



ICT-287510

RELEASE

A High-Level Paradigm for Reliable Large-Scale Server Software  
A Specific Targeted Research Project (STReP)

## D4.3 (WP4): Heterogeneous super-cluster Infrastructure

Due date of deliverable: 31st December 2012

Actual submission date: 9th January 2013 / 2nd July 2014

Start date of project: 1st October 2011

Duration: 36 months

Lead contractor: Erlang Solutions Ltd.

Revision: 2.0

**Purpose:** To build and describe a framework that enables automatic deployment and monitoring of an Erlang/OTP application into a given number of Erlang nodes running in a heterogeneous cloud environment.

### Results:

- We revisited the CCL architecture presented in deliverable *D4.2*, moving from a command-line tool to a fully-blended virtualization infrastructure.
- We moved from a homogenous deployment environment to a heterogeneous one.
- We presented the RELEASE project at two major conferences (HiPEAC and Codemotion), collaborated with a team from a parallel FP7 project (*Prowess*) and started a thesis project about the implementation of a CCL plugin.

**Conclusion:** In this deliverable we have introduced a revised architecture for the developed prototype such that we are now capable of deploying Distributed Erlang applications on different Cloud infrastructures. The introduced architectural changes pave the way for adding control and monitoring functionalities which will enable users to deploy Distributed Erlang applications in a heterogeneous infrastructure, based on capability profile matching.

Project funded under the European Community Framework 7 Programme (2011-14)			
Dissemination Level			
PU	Public		*
PP	Restricted to other programme participants	(including the Commission Services)	
RE	Restricted to a group specified by the consortium	(including the Commission Services)	
CO	Confidential only for members of the consortium	(including the Commission Services)	

# Heterogeneous super-cluster Infrastructure

Roberto Aloï <roberto.aloi@erlang-solutions.com>  
 Enrique Fernandez <enrique.fernandez@erlang-solutions.com>  
 Torben Hoffmann <torben.hoffmann@erlang-solutions.com>  
 Csaba Hoch <csaba.hoch@erlang-solutions.com>

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	A note about terminology	5
1.2	Suggestions proposed during the second-year review	5
<b>2</b>	<b>Architecture</b>	<b>7</b>
2.1	WombatOAM's vertical architecture	9
2.2	WombatOAM's horizontal architecture	10
2.3	Summary of the horizontal and vertical architecture	12
<b>3</b>	<b>Implementation</b>	<b>12</b>
3.1	A solution to the heterogeneity problem: Libcloud	12
3.2	The RESTful Interface of WombatOAM	14
3.3	The WombatOAM Web Dashboard	15
3.4	Code Injection in Erlang	15
3.5	Databases	16
<b>4</b>	<b>Interacting with WombatOAM</b>	<b>17</b>
4.1	Deploying a distributed Erlang application	17
4.1.1	Registering Cloud Providers	17
4.1.2	Creating and Uploading an Erlang Release	21
4.1.3	Creating Node Families	24
4.1.4	Deploying a Node	28
4.1.5	Stopping the Cluster and Cleaning Up	31
4.2	An overview of WombatOAM's monitoring features	31
4.2.1	Topology	32
4.2.2	Metrics	34
4.2.3	Notifications	35
4.2.4	Alarms	37
<b>5</b>	<b>Conclusions and Future Work</b>	<b>38</b>
<b>A</b>	<b>Historical notes</b>	<b>39</b>
A.1	From a Command Line Tool to a Fully-blended Virtualization Infrastructure	39
A.1.1	Evolving the CCL Architecture	39
A.1.2	The CCL Components	39

A.1.3 Cross-component Communication . . . . .	40
A.2 Evolving the CCL Infrastructure to WombatOAM . . . . .	42

## Executive Summary

The main objective of this deliverable (D4.3) is to move from a homogeneous architecture which only allows the user to deploy Erlang applications on a single cloud infrastructure, to a heterogeneous one which, given a distributed Erlang application, allows users to automatically deploy the application into heterogeneous clusters running in different cloud environments.

To succeed in our journey from a homogeneous infrastructure to a heterogeneous one, we have heavily revised the architecture of the developed prototype. We also gave it a new name (WombatOAM). In the previous deliverable (D4.2) we presented a *command-line interface* that has now been transformed into a full-fledged *operations and maintenance* application. At the heart of the new node deployment infrastructure lies *Libcloud*, an open-source Apache project enabling the communication towards different cloud providers via a common interface. After the nodes are deployed, WombatOAM monitors their availability, metrics, logs and alarms, and stores all of this in its database. Last but not least, WombatOAM exposes a *RESTful* interface; atop of which sits a web dashboard intended for easing the deployment and management of Erlang applications on heterogeneous super-clusters by displaying all the collected information mentioned above, and by providing an easy access to the deployment functionalities.

The revised architecture along with all the other newly implemented features presented in this deliverable represent a clear step forward towards a broker layer capable of creating, managing and dynamically scaling super-clusters of smaller heterogeneous clusters, based on capability profile matching.

# 1 Introduction

Deliverable *D4.3* is part of the Work Package *WP4* of the *Release* project. The Work Package, titled *Scalable Virtualization Infrastructure*, is aimed at providing a broker layer capable of creating, managing and dynamically scaling heterogeneous super-clusters of Erlang nodes, based on capability profile matching. More specifically, deliverable *D4.3* focuses on heterogeneous environments, as opposed to the homogeneous environments analyzed as part of deliverable *D4.2*. When deploying new nodes, though, it still requires to be told explicitly which cloud provider should run the new nodes; rather than relying on a dynamic, capability-driven approach which will be considered during deliverable *D4.4*.

For achieving our goal of supporting heterogeneous deployment (i.e. deploying the nodes of an Erlang system in a way that they are distributed among several cloud providers), we needed to overcome a challenge. By looking at the current cloud infrastructure offering, one thing is clear – cloud interoperability has not been one of the main drivers in building these infrastructures. Due to the lack of cooperation between the main cloud service providers where the leading vendors in the cloud market have only focused on promoting their own, incompatible cloud technologies, customers of current cloud infrastructure offerings are locked into a single cloud infrastructure. Since the big players seem not to be interested in providing a common framework for accessing cloud services from different sources, several standardisation initiatives have popped up in order to help out with this. It is clear that having such common framework would allow cloud customers to easily migrate their cloud resources between cloud service providers, compare the characteristics of different cloud offerings in order to choose the one that better suits their requirements, etc. In the context of the RELEASE project, the main benefit for WombatOAM would be the possibility to deploy Erlang nodes on different clouds without the burden of having to deal with several, non-consistent interfaces.

The implementation of CCL (which we have renamed to WombatOAM) presented in *D4.2* [32] was strictly tightened to the Amazon EC2 cloud infrastructure offering. The major challenge we agreed on facing in the next phase of the project (i.e. in the current deliverable) was to move from a homogeneous environment to a heterogeneous one where clusters were not necessarily Amazon-based. After all, the ultimate goal of the Release *WP4* is to provide a broker layer capable of creating, managing and dynamically scaling super-clusters of smaller heterogeneous clusters, based on capability profile matching.

We solved the challenge by integrating with Libcloud, a library for interfacing different cloud providers (see section 3.1). In this deliverable, we describe the architecture and implementation of WombatOAM and demonstrate that it can perform heterogeneous deployments by deploying an Erlang application to seven virtual machine instances that belong to four different provider.

This document is structured as follows:

**Section 2** describes WombatOAM's architecture, including the general principles and the concrete components.

**Section 3** gives an overview of the technologies used to implement WombatOAM, explaining the reasons behind their adoption.

**Section 4** provides an introduction to using WombatOAM via both its web dashboard and its REST API. It walks the reader through a simple scenario in which a user wants to build, deploy and monitor an Erlang application into a heterogeneous node family.

**Section 5** contains our conclusions and some notes about the expected future work.

## 1.1 A note about terminology

The title of this document contains the term *super-cluster*. It should be noted here that as part of another RELEASE deliverable (*D6.3, DECO, Distributed Erlang Component Ontology*[33]), the terminology of distributed Erlang systems was clarified and disambiguated. In WP4 the term *cluster* was used to mean *Erlang nodes that execute the same release* and *super-clusters* were used to mean a set of clusters, i.e. *Erlang nodes that don't all execute the same release*. While working on DECO, we decided that the term *cluster* is better to be used as a set of Erlang nodes that work together and achieve a common function, but possibly run different releases and have different tasks. For example a cluster might contain front-end nodes and back-end nodes. We decided to resolve this conflict by finding a new name for what we had called clusters in WP4, and we chose the term *node family*, because it fits well with the meaning we want to use it for, and it is a term not yet used in the Erlang community. (We were also considering the term *node group*, but that could have been confused with the term *S-groups*, another central concept in the RELEASE project.)

## 1.2 Suggestions proposed during the second-year review

During the second-year review at the end of 2013, the reviewers had some comments and suggestions about our original D4.3 deliverable. These comments and suggestions were the following:

Indeed the deliverable does not stand on its own and the reviewers had not understood a number of things which only became clear as a result of the review meeting presentation. Deliverable D4.3 should therefore be re-submitted with the following improvements:

1. It should be clarified that the previous tool, named CCL, described in period 1 and in deliverable D4.2 of period 2, was abandoned and replaced by the new tool, named Wombat, and a justification provided.
2. It should be clarified that the architecture of CCL, in particular its reliance on RabbitMQ as a middleware layer, was abandoned, and a justification provided.
3. It should be clarified that so far only one attempt was made to deploy an Erlang application, namely Riak, on a heterogeneous cluster of 2 cloud platform providers, and this attempt described.
4. The current scope of Wombat should be clarified, i.e. just deploying an existing Erlang application on a cloud and providing metrics. It should be explained that enabling this application's components to communicate with each other in a heterogeneous cluster is entirely up to the application's developers. If this scope is planned to be widened during the project these plans should be described. The current architecture of Wombat, which was adequately described during the review presentation, should be included in the deliverable.

In this resubmitted version of D4.3, we have added much information and details about Wombat-OAM. The added details also answer the questions above. Here we collected a summary about the answers and where to find more information about them.

1. During the one year between submitting the original version of the D4.3 deliverable and reviewing it, CCL organically evolved. This evolution includes modifying the architecture (see the next point). During this year, it has also been renamed to WombatOAM to signify that it is now closer to being a product (as opposed to being a prototype). These two changes might have given the false impression that CCL was abandoned and a new tool was written instead, but actually WombatOAM is the continuation of CCL, only with a different name. For example the code base

evolved continuously. This process (including the last half year since the review) is described in detail in appendix section A.2.

2. CCL's architecture evolved based on our own experiences and others' feedback regarding what worked well, what did not, what was missing. This evolution includes modifying its vertical architecture by having middle manager components instead of CCL-Clusters and CCL-O&M components: the reason for this change was that middle managers provide more flexibility in terms of distributing which node is managed by which middle manager. (See section A.2 item 2 for more details.) Another significant architectural change was adding a horizontal dimension to the architecture in order to obtain better extensibility (see section 2.2). The third significant change was to stop using RabbitMQ for the internal communication between WombatOAM's components and use Erlang distribution instead. Section A.2 item 3 describes the reasons in detail. To sum up, the way we implemented remote procedure calls using RabbitMQ's queues made it difficult to handle errors the correct way, and the fact that we used an external tool added more complexity for WombatOAM users when installing and configuring WombatOAM. Using Erlang distribution is simple and fits well with Erlang's logic, making it simpler to build WombatOAM in a reliable way. That said, there are reasons for considering other protocols, so replacing AMQP and RabbitMQ with Erlang distribution is not a final decision.
3. At the time of the second-year review, WombatOAM worked with two cloud providers, Amazon EC2 and Rackspace. We also had a provider type called *physical* provider, with which nodes can be deployed to already running machines. Since then HP cloud has been added to the list of supported providers. Section 3.1 articulates that these are the only supported providers, and it also explains that supporting other providers would need some (but not a lot of) extra work in each case because of the differences between them that are not hidden by tools that provide a common API towards them, like DeltaCloud or Libcloud. WombatOAM is used by the Megaload project to deploy Erlang nodes into homogeneous clusters (to Amazon EC2 or to physical machines), but indeed as the reviewers noted, WombatOAM is not yet used by others to deploy heterogeneous clusters, so the only attempts to do so have been our own attempts, which include the one presented in this document in section 4.1, which describes deploying a 7-node Riak cluster spread among 4 providers (3 cloud providers and a physical provider).
4. Since the second-year review, WombatOAM has learned to collect not only metrics, but also notifications and alarms from the managed nodes. The scope of its deployment part didn't change though, so indeed, its responsibility of making sure that the nodes connect to each other is limited to making sure that the nodes have appropriate names matching the host machines (otherwise they wouldn't be able to connect); executing a bootstrap script on the nodes, which can do the actual job of connecting the nodes in a way preferred by the application; opening the firewall ports specified by the user so that communication can take place; and other infrastructural duties. Section 2.2 item 6 describes on a high level what Orchestration does and doesn't do. Section 4.1 describes concrete details of the infrastructure with which WombatOAM helps the deployment work, and it includes the few simple steps that need to be performed (by the application developer or someone else) in order to make an Erlang application deployable by WombatOAM. The current WombatOAM architecture is described in section 2. The section also explain why the different services (including Orchestration) are a bit different from each other. In the architecture diagram shown during the second-year review presentation, the Orchestration service was even more different from the others than it is now. That has a historical reason: the Orchestration service was then not yet fully modified according to the new vertical/horizontal architecture.

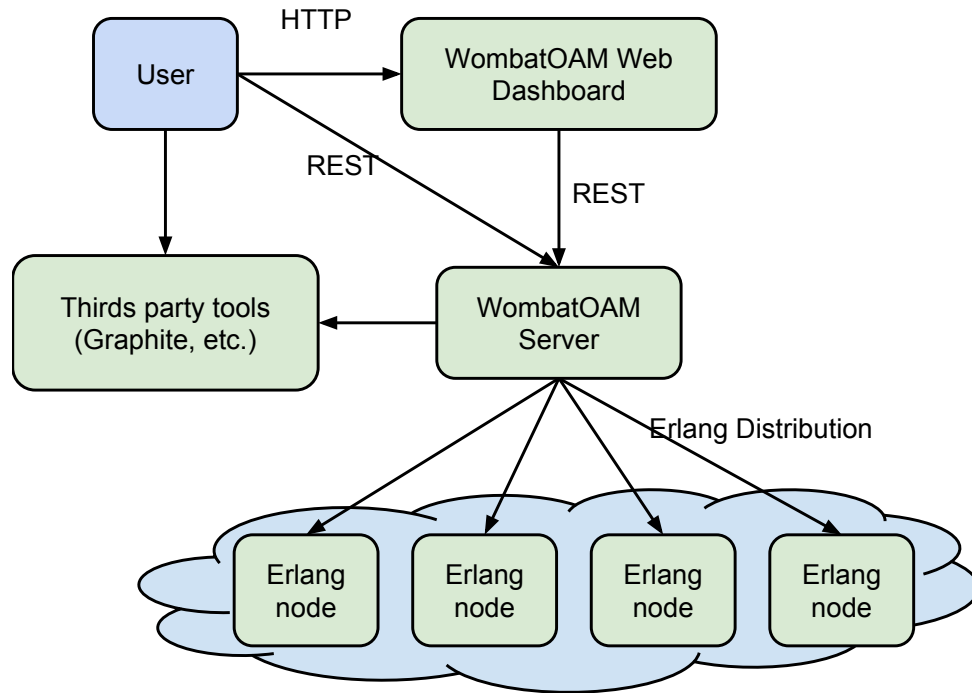


Figure 1: A system that uses WombatOAM

## 2 Architecture

A system that uses WombatOAM is illustrated in figure 1. The WombatOAM server (which is written in Erlang) handles the Erlang nodes in the system – this may include deployment and/or monitoring. The Erlang nodes may be located in Infrastructure-as-a-Service clouds (like Amazon EC2, HP Cloud or Rackspace), or they can be on physical machines. In either case, WombatOAM can be used for deployment, monitoring, or both.

The user usually communicates with WombatOAM via its web dashboard (see section 3.3), which is written in JavaScript. It is also possible to directly communicate with the WombatOAM server using its REST interface (see section 3.2), which is useful for writing programs that talk to WombatOAM. WombatOAM also has the capability to push the metrics it collected into *Graphite*[10], which is a tool for storing and visualizing metrics. Integrating WombatOAM with other tools as well like Nagios for storing and visualizing alarms is a future possibility.

Let's zoom in to WombatOAM server and examine its internal structure. The generic architecture of WombatOAM has two important aspects (a vertical and a horizontal aspect), which arise from two important requirements: scalable architecture and extensible infrastructure.

The scalability is provided by different vertical levels (user interfaces, REST interface, master, middle managers, agents). The extensibility is provided by a modular code structure where different services provide different functionalities (Core, Topology, Metrics, Notifications, Alarms and Orchestration are all implemented as services).

The overall architecture of WombatOAM is depicted in figure 2. Each vertical level is an architectural level, while each column is a service. The vertical levels are described in section 2.1. Services look different because not all of them need all levels, and also because the Orchestration service uses an external component called Libcloud. The reasons behind these differences are detailed in section 2.2. Note that not all services are on the architecture diagram: for sake of brevity, the Custom Command service is not shown.



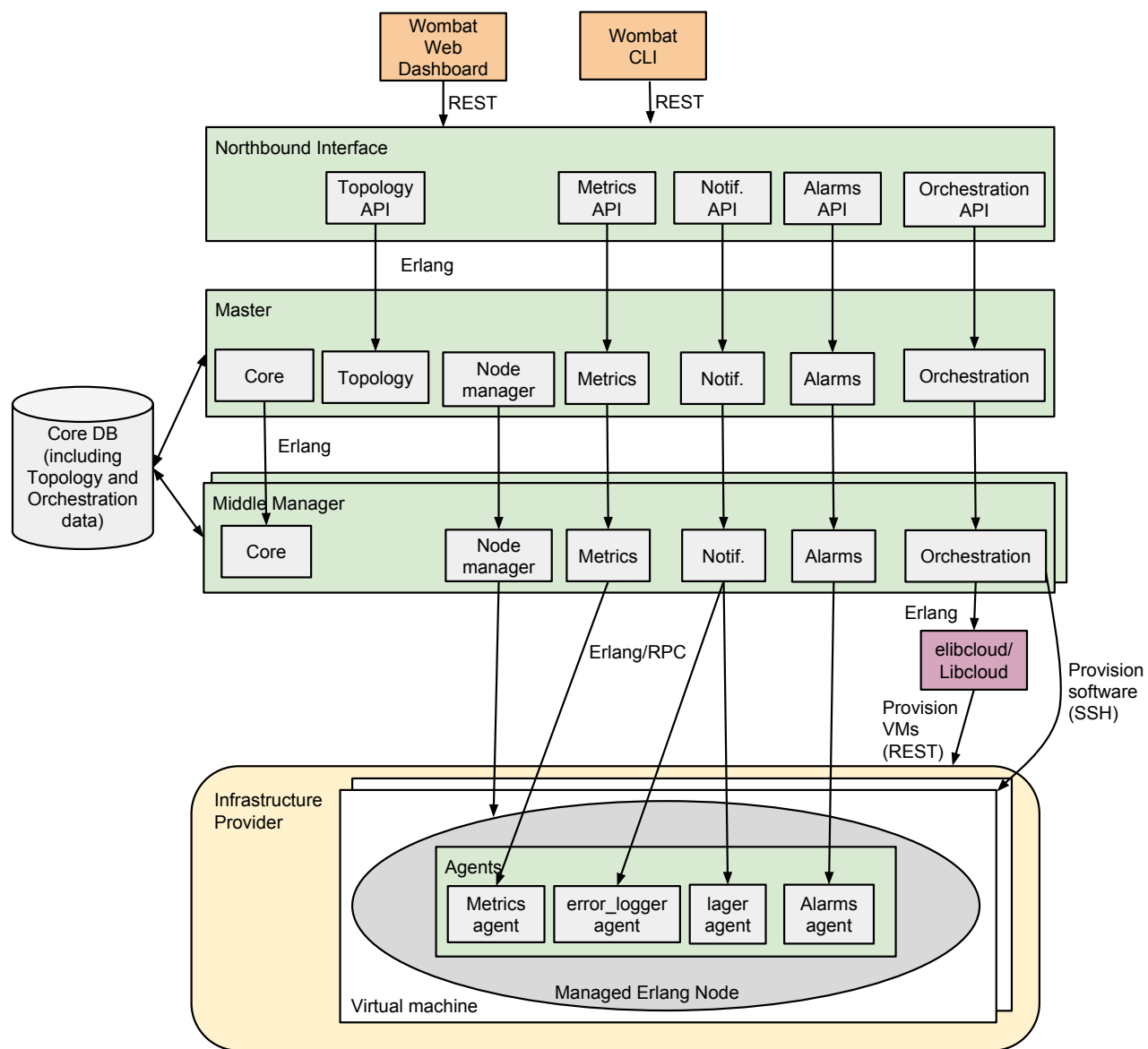


Figure 2: WombatOAM's main components

## 2.1 WombatOAM's vertical architecture

Vertically, the WombatOAM architecture has five levels. Each level talks only to the levels below and above it. In the description below, we use the example of adding a new node to WombatOAM to make the roles of the levels easier to grasp.

The five levels are the following:

1. **WombatOAM web dashboard and WombatOAM CLI:** These interfaces translate the user's actions into REST requests.

E.g. when the user adds an existing Erlang node to WombatOAM by submitting a form on the Web interface, the web dashboard will send a `POST /api/actions/add-node` request to the REST interface, where the request body contains the name and cookie of the node (wrapped into a JSON object).

2. The **Northbound interface** (a.k.a. WombatOAM API) is a REST interface. When it receives a request, it calls functions on the master level to provide the answer.

E.g. when it receives `POST /api/actions/add-node`, it calls `wo_node_disc:add_node(Node, NodeDiscovery)`.

3. The **master level** contains master components that are global to the WombatOAM system. They manage and use the middle managers (see the next bullet point). There is a master process for each service. The core master process is special, because its role is to manage the other services. This includes managing the master component of the other services, and managing its own middle manager components (which in turn manage the middle manager component of other services). The master components of other services perform tasks that are better to perform centrally, for instance because the caller does not know which middle manager should handle its request. Then the master components can forward the request to the appropriate middle manager if necessary. Another reason for performing an operation on the master node might be that the request can be answered by reading something from the central database, so there is no need to involve a middle manager at all when serving the request.

E.g. when adding a node, the Northbound interface calls `wo_node_disc:add_node(NodeName, NodeDiscovery)`. (The `wo_node_disc` module is part of the master component of the Node Manager service.) This will call `wo_node_mgr_master:do_manager_node` (still on the master), which decides which middle manager to use, and calls `wo_node_mgr_mm:manage_node(MiddleManagerId, NodeId)` to ask the appropriate middle manager to start managing the node.

4. A **middle manager** (abbreviated as *mm*) is a set of components that manages a number of Erlang nodes by directly connecting to them. It may contain at most one component from each service. A middle manager has a core middle manager process and one middle manager process for each plugin. The core middle manager process manages the plugin middle manager processes. We introduced middle managers for scalability reasons: if we want to manage so many Erlang nodes that one WombatOAM node is not enough, we can have one master WombatOAM node and several middle manager WombatOAM nodes. Then the managed nodes can be divided among several middle managers.

Another possible application of middle managers is storing data. Currently we store almost all data in Mnesia tables (see section 3.5), which can be accessed both from the master and the middle manager levels, but it is worth considering storing some data in the middle managers, which could serve them to the master when asked to do so. Metric data is the most obvious use case for this method, because by default the data collected there is aggregated on multiple levels, e.g. the original data points are deleted within an hour.

When a middle manager starts managing a node, its processes may inject agent modules into those nodes, and call them when they need some information from them.

E.g. when `wo_node_mgr_mm:manage_node(MiddleManagerId, NodeId)` is called, the given middle manager connects to the given node with the given cookie (the latter is read from the database), and injects all agent modules of all services into the node, e.g. the `wo_log_lager_agent` and `wo_log_elflogger_agent` agent modules for the Notifications service, which will be used by the Notifications middle manager to subscribe to log entries created on the node.

5. **Agent processes** run on the Erlang nodes of the managed system. We are writing them to be as non-intrusive as possible. `wo_log_lager_agent` and `wo_log_elflogger_agent` are examples of agents.

WombatOAM's components communicate via Erlang distribution, which is Erlang's built-in mechanism for processes to communicate with each other (whether they are on the same Erlang node or not). We have been experimenting with using the AMQP protocol and the RabbitMQ message exchange for the communication between WombatOAM's components, but that proved to be more cumbersome to use than we expected. As soon as there is a good reason though (scalability, security, market demands, etc.), we can consider again using AMQP or some other protocol for WombatOAM's internal communication. Section A.2 gives a more detailed analysis regarding this question.

## 2.2 WombatOAM's horizontal architecture

Horizontally, WombatOAM has a number of services. The Core service implements the service infrastructure itself. It handles the other services uniformly, thus it is easy to add new services.

1. The **Topology** service handles adding, deleting and discovering nodes. It also monitors whether they are accessible, and if not, it notifies the other services, and periodically tries to reconnect. When the nodes are available again, it also notifies the other services. It doesn't have a middle manager part, because it doesn't talk to the nodes directly: instead it asks the Node manager service to do so.
2. The **Node manager** service maintains the connection to all managed nodes via the Erlang distribution protocol. (That's why figure 2 shows it being connected to the Managed Erlang node itself, as opposed to a concrete agent like Metrics, Notifications and Alarms plugins.) If it loses the connection towards a node, it periodically tries to reconnect.

It also maintains the states of the nodes in the database (e.g. if the connection towards a node is lost, the Node manager changes the node state to `DOWN` and raises an alarm). The Node manager doesn't have a REST API, since the node states are provided via the Topology service's REST API.

3. The **Metrics** service collects metrics from the managed nodes, stores them in the database. It provides the metrics via the REST interface, and by optionally pushing them into an external tool called *Graphite* [10]. It collects around 90 built-in metrics (e.g. total memory usage of the node), and it also collects the metrics stored on the node by the *Folsom* [8] and *Exometer* [11] libraries.
4. The **Notifications** service collects notifications and logs from the managed nodes, stores them in the database and provides them via the REST interface. It collects logs reported to the `SASL error_logger` [5] and the *Lager* [37] library; for these tasks, it uses two separate agents (one for `error_logger` and one for `lager`).

5. The **Alarms** service collects alarms from the managed nodes, stores them in the database and provides them via the REST interface. It collects around 20 built-in alarms (e.g. an alarm is raised when the number of processes approaches the process limit), and alarms reported to the SASL `alarm_handler` [5] or the *Elarm* library [28].
6. The **Orchestration** service can deploy new Erlang nodes on already running machines, and it can also provision new virtual machine instances using several cloud providers, and deploy Erlang nodes on those instances. For communicating with the cloud providers, the Orchestration service uses an external library called *Libcloud* [17], for which we have written an open source Erlang wrapper called *elibcloud* [29], to make Libcloud easier to use from WombatOAM.

Note that WombatOAM Orchestration doesn't provide a platform for writing Erlang applications: it provides infrastructure for deploying them. As without WombatOAM, a distributed Erlang program should make sure that its components talk to each other either after being connected by invoking a (bootstrap) command on them, or after being connected via Erlang distribution. A good example is Riak [38], which is written in a way that the user can independently deploy the Riak nodes, and join them using the `riak-admin cluster join` command (`riak-admin` is a script delivered with Riak). WombatOAM makes sure that the nodes have appropriate node names (e.g. the node name should contain the IP address or domain name of the host on which it is running), opens the necessary firewall ports, invokes the specified bootstrap command, and performs other helpful operations on the infrastructural level. But it doesn't provide a library, a framework or a platform that the applications should use: this way WombatOAM can be used to deploy any distributed Erlang application that doesn't deploy its own nodes but lets its node deployed, and lets them connect by calling scripts or Erlang function calls.

7. The **Custom Command** service provides a REST interface for invoking Erlang functions on the managed nodes.

A service must provide at least two callback modules that are used by the core service to manage it: one implementing the master process of the service, and one implementing the middle manager process of a service:

1. The service's master process is started by the core master process, and it can use WombatOAM's libraries to extend WombatOAM's REST interface (e.g. the Orchestration service adds a REST handler module to the REST level, which will handle the requests whose URL starts with `/api/orch`).
2. In each middle manager, the core middle manager process starts the middle manager processes of the other services. Those middle manager processes can use WombatOAM's libraries to inject its own agents into the managed nodes.

A service will often have a component in each level: it has its own REST interface, its own master process, its own middle manager processes, and its own agents. It is not mandatory though to have components on all levels; for example the Orchestration service does not have an agent, because it doesn't need one.

When WombatOAM is started, the services are started one by one: the next is started only when starting the previous one has finished. For each service, first the master is started, then the middle managers. An implementation detail that is worth mentioning is that the timeout in both cases is *infinity*, so the services can have as much time for starting up as needed. This follows the usual Erlang model for starting applications and supervisor trees.

Services (and any other WombatOAM code) can use events to be notified about new nodes being added, nodes going down, etc. The `wo_event` module implements a publish/subscribe library (it is

currently a thin wrapper around the *gproc*[39] library). Services like the Topology service add their own layer on the top of `wo_event` to make sending notifications and subscribing to them more convenient for other processes.

## 2.3 Summary of the horizontal and vertical architecture

The **vertical architecture** of WombatOAM helps **scalability**, because we can start as many middle manager nodes as needed to manage the nodes, and the middle manager processes can filter information to the master processes. One WombatOAM master can handle many middle managers, and one middle manager can handle many managed nodes.

The **horizontal architecture** of WombatOAM provides the easy way to implement **services** by managing them in a well-defined way and by providing them with libraries for talking to the managed nodes and providing a REST interface.

## 3 Implementation

The purpose of this section is to give an overview of the technologies used to implement the WombatOAM components presented in the previous sections and to illustrate some of the reasons for using them.

### 3.1 A solution to the heterogeneity problem: Libcloud

In the previous release of WombatOAM (at that time it was called CCL) we made use of *erlcloud* [19], a popular open-source Erlang library, for deploying Erlang applications on Amazon EC2. Sadly, due to its lack of support for other cloud service providers and our aim at supporting a heterogeneous infrastructure involving multiple cloud service providers, we have been forced to dismiss *erlcloud* and to consider other alternatives enabling us to interact not only with Amazon EC2, but also with other big players such as OpenStack, OpenNebula and Eucalyptus, to cite some. After having evaluated a set of different options including OCCI [12] and CIMI [23] and based on the maturity of the projects and the simplicity in terms their ease of integration into WombatOAM, we have finally decided upon using Deltacloud[13], an open-source Apache project that implements a simple REST-based API for accessing a broad list of cloud service providers or, in words of its authors, an API that abstracts the differences between cloud service providers.

Afterwards, however Red Hat scaled back its sponsorship of the Deltacloud project (3 years after creating it) [6]. Being the main sponsor, this meant that the development has been slowed down so much that Deltacloud couldn't follow the new cloud APIs. A practical example is that at the end of 2013, it's Rackspace Driver still used OpenStack 1.0 authentication, which was deprecated at Rackspace, who allowed only OpenStack 2.0 authentication for newly created accounts[34]. Another example is that it does not support the Google Compute Engine, which is a relatively new player in this field.

We started investigating cloud management tools again, and arrived at *Apache Libcloud*[17]. It has the same goal as Deltacloud, making it possible to communicate with different cloud providers using one tool, and abstracting away the differences as much as possible. On the other hand, it has an active community, and its scope goes beyond Deltacloud, which supported two kinds of services[14]:

- the Compute services of 15 providers and
- the Storage services of 7 providers.

Libcloud supports four kinds of services[15]:

- the Compute services of 41 providers,

- the Storage services of 8 providers,
- the Load balancer services of 7 providers, and
- the DNS services of 7 providers.

Libcloud is a Python library. In order to programmatically interact with it, and due to the fact that there was no Erlang client available at the time of this writing, we have implemented our own simple Erlang wrapper for Libcloud enabling us to programmatically manage virtual machines. We named the wrapper *elibcloud*[29] and released it as open source. It supports the Compute service on 3 providers (Amazon, HP Cloud and Rackspace).

New providers can be added with a very minimal amount of effort. This small effort is still needed because Libcloud doesn't abstract away all differences between providers: it particular, security groups and floating IP addresses are handled differently in case of different providers.

Even those providers who use the same standard like HP Cloud and Rackspace, who both provide an OpenStack API, use the standard in a different way: when e.g. provisioning virtual machine instances, Rackspace will automatically assign public IP addresses to them, while in case of the HP Cloud, the "Floating IP address OpenStack API" must be used to attach public IP addresses to the instances. *elibcloud* does a relatively good job of hiding the differences between providers (even most of those differences that are not hidden in Libcloud, e.g. handling the public IP addresses). There are still some differences though that it does not hide from the application: for example Amazon EC2 and HP Cloud instances contain a non-root user account by default, while instances on Rackspace are delivered only with a root account, so the application (in this case WombatOAM) needs to connect to the new instance and create a non-root account. (We could have used *elibcloud* to hide this difference as well, but our design decision was that like Libcloud, *elibcloud* should talk only to the cloud provider via its Cloud API, it should never communicate directly with the instances.)

Figure 3 illustrates how WombatOAM makes use of Libcloud for deploying Erlang nodes on several cloud infrastructures.

Assuming we want to deploy a virtual machine on Amazon EC2, all we need to do is interact with the implemented wrapper as illustrated in listing 1.

#### Listing 1: Provisioning a virtual machine instance with *elibcloud* (from an Erlang shell)

```
% Create credentials for our Amazon EC2 account
> {ok, Cred} = elibcloud:create_credentials("EC2", Username, Password).

% Upload our SSH keys to the cloud, so that when we create the virtual machine,
% it can use this SSH key to let us log in.
> elibcloud:import_key_pair_from_file(Cred, "My SSH key",
                                     "/home/user/.ssh/id_rsa.pub").

{ok, [{}]}

% A virtual hardware configuration we will use.
> SizeId = "t1.micro".

% The identifier of the virtual machine image we will use.
> ImageId = "ami-fac0cd8e".

% Create a virtual machine instance using the specified image, hardware profile
% and SSH key.
> elibcloud:create_node(Cred, "My new node", SizeId, ImageId, "My SSH key", []).
{ok, [{<<"publicIps">>, [<<"15.125.113.68">>]},
      [<<"id">>, <<"febec630-b0f4-48aa-a510-0e137157331a">>]]}

% Destroy the virtual machine instance.
```

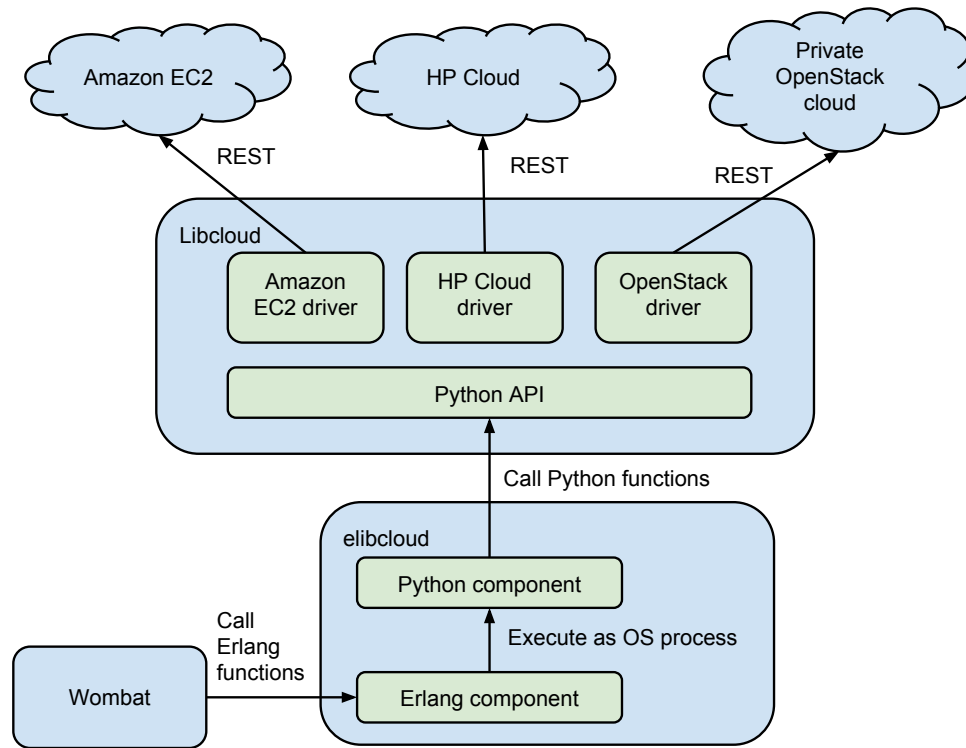


Figure 3: elibcloud/Libcloud Integration

```
> elibcloud:destroy_node(Cred, {name, "My new node"}).
```

Beyond handling SSH keys and virtual machine instances, *elibcloud* also supports creating and configuring security groups, which are the means of configuring firewall rules between our nodes in the cloud and the rest of the world.

### 3.2 The RESTful Interface of WombatOAM

All of the WombatOAM functionalities are exposed to the outer world via a RESTful (REpresentational State Transfer) interface. Real-time data are pushed to clients using the *Websocket*[21] protocol. *Cowboy*[30], a small, fast and modular HTTP server written in Erlang, has been used to dispatch HTTP requests to Erlang handlers and to handle websocket connections.

The example in listing 2 shows how to deploy a node of a certain node family using the *cURL*[36] utility. WombatOAM is running on port 8080 of *localhost*. The the long UUID is the identifier of the node family from which WombatOAM knows which provider to deploy the nodes on and which Erlang release to copy to them and start on them. The extra information needed to fulfil the request (i.e. the amount of nodes that we want to deploy) is passed as the HTML request's body in JSON format.

Listing 2: Deploying a node using WombatOAM's REST interface

```
$ NF="31743d2a-9f1d-4ec1-96e4-13c0cf2f23b1"
$ curl -X POST http://localhost:8080/api/orch/node_families/$NF/nodes \
  -H 'Content-Type: application/json' \
  -H 'Accept: application/json' \
  -d '{"amount": 3}'
```



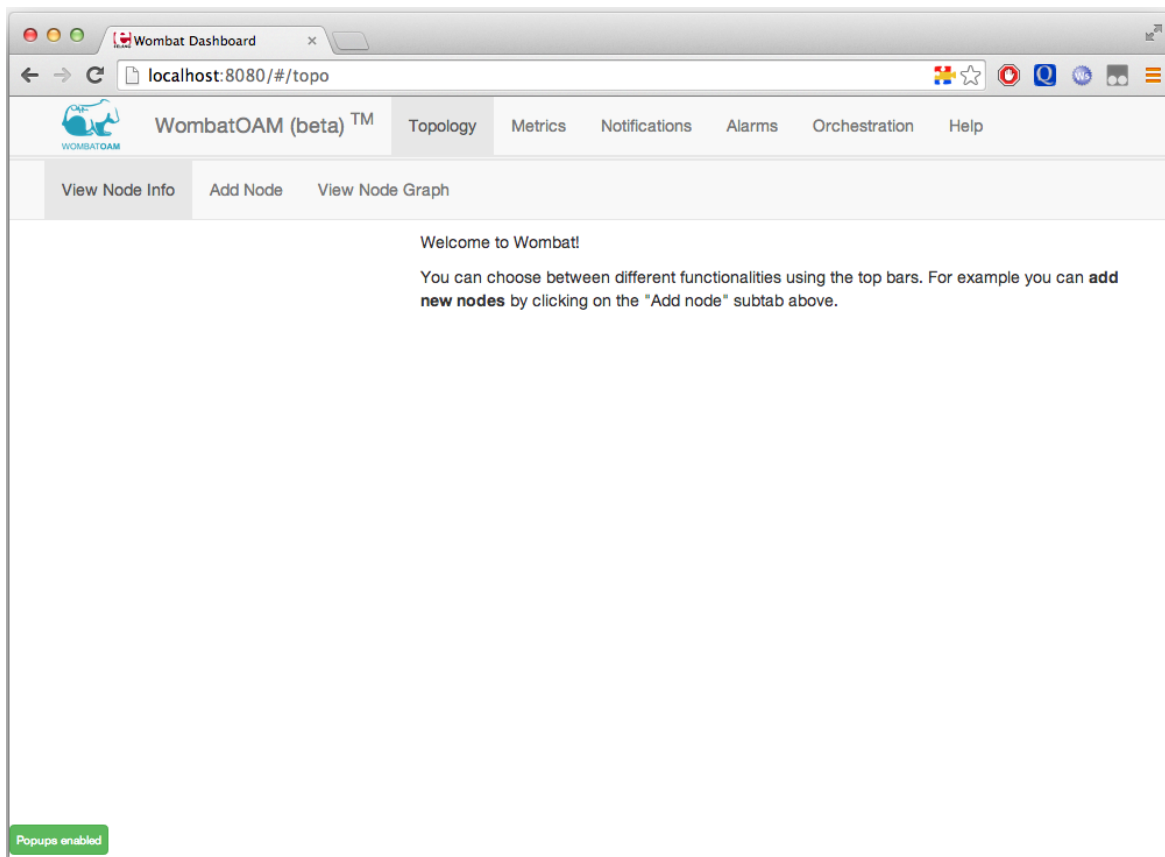


Figure 4: The welcome screen of the WombatOAM Web Dashboard connected to a WombatOAM server without managed nodes

### 3.3 The WombatOAM Web Dashboard

On the top of the REST API, we built a web application called *WombatOAM Web Dashboard*. To build the web dashboard, we made use of the *Twitter Bootstrap*[26] framework and the *jQuery*[27] library. A special note goes to the MVC library *AngularJS*[24], which allowed us to represent our data as *Models*, revealing a perfect fit for the RESTful API exposed by Cowboy. Finally, the JavaScript library *Highcharts.js*[7] has been used for creating the charts visible in the web dashboard.

The main functionalities of WombatOAM are grouped into categories, which correspond well with the services described in section 2.2. On the web dashboard, each such group of functionality is available from a tab, as shown in figure 4: *Topology*, *Metrics*, *Notifications*, *Alarms* and *Orchestration*. Under the tabs, there are subtabs: for example under the *Topology* tab, the different functionalities are available under the *View Node Info*, *Add Node* and *View Node Graph* subtabs.

Section 4 explains how to use the Web Dashboard, especially the *Orchestration* tab. It also shows how to use the REST API for *Orchestration*.

### 3.4 Code Injection in Erlang

We have mentioned in section 2.1 that the WombatOAM server *injects* some code into the Erlang nodes it monitors. Injecting a module and starting a process executing that module can be achieved in Erlang by means of two lines of code shown in listing 3; the example shows starting the alarm agent.

Listing 3: Injecting code into a remote Erlang node



```
rpc:call(Node, code, load_binary, [Mod, Filename, Bin]),
rpc:call(Node, wo_alarm_agent, start, [self(), ...]);
```

The Erlang *Remote Procedure Call* services allow us to evaluate a function call on a remote Erlang node. In this case we are remotely invoking the two functions shown in listing 4.

Listing 4: Functions remotely executed in order to inject code

```
code:load_binary(Mod, Filename, Bin).
wo_alarms_agent:start(self(), ...)
```

The first function call makes use of the `code` module. The module is a mere interface to the Erlang *code server*, a process which deals with the loading of compiled code into a running Erlang runtime system. This function is used to load object code on the remote Erlang node. The argument `Bin` contains the object code for the module `Mod`. `Filename` is only used by the code server to keep a record of from which file the object code for `Mod` comes.

The second function call starts a `gen_server` (the agent) on the remote node. The agent will be a backdoor for the WombatOAM node. Please note how the process that performs the injection (which is the middle manager process of the Alarms service in the example) passes a reference to itself to the remote node, using the built-in function `self()`. This reference will be used by the agent to communicate back to middle manager process. It also plays a role in case of failure: the agent process uses `erlang:monitor` to monitor the middle manager process, and in case the latter dies, or the network connection between the nodes goes down, the agent process terminates automatically. When the middle manager process is restarted, or the nodes can reconnect with each other, the middle manager process will inject the agent again. This mechanism is useful because in any of these scenarios, WombatOAM won't leave needless processes running on the managed node.

An alternative to dynamic code injection would be to provide the WombatOAM users with a WombatOAM agent application that they can include in their nodes. This might be preferred by some organizations who are not comfortable with WombatOAM injecting its code into their system, so creating an agent application is a possibility for the future; but it is certainly worth to keep the code injection method as well, because it gives an out-of-the-box experience to the users not easily mimicked by OAM tools for most other programming languages and technologies.

### 3.5 Databases

The different kinds of data that WombatOAM stores can be grouped into three categories based on their read/write/deletion patterns and object sizes. For different categories, different database performance characteristics are important to achieve maximum performance. For the time being, we use Mnesia to store all data (except for notifications) because of its simplicity and convenience: since it is built in to Erlang/OTP, it is easier both for the developer and for the user. It also support transactions, which makes the library layer on the top of the database easier to implement. The three categories are the following:

WombatOAM Core provides a database layer called Core DB that any WombatOAM services may use (see also figure 2 on page 8). Among other things, it stores the node data (details about nodes, node families, etc), the alarm data (active alarms) and the orchestration data (providers, releases). We need to store only a small amount of information about each entry, and those entries may change. Several processes might want to change the same data, or maybe even multiple data points at the same time, so transactions make development simpler. Mnesia would be ideal for this, and it even supports replicating tables between nodes, which would make it convenient to access this data from different nodes. The only problem is that network splits are difficult to handle with Mnesia, and those cannot be discounted in a cloud environment.

Metrics are also stored in Mnesia, but we are investigating other solutions, because in that case it is not a very good fit. The characteristics of the metric data is quite different from that of the node data: here we write a large amount of information continuously, and we also continuously delete the old entries. We never change existing data.

Notifications are stored in `disk_log`[1], which is a “disk based term logger which makes it possible to efficiently log items on files.” We are investigating other solutions regarding storing notifications as well, because `disk_log` cannot efficiently read the last log entries (they can be read only sequentially from the beginning), which is often needed when displaying the latest notifications.

## 4 Interacting with WombatOAM

This section gives you an introduction to using WombatOAM through the WombatOAM Web Dashboard. First we walk through a simple scenario in which we build and deploy an Erlang application (Riak) into a homogeneous cluster. Then we briefly show WombatOAM’s main monitoring features.

In order to perform the steps in this section, we first need to install WombatOAM as described in its README. Its dependencies are Erlang/OTP, OpenSSL, Libcloud and elibcloud (see section 3.1). After installation, a minimal amount of configuration is needed, which is also described in the README: the location of the public and private SSH keys with which we would like to access the provisioned virtual machine instances should be added to WombatOAM’s configuration file.

### 4.1 Deploying a distributed Erlang application

This section shows how to create an Erlang cluster consisting of several nodes using WombatOAM. The cluster will span across multiple cloud providers and a non-cloud machines, hence it will be a heterogeneous deployment.

First we register the cloud providers in WombatOAM (section 4.1.1). Then we create a WombatOAM-compatible Erlang release from the application that we want to deploy, and upload it to WombatOAM (section 4.1.2). Afterwards we create a node family, which references both the cloud providers and the release, and contains some additional data, e.g. firewall ports to open (section 4.1.3). This means that the node family knows everything that is needed to deploy and start the Erlang application. In the actual deployment step, we ask WombatOAM to deploy nodes of this node family into all providers (section 4.1.4). After examining that the deployed nodes are connected and actually form a cluster, we stop them and delete the node families (section 4.1.5). We show all steps first on the WombatOAM Web Dashboard, and afterwards we show how the same step can be performed using WombatOAM’s REST API.

We have also created a screencast that is available online [20], which shows deploying Riak in a slightly simpler scenario: in the video, it is only deployed to one provider (i.e. a homogeneous cluster).

A simple but useful and powerful concept that makes this chapter easier to understand is *Universally unique identifier*, or UUID for short. This is a standard that lets us generate unique identifiers, for example `550e8400-e29b-41d4-a716-446655440000`, which can be used to identify various objects. In WombatOAM, each provider, uploaded release, node family and node has a UUID generated by WombatOAM that identifies it.

#### 4.1.1 Registering Cloud Providers

In order to deploy an Erlang application in the cloud, we need to have access to an Infrastructure-as-a-Service Cloud, i.e. an account to such a cloud provider is needed. These clouds provide an interface for provisioning virtual machine instances in their network. WombatOAM will use this interface, and then deploy the Erlang nodes on these instances. In the example, we will use the cloud services of Amazon

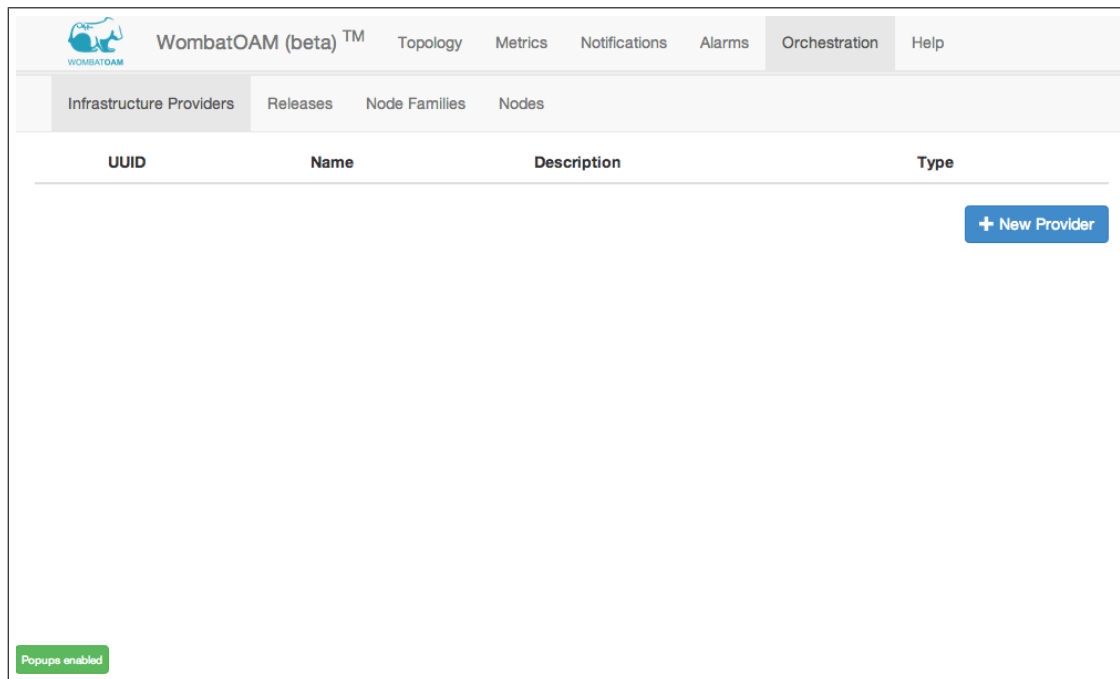


Figure 5: The *Infrastructure Providers* subtab on the WombatOAM Web Dashboard

EC2, HP Cloud and Rackspace. We have created accounts for all of them beforehand. We have chosen these clouds because of their popularity, but there is no reason why we couldn't easily add support for any of the 38 other providers that Libcloud supports. In WombatOAM, these cloud providers are called *virtual providers*.

An alternative to the cloud providers is to deploy on already running machines. In this case, WombatOAM doesn't need to provision instances. We will also include an already running machine in the cluster, to highlight WombatOAM's support for heterogeneity. In WombatOAM, when we have several running machines, we can create a so-called *physical provider*, and we can deploy to these machine via this provider. Note that a *physical provider* does not necessarily mean physical machines: both virtual and physical machines can be behind such a provider. The only criterion is that it has a host name (IP address or domain name) and an SSH server, with which WombatOAM can log into it.

After creating the cloud provider accounts for virtual providers and/or collecting the host names for the physical providers, we need to register them in WombatOAM, which can be done by clicking on the *New Provider* button in the *Infrastructure Providers* subtab of the *Orchestration tab* (see figure 5).

After selecting the Infrastructure Provider (Amazon EC2, HP Cloud, Rackspace or Physical provider), we can fill in the account details as shown in figure 6. For the **Amazon EC2** provider, the following fields are needed:

- *name*: A custom string with which we would like to identify the provider. In this example, we use EC2.
- *username*: The API key generated by Amazon; accessible from Amazon's web console.
- *password*: The API code generated by Amazon; accessible from Amazon's web console.

In case of different providers, different authentication details are needed. The **HP Cloud** is an OpenStack-based provider, so several more items need to be specified:

- *name*: A custom string; we use HP.

Figure 6: Registering an Amazon EC2 provider in WombatOAM

- *username*: The HP Cloud username.
- *password*: The HP Cloud password.
- *auth\_url*: The URL that serves the OpenStack authentication API. This is available in the HP Cloud web console. We use the following URL:  
<https://region-a.geo-1.identity.hpcloudsvc.com:35357/v2.0/tokens>.
- *tenant\_name*: The *tenant* is also an OpenStack concept. It is a string that can be set in the HP Cloud web console. The same string needs to be given to WombatOAM when creating the provider.
- *service\_region*: A cloud provider can have several clouds, which are identified by their region code. The HP Cloud has two such regions, US West (region code `region-a.geo-1`) and US East (region code `region-b.geo-1`). We use the US West region, so we specify `region-a.geo-1` in this field.

**Rackspace**, the third cloud provider that WombatOAM supports needs four pieces of information:

- *name*: A custom string; we use Rackspace.
- *username*: The Rackspace username.
- *password*: The Rackspace password.
- *service\_region*: Region code, as in the HP Cloud's case. We use the London cloud, therefore we specify `LON`.

WombatOAM will use these details to communicate with the REST API of the cloud providers. In case of choosing the **Physical provider**, only two fields need to be filled in:

UUID	Name	Type
<input type="checkbox"/> 480fc543-f446-441c-bc9b-4f7e8dcfaf14	Local	physical
<input type="checkbox"/> c056a8f3-8e63-45a7-b6ca-a9da5853e0c6	HP	hp
<input type="checkbox"/> 9d0a3231-5e96-4fb7-b6c4-3ad3c2750825	EC2	ec2
<input type="checkbox"/> f958c040-b3fc-45b8-b1cd-b39f12ec8e75	Rackspace	rackspace

+ New Provider

Popups enabled

Figure 7: The *Infrastructure Providers* subtab showing the four providers that we have registered

- *name*: A custom string; we use `Local`.
- *servers*: The list of the host names or IP addresses of the servers that we want to deploy on when using this provider. We specify `78.47.223.138`: the IP address of a server at our disposal.

In case of a physical provider, WombatOAM requires that the machines are accessible from the WombatOAM machine via SSH using passwordless authentication. This is naturally not a prerequisite in case of cloud providers, where WombatOAM will upload the SSH key to the provider, and specify it when provisioning the instances, thereby ensuring passwordless authentication.

After creating the four providers described above, the *Infrastructure Providers* subtab will show their most important details in a table, as figure 7 illustrates.

If we wish to use WombatOAM's REST API instead of the web dashboard, listing 5 shows how to create the HP provider using a REST request. (We picked the HP one because that is the most complex.)

#### Listing 5: Creating the HP provider using a REST request

```
$ AUTH_URL="https://region-a.geo-1.identity.hpcloudsvc.com:35357/v2.0/tokens"

# Perform the request. The WombatOAM server will reply with a JSON object
# containing a few details about the new provider.
$ curl -X POST http://<WombatServerUrl>/api/orch/providers \
  -H 'Content-Type: application/json' \
  -H 'Accept: application/json' \
  -d '{"type": "virtual",
      "vprovider": "hp",
      "name": "HP",
      "username": "<UserName>",
      "password": "<Password>",
      "auth_url": "$AUTH_URL",
      "tenant_name": "<TenantName>"}
```

```

        "service_region": "region-b.geo-1"}'
{"uuid":"c056a8f3-8e63-45a7-b6ca-a9da5853e0c6", "name":"HP", "type":"hp",
 "driver":"wo_orch_elibcloud_driver", "opts":{"name":"HP", "username":"...",
 "password":"...", "type":"virtual", "vprovider":"hp",
 "auth_url":"https://region-a.geo-1.identity.hpcloudsvc.com:35357/v2.0/tokens",
 "tenant_name":"...", "service_region":"region-a.geo-1"}}

# Store the id of the new provider in a variable; this will be useful when
# creating the node family.
$ PROVIDER_ID_HP=c056a8f3-8e63-45a7-b6ca-a9da5853e0c6

```

---

#### 4.1.2 Creating and Uploading an Erlang Release

In the previous section, we created a provider to deploy an Erlang release to; in this section, we create the Erlang release itself, and upload it to WombatOAM.

As the example Erlang release, we use Riak [38], which is a distributed database written in Erlang. We have chosen Riak because it is a well-known and successful Erlang project, it is open-source, and it is a database application which typically runs for a long time, so WombatOAM is a perfect fit for deploying and monitoring it.

Erlang/OTP has a well-defined format for releases [4]: a release is a directory that contains certain files (applications, release resource file, boot script, etc.) in a certain directory layout. It is often packaged in a `tar.gz` file, which is called a *release archive*.

WombatOAM deploys Erlang/OTP release archives, but only after making a few minor changes to them. After these changes, we call the release archive a WombatOAM Orchestration-compatible release, or WombatOAM-compatible release for short. To make any Erlang release WombatOAM-compatible, the following changes should be applied:

1. The `vm.args` file needs to be edited, and the hard coded node name needs to be replaced with the string `{{ node_name }}`. The release should not contain a hard coded node name anywhere else. This is required because WombatOAM will give different names to different nodes after deploying them. Otherwise they couldn't form an Erlang cluster; in an Erlang system where the nodes are connected, each node needs to have a unique name. WombatOAM will replace the `{{ node_name }}` string with the actual node name (which will be `<generated_uuid>@<ip-address-of-instance>`) when deploying new nodes. The path to `vm.args` is given to WombatOAM when uploading the release.
2. If we want to connect the nodes after deployment, a bootstrap script needs to be added to the release, which will be invoked when a new node is added to a node family. The bootstrap script in the Riak release archive is shown in listing 6. It receives the name of a node already in the cluster in its first argument, and connects to that node using the `riak-admin cluster join` command.
3. The TCP ports allocated for distributed Erlang traffic need to be narrowed down. The port range can be usually set in the `sys.config` (or sometimes `app.config`) file within the release archive with the settings shown in listing 7. The same range should be given to WombatOAM when the node family is defined, so that WombatOAM will ask the cloud providers to open the firewall ports that are used by the nodes to communicate with WombatOAM and each other. This point is not as strict as the previous too: if the provider is such that it allows all ports to go through the firewalls, and we want to keep it that way, or we are using our own internal network where that is the case, narrowing down the port range is unnecessary. Also, in some releases, the port range may already be narrowed down.

Listing 6: The bootstrap script in the WombatOAM-compatible Riak release archive

```
#!/bin/bash

BOOTSTRAP_NODE=$1
DIRNAME=$(dirname $0)

echo $($DIRNAME/riak-admin cluster join $BOOTSTRAP_NODE)
echo $($DIRNAME/riak-admin cluster plan)
echo $($DIRNAME/riak-admin cluster commit)
```

Listing 7: Configuring a release to use a given port range for Erlang distribution

```
{kernel, [{inet_dist_listen_min, 6000}, # lowest port of the range
           {inet_dist_listen_max, 7999}]} # highest port of the range
```

After the WombatOAM-compatible release archive has been created, it can be uploaded to WombatOAM by clicking on the *New Release* button on the *Releases* subtab. The following fields are available (see also the web form on figure 8):

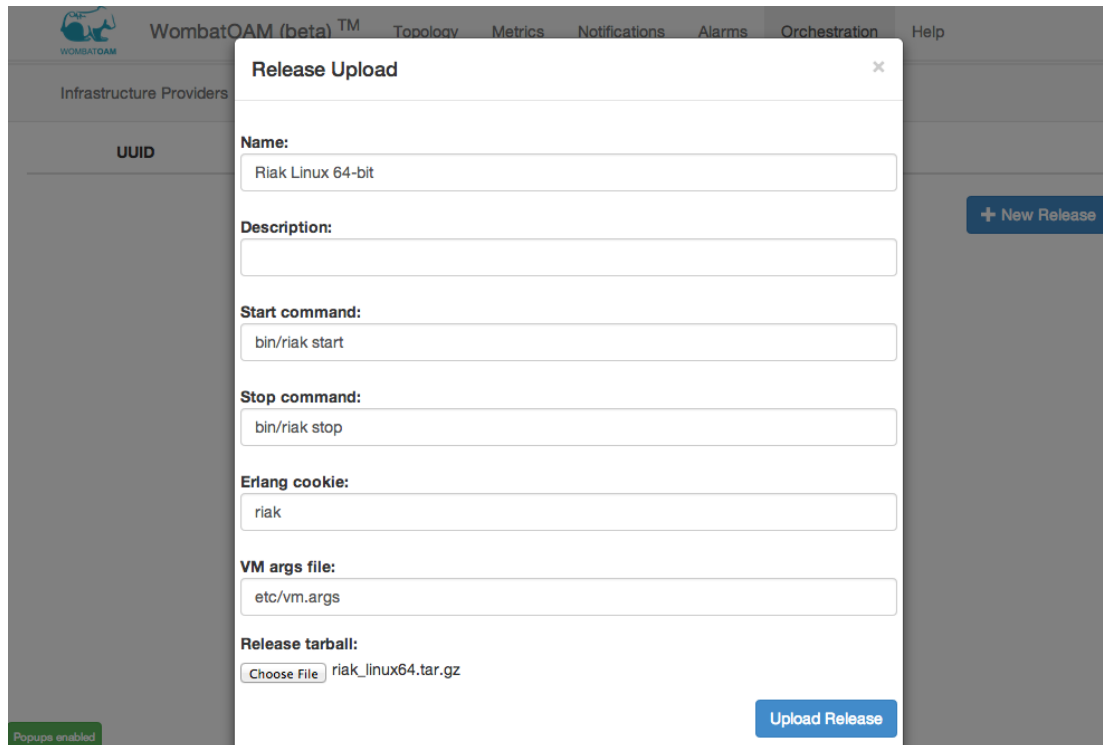
- *Name*: A custom string with which we would like to identify the release. In this example, we use Riak Linux 64-bit.
- *Description*: A custom string that can contain a more detailed description about the release.
- *Start command*: WombatOAM will use this command to start the release once deployed. We specify `bin/riak start`. The `bin/riak` script is inside the release archive; it is part of Riak.
- *Stop command*: WombatOAM will use this command to stop the release. We specify `bin/riak stop`.
- *Erlang cookie*: WombatOAM will use this cookie to connect to the deployed node. We specify `riak`, because that is the cookie defined in the `etc/vm.args` file within the release archive.
- *VM args file*: We specify `etc/vm.args`, because that is the path to the `vm.args` file within the release archive.
- *Release tarball*: We need to click on the *Choose file* button below this label and select the release archive from the disk.

After uploading the release, the *Releases* tab will show the most important details of the release, as can be seen on figure 9.

Listing 8 shows how to upload the Riak release using a REST request instead of the WombatOAM Web Dashboard. The request body (as always with WombatOAM) is a JSON object, and the `release` field of this object is the base64-encoded value of Riak's Erlang release archive. This is a few dozen megabytes, so we don't want to supply it as a command line argument to `curl`. Therefore we create a file (`/tmp/release.request.json`) to hold the JSON object (including the release), and when calling `curl`, we use its `@FILE` notation to specify that the body of the request should be read from this file.

Listing 8: Uploading a Riak release using a REST request

```
$ echo -n '{"name": "Riak Linux 64-bit",
           "start_cmd": "bin/riak start",
           "stop_cmd" : "bin/riak stop",
           "cookie": "riak",
           "node_name_templates": ["etc/vm.args"]},'
```

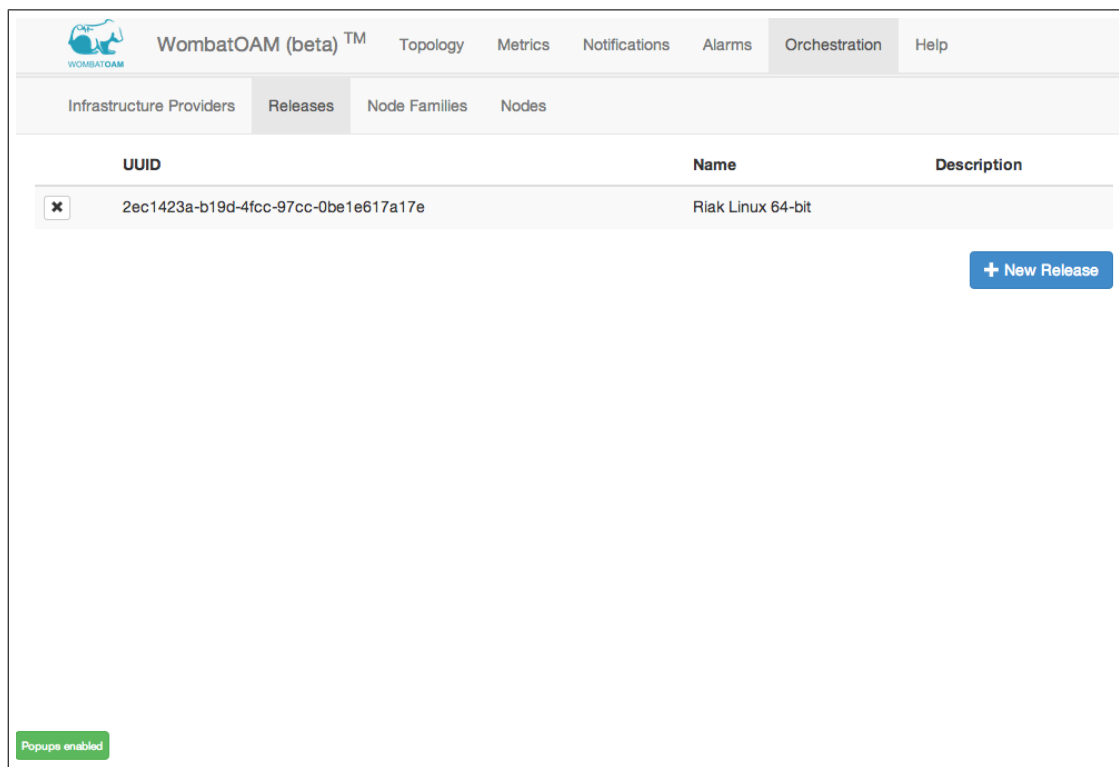


The screenshot shows the 'Release Upload' dialog box in the WombatOAM (beta) interface. The dialog has a close button (X) in the top right corner. It contains several input fields for release configuration:

- Name:** Riak Linux 64-bit
- Description:** (empty field)
- Start command:** bin/riak start
- Stop command:** bin/riak stop
- Erlang cookie:** riak
- VM args file:** etc/vm.args
- Release tarball:** Choose File riak\_linux64.tar.gz

At the bottom right of the dialog is an 'Upload Release' button. In the background, the main interface shows the 'Orchestration' tab selected, with sub-tabs for 'Infrastructure Providers', 'Releases', 'Node Families', and 'Nodes'. A '+ New Release' button is visible on the right side of the background interface.

Figure 8: Uploading a release to WombatOAM



The screenshot shows the 'Releases' subtab in the WombatOAM (beta) interface. The 'Orchestration' tab is selected in the top navigation bar. The 'Releases' subtab is active, showing a table of uploaded releases. The table has three columns: 'UUID', 'Name', and 'Description'. A single release is listed with the UUID '2ec1423a-b19d-4fcc-97cc-0be1e617a17e' and the Name 'Riak Linux 64-bit'. A '+ New Release' button is located at the bottom right of the table. A 'Popups enabled' status indicator is visible in the bottom left corner.


UUID	Name	Description
 2ec1423a-b19d-4fcc-97cc-0be1e617a17e	Riak Linux 64-bit	

Figure 9: The *Releases* subtab showing the release that we have uploaded



```

        "release": "" > /tmp/release_request.json
$ base64 riak_linux64.tar.gz >> /tmp/release_request.json
$ echo -n '{}' >> /tmp/release_request.json

# Perform the request itself. The WombatOAM server will reply with a JSON object
# containing a few details about the uploaded release.
$ curl -X POST http://<WombatServerUrl>/api/orch/releases \
      -H 'Content-Type: application/json' \
      -H 'Accept: application/json' \
      -d @/tmp/release_request.json
{"uuid":"2ec1423a-b19d-4fcc-97cc-0bele617a17e","name":"Riak Linux 64-bit",
 "state":"UPLOADED","path":"../2ec1423a-b19d-4fcc-97cc-0bele617a17e.tar.gz",
 "templates":["etc/vm.args"],"cookie":"riak","start_cmd":"bin/riak start",
 "stop_cmd":"bin/riak stop"}

# Store the id of the new release a variable; this will be useful when creating
# the node family.
$ RELEASE_ID=2ec1423a-b19d-4fcc-97cc-0bele617a17e

```

---

### 4.1.3 Creating Node Families

After having created the providers and a release, we need to connect them. We need to create a node family, which can deploy a certain release to certain providers. It also stores some other information, for example which virtual machine image to use, which ports to open, etc.

The node family can be created by clicking on the *New Family* button on the *Node families* subtab. The web form to be filled in is shown on figure 10, and detailed below:

- *Name*: A custom string with which we would like to identify the node family. In this example, we use `Riak Cluster`.
- *Release*: The release to be deployed on nodes in this node family. The form lets us select a release from the releases previously uploaded. Select `Riak Linux 64-bit` (which is the only release we have now in the list).
- *Firewall rules*: Firewall rules that define network ports to be open on the virtual machine instances. We specify the following string:  
`tcp:4369:4369,tcp:8097:8099,tcp:8087:8087,tcp:8069:8069,tcp:6000:7999`  
 TCP port 4369 is the port used by EPMD (Erlang Port Mapper Daemon). TCP ports 8097, 8098, 8099, 8087 and 8069 are used by Riak. Finally, ports between 6000 and 7999 are used for Erlang distribution, as we configured in section 4.1.2. These are all inbound ports: the cloud providers we used didn't restrict the outbound ports by default.
- *Node selection strategy*: When a new node joins a cluster, its bootstrap script gets another node as a parameter. This field defines how to select that other node. Currently the only option is `Random`, which means that a random node is selected. We choose that option.
- *Bootstrapping strategy*: Here we can choose what WombatOAM should do when it deploys a new node to a node family. The two available options are `Distributed Erlang` and `Custom`. With the former, WombatOAM only makes sure that the nodes are connected with each other via Erlang Distribution. With the latter, we can specify a script to be executed. We choose `Custom`, after which a new text field appears, called *Bootstrapping Options*.
- *Bootstrapping options*: Here we can give a more detailed description of what should happen when a new node is deployed into a node family. We type the string `cmd:bin/bootstrap`, which means

that the `bin/bootstrap` script (included in the release archive) should be executed, with its only parameter being the name of the node selected by the chosen *node selection strategy*.

- *Deployment Domains*: We can add a number of domains to the node family by clicking on the *Add Domain* button and filling in the following fields for each of them. A deployment domain is a part of the node family which uses the same provider with the same virtual image and hardware profile. When a node is deployed to a node family, it is actually deployed into a domain.
  - *Infrastructure provider*: Each domain uses one provider. For the first domain, we select EC2. Below we will configure domains for all the other providers too.
  - *SSH user*: The user with which WombatOAM can SSH into virtual machine instances created by this provider. We will use an Ubuntu image (see the next point), and the username is always `ubuntu` on such images, so we type `ubuntu`.
  - *Virtual disk*: The system image that should be placed in the virtual machine instance. The available images are listed in Amazon’s web console (can be accessed only after logging in). We use `ami-0cdf4965`, which is a 64-bit Ubuntu server image.
  - *Hardware profile*: The hardware profile of the virtual machine instances. Different providers use different terminology: Amazon calls them *instance types* [22]; OpenStack, HP Cloud and Rackspace calls them *flavors* [18, 9, 25]; Libcloud calls them *sizes* [16]. We use `t1.micro`, which is a rather small instance type with the following specification: 1 virtual CPU, 0.613 GB RAM, “very low” networking performance, and EBS-only storage [22].

For the other providers, we specify the following values:

- HP Cloud:
  - \* Infrastructure provider: HP
  - \* SSH user: `ubuntu`
  - \* Virtual disk: `2925ec9a-3c29-4960-acd8-ffe33007ad62`
  - \* Hardware profile: `100`
- Rackspace:
  - \* Infrastructure provider: `Rackspace`
  - \* SSH user: `myuser`. The username can be any string (as long as it is different from `root`), because there is no non-root user on the instances provided by Rackspace, so WombatOAM creates the username specified by the user.
  - \* Virtual disk: `5cc098a5-7286-4b96-b3a2-49f4c4f82537`
  - \* Hardware profile: `2`
- Physical provider:
  - \* Infrastructure provider: `Local`
  - \* SSH user: `localuser`. Here the SSH user must be the user with which WombatOAM can log in to the machines.
  - \* Virtual disk: `anything`. This field is ignored (since WombatOAM won’t provision a machine).
  - \* Hardware profile: `anything`. This field is ignored (since WombatOAM won’t provision a machine).

This procedure will not only create the node family in WombatOAM, but upload the SSH key to all the cloud providers and create security groups there with the security rules above. The name of the SSH key and the security groups are generated from the UUID of the node family. When the node family is deleted from WombatOAM, the SSH keys and the security rules are also automatically deleted (by WombatOAM) from the cloud providers.

After uploading the node family, the *Node families* tab will show the most important details of the node family, as can be seen on figure 11.

Erlang releases often include the Erlang runtime system in its compiled form, which is architecture-dependent. Thus e.g. a Riak release file for 64-bit Linux will not work on 32-bit Linux or 64-bit Mac OS X. If WombatOAM were to be used to deploy nodes in a node family on machines with different architectures, it would be a logical improvement to implement the ability to specify several releases for one node family for different architectures.

The node family can also be created with the REST requests shown in listing 9. In the last request of the listing, we add a domain for the EC2 provider; the other providers can be added in the same way.

#### Listing 9: Creating a node family using REST requests

```
# Create the family without any domains. The request returns a JSON object which
# contains the UUID of the new family.
$ curl -X POST http://<WombatServerUrl>/api/topo/node_families \
      -H 'Content-Type: application/json' \
      -H 'Accept: application/json' \
      -d '{"name": "Riak Cluster",
          "bootstrap_node_selection": "random",
          "bootstrap_strategy": "custom",
          "bootstrap_strategy_opts": [{"cmd": "bin/bootstrap"}]}'
{"id": "91473e89-bfc9-4021-84cc-4a70ed0955e8", "name": "Riak Cluster",
 "description": [], "node_selection": "random", "bootstrap_strategy": "custom",
 "bootstrap_strategy_opts": {"cmd": "bin/bootstrap"}, "neighbors": [], "nodes": [],
 "plugins_opts": [], "deleted": false}

# Store the id of the new node family.
$ NF_ID="91473e89-bfc9-4021-84cc-4a70ed0955e8"

# We connect the node family with the release that we have uploaded previously.
# The RELEASE_ID variable was set in a previous listing.
$ curl -X POST http://<WombatServerUrl>/api/orch/node_families/$NF_ID/release \
      -H 'Content-Type: application/json' \
      -H 'Accept: application/json' \
      -d '{"release": "'$RELEASE_ID'"}';

# We set the firewall rules for the node family.
$ curl -X POST http://<WombatServerUrl>/api/orch/node_families/$NF_ID/firewall \
      -H 'Content-Type: application/json' \
      -H 'Accept: application/json' \
      -d '[{"protocol": "tcp", "port_from": "4369", "port_to": "4369"},
          {"protocol": "tcp", "port_from": "8097", "port_to": "8099"},
          {"protocol": "tcp", "port_from": "8087", "port_to": "8087"},
          {"protocol": "tcp", "port_from": "8069", "port_to": "8069"},
          {"protocol": "tcp", "port_from": "6000", "port_to": "7999"}]'

# We create a domain in the node family for the EC2 provider. We stored the
# provider id beforehand in the PROVIDER_ID_EC2 variable.
$ curl -X POST http://<WombatServerUrl>/api/orch/node_families/$NF_ID/domains \
      -H 'Content-Type: application/json' \
      -H 'Accept: application/json' \
      -d '{"provider": "'$PROVIDER_ID_EC2'",
          "ssh_user": "ubuntu",
```

Node Family Definition

Name:

Riak cluster

Release:

Riak Linux 64-bit

Deployment Domains:

×

EC2t1.microami-0cdf4965ubuntu

Add Domain

Infrastructure Provider:

HP

SSH User:

ubuntu

Virtual disk:

2925ec9a-3c29-4960-acd8-ffe33007ad62

Hardware profile:

100

Add Domain

Firewall rules:

tcp:4369:4369,tcp:8097:8099,tcp:8087:8087,tcp:8069:8069,tcp:6000:7999

Node selection strategy:

Random

Bootstrapping strategy:

Custom

Bootstrapping options:

cmd:bin/bootstrap

Create Family

Figure 10: Creating a new node family in WombatOAM. We have just configured a domain for the EC2 provider, and we are in the process of configuring one for the HP provider.

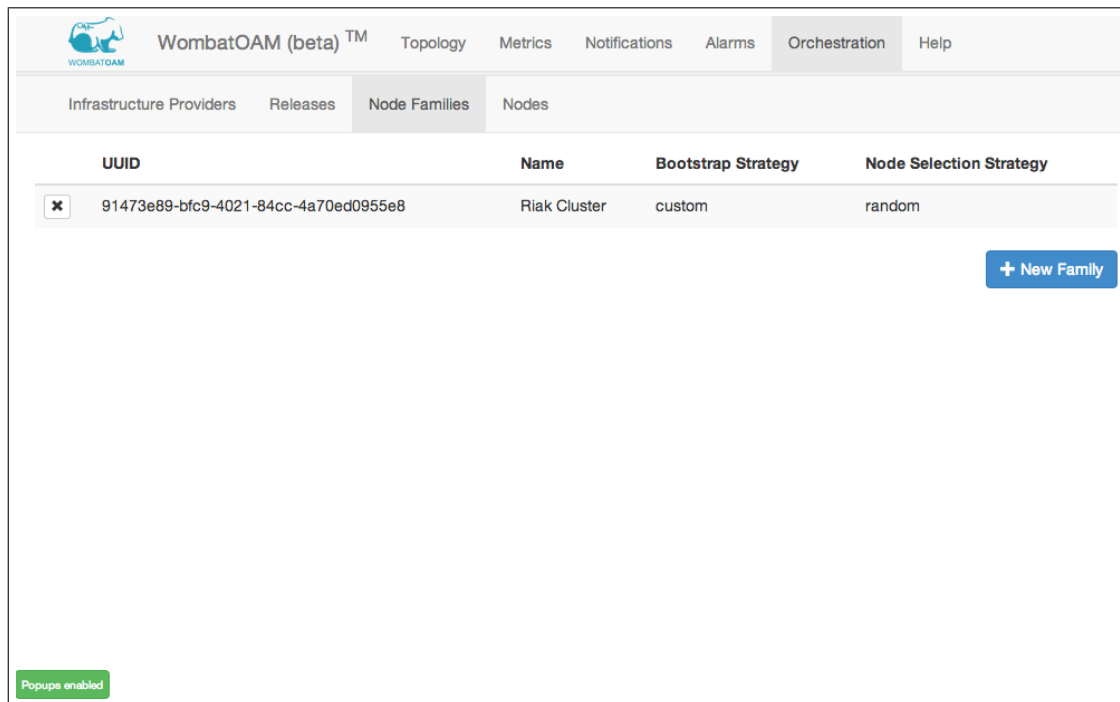


Figure 11: The *Node Families* subtab showing the node family that we have created

```
"vdisk": "ami-0cdf4965",
"hw_profile": "t1.micro",
"type": "virtual"}
```

#### 4.1.4 Deploying a Node

In the previous step, we have defined a node family, which stores many details that are needed for creating nodes (i.e. instantiating the node family). In this step, we deploy two nodes on each cloud provider that we have configured (Amazon EC2, HP Cloud, Rackspace), and one node on our own server.

The fourth subtab (*Nodes*) lets us create nodes after clicking on the *Deploy node(s)* button. The short form that pops up contains the following fields (see also figure 12):

- *Node Family*: We can select the node family that we want to deploy the node into. This will determine the release to be used, the firewall rules, the bootstrap strategy, and so on. We select the Riak Cluster family.
- *Provider*: We can select one of the providers used by the node family. The nodes will be deployed using this provider. We will deploy the first two nodes into Amazon EC2, so we select EC2.
- *Amount*: The number of nodes to deploy. We type 2.
- *Automatically start node(s) after deployment*: If this checkbox is checked, WombatOAM will not only copy the release archive file to the instances, but it will also start the nodes (see below).

After initiating the deployment of the EC2 nodes, we use the same process to deploy two nodes on the HP Cloud, two nodes on Rackspace, and finally a node to our own server.

At this point the *Nodes* tab will show our 7 nodes, all of them in UNINIT state. In the background, WombatOAM is performing the following steps for each node:

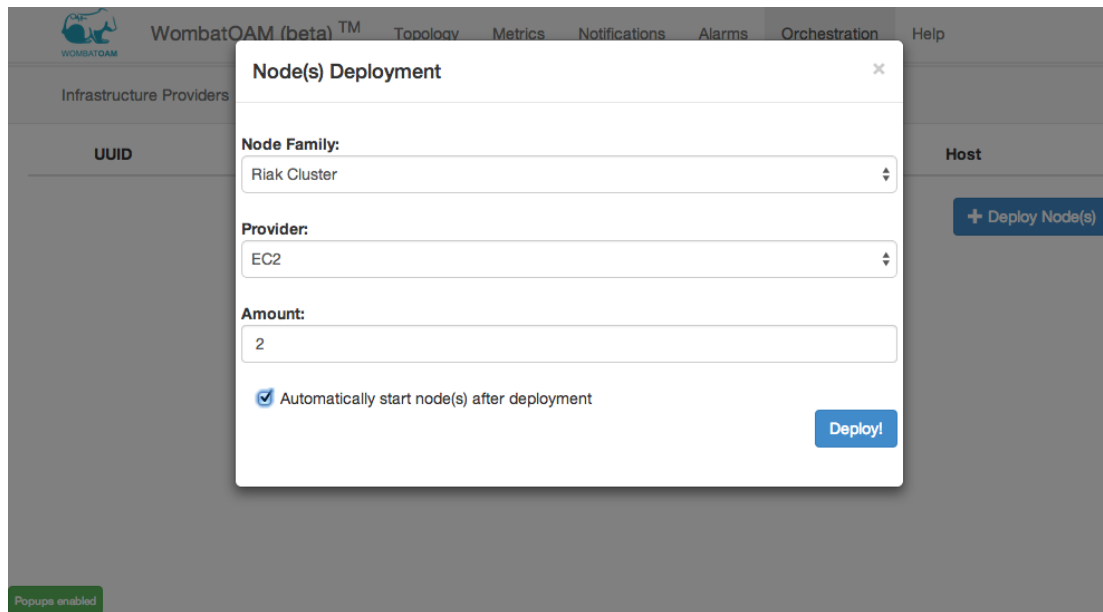


Figure 12: Deploying two nodes to Amazon EC2

1. Ask the provider to create the virtual machine instance (except in case of the physical provider), using the image, hardware profile, SSH keys, security groups defined by the node family.
2. Do any additional configuration that might be necessary, e.g. on Rackspace a non-root user needs to be created.
3. Create a custom release archive file by replacing the `{{ node_name }}` string in the `vm.args` file with the node name to be used (`<generated_uuid>@<ip_address_of_instance>`), upload this release archive file to the instance and unpack it, and set the node state from `UNINIT` to `STOPPED`.
4. If the *Automatically start node(s) after deployment* checkbox is checked, proceed with the next step. Otherwise finish; the deployment is done.
5. Start the Erlang release on the instance with the start command of the release (see section 4.1.2).
6. If this is not the first node in the node family that is deployed, use the specified node selection strategy, bootstrapping strategy and bootstrapping options to make the node connect to its family. (See section 4.1.3 for the details.)
7. Set the node state from `STOPPED` to `DOWN` and add the node to WombatOAM's node manager. The node manager will periodically try to connect to the node via Erlang Distribution. (By default, the period is 2 seconds in the beginning, then gradually increases up to 30 seconds. WombatOAM will not stop trying as long as the node is in `DOWN` state).
8. When the node manager succeeds in connecting the node, its state will become `UP`, and WombatOAM starts gathering metrics, logs and alarms from the node. Whenever the connection between WombatOAM and the node breaks, the node's state will change to `DOWN`, and the node manager will periodically try to re-establish the connection.
9. If any error occurs along the steps above, the node's state will change to `ERROR`.

After a few minutes, all nodes will go into `UP` state, as shown in figure 13.

	UUID	Status	Family	Domain	Host
Action ▾	33639166-1ea4-42bc-b09c-a9df591c41fb	UP	91473e89-bfc9-4021-84cc-4a70ed0955e8	de55c609-edf2-4047-bac9-8c35014ed22e	54.80.88.250
Action ▾	43646bf1-a3a4-4e95-bc14-6a6bf74e0882	UP	91473e89-bfc9-4021-84cc-4a70ed0955e8	de55c609-edf2-4047-bac9-8c35014ed22e	107.20.58.192
Action ▾	f6c6dac3-1f47-4da0-aec1-fbf453500854	UP	91473e89-bfc9-4021-84cc-4a70ed0955e8	46f731ef-5939-42df-942c-e975f9c67b5a	78.47.223.138
Action ▾	72218b3e-960b-49ce-ac66-2a4b56f0756c	UP	91473e89-bfc9-4021-84cc-4a70ed0955e8	5b8adca2-6540-455e-a6f4-49150a74190d	95.138.172.39
Action ▾	fa45ccda-da4a-436c-979f-c14ca0f77889	UP	91473e89-bfc9-4021-84cc-4a70ed0955e8	5b8adca2-6540-455e-a6f4-49150a74190d	95.138.163.243
Action ▾	10a38ed6-e818-4ebc-b5cb-443037965dc1	UP	91473e89-bfc9-4021-84cc-4a70ed0955e8	165fc9ef-eeff-4f57-920c-a864c16c0d6e	15.125.85.201
Action ▾	0406b168-6c04-4128-b263-f41031fb77b7	UP	91473e89-bfc9-4021-84cc-4a70ed0955e8	165fc9ef-eeff-4f57-920c-a864c16c0d6e	15.125.85.188

Figure 13: The *Nodes* subtab showing the nodes that we have deployed. In the *Domain* column, there are 4 different domains; they use 4 different providers.

Nodes can be stopped by selecting the *STOP* action, which means that WombatOAM will execute their stop command (see section 4.1.2). When this happens, the node will go into *STOPPED* state, and can be restarted by selecting the *START* action.

Nodes in *DOWN* state can also be restarted. Users need to be careful with this option, because if a node is *DOWN*, that only means that WombatOAM cannot reach the node. That can mean either that the node is not running, or that the network connection between its host and WombatOAM doesn't work.

Naturally, the nodes can be deployed via the REST interface too, as shown in listing 10, which deploys two nodes on Amazon EC2.

#### Listing 10: Deploying nodes using REST requests

```
# Deploy 2 nodes into our Amazon EC2 provider. The PROVIDER_ID_EC2 and NF_ID
# variables were set in the previous listings.
$ curl -X POST http://<WombatServerUrl>/api/orch/node_families/$NF_ID/nodes \
  -H 'Content-Type: application/json' \
  -H 'Accept: application/json' \
  -d '{"amount": 2,
      "autostart": true,
      "provider": "'$PROVIDER_ID_EC2'"}'
```

Let's test our heterogeneous deployment. Let's verify that the 7-node Riak cluster we have deployed over 4 different providers works indeed as one cluster. To do that, we log in to one of the instances and issue the `riak-admin member-status` command, which shows us all the nodes in the Riak cluster. The IP address of the instances is shown in the web dashboard. The commands we used for this test and their output are shown in listing 11.

**Listing 11: Checking our newly deployed Erlang system**

```
# We log into one of the host machines
$ ssh ubuntu@78.47.223.138

# We have only one non-hidden directory, which is the directory containing the
# Erlang release.
$ ls
f6c6dac3-1f47-4da0-aec1-fbf453500854

# We use Riak's member-status command to list all members of this cluster. We
# can see all seven Riak instances, which means that WombatOAM automatically
# made them connect to each other.
$ */bin/riak-admin member-status
===== Membership =====
Status      Ring      Pending   Node
-----
joining     0.0%      --        '10a38ed6-e818-4ebc-b5cb-443037965dc1@15.125.85.201'
valid       15.6%      --        '0406b168-6c04-4128-b263-f41031fb77b7@15.125.85.188'
valid       17.2%      --        '33639166-1ea4-42bc-b09c-a9df591c41fb@54.80.88.250'
valid       17.2%      --        '43646bf1-a3a4-4e95-bc14-6a6bf74e0882@107.20.58.192'
valid       17.2%      --        '72218b3e-960b-49ce-ac66-2a4b56f0756c@95.138.172.39'
valid       17.2%      --        'f6c6dac3-1f47-4da0-aec1-fbf453500854@78.47.223.138'
valid       15.6%      --        'fa45ccda-da4a-436c-979f-c14ca0f77889@95.138.163.243'
-----
Valid:6 / Leaving:0 / Exiting:0 / Joining:1 / Down:0
```

Since we can see all 7 nodes in the list, that means that they were able to form one cluster and they are able to communicate with each other.

(Note that this is not the recommended way to use Riak, which runs best on physical machines, and even when virtualized, its host machines should be in the same data center. We used Riak here to demonstrate how to deploy a distributed Erlang system in a heterogeneous way.)

#### 4.1.5 Stopping the Cluster and Cleaning Up

There are two important steps with regard to WombatOAM Orchestration when we want to stop a deployed Erlang system:

1. *Delete the nodes:* We select the DELETE action next to one of the nodes. WombatOAM will stop the Erlang node and ask the provider to destroy the virtual machine instance. We repeat this for each node.
2. *Delete the node family:* We click on the x button next to the node family. WombatOAM will ask the provider to delete the SSH keys and the security groups that were created for this node family.

The release and the providers might also be deleted, but that is not necessary, since those objects don't create other (remote) objects that need to be cleaned up.

## 4.2 An overview of WombatOAM's monitoring features

In this section, we will look at the monitoring features of WombatOAM. We start with the Topology service (section 4.2.1), which is about the set of nodes we handle and their status. We continue with the Metrics service (section 4.2.2), the Notifications service (section 4.2.3) and the Alarms service (section 4.2.4), which collect, store and display metrics, notifications and alarms from the managed nodes. These services work both with nodes deployed by WombatOAM and with nodes that were added to WombatOAM after they were started.



WombatOAM (beta)™

Topology Metrics Notifications Alarms Orchestration Help

View Node Info Add Node View Node Graph

### Add new node

**Node name**

wombat@127.0.0.1

**Cookie**

wombat

☐ Discover connected nodes

Add node

Popups enabled

Figure 14: Add WombatOAM’s own node to WombatOAM

#### 4.2.1 Topology

The Topology tab is about maintaining which set of nodes and node families we manage from WombatOAM. From the Topology tab, we can connect to already running Erlang nodes. For example it is a good practice to add the WombatOAM node itself to WombatOAM, which can be done on the *Add Node* subtab (see figure 14). Afterwards, on most tabs the list of node families and nodes will be shown in the left side of the screen in the node tree. Figure 15 shows the *View Node Info* subtab, where we see that we have only one node family in the system (called `wombat pre-0.7.1`; see the explanation below) and only one node (called `wombat@127.0.0.1`). The node tree also shows the state of each node (currently our only node is UP).

When using WombatOAM Orchestration, *node families* are complex entities that know which Erlang release they should run, on which providers, with which firewall rules, etc. When we add nodes already running to WombatOAM, it will automatically generate node families for these nodes, because each node must be the member of a node family. But these automatically generated node families don’t have a release, providers, firewall rules, etc. attached to them, so they cannot be used for deploying new nodes. They are simply used for grouping the nodes. These node families are generated based on the boot script ids of the Erlang nodes: their name (displayed on the dashboard in the node tree) is actually a stringified version of the boot script id. If two nodes have the same boot script id, they will belong to the same (auto-generated) node family.

If we have a set of connected nodes, for example a Riak cluster, we can tick the *Discover connected nodes* checkbox in the *Add new node* form. In that case WombatOAM will automatically add all nodes connected to the specified node that use the same cookie; it will also add the nodes connected to those nodes; and so on, recursively.

Finally, the Topology tab has subtab called *View Node Graph*; here the node information is visualized as a graph, whose nodes can be dragged and dropped, opened and closed (see figure 16).

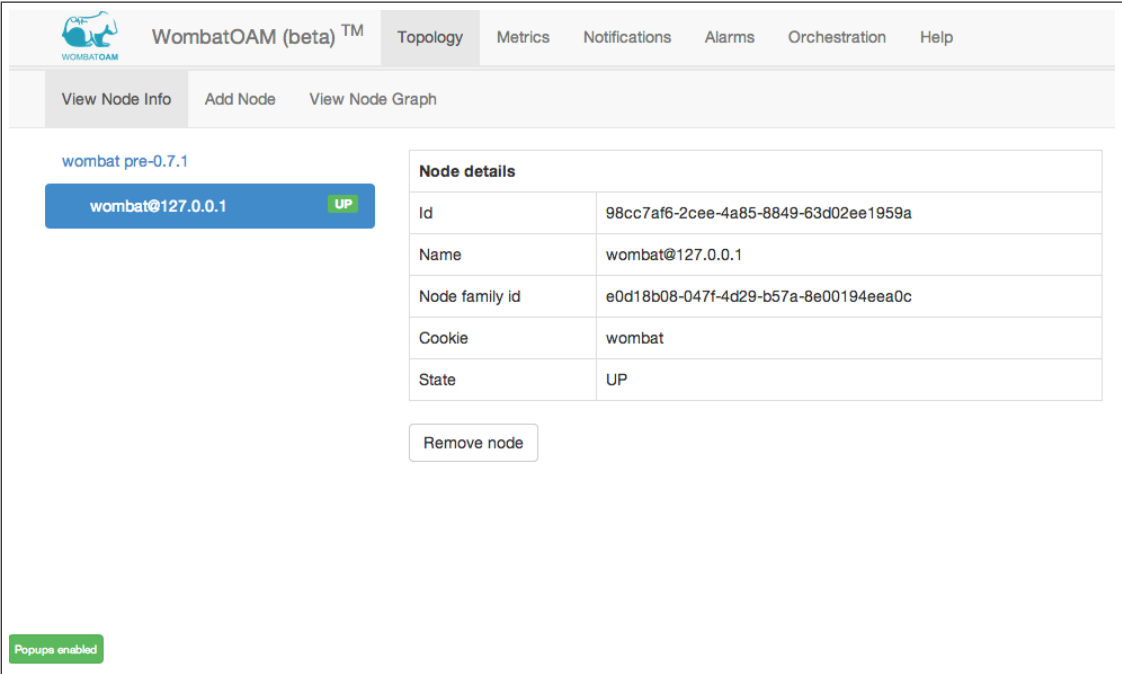


Figure 15: Show topology details about a node. A node can be removed by clicking on the *Remove* button. This will not stop the node; only WombatOAM will stop monitoring it.

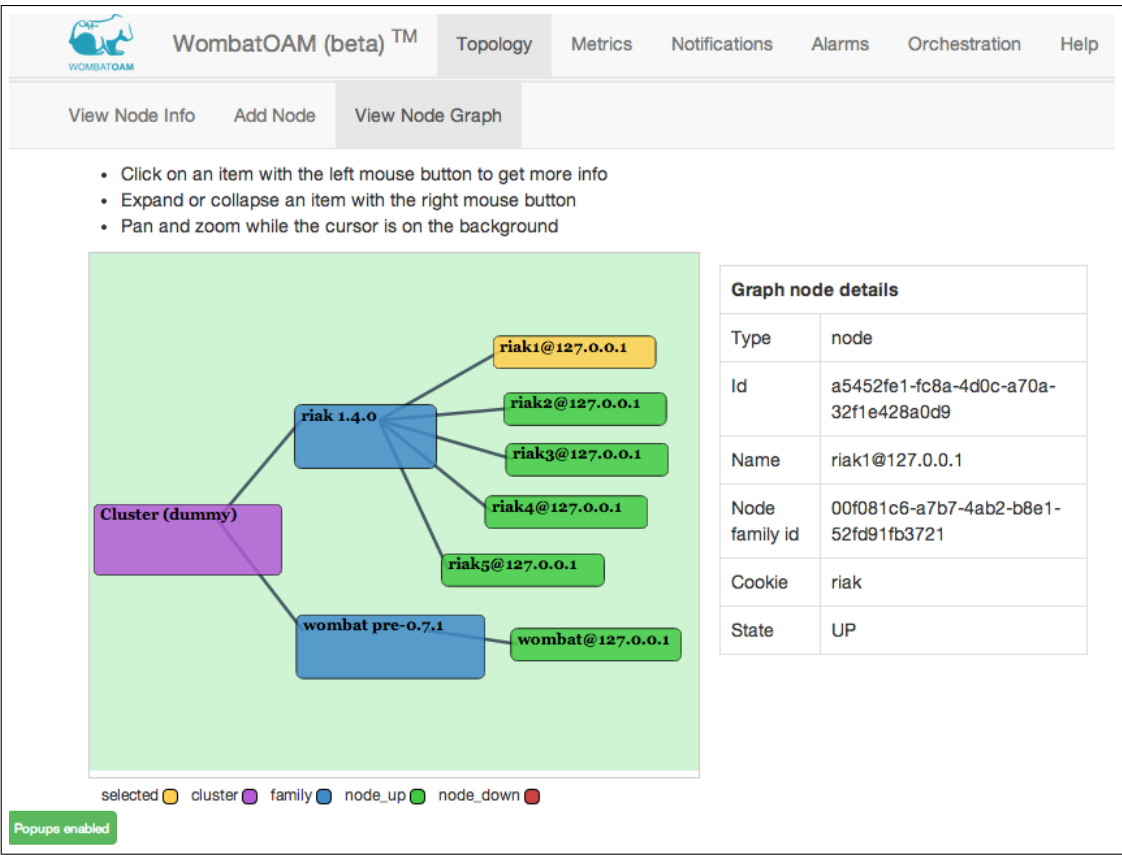


Figure 16: The *View Node Graph* subtab shows the nodes in a graph

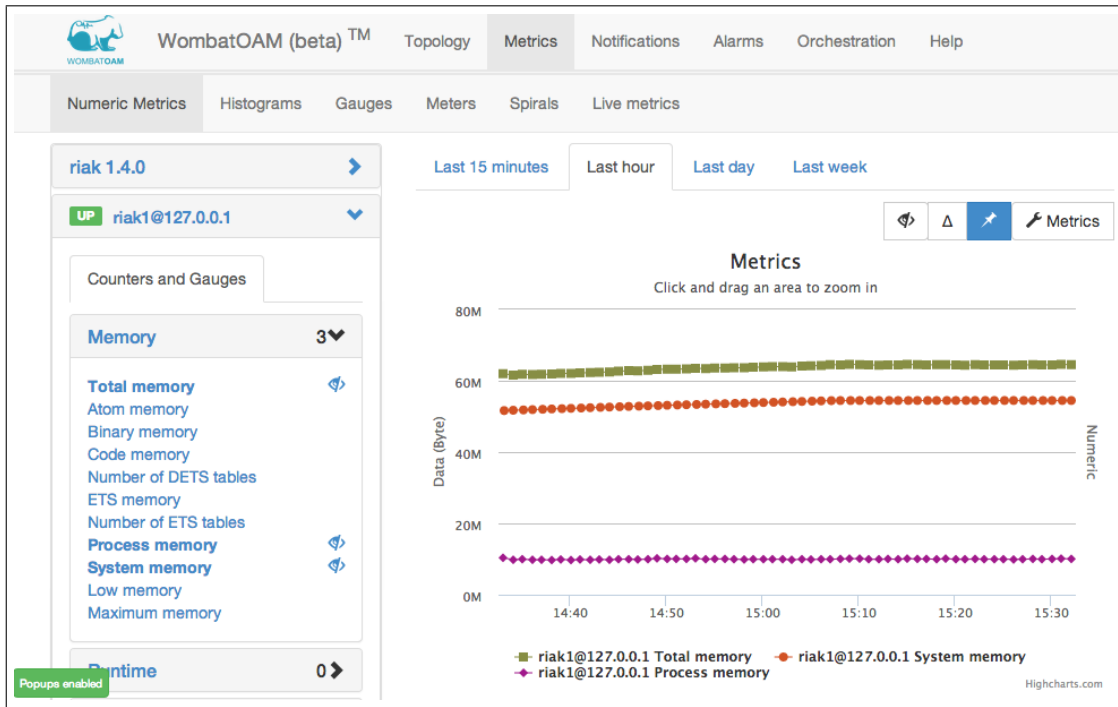


Figure 17: The web dashboard displaying a few different metrics collected from a node

#### 4.2.2 Metrics

The Metrics service collects metrics from the nodes it connects to. By default it collects around 90 built-in metrics that are related to the Erlang VM. A few examples are sizes of different memory areas like system memory, binary memory, atom memory; the sum of the length of the run queues of all processes; number of processes and different kinds of ports and sockets. If a managed node uses the Erlang libraries *Folsom* [8] or *Exometer* [11] for storing metrics, WombatOAM will automatically collect all Folsom/Exometer metrics as well.

The collected metric values are stored in WombatOAM. They are consolidated according to a configurable schedule. By default, the metrics are collected each minute, but after an hour, we calculate an average for each 5-minute interval and store only the average values. After a day, we store only values averaged over 15-minute intervals. After a week, we discard the values. With this mechanism (which is also used by tools like Graphite [10]), we can ensure that we have detailed data about the latest periods, but still keep the size of our database reasonable.

There are three ways to obtain the metric values from WombatOAM:

1. The WombatOAM Web Dashboard can visualize the metrics. In this section, we will focus on this method.
2. WombatOAM can be configured to push its data into external tools; currently we support Graphite [10]. A screenshot will demonstrate this method at the end of the section.
3. WombatOAM's REST API provides queries for retrieving the metric values.

Figure 17 shows the visualization of the *Total memory*, *Code memory* and *System memory* metrics on the `riak1@127.0.0.1` node in the same graph, thus making it easy to compare them.

If a metric is present in all nodes of a node family, it can also be selected under the node family. In that case, WombatOAM will display a stacked graph as shown in figure 18.

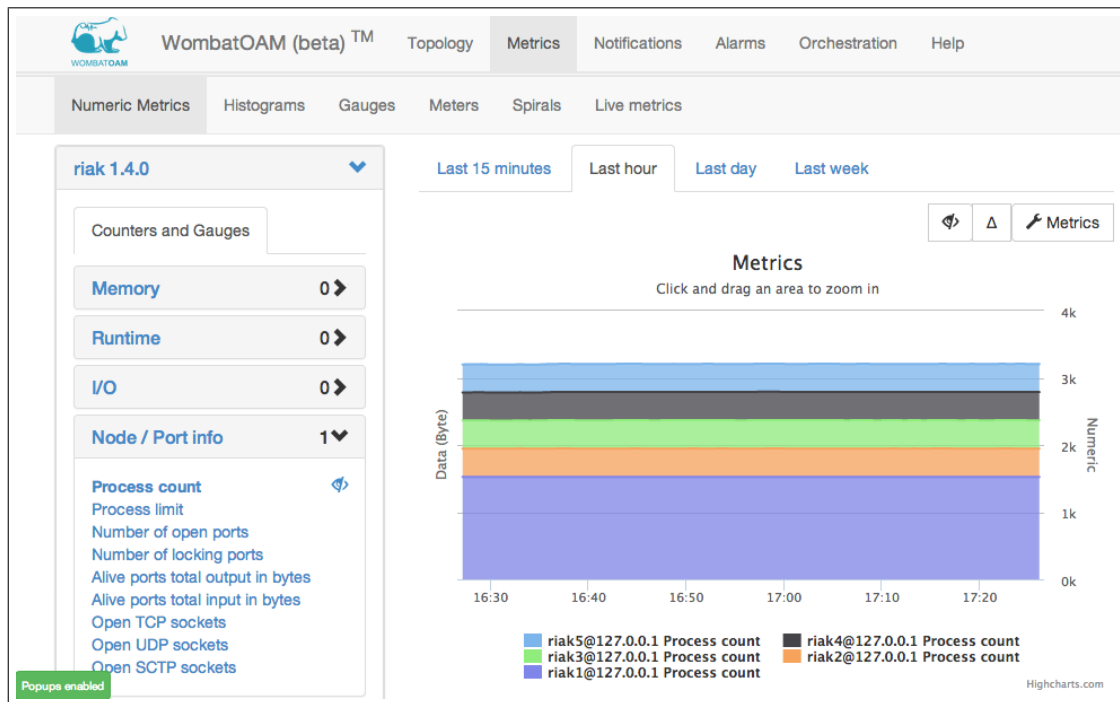


Figure 18: The process\_memory metric on all nodes of the Riak node family, stacked onto each other. The graph shows that riak1 has around three times as many processes as the other four Riak nodes.

The Folsom and Exometer libraries (metric libraries that run on the managed nodes) can not only store metrics, but they support different metric types and perform calculations on them, providing derived metrics. Besides simple numeric metrics like gauges and counters, they support derived metrics like histograms, meters and spirals. WombatOAM collects the derived metrics as well, and since the best way to visualize those is different from the best way to visualize numeric metrics, it provides different subtabs for them. For example figure 19 shows a histogram metric.

The last subtab is called *Live metrics*. By selecting this tab, the user can display numeric metrics as in the *Numeric Metrics* tab, but here they will be updated every second. The aim with this tab is to make it easier to examine specific metrics of specific nodes more closely. We don't want to collect every metric on every node in every second, because that would put too much load on the nodes, the network, and WombatOAM itself.

As already mentioned, WombatOAM can be configured to push its data into Graphite [10], which is an often used tool for storing and visualizing metrics. Figure 20 shows the Graphite web dashboard displaying metrics collected and pushed into it by WombatOAM.

### 4.2.3 Notifications

If a managed node uses the *SASL (System Application Support Libraries)* [5] and/or the *lager* [37] application to store logs, WombatOAM will automatically collect those new log entries whose severity is at least *error*. (This level can be configured on the web dashboard on a per node basis.) WombatOAM can generate its own notifications too: when an alarm is raised or cleared for example, it generates a notification. Figure 21 shows a case where first two alarms were raised on the node, then a *Test error* was manually generated on the node by calling `lager:log(error, self(), "Test error")`. This error was instantly picked up by WombatOAM, and the server instantly reported it to the web dashboard via websockets. Afterwards the dashboard shows a popup window for a few seconds, and adds it to the notification table.

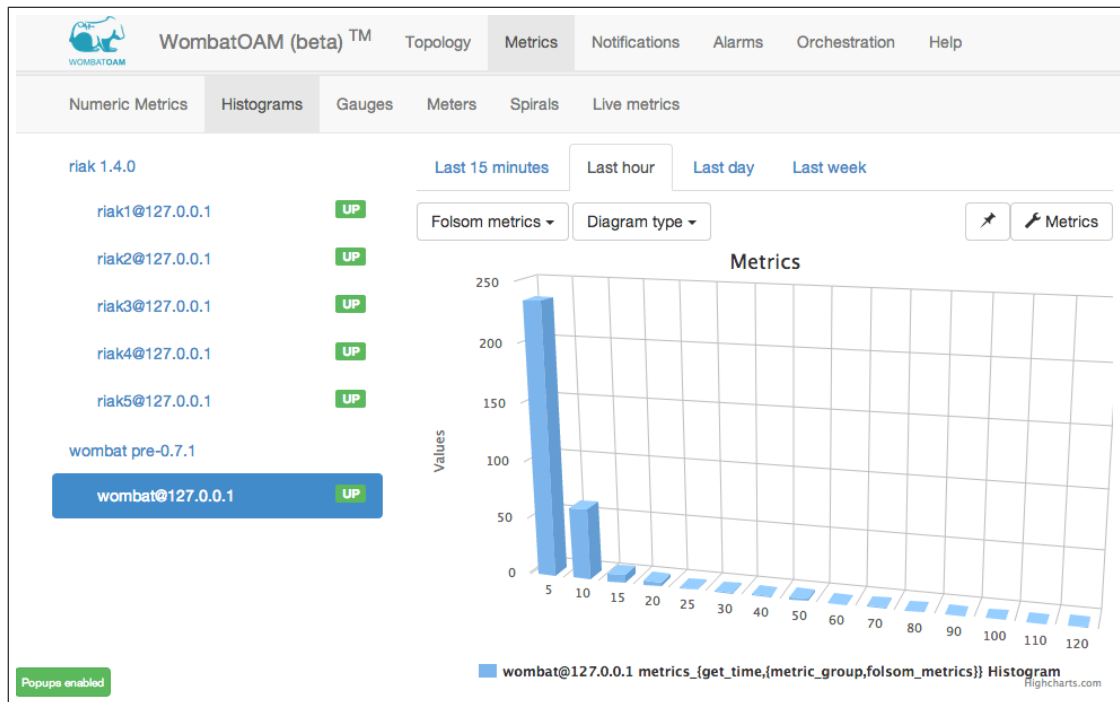


Figure 19: A histogram metric that shows the distribution of the metric called `{get_time,{metric_group,folsom_metrics}}`. This metric is generated on the WombatOAM node, and it is about how much time it took to collect the Folsom metrics from the managed nodes. The histogram shows that in most cases (around 240 times), the collection time was under 5 milliseconds; in many cases (around 55 times) it was between 5 and 10 milliseconds; in a few cases it was between 10 and 15 milliseconds; and so on. The last value is 120, which means that the highest collection time was 120 milliseconds.

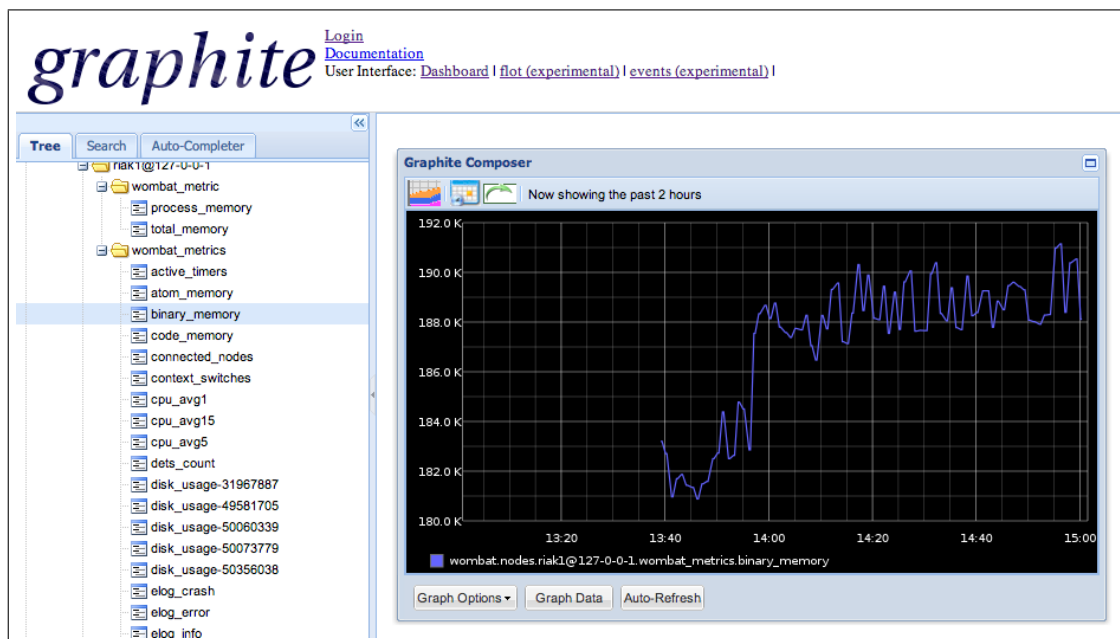


Figure 20: The Graphite web dashboard showing the binary memory metric, which was collected by WombatOAM and pushed into Graphite

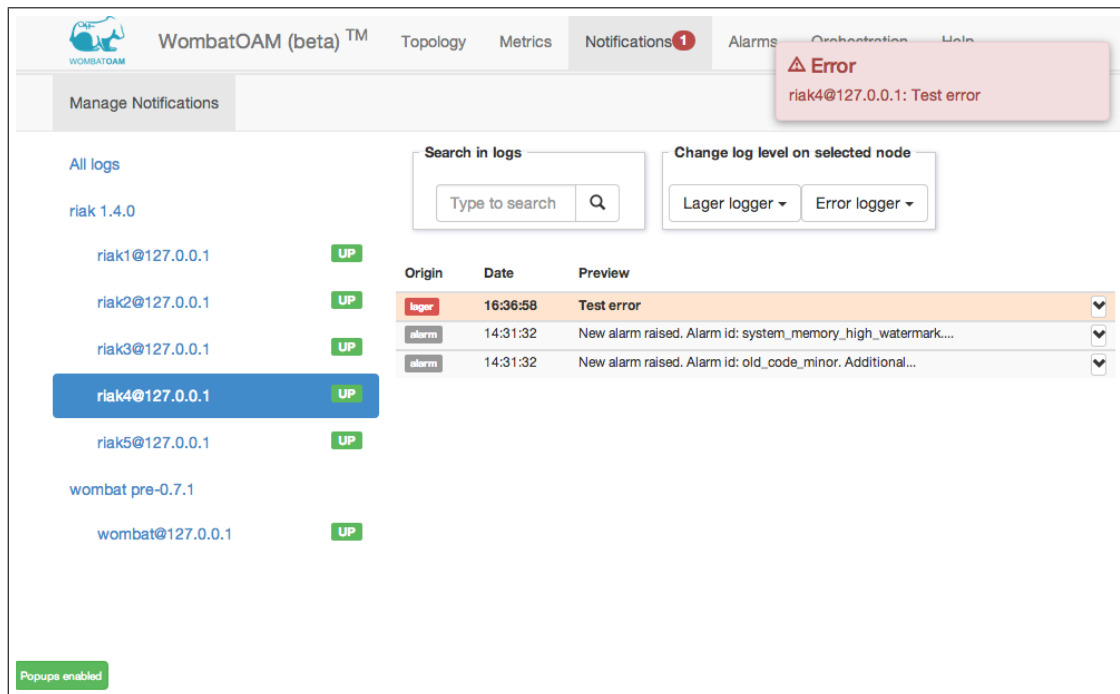


Figure 21: The *Notifications* tab with two test notifications and two notifications about alarms. When a new error level notification arrives, a popup is also visible for a few seconds; those are the red popups in the top right corner.

#### 4.2.4 Alarms

The final service that we examine is the Alarm service. It is similar to Notifications, but instead of notifications, it collects alarms. Notifications are one-time entries, while alarms are more complex entities.

Alarms have a life cycle and a state. After an alarm is raised, its initial state is *new*, which means that it needs attention. If a user sees an alarm, he may acknowledge it (e.g. on the web dashboard), thereby putting it into the *acknowledged* state, which indicates that someone is working on the issue or knows about it. The system that raised the alarm might also clear the alarm, in which case it will not be shown any more.

Alarms also have identities. If the same alarm is reported several times, the alarming system (WombatOAM in this case) will recognize this and store it only once.

As with notifications, WombatOAM collects alarms from SASL; it collects alarms from an open source Erlang alarming library; and it generates its own alarms. The library in this case is *Elarm* [28], which we created and open sourced. The reason for creating *Elarm* was that just like Erlang's built-in SASL solution was not optimal for more complex problems and hence Lager was created as a more advanced logging library, we judged that SASL's alarming solution is also too simple for some applications. Besides having more features, the most important distinction is that *Elarm* stores many pieces of additional information about the alarms like state, severity, probable cause, etc, which is very useful if one wants to expose the alarms of the node via SNMP, REST or a web interface.

WombatOAM has around 20 built-in alarms. A few examples are: the message queue of a process is too long; the atom table is almost full; the number of processes is approaching the limit.

Figure 22 shows the alarms on the *riak4* node. There are two alarms on this node. The *old\_code\_minor* alarm was generated by WombatOAM, because it detected that there are a few modules on the node that have two versions in the system, which may cause problems in the future. The

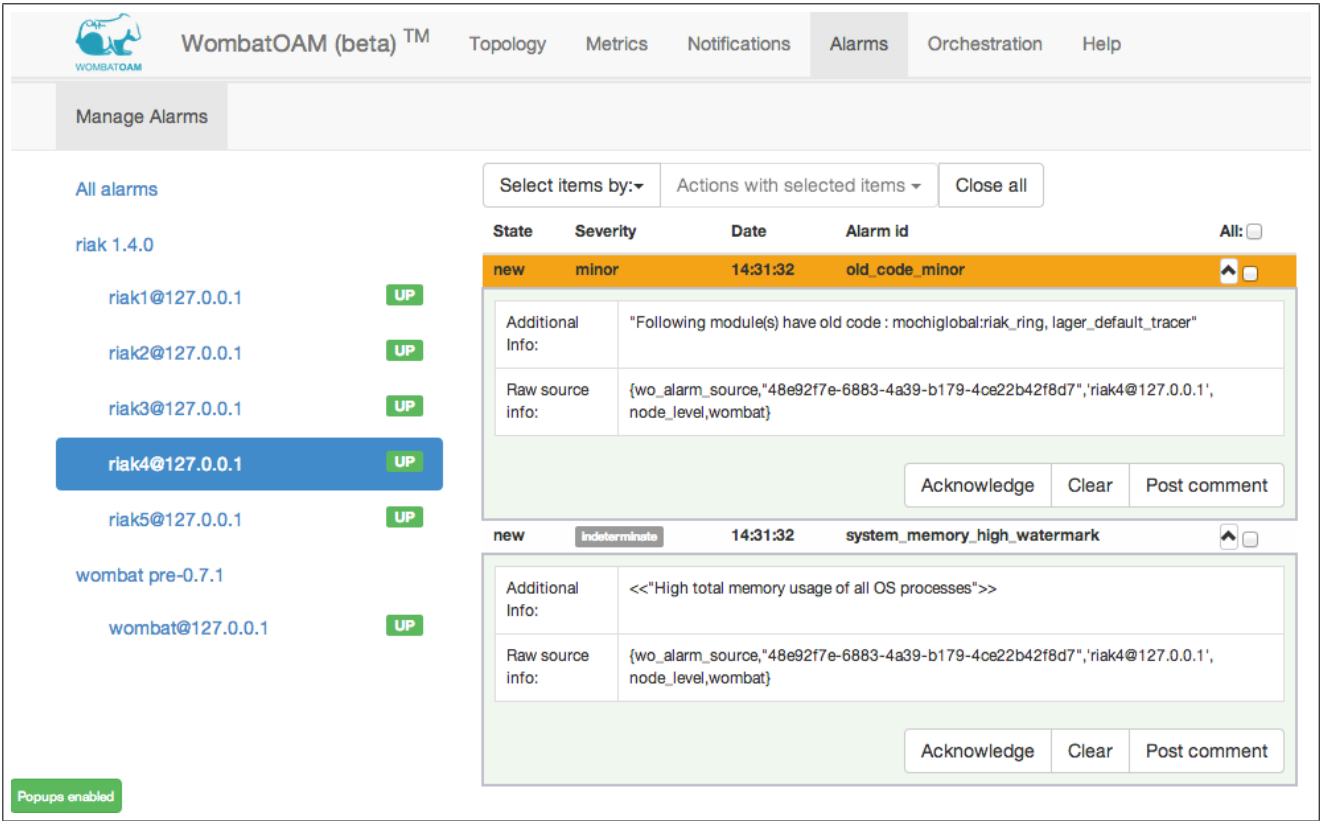


Figure 22: The active alarms on a managed node: an `old_code_minor` alarm and a `system_memory_high_watermark` alarm.

other alarm is `system_memory_high_watermark`, which was reported on the managed node by OTP itself towards SASL, and collected by WombatOAM. This alarm means that the amount of free memory in the system is decreasing.

## 5 Conclusions and Future Work

In this deliverable we have presented how we have tackled the heterogeneity challenge. To this end, we have introduced a revised architecture for WombatOAM which has allowed us to move from an Amazon-dependant *command-line interface* to a fully-fledged operations and maintenance application capable of deploying and monitoring distributed Erlang applications on different Cloud infrastructures. From a prototype development stand-point, we have also made good progress, introducing several important changes such as the development of an Erlang wrapper for interacting with Libcloud, the implementation of a RESTful interface and a web dashboard enabling end-users to easily interact with WombatOAM, and developing monitoring features, which can collect, store and present metrics, notifications and alarms from the managed nodes.

All the good progress made in this deliverable allows us to focus now on the upcoming challenges including measuring the performance of our system as part of *D4.5*, and possibly optimize it to make sure that it scales well even when deploying a large number of nodes.



## A Historical notes

This appendix section explains some details about the history of WombatOAM. Having this section is justified by the fact that this deliverable was originally submitted in January 2013, and then rewritten in June 2014. Much has changed between these dates: among other things, the name of the application was changed from CCL to WombatOAM.

Section A.1 was taken as is from the 2013 version of this deliverable. It explains how CCL evolved from its first prototype submitted in July 2012 as deliverable *D4.2*[32] until January 2013. Section A.2 is a new section, showing how the CCL of January 2013 evolved into the WombatOAM of June 2014.

### A.1 From a Command Line Tool to a Fully-blended Virtualization Infrastructure

In the past few months<sup>1</sup>, CCL has evolved from a simple command-line tool to a fully-blended virtualization infrastructure, consisting of several inter-connected Erlang nodes. In this section we describe this evolution process and we present the components making up the CCL architecture, with focus on the concept of *heterogeneity*, driving force of this deliverable.

#### A.1.1 Evolving the CCL Architecture

In the RELEASE deliverable *D4.2*[32] we described CCL as a command-line tool used by the end user to build and deploy Erlang applications into the Amazon EC2 cloud environment. The communication between the CCL node and the other Erlang nodes composing a cluster was mainly happening over SSH channels, not taking direct advantage of the Erlang distribution protocol. In the same deliverable we introduced for the first time the concept of a *esl-core* (hidden) node, an *Operations and Maintenance* (or, more simply, *OM*) node responsible of monitoring and operating on the other Erlang nodes in the cluster.

We have recently decided to split the *esl-core* node into two separate entities: *ccl-oam* and *ccl-cluster*, as depicted in figure 23. *Ccl-oam* runs within the same environment as the other Erlang nodes and can therefore take direct advantage of the Erlang Distribution protocol while interacting with the rest of the cluster. On the contrary, we place the *ccl-cluster* nodes outside that environment, laying the foundation for the concept of a *super-cluster*, which will be exploited as part of the *D4.4* (i.e. *Capability-driven Cluster On Demand*) deliverable.

Figure 23 depicts an overview of the current CCL architecture. Let's analyze the CCL components in detail and the way they communicate each other. Please note that a *dashed* arrow in the figure represents that the connected components communicate via *Remote Procedure Call (RPC)*, normally implemented over the *AMQP*[31]; while a *straight* line means the communication happens over the *Erlang Distribution Protocol*.

#### A.1.2 The CCL Components

**CCL-API** The *CCL-API* node represents the entry point to the CCL ecosystem. This component can be seen as an orchestration layer, used to interact with the rest of the system. Interaction with users and other CCL components is performed via the exposed RESTful and RPC interfaces, respectively. *CCL-API* is also responsible for maintaining a global database up-to-date. This database contains information such as user accounts, cloud service providers credentials linked to each user account, information about the available clusters in the system, etc. The *CCL-API* node is in charge of forwarding any incoming request to the CCL component responsible for handling it (e.g. it forwards a *deploy a new Erlang node* request to the *CCL-Cluster* node responsible for managing the cluster where the

<sup>1</sup>This section (A.1) was written in January 2013.



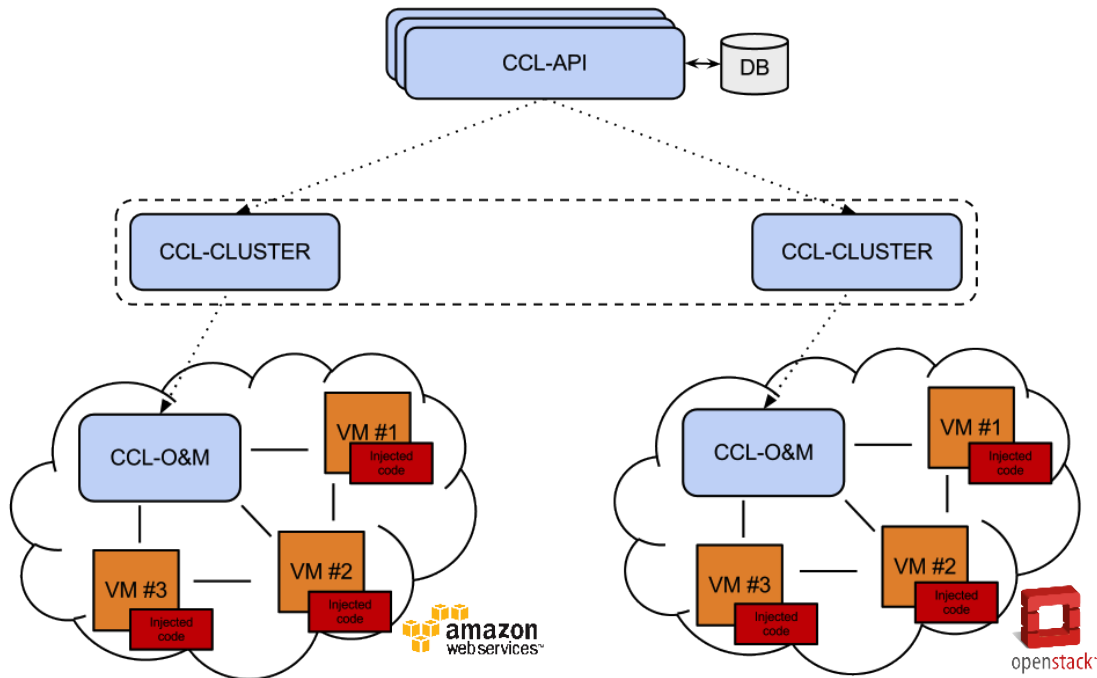


Figure 23: Overview of CCL Architecture

new Erlang node is going to be deployed on). A normal deployment of CCL could consist of several *CCL-API* nodes and by a HTTP proxy balancing the load across them.

**CCL-CLUSTER** This component is responsible for managing a *logical* cluster. In CCL terms, a cluster is a set of homogeneous Erlang nodes deployed within the same Cloud infrastructure, all of them running the same Erlang release (e.g. Riak). A *CCL-Cluster* node handles all the requests related to the creation and deletion of Erlang nodes on the cluster it is responsible for. When it comes to deploying an Erlang application, this component can also be seen as a cache that will serve the source code of the application to the Erlang nodes that are going to be running the application, avoiding then having to access the project repository where the application is hosted as many times as nodes are deployed.

**CCL-O&M** The *CCL-O&M* component consists of a *hidden* Erlang node[2]. In Distributed Erlang, hidden nodes (started with the `-hidden` command line flag) are commonly used to inspect the status of a system in an unobtrusive way. In fact, connections between hidden nodes and other nodes are not transitive. Using the Distributed Erlang protocol, the *CCL-O&M* node is responsible of injecting a lightweight Erlang process (an *agent*) into each of the other Erlang nodes belonging to the cluster. Injected processes represent a *backdoor*, used by *CCL-O&M* to control and monitor a cluster.

### A.1.3 Cross-component Communication

In the previous development phase of CCL we did not require any complex messaging since all we had was a command-line interface that could be used for deploying a given distributed Erlang application into a set of machines running on the Amazon EC2 cloud environment. Nevertheless, as already described in Section A.1.1, CCL has evolved into a multi-component application which does now require a more complex messaging protocol. To this end, we have decided upon using RabbitMQ [40], an open-source message broker that implements the AMQP protocol [31], for all the cross-component communication. One thing that makes RabbitMQ interesting for us is its simplicity for clustering

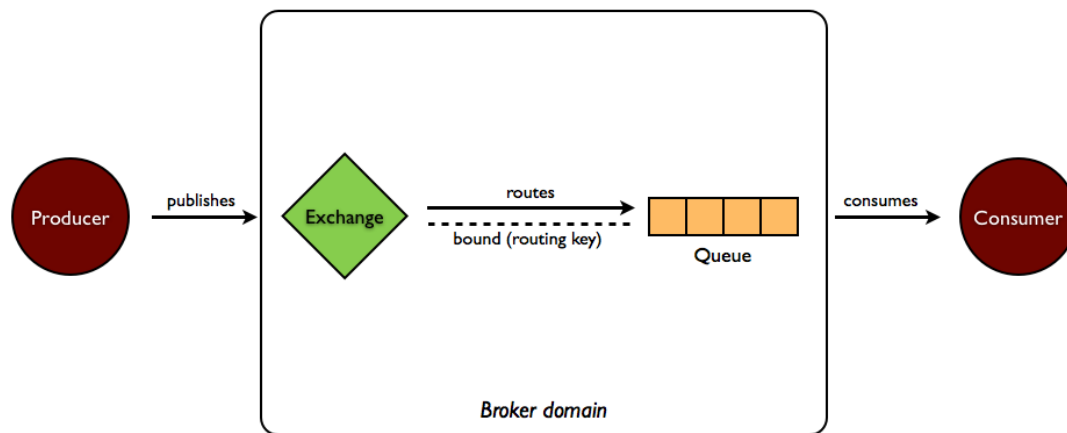


Figure 24: AMQP messaging model

several RabbitMQ servers on the same local network as a single logical broker, making the whole messaging infrastructure more scalable and resilient to failures.

In order to better understand how CCL benefits from using RabbitMQ, it is important to grasp how RabbitMQ and AMQP work. Figure 24 illustrates in brief the AMQP model where:

1. messages are published by producers to exchanges;
2. exchanges then distribute these messages to queues according to some routing rules;
3. either messages are delivered to consumers subscribed to queues, or consumers directly pull messages from queues

This does not end here, though. The AMQP protocol features different built-in exchange types influencing how messages are routed to queues. Four exchange types are defined in AMQP:

**Direct exchanges** They deliver messages to queues based on the specified routing key, which makes it ideal for unicast communication. A queue is bound to an exchange with a routing key, which is then matched when a message arrives to the exchange. If both routing keys are the same, the message is delivered to the queue.

**Fanout exchanges** They deliver messages to all queues that are bound to it regardless of the routing key.

**Topic exchanges** They treat the routing key as a list of zero or more words, delimited by '.', and supports special wild characters, such as '\*' and '#', matching a single word or zero or more words, respectively.

**Header exchanges** They route messages to queues not accordingly to a routing key, but accordingly to a routing algorithm specified using a special argument, which can either be set to *all* or *any*. Queues are bound to this exchange with a table of arguments and will only receive messages if all or any of their binding arguments - depending on the policy set on the exchange - matches the routing argument of the message.

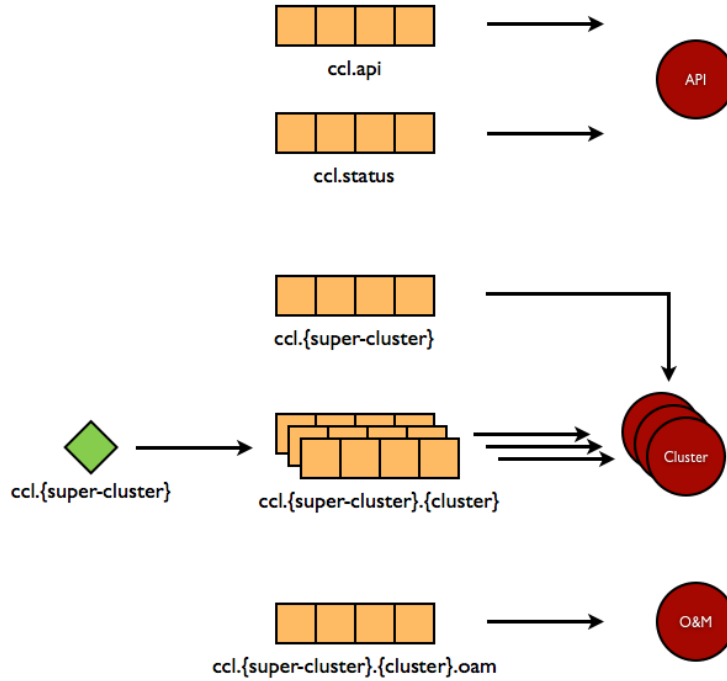


Figure 25: AMQP messaging model in CCL

Now that we know how the AMQP protocol works we can move on and see how we have integrated RabbitMQ into CCL. the different exchanges and queues that CCL makes use of.

- The *ccl.api* queue is used to remotely invoke functions in the *CCL-API* node. Analogously, *ccl.{super-cluster}.{cluster}* and *ccl.{super-cluster}.{cluster}.oam* queues are used to remotely invoke functions in the corresponding cluster and OAM node, respectively.
- All *ccl.{super-cluster}.{cluster}* queues are bound to the *ccl.{super-cluster}* fanout exchange, such that all messages sent to the latter will be forwarded to all the slaves deployed within the encompassing cluster.
- The *ccl.{super-cluster}* queue is shared among all cluster nodes within the same cluster. It is a task queue used to balance the load among all available clusters.
- The *ccl.{super-cluster}.status* is a queue where all the clusters periodically send their status updates. This information is used by *CCL-API* to check the availability of the different clusters.

## A.2 Evolving the CCL Infrastructure to WombatOAM

As the previous section describes, by the beginning of 2013 CCL was able to deploy Erlang releases to different Cloud providers. It used a program called *DeltaCloud* [13] to talk to the providers. Clusters could be created that knew the details needed for deploying nodes. Finally, nodes were deployed based on the information in their cluster in CCL. After deploying the nodes, CCL injected an agent which collected the metrics previously specified in the *oam.config* file in the release. Its internal architecture consisted of several layers: a web dashboard, a REST API (a.k.a. *CCL-API*), cluster managers (*CCL-CLUSTER*), node managers (*CCL-O&M*) and agents.

In the last 1.5 years, we added new functionality to the application and refactored existing functionalities. But we never changed it completely or started from scratch: this was an organic development

process based on our own experiences and others' feedback regarding what worked well, what did not, what was missing. We kept the hierarchical architecture (although added a new, horizontal dimension as explained in section 2.2). We kept the techniques developed for connecting to the managed nodes via our hidden nodes and for injecting our agent modules into them. We also kept the philosophy of the Orchestration features with providers, releases, clusters (although they were renamed to node families, see below), provisioning virtual machines, performing node deployments, and using a third-party tool for communicating with the different cloud providers (although we used to use DeltaCloud, and now we use Libcloud, as explained in section 3.1), and having an Erlang wrapper around that tool.

We have implemented many improvements and other changes since January 2013. A few very noticeable changes (modifying the terminology by renaming clusters to node families, making changes in the vertical architecture and creating the horizontal architecture) happened around the same time, so we grabbed the opportunity and also renamed CCL to WombatOAM. By this change, we also signified that while CCL was more a prototype, WombatOAM is more a product.

Out of the improvements and changes, a few important ones are highlighted in the following list:

1. *Clusters* were renamed to *node families*. In deliverable D6.3 (*Distributed Erlang Component Ontology*) [33], the consortium members made a joint effort of creating an ontology for distributed Erlang systems. During this work, it was realized that the term *cluster* is better to be used as a set of Erlang nodes that work together and achieve a common function, but possibly run different releases and have different tasks. For example a cluster might contain front-end nodes and back-end nodes. CCL used this term for a set of nodes running the same release on the same set of providers, with the same basic properties (firewall rules, etc.) We decided to resolve this conflict by finding a new name for CCL clusters, and we chose the term *node family*, because it fits well with the meaning we want to use it for, and it is a term not yet used in the Erlang community.
2. The architecture shown in figure 23 and described in section A.1.1 was replaced by the architecture shown in figure 2 (page 8) and described in section 2 (page 7).

Instead of the CCL-Cluster component and CCL-O&M component, we have *middle manager* components. A CCL-Cluster was responsible for handling a set of homogeneous Erlang nodes running the same Erlang release. A CCL-O&M component was responsible for those nodes in a cluster that ran on the same cloud provider. A middle manager also handles a set of Erlang nodes, but it does not have the aforementioned restrictions, which is a scalability advantage: If one has a huge cluster/node family within one provider, one might want to distribute them into several WombatOAM nodes (i.e. several middle managers). On the other hand, if someone has several releases and several providers, but runs only a few dozen nodes, he probably does not want a node for each combination, only one WombatOAM node.

This change allowed us to create a one-node version of WombatOAM, where all of its components run in one WombatOAM node (except the agents of course which run on the managed nodes). Together with the next item, this made WombatOAM significantly easier to install and manage.

3. The remote procedure calls (RPCs) used for communication between CCL/WombatOAM components used to be performed using the AMQP protocol and RabbitMQ. WombatOAM components now use Erlang's built-in mechanisms for communication: message passing and the *rpc* module [3] shipped with Erlang/OTP. The latter proved to be simpler both for developers and users.

From the development perspective, we are not stating that RabbitMQ is not useful or cannot be used well from an Erlang application, but the way it was used in CCL was too simplistic. For example if a process performed RPC on another process, it placed the call in RabbitMQ, and the other process needed to read that call, interpret it, and place the answer in RabbitMQ so that the original caller could retrieve it. But if anything bad happened (e.g. the receiver process

crashed), the answer was not placed in RabbitMQ, so the caller process waited indefinitely. Using Erlang/OTP's built-in mechanism (a call between `gen_servers`), the same scenario would result in an error being generated in the caller process, which can then either handle the error or propagate it.

From the user perspective, RabbitMQ was just another moving part that could go wrong. For example one of our users had a trouble with WombatOAM: processes started to hang. Because of the problematic error handling mechanism described above, we suspected a programming error in WombatOAM, but after some investigation, it turned out that RabbitMQ throws away new messages by default if there is not at least 1 GB of free disk space on the partition it uses<sup>2</sup>, so the RPCs of WombatOAM processes were not delivered to other WombatOAM processes. Again, the same scenario would not have happened using Erlang/OTP's mechanisms instead of RabbitMQ.

It should be noted though that if there were a good reason for using AMQP/RabbitMQ for messages and/or RPCs, it could be investigated again, and it may turn out to be beneficial in some cases if used with the right approach. One example where moving away from Erlang distribution might help is that it would be more convenient if WombatOAM components used only one port to communicate with each other (this would make firewall configuration simpler). Another example is that in Erlang, nodes can have short or long names, but a connected set of nodes must have the same (i.e. either all nodes use short names or all use long names). This means that currently if WombatOAM uses long names, it can manage only nodes with long names; if it uses short names, it can manage only nodes with short names. If the WombatOAM nodes didn't connect to each other via Erlang distribution, this limitation would be lifted, so we could have some middle managers using long names (and thus being able to manage nodes with long names), and some with short names (these would manage nodes with short names). Finally, plain Erlang distribution as a communication channel between nodes is not secure; for that, we could use either Erlang distribution over SSL or some other protocol over SSL.

4. DeltaCloud as the tool for communicating with the providers was replaced with Libcloud; see section 3.1 for the details.
5. A new Web Dashboard was built; see section 3.3 for its description.
6. The monitoring part of WombatOAM went from gathering only the metrics specified by the user and showing them in real-time to gathering, storing and displaying 90 built-in metrics plus all metrics stored in the Folsom and Exometer libraries; 20 built-in alarms plus all alarms raised in the SASL `alarm_handler` or Elarm library; and all log entries (above the configured level) that are sent to the SASL `error_logger` or the Lager library.

---

<sup>2</sup>This limit has been lowered to 50MB since then. It can be verified though that the configuration page in 2013 though stated that "By default free disk space must exceed 1GB." [35]

## References

- [1] Ericsson AB. *Erlang - A disk based term logging facility*.  
[http://www.erlang.org/doc/man/disk\\_log.html](http://www.erlang.org/doc/man/disk_log.html).
- [2] Ericsson AB. *Erlang - Hidden Nodes*.  
[http://www.erlang.org/doc/reference\\_manual/distributed.html#id83184](http://www.erlang.org/doc/reference_manual/distributed.html#id83184).
- [3] Ericsson AB. *Erlang - Remote Procedure Call Services*.  
<http://www.erlang.org/doc/man/rpc.html>.
- [4] Ericsson AB. *Erlang/OTP Releases*.  
[http://www.erlang.org/doc/design\\_principles/release\\_structure.html](http://www.erlang.org/doc/design_principles/release_structure.html).
- [5] Ericsson AB. *System Application Support Libraries (SASL) Reference Manual*.  
<http://www.erlang.org/doc/apps/sasl/>.
- [6] Marios Andreou. *Announcement email titled "Red Hat and Apache Deltacloud"*.  
[http://mail-archives.apache.org/mod\\_mbox/deltacloud-dev/201305.mbox/%3C518D0F79.4000901@redhat.com%3E](http://mail-archives.apache.org/mod_mbox/deltacloud-dev/201305.mbox/%3C518D0F79.4000901@redhat.com%3E).
- [7] Highsoft Solutions AS. *Highcharts JS*.  
<http://www.highcharts.com/>.
- [8] Boundary. *Folsom*.  
<https://github.com/boundary/folsom>.
- [9] Hewlett-Packard Company. *HP Cloud flavors*.  
<http://www.hpcloud.com/pricing#Compute>.
- [10] Chris Davis. *Graphite*.  
<https://graphite.readthedocs.org/>.
- [11] Feuerlabs. *Exometer*.  
<https://github.com/Feuerlabs/exometer>.
- [12] Open Grid Forum. *Open Cloud Computing Interface (OCCI)*.  
<http://occi-wg.org/>.
- [13] Apache Software Foundation. *Deltacloud*.  
<http://deltacloud.apache.org/>.
- [14] Apache Software Foundation. *Deltacloud drivers*.  
<https://deltacloud.apache.org/drivers.html>.
- [15] Apache Software Foundation. *Libcloud - Supported providers*.  
[https://libcloud.readthedocs.org/en/latest/supported\\_providers.html](https://libcloud.readthedocs.org/en/latest/supported_providers.html).
- [16] Apache Software Foundation. *Libcloud documentation for sizes*.  
<https://libcloud.readthedocs.org/en/latest/compute/api.html#libcloud.compute.base.NodeSize>.
- [17] Apache Software Foundation. *Libcloud*.  
<https://libcloud.apache.org/>.

- [18] OpenStack Foundation. *OpenStack - Manage flavors*.  
[http://docs.openstack.org/user-guide-admin/content/dashboard\\_manage\\_flavors.html](http://docs.openstack.org/user-guide-admin/content/dashboard_manage_flavors.html).
- [19] Gleber. *Cloud Computing APIs For Erlang*.  
<https://github.com/gleber/erlcloud>.
- [20] Csaba Hoch. *Wombat Orchestration Screencast*.  
<https://vimeo.com/85243629>.
- [21] IETF. *The WebSocket protocol*.  
<http://www.websocket.org/>.
- [22] Amazon Inc. *Amazon EC2 instance types*.  
<http://aws.amazon.com/ec2/instance-types/>.
- [23] Distributed Management Task Forum Inc. *CIMI*.  
<http://dmtf.org/standards/cloud>.
- [24] Google Inc. *AngularJS*.  
<http://angularjs.org>.
- [25] Rackspace Inc. *Rackspace flavors*.  
[http://docs.rackspace.com/servers/api/v2/cs-releasenotes/content/supported\\_flavors.html](http://docs.rackspace.com/servers/api/v2/cs-releasenotes/content/supported_flavors.html).
- [26] Twitter Inc. *Twitter Bootstrap*.  
<http://twitter.github.com/bootstrap/>.
- [27] The jQuery Foundation. *jQuery*.  
<http://jquery.com/>.
- [28] Erlang Solutions Ltd. *Elarm*.  
<https://github.com/esl/elarm>.
- [29] Erlang Solutions Ltd. *elibcloud*.  
<https://github.com/esl/elibcloud>.
- [30] Nine Nines. *Cowboy*.  
<https://github.com/extend/cowboy>.
- [31] OASIS. *AMQP*.  
<http://www.amqp.org/>.
- [32] RELEASE Project. *D4.2 (WP4): Homogeneous Cluster Infrastructure*.  
<http://www.release-project.eu/documents/D4.2.pdf>.
- [33] RELEASE Project. *D6.3 (WP6): Distributed Erlang Component Ontology*.  
<http://www.release-project.eu/documents/D6.3.pdf>.
- [34] John Shafer. *GitHub issue describing Rackspace deprecating its OpenStack 1.0 API*.  
[https://github.com/rightscale/rightscale\\_cookbooks/issues/156](https://github.com/rightscale/rightscale_cookbooks/issues/156).
- [35] Pivotal Software. *RabbitMQ Configuration (2013 Archive)*.  
<https://web.archive.org/web/20130515082212/http://www.rabbitmq.com/configure.html>.

- [36] Daniel Stenberg. *cURL*.  
<http://curl.haxx.se/>.
- [37] Basho Technologies. *Lager*.  
<https://github.com/basho/lager>.
- [38] Basho Technologies. *Riak*.  
<http://basho.com/riak/>.
- [39] Joseph Wayne Norton Ulf Wiger. *gproc*.  
<https://github.com/uwiger/gproc>.
- [40] VMware. *RabbitMQ*.  
<http://www.rabbitmq.com/>.