ICT-287510
RELEASE
A High-Level Paradigm for Reliable Large-Scale Server Software
A Specific Targeted Research Project (STReP)

D4.5 (WP4): Scalable Infrastructure Performance Evaluation report

Due date of deliverable: 31st January 2015
Actual submission date: 16th February 2015

Start date of project: 1st October 2011 Duration: 36 months

Lead contractor: Erlang Solutions Ltd. Revision: 1.0

Purpose: To evaluate the performance of the scalable infrastructure (called WombatOAM) developed as part of the RELEASE project.

Results: In the course of this deliverable we:

- described how we improved the infrastructure to be more scalable, and
- presented the performance evaluation of the scalable infrastructure.

Conclusion: In this document we have presented an adequate evaluation of the scalable infrastructure focusing on the performance achieved, on the intrusiveness of WombatOAM and on the ability to scale out. We have also given an insight into the final efforts aiding in improving the scalability and robustness capabilities of the infrastructure.

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)	

Scalable Infrastructure Performance Evaluation report

Csaba Hoch <csaba.hoch@erlang-solutions.com>
Viktória Fördős <viktoria.fordos@erlang-solutions.com>
Eva Bihari <eva.bihari@erlang-solutions.com>
Torben Hoffmann <torben.hoffmann@erlang-solutions.com>

Contents

Executive Summary

The goal of the D4.5 deliverable is to evaluate the performance of the developed scalable infrastructure, which we named WombatOAM. We performed initial performance tests (soak tests), based on which we implemented a dozen of performance, scalability and error-handling improvements. We also added a few new features, ensuring that WombatOAM can deploy applications that use SD-Erlang. Afterwards we executed performance tests on a large scale (with deploying on up to ten thousand nodes).

This document describes all of the steps above, presents the measured experimental data, and provides a report on it, including a detailed description of the applied evaluation method. The measurement data shows two important facts: it shows that WombatOAM scales well (we deployed up to 10 000 Erlang nodes), and that WombatOAM is non-intrusive because its overhead on the monitored node is less than 1.5%. The deliverable closes by highlighting the conclusions and briefly defines some future work.

1 Introduction

Despite Moore's law, nowadays the computational capacity is being enlarged by increasing the number of cores instead of the clock frequencies of the processors. This new situation forces the IT to change their approach in order to exploit the resources of a many-core processor. Only massive concurrency and distributed systems can solve this new challenge. In parallel, cloud computing [4] has become a mainstream method. However, using only the resources of a single cloud is not always the most economical decision, thus cost optimisation methods have been developed [11]. Therefore, large scale distributed systems need to adapt themselves to heterogeneous super-clusters [10]. These super-clusters may be built from personal computers, as the price/performance ratio of these computers has improved. Even building a super-cluster from off-the-shelf technology has become financially attractive.

At the moment there are a few programming languages that are capable of exploiting all the computational resources that are provided either by a cluster or by many-core processors. Erlang [2] belongs to these programming languages. It can transparently utilise all the given computational capacity of a processor, and can quickly adapt to dynamically growing clusters.

As Erlang provides a built-in support in term of both language features and libraries for distributed systems, making Erlang systems that scale to many machines is an easier task than in case of most other languages and technologies. However, deploying and managing a system which is prepared for massive load to thousands of Erlang nodes remains a tough challenge. Erlang Solutions, as a consulting company, have worked with a number of clients, who have proprietary solutions to this problem.

In the context of the RELEASE project, the main task of WombatOAM is to provide the scalable infrastructure for deploying thousands of Erlang nodes. The challenge has been tackled by designing and engineering WombatOAM. WombatOAM is aimed at providing a broker layer capable of creating, managing and dynamically scaling heterogeneous clusters, even based on capability profile matching.

It is also worth to highlight the fact that WombatOAM is currently being used by paying customers, the largest one with 200 nodes in production. This deliverable allows us to target customers who have thousands of nodes in production; many of whom have already shown interest.

In this deliverable, the final efforts aiding in improving WombatOAM's scalability and robustness capabilities are discussed in Section 2. This is followed by an adequate evaluation of WombatOAM using the Athos cluster available at EDF in Section 3. The evaluation consists of three main parts. Section 3.1 gives the necessary background, Section 3.2 presents the designed tests for the evaluation, whilst Section 3.3 gives some insight to the realisation of tests and highlights the achievements and the lessons learnt. Finally Section 4 concludes the deliverable and briefly defines some future work.

2 Improvements Targeting WombatOAM's Scalability and Robustness Capabilities

The final efforts aiding in improving WombatOAM's scalability and robustness capabilities are presented in this section. As the goal of the deliverable is to demonstrate these properties of WombatOAM by providing the results of performance measurements, we performed activities before the final measurements that ensure that WombatOAM actually has these properties. These tasks include the analysis of WombatOAM, several improvements in terms of scalability and robustness, and the introduction of Meta Wombat.

2.1 Meta Wombat

While we were surveying the capabilities of WombatOAM, we were faced with results not guaranteeing the robustness and scalability of WombatOAM, in cases where thousands of managed nodes need to be handled. We analysed the architecture of the tool and found the underlying problem. To solve the problem we adopted a new approach. Bearing the new approach in mind, we changed the architecture of WombatOAM by extending it with a new actor. The new actor is called *Meta Wombat*, and it is capable of meeting all the requirements of Task 4.5. After the implementation of the required modifications, WombatOAM is not only capable of handling hundreds of nodes, but it is able to manage thousands of nodes as a reliable, scalable and robust distributed system. In the rest of this subsection, we describe the problem and present how the problem was tackled – i.e., we describe the improved architecture of WombatOAM.

2.1.1 Problem Description

The main problem with the old approach became apparent when the limits of WombatOAM were being determined. When the number of managed nodes was larger than 150, WombatOAM was struggling. A huge number of processes were just waiting, and their number was continuously increasing as time elapsed. This resulted in an unresponsive tool, which terminated abnormally after weeks of soak and load testing.

We found that the main problem was that the architecture was too centralised. Of course, some of the bottlenecks could be eliminated, but the ideal state could not be achieved without altering some aspects of our approach. That architecture was introduced in deliverable D4.3 [15], and for the sake of comprehension, we briefly summarise its relevant parts. Figure 1 shows its architecture diagram, which illustrates its layered architecture as well. D4.3 introduced the *middle managers*: a set of components managing a number of Erlang nodes by directly connecting to them. Although the architecture seemed to be scalable, there were some aspects that obstructed scaling WombatOAM to the desired 10 000 nodes.

The main problem of the architecture was caused by the presence of the master component, which contained the global information in central server processes. At the time we designed the master component to ensure the consistency of the global state. One of these aspects of the old system was that all the requests were routed through all the layers of WombatOAM, including the master component, which meant that these processes became bottlenecks. Furthermore, this master component also meant a single point of failure.

Thus, we realised that a new approach needed to be adopted and we needed to rethink the architecture, in a more distributed fashion. This resulted in creating a new component called Meta Wombat, employing the techniques of distributed systems and exploiting the possibilities of a distributed environment. It introduces decentralisation into the system to eliminate performance bottlenecks.

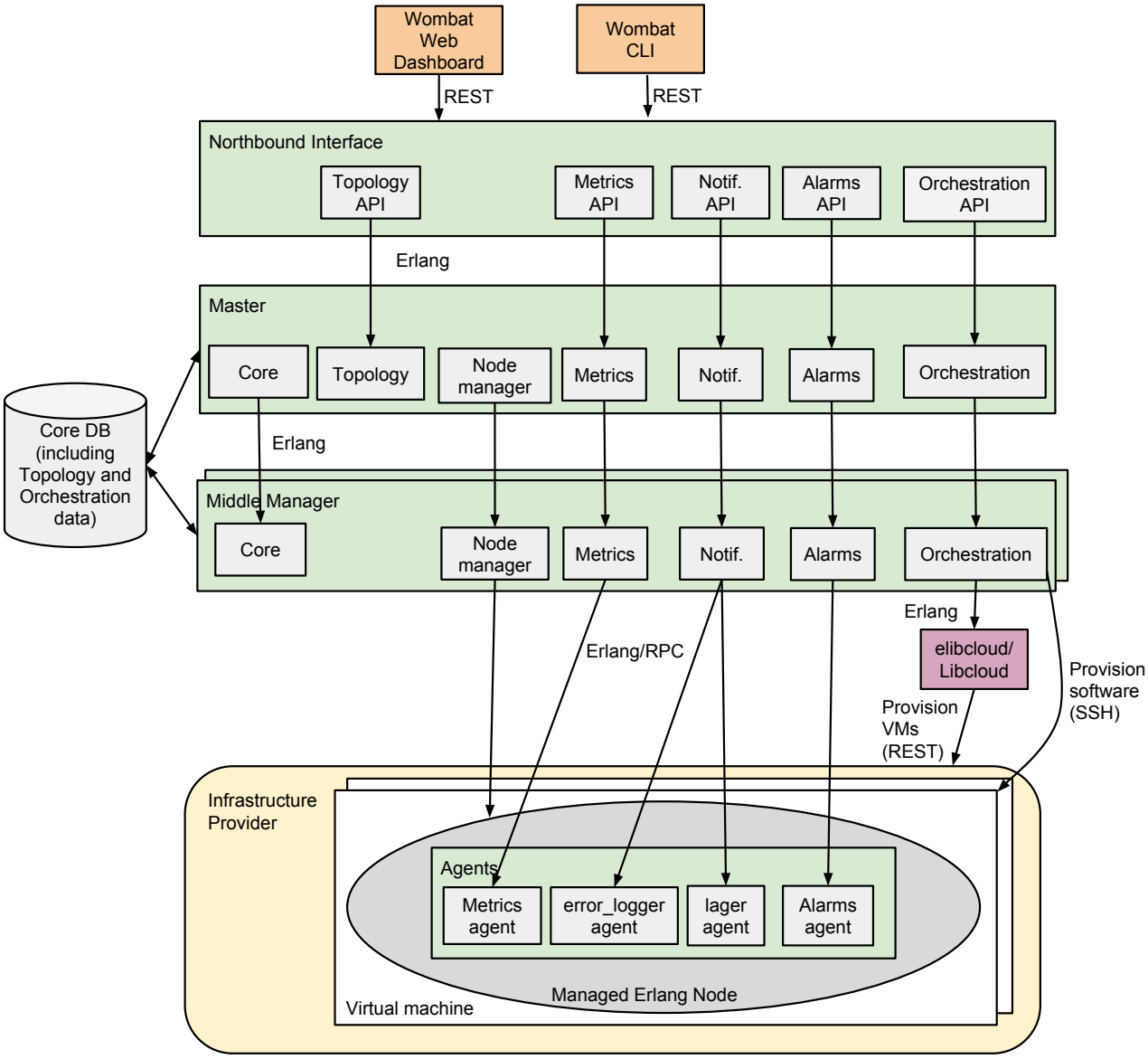

Figure 1: Architecture of WombatOAM [15].

2.1.2 Introduction to Meta Wombat

The underlying idea of Meta Wombat comes from generalising the Middle Manager concept. Meta Wombat is the master of the whole system in a distributed environment. Lessons learned about the centralised approach were taken into account while Meta Wombat was being designed. Namely, it needs to avoid bottlenecks, and no single point of failure can be introduced within. Last, but, not least, the complete re-implementation of WombatOAM must be avoided.

To satisfy all criteria, a *Wombat-tree* was developed, which is illustrated by Figure 2. The root of the Wombat-tree is Meta Wombat, the children of Meta Wombat are instances of the original WombatOAM (with the master layer removed so that now the REST handler communicate directly with the middle manager processes). The leaves of the tree are the managed nodes. It can be said that the Middle Manager environment evolved to be the building block of a Wombat-tree. These building blocks are often denoted as *Worker Wombats*.

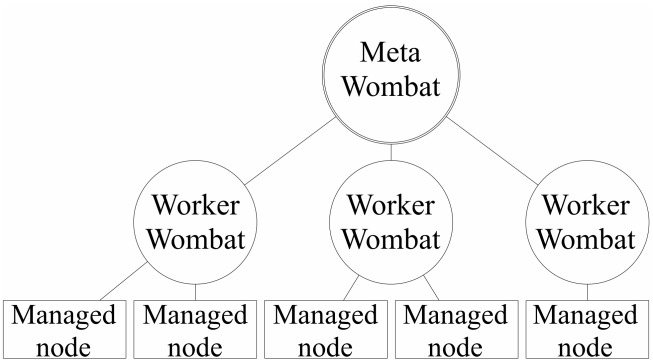

Figure 2: Wombat-tree.

Let us first roughly examine the overall system. Observe that Worker Wombats are not aware of each other; they exist in their separated worlds. In the world of a Worker Wombat, there are the Meta Wombat and the managed nodes belonging to the sub-tree of the Worker Wombat. This world restricts and defines the responsibility of a Worker Wombat at the same time. As not all the managed nodes are visible to a Worker Wombat, it is only responsible for the set of managed nodes it knows about. By limiting the world of Worker Wombats, the volume of their available information is also limited. It is not a problem, it is a benefit. It becomes obvious if we consider that the less information is present, the less likely it is to overload a Worker Wombat with it. Next, consider that Meta Wombat is not directly linked to the managed nodes. Hence Meta Wombat cannot be overloaded, but what will happen to the managed nodes of a dying Worker Wombat? Obviously, the managed nodes should not be left without supervision, thus Worker Wombats inform Meta Wombat about all necessary properties of the managed nodes. As Meta Wombat is aware of these properties, once a Worker Wombat terminated, Meta Wombat reassigns the orphaned managed nodes to another Worker Wombat.

As an overview of the system, the actors of the overall system and their roles are described. Figure 3 illustrates the actors in the context of the Erlang nodes with which the actors communicate.

- *Meta Wombat*. Materialised as an Erlang node (or maybe two nodes for takeovers). It is responsible for managing the Wombat-tree as follows:
  - to start new instances of Worker Wombat,
  - to supervise the Worker Wombats,
  - to stop Worker Wombats,
  - to distribute deployment jobs among Worker Wombats,

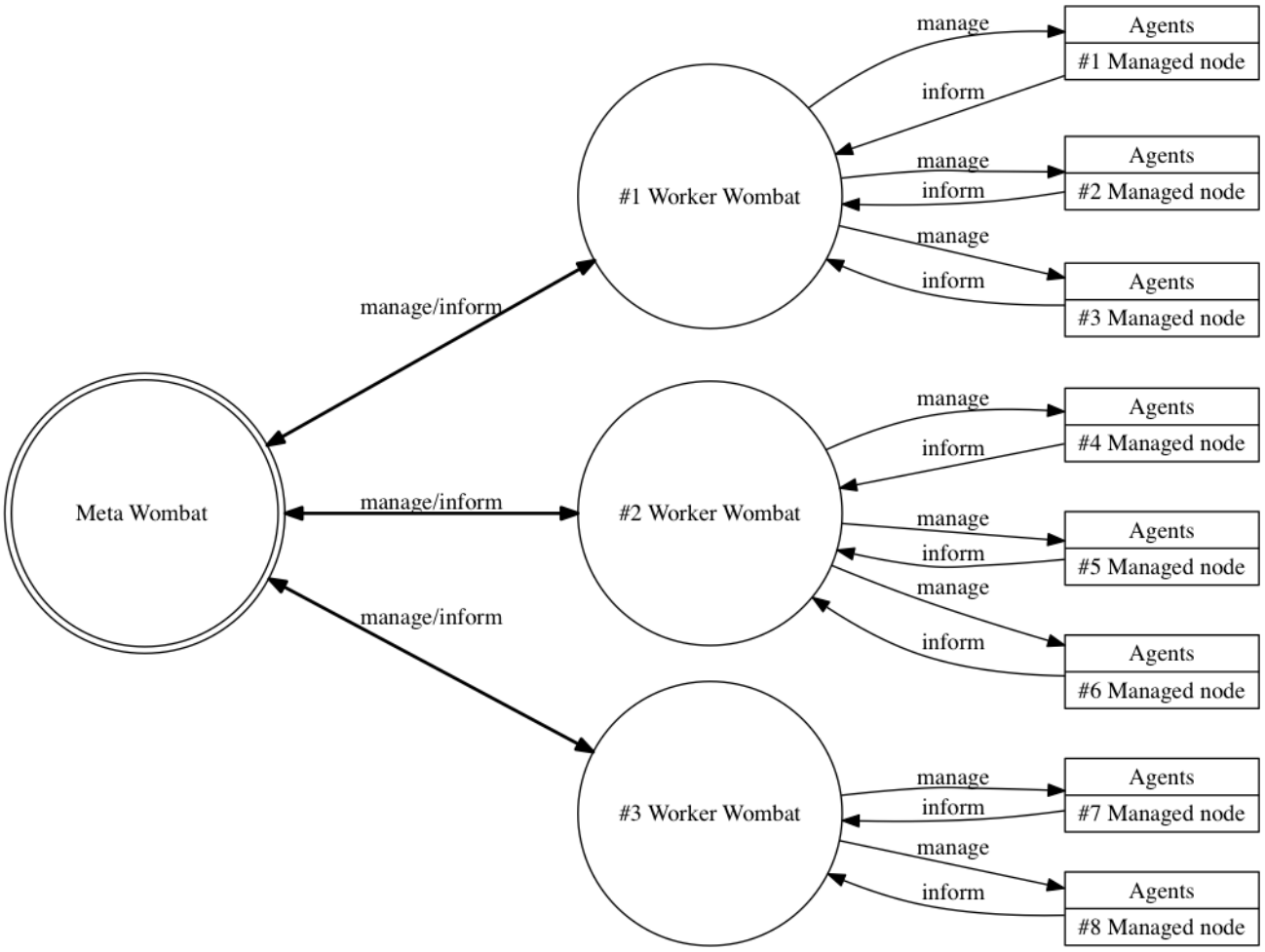

Figure 3: Actors.

– to maintain important properties related to managed nodes.

It is the main interface of the system with which the users can interact. Hence, Meta Wombat provides a REST API.

- *Worker Wombats.* Started by Meta Wombat if and only if the free capacity of the whole system is not sufficient for incoming requests. Meta Wombat instructs them to deploy and manage nodes and Worker Wombats monitor those nodes. The maximum number of managed nodes assigned to each Worker Wombat is determined based on the results of the soak test (to be discussed in Section 2.2). Worker Wombats must inform Meta Wombat about the important properties related to their managed nodes (e.g.: node names, cookies). Regardless of a failure of a Worker Wombat, Meta Wombat can start a new Worker Wombat and forward the responsibilities to the new instance (e.g.: monitor the orphan managed nodes), thus neither data nor managed nodes are lost.

- *Managed nodes.* Managed nodes are the Erlang nodes under the supervision of Worker Wombats. In the course of the RELEASE project, the managed nodes are the goal of deployment tasks initiated by Worker Wombats. However, managed nodes can be dynamically added and removed to/from Worker Wombats in general.

- *Agents.* Erlang modules and processes injected by Worker Wombats. They run on the managed nodes in order to provide monitoring information to WombatOAM.

The new system is a completely distributed approach even in terms of persistence. The approach makes Worker Wombats independent from each other, thus they do not effect the scalability of the others. Each Worker Wombat uses a dedicated data store (including a Mnesia database) in which only those data is present that is relevant to the Worker Wombat. There is only one exception that: the data related to the Topology and Orchestration subsystems. As the main entry point of the system is Meta Wombat, all deployment related data (e.g., release files, SSH keys) is essential to perform the tasks, because specific parts of the deployment process are done by Worker Wombats. Further details about the deployment process are given in Section 2.3.

As the determination of the required number of hosts (denoted by H_C^{Σ}) for C_M managed nodes is important, but not trivial, let us define it by an equitation as follows. As mentioned above, limits are set for Worker Wombats in terms of the maximum number of their managed nodes, which is denoted by WW_{Cap} and is equal to the capacity of a Worker Wombat. Considering the requirements of different releases there can be great differences. Thus, let us denote the maximum number of managed nodes deployed to a single host by H_{Cap}.

$$H_C^{\Sigma} = \overbrace{\left\lceil \frac{C_M}{WW_{Cap}} \right\rceil}^{\text{Worker Wombats}} + \underbrace{\left\lceil \frac{C_M}{H_{Cap}} \right\rceil}_{\text{Managed nodes}} + \overbrace{1}^{\text{Meta Wombat}} .$$

Although the implementation of Meta Wombat is in the next subsection, the underlying concepts and the key factors of the advanced system have been clarified here. The improved WombatOAM does satisfy the requirements about dynamic scalability, reliability and the ability to adapt to heterogeneous clusters under realistic loads. Meta Wombat is dynamically scalable, because it starts as many Worker Wombats as needed, and Worker Wombats are scalable while the load is lower than the determined upper bound. As Meta Wombat and the Worker Wombats are scalable subsystems, the improved overall system is not only scalable, but dynamically scalable (in the function of the managed nodes) because the coordinating actor is Meta Wombat. The reliability of the system has been confirmed by the fact that both Worker Wombats and Meta Wombat were able to resist the soak test even for weeks. In D4.3 [15] and in D4.4 [17] the ability to adapt to heterogeneous clusters under realistic loads had already been presented. Of course, this ability still remains.

2.1.3 The Realisation of Meta Wombat

As it was discussed in this section, Meta Wombat is the root of the Wombat-tree. It is the main entry point of the system because the transparency of the Wombat-tree is preferred, thus the Wombat-tree is hidden from the user. Therefore Meta Wombat acts as a router that dispatches and forwards the user's requests to the relevant Worker Wombat(s), and returns the accumulated result collected from the Worker Wombat(s).

The Wombat-tree is implemented by extending Wombat with a new application called *Meta*. The *Meta* application consists of all the services that are responsible for:

- creating and maintaining the Wombat-tree,
- starting deployment tasks,
- forwarding requests and the given responses between the user and the Worker Wombats.

As scalability and reliability are the key factors, the *Meta* application was implemented as cooperating server processes. This enables the concurrent execution of tasks, and guarantees that no dead-lock will appear during the execution.

The *Meta* application is the part of both Meta Wombat and Worker Wombat release. However, the behaviour of *Meta* is totally different in these releases. In case of a Meta Wombat release it provides the services described above, whilst in case of a Worker Wombat release it provides the main interface to communicate with Meta Wombat. The different behaviours are implemented by starting different server processes depending on the kind of Wombat (which can be either *worker* or *meta*) during the start procedure of the framework.

Considering a Meta Wombat release the description of the *Meta* application is the following. The *Meta* application is a standard Erlang application that initialises the main supervisor of the application. Figure 4 illustrates the supervisor tree of the application. The children of the supervisor, which are all server processes with limited responsibilities, can be summarised as follows.

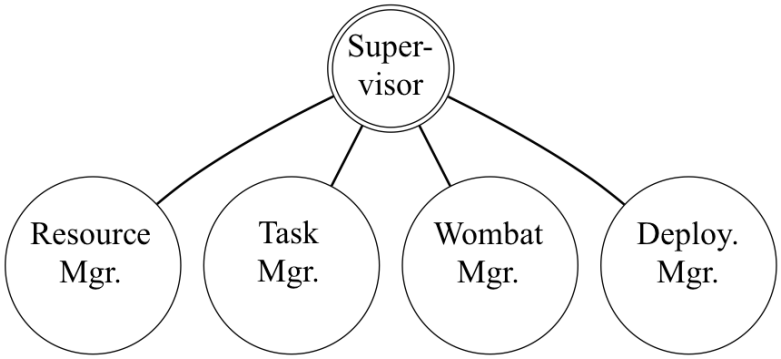

Figure 4: Top level supervisor of the *Meta* application.

- *Resource Mgr.* It is a resource server responsible for allocating and deallocating resources used either only by Worker Wombats or by both Worker Wombats and the managed nodes. The server ensures that no resource will be overused to provide the best performance for the Wombat-tree.

- *Wombat Mgr.* It is aware of the whole Wombat-tree. Worker Wombats are selected for and assigned to deployment tasks by the *Wombat Mgr.* server. It is also responsible for maintaining Worker Wombats. It monitors Worker Wombats and controls such scenarios as when a Worker Wombat is down. The performed recovery procedure varies based on the current usage level of the Worker Wombat. If this WombatOAM monitors no managed nodes and does not take part of any global actions (e.g.: deploying nodes), the Worker Wombat is terminated. Otherwise, as no managed node can become orphan, the recovery procedure tries to restart the Worker Wombat at first. If the restart fails, a new Worker Wombat is started on another host to takeover the managed nodes and tasks.

- *Task Mgr.* As the responsibility of *Wombat Mgr.* is limited to maintaining the Wombat-tree and the *Wombat Mgr.* is not aware of actions performed at a higher level, the *Task Mgr.* is responsible for extending the tree with a new Worker Wombat. The *Task Mgr.* ensures that all necessary steps are successfully performed and protects *Wombat Mgr.* from overload.

  As an example for a task handled by *Task Mgr.*, consider the task that consists of extending the tree with 40 Worker Wombats. The task is accomplished if and only if all the 40 Worker Wombats have been started and registered to *Wombat Mgr.* It can happen that 39 of the 40 Worker Wombats have been successfully started, but the initialisation of the remaining Worker

Wombat has failed. Thus, the task is not complete. In this event *Task Mgr.* tries to initialise the last Worker Wombat again. Note that the number of attempts are limited, hence no live lock can occur. If all the Worker Wombats are up or the number of attempts has reached the limitation, the task is finished. Next, *Task Mgr.* informs the client about the result of the task.

- *Deployment Mgr.* It is in charge of all currently running deployment tasks related and limited to Worker Wombats in the system. Comparing *Task Mgr.* with *Deployment Mgr.* the main difference is their abstraction level. Contrary to *Task Mgr.*, *Deployment Mgr.* operates on a low level and executes tasks that are the realisations of commands directly instructing the operation system of the physical machines.

2.1.4 WombatOAM's Built-in Recovery Mechanism and Fault Tolerance

At first sight, the root of the Wombat-tree – Meta Wombat – can be seen as a single-point of failure, however, this not true (not considering hardware failure, see later). The reasons are discussed in detail below.

First, Meta Wombat and its Worker Wombats are loosely connected. When a Worker Wombat crashes, Meta Wombat is aware of it and performs the previously discussed necessary actions. When a Meta Wombat crashes, the Worker Wombats still operate and Meta Wombat is restarted using the *heart* mechanism [7], which is Erlang's built-in subsystem for restarting Erlang VMs in case they terminate unexpectedly. Once Meta Wombat has been restarted, Meta Wombat checks the state of the Wombat-tree. It tries to connect to its Worker Wombats, and realises if some of its Worker Wombats have terminated. If so, it restarts the Worker Wombats. Second, all kinds of Wombats use persistent databases to maintain the statuses of the actions being executed, together with the actions to be executed in the future. This implies that all actions will be completed even if an action is interrupted during its execution.

The previously described concepts do not only hold for the high level actors (e.g., for Worker Wombat and for Meta Wombat), but hold for the subsystems, too. Thus, considering the case when a deployment process got interrupted because of a failure, once the Wombat-tree functions again, the deployment process continues.

Even if the hardware under Meta Wombat fails (so that Meta Wombat cannot be automatically restarted), the Worker Wombats will continue to run and monitor the managed nodes. When a new Meta Wombat is started, it will be able to connect to them to restore all capabilities of the Wombat-tree, including the ability to deploy new nodes.

2.2 Determining the Limits of Worker Wombats Using Soak Test

We employed the soak testing technique to detect any weaknesses of Worker Wombats in terms of several quality characteristics. We were focusing on reliability, robustness, responsiveness, fault tolerance and performance. Moreover, we applied this technique to determine the capability limit of a Worker Wombat. That is the maximum number of managed nodes that can be safely monitored by a Worker Wombat even for weeks.

Many lessons were learnt from each execution of the soak test that helped us to advance WombatOAM. The iteratively applied workflow consists of the following actions. At first we executed a soak test at least for a week, next the results of the execution were analysed to identify any weaknesses. Finally the weak points determined got addressed resulting in a better software. After several iterations were completed, the application of the workflow resulted in a massively reliable, effective and stable version of WombatOAM.

Throughout the rest of this subsection we present details about soak testing. At first soak testing in general is introduced, next the elaborated soak testing technique together with its execution envi-

ronment is described. Last but not least, we close this subsection by highlighting the outcomes of soak testing.

2.2.1 Soak Testing In General

Soak testing [13, 21] is a technique used to determine how the subject of the test behaves under massive load for certain period of time. Soak tests can point out such weaknesses of a system that cannot be detected by usual testing activities. As it is likely to happen that the proper functioning of a system will be degraded only after some days of intensive usage, no serious conclusions regarding the quality of the system can be drawn from testing it only for an hour. To put it in other words, the system might resist even heavy load for a short period of time, but cannot sustain the continuous expected load while being used for a much longer period of time.

2.2.2 Description of Soak Test

In case of WombatOAM, soak testing aims to detect the weakest parts of the system based on the observation of WombatOAM while WombatOAM manages a huge number of Erlang nodes for a long period of time. All managed nodes perform several actions that greatly exercise all the subsystems of WombatOAM., hence generating a huge amount of input with which WombatOAM needs to deal. Moreover, WombatOAM needs to tolerate such events when the releases running on the managed nodes are error-prone. But, these requirements are only one point of view.

Considering the user's perspective, the responsiveness of WombatOAM needs to be maintained even if a huge amount of data coming from its managed nodes flood WombatOAM. We developed a soak test technique that pays attention to both perspectives by constantly querying WombatOAM while it is being flooded with input data. Note that as soak test lasts for more than a week, flooding WombatOAM implies a continuously growing data set that WombatOAM needs to preserve and also services supporting both data retrievals and modification need to be available.

Now we present all the actors of the soak test, whose cooperation exercises WombatOAM as planned and described above.

- *Basho Bench* [1]. Basho Bench is a load-generation and testing tool for products implemented in Erlang. It was originally developed for benchmarking Riak by Basho Technologies. In the context of our soak testing technique, it commands the managed nodes to raise alarms and to generate logs either directly or by requesting the managed nodes to simulate an Erlang node being in a critical state. So, the subsystems – alarms, notifications, metrics – are exercised well. In each 1/100 second, 25 concurrent workers of Basho Bench instruct the managed nodes using RPC to do any of the above defined actions. Thus, Basho Bench generates the input for WombatOAM. The duration of the input generation is the complete execution of the soak test.

- *Managed nodes.* Managed nodes are a set of Erlang nodes executing the same release. They are the interfaces, through which Basho Bench generates input for WombatOAM. As it is described above, they can directly generate alarms and logs or can become as error-prone as possible. Whenever Basho Bench instructs the nodes to behave so, thousands of processes are being spawned each second. These processes send a huge number of messages to each other, and finally they crash because of various reasons. These operations turn the Erlang VMs into environments being in critical status, because the thousands of messages waiting in the messages boxes consume much memory, all the thousands of processes are starving in terms of CPU time and so on. Moreover, the processes are continuously crashing, which generates large error reports. Note that in such a scenario, the WombatOAM Agents still need to function normally regardless of their improper execution environments.

- *MegaLoad* [18]. MegaLoad is a load testing tool, developed by Erlang Solutions. The tool is developed in the context of the PROWESS project [8], which is funded under the FP7-ICT programme. Regarding soak test, the role of MegaLoad is to query data from WombatOAM via the REST interface (GUI is not included in the test) to simulate users by data retrievals. It employs a maximum of 10 concurrent processes to perform iteratively the different scenarios defined within the phase. A *phase* describes how test scenarios will be executed. Regardless of the currently executed scenario, the following important properties always characterise the execution. The number of requests sent within a second is limited to two requests, and HTTP connections are not closed at the end of each phase to test the web server component of WombatOAM in terms of availability, throughput and robustness. Each phase executes a randomly selected scenario from the six implemented scenarios. As scenarios describe user stories, the execution of a scenario manifests itself as a sequence of HTTP requests sent sequentially to the server. Although GUI is not included in the test, the user stories are designed by considering the most frequent use cases of the GUI. Not surprisingly, scenarios include exploring the collected logs, listing alarms and visualising metrics. The average number of the HTTP requests sent within a scenario is 8.

- *WombatOAM.* Obviously, the subject of the soak test is also an actor. For each execution of the soak test WombatOAM built from the latest stable source code was used. What is more interesting is the fact that not only one WombatOAM is included in tests, but two instances of them. The second WombatOAM plays an interesting role in the soak test. Namely, it is used to monitor the soak tested WombatOAM. All the collected data produced by the monitoring WombatOAM has been considered extremely valuable. Visualising the metric data profiling the soak tested WombatOAM from different points of view helped us a lot in various ways. For instance, it helped us to figure out the reason of errors, to point out the weakest parts of the system and to predict possible performance bottlenecks.

The whole system was designed for operating in a distributed environment to avoid drawing incorrect conclusions because of the required operations are beyond the limits of a single machine. Considering that more than 100 managed nodes were monitored by WombatOAM, the available CPU time for an Erlang VM would be infinitesimal if the whole system were deployed to a single machine. In such an environment the performance of any release is degraded, thus we employed two servers. Both of the machines are internal servers of a cluster maintained by Erlang Solutions. As the performance of WombatOAM is really relevant to us, the two WombatOAMs were run on one machine, whilst the others were deployed to the another machine. Figure 5 illustrates the deployment. This kind of deployment strategy allowed us to completely separate WombatOAMs from the other actors in terms of available resources and to test WombatOAM's behaviour in the worst case. By deploying all the managed nodes into one host, we could simulate the worst environment. Hence, we were able to analyse the WombatOAM agents whether they still function normally under difficult conditions.

2.2.3 The Outcome of the Soak Test

Thanks to soak testing, the quality properties of WombatOAM have been improved greatly. We have improved the maximal throughput, the computational capacity, the scalability and stability together with some other properties. As a single WombatOAM is the subject of the presented testing technique, the improved properties led to increased inner-node performance. For instance, consider that the monitoring capability of WombatOAM was more than tripled, *currently a single WombatOAM can manage 150 nodes reliably*, as opposed to the 50 nodes it could manage before these enhancements. This means that the current WombatOAM is able to manage as many nodes alone as three old WombatOAMs could.

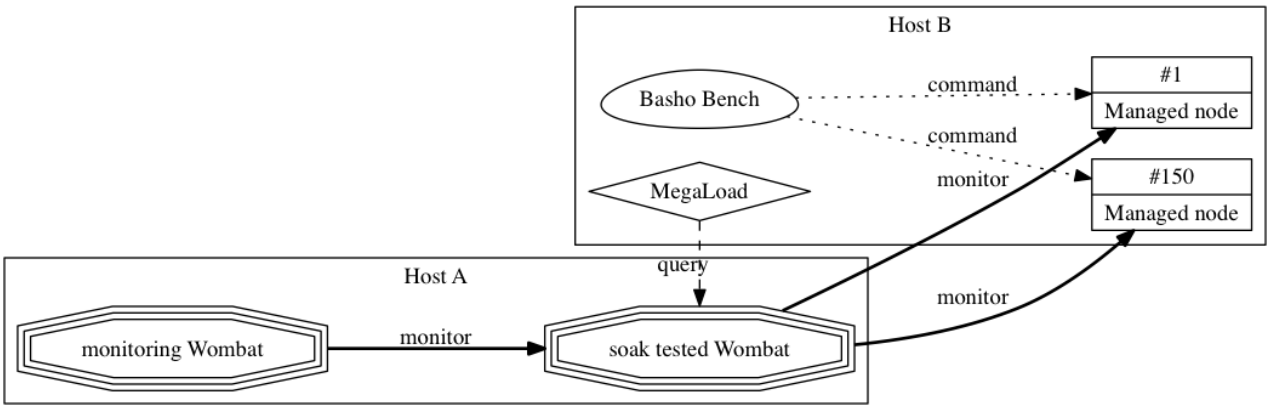

Figure 5: Deployment diagram of soak test.

All the subsystems scale well, and the level of concurrency is increased. Breaking down the improvements into the different subsystems:

- The subsystem of notifications is capable of handling logs that has the total size of 23 GB. Contrary to that, the limit of the old WombatOAM was around 200 MB.
- The subsystem of alarms can survive scenarios when the monitored system is totally overloaded generating continuously many kinds of alarms. The old version of this subsystem was overloaded in this case and continuously crashed. However, it must be noted that after each crash the old subsystem was able to restart usually, but it caused a negative effect on the whole system.
- In the subsystem of metrics, serious improvements were realised. Now, the system is able to receive and process a huge amount of data characterising 150 nodes using 214 metrics even for two weeks. The old version of the subsystem failed even in case of browsing historical data of 150 metrics collected from 100 nodes for 2 days. (Browsing historical data means that the managed nodes are not connected, thus no new input arrives, only the data already collected can be visualised.) Moreover, the storage requirement of metrics has decreased by a factor of ten.

As it might be already apparent, the stability of the tool was enhanced together with the scalability. WombatOAM can tolerate even a massive load for weeks. Although some errors have been eliminated as well, under some abnormal circumstances the tool may still crash, but it always manages to successfully recover itself. It can recover even after some partial data loss occurred or after abnormal termination.

As mentioned above, the storage space requirement was consolidated: in the soak test scenario (where we automatically generated lots of notifications and alarms on the nodes) the data collected from about 150 managed nodes during a two week period occupied less than 23 GB.

Considering memory requirement, WombatOAM was improved so much that it has a constant, relatively low memory footprint in function of the number of the managed nodes. The memory usage still remains constant even if WombatOAM operates for weeks.

To summarise this section it can be said that soak testing was worth the effort. By soak testing WombatOAM, it has evolved into a stable, scalable, reliable, robust and fault-tolerant tool that is capable of operating in a real-life environment.

2.3 Perfecting and Tailoring Orchestration

WombatOAM Orchestration is the subsystem of WombatOAM that is responsible for deploying Erlang nodes into the cloud or specified host machines. In this section, we will describe the improvements we

made in Orchestration since the last deliverable. We had two main goals that we achieved with these improvements:

1. To be able to deploy a wider range of applications and systems, most importantly systems that use SD-Erlang [3, 16].
2. To be more performant, scalable in size, reliable [23] and robust.

When we say Orchestration v1, we mean the previous implementation of Orchestration before these improvements, while Orchestration v2 is the improved version. Orchestration v2 kept the support for all the features that we have delivered in the previous deliverables, most notably:

- the ability to deploy on both homogeneous and heterogeneous clouds,
- supporting Amazon EC2, HP Cloud and Rackspace, which was the focal point of deliverable D4.3 [15] and D6.5 [18];
- and capability-matching deployment, which was detailed in deliverable D4.4 [17] and D6.6 [19].

Even though we used a traditional cluster for executing the measurements in this deliverable, *all of these improvements are as valuable in the cloud as on the cluster*, and some of them (like making the handling of external resources like virtual machine instances more reliable) are actually more vital in the cloud.

Orchestration summary In order to explain the improvements we made, we first need to describe how WombatOAM Orchestration works on the high level. In order to deploy a set of Erlang nodes, the user needs to perform the following steps:

- Register an infrastructure provider in WombatOAM that can host our nodes. WombatOAM can use cloud providers (like Amazon EC2, HP Cloud and Rackspace) and it can also use a "physical" provider (i.e., utilize already running machines). Registering a cloud provider means giving WombatOAM access to the cloud account of the user.

- Upload an Erlang release archive (*release* for short) to WombatOAM that can be transferred to the host machines and started. The *Deployment hooks* paragraph below provides an example of a release.

- Create a node family. A *node family* is an entity that knows everything that is needed for deploying the node: it points to a release, and contains any number of deployment domains, which all point to an infrastructure provider.

- Nodes can be deployed by simply asking for a number of new nodes of a certain node family. First one of the deployment domains is selected, then the necessary hosts are allocated, and finally the release referred by the node family is transferred to all hosts and started, thereby creating the requested number of new Erlang nodes. (D4.4 [17] and D6.6 [19] detail WombatOAM's domain selection mechanism.)

Deployment hooks When a release was uploaded into WombatOAM using Orchestration v1, the user had to specify a start command, a stop command and a bootstrap command. These were usually shell scripts added to the release archive that WombatOAM could call when it wanted to start the node, stop the node, or make the node connect the other nodes in the family.

Let's examine the Riak example that we have shown in deliverable D4.3 [15]. The following commands were configured when the release archive was uploaded to WombatOAM:

- Start command: `bin/riak start`. The `bin/riak` script is part of Riak (and the release archive), which starts Riak when invoked with the `start` parameter.
- Stop command: `bin/riak stop`. The `bin/riak` script stops Riak when invoked with the `stop` parameter.
- Bootstrap command: `bin/bootstrap`, which is a simple script we created and the source code of which is shown in listing 1. When WombatOAM calls the bootstrap command, it will pass the name of the bootstrap node (i.e., a node that is already connected to the other nodes of the node family) as the last argument. This way the bootstrap script will know which node to join to.

Listing 1: 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)
```

This works well with releases like Riak, but it doesn't work with all Erlang systems. For example the applications that use SD-Erlang [3, 16] usually have a master-workers architecture in which first we deploy all worker nodes, and then we deploy the master node and ask it to perform a calculation. The master node expects the list of workers to be provided in a configuration file. So we could deploy the Erlang nodes, use WombatOAM's REST interface to retrieve the list of worker nodes, then generate and copy the configuration file to the master node, and then ask the master node to start the calculation.

The question is, is it possible to automatically generate the configuration file? We could try to add that logic to the start command of the Erlang release of the master node, but then we would need to hard code in it not only WombatOAM's location but even the node family in which we are interested. Another solution would be to write a script that runs besides WombatOAM and generates and transfers the configuration file, but we would still need to manually invoke that script.

The solution was to add a feature to WombatOAM called *node hooks*: with that feature, we can create a pre node start hook for the master node that will generate the configuration file on the WombatOAM host (either as a shell/Python/etc. program or as an Erlang function call), and create another pre node start hook for copying the configuration file to the master node. The hooks are executed sequentially, so the sequence of actions will be generating the configuration file, transferring the configuration file, and starting the master node.

In general, when deploying a set of nodes with WombatOAM, the user may specify a set of actions that need to be executed before or after certain events happen. Each such occasion where WombatOAM is able to execute a list of custom actions is called a hook.

There are six hooks: `pre_node_start`, `post_node_start`, `pre_node_stop`, `post_node_stop`, `join` and `leave`. Their names describe when they fire, e.g., the actions assigned to the `pre_node_start` hook will be executed when the user issues WombatOAM to start a node, but before it is actually started. Both node families and nodes may have any number of actions assigned to different hooks. (But note that these are all *node hooks*, even those that are assigned to the node family, so they fire when any node of the family is started or stopped, and not when the node family itself is created or deleted.)

There are three kinds of actions: `run`, `call` and `send_file`. `run` executes an executable file with the given parameters, `call` calls on Erlang function and `send_file` transfers a file to the node. The first two actions can be executed either on the WombatOAM side or on the deployed node side.

Hooks can be used not only besides the start command and stop command but even instead of them. In section 3.1.2, we will use a pre start hook to start the node by executing a `run` action that starts the Erlang node itself.

See the detailed description of node hooks in appendix 4.

Introducing node family processes and node processes In Orchestration v1, there was one process in a Middle Manager to handle all node families and one process to handle all nodes. In Orchestration v2, we eliminated these global processes. Now each node family and each node managed by WombatOAM Orchestration has two processes: a manager and a worker process. Other components of WombatOAM can ask the manager process to perform an operation on the node family or node. E.g., if the user asks the node to be started, the REST handler process of this request will ask the manager process of that node to perform the start operation. The manager process will return an acknowledgement and delegate the work itself to the worker, since it may take a long time, and we want the manager process to stay responsive.

These processes use the database in an efficient way. We keep using the Mnesia database, but we were careful to design the new system in a way that for each row in the database there is always only one process that might update that row, hence there is no need for locking and transactions. (As a rule of thumb, using transactions is on average more than 10 times slower than not using them [12].) The manager processes have their own persistent tables (one for node families and one for nodes), where each row represents a node family or node. Each row can be written only by the corresponding manager process and nobody else. If another process wants to execute an operation that might change the state of the node family or node, it needs to contact the manager process. On the other hand, for maximizing efficiency, if another process wants only to read the state of a node family or node, it can do that by reading the state from Mnesia. There is no need to involve the manager process in this case, so we avoid a possible bottleneck. The worker processes also have their rows in another table, in which they store the external resources (hosts, SSH keys, security groups in the cloud) that need to be released when the node family or node is deleted.

The manager processes that can react to the events that happen on the nodes, including normal behaviour and errors, which improves reliability and robustness. If we have many node families or nodes, we will have many processes, which, together with the fact that they use the database without transactions, improves scalability and performance drastically.

Handling external resources When using a cloud provider, WombatOAM allocates resources like virtual machine instances, SSH keys and security groups for node families and nodes. When the nodes are terminated and node families are deleted from WombatOAM, these resources are expected to be released. We designed Orchestration v2 to minimize the probability of not releasing an external resource. Note that it is not possible to reduce this probability to 0: consider the scenario where the user deploys a virtual machine and then shuts down WombatOAM. Without an explicit instruction, WombatOAM must not stop the deployed nodes, so if it is never restarted, or its database is deleted, the resources cannot be released.

We made releasing resources more robust by periodically retrying the deletion of the resources many times, since it may happen that our failure to delete the resource is due to not being able to communicate with the cloud provider, or due to the cloud provider refusing our request because the resource to be deleted is used by another resource (which itself may be under deletion, in which case the deletion of our resource will be allowed when the deletion of the other resource has been completed). The external resources are stored in a persistent Mnesia table, so even if processes crash inside WombatOAM or WombatOAM itself is restarted, retrying releasing the resource can continue.

Rendering the templates on the host machine When a node was deployed in Orchestration v1, the release archive was altered before it was transferred to the node. The reason is that in most Erlang releases, the name of the node is stored in a file in the release (this file is typically called `vm.args`, so let's use that as an example), and since the user wants the nodes to have different names, WombatOAM needs to modify each release archive so that the `vm.args` file in it contains the node name calculated by WombatOAM. This was done by expecting the user to have `vm.args` within the release archive contain the string {{ `node_name` }} instead of a real node name, which then WombatOAM would replace with the concrete node name before transferring the release. This procedure is called *template rendering*, and it was implemented by unpacking the release, replacing a string in `vm.args` and packing it again, creating the final release archive that could be copied to the host machine.

This procedure of performing template rendering on the WombatOAM host machine for each deployed node works on a small scale, but it would take up a significant amount of resources when deploying more than a handful of nodes. Unpacking and repacking a 50MB release archive takes altogether 4-5 seconds on a high-end laptop.

Hence in Orchestration v2, we transfer the release archive unchanged to the host (i.e., its `vm.args` file will contain {{ `node_name` }}), and perform the template rendering on the host machines after unpacking the release archive there. Both raw performance and scalability increases: the former because we eliminated an unpack-pack cycle, and second because the template rendering is distributed, with each node having to perform it only once.

Node name base templates As described in the previous paragraphs, different deployed nodes need to have different names. Previously WombatOAM was using a simple method for ensuring this, namely the node id (which is a generated UUID) was used as the node name base, hence node names like `54c4544a-c0e5-4400-8865-1b7add2163dd@host` were created. In order to have more human-friendly node names, the users can now specify a node name base template when deploying a set of nodes.

For example if the nodes are deployed with the following REST request:

```
URL: <wombat>/orch/node-family/<family-id>/node
Body: {"amount": 2,
       "node_name_base_template": "mynode-{{ index 4 }}"},
```

then the resulted node names will be `mynode-0001@host` and `mynode-0002@host`.

In general, the node name base template is a string that might contain patterns which are expanded. All other characters will go into the node names unmodified. The following patterns are accepted:

- {{`node_id`}}: Will be replaced with the identifier of the node.
- {{`index`}}: When the user deploys N nodes, this pattern is replaced with the index of the node (which goes from 1 to N by default). So if, e.g., 3 nodes are deployed with template `node-`{{`index`}}, the generated node name bases will be `node-1`, `node-2` and `node-3`.
- {{`index PADDING`}}, e.g., {{`index 4`}}: Similar to {{`index`}}, but the index will always contain at least `PADDING` characters, padded with zeroes. So, e.g., `node-`{{`index 4`}} might become `node-0001`.
- {{`startindex DIGITS`}}, e.g., {{`startindex 20`}}: This is a global setting for the template, which specifies the first index to be used. So if, e.g., 3 nodes are deployed with the template `node-`{{`index`}}{{`startindex 20`}}, then the generated node name bases will be `node-20`, `node-21` and `node-22`.

The default node name base is {{`node_id`}}, which is the original behaviour of using the node id as the node name base.

Integrating Orchestration v2 to Meta Wombat When integrating Orchestration with the distributed approach introduced by the Wombat-tree, we needed to decide which component of Orchestration needs to be executed in Meta Wombat, and which in the Worker Wombats.

Since nodes belonging to the same family can be distributed among different Worker Wombats, we keep the family processes and family-level data on the Meta Wombat, so that they can orchestrate how the nodes in the node family are deployed, started and connected with each other, and they can also manage family level tasks like creating a security group in the cloud for the node family's deployment domains.

The node processes on the other hand are running on the Worker Wombats, which provides us the scalability that we have been aiming for. When the nodes are deployed, the node processes receive enough information from their family process to be able to perform the deployment (e.g., they receive the details of the provider), but they don't receive any information they don't need. For instance they don't know about the nodes, and the whole Worker Wombat doesn't know about the nodes in other Worker Wombats; even those that are in the same node family.

3 Measurements

In this section the evaluation of WombatOAM's performance and scalability is presented. The section first gives the necessary background by introducing the cluster, the release deployed to the cluster and the automated deployment technique used to execute the tests. Next, the evaluation strategy and the designed tests are presented, followed by a discussion about the validity of the tests. Last, but not least, the concrete test configurations are described, and the corresponding results and measurements are shown. In conclusion, the achievements and the lessons learned are highlighted.

3.1 Background

3.1.1 The Athos Cluster

The measurements described in this deliverable were performed on the Athos cluster provided by the consortium partner EDF. Athos has 776 nodes (we will refer to them as hosts, in order to avoid confusion with Erlang nodes), each with 24 Intel Xeon E5-2697 v2 CPUs and 64GB of RAM. In the RELEASE project we have a simultaneous access to up to 256 hosts (6 144 cores) for up to 8 hours at a time. Each host is a GNU/Linux system: both SSH and SCP can be used on them (to execute remote commands and to copy files between the machine), and they also share an NFS drive.

Hosts can be allocated using the SLURM resource manager [22], which provides the `sbatch` command for scheduling the execution of a program as a job, the `SLURM_JOB_NODELIST` environment variable within that program to retrieve the list of hosts available for this job in a compact format, and the `scontrol` command to convert that compact format into a simple host list.

Listing 2 shows a very simple script that can be scheduled with SLURM: during its execution, it prints the host names of the available hosts.

Listing 2: A simple example script that can be scheduled and executed by SLURM

```
#!/bin/bash

hosts=$(scontrol show hostname $SLURM_JOB_NODELIST)
echo "The following hosts were assigned to this job:"
for host in $hosts; do
    echo "$host"
done
```

The following line shows how we can ask SLURM to schedule this script (called `myscript.sh`):

```
$ sbatch --exclusive --hosts 3 --time 00:01:00 --output out.txt myscript.sh
Submitted batch job 2386574
```

We specify that we need exclusive access to 3 hosts for one minute, and that when our hosts are granted, we want `myscript.sh` to be executed, and its output should be redirected into `out.txt`. SLURM returns the identifier of our job. When SLURM decides how to schedule jobs, it takes into account both the number of hosts requested and the time requested. Generally, small and short jobs need to wait less than large and long jobs.

After our job has been executed, we can see that the output file contains the list of hosts printed by our script:

```
$ cat out.txt
I have the following hosts:
host199
host200
host213
```

Note that our script was executed in one copy, irrespective of the number of hosts allocated. Each host is running an SSH server, which makes it easy to execute remote commands on the allocated hosts.

3.1.2 The ACO Benchmark and Its Deployment with WombatOAM

In our performance measurements, we deployed and executed a system developed for benchmarking purposes by The University of Glasgow. This system is called ACO, which stands for Ant Colony Optimization. The details of ACO are described in deliverable D3.4 [20].

As for this deliverable, there are two important aspects of ACO. The first is that it is a distributed system where each Erlang node participates as an ant colony, processes within that node implementing the single ants. The other important aspect is that when starting ACO, we parameterise it by specifying the problem to be optimized (in an input file), the number of ants per colony, the number of global iterations (the colonies exchange information at the end of each global iteration), and the number of local iterations within each global iteration. This parametrisation means that we can determine the size of the computing task quite precisely, which makes this an ideal benchmark.

The ACO version we used for our measurements has a simple collaboration mechanism: we start any number of worker nodes and one master node, and call `aco:run` on the master node and pass to it the list of worker nodes to be used for the calculation (along with the list of other parameters described above). After completing the specified number of iterations, `aco:run` returns with the best solution found by the ant colonies.

We made only one modification to ACO: we modified the `aco:run` function not only to execute the ACO calculation, but also to store the start time and end time of the calculation.

The ACO was deployed using the same general procedure as described in deliverable D4.3 [15]: registering a provider, uploading a release, creating a node family and finally deploying the nodes. The details are different however from the Riak deployment showed in D4.3, for two reasons: (1) we used a cluster instead of a cloud; (2) we deployed a system with a master-worker architecture (ACO) instead of a masterless system (Riak). Furthermore, since this time we were creating benchmarks (which could be scheduled by SLURM), we needed to write a script that automated the whole deployment and measurement process.

In the following paragraphs, we detail how the benchmarks script deploys ACO.

Registering a provider WombatOAM provides the same interface for different cloud providers which support the OpenStack standard or the Amazon EC2 API. WombatOAM also provides the same interface for using a fixed set of machines. In WombatOAM's backend, this has been implemented as

two driver modules: the *elibcloud* driver module which uses the elibcloud and Libcloud [9] libraries to communicate with the cloud providers, and the *SSH* driver module that keeps track of a fixed set of machines. (Note that both the elibcloud and SSH driver module use SSH and SCP to configure the machines.) For this deliverable, we introduced a third driver module called *Meta*, which dynamically asks machines from the Resource Manager of Meta Wombat (see section 2.1.3). This was necessary because Meta Wombat's Resource Manager handles all hosts provided by Athos, and WombatOAM Orchestration should not interfere with it. Furthermore, the Resource Manager is more sophisticated than the original SSH driver module, because it allows us to let a specified number of nodes to run on the same host.

In the benchmark script, the provider is registered with the following REST request:

```
URL: <wombat>/api/orch/provider
Body: {"name": "Meta Provider",
       "type": "wo_orch_meta_driver"},
```

Uploading a release In D4.3, we showed a scenario where a few Riak nodes were deployed. The user uploaded the Riak release to WombatOAM, and WombatOAM used SCP to transfer the Riak releases to the hosts.

In this deliverable, we created a package that contained the compiled ACO code. Since ACO is not implemented as an Erlang release, but simply as a set of Erlang modules, the ACO package is not a real Erlang release archive. But this not a problem, since the only important aspect from WombatOAM's point of view is that we can start and stop an Erlang node using this package.

The ACO release can be uploaded to WombatOAM using the following REST call:

```
URL: <wombat>/api/orch/release
Body: {"name": "ACO Worker Release",
       "cookie": "wombat",
       "start_cmd": "nop.sh",
       "stop_cmd": "nop.sh",
       "release": "<base64 encoded tar.gz>"},
```

First the request states the name of the release and the cookie to be used for connecting to the nodes after deployment. Then it specifies the start and stop commands, which are called when WombatOAM wants to start or stop the node. In this case, they are empty scripts, because we will use node hooks to actually start the nodes. Finally the release archive in sent in a base64 encoded format.

Defining a node family The next step is creating the node family, which is the entity that refers to a certain release, contains deployment domains that refer to certain providers, and contains other information necessary to deploy a node.

First we create the node family without any properties besides its name:

```
URL: <wombat>/orch/node-family
Body: {"name": "ACO Worker Node Family"}
```

Then (using the family id returned in the response) we attach the previously uploaded release to the node family:

```
URL: <wombat>/orch/node-family/<family-id>/release
Body: {"release": "<release_id>"}
```

We also need to create a deployment domain that specifies (1) which provider we want to use for provisioning machines; (2) the username that shall be used when WombatOAM connects to the hosts using SSH:

```
URL: <wombat>/orch/node-family/<family-id>/domain
Body: {"provider": "<provider_id>",
       "ssh_user": "<current_username>"}
```

In the previous deliverables, we used a *start command* to start a node once deployed by WombatOAM. The start command was specified when creating the node family, so that the node family knows how to start all of its nodes. The start command is a string (e.g., `bin/mysystem start`), which is executed by WombatOAM when it wants to ask the node to start. The stop command is analogous to the start command, but it is called when the node should be stopped. In case of deploying a standard Erlang release as we did with Riak, WombatOAM added the name of the node to the `vm.args` file of the node, thus the start command didn't need any other parameters to start the node.

In section 2.3 we introduced a more powerful method for specifying how to start and stop nodes, called *hooks*, so we used a hook here instead of a start command. This way we can pass the name of the node to the start script of the release.

The following REST call creates a pre node start hook, which is executed when WombatOAM is starting the node:

```
URL: <wombat>/orch/node-family/<family-id>/hook
Body: {"node_hooks":
       {"pre_node_start":
        [{"action": "run",
          "location": "node",
          "executable": "bin/aco_start",
          "args": ["-sname", {"var": "node_name"},
                   "-setcookie", "wombat",
                   "-detached"]}]}},
```

The meaning of the fields is defined in appendix 4: in short, we ask WombatOAM to run the `bin/aco_start` program on the host with the `args` list as parameters. `bin/aco_start` is contained by the ACO release package. The string arguments in `args` are kept as they are, but the `{"var": "node_name"}` argument is replaced with the name of the node.

The release contains the following file as `bin/aco_start`:

```
#!/bin/bash

# Add the "ebin" directory to the Erlang path, and pass all arguments to erl.
erl -pa ebin "$@"
```

As the comment points out, `bin/aco_start` passes its parameters to `erl`, plus it specifies the `-pa` option. `erl` is the program that starts the Erlang Virtual Machine. The arguments have the following meaning for `erl` [6]:

- `-pa ebin`: Add the `ebin` directory (the directory containing ACO modules) to the path list. This means that if a function in an ACO module is called, Erlang will find the module and will be able to call the function.
- `-sname node_name`: Use the given node name for this node. The node name is a short name, which means that they have the format `name@host`, where `host` is a locally known name of the host machine. The alternative is using a long name, in which `host` is a fully qualified domain name or an IP address. Since SLURM provided us with local host names, we used short names. (WombatOAM itself supports both long and short names.)

- `-setcookie wombat`: Erlang nodes can connect with each other when they use the same cookie (or when the cookie of the target node is explicitly set in the source node). By using the same cookie (`wombat`) on WombatOAM and all the hosts, we ensure that they can communicate with each other via Erlang Distribution.

- `-detached`: This option makes `erl` start the node in the background and then return. This is necessary because WombatOAM waits for each hook to return, and proceeds with the next step (executing the next hook, trying to establish a connection towards the node, etc.) only afterwards. Without `-detached`, the execution of `erl` would not return, and WombatOAM would wait indefinitely for it to terminate.

The reader may notice that we haven't asked the node to actually do anything. This is because in ACO, the worker nodes sit passively until they are asked by the master node to perform some work. The reader may also notice that we haven't defined a stop command or a pre node stop hook. That would be usually required, but in our case it can be skipped, because after our script is executed, SLURM will terminate all processes that our script has started.

After creating the node family, we deploy the worker nodes with the following REST request:

```
URL: <wombat>/orch/node-family/<family-id>/node
Body: {"amount": <number of nodes to deploy>,
       "autostart": true,
       "node_name_base_template": "aco{{ index 4 }}"},
```

Because of specifying {`"autostart": true`}, the nodes will be not only deployed but also started using the pre node start hook. The `node_name_base_template` option is used to generate the node names, which in our case will be `aco0001@host`, `aco0002@host`, etc.

After starting a node, WombatOAM is trying to establish a connection towards it. When a connection is established towards all nodes, our benchmark script retrieves the list of all ACO worker nodes from WombatOAM and puts the list into a file called `aco_nodes`. Then it starts an ACO master node and calls `aco:run` in order to run the ACO algorithm:

```
erl -pa "<ACO ebin directory>"
    -sname aco_master
    -setcookie wombat
    -s aco run ../aco_inp 100 50 100 aco_nodes
    -s init stop
    -noinput
```

The `-pa`, `-sname` and `-setcookie` options are the same options we used for starting the worker nodes. The additional parameters are the following:

- `-s aco run ../aco_inp 100 50 100 aco_nodes`: Call the `run` function in the `aco` module with a list as a parameter. The list will contain the rest of the arguments: the name of the input file describing the ACO problem to be calculated; the number of ants per colony; the number of global and local iterations; and the name of the file containing the names of worker nodes.

- `-s init stop`: Stop the Erlang node by calling `init:stop()` when `aco:run` has finished.

- `-noinput`: Don't try to read any input from the standard input or terminal.

3.2 Designing Tests

In the context of the RELEASE project, the main role of WombatOAM is to provide the scalable infrastructure for deploying thousands of Erlang nodes. Thus, we designed tests that allow us to characterise the deployment capability of WombatOAM. As the deployed nodes can be monitored by WombatOAM and monitoring is very likely to effect the performance of the managed nodes, we devised tests to determine the overhead of monitoring.

3.2.1 Description of the Tests

Characterising the deployment capability of WombatOAM As we wish to determine the scalability of WombatOAM in terms of deploying nodes, we need to characterise the cost of deployment (in terms of time). As a sequence of requests must precede the real deployment procedure that cannot be performed within one step, we define the *deployment time* as the elapsed time between two specific checkpoints. The first checkpoint is the beginning of the real deployment procedure, which is the moment when a deployment request describing a concrete number of nodes executing ACO arrives. The second checkpoint is the end of the procedure, which is the moment when WombatOAM becomes aware of the fact that the deployment of all the ACO nodes has finished.

The deployment time greatly depends on the number of the nodes to be deployed (further denoted by N_C), thus the time required to accomplish the deployment procedure can be given as a function of N_C. At this point, let us highlight the fact that the deployment time can also be affected by WombatOAM's monitoring service! Moreover, the variable expressing the maximum number of nodes deployed to the same host may influence the deployment time as well. Note that if the maximum number of nodes deployed to the same host was determined in such a way that the nodes compete for the resources of the host, the performance of all the deployed nodes are very likely to be degraded.

Ergo, the deployment time of the procedures described by the two scenarios, which have been named E_{fair} and E_{unfair}, needs to be determined as described above. The fairness of the scenario corresponds to the deployment strategy from the view point of the ACO application. Namely, E_{fair} pays attention to the available cores of a host to ensure that there is at least one core for each deployed node. Hence, the maximum number of nodes deployed to the same host is less than or equal to the number of cores available on the host. On the other hand, in the E_{unfair} scenario, the number of nodes deployed to the same host is three times larger than the number of available cores. Since we had access to 256 physical hosts with 24 cores each (i.e., 6 144 cores altogether) in the Athos cluster, using E_{unfair} was necessary to test the deployment of 10 000 nodes.

Besides these, as we also wish to investigate the effect of WombatOAM's monitoring, thus we define two variants of E_{fair}. The only difference between $E_{fair_{on}}$ and $E_{fair_{off}}$ is the status of WombatOAM's monitoring. Monitoring can be switched off or on. The former means that WombatOAM can be used only to deploy and to start the nodes, whilst the latter means that WombatOAM can also be used to collect metrics, notifications and alarms from the running nodes.

By separating all these scenarios, the scalability of WombatOAM can be adequately characterised, because the time required to accomplish the deployment procedure can be given as a function of N_C. Therefore, the scalability of WombatOAM will be depicted using three functions expressing the deployment time. To approximate these functions, the required time to carry out the deployment procedure specified by the scenarios needs to be determined for different values of N_C.

Determining the overhead of monitoring As the overhead of any feature not contributing to the result of an execution needs to be infinitesimal, we want to make sure that the generated overhead of WombatOAM is small enough. Therefore, we measure the execution time of ACO with and without monitoring. The difference between the measured values is caused by the monitoring, thus we can determine the overhead of monitoring in the function of the original execution time. This means

that we can forecast the execution time of the program when the program is being monitored by WombatOAM.

3.2.2 Evaluation Strategy

To adequately evaluate each attribute of WombatOAM, m experiments need to be carried out. Each experiment is a sequence of observations. An experiment records the observed values corresponding to all possible values of a predictor variable. The domain of the predictor variable is a fixed set containing n items. Thus, the result of an observation is a pair associating the value of the predictor variable with the observed value, whilst an experiment contains all the n independent observations covering the domain of the predictor variable. Based on the experiments we are able to approximate the function characterising WombatOAM's performance.

Regarding the deployment capability of WombatOAM, there is one significant difference between the E_{fair} and E_{unfair} trials. That is the domain of the predictor variable expressing the number of nodes to be deployed. Regarding E_{fair} the domain is $X_{fair} = \{x_i \quad | \quad 500 \leq x_i \leq 5000, \quad x_i \pmod{500} = 0\}$, whilst in the context of E_{unfair} it is $X_{unfair} = \{x_i \quad | \quad 1000 \leq x_i \leq 10000, \quad x_i \pmod{1000} = 0\}$. These items express the number of the nodes to be deployed (N_C). For all x_i the time required to accomplish the deployment procedure is recorded as the result of the observation. As $m = 5$ experiments are planned to be carried out for each trials, 5 observed values are recorded for each x_i. The trimmed mean of each 5 values are calculated and used to plot the approximated function of the examined characteristic. The trimmed mean is better than the standard mean, because it is less sensitive to outliers, thus it is more robust [5].

Regarding the overhead of monitoring, a similar approach is employed. The domain of the predictor variable is the following set: $X = \{x_i \quad | \quad 10 \leq x_i \leq 150, \quad x_i \pmod{20} = 0\}$. These items express the number of the nodes (N_C) executing ACO. As the status of the monitoring service, which can be on or off, plays a key role here, two functions need to be approximated, which are $T_{E_{on}}(N_C)$ and $T_{E_{off}}(N_C)$, therefore two trials of experiments need to be performed. Each trial defines $m = 5$ experiments to record the execution time of ACO with a fixed number of iterations. Again, the trimmed means of each 5 values are calculated and used to plot the approximated functions of the examined feature.

Besides these, during the experiments the following properties need to be checked:

- WombatOAM remains responsive and reliable.
- WombatOAM's knowledge about its managed nodes is complete (nothing is left out) and correct (information about nodes is valid).

3.2.3 Validity of Tests

Here, we reason about the validity of the designed tests, meaning why the experiments can prominently confirm both that WombatOAM is capable of deploying thousands of nodes in a scalable way and that the overhead of monitoring is not intrusive.

At first let us raise a question. Which requirements does a software need to fulfil to be considered as a scalable software in terms of size? A system scales in terms of size if it has the ability to easily expand and contract its resource pool to accommodate heavier or lighter loads or number of inputs. Considering the function f that expresses the execution time in function of the size of the task, the slope of f is small if f expresses the execution time of a scalable system. For instance, considering linear functions, their gradient is a constant. This property implies that f is linear in worst case, thus the execution time increases linearly with the size of the input. However, observe that the level of scalability is neither uniform nor irrelevant. It can be considered as a key property of the software if the size of the input is enormous. It is especially true in the context of the RELEASE project where we are specialised for large scale systems. Thus, the level of scalability is good enough for us if and

only if the slope of f is less than 1. In case of a function expressing the deployment time it means that deploying $2x$ nodes is faster than deploying x nodes twice. Therefore, if the level of scalability related to the function expressing the deployment time is good enough, we can say that WombatOAM is capable of deploying thousands of nodes in a scalable way.

The overhead of monitoring can be said to be non-intrusive if the increase of the execution time of a given task on the monitored node caused by monitoring is relatively small. This means that the influence of monitoring does not alter the characteristic of the monitored software.

3.3 Executing Tests

Before we deeply discuss the implementation of tests, let us highlight some important properties. First, note that experiments were performed automatically using the technique introduced in Section 3.1.2. Second, be aware of the independence of any two observations. We allocated the machines via SLURM in the beginning of each observation, and released all of them at the end. Hence any two observations are independent from each other, because they were performed individually, and they had no effect on each other. However, it can be said that observations belonging to the same experiment slightly correlate with each other considering the noises of the cluster that took place while these observations were being performed. An example of such noise is the outlier load temporary occurring in the internal network of the cluster. As not only one experiment took place, the effect of such noises can be relaxed by using the robust method of averaging. In conclusion, the experiments are not misleading.

3.3.1 Test Configuration

Section 3.2.1 described the two main performance characteristics of WombatOAM that we set out to measure: WombatOAM's scalability and the additional load that is placed on the managed nodes by monitoring. This section details the test configurations for measuring both of these.

Measuring deployment time of WombatOAM We executed three trials of experiments ($E_{fair_{on}}$, $E_{fair_{off}}$ and E_{unfair}) to test WombatOAM's scalability by deploying the ACO release on a certain number of nodes, and calculating the average deployment time for each node count as described in Section 3.2.2.

The fixed parameters of these trials are summarised in Table 1. (As a reminder, WW_{Cap} is the maximum number of ACO nodes managed by one Worker Wombat, while H_{Cap} is the maximum number of ACO nodes deployed on the same host.)

	$E_{fair_{on}}$	$E_{fair_{off}}$	E_{unfair}
WW_{Cap}	150	150	150
H_{Cap}	24	24	72
WombatOAM monitoring	on	off	on

Table 1: Attributes of test configurations.

Measuring WombatOAM's effect on the monitored Erlang nodes We executed two trials of experiments to determine the cost of WombatOAM's monitoring by executing the ACO application. Just for an easier grasp, the two trials correspond to the functions $T_{E_{on}}(N_C)$ and $T_{E_{off}}(N_C)$ respectively. The first trial (E_{on}) was performed with WombatOAM monitoring disabled. In the second trial (E_{off}), WombatOAM monitoring was enabled. In both trials, we measured the time it took the ACO nodes to execute a fixed number of local and global iterations.

The fixed parameters of these trials are summarised in Table 2.

	E_{on}	E_{off}
WW_{Cap}	150	150
H_{Cap}	3	3
WombatOAM monitoring	on	off
ACO: number of ants per colony	100	100
ACO: number of global iterations	100	100
ACO: number of local iterations	100	100

Table 2: Attributes of test configurations.

3.3.2 Test results

Here, we present the results of our experiments. Each description corresponds to an experiment, which includes plot(s) to visualise the experiment, a table to list the experimental data together with some certain values corresponding to important properties of the framework, and a brief summary about our related observations.

Note that we used WolframAlpha [24] to determine the best fitting curve based on the experimental data for each trial. Both experimental data and the best fitting curve are displayed on the same figure.

Regarding tables, we present the following key properties of the system as follows.

- Each column represents data that corresponds to exactly one value, which belongs to the domain of the predictor variable. As the predictor variable always expresses the number of nodes to be deployed, it can be said that each column contains data related to a certain number of nodes, which is displayed in the head cell of the column.
- Rows whose label begin with 'T' contain the trimmed mean of measured time in seconds.
- Rows labelled by WW_C show the number of Worker Wombats used to solve the task. As the allocation strategy for Worker Wombats is to assign a separate, unused host for each Worker Wombat, WW_C also represents the number of hosts exclusively used by Worker Wombats. The determination of the sufficient number of Worker Wombats, which was presented in Section 2.1.2, is repeated here again, for the user's convenience: $WW_C = \left\lceil \frac{C_M}{WW_{Cap}} \right\rceil$.
- Rows labelled by H_C present the numbers of hosts to which the ACO Erlang nodes were deployed. It was introduced in Section 2.1.2, and it is repeated here again: $H_C = \left\lceil \frac{C_M}{H_{Cap}} \right\rceil$.
- Rows labelled by H_C^{Σ} present the total number of hosts that were used by any actor of the framework during the execution. It was also defined in Section 2.1.2. Its calculation using the definition above is the following: $H_C^{\Sigma} = WW_C + H_C + 1$. Note that 1 expresses the host used for running Meta Wombat.

Measuring deployment time of WombatOAM Regarding this experiment we defined 3 trials. The first trial presented is E_{unfair}. The goal is to determine the required time for deployment procedure in the widest possible range in terms of the number of the deployed nodes. As each host belonging to the ATHOS cluster has 24 CPU cores, we need to release the fair deployment strategy from the view point of the deployed nodes. Henceforth H_{Cap} is tripled, meaning, 3 Erlang nodes share the same CPU core in the course of this trial. The result of E_{unfair} is illustrated on Figure 6 and on Table 3.

Based on the experimental data the best fitting curve was determined using the least squares method. Its residuals were in $[-4, 4]$. The main observation is that the required time can be expressed

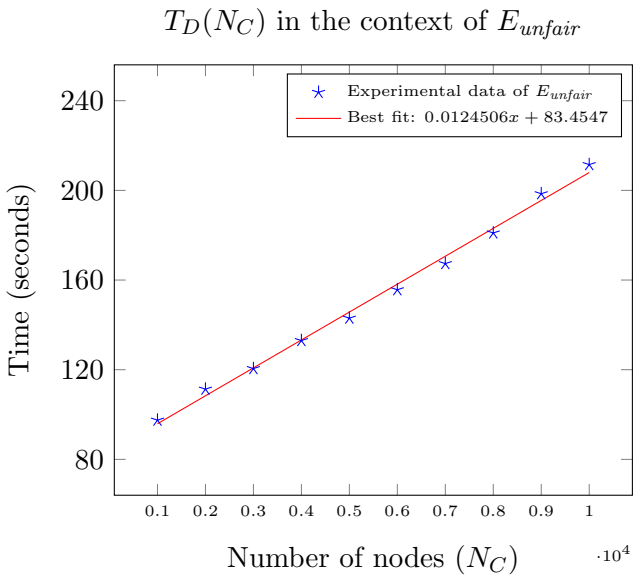

Figure 6: Deployment time.

	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
T_D (sec)	97.5	111.33	120.5	133.0	143.0	155.67	167.33	181.0	198.5	211.5
WW_C	7	14	20	27	34	40	47	54	60	67
H_C	14	28	42	56	70	84	98	112	125	139
H_C^{Σ}	22	43	63	84	105	125	146	167	186	207

Table 3: Data related to E_{unfair}.

linearly with the size of the input. The slope of the function is 0.012, which implies a great scalability – hundred of nodes can be deployed within two seconds. However, observe that there is a not so small constant (83.4547) that yields in imperfect but acceptable deployment time in terms of small tasks. Nevertheless, this is not an issue considering the aim of the RELEASE project.

Turning to the another two trials (E_{fair_*}), the goal of the trials was to determine the effect of the monitoring on the deployment procedure. Based on the determined experimental data we can observe the effect of monitoring. The experimental data and the best fitting curves corresponding to these trials are shown on Figure 7, whilst Table 4 presents the relevant attributes.

	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
$T_{D_{off}}$ (sec)	51.67	55.33	60.33	66.33	71.67	77.5	83.0	89.0	95.0	101.0
$T_{D_{on}}$ (sec)	*51.33*	56.67	61.0	66.33	*71.0*	*77.0*	83.33	89.0	*94.0*	*100.5*
WW_C	4	7	10	14	17	20	24	27	30	34
H_C	21	42	63	84	105	125	146	167	188	209
H_C^{Σ}	26	50	74	99	123	146	171	195	219	244

Table 4: Data related to $E_{fair_{off}}$ and $E_{fair_{on}}$.

Based on the experimental data the best fitting curves were determined using the least squares method. Their residuals were in $[-2, 1]$. First, observe that there are some columns in Table 4 (marked as italic) which show that the deployment time in the context of $E_{fair_{on}}$ is *less* than the deployment time

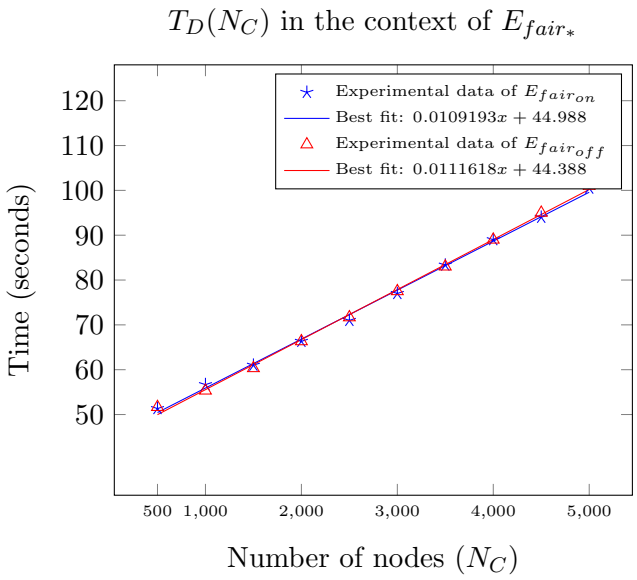

Figure 7: Effect of monitoring on deployment time.

experienced in $E_{fair_{off}}$. As WombatOAM introduces indeterminism to uniformly distribute massive load by randomisation, these unexpected patterns can be tolerated. Hence, we state that monitoring has no effect on the deployment procedure.

Measuring WombatOAM's effect on the monitored Erlang nodes In the context of this experiment we were interested in the effect of WombatOAM on the monitored Erlang nodes in terms of intrusiveness. So, two trials were performed to determine the *execution time* of the ACO algorithm in two cases: monitoring is turned (1) on and (2) off. Note that the only difference between the trials was the status of monitoring. Figure 8 illustrates the experimental data and the best fitting curves corresponding to these trials, whilst Table 4 describes the relevant attributes.

	10	30	50	70	90	110	130	150
$T_{E_{off}}$ (sec)	93.67	95.33	96.0	97.0	98.0	99.0	100.0	100.0
$T_{E_{on}}$ (sec)	93.67	95.67	97.0	97.67	99.0	99.0	101.0	101.33
Overhead (sec)	0	0.34	1	0.67	1	0	1	1.33
Overhead (%)	0	0.36	1.04	0.69	1.02	0	1	1.33
WW_C	1	1	1	1	1	1	1	1
H_C	4	10	17	24	30	37	44	50
H_C^{Σ}	6	12	19	26	32	39	46	52

Table 5: Data related to E_{on} and E_{off}.

For each column of the Table 5, we calculated how much longer the ACO algorithm takes when WombatOAM Monitoring is turned on. We calculated the overhead both in seconds ($T_{E_{on}} - T_{E_{off}}$) and in percentage ($(\frac{T_{E_{on}}}{T_{E_{off}}} - 1) \cdot 100$). As the table shows, the overhead is between 0% and 1.33%. The differences between the overheads in different executions are due to the noise in the experiments. The measured overhead is relatively small, henceforward *WombatOAM is considered to be non-intrusive.*

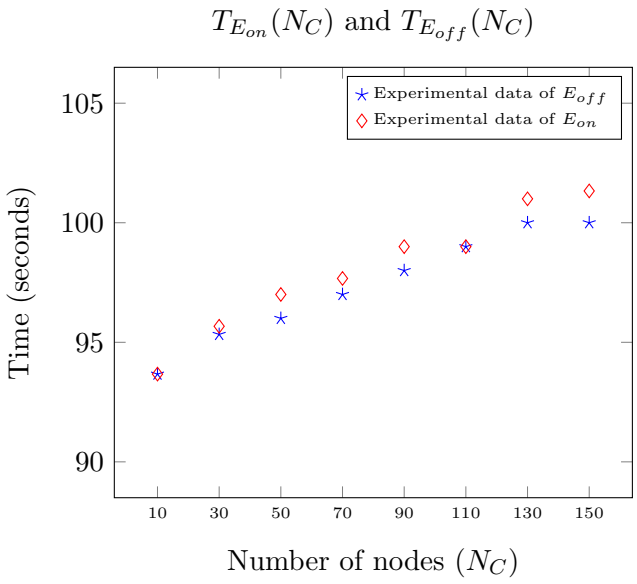

Figure 8: Execution time.

3.3.3 Evaluation

First, we wish to clarify that our goal was to create a general purpose framework, not a prototype specialised for the EDF cluster. Thus, our tool performs several actions that are unnecessary in case of EDF. For instance, WombatOAM literally copies the release to be deployed for all the Erlang nodes. This step is necessary when the environment is heterogeneous or the host machines are real servers. Nevertheless, copying the releases is worth the efforts as it needs to be done only once to support starting and stopping the Erlang nodes again whenever it is required. According to our experiences, the construction of the Wombat-tree requires a constant time. It is between 9 and 11 seconds. It implies a massively concurrent system. Nevertheless, the whole system cannot be such concurrent, because of some bottlenecks. For instance, consider the bounded throughput of the network. Moreover, we designed WombatOAM to be an enabler for high volume tasks that cannot be solved manually. For instance, to aid in executing simulations running on thousands of nodes. Comparing the manual and the automated deployment of thousands of nodes, the difference is enormous. Considering that these simulations could run for several hours (or days), the automated deployment time is infinitesimal. Therefore, WombatOAM provides a valuable support with the following characteristics.

We have measured WombatOAM's performance from different aspects, based on which we were able to assess the scalability of WombatOAM Orchestration and intrusiveness of WombatOAM Monitoring. As it might be is apparent, the observed performance of WombatOAM is outstanding. WombatOAM does have the ability to dynamically scale out (in size) in an impressive degree depending on the size of the given task. For instance, study again Figure 6. Considering that deploying 1 000 nodes required less than a 100 seconds, deploying 10 000 nodes within 212 seconds is amazing. As the shapes of all the graphs suggest linearity, the previously highlighted two observations are more dramatic.

We were also curious about the limits of WombatOAM, i.e., what is the size of the task which WombatOAM cannot accommodate. Regardless of our attempts, we were not able to determine the limit, since that would have required more resources.

We examined the overhead of the monitoring from two points of view. First we were interested in the effect of the monitoring on the deployment procedure. Here we found that the monitoring does not influence the procedure. Second we were engaged in a more serious experiment as we wished to

determine whether the performance of the managed node is significantly degraded by the monitoring. The experiment showed comforting observations, which seems to indicate that the design and implementation is sound. We found that the overhead of monitoring is relatively small (less than 1.5%), hence WombatOAM is not intrusive.

In conclusion, the evaluation process established the following characteristics of WombatOAM:

- The ability to dynamically scale out (in size).
- The deployment time is linear.
- The effect of monitoring on the managed nodes is relatively small.
- WombatOAM is non-intrusive.

4 Conclusions and Future Work

In this deliverable we presented our efforts resulting in a scalable infrastructure provided by WombatOAM, that is a dynamic scalable and reliable tool, which is able to adapt to heterogeneous clusters under realistic loads. At the beginning of the project the following project objectives were defined as the Statement of Aims for our Scalable Virtualisation Infrastructure.

> The Statements of Aims is to help build, test, deploy and operate distributed scalable systems written in Erlang. This serves the needs of development teams building scalable systems and allows companies to improve their time-to-market, quality, customer engagement and overall development flow. [14]

Now, as the end of the project is close, we are delighted to state that the developed infrastructure meets the requirements of the defined project objectives.

In this delivery we presented our related work that covers tasks that either helped us to determine WombatOAM's features (including the limits) or aided in improving various properties of the whole system. In this delivery an overview of these tasks were given together with the lessons learned. The prominent quality properties of WombatOAM were proven by different methods including the evaluation of performance measurements and the description of the workflow applied in the context of soak testing. Measurements were performed using the Athos cluster available at EDF, and the experiences were discussed and presented as the evaluation of WombatOAM.

Considering a user interface that shows information about enormously large managed systems, only the key properties of that system should be presented in order to avoid overloading the users with irrelevant details, but still to aid them in getting the information they need about their system. Our future work includes the development of such presentation techniques, which will result in an improved user interface for WombatOAM. The user will interact with the new interface that provides an aggregated, Erlang specific view of the system. This view will highlight the most important properties of the system and provide access to a detailed view of the managed nodes.

Besides these, we wish to further study and improve WombatOAM so that it becomes a mainstream tool for Erlang users dealing with various tasks. We hope that WombatOAM will be considered as a framework that is able to ease tedious tasks (e.g., the deployment of distributed systems), and to highlight anomalies in the system, thus helping to avoid abnormal terminations and outages.

Node Hooks

Section 2.3 gave a brief introduction to the new *node hooks* feature of WombatOAM. This appendix gives a more technical description of the node hooks.

When deploying a set of nodes with WombatOAM, the user may specify a set of actions that need to be executed before or after certain events happen. Each such occasion where WombatOAM is able to execute a list of custom actions is called a hook.

There are six hooks: `pre_node_start`, `post_node_start`, `pre_node_stop`, `post_node_stop`, `join` and `leave`. Their names describe when they fire, e.g., the actions assigned to the `pre_node_start` hook will be executed when the user issues WombatOAM to start a node, but before it is actually started. Both node families and nodes may have any number of actions assigned to different hooks. (But note that these are all *node hooks*, even those that are assigned to the node family, so they fire when any node of the family is started or stopped, and not when the node family itself is created or deleted.)

Starting and stopping nodes When a node is *started*, the actions are executed in the following order:

- `pre_node_start` hook actions that are assigned to the family of the node.
- `pre_node_start` hook actions that are assigned to the node.
- The node is started by executing the start command.
- `post_node_start` hook actions that are assigned to the family of the node.
- `post_node_start` hook actions that are assigned to the node.

When a node is *stopped*, the actions are executed in the following order:

- `pre_node_stop` hook actions that are assigned to the node.
- `pre_node_stop` hook actions that are assigned to the family of the node.
- The node is stopped by executing the stop command.
- `post_node_stop` hook actions that are assigned to the node.
- `post_node_stop` hook actions that are assigned to the family of the node.

When multiple nodes are deployed, the actions of different nodes are performed in parallel.

The start command and stop commands are not necessary: in that case one of the pre hooks should perform the actual starting or stopping of the node.

Nodes joining and leaving the cluster When a node *joins* a cluster, the actions are executed in the following order:

- `join` hook actions that are assigned to the family of the node.
- `join` hook actions that are assigned to the node.

When a node *leaves* the cluster, the actions are executed in the following order:

- `leave` hook actions that are assigned to the node.
- `leave` hook actions that are assigned to the family of the node.

Note that there is no *join command* or *leave command*, so these hooks are themselves responsible for asking the nodes to join/leave the cluster, and return only when that operation has finished.

The user may specify the *autojoin* option when deploying the nodes. In that case, when the first node of the node family is started, it will be appointed as the *bootstrap* node, and it will be declared

a part of the cluster without executing anything. For each successive node that is started, the *join* is performed towards the bootstrap node (i.e., the join hooks are executed with the `bootstrap_node_id` and `bootstrap_node_name` variables set accordingly). The *join* operations are performed sequentially (i.e., join for a node is started only if the join for the previous node has been finished).

Actions There are three kinds of actions: `run`, `call` and `send_file`. If the action shall be executed on a node, and the node is not (yet) available, then the action is retried (10 times by default, after waiting a randomized number seconds between retries).

Run action This action runs a program (either a script or an executable binary). The following options are available:

- `executable` (mandatory): The path to the file that should be executed. If the script is executed on WombatOAM, the path shall be absolute. If it is executed on the node, this path can be either absolute or relative to the release directory (`$HOME/<node_id>/`).
- `location` (mandatory): Whether the script should be executed on the WombatOAM host or the managed node's host.
- `args` (optional): The arguments to be passed to the executable. Each argument can be either a string (which is passed unmodified), the variable `node_id` or `node_name` (whose value is passed) or a concatenation of strings and variables. In case of the *join* and *leave* hooks, the variables `bootstrap_node_id` and `bootstrap_node_name` can also be used.

Running a script on a remote node is performed using SSH. As an example, if the following action is executed on node with id `d010c157-dd36-4990-b894-eeccce873c09` and name `mynode@127.0.0.1`:

```
{"action": "run",
 "executable": "script.sh",
 "location": "node",
 "args": ["node started",
          {"var": "node_id"},
          "--log-dir",
          {"concat": [{"var": "node_id"},
                      "/log/",
                      {"var": "node_name"}]}]}
```

then the following will be executed:

```
script.sh "node started" d010c157-dd36-4990-b894-eeccce873c09 \
          --log-dir d010c157-dd36-4990-b894-eeccce873c09/log/mynode@127.0.0.1
```

The return code of the script is checked and if it is not zero, the execution is considered to be failed, further actions on the node are cancelled, and the error is reported.

Call action This action calls an Erlang function. The following options are available:

- `mod_name`, `fun_name` (mandatory): The module and name of the function to be called.
- `location` (mandatory): Whether the function should be called on the WombatOAM node or the managed node.
- `args` (optional): The arguments to be passed to the function. Each argument can be either a term (which is passed unmodified after being parsed from the JSON string) or the variable `node_id` or `node_name` (whose value is passed). In case of the *join* and *leave* hooks, the variables `bootstrap_node_id` and `bootstrap_node_name` can also be used.

Calling a function on a remote node is performed using Erlang RPC. As an example, if the following action is executed on node with id `d010c157-dd36-4990-b894-eeccce873c09` and name `mynode@127.0.0.1`:

```
{"action": "call",
 "mod_name": "mymod",
 "fun_name": myfun",
 "location": "node",
 "args": [{"term": "node_start"},
          {"term": "{1, [2,3]}"}
          {"var": "node_name"},
          {"var": "node_id"}]}
```

then the following function call will be executed:

```
mymod:myfun(node_start,
            {1, [2, 3]},
            'mynode@127.0.0.1',
            "d010c157-dd36-4990-b894-eeccce873c09").
```

In order to execute an Erlang function on WombatOAM, WombatOAM needs to find the containing module. To do that, the directory that contains the compiled version of the module shall be added to the `"wo_orch/hook_dirs"` option in WombatOAM's configuration file. For example:

```
{wo_orch, [
           {hook_dirs, ["/home/user/my_wombat_hook_dir"]}
          ]}
```

In order to execute an Erlang function on the deployed node, the creator of the release package needs to make sure that the node will find the module.

Since Erlang functions on the node can be executed only when the node is running, *call* actions on nodes should not be specified in `pre_node_start` and `post_node_stop` hooks.

Send file action This action copies a file from WombatOAM to the node's host machine. The following options are available:

- `source` (mandatory): The path to the file to be copied on the WombatOAM host. It is recommended to use an absolute path.
- `target` (mandatory): The path that the file shall have on the target machine, or the directory into which it should be copied. It can be either relative or absolute. In the latter case, it is relative to the node's main directory (which is `$HOME/<node_id>`).

Copying a file to the node is performed using SCP.

Specifying the hooks and actions to WombatOAM The exact structure of the `NodeHooks` JSON object that can be included in the REST request body when specifying node hooks is the following:

```
NodeHooks =
{
  // All fields are optional
  "pre_node_start": [HookAction],
  "post_node_start": [HookAction],
  "pre_node_stop": [HookAction],
```

```
  "post_node_stop": [HookAction],
  "join": [HookAction],
  "leave": [HookAction]
}

HookAction =
    RunHookAction OR   // Hook that runs a script
    CallHookAction OR  // Hook that calls an Erlang function
    SendFileHookAction // Hook that copies a file to the node

RunHookAction =
{
  // Mandatory fields
  "action": "run",
  "executable": "PathToScript",
  "location": "wombat" OR "node",

  // Optional fields
  "args": [RunArg]
}

RunArg =
    "ArgString" OR
    {"var": "node_id"} OR
    {"var": "node_name"} OR
    {"concat": [RunArg]}

CallHookAction =
{
  // Mandatory fields
  "action": "call",
  "mod_name": "ModuleName",
  "fun_name": "FunctionName",
  "location": "wombat" OR "node",

  // Optional fields
  "args": [CallArg]
}

CallArg =
  // Note that a term shall be specified as a JSON string, not as a JSON
  // object. E.g. "[1, 2]" as opposed to simply [1, 2].
  "ErlangTerm" OR

  {"var": "node_id"} OR
  {"var": "node_name"}

SendFileHookAction =
{
  "action": "send_file",
  "source": "PathToSourceFileOrDir",
  "target": "PathToTargetFileOrDir"
}
```

References

[1] Basho Technologies. Basho Bench, Januar 2015. https://github.com/basho/basho_bench.

[2] Francesco Cesarini and Simon Thompson. *ERLANG Programming*. O'Reilly Media, Inc., 1st edition, 2009.

[3] Natalia Chechina, Huiqing Li, Amir Ghaffari, Simon Thompson, and Phil Trinder. Improving the Network Scalability of Erlang. *Submitted to Journal of Parallel and Distributed Computing*, December 2014.

[4] Daniel de Oliveira, FernandaAraujo Baio, and Marta Mattoso. Towards a Taxonomy for Cloud Computing from an e-Science Perspective. In Nick Antonopoulos and Lee Gillam, editors, *Cloud Computing*, Computer Communications and Networks, pages 47–62. Springer London, 2010.

[5] Michael J de Smith. *STATSREF: Statistical Analysis Handbook – a web-based statistics resource.* 2010 – 2015.

[6] Ericsson AB. Erlang Emulator. http://erlang.org/doc/man/erl.html.

[7] Ericsson AB. Heartbeat Monitoring of an Erlang Runtime System. http://www.erlang.org/doc/man/heart.html.

[8] European Commission – CORDIS. PROWESS – Property-based testing of Web services. http://cordis.europa.eu/project/rcn/105389_en.html.

[9] Apache Software Foundation. *Liblcoud*. https://libcloud.apache.org/.

[10] Wajdi Louati, Ines Houidi, Manel Kharrat, Djamal Zeghlache, and Hormuzd M. Khosravi. Dynamic service deployment in a distributed heterogeneous cluster based router (DHCR). *Cluster Computing*, 11(4):355–372, 2008.

[11] J.L. Lucas Simarro, R. Moreno-Vozmediano, R.S. Montero, and I.M. Llorente. Dynamic Placement of Virtual Machines for Cost Optimization in Multi-Cloud Environments. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 1–7, July 2011.

[12] Haakan Mattsson, Hans Nilsson, and Claes Wikstrom. Mnesia - A Distributed Robust DBMS for Telecommunications Applications. In *PADL '99: Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 152–163. Springer-Verlag, 1998.

[13] Ian Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, Inc., 1st edition, 2009.

[14] RELEASE Project. Deliverable D4.2: Homogenous super-cluster infrastructure, July 2012. http://release-project.softlab.ntua.gr/documents/D4.2.pdf.

[15] RELEASE Project. Deliverable D4.3: Heterogeneous super-cluster infrastructure, December 2012. http://release-project.softlab.ntua.gr/documents/D4.3.pdf.

[16] RELEASE Project. Deliverable D3.2: Scalable SD Erlang Computation Model, August 2013.

[17] RELEASE Project. Deliverable D4.4: Capability-driven Deployment, May 2013.

[18] RELEASE Project. Deliverable D6.5: Heterogeneous Deployment of a Load-testing Tool, October 2013.

[19] RELEASE Project. Deliverable D6.6: Capability-matching Heterogeneous deployment of a Load Testing Tool, October 2014.

[20] RELEASE Project. Scalable Reliable OTP Library Release: Patterns for Scaling Distributed Erlang Applications, September 2014.
http://release-project.softlab.ntua.gr/documents/D3.4.pdf.

[21] S. Santhosh, M. Vaden, and B. Fernandes. Method and apparatus for load testing online server systems, June 24 2014. US Patent 8,762,113.

[22] SLURM Team. SLURM.
https://computing.llnl.gov/linux/slurm/team.html.

[23] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[24] Wolfram Alpha LLC – A Wolfram Research Company. Wolfram Alpha: Computational Knowledge Engine.
http://www.wolframalpha.com.