http://wso2.com/

Stratos 2.0 Architecture Guide

Date: 5 April 2013

Email: support@wso2.com

WSO2 Stratos 2.0 Architecture Guide

Contents

Introduction

Stratos 2.0 is the next generation of WSO2 Stratos platform. At a glance Stratos 2.0 is

- A Public / Private PaaS provider.
- A Multi-tenant WSO2 middleware products pluggable as cartridges. Eg. WSO2 AS, WSO2 ESB, WSO2 BPS.
- Bring non-cloud middleware services and containers into the Cloud platform as single-tenant cartridges. Eg. PHP, Jetty.
- Includes core Stratos services such as Identity, Billing/Metering, Logging, Domain Mapping, Load balancing, Auto scaling etc.

The aim of this document is to provide some insight into Stratos2 architecture. It is not meant as an instructional document on Stratos2 installation or deployment. For such instructions please refer to **Stratos2 Installation Guide**, **Stratos2 Cartridge Development Guide** and **Stratos User Guide** shipped along with this document.

The architecture of Stratos2 is well designed to face current challenges of Cloud PaaS space. What make Stratos2 architecture unique from its predecessors is the **Cartridge** concept. A Cartridge is the core functional component of Stratos 2.0 which is pluggable. A Cartridge component can take use of core services provided by WSO2 Stratos 2.0 (e.g. auto-scaling, load-balancing, health monitoring, metering, billing, tenant provisioning, code deployment, identity management, and entitlement), to build a platform in which tenants can deploy their applications. **In a summary Stratos 2.0 Cartridge is a Cloud-aware platform environment, which extends legacy technologies into the Cloud and deliver Cloud benefits.**

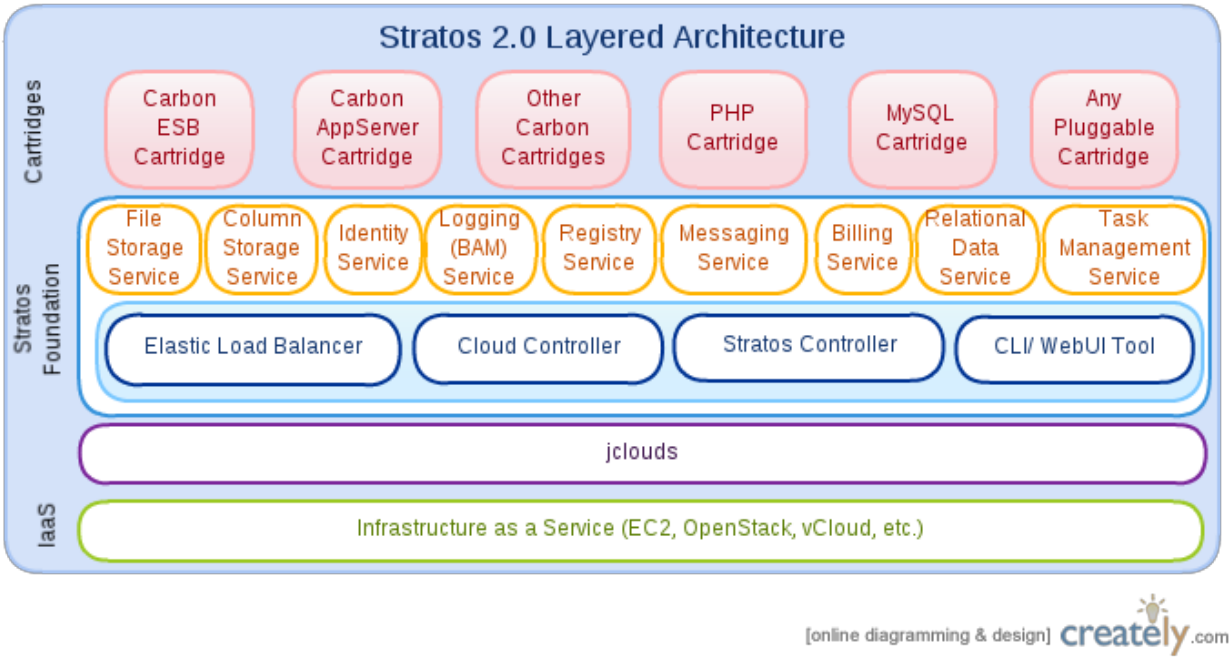


figure 1 - Stratos 2.0 Layered Architecture

Cartridges may wrap traditional, non-Cloud-aware application platform containers extending the traditional technology to the Cloud and provides elastic scalability, resource pooling, on-demand self-service, and consumption pricing. For example, WSO2 Stratos 2.0 ships with cartridges for PHP and MYSQL.

Stratos Operations teams may create custom cartridge types and host any application, container, or framework in a Stratos 2.0 Cloud. For example, a team may create a custom cartridge type to bring cloud characteristics to IBM Websphere Application Server, IBM WebSphere ESB, Oracle WebLogic, or JBoss SOA Platform.

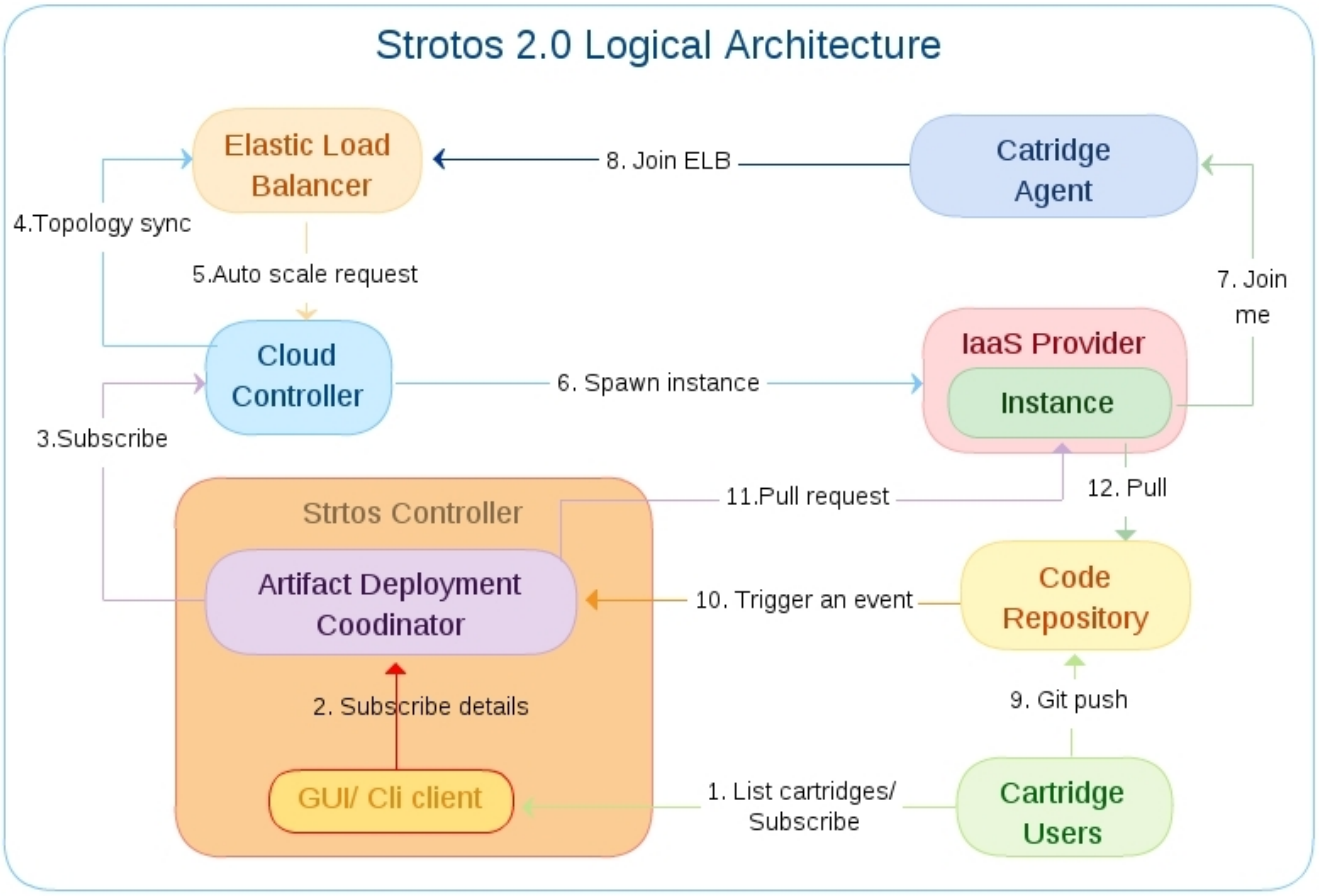


figure 2 - Stratos 2.0 Logical Architecture

Stratos 2.0 Foundation

Stratos 2.0 foundation layer, as the name suggests, build a strong foundation layer for a Cloud PaaS. It consists of some core components mandatory for any Stratos 2.0 cloud deployment.

Elastic Load Balancer(ELB)

- Load monitor
  - Probably co-located with ELB to start
  - Receive load events from various places (e.g. ELB)
  - Sends requests for load up/down to Cloud Controller
- Make autoscaling decisions

- Accept dynamic cluster domain registrations
- Accept static cluster domain registrations at startup reading loadbalancer.conf

Auto Scaling Decision Maker

Where does this autoscale decision making task reside?

The 'autoscaling decision making' task currently resides in WSO2 Elastic Load Balancer. Default implementation is

org.wso2.carbon.mediator.autoscale.lbautoscale.task.ServiceRequestsInFlightAutoscaler

What is the basis for autoscaling?

Current default implementation (**ServiceRequestsInFlightAutoscaler**) considers number of requests in-flight as the basis for making autoscaling decisions. We follow the paradigm; "scale up early and scale down slowly" in the default algorithm.

What are the decision making variables?

There are few of them and all of the vital ones are configurable using **loadbalancer.conf** file. (sample configuration files are provided at the end of this document [a][b].)

1. **autoscaler_task_interval (t)** - time period between two iterations of 'autoscaling decision making' task. When configuring this value, you are advised to consider the time 'that a service instance takes to join ELB'. This is in milliseconds and the default value is 30000ms.

2. **max_requests_per_second (Rps)** - number of requests, a service instance can withstand per a second. It is recommended that you calibrate this value for each service instance and may also for different scenarios. Ideal way to estimate this value could be by load testing a similar service instance. Default value is 100.

3. ***rounds_to_average (r)*** - an autoscaling decision will be made only after this much of iterations of 'autoscaling decision making' task. Default value is 10.

4. **alarming_upper_rate (AUR)** - without waiting till the service instance reach its maximum request capacity (alarming_upper_rate = 1), we scale the system up when it reaches the request capacity, corresponds to alarming_upper_rate. This value should be 0<AUR<=1 and default is 0.7.

5. **alarming_lower_rate (ALR)** - lower bound of the alarming rate, which gives us a hint; that we can think of scaling down the system. This value should be 0<ALR<=1 and default is 0.2.

6. **scale_down_factor (SDF)** - this factor is needed in order to make the scaling down process slow. We need to scale down slowly to reduce scaling down due to a false-positive event. This value should be 0<SDF<=1 and default is 0.25.

How is the number of requests in-flight gets calculated?

We keep track of the requests that come to Elastic Load Balancer (ELB) for various service clusters. For each incoming request, we add a token, against the relevant service cluster and when the message left ELB or got expired, we remove the corresponding token.

What are the decision making functions?

We always respect the minimum number of instances value and maximum number of instances value of service clusters. We make sure that the system always maintains the minimum number of service instance requirement and also system will not scale beyond its limit.

We calculate,

average requests in-flight for a particular service cluster (avg) =
total number of requests in-flight * (1/r)

Scaling up

number of maximum requests that a service instance can withstand over an autoscaler task interval (maxRpt) =
(Rps) * (t/1000) * (AUR)

then, we decide to scale up, if,

avg > maxRpt * (number of running instances of this service cluster)

Scaling down

imaginary lower bound value (minRpt) = (Rps) * (t/1000) * (ALR) * (SDF)

then, we decide to scale down, if,

avg < minRpt * (number of running instances of this service cluster - 1)

Can I plug my own implementation?

You can write your own Java implementation which implements **org.apache.synapse.task.Task and org.apache.synapse.ManagedLifecycle** interfaces. Wrap the implementation class to an OSGi bundle and deploy in WSO2 ELB. Then, point to that class from the **{ELB_HOME}/repository/conf/loadbalancer.conf** file's **loadbalancer** section as follows.

```
loadbalancer {
        ……..
        # autoscaling decision making task
        autoscaler_task  org.wso2.carbon.mediator.autoscale.lbautoscale.task.ServiceRequestsInFlightAutoscaler;
        ……..
}
```

Artifact Distribution Coordinator (ADC)

ADC (Embedded into Stratos Controller from Beta onwards) is responsible for distribution of Artifact. Artifacts can upload using git push. ADC a has listener service. The user configured Git repositories with Stratos 2.0 ADC can be pre configured to trigger notify above service in every git push. Github repositories can use WebHook URL to set this notifier. When a trigger event happens ADC lookup for topology and send notifications to appropriate cartridge instances. For carbon cartridges it will send DepSync cluster message notifications. Then instances do git pull and updates their artifacts.

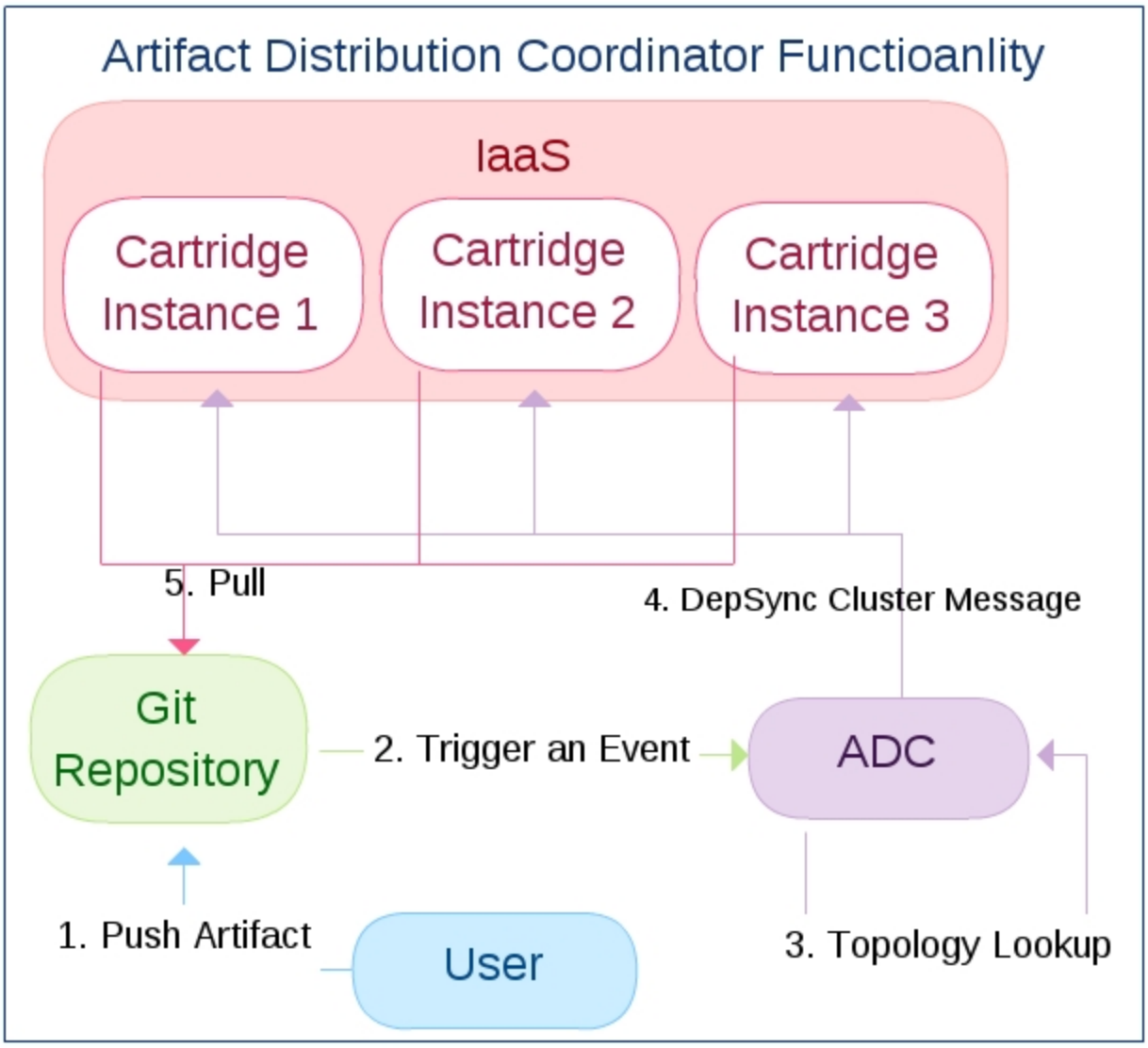


figure 3 - ADC Functionality Architecture

Cloud Controller

What is Cloud Controller?

Cloud Controller plays a vital role in Stratos 2.0 and given below is a list of its capabilities and duties.

WSO2 Cloud Controller,

- is acting as a bridge between application level and Infrastructure as a Service (IaaS) level via

Jclouds API.

- enables your system to scale across multiple IaaS providers.
- is the central location where the service topology resides.
- is responsible for sharing the up-to-date service topology among other Stratos 2.0 core services, periodically.
- supports hot update and deployment of its configuration files.
- has inbuilt support for AWS EC2 IaaS provider, Openstack Nova IaaS provider and latestly VMWare vCloud provider as well.
- enables you to cloud burst your system across multiple IaaS providers.
- allows you to plug an implementation of any IaaS provider supports by Jclouds, very easily.
- enables you to spawn new service instances, while associating a public IP automatically, in order to reduce the instance boot-up time.
- enables you to terminate an already started instance of a particular service cluster.
- can be configured to cover many scenarios, using its well-thought-out configuration files.

Service Interface of Cloud Controller?

WSO2 CC exposes a service which has the following interface.

```
/**
    * Registers the details of a newly created service cluster. This will override an already
    * present service cluster, if there is any. A service cluster is uniquely identified by its
    * domain and sub domain combination.
    *
    * @param domain
    *          service cluster domain
    * @param subDomain
    *          service cluster sub domain
    * @param tenantRange
    *                      tenant range eg: '1-10' or '2'
    * @param cartridgeType
    *          cartridge type of the new service. This should be an already registered cartridge
    *          type.
    * @param hostName
    *                      host name of this service instance
    * @param payload
    *          payload which will be passed to instance to be started. Payload shouldn't contain
    *          xml tags.
    * @return whether the registration is successful or not.
```

```
     *
     * @throws UnregisteredCartridgeException
     *             when the cartridge type requested by this service is
     *             not a registered one.
     * @throws CloudControllerException
     *                                 when the operation fails for internal reason.
     */
     public boolean registerService(String domain, String subDomain, String tenantRange, String
cartridgeType,
     String hostName, byte[] payload) throws UnregisteredCartridgeException,
CloudControllerException;

     /**
     * Calling this method will result in an instance startup, which is belong
     * to the provided service domain. This method is non-blocking, means we do not
     * wait till the instance is started up. Also note that the instance that is starting up
     * belongs to the group whose name is derived from its service domain, replacing <i>.</i>
     * by a hyphen (<i>-</i>).
     *
     * @param domainName
     *             service clustering domain of the instance to be started up.
     * @param sudDomainName
     *             service clustering sub domain of the instance to be started up.
     *             If this is null, the default value will be used. Default value is
     *             {@link Constants}.DEFAULT_SUB_DOMAIN.
     * @return public IP which is associated with the newly started instance.
     */
     public String startInstance(String domainName, String sudDomainName);

     /**
     * Calling this method will result in termination of an instance which is belong
     * to the provided service domain and sub domain.
     *
     * @param domainName
     *             service domain of the instance to be terminated.
     * @param sudDomainName
     *             service clustering sub domain of the instance to be started up.
     *             If this is null, the default value will be used. Default value is
     *             {@link Constants}.DEFAULT_SUB_DOMAIN.
     * @return whether an instance terminated successfully or not.
     */
     public boolean terminateInstance(String domainName, String subDomainName);
```

```
        /**
        * Calling this method will result in termination of the lastly spawned instance which is
        * belong to the provided service domain and sub domain.
        *
        * @param domainName
        *          service domain of the instance to be terminated.
        * @param sudDomainName
        *          service clustering sub domain of the instance to be started up.
        *          If this is null, the default value will be used. Default value is
        *          {@link Constants}.DEFAULT_SUB_DOMAIN.
        * @return whether the termination is successful or not.
        */
        public boolean terminateLastlySpawnedInstance(String domainName, String
subDomainName);

        /**
        * Calling this method will result in termination of all instances belong
        * to the provided service domain and sub domain.
        *
        * @param domainName
        *          service domain of the instance to be terminated.
        * @param sudDomainName
        *          service clustering sub domain of the instance to be started up.
        *          If this is null, the default value will be used. Default value is
        *          {@link Constants}.DEFAULT_SUB_DOMAIN.
        * @return whether an instance terminated successfully or not.
        */
        public boolean terminateAllInstances(String domainName, String subDomainName);

        /**
        * Calling this method will result in returning the pending instances
        * count of a particular domain.
        *
        * @param domainName
        *          service domain
        * @param sudDomainName
        *          service clustering sub domain of the instance to be started up.
        *          If this is null, the default value will be used. Default value is
        *          {@link Constants}.DEFAULT_SUB_DOMAIN.
        * @return number of pending instances for this domain. If no instances of this
        *          domain is present, this will return zero.
        */
        public int getPendingInstanceCount(String domainName, String subDomainName);
```

```
        /**
        * Calling this method will result in returning the types of {@link Cartridge}s
        * registered in Cloud Controller.
        *
        * @return String array containing types of registered {@link Cartridge}s.
        */
        public String[] getRegisteredCartridges();

        /**
        * This method will return the information regarding the given cartridge, if present.
        * Else this will return <code>null</code>.
        *
        * @param cartridgeType
        *          type of the cartridge.
        * @return {@link CartridgeInfo} of the given cartridge type or <code>null</code>.
        * @throws UnregisteredCartridgeException if there is no registered cartridge with this type.
        */
        public CartridgeInfo getCartridgeInfo(String cartridgeType) throws
UnregisteredCartridgeException;

        /**
        * Creates a key pair in all IaaSes that are configured for the given cartridge,
        * having the given name and public key.
        *
        * <p/>
        * <h4>Supported Formats</h4>
        * <ul>
        * <li>OpenSSH public key format (e.g., the format in ~/.ssh/authorized_keys)</li>
        * <li>Base64 encoded DER format</li>
        * <li>SSH public key file format as specified in RFC4716</li>
        * </ul>
        * DSA keys are not supported. Make sure your key generator is set up to create RSA keys.
        * <p/>
        * Supported lengths: 1024, 2048, and 4096.
        * <p/>
        *
        * @param cartridgeType
        *          type of the cartridge. Note this cartridge type should be already
        *          registered one.
        * @param keyPairName
        *          name of the key pair which is going to get created in IaaSes.
        * @param publicKey
```

```
*           The public key.
*
*/
public boolean createKeyPairFromPublicKey(String cartridgeType, String keyPairName,
String publicKey);
```

What are the configuration files used by Cloud Controller and where are they reside?

In a fresh Cloud Controller pack, you have only the main configuration file which is named as **"cloud-controller.xml"** and resides in **${WSO2-CC}/repository/conf/**. This file mainly consists 3 parts.

1. **BAM data publisher section**

This section contains the information related to BAM data publisher of Cloud Controller, such as BAM server information, data publishing task interval etc. You can disable BAM data publisher by setting ***enable*** attribute to ***false***.

```
<dataPublisher enable="true">
        <!-- BAM Server Info - default values are 'admin' and 'admin'
                Optional element. -->
        <bamServer>
        <!-- BAM server URL should be specified in carbon.xml -->
                <adminUserName>admin</adminUserName>
                <adminPassword
svns:secretAlias="cloud.controller.bam.server.admin.password">admin</adminPassword>
        </bamServer>
        <!-- Default cron expression is '1 * * * * ? *' meaning 'first second of every minute'.
                Optional element. -->
        <cron>1 * * * * ? *</cron>
</dataPublisher>
```

2. **Topology sync section**

This section contains the information related to topology synchronization, mainly WSO2 Message Broker server information.

```
<topologySync enable="true">
        <!-- MB server info -->
        <mbServerUrl>localhost:5673</mbServerUrl>
        <cron>1 * * * * ? *</cron>
</topologySync>
```

3. **IaaS providers section**

Here, you can define the information of IaaS providers, that may or not dependent on the different service clusters. This section is provided to make our users' lives easy. If they want they can remove this section, but we recommend you to utilize this feature.

```
<!-- Specify the properties that are common to an IaaS here. This element
        is not necessary [0..1]. But you can use this section to avoid specifying
        same property over and over again. -->
 <iaasProviders>
        <!-- type attribute is a must and should be unique. name attribute is optional, and you can add
any string value -->
        <iaasProvider type="ec2" name="ec2 specific details">
                <!-- full qualified name of the IaaS implementation  -->
                <className>org.wso2.carbon.stratos.cloud.controller.iaases.AWSEC2Iaas</className>
                <!-- provider string -->
                <provider>aws-ec2</provider>
                <!-- identity and credentials of your IaaS account. We use secure vault here. If we
cannot find a value via secure vault, we will look for the text value of this element. -->
                <identity svns:secretAlias="cloud.controller.ec2.identity"></identity>
                <credential svns:secretAlias="cloud.controller.ec2.credential"></credential>
                <!-- these orders will be used when scaling up and down instances. 1 has the highest
        precedence -->
                <scaleUpOrder>1</scaleUpOrder>
                <scaleDownOrder>2</scaleDownOrder>
                <!-- you can define any required property here. -->
                <property name="jclouds.ec2.ami-query"
value="owner-id=XXXX-XXX;state=available;image-type=machine"/>
                <property name="availabilityZone" value="us-east-1c"/>
                <property name="securityGroups" value="default"/>
                <property name="instanceType" value="m1.large"/>
                <property name="keyPair" value="nirmal-key"/>
                 <imageId>us-east-1/ami-XXXXX</imageId>
        </iaasProvider>
        <iaasProvider type="openstack" name="openstack specific details">
                <className>org.wso2.carbon.stratos.cloud.controller.iaases.OpenstackNovaIaas</className>
                 <provider>openstack-nova</provider>
                 <identity svns:secretAlias="cloud.controller.openstack.identity"></identity>
                 <credential svns:secretAlias="cloud.controller.openstack.credential"></credential>
                 <property name="jclouds.endpoint" value="http://192.168.16.20:5000/" />
```

```
                <property name="jclouds.openstack-nova.auto-create-floating-ips" value="false"/>
                <property name="jclouds.api-version" value="2.0/" />
                <scaleUpOrder>2</scaleUpOrder>
                <scaleDownOrder>3</scaleDownOrder>
                <property name="X" value="x" />
                <property name="Y" value="y" />
                <imageId>nova/dab37f0e-cf6f-4812-86fc-XXXXXXXXXXXX</imageId>
        </iaasProvider>
</iaasProviders>
```

How does the architecture look like?

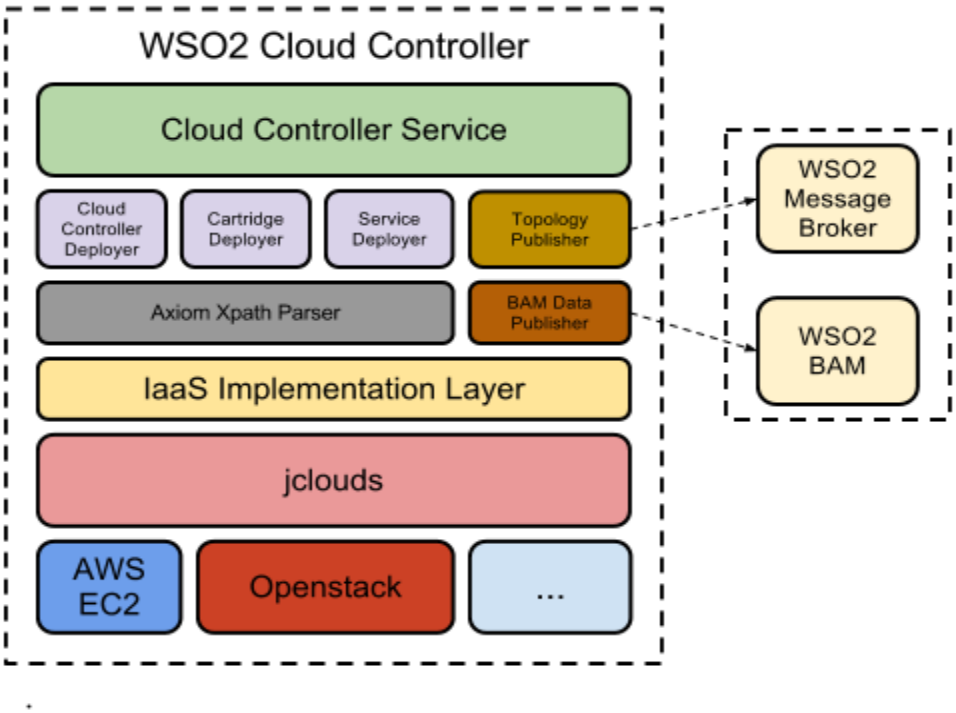


figure 3 - Cloud Controller Architecture

Above diagram depicts the high level architecture of the WSO2 Cloud Controller. WSO2 Cloud Controller is a WSO2 Carbon based server and supports the features mentioned at "What is Cloud Controller?" section of this document, through its number of architectural components. Those are briefed as below.

Cloud Controller Service

The service interface is defined in the section "Service Interface of Cloud Controller". This is the main entry point for a Cloud Controller service consumer. Currently this is a SOAP API and we might consider the possibility of providing a REST API, in a future Stratos 2 release.

Cloud Controller Deployer

This is an Axis2 deployer responsible for deploying/updating the "**cloud-controller.xml**" file defined in the section What are the configuration files used by Cloud Controller and where are they reside?.

Cartridge Deployer

This is an Axis2 deployer responsible for deploying/updating the cartridge definition files reside in "**${WSO2-CC}/repository/deployment/server/cartridges/"** folder.

You should define the Cartridges that the system going to support, in Cloud Controller. This can be done easily, by dropping a XML file which defines your Cartridge, into the "**${WSO2-CC}/repository/deployment/server/cartridges/"** folder. You can do this either before starting the server or while the server is running.

Cartridge XML file should have a unique file name and should adhere any of following two XML schemas.

1. Cartridge schema - you can define only one Cartridge in a file.
2. Cartridges schema - you can define multiple Cartridges in a single file.

NOTE: **There's a limitation in XML 1.0 specification and hence, you need to respect the sequential order of the elements of these configuration files. This limitation is rectified in XML 1.1.**

Sample Cartridge XML file (comments in the file explains each element):

```
<!-- You can have 1..n cartridge elements. -->
 <cartridge type="php" host="php.slive.com" provider="php" version="5">
        <!-- cartridge element can have 0..n properties, and they'll be overwritten by the properties specified under iaasProvider child elements of cartridge element. -->
         <property name="ss" value="slsls"/>
        <!-- Cartridge name -->
        <displayName>PHP</displayName>
        <!-- Cartridge description -->
        <description>a php cartridge</description>
        <!-- A cartridge element should add a reference to an existing IaaS provider (specified in the above &lt;iaasProviders&gt; section) or it can create a completely new IaaS Provider (which should have a unique "type" attribute. -->
         <iaasProvider type="openstack" >
              <imageId>nova/250cd0bb-96a3-4ce8-bec8-XXXXXXXXXX</imageId>
             <property name="keyPair" value="demo"/>
         </iaasProvider>
        <!-- This is required for the GIT repo creation. 'baseDir' attribute should point to a valid path in the service instance, and it's the place, where we will clone the GIT repo -->
        <deployment baseDir="xyz">
```

```
                <!-- Following directories will be created in the newly created GIT repo. -->
                <dir>abc</dir>
                <dir>abcdef</dir>
            </deployment>
            <!-- Following element is required, if this is a non-carbon cartridge. -->
            <portMapping>
                <http port="80" proxyPort="8280"/>
                <https port="443" proxyPort="8243"/>
            </portMapping>
            <!-- This element will be used to define application types of a cartridge that ca be given
a domain mapping. E.g. In WSO2 AppServer cartridge, app types are Axis2Services, Webapps,
JaxWebapps, and Jaggeryapps. -->
            <appTypes>
                <appType name="axis2services" />
                <appType name="services" />
                <appType name="webapps" appSpecificMapping="false"/>
            </appTypes>
        </cartridge>
```

Service Deployer

This is an Axis2 deployer responsible for deploying/updating the service definition files reside in **{WSO2-CC}/repository/deployment/server/services/** folder.

For Carbon services, you should define the Services that the system going to support, in Cloud Controller. But for non-carbon services, you don't need to be concerned, since relevant services will automatically get registered in Cloud Controller, at the time of your subscription to the Stratos-2.

This can be done easily, by dropping a XML file which defines the service configuration, into the **{WSO2-CC}/repository/deployment/server/services/** folder. You can do this either before starting the server or while the server is running.

Service XML file should have a unique file name and should adhere any of following two XML schemas.

1. Service schema - you can define only one Service in a file.
2. Services schema - you can define multiple Services in a single file.

NOTE: There's a limitation in XML 1.0 specification and hence, you need to respect the sequential order of the elements of these configuration files. This limitation is rectified in XML 1.1.

Sample Service XML file (comments in the file explains each element):

```
<service domain="nir" subDomain="mgt" tenantRange="*">
 <!-- reference to the Cartridge -->
 <cartridge type="php"/>
 <!-- host name of this service instance -->
 <host>nir.mgt</host>
 <!-- path to the payload file -->
 <payload>/wso2as-5.0.1-cloud-controller/resources/payload/nir-mgt.txt</payload>
</service>
```

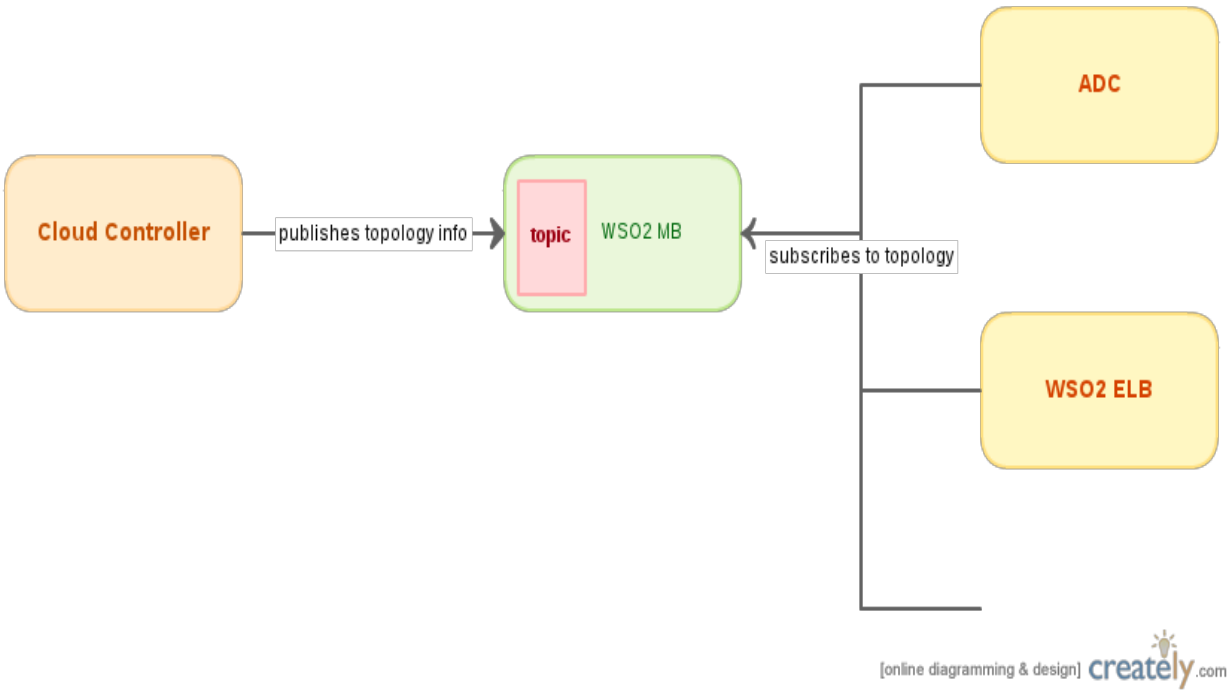
Axiom Xpath Parser

Axiom Xpath based parser is responsible for parsing all kinds of *.xml configuration files of Cloud Controller.

Topology Publisher

We keep the topology configuration in a central place - i.e. in Cloud Controller. What topology configuration means here, is the information about the different service clusters of your Cloud environment. As an example you might have an application service cluster and an ESB service cluster, and the corresponding topology configuration might look like this (in Nginx format).

But we can't survive by keeping this configuration only in Cloud Controller, because we need this configuration for other places like ELB, Artifact Distribution Coordinator (ADC) etc. Hence, it is Cloud Controller's duty to somehow sync this topology configuration it maintains, with others. Following image depicts how that is done.



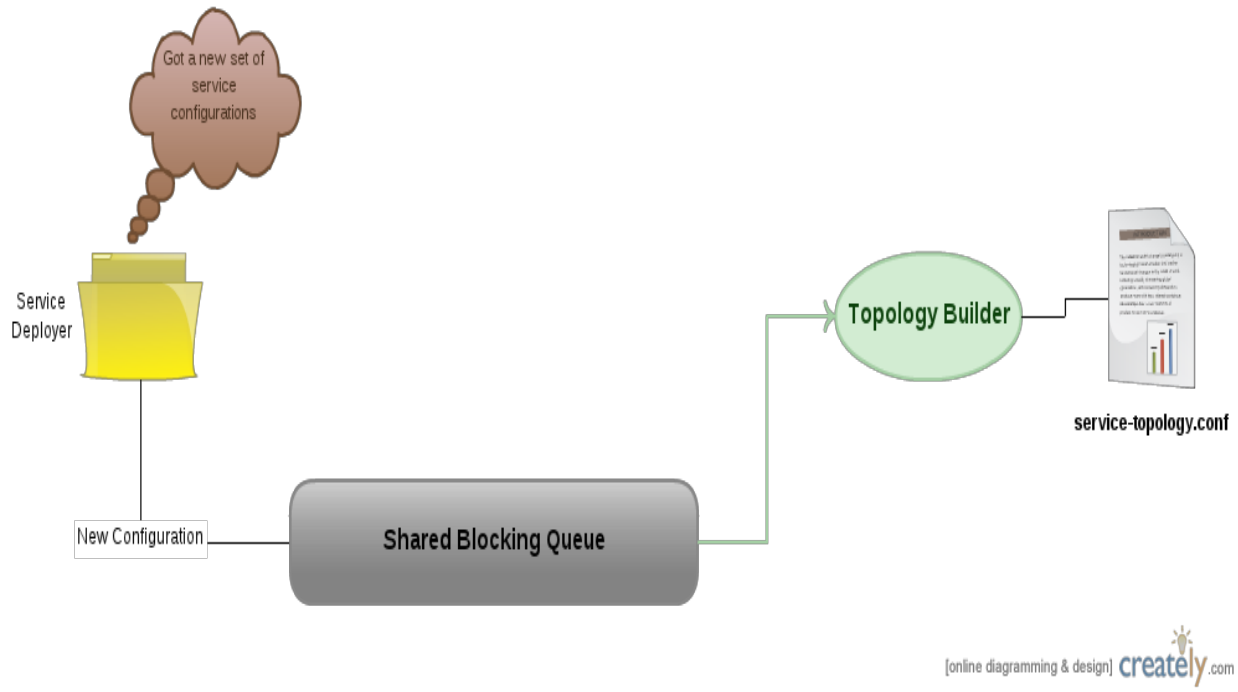
- Cloud Controller keeps building the topology configuration (since, it can be changed dynamically) and publishes it in a periodical manner (in addition to publishing upon a change) to a topic created in WSO2 Message Broker (Also known as MB which is embedded into CC).
  Note: **here, we publish the whole configuration since that will make sure, a restart of a subscriber of this topic won't have any issue in synching up.**

- Anyone who needs this topology configuration can subscribe to the (relevant) topic and get synced.

- Subscribers should ideally generate the difference of configuration (between what it received and what it has) and act upon the diff (if there's any).

How does the topology get built?

As mentioned earlier, topology configuration can be changed dynamically (Added topology cannot be removed (without restarting servers), but can update an existing one) and we need to publish an up-to-date version to the topic.



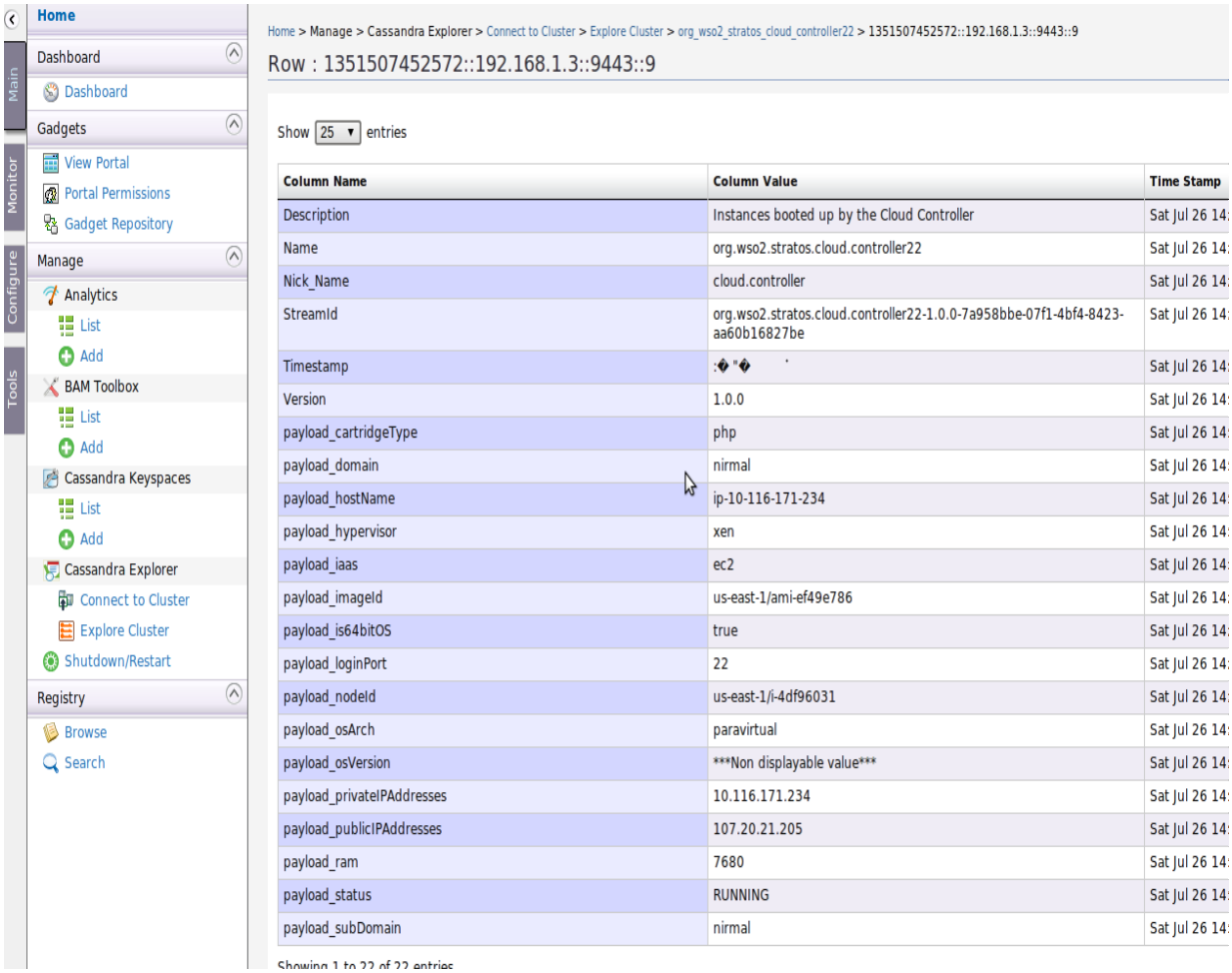
We used producer-consumer pattern here. Cloud Controller's Service Deployer (producer) will track the changes and put them into a shared blocking queue (which is shared with consumer). Topology Builder (consumer) grab changes from the queue and add/update the topology configuration ("**${WSO2-CC}/repository/conf/service-topology.conf"**) as shown in the above diagram.

BAM Data Publisher

We publish the information of the instances that are started up by the Cloud Controller to WSO2 BAM server. BAM data publisher is intelligent enough to publish events related to 'a newly spawned instance' or 'a state changed existing instance'.

Following screenshot shows a view of published data, from BAM's Cassandra explorer.

NOTE: You need to configure BAM server URL in the {WSO2-CC}/repository/conf/carbon.xml file, in addition to BAM data publisher configurations in {WSO2-CC}/repository/conf/cloud-controller.xml file.



IaaS Implementation Layer

This layer contains the IaaS implementations. All these implementations are extensions of Cloud Controller's Iaas abstract class. This class is added here and it resides in Cloud Controller's **org.wso2.carbon.stratos.cloud.controller.interfaces** package.

jclouds

This layer is the jclouds API. jclouds is an open source library that helps you get started in the cloud. The jclouds API gives you the freedom to use portable abstractions or cloud-specific features.

Does Cloud Controller supports hot update and hot deployment of its configuration files?

Yes, and that makes Cloud Controller much easier to use. Let's say you have updated a configuration file (abc.xml) and it has some configuration issue. But, you don't need to worry, we'll let you know the error and will back-up your configuration file by adding a suffix (.back) to the original file name (abc.xml.back).

What IaaS providers you support by default?

We support AWS EC2 and Openstack IaaS providers by default. But theoretically we can provide support for any IaaS that jclouds supports.

Can it support IaaS providers other than ones supported by Jclouds?

No, we only support IaaS providers which are/will support by jclouds.

How easy it is to provide support for a new IaaS provider?

You are few steps away, watch closely!

1. Your IaaS provider implementation (say **org.wso2.carbon.stratos.iaas.VcloudIaas.java**) should extend the Iaas abstract class provided by Cloud Controller.

2. You should wrap the implementation in an OSGi bundle and it should be a fragment bundle of cloud controller. You can do this via your bundle's pom file.

    Add following line as a configuration instruction of maven bundle plugin:

    <Fragment-Host>{symbolic-name-of-cloud-controller}</Fragment-Host>

    ```
    {symbolic-name-of-cloud-controller} = org.wso2.carbon.stratos.cloud.controller
    ```

3. Also, it is pretty obvious that you need to add 'cloud-controller' component as a dependency in your bundle's pom file.

4. Next, you can drop the built bundle to **{WSO2-CC}/repository/components/dropins/**.

5. Finally, in Cloud Controller's **cloud-controller.xml**, you need to define the IaasProvider you are going to add and your implementation class.

eg:

```
<iaasProvider type="vcloud" name="vcloud specific details">
        <className>org.wso2.carbon.stratos.iaas.VcloudIaas</className>
        <provider>vcloud</provider>
        <identity svns:secretAlias="cloud.controller.vcloud.identity"/>
        <credential svns:secretAlias="cloud.controller.vcloud.credential"/>
        <scaleUpOrder>1</scaleUpOrder>
        <scaleDownOrder>2</scaleDownOrder>
        <property name="securityGroups" value="default"/>
        <property name="instanceType" value="m1.large"/>
        <property name="keyPair" value="abc-key"/>
        <imageId>us-east-1/ami-abcdef123</imageId>
</iaasProvider>
```

Now you can start using the newly added IaaS, in your Cartridge!

Cartridges

Single-Tenant cartridges

With single-tenant containers and frameworks, Stratos Cartridges provide process-level isolation and instance-level dedicated tenancy. A tenant maps to one or more cartridge instances, and multiple tenants are not hosted on a single cartridge instance (unlike multi-tenant shared containers where teams may define multiple tenants per cartridge). Stratos load-balancing and security mechanisms partition tenants based on network Uniform Resource Locator (URL) and hostname identifiers.

When single-tenant containers and frameworks are hosted within a Stratos Cartridge, teams configure elastic scalability limits by specifying a pre-defined maximum process instance limit per tenant, and the cartridge environment operates in a manner similar to application server clustering. When a cartridge instance reservation is activated, the activation decreases the available resource pool for other tenants.

Multi-Tenant cartridges

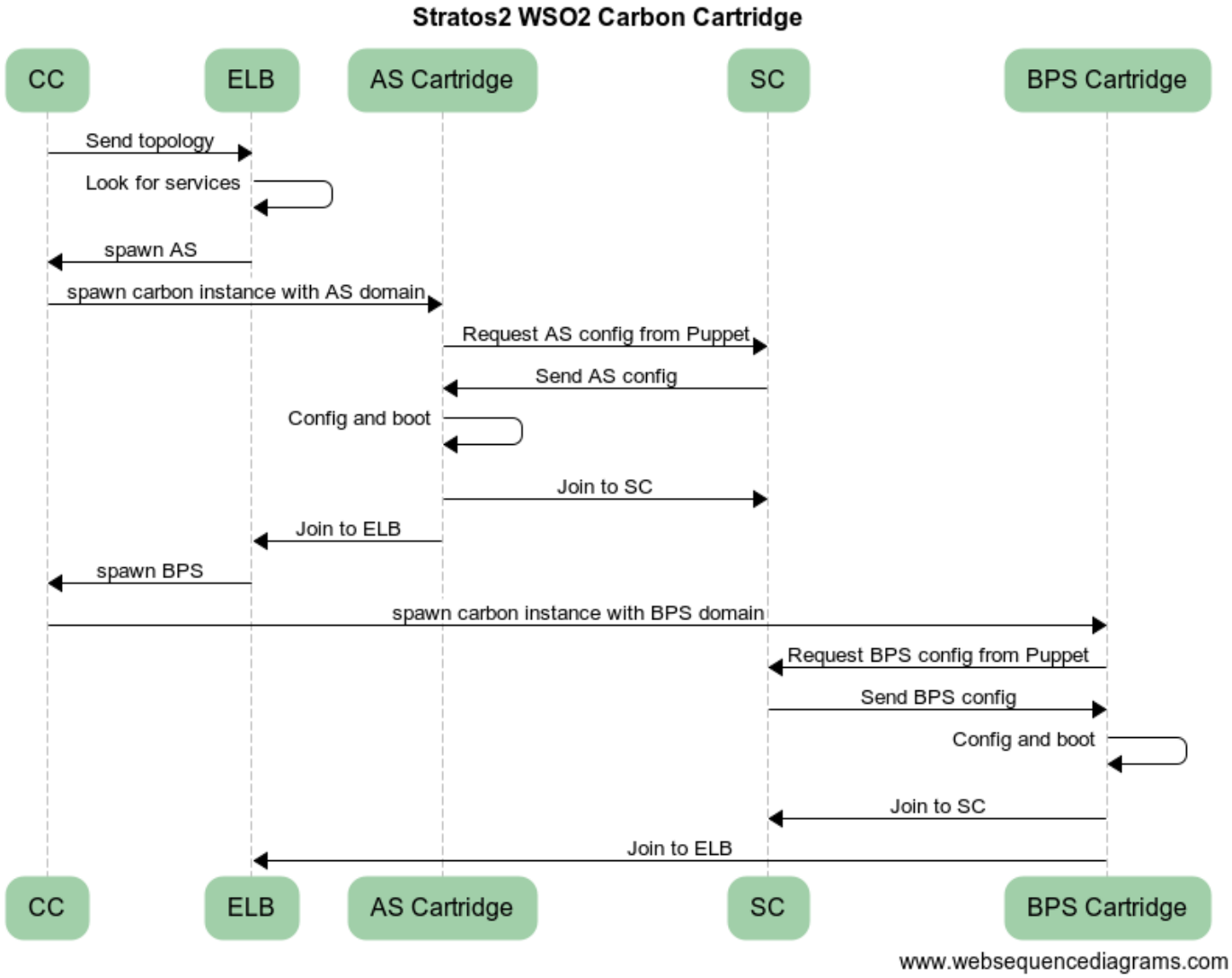
Stratos Cartridges hosting WSO2 multi-tenant middleware services delivers multi-tenancy within each cartridge instance. Within the cartridge instance, tenant traffic is securely directed to isolated

application code. Cartridge instance activations increase the available resource pool for all tenants associated with the service partition. For more information on multi-tenant, shared container technology, architecture, and benefits, refer [2].

WSO2 Stratos ships with pre-defined Stratos Cartridges for all multi-tenant WSO2 middleware product and rapidly reduces the time required to create a Cloud environment. Stratos Cartridges host multi-tenant WSO2 middleware platform servers and deliver integration services, SOA services, application development services, governance services, identity services, presentation services, and business process execution services.

Puppet based WSO2 Carbon Cartridges

The cartridges for WSO2 Carbon based products are configured using Puppet. For example, when the Cloud Controller requests an instance with “AS” (Application Server) domain, a “Carbon base cartridge” will be spawned and the Puppet will configure it for the AS specific configuration. If an instance of BPS domain was requested, the same “Carbon base cartridge” will be spawned with BPS configurations from Puppet.



User Roles

Cartridge Developer

(e.g. In StratosLive case WSO2 PHP cartridge developers)
Creates image for different IaaS

Cartridge Deployer

(e.g. In StratosLive case WSO2 DevOps team)
Registers Cartridge with Stratos to create Stratos Cartridges

Cartridge Subscriber

(Eg. abc.com DevOps team/ Tenant admin)

Subscribes to the cartridge

- with other resource such as a persistent file system, a DB with scaling parameters
- Subscribe details will be kept in Stratos DB
- has the option of asking “create cartridge instance now” (vs. create on demand after apps are loaded / first request comes in). *
- This is because there may be embedded applications in the image.

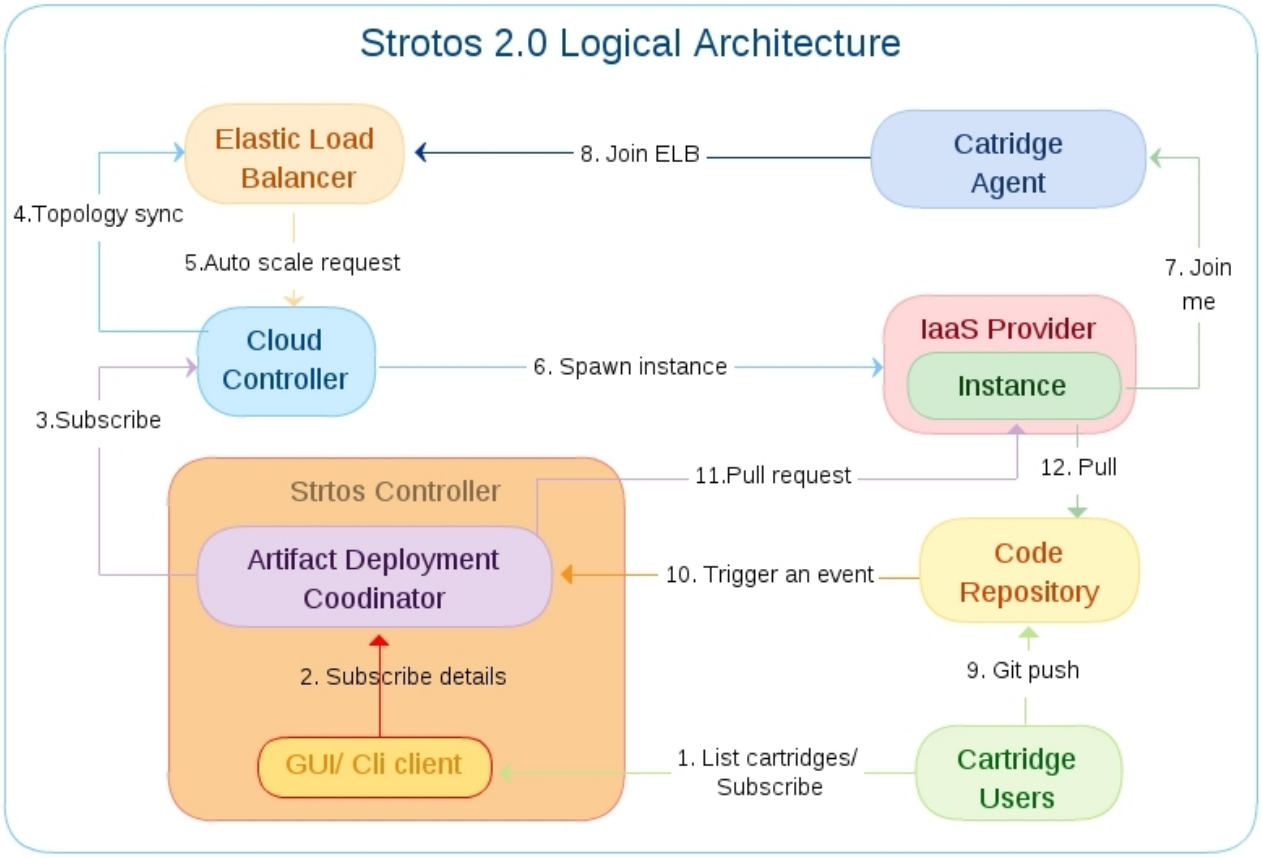
Upload applications to the cartridge

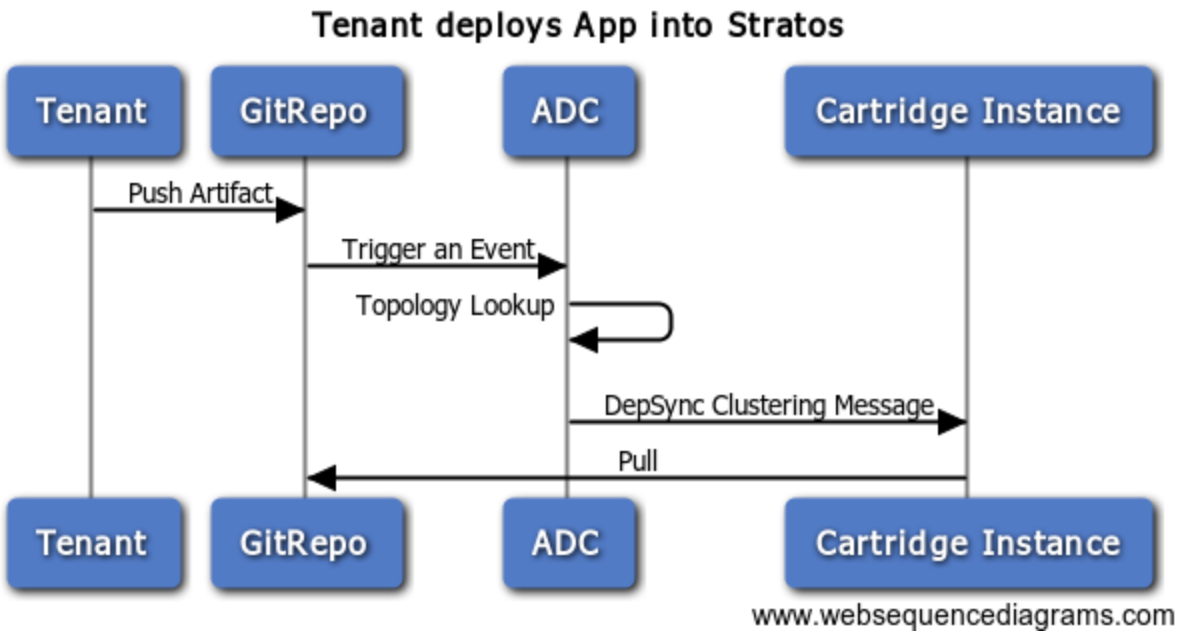
This step is optional step when subscribed with the create cartridge instance now option. When first cartridge application is uploaded, Cartridge instance will be created if it is not already created.

Cartridge Users

Eg. (Eg. abc.com tenant users)

- Use deployed apps
- Upload apps to cartridge(optional)


Strotos 2.0 Logical Architecture
Elastic Load Balancer
8. Join ELB
Catridge Agent
4.Topology sync
5.Auto scale request
7. Join me
Cloud Controller
6. Spawn instance
IaaS Provider
Instance
3.Subscribe
11.Pull request
12. Pull
Strtos Controller
Artifact Deployment Coodinator
10. Trigger an event
Code Repository
2. Subscribe details
9. Git push
GUI/ Cli client
1. List cartridges/ Subscribe
Cartridge Users

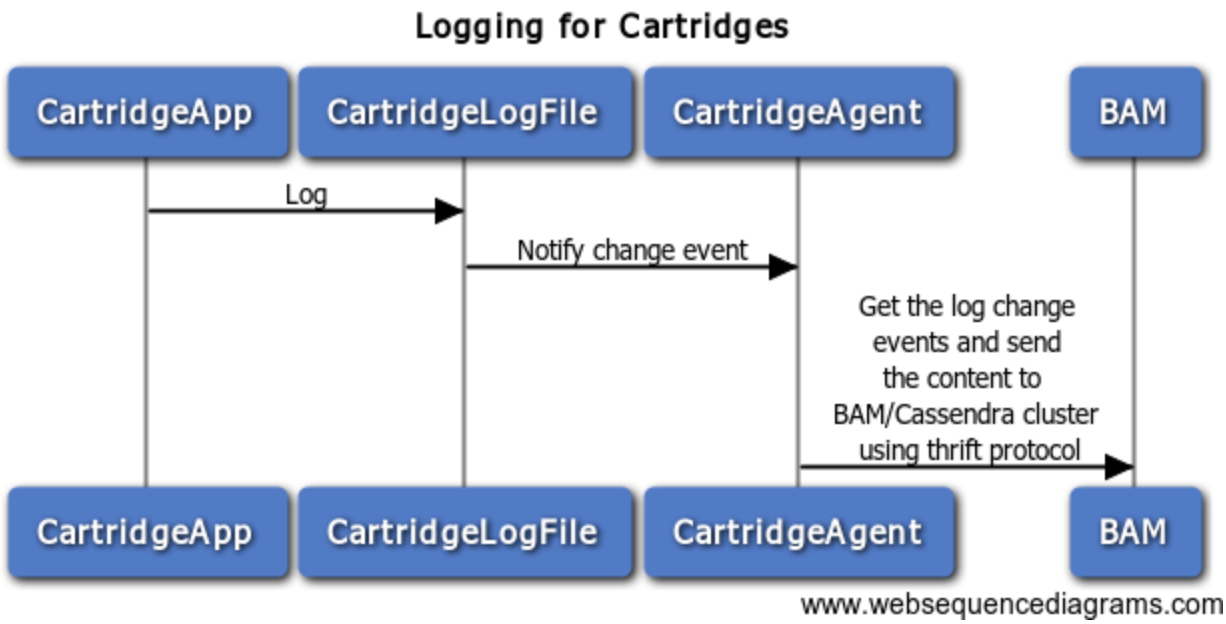

Tenant deploys App into Stratos
Tenant
GitRepo
ADC
Cartridge Instance
Push Artifact
Trigger an Event
Topology Lookup
DepSync Clustering Message
Pull
Tenant
GitRepo
ADC
Cartridge Instance
www.websequencediagrams.com

title: Tenant deploys App into Stratos

```
Tenant -> GitRepo: Push Artifact

GitRepo -> ADC : Trigger an Event

ADC -> ADC : Topology Lookup

ADC -> Cartridge Instance: DepSync Clustering Message

Cartridge Instance -> GitRepo: Pull
```

Cartridge Agent

Logging




```
title Logging for Cartridges

CartridgeApp -> CartridgeLogFile: Log
```

CartridgeLogFile -> CartridgeAgent : Notify change event

CartridgeAgent -> BAM : Get the log change\n events and send \nthe content to \nBAM/Cassendra cluster\n using thrift protocol

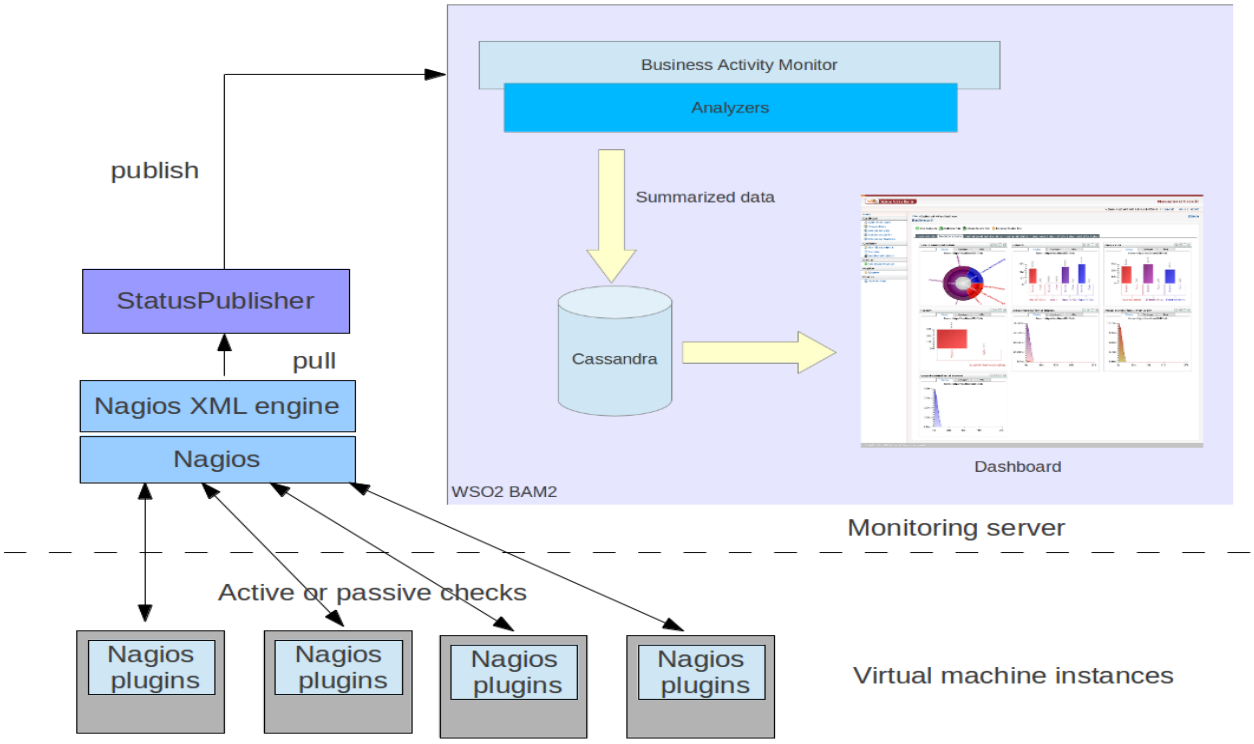
Health Monitoring

Stratos 2.0 health monitoring falls into two categories.
1) Service Level Health Monitoring
2) System Level Health Monitoring

Service Level Health Monitoring will periodically check the health of the services up and running in Stratos. An operations team member with appropriate rights entitled by super admin can use this health monitoring facility to monitor all stratos services. This component is not yet available for tenants to check their own services health.

System level health monitoring will periodically check the cartridge instances for health status. An operations team member with appropriate rights entitled by super admin can use system level health monitoring facility to monitor all stratos cartridges. Also a cartridge subscriber can use system level health monitoring to check the health status of his own cartridges.

Stratos 2.0 System Level Health Monitoring



Nagios

Nagios core (GNU GPL licensed) will be used to collect detailed resource/health status of instances.

Active and passive checks

Active checks are executed:

At regular intervals, On-demand as needed

Passive checks are used:

Asynchronous in nature and cannot be monitored effectively by polling their status on a regularly scheduled basis

Status Publisher

StatusPublisher will be used to pull status information from nagios as XML and publish it in to BAM as events.

Visualization

Collected status information will be analysed and visualized

1. as historical information over customizable periods
2. as states of monitoring IaaS instances, through alert and notifications

Note: Slight change in the architecture

without using Nagios XML engine; mk_livestatus http://mathias-kettner.de/checkmk_livestatus.html will be used to pull data from the Nagios server.this change was made due to several latency, effcient reasons with respect to Nagios XML engine.

User Stories

- A Cloud Administrator or User need to receive notifications for cloud resources life cycle events in a timely manner so that he can respond to emergencies.
- A Cloud Administrator or User need to support on-going health status and notifications for cloud resources that he is managing.
- As Cloud Administrator or User need to support, analyze; performance, resource usage, resource consumption, etc.
- A Cloud Administrator or User want to configure policies to automatically address notifications within his cloud infrastructure.

Custom Domain Mapping

User can add an own domain per cartridge using domain mapping functionality. At the time of subscription tenant does not have to provide own domain information. After the subscription, he can add the domain mapping through cli tool or GUI.

Say the tenant **abc.com**s own domain is, **http://mysite.com,**

Then she can register a domain mapping per cartridge,
mysite.com -> Cartridge_alias (eg. **abcphp**)

After that he need to configure DNS with a CNAME record pointing the his domain to the published WSO2 hostname(eg: php.slive.com).

Security for Cartridge Applications

Cartridge applications can be required to authenticated against a user store. We can use WSO2 Identity Server as it is already included in the WSO2 server list. It might require a sample for different types of cartridge applications on how they can get the use of WSO2 Identity Server.

Stratos 2.0

The topmost layer of Stratos2 consist of Stratos Services. These services run inside a cartridge. For example we have ESB cartridge, AS cartridges, PHP Cartridges etc.

Usage Scenarios

Scenario 1: WSO2 Public PaaS

WSO2 PHP Cartridge Developer team, develop the PHP cartridge image. He may start from a base image provided by a cloud operating system provider(Eg. Ubuntu UEC [1]), and install software packages, for eg, PHP, Apache HTTP Server etc. Those software installation can be done by using set of scripts (available at STRATOS2_DOWNLOAD_PACK/tools/cartridge_create folder) provided by WSO2. After creating the cartridge he will create **php.xml** cartridge configuration file.

WSO2 DevOps team register it with the WSO2 StratosLive environment. They will need to update three configuration files.
1)<stratos deployment>/conf/stratos-controller.conf
2)<stratos deployment>/conf/cartridges/php/cartridge.conf
3)<stratos deployment>/deployment/server/php/stratos-controller.conf

abc.com wants to host abc.com php site in Stratos Live cloud. abc.com DevOps team subscribe to the PHP carrtidge in Stratos Live cloud.

abc.com DevOps team upload the abc.com site PHP application into the cartridge. At the time of deploying he has to mention some parameters like scaling parameters, min and max instance count, resources such as FS or DB etc. When the application is uploaded a cartridge instance is created for him.

Now abc.com users can access the abc.com site for their needs.

Scenario 2: Private PaaS

Now consider a scenario where abc.com is a Stratos customer who run their own private stratos cloud. Also assume that they need to create their own cartridge(say MongoDB).
In this scenario they install Stratos cloud and follow the guidelines in **Stratos2 Cartridge Development Guide** shipped along with this document, to create the cartridge.

Annex

Properties defined in the defaults section.

```
loadbalancer {
        # minimum number of load balancer instances
        instances           1;
        # whether autoscaling should be enabled or not.
        enable_autoscaler   true;
        #please use this whenever url-mapping is used through LB.
        #size_of_cache              100;
        # autoscaling decision making task
        autoscaler_task
org.wso2.carbon.mediator.autoscale.lbautoscale.task.ServiceRequestsInFlightAutoscaler;
        # End point reference of the Autoscaler Service
        autoscaler_service_epr <autoscaler_service_epr>;
        # interval between two task executions in milliseconds
        autoscaler_task_interval 30000;
        # after an instance booted up, task will wait maximum till this much of time and let the server started up
        server_startup_delay 60000; #default will be 60000ms
        # session time out
```

```
        session_timeout 90000;
        # enable fail over
        fail_over true;
}

# services' details which are fronted by this WSO2 Elastic Load Balancer
services {
        # default parameter values to be used in all services
        defaults {
        # minimum number of service instances required. WSO2 ELB will make sure that this much of instances
        # are maintained in the system all the time, of course only when autoscaling is enabled.
        min_app_instances       1;
        # maximum number of service instances that will be load balanced by this ELB.
        max_app_instances       3;
        max_requests_per_second   5;
        rounds_to_average       2;
        alarming_upper_rate 0.7;
        alarming_lower_rate 0.2;
        scale_down_factor 0.25;
        message_expiry_time     60000;
        }


        appserver {
        hosts   appserver.cloud-test.wso2.com;
        domains  {
                3.appserver.domain {
                tenant_range    *;
                min_app_instances       0;
                }
        }
        }

}
```

Properties defined within the service element

```
loadbalancer {
        # minimum number of load balancer instances
        instances           1;
        # whether autoscaling should be enabled or not.
        enable_autoscaler   true;
        #please use this whenever url-mapping is used through LB.
        #size_of_cache              100;
        # autoscaling decision making task
        autoscaler_task
org.wso2.carbon.mediator.autoscale.lbautoscale.task.ServiceRequestsInFlightAutoscaler;
        # End point reference of the Autoscaler Service
        autoscaler_service_epr <autoscaler_service_epr>;
        # interval between two task executions in milliseconds
        autoscaler_task_interval 30000;
        # after an instance booted up, task will wait maximum till this much of time and let the server started up
        server_startup_delay 60000; #default will be 60000ms
        # session time out
        session_timeout 90000;
        # enable fail over
        fail_over true;
}

# services' details which are fronted by this WSO2 Elastic Load Balancer
services {
        # default parameter values to be used in all services
        defaults {
                # minimum number of service instances required. WSO2 ELB will make sure that this much of
        instances
                # are maintained in the system all the time, of course only when autoscaling is enabled.
                min_app_instances       1;
                # maximum number of service instances that will be load balanced by this ELB.
                max_app_instances       3;
                max_requests_per_second   5;
                rounds_to_average       2;
                alarming_upper_rate 0.7;
```

```
            alarming_lower_rate 0.2;
            scale_down_factor 0.25;
            message_expiry_time         60000;
        }


        appserver {
            hosts      appserver.cloud-test.wso2.com;
            domains   {
                3.appserver.domain {
                    tenant_range       *;
                    min_app_instances           0;
                    max_requests_per_second   5;
                    alarming_upper_rate 0.6;
                    alarming_lower_rate 0.1;
                }
            }
        }

}
```

{WSO2-CC}/repository/conf/etc/cartridge.xsd

```
<xs:element name="cartridge">
        <xs:annotation>
        <xs:documentation>You can have 1..n cartridge elements.</xs:documentation>
        </xs:annotation>
        <xs:complexType>
         <xs:choice maxOccurs="unbounded">
             <xs:element name="property" maxOccurs="unbounded"
                 minOccurs="0">
                 <xs:annotation>
```

```
                <xs:documentation>
                        cartridge element can have 0..n properties, and
                        they'll be overwritten by the properties
                        specified under iaasProvider child elements of
                        cartridge element.
                </xs:documentation>
        </xs:annotation>
        <xs:complexType>
                <xs:simpleContent>
                        <xs:extension base="xs:string">
                                <xs:attribute type="xs:string" name="name" />
                                <xs:attribute type="xs:string" name="value" />
                        </xs:extension>
                </xs:simpleContent>
        </xs:complexType>
</xs:element>
<xs:element name="displayName" maxOccurs="1" minOccurs="0" />
<xs:element name="description" maxOccurs="1" minOccurs="0" />
<xs:element name="iaasProvider" maxOccurs="unbounded"
        minOccurs="1">
        <xs:annotation>
                <xs:documentation>
                        A cartridge element should add a reference to an
                        existing IaaS provider (specified in the above
                        &amp;lt;iaasProviders&amp;gt; section) or it can
                        create a completely new IaaS Provider (which
                        should have a unique "type" attribute.
                </xs:documentation>
        </xs:annotation>
        <xs:complexType>
                <xs:choice maxOccurs="unbounded">
                        <xs:element type="xs:string" name="imageId"
                                minOccurs="0" maxOccurs="1" />
                        <xs:element name="property"
                                maxOccurs="unbounded" minOccurs="0">
                                <xs:complexType>
                                        <xs:simpleContent>
                                                <xs:extension base="xs:string">
                                                        <xs:attribute type="xs:string"
```

```
                                                                name="name" />
                                                        <xs:attribute type="xs:string"
                                                                name="value" />
                                                    </xs:extension>
                                                </xs:simpleContent>
                                            </xs:complexType>
                                        </xs:element>
                                    </xs:choice>
                                    <xs:attribute type="xs:string" name="type" />
                                </xs:complexType>
                            </xs:element>
                            <xs:element name="deployment" maxOccurs="1" minOccurs="1">
                                <xs:complexType>
                                    <xs:choice maxOccurs="unbounded">
                                        <xs:element name="dir" maxOccurs="unbounded"
                                            minOccurs="0" type="xs:string">
                                        </xs:element>
                                    </xs:choice>
                                    <xs:attribute name="baseDir" type="xs:string">
                                    </xs:attribute>
                                </xs:complexType>
                            </xs:element>
                            <xs:element name="portMapping" maxOccurs="1"
                                minOccurs="0">
                                <xs:complexType>
                                    <xs:choice maxOccurs="unbounded">
                                        <xs:element name="http" maxOccurs="1"
                                            minOccurs="1">
                                            <xs:complexType>
                                                <xs:attribute name="port"
                                                    type="xs:string" />
                                                <xs:attribute name="proxyPort"
                                                    type="xs:string" />
                                            </xs:complexType>
                                        </xs:element>
                                        <xs:element name="https" maxOccurs="1"
                                            minOccurs="0">
                                            <xs:complexType>
                                                <xs:attribute name="port"
```

```
                                                    type="xs:string" />
                                                <xs:attribute name="proxyPort"
                                                    type="xs:string" />
                                            </xs:complexType>
                                        </xs:element>
                                    </xs:choice>
                                </xs:complexType>
                            </xs:element>
                            <xs:element name="appTypes" maxOccurs="1"
                                minOccurs="1">
                                <xs:complexType>
                                    <xs:choice maxOccurs="unbounded">
                                        <xs:element name="appType" maxOccurs="unbounded"
                                            minOccurs="1">
                                            <xs:complexType>
                            <xs:attribute name="name" type="xs:string" use="required" />
                            <xs:attribute name="appSpecificMapping"
                                                    type="xs:string" />
                                            </xs:complexType>
                                        </xs:element>
                                    </xs:choice>
                                </xs:complexType>
                            </xs:element>
                        </xs:choice>
                        <xs:attribute type="xs:string" name="type" />
                        <xs:attribute type="xs:string" name="host" />
                        <xs:attribute type="xs:string" name="provider" />
                        <xs:attribute type="xs:string" name="version" />
                        <xs:attribute type="xs:boolean" name="multiTenant" />
                    </xs:complexType>
</xs:element>
```

{WSO2-CC}/repository/conf/etc/cartridges.xsd

```
<xs:element name="cartridges" >
        <xs:annotation>
        <xs:documentation>Use below section to specify properties that are needed in order to start Cartridges.</xs:documentation>
        </xs:annotation>
        <xs:complexType>
        <xs:sequence>
        <xs:choice maxOccurs="unbounded">
        <xs:element name="cartridge" maxOccurs="unbounded" minOccurs="1">
        <xs:annotation>
        <xs:documentation>You can have 1..n cartridge elements.</xs:documentation>
        </xs:annotation>
        <xs:complexType>
        <xs:choice maxOccurs="unbounded">
                <xs:element name="property" maxOccurs="unbounded"
                        minOccurs="0">
                        <xs:annotation>
                                <xs:documentation>
                                        cartridge element can have 0..n properties,
                                        and they'll be overwritten by the properties
                                        specified under iaasProvider child elements
                                        of cartridge element.
                                </xs:documentation>
                        </xs:annotation>
                        <xs:complexType>
                                <xs:simpleContent>
                                        <xs:extension base="xs:string">
                                                <xs:attribute type="xs:string"
                                                        name="name" />
                                                <xs:attribute type="xs:string"
                                                        name="value" />
                                        </xs:extension>
                                </xs:simpleContent>
                        </xs:complexType>
                </xs:element>
                <xs:element name="displayName" maxOccurs="1" minOccurs="0" />
                <xs:element name="description" maxOccurs="1"
                        minOccurs="0" />
                <xs:element name="iaasProvider" maxOccurs="unbounded"
```

```
                        minOccurs="1">
                        <xs:annotation>
                                <xs:documentation>
                                        A cartridge element should add a reference
                                        to an existing IaaS provider (specified in
                                        the above &amp;lt;iaasProviders&amp;gt;
                                        section) or it can create a completely new
                                        IaaS Provider (which should have a unique
                                        "type" attribute.
                                </xs:documentation>
                        </xs:annotation>
                        <xs:complexType>
                                <xs:choice maxOccurs="unbounded">
                                        <xs:element type="xs:string" name="imageId"
                                                minOccurs="0" maxOccurs="1" />
                                        <xs:element name="property"
                                                maxOccurs="unbounded" minOccurs="0">
                                                <xs:complexType>
                                                        <xs:simpleContent>
                                                                <xs:extension
                                                                        base="xs:string">
                                                                        <xs:attribute
                                                                                type="xs:string"
name="name" />
                                                                        <xs:attribute
                                                                                type="xs:string"
name="value" />
                                                                </xs:extension>
                                                        </xs:simpleContent>
                                                </xs:complexType>
                                        </xs:element>
                                </xs:choice>
                                <xs:attribute type="xs:string" name="type" />
                        </xs:complexType>
                </xs:element>
                <xs:element name="deployment" maxOccurs="1">
                        <xs:complexType>
                                <xs:choice maxOccurs="unbounded">
                                        <xs:element name="dir" maxOccurs="unbounded"
```

```
                                        minOccurs="0" type="xs:string">
                                </xs:element>
                        </xs:choice>
                        <xs:attribute name="baseDir" type="xs:string">
                        </xs:attribute>
                </xs:complexType>
        </xs:element>
        <xs:element name="portMapping" maxOccurs="1"
                minOccurs="0">
                <xs:complexType>
                        <xs:choice maxOccurs="unbounded">
                                <xs:element name="http" maxOccurs="1"
                                        minOccurs="1">
                                        <xs:complexType>
                                                <xs:attribute name="port"
                                                        type="xs:string" />
                                                <xs:attribute name="proxyPort"
                                                        type="xs:string" />
                                        </xs:complexType>
                                </xs:element>
                                <xs:element name="https" maxOccurs="1"
                                        minOccurs="0">
                                        <xs:complexType>
                                                <xs:attribute name="port"
                                                        type="xs:string" />
                                                <xs:attribute name="proxyPort"
                                                        type="xs:string" />
                                        </xs:complexType>
                                </xs:element>
                        </xs:choice>
                </xs:complexType>
        </xs:element>
        <xs:element name="appTypes" maxOccurs="1"
                minOccurs="1">
                <xs:complexType>
                        <xs:choice maxOccurs="unbounded">
                                <xs:element name="appType" maxOccurs="unbounded"
                                        minOccurs="1">
                                        <xs:complexType>
```

```
                <xs:attribute name="name" type="xs:string" use="required" />
                <xs:attribute name="appSpecificMapping"
                                                                type="xs:string" />
                                        </xs:complexType>
                                </xs:element>
                        </xs:choice>
                </xs:complexType>
                </xs:element>
        </xs:choice>
        <xs:attribute type="xs:string" name="type"/>
        <xs:attribute type="xs:string" name="host" />
        <xs:attribute type="xs:string" name="provider" />
         <xs:attribute type="xs:string" name="version" />
         <xs:attribute type="xs:boolean" name="multiTenant" />
        </xs:complexType>
        </xs:element>
        </xs:choice>
        </xs:sequence>
        </xs:complexType>
</xs:element>
```

{WSO2-CC}/repository/conf/etc/service.xsd

```
<xs:element name="service">
        <xs:annotation>
        <xs:documentation>you can have 0..n service elements</xs:documentation>
        </xs:annotation>
        <xs:complexType>
                <xs:choice maxOccurs="unbounded">
                        <xs:element name="cartridge">
                                <xs:annotation>
                                        <xs:documentation>
                                                this element's value should be a reference
                                                to an existing cartridge
                                        </xs:documentation>
                                </xs:annotation>
                                <xs:complexType>
                                        <xs:simpleContent>
```

```
                                        <xs:extension base="xs:string">
                                            <xs:attribute type="xs:string"
                                                name="type" />
                                        </xs:extension>
                                    </xs:simpleContent>
                                </xs:complexType>
                            </xs:element>
                            <xs:element name="payload" type="xs:string" maxOccurs="1"
minOccurs="0"></xs:element>
                            <xs:element name="host" type="xs:string" maxOccurs="1"
minOccurs="0"></xs:element>
                        </xs:choice>
                        <xs:attribute type="xs:string" name="domain" />
                        <xs:attribute type="xs:string" name="tenantRange" />
                        <xs:attribute type="xs:string" name="subDomain" />
                </xs:complexType>
            </xs:element>
```

{WSO2-CC}/repository/conf/etc/services.xsd

```
<xs:element name="services">
        <xs:annotation>
        <xs:documentation>Here you specify the service domains related details.</xs:documentation>
        </xs:annotation>
        <xs:complexType>
        <xs:choice maxOccurs="unbounded">
        <xs:element name="service" maxOccurs="unbounded" minOccurs="1">
        <xs:annotation>
        <xs:documentation>you can have 0..n service elements</xs:documentation>
        </xs:annotation>
        <xs:complexType>
        <xs:choice maxOccurs="unbounded">
        <xs:element name="cartridge">
        <xs:annotation>
        <xs:documentation>this element's value should be a reference to an existing
cartridge</xs:documentation>
        </xs:annotation>
        <xs:complexType>
        <xs:simpleContent>
```

```
        <xs:extension base="xs:string">
                <xs:attribute type="xs:string" name="type"/>
        </xs:extension>
        </xs:simpleContent>
        </xs:complexType>
        </xs:element>
        <xs:element name="payload" type="xs:string" maxOccurs="1" minOccurs="0"></xs:element>
        <xs:element name="host" type="xs:string" maxOccurs="1" minOccurs="0"></xs:element>
        </xs:choice>
        <xs:attribute type="xs:string" name="domain"/>
        <xs:attribute type="xs:string" name="subDomain"/>
        <xs:attribute type="xs:string" name="tenantRange" />
        </xs:complexType>
        </xs:element>
        </xs:choice>
        </xs:complexType>
</xs:element>
```

Sample Topology Configuration

```
services {
        appserver {
                domains {
                        as1.domain {
                                hosts     mgt.as.slive.com;
                                sub_domain        mgt;
                                tenant_range      *;
                        }
                         as1.domain {
                                hosts     as.slive.com;
                                sub_domain        worker;
                                tenant_range      *;
                        }
                }
        }
        esb {
                domains {
                        esb.domain {
```

```
                        hosts      mgt.esb.slive.com;
                        tenant_range      *;
                    }
                }
            }
}
```

Iaas Abstract Class

```
/**
* All IaaSes that are going to support by Cloud Controller, should extend this abstract class.
*/
public abstract class Iaas {

        /**
        * This should build the {@link ComputeService} object and the {@link Template} object,
        * using the information from {@link IaasProvider} and should set the built
        * {@link ComputeService} object in the {@link
IaasProvider#setComputeService(ComputeService)}
        * and also should set the built {@link Template} object in the
        * {@link IaasProvider#setTemplate(Template)}.
        * @param iaasInfo corresponding {@link IaasProvider}
        */
        public abstract void buildComputeServiceAndTemplate(IaasProvider iaasInfo);

        /**
        * This method provides a way to set payload that can be obtained from {@link
IaasProvider#getPayload()}
        * in the {@link Template} of this IaaS.
        * @param iaasInfo corresponding {@link IaasProvider}
        */
        public abstract void setDynamicPayload(IaasProvider iaasInfo);

        /**
        * This will obtain an IP address from the allocated list and associate that IP with this node.
        * @param iaasInfo corresponding {@link IaasProvider}
        * @param node Node to be associated with an IP.
        * @return associated public IP.
        */
        public abstract String associateAddress(IaasProvider iaasInfo, NodeMetadata node);

        /**
```

```
        * This method should create a Key Pair corresponds to a given public key in the respective region having the name given.
        * Also should override the value of the key pair in the {@link Template} of this IaaS.
        * @param iaasInfo {@link IaasProvider}
        * @param region region that the key pair will get created.
        * @param keyPairName name of the key pair. NOTE: Jclouds adds a prefix : <code>jclouds#</code>
        * @param publicKey public key, from which the key pair will be created.
        * @return whether the key pair creation is successful or not.
        */
  public abstract boolean createKeyPairFromPublicKey(IaasProvider iaasInfo, String region, String keyPairName, String publicKey);
}
```

References

[1]http://uec-images.ubuntu.com/releases/precise/release-20120424/

[2]http://wso2.com/whitepapers/cloud-native-advantage-multi-tenant-shared-container-paas/