
        "intentType": "CreateAssembly",
        "intent": {
           "assemblyName": "example3",
           "descriptorName": "assembly::t_single::1.0",
           "intendedState": "Active",
           "properties": {
              "data": "Example Data",
              "deplomentLocation": "admin@local"
           }
        },
        "processState": "Failed",
        "processStateReason": "Exception ...",
        "processStartedAt": "2017-09-14T13:13:58.966Z",
        "processFinishedAt": "2017-09-14T13:13:59.616Z",
        "eventId": "11acf385-e6f9-41cf-9651-38dc3ed6a53a",
        "eventCreatedAt": "2017-09-14T13:13:59.616Z",
        "eventType": "ProcessStateChangeEvent"
     }
```

Table 19. ProcessStateChangeEvent fields

Field	Description	Mandatory
processId	The ID given to the process that was initiated by an Intent request	yes
assemblyId	The Agile Lifecycle Manager internal ID for the assembly instance associated with the process	yes
assemblyName	The name of the assembly instance as supplied in the Intent request	yes
intentType	The name of the intent type. The values correspond to the Intents described in the Managing Assembly section,	yes
intent	Contains details of the intent request supplied	yes
processState	The processState may contain 'In Progress', 'Completed' or 'Failed'	yes
processStartedAt	The data and time the process was started	yes

Table 19. ProcessStateChangeEvent fields (continued)

Field	Description	Mandatory
eventId	The ID of this event from the Agile Lifecycle Manager point of view (each message will have a unique ID)	yes
eventCreatedAt	The date and time when the event happened from the Agile Lifecycle Manager point of view	yes
eventType	Will always contain 'ProcessStateChangeEvent'	yes

ComponentStateChangeEvent

These events are sent when the root assembly changes state and when each of its associated resources successfully transitions to a new state. In the event of a failure of the process no events will be sent.

The following example shows a resource transitioning to the Installed state. The previous state is null indicating the resource did not exists before.

First message sent indicating first component transitioning to the Installed State

```
{
   "eventId": "901d4794-7734-4511-8e24-6035ee5cb22a",
   "eventCreatedAt": "2017-09-14T13:06:18.37Z",
   "rootAssemblyId": "e4c198d1-2dbb-4557-baae-b5891fa258cf",
   "rootAssemblyName": "example2",
   "resourceId": "7a03bc63-bccf-4731-b6b2-9389609d9fa5",
   "resourceName": "example2__A",
   "resourceManager": "test-rm",
   "deploymentLocation": "admin@local",
   "externalId": "06d20929-a1c0-44cf-8009-aee47bac3f99",
   "previousState": null,
   "newState": "Installed",
   "eventType": "ComponentStateChangeEvent"
}
```

An event indicating the root assembly has transitions to Installed State

The following example will be sent when all resources associated with the root assembly have successfully transitioned to the Installed State.

```
{
   "eventId": "33a63fca-2bed-49c3-8615-56e5b35aa3bb",
   "eventCreatedAt": "2017-09-14T13:06:18.572Z",
   "rootAssemblyId": "e4c198d1-2dbb-4557-baae-b5891fa258cf",
   "rootAssemblyName": "example2",
   "previousState": null,
   "newState": "Installed",
   "eventType": "ComponentStateChangeEvent"
}
```

Table 20. ComponentStateChangeEvent fields

Field	Description	Mandatory
eventType	The expected value is 'ComponentStateChangeEvent'	yes
eventId	The internal id generated by Agile Lifecycle Manager in response to an orchestration event request	yes

Table 20. ComponentStateChangeEvent fields (continued)

Field	Description	Mandatory
eventCreatedAt	The date and time that the event took place as recorded by Agile Lifecycle Manager	yes
rootAssemblyId	The internal Agile Lifecycle Manager ID for the assembly instance associated with the event	yes
rootAssemblyName	The name of the root assembly as supplied in the Intent request	yes
resourceId	Th ID of the resource defined by Agile Lifecycle Manager	no (used for resources only)
resourceName	The name of the resource defined by Agile Lifecycle Manager	no (used for resources only)
resourceManager	The name of the resource manager that manages the resource	no (used for resources only)
deploymentLocation	The location that the resource manager was requested to install the resource	no (used for resources only)
externalId	The ID of the resource as defined by the resource manager	no (used for resources only)
previousState	The state that the assembly or component was in before the state change happened. Allowed values: Installed, Inactive, Active. When a Heal event has been requested Agile Lifecycle Manager puts the component into the Broken state. This is a temporary state that is used to trigger the Heal processing. This will be set to 'null' when the resource or assembly is transitioning to the Installed State.	yes
newState	The state to which the assembly or component instance transitioned in the event of a successful state change, or the state that would have resulted if a failure had not occurred. This will be 'null' when the resource or assembly is being uninstalled.	yes

Regarding the previousState and newState fields: When Agile Lifecycle Manager is requested to heal a component, it will indicate this with a set of state transitions from Active to Broken, and Broken to Inactive.

Related tasks:

Configuring Agile Lifecycle Manager
To configure Agile Lifecycle Manager for use, you override the default application properties, and then configure external instances of Kafka, Cassandra and Elasticsearch. You can also add new OpenLDAP users, modify an OpenLDAP user password, and add new API Clients if required.

Related reference:

"Functionality" on page 6
Agile Lifecycle Manager provides continuous integration and deployment of resources, intent-driven operations to automate lifecycle processes, and an open

framework.

"Configuration reference" on page 47
This topic provides you with an overview of the Agile Lifecycle Manager services settings you need to know when configuring the solution for your own environment, such as port numbers, Swagger URLs, and API details.

Resource health events

Resource health events include integrity and load metric events.

API definition conventions:

A table is associated with each example that explains the fields it contains, including the name of the field, a brief description, and whether the field is mandatory. Whether a field is required or not is based on the context of the examples. The underlying API definition may mark a field as optional, but in some contexts, the fields must be supplied.

Any field names in italics are examples only.

Integrity events

Integrity events are sent to enable a resource to indicate whether it is working or broken.

Example integrity metric events

```
{
  "metricKey" : "142971c5-a84b-4d34-af15-435ba8640aec",
  "metricName" : "h_integrity",
  "integrity" : "OK",
  "message" : "Everything is working"
}
```

Table 21. Integrity event fields

Field	Description	Mandatory
metricKey	The key given to the resource manager when the resource was created as a token to be used within these messages	yes
metricName	The name of the metric as defined in the resource descriptor	yes
integrity	A value indicating if the resource associated with the metric Key is working. Allowed values are 'OK' for working and 'BROKEN' when healing is required.	yes
message	An optional test string to include information about the integrity of the resource. For example, it can include an error code.	no

Load events

Load events indicate a resources load. This may be an aggregation across many resources as seen for example by a load balancer.

Example load metric events

```
{
  "metricKey" : "818127b3-1904-4737-a60c-8c7bab73532d",
  "metricName" : "h_load",
  "load" : 76,
  "message" : "Load is high"
}
```

Table 22. Load event fields

Field	Description	Mandatory
metricKey	The key given to the resource manager when the resource was created as a token to be used within these messages	yes
metricName	The name of the metric as defined in the resource descriptor	yes
load	A value between 0 and 100, indicating the load on the resources	yes
message	An optional test string to include information about the integrity of the resource. For example, it can include an error code.	no

Topology

This topic lists the topology API specifications.

API definition conventions:

A table is associated with each example that explains the fields it contains, including the name of the field, a brief description, and whether the field is mandatory. Whether a field is required or not is based on the context of the examples. The underlying API definition may mark a field as optional, but in some contexts, the fields must be supplied.

Any field names in italics are examples only.

Get assembly by id

Gets the assembly with the given ID.

Request format

Aspect	Value
Endpoint URL	/api/topology/assemblies/{assemblyId}
HTTP method	GET

Request parameters

Field	Description	Mandatory
id	The internal id of the assembly	yes
numEvents	Number of historical events to show in response. If numEvents is set to `0`, only the structure of assembly is shown. If left blank, the default value 3 is used.	no

Response format

Aspect	Value
Response Code	200 Ok

Example response

```
{
  "type": "Assembly",
  "id": "bf649336-c8c5-49d9-9f4e-60567fe54135",
  "name": "test_1",
  "state": "Active",
  "descriptorName": "assembly::t_bta::1.0",
  "properties": [
    {
      "name": "data",
      "value": "data"
    },
    ...
  ],
  "createdAt": "2017-08-02T22:28:41.906+0000",
  "lastModifiedAt": "2017-08-02T22:47:46.189+0000",
  "children": [
    {
      "type": "Component",
      "id": "aa56626d-cfec-410b-afb7-7160019bdff0",
      "name": "test_1__A",
      ...
  ],
  "relationships": [
    {
      "name": "third-relationship__1",
      "sourceId": "aa56626d-cfec-410b-afb7-7160019bdff0",
      "targetId": "9c525d0c-18d4-404f-a5b2-8a55480660a8",
      "properties": [
        {
          "name": "source",
          "value": "test_1__A"
        }
      ]
    }
  ],
  "references": [
    {
      "id": "1c269f9d-fcca-4754-946c-6f3e6179bf38",
      "name": "internal-network",
      "type": "resource::openstack_neutron_network::1.0",
      ...
    },
     ...
  ]
}
```

Table 23. Get assembly by id fields

Field	Description	Mandatory
type	The type of entity being returned (always 'assembly')	yes
id	The internal id of the assembly	yes
name	The name of the assembly as provided by the external system	yes

Table 23. Get assembly by id fields (continued)

Field	Description	Mandatory
state	The state of the assembly. Allowed values: Installed, Inactive, Active. This field may be missing if the assembly has not reached the Installed state	no
descriptorName	The name of the assembly descriptor associated with the assembly instance	yes
properties	A collection of assembly level properties. Each property will have a name and value field.	yes
createdAt	The date and time the assembly was created	yes
lastModifiedAt	The date and time the assembly was last modified	no
children	A collection of components that make up the assembly. When the component is of the type 'Assembly' the contents are the same as for the top level assembly. When the type is 'component' the entry is in fact a resource. This will have a type, name and id and a set of associated properties.	yes
relationships	A collection of relationships associated with the assembly instance. Each relationship has a name and the id of the source and target components involved in the relationship. Relationships also have a property section.	no
references	A collection of references used by the assembly. References can be provided to resources by resource managers, but cannot be created using any assembly and other existing assembly instances.	no

Get assembly by name

Gets the assembly with the given name.

Request format

Aspect	Value
Endpoint URL	/api/topology/assemblies?name={name}
HTTP method	GET

Request parameters

Field	Description	Mandatory
name	The name of the assembly	yes
numEvents	Number of historical events to show in response. If numEvents is set to 0, only the structure of assembly is shown.	no

Response format

Aspect	Value
Response Code	200 Ok

Example response

```
{
  "type": "Assembly",
  "id": "bf649336-c8c5-49d9-9f4e-60567fe54135",
  "name": "test_1",
  "state": "Active",
  "descriptorName": "assembly::t_bta::1.0",
  "properties": [
    {
      "name": "data",
      "value": "data"
    },
    ...
  ],
  "createdAt": "2017-08-02T22:28:41.906+0000",
  "lastModifiedAt": "2017-08-02T22:47:46.189+0000",
  "children": [
    {
      "type": "Component",
      "id": "aa56626d-cfec-410b-afb7-7160019bdff0",
      "name": "test_1__A",
      ...
  ],
  "relationships": [
    {
      "name": "third-relationship__1",
      "sourceId": "aa56626d-cfec-410b-afb7-7160019bdff0",
      "targetId": "9c525d0c-18d4-404f-a5b2-8a55480660a8",
      "properties": [
        {
          "name": "source",
          "value": "test_1__A"
        }
      ]
    }
  ],
  "references": [
    {
      "id": "1c269f9d-fcca-4754-946c-6f3e6179bf38",
      "name": "internal-network",
      "type": "resource::openstack_neutron_network::1.0",
      ...
    },
     ...
  ]
}
```

Table 24. Get assembly by name topology fields

Field	Description	Mandatory
type	The type of entity being returned (always 'assembly')	yes
id	The internal id of the assembly	yes
name	The name of the assembly as provided by the external system	yes
state	The state of the assembly. Allowed values: Installed, Inactive, Active. This field may be missing if the assembly has not reached the Installed state.	no

Table 24. Get assembly by name topology fields (continued)

Field	Description	Mandatory
descriptorName	The name of the assembly descriptor associated with the assembly instance	yes
properties	A collection of assembly level properties. Each property will have a name and value field.	yes
createdAt	The date and time the assembly was created	yes
lastModifiedAt	The date and time the assembly was last modified	no
children	A collection of components that make up the assembly. When the component is of the type 'Assembly' the contents are the same as for the top level assembly. When the type is 'component' the entry is in fact a resource. This will have a type, name and id and a set of associated properties.	yes
relationships	A collection of relationships associated with the assembly instance. Each relationship has a name and the id of the source and target components involved in the relationship. Relationships also have a property section.	no
references	A collection of references used by the assembly. References can be provided to resources by resource managers, but cannot be created using any assembly and other existing assembly instances.	no

Catalog API

Use the following catalog API details to manage the descriptors in the catalog. You can add, list, update or delete assembly descriptors from the catalog. You can list or delete resource descriptors, but can only add or update them through the resource manager API.

API definition conventions:

A table is associated with each example that explains the fields it contains, including the name of the field, a brief description, and whether the field is mandatory. Whether a field is required or not is based on the context of the examples. The underlying API definition may mark a field as optional, but in some contexts, the fields must be supplied.

Any field names in italics are examples only.

Get a summary of all descriptors

The following request returns a summary of the descriptors from the Agile Lifecycle Manager catalog.

Request format

Aspect	Value
Endpoint URL	/api/catalog/descriptors
HTTP method	GET

Response format

Aspect	Value
Response Code	200 Ok

Example response

```
[
  {
    "name": "resource::t_simple::1.0",
    "description": "resource for  t_simple",
    "links": [
      {
        "rel": "self",
        "href": "http://192.168.99.100:8280/api/ /catalog/descriptors/
resource::t_simple::1.0"
      }
    ]
  },
  {
    "name": "resource::h_simple::1.0",
    "description": "resource for  t_simple",
    "links": [
      {
        "rel": "self",
        "href": "http://192.168.99.100:8280 /api/catalog/descriptors/
resource::h_simple::1.0"
      }
    ]
  }
]
```

Table 25. Response properties

Field	Description	Mandatory
name	The name of the descriptor	yes
description	The descriptor description	yes
links	A collection of links to the descriptor	yes
rel	Will always be set to self	yes
href	The URL to retrieve the descriptor from the catalog	yes

Create assembly descriptor

The following request creates a new assembly descriptor in the Agile Lifecycle Manager catalog.

Request format

Aspect	Value
Endpoint URL	/api/catalog/descriptors
Content type	application/yaml
HTTP method	POST

Example request

The content of the request will be an assembly descriptor in YAML format. For more information see "Assembly descriptor YAML specifications" on page 147.

Response format

Aspect	Value
Response Code	201 Ok

Example response

```
{
  "validationWarnings": []
}
```

Table 26. Response properties

Field	Description	Mandatory
validationWarnings	Will contain a list of warnings about the descriptor that has been created, if empty the descriptor is valid.	yes

Delete assembly descriptor

The following request deletes an assembly descriptor from the Agile Lifecycle Manager catalog.

Request format

Aspect	Value
Endpoint URL	/api/catalog/descriptors/{descriptorName}
HTTP method	DELETE

The descriptor name is the full name of the descriptor, for example `assembly::t_single::1.0`

You must encode this appropriately for use as a URL, for example `assembly%3A%3At_single%3A%3A1.0`

Response format

Aspect	Value
Response Code	204 Ok

Get assembly descriptor by name

The following request returns an existing assembly descriptor from the Agile Lifecycle Manager catalog.

Request format

Aspect	Value
Endpoint URL	/api/catalog/descriptors/{descriptorName}
HTTP method	GET

The descriptor name is the full name of the descriptor, for example `assembly::t_single::1.0`

You must encode this appropriately for use as a URL, for example `assembly%3A%3At_single%3A%3A1.0`

Response format

Aspect	Value
Response Code	201 Ok

The response body will contain the descriptor in YAML format.

Update assembly descriptor

The following request updates an existing assembly descriptor in the Agile Lifecycle Manager catalog.

Request format

Aspect	Value
Endpoint URL	/api/catalog/descriptors/{descriptorName}
Content type	application/yaml
HTTP method	PUT

The descriptor name is the full name of the descriptor, for example `assembly::t_single::1.0`

You must encode this appropriately for use as a URL, for example `assembly%3A%3At_single%3A%3A1.0`

Example request

The content of the request will be an assembly descriptor in YAML format. For more information see "Assembly descriptor YAML specifications" on page 147.

Response format

Aspect	Value
Response Code	200 Ok

Example response

```
{
  "validationWarnings": []
}
```

Table 27. Response properties

Field	Description	Mandatory
validationWarnings	Will contain a list of warnings about the descriptor that has been created, if empty the descriptor is valid.	yes

Resource Manager API

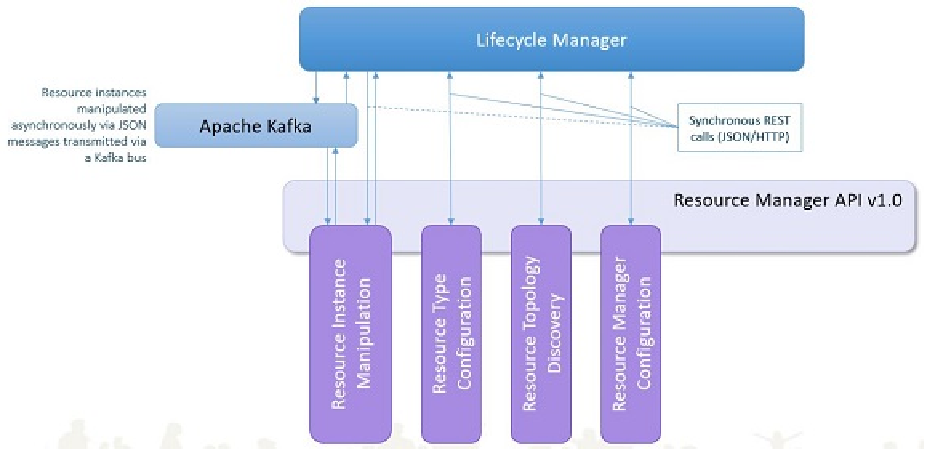
The Resource Manager API is responsible for defining the interactions between a lifecycle manager and the resource managers that are used to manage resources within virtual (or physical) infrastructures.

Interface architecture

Four groups of interface endpoints are referenced within the Resource Manager API. These groups are responsible for the management or retrieval of different entities from the resource managers.

High-level interface architecture

The following diagram shows the high-level functional architecture referenced in the Resource Manager API specifications.


Lifecycle Manager
Resource instances manipulated asynchronously via JSON messages transmitted via a Kafka bus
Apache Kafka
Synchronous REST calls (JSON/HTTP)
Resource Manager API v1.0
Resource Instance Manipulation
Resource Type Configuration
Resource Topology Discovery
Resource Manager Configuration

API interaction principles

All of the message descriptions are in JSON format and should be submitted with the HTTP content-type header of `application/json`.

The REST endpoints can be secured using HTTPS, but there is no current provision for further authentication across the interface. Future changes could add support for HTTP basic authentication or the use of tokens (such as JWT, OAuth, etc).

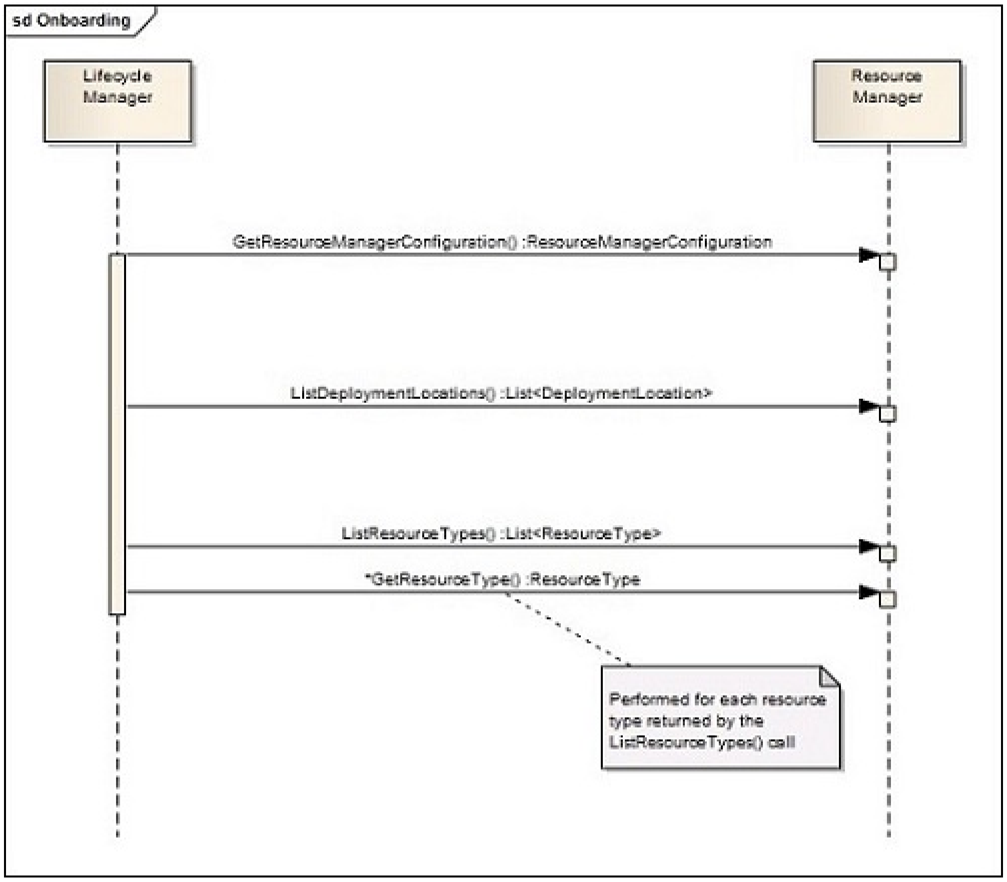
All of the REST endpoints should be accessible from the same root URL, for example http://localhost:8080/api/configuration and http://localhost:8080/api/types

Interface interaction patterns

This topic outlines the potential interaction patterns with how Agile Lifecycle Manager calls the Resource Manager APIs.

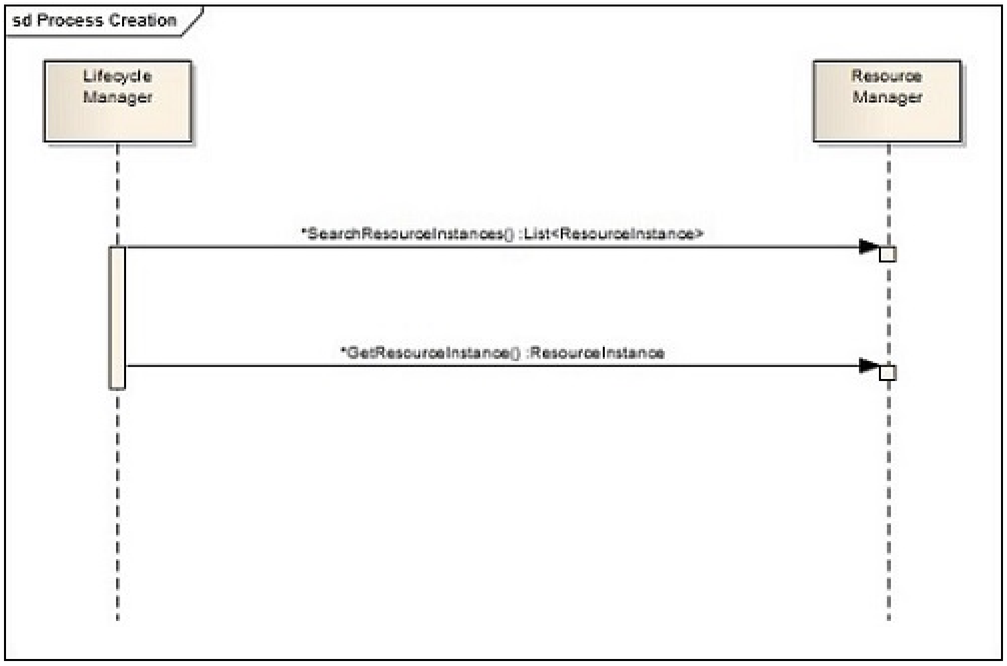
Resource Manager onboarding

When a new Resource Manager is being onboarded into the Agile Lifecycle Manager, the following calls will be made.


sd Onboarding
Lifecycle Manager
Resource Manager
GetResourceManagerConfiguration() :ResourceManagerConfiguration
ListDeploymentLocations() :List<DeploymentLocation>
ListResourceTypes() :List<ResourceType>
*GetResourceType() :ResourceType
Performed for each resource type returned by the ListResourceTypes() call

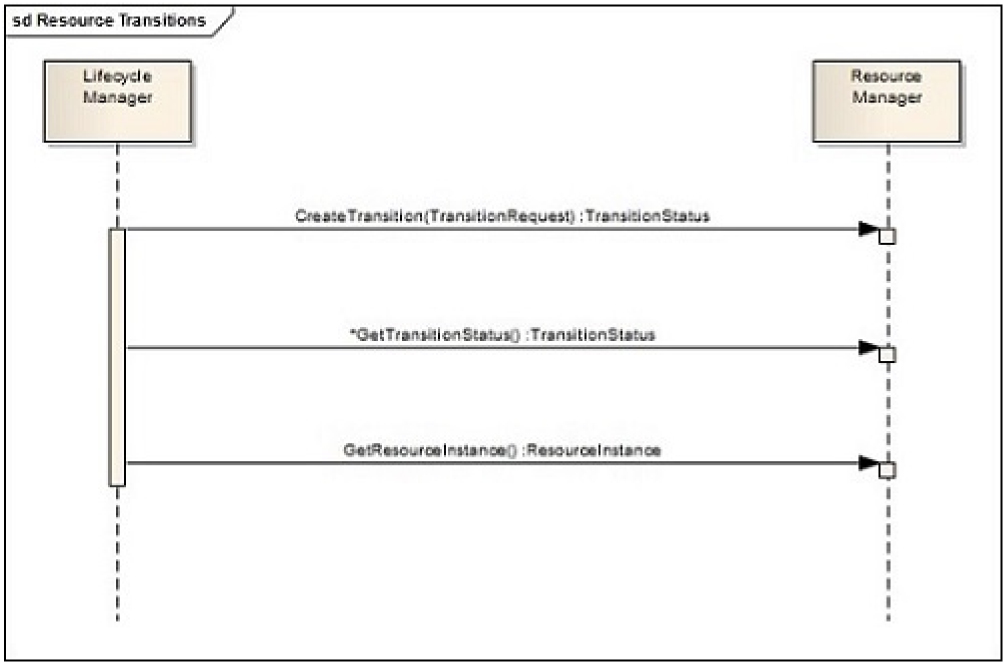
Process creation

When a new transition process is being created within the Agile Lifecycle Manager the following calls may be made to a Resource Manager.


sd Process Creation
Lifecycle Manager
Resource Manager
*SearchResourceInstances() :List<ResourceInstance>
*GetResourceInstance() :ResourceInstance

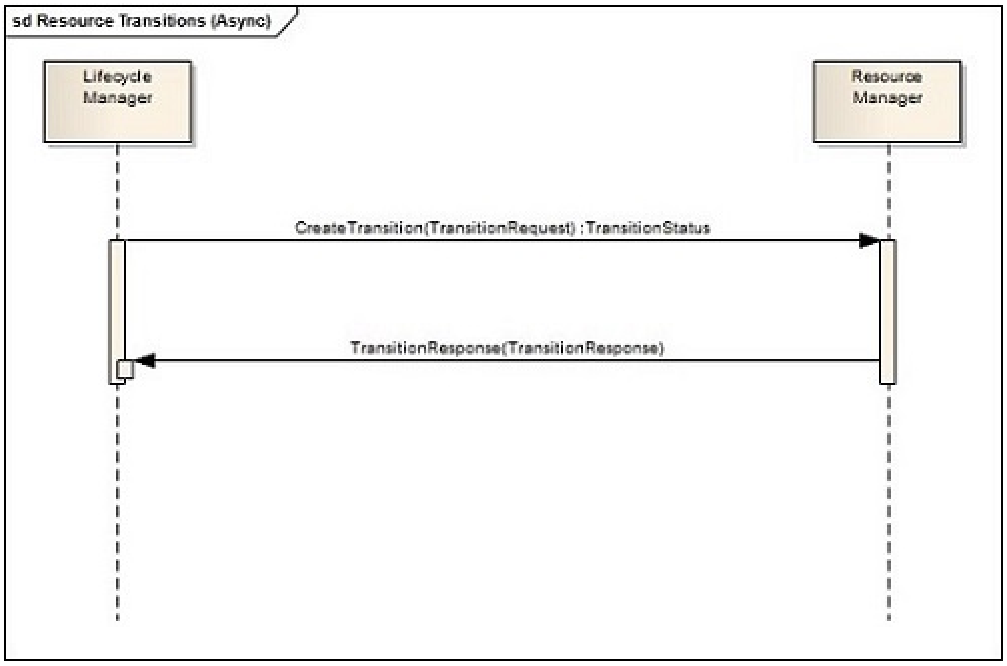
Resource transitions (synchronous)

As part of an assembly transition, the Agile Lifecycle Manager may call out to a Resource Manager to transition or perform an operation on a Resource. This is the interaction pattern for this scenario when the Resource Manager does not support asynchronous response messages.


sd Resource Transitions
Lifecycle Manager
Resource Manager
CreateTransition(TransitionRequest) :TransitionStatus
*GetTransitionStatus() :TransitionStatus
GetResourceInstance() :ResourceInstance

Resource transitions (asynchronous)

As part of an assembly transition, the Agile Lifecycle Manager may call out to a Resource Manager to transition or perform an operation on a Resource. This is the interaction pattern for this scenario when the Resource Manager supports asynchronous response messages.


sd Resource Transitions (Async)
Lifecycle Manager
Resource Manager
CreateTransition(TransitionRequest) :TransitionStatus
TransitionResponse(TransitionResponse)

Resource manager configuration

This topic describes the Resource Managers configuration.

Get Resource Manager configuration

Returns high-level information about the configuration of this Resource Manager.

This endpoint is called when onboarding the Resource Manager.

The supportedFeatures section allows the resource manager to inform the ALM that a feature is supported by the resource manager. The only value supported at the moment is AsynchronousTransitionResponses. This informs ALM that the resource manager will be using the asynchronous response mechanism described here.

Request format

Aspect	Value
Endpoint URL	/configuration
HTTP method	GET
Parameters	none

Response format

Aspect	Value
Return Code	200 OK

Example response

```
{
 "name": "default-rm::dev",
 "version": "1.0.0",
 "supportedApiVersions": [ "1.0" ],
 "supportedFeatures": {
```

```
"AsynchronousTransitionResponses": "false"
},
"properties": {
 "responseKafkaConnectionUrl": "zookeeper://localhost:2181",
 "responseKafkaTopicName": "lm-responses"
}

}
```

Table 28. Get Resource Manager fields

Field	Description	Mandatory
name	The name of the resource manager instance	yes
version	The version of the resource manager instance	yes
supportedApiVersions	A list API version supported – currently on 1.0	no
supportedFeatures	A list of features supported by the resource manager	no
AsynchronousTransition Responses	Indicates if the resource manager supports asynchronous responses via Kafka	no
properties	A set of key value pair with properties describing key behaviour	no
responseKafkaConnectionUrl	The URL for Kafka where the asynchronous responses will be sent	no
responseKafkaTopicName	The name of the topic for the asynchronous responses	no

Resource type configuration

This topic describes the resource type configuration.

List resource types

Returns a list of all resource types managed by this Resource Manager.

Note: The descriptor is not returned in this list.
Allowable states for the resource types are:

- UNPUBLISHED
- PUBLISHED
- DELETED

Request format

Aspect	Value
Endpoint URL	/types
HTTP method	GET
Parameters	none

Response format

Aspect	Value
Return Code	200 OK

Example response

```
[
{
  "name": "resource::openstack-network::1.0",
  "state": "PUBLISHED",
  "createdAt": "2017-05-01T11:22:33Z",
  "lastModifiedAt": "2017-05-04T12:13:14+01:00"
}
]
```

Table 29. List resource types fields

Field	Description	Mandatory
name	The name of the resource type. Must follow the naming structure defined in the Assembly Specification document.	yes
state	The state of the resource descriptor. Currently only PUBLISHED is allowed.	yes
createdAt	The date the resource type was created. XML Date time format.	yes
lastModifiedAt	The date the resource type was last changed. XML Date time format.	yes

Get resource type

Returns information about a specific resource type, including its YAML descriptor.

Request format

Aspect	Value
Endpoint URL	/types/{name}
HTTP method	GET
Parameters	name - Unique name for the resource type requested

Response format

Aspect	Value
Return Code	200 OK

Example response

```
{
 "name": "resource::openstack-network::1.0",
 "descriptor": "YAML Descriptor for this resource type",
 "state": "PUBLISHED",
 "createdAt": "2017-05-01T11:22:33Z",
 "lastModifiedAt": "2017-05-04T12:13:14+01:00"
}
```

Table 30. Get resource type fields

Field	Description	Mandatory
name	The name of the resource type. Must follow the naming structure defined in the Assembly Specification document.	yes
descriptor	The resource descriptor. A valid YAML document as a string.	yes
state	The state of the resource descriptor. Currently only PUBLISHED is allowed.	yes
createdAt	The date the resource type was created. XML Date time format.	yes
lastModifiedAt	The date the resource type was last changed. XML Date time format.	yes

Resource topology

This topic describes the resource topology.

List deployment locations

Returns a list of all deployment locations available to this Resource Manager.

Request format

Aspect	Value
Endpoint URL	/topology/deployment-locations
HTTP method	GET
Parameters	none

Response format

Aspect	Value
Return Code	200 OK

Example response

```
[
{
  "name": "dev-cloud",
  "type": "default-rm::Cloud"
},
{
  "name": "test-cloud",
  "type": "default-rm::Cloud"
}
]
```

Table 31. List deployment location fields

Field	Description	Mandatory
name	The name of the location managed by the resource manager.	yes

Table 31. List deployment location fields (continued)

Field	Description	Mandatory
type	The type of location Defined by the resource manager Any valid string	yes

Get deployment location

Returns information for the specified deployment location.

Request format

Aspect	Value
Endpoint URL	/topology/deployment-locations/{name}
HTTP method	GET
Parameters	name - Unique name for the deployment location requested

Response format

Aspect	Value
Return Code	200 OK

Example response

```
{
 "name": "dev-cloud",
 "type": "default-rm::Cloud"
}
```

Table 32. Get deployment location fields

Field	Description	Mandatory
name	The name of the location managed by the resource manager.	yes
type	The type of location Defined by the resource manager Any valid string	yes

Search for resource instances

Searches for resource instances managed within the specified deployment location.

The search can be restricted by the type of the resources to be returned, or a partial match on the name of the resources.

Request format

Aspect	Value
Endpoint URL	/topology/deployment-locations/{name}
HTTP method	GET

Aspect	Value
Parameters	name - Unique name for the deployment location instanceType - Limits results to be of this resource type (optional, exact matches only) instanceName - Limits results to contain this string in the name (optional, partial matching)

Response format

Aspect	Value
Return Code	200 OK

Example response

```
[
{
  "resourceId": "c675e0bd-9c6c-43ca-84bf-2c061d439c6b",
  "resourceName": "dev-network",
  "resourceType": "resource::openstack-network::1.0",
  "resourceManagerId": "default-rm::dev",
  "deploymentLocation": "dev-cloud",
  "properties": {
   "propertyName": "propertyValue"
},
  "createdAt": "2017-05-01T12:00:00Z",
  "lastModifiedAt": "2017-05-01T12:00:00Z"
}
]
```

Table 33. Search for resource instances fields

Field	Description	Mandatory
resourceId	The id of the instance of a resource	yes
resourceName	The name of the resource	yes
resourceType	The name of the resource type	yes
resourceManagerId	The id of the resource manager instance This ID is the same attribute as 'name', which is returned in the get configuration request response	yes
deploymentLocation	The name of the deployment location where the instance exists	yes
properties	A set of key value pair with properties describing key behaviour	no
propertyName	A name of a property to be used in the search	no
propertyValue	The value associated with the propertyName	no
createdAt	The date the resource type was created. XML Date time format	yes

Table 33. Search for resource instances fields (continued)

Field	Description	Mandatory
lastModifiedAt	The date the resource type was last changed XML Date time format	yes

Get resource instance

Returns information for the specified resource instance.

When Agile Lifecycle Manager requests a resource manager to create a resource, the resource manager may instantiate a number of underlying virtual resources as part of implementing the resource. In the response the resource manager is expected to return details of any underlying virtual resources that have been instantiated. This should be put in the internalResourceInstances section of the response.

Request format

Aspect	Value
Endpoint URL	/topology/instances/{id}
HTTP method	GET
Parameters	id - Unique id for the resource instance

Response format

Aspect	Value
Return Code	200 OK

Example response

```
{
 "resourceId": "default-rm://c675e0bd-9c6c-43ca-84bf-2c061d439c6b",
 "resourceName": "dev-network",
 "resourceType": "resource::openstack-network::1.0",
 "resourceManagerId": "default-rm::dev",
 "deploymentLocation": "dev-cloud",
 "properties": {
  "propertyName": "propertyValue"
},
 "createdAt": "2017-05-01T12:00:00Z",
 "lastModifiedAt": "2017-05-01T12:00:00Z",
 "internalResourceInstances": [
{
   "id": "3cb7822b-fc44-46ab-8072-9c65cd778d1f",
   "name": "MGMT-NETWORK",
   "type": "OpenDaylight::PrivateNetwork"
  }
]
}
```

Table 34. Get resource instance fields

Field	Description	Mandatory
resourceId	The id of the instance of a resource	yes
resourceName	The name of the resource	yes

Table 34. Get resource instance fields (continued)

Field	Description	Mandatory
resourceType	The name of the resource type	yes
resourceManagerId	The ID of the resource manager instance	yes
deploymentLocation	The name of the deployment location where the instance exists	yes
properties	A set of key value pair with properties describing key behavior	no
propertyName	A name of a property to be used in the search	no
propertyValue	The value associated with the propertyName	no
createdAt	The date the resource type was created. XML Date time format	yes
lastModifiedAt	The date the resource type was last changed XML Date time format	yes
internalResourceInstances	A list of resources that have been created by the resource manager in response to the request	no
Id	The id of the internal resource	no
Name	The name of the internal resource	no
Type	The type of the internal resource	

Resource lifecycle management

This topic describes the resource lifecycle management.

Create resource transition

Requests this Resource Manager performs a specific transition against a resource.

Request format

Aspect	Value
Endpoint URL	/lifecycle/transitions
HTTP method	POST
Parameters	none

Example request

```
{
 "resourceManagerId": "default-rm::dev",
 "deploymentLocation": "dev-cloud",
 "resourceType": "resource::openstack-network::1.0",
 "transitionName": "Install",
 "resourceName": "dev-network-c675e0bd",
 "metricKey" : "818127b3-1904-4737-a60c-8c7bab73532d"

 "properties": {
  "propertyName": "propertyValue"
},
"context": {}
}
```

Table 35. Create resource transition request fields

Field	Description	Mandatory
resourceManagerId	The id of the resource manager instance	yes
deploymentLocation	The name of the deployment location where the resource will be created	yes
resourceType	The name of the resource type to be created	yes
transitionName	The name of the Transition to be enacted against the resource Allowed values for the transitionName are Install, Configure, Start, Integrity, Stop, Uninstall, as well as any operation names supported by the resources	yes
resourceId	The unique id of the resource. This field is mandatory for all **non-Install** transitions and will not be present during 'Install' transitions, as this is allocated by the resource manager once it has been created.	yes
resourceName	The name of the resource.	yes
metric key	A key provided to manage metrics.	yes
properties	A section that contains a set of key/value pairs for the properties for the resource These will match those defined in the resource descriptor	yes
context	Context is included for future use	no

Response format

Aspect	Value
Return Code	202 ACCEPTED

Example response

```
{
 "requestId": "80fc4a66-7e92-41f8-b4bb-7cb98193f5fa",
 "requestState": "PENDING",
 "context": {
  "AsynchronousTransitionResponses": "false"
}
}
```

Table 36. Create resource transition response fields

Field	Description	Mandatory
requestId	The id of the request defined by the resource manager This should be a GUID	yes

Table 36. Create resource transition response fields (continued)

Field	Description	Mandatory
requestState	A string representing the state of the request Allowable states for the request state are: • PENDING • IN_PROGRESS • COMPLETED • CANCELLED • FAILED	yes
context	Context is included for future use	no
AsynchronousTransition Responses	Indicates whether this transition will send responses asynchronously via Kafka	no

Get resource transition status

Returns information about the specified transition or operation request. The id passed in is the value that the resource manager generated as the requestId in the response of the previous call.

Request format

Aspect	Value
Endpoint URL	/lifecycle/transitions/{id}/status
HTTP method	GET
Parameters	id - Unique identifier for the resource transition

Response format

Aspect	Value
Return Code	200 OK

Example response

```
{
 "requestId": "80fc4a66-7e92-41f8-b4bb-7cb98193f5fa",
 "requestState": "COMPLETED",
"requestStateReason": "Transition successfully completed in 324 msecs",
"resourceId": "e09dbfcf-bb70-42ee-8c32-bdb83a22fb5d",
 "startedAt": "2017-05-01T12:00:00Z",
 "finishedAt": "2017-05-01T12:00:00Z"
 "context": {
  "AsynchronousTransitionResponses": "false"
}
```

Table 37. Get resource transition status fields

Field	Description	Mandatory
requestId	The id of the request defined by the resource manager This should be a GUID	yes

Table 37. Get resource transition status fields (continued)

Field	Description	Mandatory
requestState	A string representing the state of the request Allowable states for the request state are: • PENDING • IN_PROGRESS • COMPLETED • CANCELLED • FAILED	yes
requestStateReason	A string giving the reason for the State of the request	no
resourceId	The id of the resource within the context of the resource manager This should be a GUID	yes
startedAt	The time the transition was started	yes
finishedAt	The time the transition was completed	no
context	Context is included for future use	no
AsynchronousTransition Responses	Indicates whether this transition will send responses asynchronously via Kafka.	no

Resource type configuration (asynchronous)

Optionally, a resource manager can emit events when the definition of a resource type changes, or a new resource type is created or deleted. This allows the resource manager to inform Agile Lifecycle Manager that a resource description has changed.

Note: The asynchronous resource type configuration described here is not related to the resource manager supporting asynchronous responses, which is referenced in the "Get Resource Manager configuration" on page 124 topic and described in more detail in the "Resource lifecycle management (asynchronous)" on page 135 topic.

Resource type update message example

```
{
 "name": "resource::openstack-network::1.0",
 "descriptor": "YAML Descriptor for this resource type",
 "state": "PUBLISHED",
 "createdAt": "2017-05-01T11:22:33Z",
 "lastModifiedAt": "2017-05-04T12:13:14+01:00"
}
```

Table 38. Resource type configuration (asynchronous) fields

Field	Description	Mandatory
name	The name of the resource type Must follow the naming structure defined in the Assembly Specification document	yes

Table 38. Resource type configuration (asynchronous) fields (continued)

Field	Description	Mandatory
descriptor	The resource descriptor A valid YAML document as a string	yes
state	The state of the resource descriptor Currently only PUBLISHED is allowed	yes
createdAt	The date the resource types was created XML Date time format.	yes
lastModifiedAt	The date the resource type was last changed. XML Date time format.	yes

Resource lifecycle management (asynchronous)

Optionally, a resource manager can choose to emit responses when transitions are completed (either successfully or not). It is recommended that this method is used to avoid Agile Lifecycle Manager needing to poll for the status periodically.

Resource transition response message example

The resource manager informs Agile Lifecycle Manager that this method will be used alongside the polling interface by indicating that it supports this feature by setting asychronousTransitionsReponses to `true`. If the resource manager supports the asynchronous response mechanism, then it will also support the polling update method.

```
{
 "requestId": "80fc4a66-7e92-41f8-b4bb-7cb98193f5fa",
 "resourceManagerId": "default-rm::dev",
 "deploymentLocation": "dev-cloud",
 "resourceType": "resource::openstack-network::1.0",
 "transitionName": "Install",
 "resourceInstance": {
  "resourceId": "default-rm://c675e0bd-9c6c-43ca-84bf-2c061d439c6b",
  "metricKey": "2530c175-541e-43df-89ae-6c34bc351d9b",
  "resourceName": "dev-network",
  "resourceType": "resource::openstack-network::1.0",
  "resourceManagerId": "default-rm::dev",
  "deploymentLocation": "dev-cloud",
  "properties": {
   "propertyName": "propertyValue"
},
  "createdAt": "2017-05-01T12:00:00Z",
  "lastModifiedAt": "2017-05-01T12:00:00Z",
  "internalResourceInstances": [
{
    "id": "3cb7822b-fc44-46ab-8072-9c65cd778d1f",
    "name": "MGMT-NETWORK",
    "type": "OpenDaylight::PrivateNetwork"
   }
  ]
},
"context": {},
 "requestState": "COMPLETED",
"requestStateReason": "Transition successfully completed in 324 msecs",
 "startedAt": "2017-05-01T12:00:00Z",
 "finishedAt": "2017-05-01T12:00:00Z"
}
```

Table 39. Get resource transition status fields

Field	Description	Mandatory
requestId	The id of the request	yes
resourceManagerId	The id of the resource manager	yes
deploymentLocation	The name of the location associated with the resource	yes
resourceType	The type of the resource	yes
transitionName	The name of the transition associated with this response	yes
resourceInstance	A section that contains the details of the resource instance	yes
resourceId	The ID of the instance of a resource	yes
resourceName	The name of the resource type	yes
resourceType	The name of the resource type	yes
resourceManagerId	The ID of the resource manager instance	yes
deploymentLocation	The name of the deployment location where the instance exists	yes
properties	A set of key value pair with properties describing key behavior	no
propertyName	A name of a property to be used in the search	no
propertyValue	The value associated with the propertyname	no
createdAt	The date the resource type was created XML Date time format	yes
lastModifiedAt	The date the resource type was last changed XML Date time format	yes
internalResourceInstances	Contains details of the underlying instances created by the resource manager in response to the intent request. This is a list and may contain many sets of Id, name and type fields.	yes - the underlying resource does not match the information otherwise used by the resource manager
context	Future use only	no
requestState	The state of the transition.	yes
requestStateReason	A string describing the reason for the state of the request, for example an error message.	yes
startedAt	The time the transition was started	yes
finishedAt	The time the transition was completed	no

Publishing metrics

Resources publish metrics via Kafka. Each of the two metrics described in this topic, that is, integrity and load metrics, are published to a separate Kafka topic.

Integrity metrics

Integrity metrics are published to the 'alm__integrity' topic. The message contents are:

Integrity metrics message content

```
{
  "metricKey" : "142971c5-a84b-4d34-af15-435ba8640aec",
  "metricName" : "h_integrity"
  "integrity" : "OK",
  "message" : "Everything is working"
}
```

Integrity metrics message fields

Field	Description	Mandatory
metricKey	The key provided when the resource was created	yes
metricName	The name of the metric as defined in the resource descriptor	no
integrity	Allowed values: **OK** When the resource is healthy and passing its Integrity checks **BROKEN** When the checks fail	yes
message	An optional message to add value to the metric; useful in the event of BROKEN	no

Load metrics

Load metrics are published to the 'alm__load' topic. The message contents are:

Load metrics message content

```
{
  "metricKey" : "818127b3-1904-4737-a60c-8c7bab73532d",
  "metricName" : "h_load"
  "load" : 76,
  "message" : "Load is high"
}
```

Load metrics message fields

Field	Description	Mandatory
metricKey	The key provided when the resource was created	yes
metricName	The name of the metric as defined in the resource descriptor	no
load	A value between 0 and 100 indicating the level of the load a resource is experiencing. A higher value indicates a higher load.	yes

Field	Description	Mandatory
message	An optional test string to include information about the integrity of the resource. For example, it may include an error code.	no

Resource descriptor YAML specifications

This section describes the descriptors that are used by Agile Lifecycle Manager.

Agile Lifecycle Manager needs to have descriptions of the building blocks of applications that it is going to manage. The basic building blocks are described in this 'resource descriptor' section. Sets of these resource descriptors are composed into assembly descriptors, which are described in “Assembly descriptor YAML specifications” on page 147.

Within the assembly will be a description of the relationships between resources that allow configuration to be applied to the actual instances of the components that Agile Lifecycle Manager will manage. Assemblies may also reference assemblies and existing infrastructure items, such as network instantiated outside of Agile Lifecycle Manager.

Naming

The resource descriptor name field will contain the following string:

```
resource::name::1.0
```

The name must start with a letter (either case), and can include letters, numbers, underscores and hyphens. The name must not contain spaces, and the version is fixed to 1.0 for this release. Both name and version are mandatory.

Resource descriptor sections

This sections describes the resource descriptors.

Header

The header includes the name and the description of the descriptor and associated resource manager type. Each resource is associated with a resource manager that has a declared type. This is shown by the field resource-manager-type. The contents of the field must be a globally unique string.

```
name: resource::c_streamer::1.0
description: component package for  c_streamer
resource-manager-type: urbancode.ibm.com
```

'properties'

This section contains the properties that belong to the resource descriptors. These include the full set of properties that are required to orchestrate them through to the Active state. These can be understood as the context for the management of the item during its lifecycle.

```
properties:
  deploymentLocation: # the name of the property
    type: string
    required: true
    description: The name of the openstack project(tenant) to install this assembly in.
  numOfStreamers:
    type: string
    description: the number of streamers that should be created at install time
    default: 2
  tenant_key_name:
    type: string
    required: true
    description: The ssh key for the current tenant
  flavor:
    value: m1.small
  cluster_public_ip_address:
    type: string
    description: the public IP address for this cluster
    read-only: true
    value: '${balancer.publicIp}'
```

Each property name must be unique within its property section. The types of properties can be `string`. Password-indicated fields will contain passwords or sensitive data. Properties are optional unless explicitly defined as required by the inclusion of a `required: true` flag.

Properties marked as `read-only: true` will typically have that value set by the time the associated component instance is in the Active state. These fields must not be marked as `required: true`.

Properties may be declared with a `default` value or a specific `value` or neither. Where the value field is used it may either be an explicit value or it may reference to another property within the description. When referencing a property in the assemblies main property section the reference will look as follows: `value: '${max_connections}'`.

Agile Lifecycle Manager will assign an internal name and identifier for each resource instance it creates.

It also supplies the index number of a resource in a cluster. These values can be useful to give unique names for servers, for example. To access them a property may have its value set to `${instance.name}`, `${instance.id}` or `${instance.index}`. These should be placed in the value field of a property. Agile Lifecycle Manager will then replace placeholders with the appropriate value.

Note: Assembly descriptor properties are defined in the following topic: “'properties'” on page 148

Capabilities and requirements

These two sections allow designers to explain what functions the resources are implementing or need before they can work successfully. These might be expressing that networks or various types must be available for the resource instances to work or it may be describing that a resource supports, for example, incoming http requests.

The type is a string that expresses the capability or requirement. The values in these strings will have to be agreed across an organization and where possible they should be agreed by the industry. Resource capabilities should use common industry terms. In the examples below the idea is that `httpStreamOutput` indicates that the capability is using the http protocol in a stream form and in an output direction. The OS::Neutron:Net is the resource type from OpenStack associated with a network instantiated within neutron.

'capabilities'

Capabilities are used to enable service designers to understand what function a resource provides.

```
capabilities:
  VideoStream:
    type: httpStreamOutput
```

```
capabilities:
  Network:
    type: neutronNetwork
```

'requirements'

Requirements contain the list of capabilities that the resource requires in order to work.

```
requirements:
  VideoNetwork:
    type: neutronNetwork
  ManagementNetwork:
    type: neutronNetwork
  RemoteNFSMountPoint:
    type: nfsExportMountpoint
```

'operations'

This section defines operations that can be called to enable relationships to be created between resources. Operations definitions in the resource have a name and a set or properties. Where a resource descriptor describes an operation an enclosing assembly may expose this by referencing the lower level operation. As a convention the name of the operation should be linked to the capability that is being enabled through the creation of the relationship.

Resource descriptor operations

```
operations:
  RemoveHttpStreamOutput:
    description: removes the http server from being managed by the balancer
    properties:
      server_ip:
        type: string
        description: Http Server Ip Address
        default: the ip address
      server_port:
        type: string
        description: http server port number
        default: '8080'
  AddHttpStreamOutput:
    description: adds an http server to the balancer's pool
    properties:
      max_connections:
        type: string
        description: Maximum connections for the balanced server
        default: 3
      server_ip:
        type: string
        description: Ip Address of the server to be balanced
      server_port:
        type: string
        description: Port on balanced server
        default: '8080'
```

'lifecycle'

Resource descriptors must support the install and uninstall lifecycle transitions. These are mandatory.

However, they may implement the other lifecycle transitions, which are:

- Configure
- Start
- Stop
- Integrity

Where the transition is not provided by the resource, Agile Lifecycle Manager is free to change the state of the associated component instances without calling any underlying transition.

The lifecycle section will contain a list of all the transitions that the resource supports. In the case of the following example, no Configure transition is defined.

```
lifecycle:
- Install
- Uninstall
- Start
- Stop
- Integrity
```

A resource may be one that can only be used within a reference section of an assembly. These resources will not have an Install or Uninstall lifecycle defined.

Resources that are used as reference resources do not have to include the lifecycle section. Any resource without the Install and Uninstall cannot be instantiated by Agile Lifecycle Manager, and therefore should not be included in the composition section of an assembly.

Metrics and policies

A resource descriptor may indicate that the underlying resource will emit one or more metrics. A metric is defined as having a name, type and an optional publication-period.

If no publication period is given at all, a default of 60 seconds is assumed. The publication period is in seconds.

A value of 0 means no metrics will be published. The value must be +integer.

There are two reserved types that are used by the Agile Lifecycle Manager to monitor the health of the associated resource instances:

```
metric::integrity
```

and

```
metric::load
```

Example resource metrics: This example shows the policy associated with the Integrity metric. Within the resource descriptor the policies section contains details of the heal policy. This allows the smoothing interval to be defined for the resource. Each policy has a name, the associated metric, an action (heal) and a properties section.

```
metrics:
  h_integrity:
    type: "metric::integrity"
    publication-period: "${integrity_publication_period}"
  load:
    type: "metric::load"
```

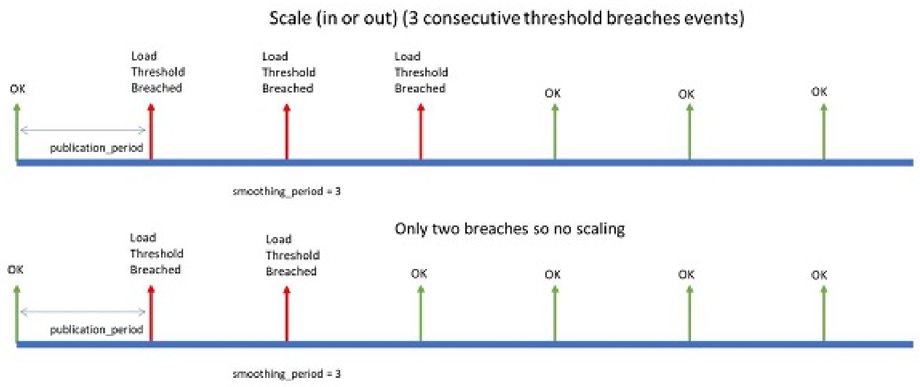
Note: Property references are used to allow the value for the publication period to be passed from a separate properties section in the resource.

Example policies section: In the following policy example the smoothing value is used to prevent 'snap' changes happening due to unusual short term conditions.

Note: Properties, smoothing, threshold, and target are all policy-specific properties that may not be required by other types of policies.

```
policies:
  heal:
    metric: "h_integrity"
    type: "policy::heal"
    properties:
      smoothing:
        value: "${number-of-intervals}"
```

Example of smoothing



Resource descriptor YAML examples

The examples included in this section show the c_balancer, c_streamer and the net_video resources.

'resource' examples

resource::net_video:1.0

The following resource creates a neutron network.

```
name: resource::net_video::1.0
description: resource to create an internal neutron network that includes
a subnet
resource-manager-type: urbancode.ibm.com
properties:
  subnetCIDR:
    type: string
    description: The subnet classless inter-domain routing
    default: '10.0.1.0/24'
  networkName:
    type: string
    description: Network Name
    value: VIDEO
  subnetDefGwIp:
    type: string
    description: Default Gateway IP address
    default: '10.0.1.1'
  network-id:
    type: string
    description: the id of the network just created
    read-only: true
capabilities:
    Network:
        type: OS::Neutron::Net
lifecycle:
- Install
- Uninstall
```

resource::h_simple::1.0

The following resource is a simple component with metrics and policies.

```
name: "resource::h_simple::1.0"
description: "resource for  t_simple"
properties:
  server_name:
    type: "string"
    value: "${instance.name}"
  referenced-internal-network:
    type: "string"
    description: "Generated to reference a network"
  reference-public-network:
    type: "string"
    description: "Generated to reference public network"
  image:
    type: "string"
    description: "The Image reference"
  key_name:
    type: "string"
    description: "SSH key"
  data:
    type: "string"
    description: "parameter passed"
    default: "data"
  integrity_publication_period:
    type: "string"
    description: "the period that should be used to publish the metrics"
    default: "60"
  publication_period:
    type: "string"
    description: "the period that should be used to publish the metrics"
    default: "60"
  number-of-intervals:
    type: "string"
    description: "The intervals before calling a Heal"
    default: "3"
  output:
    type: "string"
    description: "an example output parameter"
    read-only: true
operations:
  CreateRelationship1:
    description: "Create a new relationship"
    properties:
      source:
        type: "string"
        description: "that name of the source"
      target:
        type: "string"
        description: "that name of the target"
  CeaseRelationship1:
    description: "Cease an existing relationship"
    properties:
      source:
        type: "string"
        description: "that name of the source"
      target:
        type: "string"
        description: "that name of the target"
  CreateRelationshipr2:
    description: "Create a new relationship"
    properties:
      source:
        type: "string"
        description: "that name of the source"
      target:
        type: "string"
        description: "that name of the target"
  CeaseRelationship2:
    description: "Cease an existing relationship"
    properties:
      source:
        type: "string"
        description: "that name of the source"
      target:
        type: "string"
        description: "that name of the target"
  CreateRelationship3:
    description: "Create a new relationship"
    properties:
      source:
        type: "string"
        description: "that name of the source"
      target:
        type: "string"
        description: "that name of the target"
  CeaseRelationship3:
    description: "Cease an existing relationship"
    properties:
      source:
        type: "string"
        description: "that name of the source"
      target:
        type: "string"
        description: "that name of the target"
metrics:
  h_integrity:
    type: "metric::integrity"
    publication-period: "${integrity_publication_period}"
  load:
    type: "metric::load"
policies:
  heal:
    metric: "h_integrity"
    type: "policy::heal"
    properties:
      smoothing:
        value: "${number-of-intervals}"
lifecycle:
- "Configure"
- "Install"
- "Integrity"
- "Start"
- "Stop"
- "Uninstall"
resource-manager-type: "test-rm"
```

resource::c_streamer::1.0

The following descriptor creates a virtual server that streams video traffic using the http protocol.

```
name: resource::c_streamer::1.0
description: resource descriptor for  c_streamer
resource-manager-type: urbancode.ibm.com
properties:
  key_name:
    type: string
    required: true
    description: the ssh key-pair name to be used by openstack with the
associated VM instances
  referenced-management-network:
    type: string
    required: true
    description: The id of the network that will act in the role of a
management network
  flavor:
    type: string
    required: true
    description: Flavor to be used for compute instance
  server_name:
    type: string
    required: true
    description: the name of the server to be created
  referenced-video-network:
    type: string
    description: The id of the network that will act in the role of an
internal network
  availability_zone:
    type: string
    description: Name of availability zone in which to create the instance
    default: DMZ
  privateIp:
    type: string
    description: IpAddress of server on the internal network
    read-only: true
  mgmtIp:
    type: string
    description: IpAddress of server on the management network
    read-only: true
  integrity_publication_period:
    type: string
    description: the number of seconds between publishing integrity metric
    default: 60
  number-of-intervals:
    type: string
    description: the number of intervals for smoothing
    default: 3
capabilities:
    VideoStream:
        type: httpStreamOutput
requirements:
    VideoNetwork:
        type: neutronNetwork
    ManagementNetwork:
        type: neutronNetwork    RemoteNFSMountPoint:
        type: nfsExportMountpoint
lifecycle:
- Install
- Uninstall
- Configure
- Start
- Stop
- Integrity
operations:
  MountStorage:
    description: An operation to enable the streamer to mount a remote NFS
mount point
    properties:
      remote_nfs_port:
        type: string
        description: Port for the NFS
        default: '2049'
      remote_nfs_server_ip:
        type: string
        description: Ip Address of remote nfs server
      remote_mount_point:
        type: string
        description: Location of NFS Exported Mount Point
        default: /
      local_mount_point:
        type: string
        description: The location where the remote nfs mount will be
mounted in the local machine
        default: /mnt
  UnmountStorage:
    description: An operation to unmount a remote NFS mount point
    properties:
      local_mount_point:
        type: string
        description: The location where the remote nfs mount will be
mounted in the local machine
        default: /mnt
```

resource::c_balancer::1.0

The following example load balancer server balances connections and data streams between pools of available application servers.

```
name: resource::c_balancer::1.0
description: component package for a http loadbalancer
resource-manager-type: UrbanCode
cloud-target: OpenStack
properties:
  key_name:
    type: string
    description: ssh key_name.
  referenced-management-network:
    type: string
    description: Generated to reference a network
  referenced-internal-network:
    type: string
    description: Generated to reference a network
  referenced-public-network:
    type: string
    description: Generated to reference a network
  flavor:
    type: string
    description: Flavor to be used for compute instance
  server_name:
    type: string
    description: server name of the balancer
  availability_zone:
    type: string
    description: Name of availability zone in which to create the instance
    default: DMZ
  mgmtIp:
    type: string
    description: IpAddress of server in management network
    readOnly: true
  internalIp:
    type: string
    description: IpAddress of server on internal network
    readOnly: true
  publicIp:
    type: string
    description: Public IpAddress of server
    readOnly: true
  integrity_publication_period:
    type: string
    description: the number of seconds between publishing integrity metric
    default: 60
  number-of-intervals:
    type: string
```

```
        description: the number of intervals for smoothing
        default: 3
    capabilities:
        HttpLoadBalancer:
            type: loadbalancerHttp
    requirements:
        PublicNetwork:
            type: neutronNetwork
        ManagementNetwork:
            type: neutronNetwork
        HttpServer:
            type: http
    lifecycle:
    - Install
    - Uninstall
    - Start
    - Stop
    operations:
      RemoveBalancedHttpServer:
        description: removes the http server from being managed by the balancer
        properties:
          server_ip:
            type: string
            description: Http Server Ip Address
            default: the ip address
          server_port:
            type: string
            description: http server port number
            default: '8080'
      AddBalancedHttpServer:
        description: adds an http server to the balancer's pool
        properties:
          max_connections:
            type: string
            description: Maximum connections for the balanced server
            default: 3
          server_ip:
            type: string
            description: Ip Address of the server to be balanced
          server_port:
            type: string
            description: Port on balanced server
            default: '8080'
```

Assembly descriptor YAML specifications

This section describes the assembly descriptors that are used by Agile Lifecycle Manager.

Agile Lifecycle Manager needs to have descriptions of the building blocks of applications that it is going to manage. The basic building blocks are described in "Resource descriptor YAML specifications" on page 138. Sets of these resource descriptors are composed into assembly descriptors to allow designers to describe a complete application or service that they need Agile Lifecycle Manager to manage.

Within the assembly will be a description of the relationships between resources that allow configuration to be applied to the actual instances of the components that Agile Lifecycle Manager will manage. Assemblies may also reference assemblies and existing infrastructure items, such as network instantiated outside of Agile Lifecycle Manager.

Naming

The assembly descriptor name field will contain the following string:

```
assembly::name::1.0
```

The name must start with a letter (either case), and can include letters and numbers and underscores and hyphens. The name must not contain spaces, and must end with either a letter (either case) or a number. The version is fixed to 1.0 for this release. Both name and version are mandatory.

Related concepts:

Chapter 5, "Getting started (using the APIs)," on page 47
Agile Lifecycle Manager provides both a graphical UI and an HTTP API allowing the creation and administration of assemblies. This section describes a set of basic scenarios to get started using the APIs.

Assembly descriptor sections

This topic describes the sections that apply to the assembly descriptors.

Header

The header includes the name and the description of the descriptor.

```
name: assembly::Streamer_cluster::1.0
description: An Assembly for a front end cluster comprising of a loadbalancer supported
by an authorisation proxy and video streamers using a shared NFS based storage
```

'properties'

Note: The properties defined here apply to the top-level property section for the descriptor. Rules applicable to property names and the 'value' field can be applied to all property sections

This section contains the properties that belong to assembly descriptors. These include the full set of properties that are required to orchestrate them through to the Active state. These can be understood as the context for the management of the item during its lifecycle.

```
properties:
  deploymentLocation: # the name of the property
    type: string
    required: true
    description: The name of the openstack project(tenant) to install this assembly in.
  resourceManager:
     value: '${resourceManager}'
  numOfStreamers:
    type: string
    description: the number of streamers that should be created at install time
    default: 2
  tenant_key_name:
    type: string
    required: true
    description: The ssh key for the current tenant
  flavor:
    value: m1.small
  cluster_public_ip_address:
    type: string
    description: the public IP address for this cluster
    read-only: true
    value: '${balancer.publicIp}'
```

Each property name must be unique within its property section. The name cannot contain the period (`.`) character.

Restriction: Currently, the type `field` is not used and all properties are assumed to be of type `string`. This field will be used in the future to allow handling of different types of data, such as dates, IP addresses, and encrypted values. To avoid compatibility issues in the future, it is recommended that you do not use this field, or use the default value of `string`.

Properties are optional unless explicitly defined as required by the inclusion of a `required: true` flag. This only affects the top-level assembly and means that a value must be present (that is, not null) for a property. This can be evaluated from the 'value' field, a 'default', or passed in from the intent request.

Properties marked as `read-only: true` will not be overridden by values mapped in from an enclosing assembly or from the intent request. This is typically used for properties that are calculated from or returned by the resource itself.

Properties may be declared with a `default` value or a specific `value` or neither. Where the value field is used it may either be an explicit value or it may reference another property within the descriptor. This will happen in assemblies where properties given to the assembly may be used within the various other property sections. When referencing a property in the assembly's main property section the reference will look as follows: `value: '${max_connections}'`. If the reference is to a property within the assembly's other sections the reference must include the name of the enclosing object, such as `value: '${balancer.publicIp}'`. This references the property `publicIp` within the balancer section of the composition section.

`deploymentLocation` is a special property that is used by Agile Lifecycle Manager to place the resultant resource in the correct location. It will only appear in an assembly descriptor. The contents of the property will be specific to the resource manager that handles the resource.

`resourceManager` is another special property that passes the name of the resource manager instance that will be used to manage the resource.

Agile Lifecycle Manager will assign an internal name and identifier for each resource and assembly instance it creates. These values can be useful to give unique names for servers, etc. To access them a property may have its value set to `${instance.name}` or `${instance.id}`.

Important: It is essential that there is a space between the `value:` and the quoted property value string. If there is no space between these two items then the value string will be treated as a single string.

Capabilities and requirements

These two sections allow designers to explain what functions the assemblies are implementing or need before they can work successfully. These might be expressing that networks or various types must be available for the resource instances to work or it may be describing that a resource supports, for example, incoming http requests.

The type is a string that expresses the capability or requirement. The values in these strings will have to be agreed across an organization and where possible they should be agreed by the industry. Resource capabilities should use common industry terms. In the examples below the idea is that `httpStreamOutput` indicates that the capability is using the http protocol in a stream form and in an output direction. The OS::Neutron:Net is the resource type from OpenStack associated with a network instantiated within neutron.

'capabilities'
: Capabilities are used to enable service designers to understand what function a resource or assembly provides.

```
capabilities:
  VideoStream:
    type: httpStreamOutput
```

```
capabilities:
  Network:
    type: neutronNetwork
```

'requirements'
: Requirements contain the list of capabilities that the assembly requires in order to work.

```
requirements:
  VideoNetwork:
    type: neutronNetwork
  ManagementNetwork:
    type: neutronNetwork
  RemoteNFSMountPoint:
    type: nfsExportMountpoint
```

'operations'

This section defines operations that can be called to enable relationships to be created between assemblies. Operations definitions in the resource have a name and a set or properties. Where a resource descriptor describes an operation the enclosing assembly may expose this by referencing the lower level operation.

```
operations:
  SetLBBalancer:
    source-operation: balancer.AddHttpStreamOutput
```

Composition and references

Definition: A **component** is an assembly that is included within an assembly composition section, and will be instantiated as a result of requesting a new instance of the enclosing assembly.

Assemblies allow a designer to group a set of resources and assemblies, collectively known as components, into an assembly to create a new application/service. Those used within the composition section will be instantiated and managed by Agile Lifecycle Manager.

When Agile Lifecycle Manager has already instantiated an assembly, it is possible for another assembly to share the instance by referencing it within the references section. The references section can also refer to existing objects that may have been created outside Agile Lifecycle Manager. Agile Lifecycle Manager will resolve both of these types of references from the properties supplied, and access to the instances properties and operations is then available to the referencing assembly.

'composition'

Assemblies gather resources and other assemblies for either a whole or part of a solution. The composition section is used to reference components that will be instantiated as part of the installation of the assembly.

```
composition:
  streamer: # The name
    type: resource::c_streamer::1.0
    quantity: ${numOfStreamers}
    properties:
      #not shown for brevity
  balancer:
    type: resource::c_balancer::1.0
    quantity: 1
    properties:
      #not shown for brevity
  net_video:
    type: resource::net_video::1.0
    quantity: 1
    properties:
      #not shown for brevity
```

Each entry in this section must give a name to the item which will form the basis for the instance name for the actual running components. It also includes a quantity that defaults to 1. In a non-clustered environment, the 'quantity' property defines exactly how many instances will be created.

Remember: The rules governing properties are defined in "'properties'" on page 148.

```
composition:
  streamer:
    type: resource::c_streamer::1.0
    cluster:
      # not shown for brevity
    properties:
      deploymentLocation:
        value: '${deploymentLocation}'
      resourceManager:
        value: '${resourceManager}'
      flavor:
        value: m1.small
      server_name:
        value: ${instance.name}
      referenced-video-network:
        value: ${net_video.network-id}
      availability_zone:
        value: DMZ
      mgmtIp:
        type: string
        description: MGMT IpAddress of server
        read-only: true
```

Clusters

It is also possible to define a cluster section for a component that indicates that the component of the assembly may comprise of more than one instance of the same type (node) to support capacity and or availability requirements.

The 'initial-quantity' is an optional property that must be between the minimum and maximum nodes values. If not set, it defaults to 'minimum-nodes'.

The 'minimum-nodes' setting is optional and defaults to '1' if not set. It can be set to '0' if no instances of the component are required at initial install.

The 'maximum-nodes' setting is optional and if set it must be greater than or equal to the 'minimum-nodes' value.

The 'scaling-increment' setting is optional and defaults to '1'; it determines the number of instances added or removed from the cluster during scaling.

```
composition:
  streamer:
    type: resource:: c_streamer::1.0
    cluster:
      initial-quantity: ${numOfServers}
      minimum-nodes: 1
      maximum-nodes: 4
      scaling-increment: 1
    properties:
      data:
        value: ${data}
      ip_address:
        read-only: true
```

Note: The properties 'quantity' and 'initial-quantity' are mutually exclusive. When running in a clustered environment the 'quantity' property, if defined, will be ignored, and the value of 'initial-quantity' used instead.

'references'

The reference section is similar to the composition section except that the items referenced in this section must be pre-existing before Agile Lifecycle Manager will instantiate any of the items in the assembly's composition section.

Two types of references can be resolved by Agile Lifecycle Manager:
- Existing assembly instances
- External resources that are managed directly by a resource manager

Assembly references require the full name of the assembly within the type field. The following example shows the use of the semantic versioning to allow more flexibility when resolving to instances of the assembly. The properties are used to help Agile Lifecycle Manager to find the instance of the item required by the current assembly. With items that have been created through Agile Lifecycle Manager the referencing assembly can refer to any of the instance's properties from the items property section. Referenced assemblies can be used by the enclosing assembly to establish relationships.

Resource instances managed directly by a resource manager may be referenced. These will have resource descriptors as any resource, however they will not include the Install or uninstall lifecycle steps.

To read the references section, each item has a local name used to refer to the item with in the assembly. The type directs Agile Lifecycle Manager to fetch the required resource type. The properties are then used by Agile Lifecycle Manager to narrow down to a single instance of the resource type that can be used by the enclosing assembly. If Agile Lifecycle Manager finds more than one resource that fits the information provided an error occurs and the assembly will not be instantiated.

```
references:
  storage: # reference to an existing assembly instance
    type: assembly::storage_cluster::^1.0
    properties:
      deploymentLocation:
        value: '${deploymentLocation}'
      resourceManager:
        value: '${resourceManager}'
      name:
        value: '${storage-name}'
  management-network: # reference to a neutron network not created by the
Agile Lifecycle Manager
    type: resource::ucd_network::1.0
    properties:
      deploymentLocation:
        value: '${deploymentLocation}'
      resourceManager:
        value: '${resourceManager}'
      name:
        value: ${management-network-name}
```

Once found the properties of these referenced items may be accessed using the following method:

```
'${referenced-item-name.property-name}'
```

```
balancer:
    type: 'resource::c_balancer::1.0'
    quantity: '1'
    properties:
      ...
      referenced-management-network:
        value: '${management-network.id}'
```

All the properties from the assembly instances referenced are available for use in this manner. Resource descriptor properties are defined in the following topic: "'properties'" on page 138

'relationships'

Relationships are established between two components that enable the 'requirements' of one component (known as the 'target') to be satisfied by another component that provides the 'capability' (known as the 'source').

Defining relationships

The 'source' and 'target' of a component are defined by the following fields:

- source-capabilities
- target-requirements

In order to define a relationship between two components, the name of each component, as defined in the 'composition' or 'reference' section of the descriptor, is combined with the name of the capability or requirement, as in the following example:

```
source-capabilities:
    - A.capability-3
target-requirements:
    - B.requirement-3
```

`source-capabilities` key

A	Derived from the composition section
.	An agreed delimiter

capability-3
The name of the 'capability' defined within the organization

A reference component can only be defined as a source-capability. In this instance, only the name of the reference needs to be provided.

Within a relationship definition the 'properties' field may refer to the components defined under the 'source-capability' and 'target-requirements' fields as 'source' and 'target' respectively, as I the following example.

```
property1:
    value: ${source.name}
property2:
    value: ${target.name}
```

Above ${source.name} and ${target.name} is used to refer to the 'source' components (as defined in source-capabilities) 'name' property and 'target' components (as defined in target-requirements) 'name' property accordingly.

The 'lifecycle' section within relationships consist of two transitions: Create and Cease. The transitions described so far allow designers to specify what operations to perform during the Creation and Cessation (or removal) of a relationship for a source and target component, as in the following example.

```
lifecycle:
      Cease:
      - target.CeaseRelationship
      - source.CeaseRelationship
      Create:
      - target.CreateRelationship
      - source.CreateRelationship
```

The operations called depend on the components involved. The operations are called in the order they appear in the Create or Cease sections. A relationship may only call one operation, that is, only either a target or a source operation. Operations are referenced as `source.<operation-name>` or `target.<operation-name>` with `<operation-name>` referring to an operation defined in the assembly or resource descriptor associated with the component.

Establishing relationships

Relationships are created when the components to be related are in particular states.

The 'source-state' and 'target-state' fields are used to define the state required to establish a relationship, as in the following example.

```
source-state: Active
    target-state: Active
```

By default this means that the relationship would be created when the **source** is in the Active state, and before the **target** has transitioned to the Active state.

Further control when defining relationships is available via the 'source-state-modifier' and 'target-state-modifier' fields. These are used to define whether relationships are established before (pre) or after (post) they reach their source or target state as previously defined via source-state and target-state definitions. For example, 'source-state-modifier', if not present, is by default `post` while 'target-state-modifier' if not present is by default pre. Relationships are always ceased (removed) **before** the associated component leaves the state defined in the source-state and target-state fields.

```
relationships:
  nfs_mount:
    source-capabilities:
    - storage.NFSMountpoint
    target-requirements:
    - streamer.RemoteNFSMountPoint
    source-state: Active
    target-state: Inactive
    properties:
      remote_nfs_port:
        value: '2049'
      remote_nfs_server_ip:
        value: '${source.privateIp}'
      remote_mount_point:
        value: '/'
      local_mount_point:
        value: '/mnt'
    lifecycle:
      Create:
      - source.MountStorage
      Cease:
      - source.UnmountStorage
```

Like the overall assembly and resources, relationships have a set of properties that are available to the operations associated with the lifecycle transitions of the relationship.

Placement

To deploy components to the correct location, Agile Lifecycle Manager will use two properties called `deploymentLocation` and `resourceManager`. The resourceManager property will be used to find the correct resource manager that manages the resource for the location defined in the deploymentLocation property. The combination of these two uniquely identifies where and how a resource will be managed.

A placement is also involved when trying to resolve the instances defined in the references section. Before a reference can be resolved any associated placement rules will have been applied. This will then allow Agile Lifecycle Manager to find the appropriate instance of the reference required. The two properties will also be needed on each reference.

Metrics and policies

A resource descriptor may indicate that the underlying resource will emit one or more metrics. Example metrics are found in a resource descriptor (but **not** in the assembly descriptor).

```
metrics:
  lb_integrity:
    type: metric::integrity
    publication-period: ${integrity_publication_period}
  lb_load:
    type: metric::load
    publication-period: ${publication_period}
```

Load metrics can be promoted in an assembly using a similar mechanism to operation metrics.

```
metrics:
    b1_load:
        source-metric: B1.load
```

Within an assembly the policy section will contain details of the policies for the underlying resources load metric and how that should be used to mange the scaling of components. Each policy has a name, the associated metric, an action and a set of properties that are used to handle the policy.

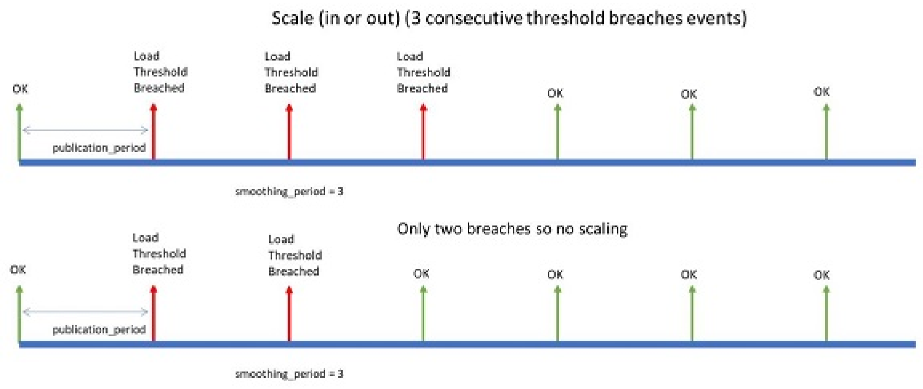
Example policies

The following example shows the policy associated with the load metric on a resource. This is used to ScaleOut and ScaleIn a component, as indicated by the value in the target properties. The example also shows that the metric produced by `A.load` will be used to indicate when the target `B` will be scaled.

```
policies:
  scaleStreamer:
    type: policy::scale
    metric: A.load
    target: B
    properties:
      scaleOut_threshold: ${scaleOut_threshold}
      scaleIn_threshold: ${scaleIn_threshold}
      smoothing: ${scale_smoothing}
```

Load is expressed as a percentage and the thresholds are simple integers. When the threshold is broken the scale event associated with the threshold will be enacted. To prevent this happening each time the load spikes, a smoothing value is applied. The threshold must be breached at least the number of times indicated by the smoothing value before the action will be enacted.

Example of smoothing



Assembly descriptor YAML examples

The examples included in this section are an assembly descriptor with policies, and another one that creates a set of video streamers and links them to a load balancer, which is also created.

assembly::h_bta::1.0

The following is an example of an assembly descriptor with policies.

```
name: assembly::h_bta::1.0
description: Basic Test Assembly
properties:
  data:
    default: "data"
    type: string
    description: 'parameter passed'
  numOfServers:
    description: number of servers
    type: integer
    default: 1
  output:
    description: an example output parameter
    type: string
    read-only: true
    value: ${B.output}
  deploymentLocation:
    type: string
    description: name of openstack project to deploy network
    default: admin@local
  resourceManager:
    type: string
    description: name of the resource manager
    default: test-rm
  scaleOut_threshold:
    type: integer
    description: threshold that the load metric must breach to potentially trigger
a scaleOut
    default: 90
  scaleIn_threshold:
     type: integer
     description: threshold that the load metric must breach to potentially trigger
a scaleOut
     default: 10
  scale_smoothing:
    type: integer
    description: the number of sequential periods the load metric must be above
threshold to trigger action
    default: 4
composition:
  A:
    type: resource::h_simple::1.0
    quantity: '1'
    properties:
      referenced-internal-network:
        value: ${internal-network.id}
      reference-public-network:
        value: ${public-network.id}
      image:
        value: ${xenial-image.id}
      key_name:
        value: "ACCANTO_TEST_KEY"
      data:
        value: ${data}
      output:
        value: "A_output"
      deploymentLocation:
        value: ${deploymentLocation}
      resourceManager:
        value: ${resourceManager}
  B:
    type: resource::t_simple::1.0
    cluster :
      initial-quantity: ${numOfServers}
      minimum-nodes: 1
      maximum-nodes: 4
      scaling-increment: 1
    properties:
      referenced-internal-network:
        value: ${internal-network.id}
      reference-public-network:
        value: ${public-network.id}
      image:
        value: ${xenial-image.id}
      key_name:
        value: "ACCANTO_TEST_KEY"
      data:
        value: ${data}
      output:
        value: ${A.output}
      deploymentLocation:
        value: ${deploymentLocation}
      resourceManager:
        value: ${resourceManager}
policies:
  scaleStreamer:
    type: policy::scale
    #metric: A.load
    #TODO hack until dto change
    metric: load
    target: B
    properties:
      scaleOut_threshold: ${scaleOut_threshold}
      scaleIn_threshold: ${scaleIn_threshold}
      smoothing: ${scale_smoothing}
references:
  internal-network:
    type: resource::openstack_neutron_network::1.0
    properties:
      deploymentLocation:
        value: ${deploymentLocation}
      resourceManager:
        value: ${resourceManager}
      name:
        type: string
        value: VIDEO
  public-network:
    type: resource::openstack_neutron_network::1.0
    properties:
      deploymentLocation:
        value: ${deploymentLocation}
      resourceManager:
        value: ${resourceManager}
      name:
        type: string
        value: public
  xenial-image:
    type: resource::openstack_glance_image::1.0
    properties:
      deploymentLocation:
        value: ${deploymentLocation}
      resourceManager:
        value: ${resourceManager}
      name:
        type: string
        value: xenial
relationships:
  third-relationship:
    source-capabilities:
    - A.capability-3
    target-requirements:
    - B.requirement-3
    source-state: Active
    target-state: Active
    properties:
      source:
        value: ${source.name}
      target:
        value: ${target.name}
    lifecycle:
      Cease:
      - target.CeaseRelationship3
      - source.CeaseRelationship3
      Create:
      - target.CreateRelationship3
      - source.CreateRelationship3
```

assembly::Streamer_cluster::1.0

The following example of an assembly descriptor will create a set of video streamers and link them to a load balancer which is also created. It requires the name of a storage assembly to be provided so that it can share the video content between the streamers.

```
name: assembly::Streamer_cluster::1.0
description: An Assembly for a front end cluster comprising of a loadbalancer
supported by an authorisation proxy and video streamers using a shared NFS based
storage properties:
  deploymentLocation:
    type: string
    required: true
    description: The location as required by the resource manager.
  resourceManager:
    type: string
    required: true
    description: The name of the resource resource manager.
  numOfStreamers:
    type: string
    description: the number of streamers that should be created at install time
    default: 2
  tenant_key_name:
    type: string
    required: true
    description: The ssh key for the current tenant
  management-network-name:
    type: uuid
    required: true
    description: the name of the management network in the tenant where the
assembly is to be installed
  public-network-name:
    type: uuid
    required: true
    description: the name of the public network associated with the tenant
where the assembly is to be installed
  max_connections:
    type: string
    description: Maximum connections for the balanced server
    default: '3'
  cluster_public_ip_address:
    type: string
    description: the public IP address for this cluster
    read-only: true
    value: '${balancer.publicIp}'
  scaleout-threshold:
    type: string
    description: the load value that when exceed will cause a scale out to be
invoked
    default: 80
  scalein-threshold:
    type: string
    description: the level of load that will cause a scale in to be invoked
    default: 10
composition:
```

```
streamer:
  type: resource::c_streamer::1.0
  cluster:
    initial-quantity: ${numOfStreamers}
    minimum-nodes: 2
    maximum-nodes: 10
    scaling-increment: 2
  properties:
    deploymentLocation:
      value: '${deploymentLocation}'
    resourceManager:
      value: '${resourceManager}'
    key_name:
      value: '${tenant_key_name}'
    referenced-management-network:
      value: '${management-network.id}'
    flavor:
      value: m1.small
    server_name:
      value: '${instance.name}'
    referenced-video-network:
      value: '${net_video.network-id}'
    availability_zone:
      value: DMZ
    integrity_publication_period:
      value: 120
    number-of-intervals:
      value: 4
balancer:
  type: 'resource::c_balancer::1.0'
  quantity: 1
  properties:
    deploymentLocation:
      value: '${deploymentLocation}'
    resourceManager:
      value: '${resourceManager}'
    key_name:
      value: '${tenant_key_name}'
    referenced-management-network:
      value: '${management-network.id}'
    referenced-internal-network:
      value: '${net_video.network-id}'
    referenced-public-network:
      value: '${public-network.id}'
    flavor:
      value: m1.large
    server_name:
      value: '${instance.name}'
    availability_zone:
      value: DMZ
    integrity_publication_period:
      value: 120
    number-of-intervals:
      value: 4
net_video:
  type: resource::net_video::1.0
  quantity: 1
  properties:
    deploymentLocation:
      value: '${deploymentLocation}'
    resourceManager:
      value: '${resourceManager}'
    subnetCIDR:
      type: string
      description: (Required)
      default: '10.0.1.0/24'
    networkName:
        type: string
        description: Network Name
        default: VIDEO
      subnetDefGwIp:
        type: string
        description: Default Gateway IP address
        default: '10.0.1.1'
references:
  management-network:
    type: resource::urbancode-network::1.0
    properties:
      deploymentLocation:
        value: '${deploymentLocation}'
      resourceManager:
        value: '${resourceManager}'
      name:
        value: ${management-network-name}
  public-network:
    type: resource::urbancode-network::1.0
    properties:
      deploymentLocation:
        value: '${deploymentLocation}'
      resourceManager:
        value: '${resourceManager}'
      name:
        value: ${public-network-name}
capabilities:
  HttpStream:
    type: httpStream
relationships:
  uses-net_video:
    source-capabilities:
    - net_video.Network
    target-requirements:
    - streamer.VideoNetwork
    - storage.VideoNetwork
    - balancer.VideoNetwork
    source-state: Active
    target-state: Inactive
  uses-management-network:
    source-capabilities:
    - management-network
    target-requirements:
    - streamer.ManagementNetwork
    - storage.ManagementNetwork
    - balancer.ManagementNetwork
    source-state: Active
    target-state: Inactive
  balancer-uses-public-network:
    source-capabilities:
    - public-network
    target-requirements:
    - balancer.PublicNetwork
    source-state: Active
    target-state: Inactive
  balanceStreamer:
    source-capabilities:
    - streamer.VideoStream
    target-requirements:
    - balancer.HttpServer
    source-state: Active
    target-state: Active
    properties:
      max_connections:
        value: '${max_connections}'
      server_ip:
        value: '${source.privateIp}'
      server_port:
        value: '8080'
    lifecycle:
      Create:
      - balancer.AddBalancedHttpServer
      Cease:
      - balancer.RemoveBalancedHttpServer
```

Notices

This information applies to the PDF documentation set for IBM Agile Lifecycle Manager.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement might not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
958/NH04
IBM Centre, St Leonards
601 Pacific Hwy
St Leonards, NSW, 2069
Australia

IBM Corporation
896471/H128B
76 Upper Ground
London
SE1 9PZ
United Kingdom

IBM Corporation
JBF1/SOM1 294
Route 100
Somers, NY, 10589-0100
United States of America

Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

Trademarks