Final Report

Authors:
Reinier Hartog
Richard van Heest
Vincent Hellendoorn
Anand Sawant

Delft University of Technology:
Dr. ir. Alexandru Iosup

COAS Software Systems:
Janco Tanis
Gerard Brussé

July 8, 2013
Preface

Early 2013, we, Reinier Hartog, Richard van Heest, Vincent Hellendoorn en Anand Sawant, have started setting up a Bachelors final project at Coas Software Systems, following their request to one of the team members. In the months April to July of 2013, we have worked on the project described in this document. As of this version, this system functions correctly and is nearly done. Research is included in this report, but has yet to be finalized.

We would like to thank those involved in this project, namely: Alexandru Iosup, our TU Delft supervisor for providing us with excellent feedback, Janco Tanis, the company supervisor for helping us establish requirements and steering us in the implementation of the system, Gert Gross, the Bachelor thesis coordinator and Martha Larson, the new Bachelor thesis coordinator.

July 6, 2013

Reinier Hartog - R.M.Hartog@student.tudelft.nl
Richard van Heest - A.W.M.vanHeest@student.tudelft.nl
Vincent Hellendoorn - V.J.Hellendoorn@student.tudelft.nl
Anand Sawant - A.A.Sawant@student.tudelft.nl
Abstract

For companies that deliver hardware services, the ability to scale dynamically has many advantages. In a special case, a company may have some computing resources available and desire to scale out when traffic peaks. A periodical budget may then be allocated, which needs to be used efficiently. This can be achieved through a combination of scheduling and provisioning policies. This report details the implementation of a system that functions in this environment. It replaces a previous system and must fulfill the tasks of handling and scheduling jobs to available nodes, provisioning nodes and providing information regarding system behavior to the user. A requirements analysis and an architectural design plan were written before implementing the system. Test driven development was applied where possible, and the test and implementation details are described further in this report. Research was conducted in scheduling and provisioning policies in the context of the problem described above.
Contents

1 Introduction ...................................................... 1
1.1 Problem description ........................................... 1
1.2 Project assignment ........................................... 1
1.3 Outline ......................................................... 1
2 Requirement Analysis ........................................... 2
2.1 Introduction .................................................... 2
2.2 Background information ....................................... 2
2.3 Existing System ................................................ 2
2.4 Problem statement ............................................. 3
  2.4.1 Purpose ...................................................... 3
  2.4.2 Project scope ............................................... 3
2.5 Modifications to existing system ............................... 4
2.6 System features ............................................... 4
  2.6.1 System feature 1: Submit a job .......................... 6
  2.6.2 System feature 2: Receive job ........................... 7
  2.6.3 System feature 3: Preprocessing of jobs ................ 8
  2.6.4 System feature 4: Retrieval of job data ................. 9
  2.6.5 System feature 5: Splitting jobs in tasks ............... 10
  2.6.6 System feature 6: Enqueue a task ...................... 11
  2.6.7 System feature 7: Retrieve task from task queue ....... 12
  2.6.8 System feature 8: Prioritization of tasks ............. 13
  2.6.9 System feature 9: Predicting runtime of task .......... 14
  2.6.10 System feature 10: Scheduling of tasks ............... 15
  2.6.11 System feature 11: Allocation of tasks to nodes ..... 16
  2.6.12 System feature 12: Gathering of statistics ............ 17
  2.6.13 System feature 13: Processing statistical data ....... 18
  2.6.14 System feature 14: Provisioning of nodes .......... 19
  2.6.15 System feature 15: Suspend a node ................. 20
  2.6.16 System feature 16: Resume a node ................... 21
  2.6.17 System feature 17: Removing a node ................. 22
  2.6.18 System feature 18: Deployment of nodes .......... 23
  2.6.19 System feature 19: Query job status ............... 24
  2.6.20 System feature 20: Retrieve job results ............. 25
  2.6.21 System feature 21: Monitoring of data ............. 26
  2.6.22 System feature 22: Garbage collect ................. 27
  2.6.23 System feature 23: Delete job result ............... 28
  2.6.24 System feature 24: Setting of system variables ...... 29
  2.6.25 System feature 25: Update node software .......... 30
2.7 Non-functional requirements ................................. 31
  2.7.1 Performance requirements .............................. 31
  2.7.2 Safety requirements .................................... 31
  2.7.3 Security requirements .................................. 31
  2.7.4 Platform requirements .................................. 31
3 System Design .................................................. 32
3.1 Architectural principles ...................................... 32
  3.1.1 Design Principles ....................................... 32
  3.1.2 Design Patterns ......................................... 33
  3.1.3 Coding practices ....................................... 34
  3.1.4 Frameworks, tools and libraries ....................... 35
3.2 Framework design and implementation ........................ 37
  3.2.1 Logical view .............................................. 37
  3.2.2 Interface view .......................................... 38
  3.2.3 Design view ............................................. 38
  3.2.4 Infrastructure view ..................................... 39
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1.8 Provisioning sequence</td>
<td>67</td>
</tr>
<tr>
<td>B.1.9 Deployment of node sequence</td>
<td>68</td>
</tr>
<tr>
<td>B.1.10 Job status sequence</td>
<td>69</td>
</tr>
<tr>
<td>B.1.11 Retrieve result sequence</td>
<td>70</td>
</tr>
<tr>
<td>B.1.12 Job monitoring sequence</td>
<td>71</td>
</tr>
<tr>
<td>B.1.13 Garbage collection sequence</td>
<td>72</td>
</tr>
<tr>
<td>B.1.14 Setting variables sequence</td>
<td>73</td>
</tr>
<tr>
<td>B.2 Interface View</td>
<td>74</td>
</tr>
<tr>
<td>B.3 Design View</td>
<td>75</td>
</tr>
<tr>
<td>B.3.1 Frontend Class Diagram</td>
<td>75</td>
</tr>
<tr>
<td>B.3.2 Scheduler Class Diagram</td>
<td>76</td>
</tr>
<tr>
<td>B.3.3 Managers Class Diagram</td>
<td>77</td>
</tr>
<tr>
<td>B.3.4 Node Class Diagram</td>
<td>78</td>
</tr>
<tr>
<td>B.3.5 Job Class Diagram</td>
<td>79</td>
</tr>
<tr>
<td>B.3.6 Commons Class Diagram</td>
<td>80</td>
</tr>
<tr>
<td>B.3.7 Garbage Collection Class Diagram</td>
<td>81</td>
</tr>
<tr>
<td>B.4 Infrastructure View</td>
<td>82</td>
</tr>
<tr>
<td>B.5 Logical View</td>
<td>83</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Problem description

Businesses that supply computing services often face irregular and strongly increasing workloads on their systems. The traditional solution of acquiring sufficient computing resources is impractical. It is both expensive, as the resources must be sufficient to handle peaks in traffic, as well as unscalable, requiring the installation of new units as overall traffic increases. For all but the largest businesses, it is also unnecessary due to the rise of cloud computing.

Cloud computing services, such as Infrastructure as a Service, offer the possibility of rapidly scaling to any number of machines, while paying by the hour or even by the minute. These services typically provide the possibility of automatic scaling. A special case arises when a company both owns computing resources and has a periodical budget available for scaling into the cloud. The problem at hand is scheduling of jobs and provisioning of nodes to effectively utilize the budget.

1.2 Project assignment

For this project, a framework has been created that will function within a job handling system. It replaces the previous framework, which used just a grid computing resource whose nodes had remote data access. The replacing framework must fulfill the task of handling jobs (or workflows) correctly, while allowing for scaling into the cloud (cloud bursting) and removing the connection of nodes to the database in favour of a headnode performing this communication.

In order to achieve these goals, a framework named Jackal was created that consists of four subsystems which can each run independently. These subsystems together make up the headnode, although they may run off multiple nodes. The framework has a frontend subsystem that handles incoming jobs and retrieves their status and result for the client. A graphing package is provided to allow live insight into the system load and other required variables. The job monitor subsystem serves as the link between the frontend and the rest of the system, handling all job related requests. The scheduler subsystem splits a job into tasks, allocates these to available nodes and recombines the job from the task results. The manager subsystem handles provisioning, deployment and all other node related affairs. Finally, a portable jar runs from each node to allow it to run tasks.

1.3 Outline

This report will proceed to detail the requirements imposed on the system, including an analysis of the problem and project, in chapter two. Chapter three elaborates on the architectural design of the system, including the applied design principles, design patterns and the used frameworks, tools and libraries. Chapter four sets forth the implementation details of several main components in the system. This includes an analysis of the two used cloud services in the system, EC2 and Azure, details regarding the connection to Cassandra and Condor, and the implementation of JMS bridging. It also details the used of algorithms for task runtime prediction and for prioritization based on pre-assigned priorities and aging.

Budget-constrained provisioning and scheduling were topics for research, the result of which can be found in chapter five. After the conclusion and bibliography, the appendices can be found, containing the orientation report and the diagrams depicting the initial architectural design of the system.
2 Requirement Analysis

2.1 Introduction

This section provides the initial description of the framework and the requirements imposed on it. It is followed by the architectural design, which elucidates the design of the software to be implemented, the test plan and the implementation plan, which provides an outset of the planned activities in the implementation period and the validation measures that will be employed.

The structure of this document is based on the format as described in the book ‘Object-Oriented Software Engineering’ by Lethridge and Laganière [1].

2.2 Background information

This project operates in the distributed system paradigm. Tanenbaum et al. define a distributed system as “a collection of independent computers that appears to its users as a single coherent system” [2]. A distributed system can therefore perform tasks with a time and/or cost efficiency beyond the reach of a single machine.

The system in this project consists of a grid of distributively owned computing resources, a grid, and the capacity to scale out to a commercial cloud service. The main difference between grid computing and cloud computing is the availability, where the grid is a set of company-owned computers and the cloud allows for dynamically renting a computing resource.

More specifically, grid computing is defined as “A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities” by Ian Foster and Carl Kesselman [3]. Amazon defines the term “cloud computing” as “the on-demand delivery of IT resources via the Internet with pay-as-you-go pricing” [4]. In this project the main distinction made between grids and clouds is that the grid resources are a fixed resource and the cloud is a scalable resource.

2.3 Existing System

COAS currently employs a grid-based system, in which there is a single head node and multiple worker nodes. The head-node receives job IDs and allocates these to the available worker nodes. All nodes have a direct link to the database and look up the job data corresponding to the given job ID. All communication between the head node and the worker nodes is done with the help of JMS. The workload management system employed is HTCondor. A global overview of the system can be seen in Figure 1.

![Fig. 1: An overview of the current system architecture.](image-url)
2.4 Problem statement

2.4.1 Purpose

The main problem can be stated as follows:

“to build a framework that can efficiently and cost-effectively schedule workloads in a grid-based system, but also has the ability to scale out to a cloud environment when more resources are required”.

This problem can be broken down into multiple subproblems:

1. Job runtime prediction
   (a) for different types of jobs
   (b) on different types of nodes
2. Cost-based scheduling and provisioning
   (a) with a maximum budget for a fixed time period
   (b) considering software licenses
   (c) providing insight to the end-user
3. Node deployment
   (a) in an automated fashion
   (b) for different versions of the software side by side
4. Secure communications and storage

In a heterogeneous environment, consisting of different types of nodes and processing different types of jobs, job runtimes can be influenced by the complexity of the job, the performance of the node it is assigned to, the load of the database and the load of the network. However, having information about job runtimes is needed for effective scheduling [5]. Furthermore, it is useful for end-users to have feedback on their actions, together with an estimate of the time they will have to wait.

While a cloud environment provides the ability to provision resources on demand, the question is whether this is economically viable. The objective is to schedule workloads and provision resources in such a way that financial constraints are not violated. The problem of optimally utilizing a budget in a certain time period requires accurate estimation and prediction of costs for the remainder of this period. Furthermore, constraints imposed on job runtimes and the usage of other resources, such as software licenses, should be respected. Also, insight into the scheduling choices, the provisioning decisions and the data they are based on is required.

To allow for smooth scaling into a cloud environment and to make maintenance of a large amount of nodes feasible, the company requires the possibility of automatic deployment of the software to these nodes. This deployment system should take into account the different versions of the software and perform automatic configuration.

Because the data that will be processed by the system may be sensitive, the communication channels, especially those travelling outside of the private network, have to be secure. Furthermore, all storage of this data has to be secured and should not be accessible without authorization.

2.4.2 Project scope

In implementing this framework, we have to limit our scope to the aforementioned (sub)problems. To narrow our scope even further, we impose the following constraints on our development:

1. The existing system will not be modified, but this framework will be designed to be integrated with the existing system.
2. Testing of this framework will be limited to the use of simulated workloads.
3. The structure of the current databases will not be altered.
4. Technologies that are currently being used, such as HTCondor[6] and JMS (Java Messaging Service)[7], will also be used by this framework where appropriate.
2.5 Modifications to existing system

Three main alterations will be made to this system. The direct connection between the worker nodes and the database will be removed, instead the head node will now be the one retrieving data from the database. The head node will distribute jobs and job data to the workers and will no longer be a singular static node. Scalability functionality will be added using a commercial cloud service. An overview of the intended system can be seen in Figure 2.

![Diagram](image)

Fig. 2: An overview of the proposed system architecture.

2.6 System features

Figure 3 depicts the various use cases and their interplay. The two actors on the left are the administrators and the user. The administrator has the capabilities to update the software running on a worker node, set the preferences of the system, and to monitor the data in the system. The user can submit a job, check its status and retrieve or delete its result. The two actors on the right are the framework and the cloud service. The cloud service is contacted in order to provision new nodes and to suspend or delete nodes. The framework handles the influx of jobs, preprocesses these jobs, allocates these jobs and gathers statistics based on the execution of these jobs. All the different use cases seen here in the diagram are reflected in system features as follows.
Fig. 3: The global use-case diagram.
2.6.1 System feature 1: Submit a job

Functional requirement
The user must be able to submit a job and its related job data.

Priority
High

Inputs
• Job ID

Outputs
• None.

Use case

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>Use case name</th>
<th>Description</th>
<th>Preconditions</th>
<th>Postconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Submit job</td>
<td>User submits a job</td>
<td>- None.</td>
<td>- The new job is added to the queue of the framework.</td>
</tr>
</tbody>
</table>

Actors
End-user
Framework

Normal flow
1. End-user stores job data and sends job ID
2. Framework executes use case 2.

Frequency of occurrence
Once per job request.

Additional information
Parameters such as job deadline and budget that can be used to execute the job can be set by the user. In a first release budget capabilities will be supported.

Related features
System feature 2.6.2
2.6.2 System feature 2: Receive job

Functional requirement
The framework must be able to receive a job, retrieve the data required for execution and split it up into executable tasks.

Priority
High

Inputs
• A job

Outputs
• A set of executable tasks

Use case

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Receive job</td>
</tr>
<tr>
<td>Description</td>
<td>User submits a job</td>
</tr>
<tr>
<td>Preconditions</td>
<td>Job has been submitted by a user.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>A set of executable tasks has been added to the framework queue.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actors</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>End-user</td>
<td></td>
</tr>
<tr>
<td>Normal flow</td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td>A job submit request is received.</td>
</tr>
<tr>
<td>2.</td>
<td>Framework executes use case 3 to preprocess the job.</td>
</tr>
</tbody>
</table>

Frequency of occurrence
Once per job request.

Related features
System feature 2.6.3
2.6.3 System feature 3: Preprocessing of jobs

Functional requirement
The framework must be able to preprocess a job to split it into tasks and queue these tasks.

Priority
High

Inputs
• A job ID

Outputs
• A set of executable tasks.

Use case

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Preprocess jobs</td>
</tr>
<tr>
<td>Description</td>
<td>The framework preprocesses a job.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>- Job ID is valid.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>- The system has a set of executable tasks corresponding to the job.</td>
</tr>
</tbody>
</table>

Actors
Framework

Normal flow
1. Execute use-case 4 to retrieve job data.
2. Execute use-case 5 to split job into tasks.
3. Execute use-case 6 to enqueue the tasks.

Alternative flow 1
1b. Use case 4 fails.
2b. Framework rejects the job and returns an error.

Frequency of occurrence
Once per job.

Related features
System feature 2.6.4, 2.6.5 and 2.6.6
2.6.4 System feature 4: Retrieval of job data

Functional requirement
The framework must be able to retrieve the data corresponding to a job, given its ID.

Priority
High

Inputs
• Job ID

Outputs
• Job data

Use case

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Retrieve job data</td>
</tr>
<tr>
<td>Description</td>
<td>The framework retrieves data for a job.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>- The job ID is known.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>- The job data is retrieved.</td>
</tr>
<tr>
<td>Actors</td>
<td>Framework</td>
</tr>
</tbody>
</table>

Normal flow

1. Framework validates job ID.
2. Framework retrieves job data.
3. Framework validates job data.

Alternative flow 1

1b. Job ID is invalid.

Alternative flow 2

2c. Job data cannot be retrieved.
3c. Retry retrieval, if succeeds continue at 3, if fails return an error.

Alternative flow 3

3d. Job data is invalid.

Frequency of occurrence
Once per job request.

Additional information
Job data is submitted though the frontend and is then passed on to the scheduling system. In the interests of extensibility job type can be specified and this can be interpreted and pre processed by use of a standard interface.

Related features
System feature 2.6.2
2.6.5 System feature 5: Splitting jobs in tasks

Functional requirement
The framework must be able to convert a job into an equivalent set of executable tasks, which can be a set containing just the job.

Priority
Medium

Inputs
• Job data

Outputs
• A set of executable tasks.

Use case
<table>
<thead>
<tr>
<th>Use case ID</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Split jobs in tasks</td>
</tr>
<tr>
<td>Description</td>
<td>The framework splits a job into a set of tasks.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>- The job data is valid.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>- The job is split into a set of tasks.</td>
</tr>
</tbody>
</table>

Actors
Framework

Normal flow
1. Framework finds best split of job data.
2. Framework validates equivalence of tasks to job.

Alternative flow 1
2b. Set of tasks is not equivalent to job.
3b. Framework returns a set with the job as the only element.

Frequency of occurrence
Once per job request.

Related features
System feature 2.6.2
2.6.6 System feature 6: Enqueue a task

Functional requirement
The framework must be able to queue any valid task.

Priority
High

Inputs
• A task.

Outputs
• None.

Use case

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Enqueue a task</td>
</tr>
<tr>
<td>Description</td>
<td>The framework enqueues tasks.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>- The task is valid.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>- The task is in the task queue.</td>
</tr>
<tr>
<td>Actors</td>
<td>Framework</td>
</tr>
<tr>
<td>Normal flow</td>
<td>1. The framework enqueues a task in the task queue.</td>
</tr>
<tr>
<td>Frequency of occurrence</td>
<td>Once per task.</td>
</tr>
</tbody>
</table>

Related features
System feature 2.6.3 and 2.6.7
2.6.7 System feature 7: Retrieve task from task queue

Functional requirement
The framework must be able to retrieve a task from the queue.

Priority
High

Inputs
• None.

Outputs
• A task.

Use case

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Retrieve task from task queue.</td>
</tr>
<tr>
<td>Description</td>
<td>The framework retrieves a task from the task queue.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>- The task queue is not empty.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>- The first item is removed from the task queue.</td>
</tr>
<tr>
<td>Actors</td>
<td>Framework</td>
</tr>
<tr>
<td>Normal flow</td>
<td>1. The framework retrieves the first element from the task queue.</td>
</tr>
<tr>
<td>Frequency of occurrence</td>
<td>Once per queued task.</td>
</tr>
</tbody>
</table>

Related features
System feature 2.6.6 and 2.6.10
### 2.6.8 System feature 8: Prioritization of tasks

**Functional requirement**
The framework must be able to preprocess jobs by retrieving the corresponding job data and splitting the job into tasks.

**Priority**
Medium

**Inputs**
- A set of tasks.

**Outputs**
- An ordering of tasks by priority.

**Use case**

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use case name</strong></td>
<td>Prioritize tasks</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>The framework processes tasks for prioritization.</td>
</tr>
<tr>
<td><strong>Preconditions</strong></td>
<td>- The queue is not empty.</td>
</tr>
<tr>
<td><strong>Postconditions</strong></td>
<td>- The queue is empty. - All tasks are ordered by priority.</td>
</tr>
<tr>
<td><strong>Actors</strong></td>
<td>Framework</td>
</tr>
<tr>
<td><strong>Normal flow</strong></td>
<td>1. Execute use-case 7 to dequeue a task.</td>
</tr>
<tr>
<td></td>
<td>2. Execute use-case 9 to acquire runtime prediction for task.</td>
</tr>
<tr>
<td></td>
<td>3. Assign a priority to the task.</td>
</tr>
<tr>
<td></td>
<td>4. While the queue is not empty, go to 1.</td>
</tr>
<tr>
<td><strong>Frequency of occurrence</strong></td>
<td>Continuously.</td>
</tr>
</tbody>
</table>

**Additional information**

All jobs get a pre assigned priority from the user. Furthermore aging is introduced in order to increase the priority of old jobs to prevent live lock. A prioritization policy can be set, for extensibility an interface is designed so that new policies can be defined.

**Related features**
System feature 2.6.9 and 2.6.10
2.6.9 System feature 9: Predicting runtime of task

Functional requirement
The framework must be able to predict the runtime of a task given the current state of the system.

Priority
High

Inputs
- A task.
- Information on the system state.

Outputs
- Runtime of the task.

Use case

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Predicting runtime of task.</td>
</tr>
<tr>
<td>Description</td>
<td>The potential runtime of a task is determined.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>- A valid task is submitted.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>- The runtime of the task is predicted.</td>
</tr>
<tr>
<td>Actors</td>
<td>Framework</td>
</tr>
<tr>
<td>Normal flow</td>
<td>1. The framework receives the task.</td>
</tr>
<tr>
<td></td>
<td>2. The predicting policy runs on the task and the response is returned.</td>
</tr>
<tr>
<td>Assumptions</td>
<td>A prediction policy is defined.</td>
</tr>
<tr>
<td>Frequency of occurrence</td>
<td>Continuously.</td>
</tr>
</tbody>
</table>

Additional information
Task runtime prediction will be based on previous runtimes of the same task, using linear regression an average task runtime is found to be the prediction.

Related features
System feature 2.6.10, 2.6.12
2.6.10 System feature 10: Scheduling of tasks

Functional requirement
The framework must be able to schedule job based on priority and available nodes.

Priority
High

Inputs
- A set of tasks.

Outputs
- A scheduling of each task for a node.

Use case

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Scheduling of tasks.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use case name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The framework schedules tasks.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Preconditions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>- The queue is not empty.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Postconditions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>- All tasks are scheduled for execution.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actors</th>
<th>Framework</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Normal flow</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Execute use-case 8 to prioritize tasks.</td>
<td></td>
</tr>
<tr>
<td>2. Execute use-case 9 to acquire predictions.</td>
<td></td>
</tr>
<tr>
<td>3. Execute use-case 11.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Frequency of occurrence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuously.</td>
<td></td>
</tr>
</tbody>
</table>

Related features
System feature 2.6.8, 2.6.9 and 2.6.11
2.6.11 System feature 11: Allocation of tasks to nodes

Functional requirement
The framework must be able to allocate tasks to the available worker nodes.

Priority
High

Inputs
- A list of available nodes.

Outputs
- A schedule of tasks.

Use case

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Allocation of tasks to nodes</td>
</tr>
<tr>
<td>Description</td>
<td>Tasks are allocated to nodes.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>- The tasks are prioritized.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>- The tasks are allocated to nodes and timeslots.</td>
</tr>
<tr>
<td>Actors</td>
<td>Framework</td>
</tr>
</tbody>
</table>

Normal flow
1. A set of tasks is taken from the queue.
2. Using the defined allocation policy, the tasks are allocated to a node.

Assumptions
An allocation policy is defined.

Frequency of occurrence
Continuously.

Related features
System feature 2.6.10
2.6.12 System feature 12: Gathering of statistics

Functional requirement
The framework must be able to gather the statistics about the performance of the system.

Priority
Low

Inputs
• None.

Outputs
• Statistics of the job.

Use case

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>Use case name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Gathering of statistics</td>
<td>The framework collects statistics about the performance of the system.</td>
</tr>
</tbody>
</table>

Preconditions  - None.

Postconditions - None.

Actors

<table>
<thead>
<tr>
<th>Framework</th>
<th>The node</th>
</tr>
</thead>
</table>

Normal flow
1. The framework retrieves data from the scheduler and the workers.

Frequency of occurrence
Continuously.

Additional information
Statistics of job runtime, task runtime, system load and provisioning cost will be gathered. These statistics will be stored in a database for their retrieval. Also for extensibility JMX (Java Management Extensions) will be made available for detailed monitoring.

Related features
System feature [2.6.9] [2.6.13] and [2.6.14]
### 2.6.13 System feature 13: Processing statistical data

**Functional requirement**
The framework must be able to process all statistical data that has been collected.

**Priority**
Medium

**Inputs**
- Unprocessed statistical data.

**Outputs**
- Processed statistical data.

**Use case**

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Processing statistical data.</td>
</tr>
<tr>
<td>Description</td>
<td>The framework processes statistical data.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>- The unprocessed data has been collected.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>- None.</td>
</tr>
<tr>
<td>Actors</td>
<td>Framework</td>
</tr>
<tr>
<td>Normal flow</td>
<td>1. The framework runs the statistical data processing policy.</td>
</tr>
<tr>
<td>Assumptions</td>
<td>A statistical data processing policy is defined.</td>
</tr>
<tr>
<td>Frequency of occurrence</td>
<td>Continuously.</td>
</tr>
</tbody>
</table>

**Related features**
System feature 2.6.12
2.6.14 System feature 14: Provisioning of nodes

Functional requirement
The framework must be able to provision nodes when required, based on the defined provisioning policy.

Priority
Medium

Inputs
- Current state.
- Historical trends.
- Maximum budget.

Outputs
- None.

Use case

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Provisioning of nodes</td>
</tr>
<tr>
<td>Description</td>
<td>The framework provisions nodes when required.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>- None.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>- Sufficient nodes have been provisioned.</td>
</tr>
<tr>
<td>Actors</td>
<td>Framework, Cloud service</td>
</tr>
</tbody>
</table>

Normal flow
1. Framework uses provisioning policy to determine whether additional nodes are needed.
2. The framework sends a request for a new node to the cloud service.
3. The cloud service provides a new node and returns the node details.
4. The framework receives the node details.
5. Execute use case 18 to deploy software on the node.
6. The node is added to the list of workers.

Alternative flow 1
1b. No additional nodes are needed.
2b. Go to 1.

Alternative flow 2
4c. A new node can not be provided by the cloud service.
5c. Retry the request, if succeeds go to 5 else if fails go to 1.

Alternative flow 3
5d. Use case 18 fails.
6d. Execute use case 17 to remove the node.
7d. Go to 1.

Assumptions
A provisioning policy is defined.

Frequency of occurrence
Continuously.

Related features
System feature 2.6.17 and 2.6.18
2.6.15 System feature 15: Suspend a node

Functional requirement
The framework must be able to suspend the assignment of tasks to a node.

Priority
Medium

Inputs
- A node ID.

Outputs
- None.

Use case

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Suspend a node</td>
</tr>
<tr>
<td>Description</td>
<td>The framework suspends the assignment of tasks to a node.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>- The node ID is valid.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>- The node cannot be assigned any new tasks.</td>
</tr>
<tr>
<td>Actors</td>
<td>Framework</td>
</tr>
</tbody>
</table>

Normal flow
1. The framework receives a node ID
2. The framework sets the state of the node to 'suspended'.

Frequency of occurrence
For every shutdown of a node and once for every node when updating the software.

Related features
System feature 2.6.17, 2.6.16 and 2.6.25
2.6.16 System feature 16: Resume a node

Functional requirement
The framework must be able to resume the assignment of tasks to a node.

Priority
Medium

Inputs
- A node ID.

Outputs
- None.

Use case

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>Use case name</th>
<th>Description</th>
<th>Preconditions</th>
<th>Postconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>Resume a node</td>
<td>The framework resumes the assignment of tasks to a node.</td>
<td>- The node ID is valid.</td>
<td>- The node is available for assignment of tasks.</td>
</tr>
</tbody>
</table>

Actors
Framework

Normal flow
1. The framework receives the node ID.
2. The framework sets the state of the node as ‘active’.

Frequency of occurrence
Once for every node after updating the software.

Related features
System feature 2.6.15 and 2.6.25
2.6.17 System feature 17: Removing a node

Functional requirement
The framework must be able to remove a node rented from the cloud service.

Priority
High

Inputs
- ID of the node to be removed.

Outputs
- None.

Use case

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Remove a node.</td>
</tr>
<tr>
<td>Description</td>
<td>The framework terminates the use of a node from the cloud.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>- None.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>- The runtime of the task is predicted.</td>
</tr>
<tr>
<td>Actors</td>
<td>Framework</td>
</tr>
</tbody>
</table>

Normal flow

1. Validate that the ID corresponds to a node that is being rented from the cloud.
2. Use case 15 is executed.
3. Return any running tasks to the scheduler.
4. The system tells the cloud service to terminate the node.
5. The cloud service shuts down the node.

Alternative flow 1

1b. The ID does not belong to a node currently being rented.
2b. Register error and return.

Frequency of occurrence
Once for every provisioned node from the cloud.

Related features
System feature 2.6.14 and 2.6.15
2.6.18 System feature 18: Deployment of nodes

Functional requirement
The framework must be able to deploy the required software on a new node.

Priority
Medium

Inputs
- The node to deploy software on.
- The required software to deploy on given node.

Outputs
- None.

Use case

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Deployment of node</td>
</tr>
<tr>
<td>Description</td>
<td>The framework deploys software on a new node.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>- The node details are known.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>- The software is deployed on the node.</td>
</tr>
<tr>
<td>Actors</td>
<td>Framework</td>
</tr>
<tr>
<td>Normal flow</td>
<td>1. Install the software on the node.</td>
</tr>
<tr>
<td>Alternative flow 1</td>
<td>2b. The installation fails.</td>
</tr>
<tr>
<td>Assumptions</td>
<td>The software can be employed on the given node.</td>
</tr>
<tr>
<td>Frequency of occurrence</td>
<td>Once for every provisioned node.</td>
</tr>
</tbody>
</table>

Related features
System feature 2.6.14
2.6.19 System feature 19: Query job status

Functional requirement
The framework returns the status of a job on user request.

Priority
High

Inputs
• Job ID.

Outputs
• Status of job.

Use case

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>Use case name</th>
<th>Description</th>
<th>Preconditions</th>
<th>Postconditions</th>
<th>Actors</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>Delete job result</td>
<td>The framework returns the status of a job.</td>
<td>- The job ID is valid.</td>
<td>- None.</td>
<td>User</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Framework</td>
</tr>
</tbody>
</table>

Normal flow
1. The user sends a job ID to the framework.
2. The framework verifies that the ID is valid.
3. The framework returns the status of the job.
4. The user receives the status.

Alternative flow 1
2b. The job ID does not correspond to a job result.
3b. The framework returns an error.

Frequency of occurrence
Low.
2.6.20  System feature 20: Retrieve job results

Functional requirement
The framework returns the result of a submitted job.

Priority
High

Inputs
• A job ID.

Outputs
• The result of the job.

Use case

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Retrieve job result</td>
</tr>
<tr>
<td>Description</td>
<td>The user retrieves the result of a job.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>- The job ID is valid.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>- None.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actors</th>
<th>User</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal flow</td>
<td>1. The user submits a request for the result with job ID.</td>
<td>2. The framework returns the job results.</td>
</tr>
<tr>
<td></td>
<td>3. The user receives the results.</td>
<td></td>
</tr>
</tbody>
</table>

| Alternative flow 1 | 2b. The results are not available. |
| | 3b. The framework returns an error. |

| Frequency of occurrence | Once per job. |

Additional information
Job results are stored in the database, once a user requests a result it is returned through the frontend.
2.6.21 System feature 21: Monitoring of data

Functional requirement
Authorized users must be able to monitor the statistical data gathered and used by the system.

Priority
High

Inputs
- Gathered statistical data.

Outputs
- The statistical data in human-readable form.

Use case

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Monitoring of data</td>
</tr>
<tr>
<td>Description</td>
<td>Authorized users monitor statistical data from the system.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>- The user is authorized to view the statistical data.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>- None.</td>
</tr>
<tr>
<td>Actors</td>
<td>Authorized user</td>
</tr>
<tr>
<td>Normal flow</td>
<td>1. The user requests statistical data.</td>
</tr>
<tr>
<td></td>
<td>3. The user views the data.</td>
</tr>
<tr>
<td>Alternative flow 1</td>
<td>2b. The data is not available.</td>
</tr>
<tr>
<td></td>
<td>3b. Show an error message.</td>
</tr>
<tr>
<td>Frequency of occurrence</td>
<td>Low.</td>
</tr>
</tbody>
</table>

Related features
System feature 2.6.12
2.6.22 System feature 22: Garbage collect

Functional requirement
The framework must be able to remove unnecessary data from the system.

Priority
High

Inputs
• None.

Outputs
• None.

Use case

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Garbage Collect</td>
</tr>
<tr>
<td>Description</td>
<td>The framework removes all unnecessary data.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>- None.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>- No unnecessary data is present in the system.</td>
</tr>
<tr>
<td>Actors</td>
<td>Framework</td>
</tr>
</tbody>
</table>

Normal flow
1. The framework uses the garbage collection policy to remove all unnecessary data.

Assumptions
A garbage collection policy is defined.

Frequency of occurrence
At fixed intervals.

Additional information
The user can define the garbage collection policy. In an early implementation a simple garbage collection policy will be defined that deletes all data after 30 days.

Related features
System feature 2.6.23
2.6.23 System feature 23: Delete job result

**Functional requirement**
The framework must be able to delete a job result from a list of results.

**Priority**
Low

**Inputs**
- A job ID.

**Outputs**
- None.

**Use case**

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Delete job result</td>
</tr>
<tr>
<td>Description</td>
<td>The framework deletes a job from the list of results.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>- None.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>- None.</td>
</tr>
<tr>
<td>Actors</td>
<td>Framework</td>
</tr>
</tbody>
</table>

**Normal flow**

1. The framework verifies that the job ID corresponds to a stored job result.
2. The framework deletes the corresponding job results.

**Alternative flow 1**

1b. The job ID does not correspond to a job result.
2b. The framework returns an error.

**Frequency of occurrence**
Once per job.

**Related features**
System feature 2.6.22
### 2.6.24 System feature 24: Setting of system variables

**Functional requirement**
The framework must allow for authorized users to edit certain system variables.

**Priority**
Low

**Inputs**
- None.

**Outputs**
- None.

**Use case**

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>Use case name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>Setting of system variables</td>
<td>The user accesses and sets system variables.</td>
</tr>
</tbody>
</table>

**Preconditions**
- None.

**Postconditions**
- The required changes are made.

**Actors**

<table>
<thead>
<tr>
<th>Normal flow</th>
<th>Authorized user</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>The user requests to set system variables.</td>
<td>2. The framework validates the new variable values.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. The framework responds with a form where the system variables can be set.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. The user changes one or more system variables.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5. The framework stores the changes.</td>
</tr>
</tbody>
</table>

**Alternative flow 1**

| 2b.          | The new values are invalid, return an error. |
|             | The user receives an error. |

**Frequency of occurrence**
Low.
2.6.25 System feature 25: Update node software

Functional requirement
The framework must be able to update all software that has been deployed on the worker nodes.

Priority
Medium

Inputs
• The software update script and required data.

Outputs
• None.

Use case

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>Update node software.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The framework updates the software on a worker node.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Preconditions</th>
<th>Postconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>- None.</td>
<td>- The node runs the updated version of the software.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actors</th>
<th>Framework</th>
<th>Node</th>
</tr>
</thead>
</table>

Normal flow
1. The framework executes use case 15 to suspend the node.
2. The framework waits for the node to finish its tasks.
3. The framework sends the update script and required data to the node.
4. The node executes the update script.
5. The node notifies the framework that it is done updating.
6. The framework executes use case 16 to resume the node.

Alternative flow 1
4b. The new values are invalid, return an error.
5b. The node notifies the framework that the update failed and rollback to previous version.
6b. The framework notifies the administrator that the update failed.

Special requirements
The node must be compatible with the software update.

Frequency of occurrence
Once for every update for every node to which the update applies.

Additional information
The update script is provided by the user and is sent to the node. When a node restarts it will automatically download the new software and run it.

Related features
System feature 2.6.15 and 2.6.16
2.7 Non-functional requirements

2.7.1 Performance requirements

PE-1: Total cost of renting cloud computing resources must not exceed the set budget for that period.

PE-2: The prediction of the execution time for a job must not be off by more than 20% in 95% of the cases.

2.7.2 Safety requirements

SA-1: The framework must not reduce the likeliness of a job being handled successfully.

SA-2: The framework must not fundamentally alter a job for any reason.

2.7.3 Security requirements

SE-1: The framework must not compromise any sensitive information through communication with or storage in the cloud.

SE-2: The framework must not maintain a lower security level than the system in which it operates at any point in time.

2.7.4 Platform requirements

PL-1: The framework will support nodes that can run on both Linux and Windows OS.

PL-2: The nodes have at the very least support for JRE version 1.6.
3 System Design

This section details the architectural the design of the framework under consideration. It elaborates on the design decisions taken in the development of the system. Section 3.1 explains the foundation of these decision by setting forth the architectural principles on which the system builds. It is followed by the framework design and implementation which serves to show how the functional requirements will be realized in the system. Consequently, the selected technology will be explained and defended and we state how the chosen architecture satisfies the non-functional requirements.

3.1 Architectural principles

In this section the various design principles and design patterns that will be taken into account while writing the framework are discussed first. Consequently, the code practices that are applied are listed, followed by the tools, libraries and frameworks that are used. Material that was used for the description of these principles and patterns includes “Software Architecture for Developers” by Simon Brown [8], “Real-Time Systems” by Herman Koppetz [9], “Design Patterns” by Gamma et al. [10] and “Head First Design Patterns” by Freeman et al. [11].

3.1.1 Design Principles

This section sets forth the design principles that apply to this system. These principles serve as the foundation of the architectural design of the system. Each principle is given with a brief explanation of its purpose and application.

Single Responsibility Principle An important principle to increase the maintainability and testability of software is the single responsibility principle. By separating an application into units that each address only one concern or responsibility, each unit becomes independent and can be maintained and tested without affecting other units. This is also known as separation of concerns.

Low coupling Coupling occurs when there are interdependencies between one module and another [1]. When interdependencies exist, a change in one place will result in the need for a change somewhere else. Ideally, interdependencies should be kept at a minimum to make the code maintainable.

High cohesion High cohesion is a form of divide and conquer where related components are kept together. This makes the system as a whole more understandable. Benefits of high cohesion are higher changeability and maintainability.

Small components Having small components in a system ensure a fine grained design. Each task is associated with one component and these are independent of each other. This is one way of achieving separation of concerns.

Modular design Modular design embodies the combination of separation of concerns, high cohesion, small components and code reusability. By applying these principles together, the application becomes separated into reusable modules that are separately testable and maintainable.

High abstraction (interfaces) Abstraction allows for the deferring of details in such a way that complexity is reduced. By using abstractions the understanding of a program can be made easier as the unnecessary details are hidden.

Code reusability When various parts of a system may be used in other contexts, designing for reusability is beneficial. Applying this principle prevents code duplication and improves code quality, as improvements and extensions to reusable modules can also be applied elsewhere.
Reuse existing code  The reuse of design or code allows you to take advantage of the investment that you or others have made into designing, building, testing and maintaining reusable components. This reduces the amount of design or development required.

Open/Closed Principle  The open/closed principle states “software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification”; \cite{fallbock2001} that is, such an entity can allow its behaviour to be modified without altering its source code. By incorporating this principle the code can be made more flexible and testable.

Platform independence  Platform independence is of special importance in the distributed systems paradigm. This design principle is concerned software that may run on diverse platforms.

Design for testability  By designing the code to be testable, it can be ensured that most functionality can be automatically tested. Thorough testing can result in robust code quality and helps in keeping errors at a minimum.

Defensive Coding - Reliability  The defensive coding principle states that all input to a component should be handled properly. Invalid input should never result in erroneous behaviour, but should be rejected instead, providing an indication of the error when possible. This makes it easier for code to be reused and for users to correct their errors.

Causality  The principle of causality ensures the deterministic nature of a distributed system \cite{thelander2001}. By making a system more deterministic all errors and failures can be traced to their origin. Also, the nature of the system is made predictable.

Observability  Understanding of system behaviour is severely impeded by non-visible communication between architectural units. To increase observability of external and internal behaviour, applying the observability principle in design is necessary.

3.1.2 Design Patterns

Architectural design patterns serve to translate the above stated design principles to practices in the system design. Applying these patterns is the foundation for realizing these principles in the implementation of the system. In this section, the design patterns that are of importance in development of the framework are listed.

Dependency injection  The dependency injection pattern, also often called the inversion of control pattern, separates the implementation of a component from its dependencies. This reduces coupling and allows isolated testing by injecting stubbed or mocked dependencies. It also makes it possible to increase cohesion and modular design. In the system design, this pattern is applied by using Google Guice’s dependency injection, see \ref{3.1.4} for more details. This pattern will be used in the constructors of most classes in the system, where the parameters are injected into the class, rather than passed into it.

Multi-tier design  In software engineering, multi-tier architecture is a client-server architecture in which presentation, application processing, and data management functions are logically separated. Each layer encapsulates its own functionality. Only adjacent layers in the architecture can communicate with each other. This design helps achieve the goal of separation of concerns. This design will be applied in every subsystem, where this division will be made to decrease coupling and increase maintainability.
**Observer pattern**  The observer pattern is a design pattern in which a set of observers is maintained by an object via notifying them of any state changes. Usually this is done by calling one of their methods. The main use of this pattern lays in implementing distributed event handling systems. In the implementation of the system, this pattern is applied to decrease computation power needed by threads, by having them wait for relevant changes rather than continuously.

**Façade pattern**  The façade pattern serves to provide a simplified interface to a large body of code and is a commonly used pattern in object-oriented programming. The façade is an object that acts as a frontend for a set of objects who deliver a set of services together [11]. The façade pattern is applied in the design of this system through the implementation of the front-end, through which users submit jobs, and of the cloud service module, which acts as a façade for the actual cloud service.

**Composite pattern**  The composite pattern is a design pattern that deals with partitioning of a system. The system is partitioned in a group of objects that are to be treated as a single instance of an object [10]. The composite pattern allows for a uniform treatment of objects that exhibit similar similar functionality. The composite pattern is found in the system in classes that combine multiple other classes, such as the NodeDetails object, which contains several node-related objects itself.

**Command pattern**  The command pattern is a behavioral pattern that makes it possible to send commands over a network. The head node of the systems issues commands for execution of certain tasks on the worker nodes that it is connected to. This functionality is provided for commands that are sent over the underlying network of the distributed system, the cloud or the grid.

**Policy/Strategy pattern**  In computer programming, the strategy pattern (also known as the policy pattern) is a particular software design pattern, whereby algorithms can be selected at runtime. Formally speaking, the strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients using it [11]. In the design of the system, this pattern becomes visible in the implementation of the policies. The policies are grouped under an interface, such as AllocationPolicy, where the implementation of the policy can vary and be swapped.

**State pattern**  The state pattern is intended to model business logic as a state machine. Using this pattern separates the behaviour of the different states and allows the addition of new states without modifying existing ones. In general, the state pattern is very similar to the strategy pattern, where the strategy is changed for every state transition. This pattern is applied in the implementation of node states, which can alter behavior throughout the system depending on the current state.

**Adapter pattern**  When dealing with different interfaces, it is often needed to adapt multiple interfaces to be compatible. This is achieved by creating an adapter or a wrapper class that conforms to the global interface and achieves the required behaviour by invoking the wrapped class. This patterns is amongst others found in the design of the data stores, which can have one or more data store interfaces depending on the used database.

### 3.1.3 Coding practices

This section lists a set of the most important coding practices that will be upheld in the implementation of the system. These practices will be further enforced through usage of the tools, frameworks and libraries listed in the next section.

**Error handling**  Errors occurring in the system will be logged with a stacktrace. Feedback of the error is given to the user in an informative manner. All internal errors will be logged in such a way that the error can be reproduced and the administrator is informed of the error.
**Logging** Logging forms an important component of this project. All activities in the system will be logged, including arrival and treatment of a job, provisioning and allocation of resources and statistical data regarding job runtimes. All information logged will be used in the various provisioning and allocation policies. Furthermore, logging will be used for the debugging of the system and for monitoring purposes, to give administrators an insight in the processes in the system.

**Precondition verification** In verification of preconditions, no ‘asserts’ are used. The validity of the input and the system state is verified before any method is executed. In the event of invalid preconditions, an exception is thrown or the error is handled internally.

**Fine grained synchronization** In order to prevent any data errors, synchronization on the smallest element of a method is used. Fine grain synchronization can result in speedups, whereas coarse grain synchronization, synchronization on methods or classes, can result in slow-downs, defeating the purpose of parallelization [13].

**Code style** All code written will be properly commented. This includes javadoc comments before every method, general comments before every class and in-line comments before complex code. The Oracle “Code Conventions for the Java Programming Language” are followed and variable and method names are chosen to accurately reflect their purpose [14]. Tools such as checkstyle will be used to enforce coding conventions.

**Testing conventions** All code will be written in such a manner that unit testing can be applied. Furthermore, acceptance testing and integration testing will be carried out in order to validate the robustness of the framework. In the procedure of unit testing, mocking will be applied to satisfy the separation of concerns principle as well as to isolate specific behaviour. All testing will be carried out in an automated manner and continuous integration tools will be employed.

### 3.1.4 Frameworks, tools and libraries

This section lists the framework, tools and libraries that are either used in the implementation process, such as the used IDE and testing tools, or needed to support required functionality, such as graphing tools and cloud libraries. The items listed here are divided into the categories ‘process’, ‘product’ and ‘infrastructure’.

**Process** The following tools, frameworks and libraries are used to aid in the implementation process. Each item is given with a rationale and a short description of its purpose.

- **Maven** Apache Maven [15] is a lifecycle management tool, based on the concept of project object model, that centralises dependency management, the building process of the project, reporting and documentation. We use maven to streamline our development process.

- **JUnit** Testing is an important part of software development. In order to make life easier, we use automated testing. JUnit [16] is a framework that allows us to do this. It supplies standard methods to build concise and accurate test cases.

- **Mockito** To do accurate unit testing, dependencies between different units should be removed. Mockito [17] allows classes to be mocked, so that their behaviour can be specified for the test-case. Interactions with mocked objects can also be verified.

- **EMMA** While testing a codebase, it is important to ensure as much situations and code as possible is covered. In order to monitor this coverage of our code, we use EMMA [18]. It is also available as a plugin for Maven.
Checkstyle  To maintain a common coding convention for all source in the project, we will use Checkstyle [19]. This tool helps enforce style rules by finding contraventions. It is also available as a plugin for Maven.

PMD and Findbugs  PMD [20] and Findbugs [21] are static code analysis tools, which find common programming flaws that may be indicative of bugs. PMD works on source code and therefore finds problems such as unnecessary object creation, empty catch blocks, unused variables and so forth, whereas Findbugs works on bytecode. Usually each of them finds a different set of problems.

Jenkins  In order to constantly monitor the quality of the project, we will perform continuous integration using Jenkins [22]. Jenkins Continuous Integration is a free open-source branch of the Hudson [23] project by the Eclipse Foundation. This tool supports automatic building, testing, reporting and deploying.

Eclipse  The Eclipse [24] programming environment will be used for development purposes. Eclipse is a community-driven project that is aimed at building an open development platform. Due to its open-source nature, many plugins for the tools that will be used in the course of this project exist.

Doxygen  Doxygen [25] will be used to generate an online version of the documentation and an offline reference manual from a set of annotated source files. Furthermore, Doxygen can be configured to automatically generate class diagrams from the code.

SVN  Subversion [26] will be used for version control of the code. Apache Subversion will be used to smoothen the development process.

Product  The following tools, frameworks and libraries are used to aid in the product. Each item is given with a rationale and a short description of its purpose.

Guice  Google Guice [27] is a dependency injection framework. It helps separate the responsibility of resolving and instantiating dependencies from the rest of the code. It also makes switching between different implementations of functionality very easy.

Log4j  Apache Log4j [28] is a high performance logging library with a lot of configurable options. Logging can be enabled and disabled without modifying the application’s binary code, which makes debugging easier and increases observability of the system.

Java Database Connectivity  The standard Java database connectivity (JDBC [29]) will be used to connect Java programs and the SQL databases. JDBC is the industrial standard for connectivity between the Java programs and a diverse assortment of databases.

Cloud libraries  APIs for commercial clouds, such as Amazon EC2 [30] and Windows Azure [31], will be used to startup a new node, deploy software on the node and delete nodes.

Jersey  Jersey [32] is the open source reference implementation of the JAX-RS (JSR 311) standard. This defines a standard for building RESTful webservice. Jersey provides an API that eases the development of HTTP-based communication in a RESTful manner.

Flot plotting library  The open source graphing package Flot [33] will be used to graph all the statistical data that is collected. Flot is a javascript plotting library for jQuery. Using jQuery helps to abstract away the differences between browser versions. Furthermore, using Javascript for the user interface separates this concern from the rest of the framework.
Bootstrap Bootstrap [34] is an open-source HTML/Javascript/CSS framework. It was developed by Twitter, in order to develop user-interface components. Bootstrap can be used to quickly design a accessible HTML frontend.

Java EE 6 The framework will be programmed in the programming language Java [35]. This is consistent with the current programming language of the system in which the framework will operate. Java Enterprise Edition 6 will be used.

Infrastructure The following tools, frameworks and libraries are used in the infrastructure of the framework. Each item is given with a rationale and a short description of its purpose.

Java Message Service The Java Message Service (JMS [36]) is a middleware API for sending messages between clients based on a publish-subscribe protocol. It has been standardized as JSR 914. It allows the communication between nodes in a distributed application to be loosely coupled. The JMS implementation provided by JBoss will be used.

HTCondor HTCondor [6] is a workload management system that schedules jobs both serially and in parallel. Condor has its own internal scheduling policy that allows for priorities to be given to jobs.

JBoss AS 7 JBoss [37] is an application platform that will be used to host the framework. It provides services such as clustering, caching and messaging.

Apache Cassandra Cassandra [38] is a distributed database system, which provides for scaling functionality. As the system grows, Cassandra can scale with it, thus making data storage easy and efficient.

3.2 Framework design and implementation

This section aims to outline the description of the software architecture, including major software components and their interactions. The book ‘Software Architecture for Developers’ is used to make these views [8].

3.2.1 Logical view

The framework is divided into logical components, each with their own responsibility. This section aims to describe the major components in the system. Each of these component and its functionality is described here, together with their interactions. The figure depicting the logical view diagram can be found in appendix B.5.

The front-end component encompasses all functionality required to interface with the user, the administrator or other systems submitting jobs. The user can submit jobs and query their status and results through the front-end. The front-end submits the jobs to the job monitor component, which manages the jobs and their statuses. The job monitor submits the job to the scheduler, which consists of two components, the preprocessor and the allocation manager.

The preprocessor retrieves additional job data from the data store if needed, and splits the job into tasks. These tasks are then handed over to the allocation manager, which first orders them based on a prioritization policy. Then the allocation manager uses the allocation policy based on run-time predictions for the task, made by the prediction manager, to allocate each task to a node. The allocation manager also sends metadata for each task to the job monitor, such as where it is allocated. This allows the job monitor to query the status of each task at each node.

The prediction manager bases its prediction on the historical data collected and processed by the data monitor. All data collected by the data monitor is stored in the data store and can be retrieved from there. These statistics are also used by the provisioning manager to determine how many resources are currently required.
The provisioning manager interfaces with a cloud service to start and stop nodes. When a new node is started, the deployment manager deploys software on the new node. The version of software to be used and other constraints imposed on the software are managed by the software manager. The versions and constraints can be managed by the administrator through the front-end.

3.2.2 Interface view

The interface view describes the internal and external interfaces at a high level. It describes amongst others the type of the interface, synchronous or asynchronous, and the type of communication that is being used. The interface view can be found in appendix B.2.

The framework consists of five subsystems. Communication between interfaces within the same subsystem happens through Java calls. The Java Messaging System is used for communication between subsystems. Here, point-to-point JMS is used whenever the messages sent between the interfaces are received by one consumer only. Publish-subscribe durable JMS is used whenever more than one consumer will receive the message.

The messages that are passed between subsystems and within subsystems are generally objects. Messaging between subsystems is asynchronous, whereas messaging within a subsystem is generally synchronous. More specifically, the following messaging links are asynchronous: links between requesters and request handlers, links between submitters and receivers and links between publishers and their subscribers. Asynchronous messages are guaranteed through the consistent use of JMS.

Idempotency of interface is of specific importance to avoid duplication of tasks through the system in order to maintain high efficiency in the system. Idempotency will be guaranteed in all interfaces through the use of IDs and checksum verification.

Messages passed between subsystems can be received out of order. Where dependencies between tasks makes this problematic, this will be checked. The connection between the provisioning manager and the allocation manager is an example of such a link, where the allocation manager may make wrong assumptions if messages aren’t received in the correct order.

3.2.3 Design view

In the design view, the low level implementation details are elucidated. The design demarcates the three layers that the system is divided into. The common patterns used in this system are shown in this chapter. The class diagrams can be found in appendix B.3. The framework is divided into three different layers. These layers are: the presentation/communication layer, the business logic layer and the data access layer.

The presentation layer exposes an interface to users and other systems. The communication between the users and the framework is done in this layer. Also the communication between the various components of the framework is represented here. The classes that publish data to which other subsystems subscribe to are part of this layer. The classes that form a part of the presentation layer are represented in blue.

The business logic layer consists of business logic contained in a code base that is separated from the data layer and the presentation, or graphical user interface, layer. The data classes that represent a job or a task are part of this layer. The managers, such as allocation manager, and the monitors, such as the job monitor, are the classes that actually manipulate these data objects; they form the crux of the operational logic of this system. These classes are primarily present in this layer. The classes that make up the business logic are marked in white.

The data access layer in computer software, is the layer of a computer program which provides simplified access to data stored in persistent storage of some kind, such as a distributed database. In this system the components that subscribe to the data that other components publish are also included in this layer. The classes that are a part of the data access layer are marked in orange.

The sequence diagrams show how the use cases are implemented in the framework. They are closely related to the class diagrams and document all communication that takes place in the framework. The sequence diagrams can be found in B.1.
Messaging in the framework is handled using an existing solution, JMS, and HTTP calls are used for communication between the system and the front-end of the framework. These solutions are tested and documented and fit the requirements of the framework.

### 3.2.4 Infrastructure view

In the infrastructure view, the physical hardware on which the software will be deployed is described. The figure depicting the infrastructure view can be found in appendix B.4. The physical hardware on which the framework relies consists of one or more head nodes that are owned, a grid of zero or more worker nodes which is permanent, multiple data stores that are rented, and zero or more worker nodes that are rented from a commercial cloud service. A dedicated logging server is used, on which all components log their relevant data.

A minimum of two load balancers is used as a redundancy measure, as this is a single point of failure otherwise. Furthermore, more than one node can be employed as head node, both for performance reasons and as a redundancy measure. The usage of databases in the system allows for disaster recovery when a failure occurs within the system. When a node fails, the cloud service provides a replacement and restores lost data.

The framework uses two computing sources, namely a grid of worker nodes and worker nodes rented from a commercial cloud. These two components are on distinct sites and connected via the Internet. The grid is of a fixed size and uses its own workload management system. For the cloud nodes, depending on the choice of cloud service, several options for the size of the node, in terms of cpu power and memory, are generally available.

### 3.2.5 Deployment view

The deployment view entails the mapping of the software to the hardware. The components that are a part of the framework are designed to be scalable and thus can run across multiple hardware nodes. This can be seen in the case of the head node in the infrastructure view where the head node can run as multiple nodes. However these components can also run on one server.

Components such as the node client are designed to run as single entities on every single node from the grid and the cloud. All components are active-passive as a fully redundant instance of each node is provided, which is only brought online when its associated primary node fails. Each component is privy only to certain set of data. This data can be shared with other components based on the need that the component has for this set of data. There is no excess replication of data across multiple sites.

### 3.2.6 Operational view

Monitoring and logging is an important part of the framework as it will function in a system and handed over to COAS. Extensive logging of the system is a functional requirement. This will be realized through logging of system state and its changes, incoming and outgoing messages, warnings and errors. Logging messages are relayed to a dedicated logging system.

The front-end provides access to the status and result of a job that is submitted to the system, which can be monitored there by an authorized user. Administrators can use the front-end for monitoring and setting of system variables. Furthermore, administrators can monitor and manage the framework using a JMX compliant tool. Using these tools, administrators can also be notified of important events in the framework.

### 3.2.7 Security view

Safety and security has to be guaranteed for the data and business sensitive information. The framework is situated in the secure environment of the system in which it functions. Therefore, any communication from the client and any communication of the framework that does not leave the network is considered to be secure.

There is communication between the framework, in a secure environment, and a commercial cloud service, in a public domain. The communication protocol between these entities is encrypted. Accessing the virtual instances run by the cloud service requires authentication, which is achieved
through the use of certificates. The databases require authentication in order to retrieve or store data. The access to the messages in the JMS message queues is in a protected environment and authentication is required to access these queued. The head node has the certificates corresponding to the software that is to be installed.

The distinction between two different kinds of authenticated users is made, namely regular users and administrators. The former group has access to status and result of a job that was submitted to the framework. The administrators have access to system variables and can monitor and manage the framework.

3.2.8 Data view

The framework stores data in a distributed database system, namely Cassandra. Since scalability is an aim of the framework design, no estimation is made regarding the amount of data that has to be stored. Using Cassandra ensures that the datastores are scalable. Cassandra introduces replication of all data that is stored, thereby ensuring constant availability of this data. This replication furthermore renders the need of an explicit backup redundant.

A garbage collector runs periodically and removes redundant data from the data stores. A policy is defined such that useful data is removed and that no data persists for longer than a user-defined cut-off period. Log files are stored with the logging server. No deleting policy is defined for these files.

3.3 Technology selection

In the design of the framework, generic interfaces were used in most cases so as to avoid selecting a specific technology. The frameworks/tools and libraries section lists the chosen technology. Most of these are workflow supporting technologies. On four occasions, a specific technology was chosen in the implementation of the framework. These, and their alternatives, are listed and defended here.

JMX was chosen for monitoring of system variables and system state. JMX makes the monitoring of these variables easy as it is a standard to which several other tools are compliant, such as JConsole and VisualVM. The other option was to create an interface which would monitor all the system variables. However, since there was a standardized tool available, the choice was made to use JMX instead of reinventing the wheel.

To store data, a choice was made for Apache Cassandra. Cassandra is an open source distributed database system which allows for scalability of a system. The alternatives to Cassandra were relational databases and Hadoop Distributed File System (HDFS). Relational databases are not scalable and thus not suitable for an application such as this. HDFS was also rejected as there was a need for a database and not simply for a file system.

The JMS implementation included with JBoss was chosen as the messaging system within this framework. There is an option to use the ActiveMQ implementation of JMS, however this was not desirable as the original system uses JBoss.

The Google Guice framework is used to provide dependency injection. Guice has been chosen over its alternatives, such as the Spring framework, as it is more lightweight and easy to use.

3.4 Architecture justification

This section aims to justify the architectural design decision made with respect to the non-functional requirements. Four key non-functional requirements, which are referenced from the requirements analysis, will be discussed.

Non-functional requirement PE-1 states that functionality should be provided to guarantee that the budget is not exceeded by the system. The provisioning policy takes the budget constraints into account in its actions, thereby insuring that the budget is not exceeded.

Safety requirement SA-2 dictates that jobs must not be fundamentally altered within the system. When the job enters the system, it is split into one or more tasks whose result is reassembled later. No other manipulation is performed on the job or tasks. The splitting policy can be defined in such a manner that splitting and reassembling the job does not corrupt it.
Security requirement **SE-1** states that no sensitive information may be compromised through communication between the framework and any external service. Furthermore, requirement **SE-2** requires that the framework is at least as secure as the system in which it operates. The security view elucidates the various security measures that are taken into account in the design to ensure that these requirements are met.
4 Testing and quality assurance

Throughout the process, measures were taken to assure high quality for our codebase. These quality assurance methods can be divided into three categories: unit testing, acceptance testing and code analysis. Unit testing refers to the testing of fine-grained units of code, such as single classes or methods, whereas acceptance testing refers to the more coarse-grained testing of whole subsystems or even the whole framework. How these tests were executed can be found in section 4.1 and 4.2 respectively. Code analysis refers to the static analysis of the source code to determine code quality metrics as well as finding patterns that are commonly indicative of bugs. The tools used for this analysis and their results are discussed in section 4.3. Finally, in section 4.4 we will discuss the feedback of the Software Improvement Group on the quality of our codebase.

4.1 Unit testing

The most common type of testing used in this project is unit testing. Test suites are created using the JUnit framework for Java. Mockito is used to isolate the unit under test from its dependencies.

Unit tests are executed manually by the programmers, as well as automatically on the continuous integration server, by Jenkins. This server continuously builds the newest version of the codebase and executes all unit tests. The number of passed tests and failed tests is recorded and trends over time are graphed, such as in figure 4.

![Fig. 4: The trend of test results over the course of the project.](image_url)

Furthermore, in order to get a reasonable indication of how much of the code is being tested, coverage metrics are being recorded using Cobertura. These metrics show what percentage of the code was exercised during the tests, such as in figure 5. In order for these metrics to give a relevant indication of the test quality, some units were excluded from this report.

![Fig. 5: The trend of test coverage over the course of the project.](image_url)
4.2 Acceptance testing

Another important part of the testing practices is acceptance testing. This entails exercising a large part of the system, including its dependencies, at once and evaluating its behaviour. This is done manually by the programmers, using different tools to verify system state.

In order to control and verify the framework’s functionality, several adaptations were made. The framework first of all outputs useful logging information that allows monitoring of its actions. Besides that, the state of the different processes can also be monitored by connecting to them with a JMX client.

4.3 Code analysis

Static code analysis can give quick insight into the code quality by calculating metrics, such as method length or number of parameters. While these metrics are not indicative of the code quality by themselves, they can often pinpoint so-called ‘code smells’ which expose an underlying problem. A method length that is too high, for example, can be an indication that this method does not have one clear concern.

Static code analysis can also be used to detect certain patterns that are associated with commonly occurring coding mistakes. These mistakes may not always be immediately obvious to someone reading the code and may cover cases that were forgotten during the test.

4.4 Software Improvement Group feedback

During this project, the codebase was evaluated by the Software Improvement Group twice. The first evaluation was received about seven weeks into the project and will be discussed in section 4.4.1. The improvements made after receiving this evaluation are explained in section 4.4.2 and the final evaluation is discussed in section 4.4.3.

4.4.1 First evaluation

At the moment of the first evaluation, the Software Improvement Group rated our code almost 4 out of 5 stars on their maintainability scale. The three main points that reduced our rating were:

- **Unit Interfacing:** the number of methods with an above average number of parameters is too high. This indicates that there may be a lack of abstraction and too much complexity in one unit.
- **Code Duplication:** the amount of redundant code is too high. There are multiple places where the same or similar code is used.
- **Unit Size:** the number of methods that are too long is too high. This indicates that these methods are too complex and should be split into smaller ones.

4.4.2 Improvements

The feedback received from the Software Improvements Group was considered for improvement during the rest of the project. However, as the project was rather large for the allocated time period, not all code could be improved. In cases where problems were not resolved, measures were taken to make sure the problem did not get worse.

In terms of unit interfacing, more abstraction was added in some places to handle this. However, especially for high-impact classes, the effort was not worth the return in the short time available.

Code duplication only occurred in specific sets of classes that were automatically generated from service descriptions or imported from another project. While it is possible to remove duplicate code by refactoring, this would defeat the purpose of generating this code, now as well as in the future.

When modifying code that contained long methods, these methods were split up into smaller methods to reduce the complexity of each code block. While it was not worth it to split up every long method, it was ensured that these methods did not grow much larger.
4.4.3 Final evaluation

In the final evaluation the Software Improvement Group noted that our code quality score had remained constant, although the size of the codebase had grown by about 25%. They noted that while not all of the previous points had been corrected, their feedback had been taken into account in the development process. Furthermore, they mentioned that it was good that the amount of test code had greatly increased.
5 Implementation Details

Implementing the framework was a rather big effort, consisting of big and small tasks. This section outlines the main challenges we faced and the main features we achieved during development. The implementation of the cloud APIs and the HTCondor API are discussed in section 4.1 and section 4.2 respectively. The main issues faced when working with Apache Cassandra are enumerated in section 4.3. A solution for connecting to nodes is explained in section 4.4. Finally, section 4.5 describes the mathematical model behind runtime prediction and section 4.6 describes the calculation of task priorities.

5.1 Cloud services

This framework provides support for leasing instances from two cloud services, namely Microsoft Azure and Amazon EC2. It was required to allow scaling dynamically using either one of the cloud services. As both cloud services have different properties and requirements, supporting both of them transparently was a challenge.

Microsoft Azure provides a REST API in order to issue command for creation or deletion of a virtual machine. Calls to this REST API are made using the HTTPS protocol. However, creating an HTTPS connection from Java that connects to the Azure Management Service requires a complicated series of steps to generate keys, which is not very well-documented. Leasing of Linux instances is particularly hard, as there are very specific requirements for the Linux image to be used. Furthermore, operations on an Azure deployment cause a lock on the whole deployment, such that all other operations have to wait.

Amazon provides a Java SDK for development of applications that can scale into the EC2 cloud. In comparison to the Azure API, this SDK has much more documentation and provides much better low level support. EC2 also does not limit operations on a deployment, meaning that adding and removing instances can be done much faster.

On both EC2 and Azure, Ubuntu 12.10 images were created. These images have startup scripts installed on them that download the client software and run it immediately. The framework then attempts to connect to the node on an HTTP port until the node responds. Then the complete connection between the framework and the node is initialized and all JMS queues and topics are initialized.

5.2 Cassandra

Apache Cassandra is the database system that was chosen for all the datastores for the framework. The Hector API was used in order to insert to and retrieve from the database. Because Hector is not a fully developed API, making more complicated queries is not fully supported, and can thus be very hard to implement. The API does not provide all the desired queries, so that the setup of the database has to be adapted to the available types of queries.

Cassandra typically uses a RandomBytePartitioner in order to partition the data over the different clusters. However, for this framework it was desired that the ByteOrderedPartitioner was used, so that a RangeSlicesQuery could be executed. This query requires that all the entries be ordered so that a range of entries could be selected. This is the Cassandra equivalent of executing a SELECT item FROM table WHERE condition in SQL.

The Cassandra service used for testing was a single node cluster. However, this cluster can easily be scaled out to a multinode cluster. The Cassandra version used was Apache Cassandra, though at first the DataStax version of Cassandra was tested as well. This proved to be too rigid as it did not support changing the partitioner.

5.3 HTCondor SOAP API

To allow tasks to be handled by a condor grid, the framework has to communicate to the condor scheduler. HTCondor provides a SOAP API for submitting and querying jobs. While this API was easily implemented in Java by importing the WSDL with the wsimport utility, setting up HTCondor to work correctly proved a bit more difficult.
After enabling the SOAP API of the HTCondor scheduler, jobs can be submitted to it and their statuses retrieved. However, without proper configuration these jobs will not be executed. While at times jobs would go to a 'held' state with a short explanation as to why, sometimes they would just be idle without any useful information in the logs. These issues were resolved by setting different properties on the jobs, relocating the executables to a specific directory and passing the main class name as the first argument to a Java job.

5.4 JMS bridging

The communication between the different subsystems is based on messaging patterns and uses the Java Messaging Service (JMS). All subsystems are connected to a central JBoss server that provides the JMS service. To communicate with the nodes, these have to be connected to this JMS service as well. However, these nodes can reside in a public cloud environment, and thus be outside of the private secured environment that resides behind the company firewall.

Each node hosts an embedded JMS service and messages are moved from and to the main service using JMS bridges. These bridges consist of a consumer and producer pair, where the consumer reads messages from the source and the producer produces the same messages on the target. There are two good reasons to do this:

- **security** - by connecting from the main server to the nodes, it is not necessary to open up the firewall. This reduces the risk of malicious connections and exploit attempts.
- **control** - the main server is in control of the connections to the nodes. This makes it possible for the system to detect when a connection was lost, monitor the throughput of the queues and adjust messaging parameters.

As the alternative solution, directly connecting the nodes to the main JMS service, does not have these benefits, bridges were chosen as the best solution. This solution has the additional benefit of keeping the required software on the node simple.

5.5 Runtime prediction

The framework handles jobs by splitting them into tasks, if needed, by executing the identity operation, which are then submitted to an allocation policy. In this process, predictions for task runtimes are needed. These runtime predictions are based on task size and type, and a task prediction is returned for every node based on node type.

In order to get predictions for every node for a certain task, registered runtimes for tasks of the same type are grouped by nodetype. Within these groups, a linear relation is established between task size and task runtime, using linear regression. Here, simple (2D) linear regression is applied to establish a relationship of the form

\[ r = \alpha + \beta s \]

where \( r \) is the runtime and \( s \) the size of a task. Given a collection of observations \( s_i \) and \( r_i \), the coefficients \( \alpha \) and \( \beta \) can be found as follows:

\[ \beta = \frac{\sum_{i=1}^{n} (s_i - \bar{s})(r_i - \bar{r})}{\sum_{i=1}^{n} (s_i - \bar{s})^2} \]

\[ \alpha = \bar{r} - \beta \bar{s} \]

where \( \bar{x} \) is the average of all elements \( x_i \). Where no information is available, for example for all nodes of a certain type, aggregation is used to fill in the blanks. When no information is available in general, that is, when a job has not been treated by the system before, predictions default to a value of 60 seconds for every entry. In other words, all tasks and nodes are treated equally until more information arrives.
5.6 Prioritization

One of the requirements was the possibility of submitting jobs with differing priorities. The system takes these priorities into account as well as aging in every policy except the first-come-first-serve allocation policy. The greedy allocation policy behaves as a FCFS policy that does take pre-assigned priorities into account.

For all policies except FCFS, when allocating, a certain local priority is computed by the allocation policy. This priority is then multiplied by the input priority, which is based on aging and the pre-assigned priority, to yield the global priority of this task. For implementation details regarding allocation, see chapter 6. This input priority is computed as follows:

\[ p_{\text{initial}}(t_i) = \text{order}(t_i) \times \max\{1000, a^{\text{rel.age}(t_i)}\} \]

where \text{rel.age}(t_i) is the time task \(i\) has waited in the allocation queue divided by its predicted runtime, and \text{order}(t_i) is the pre-assigned (user assigned) priority of the task on a scale from one to \(n\), where \(n\) is the highest priority a user can assign. Aging is done by increasing a factor exponentially based on the relative age of a task. The base \(a\) can be set to increase the speed at which the task age start to dominate its overall priority. Higher values will guarantee that tasks don’t age long at low loads, but may have averse effects at high loads.
6 Policy Investigations

6.1 Research proposal

The current problem is to effectively schedule both on the grid and the cloud. There is a condor grid available, that has a fixed number of execution slots. When the number of tasks that need to be executed exceeds the number of available condor slots, the infrastructure needs to be expanded in order to compute all tasks in a timely manner. A fixed budget is available per specified time period from which Infrastructure as a Service resources can be leased.

6.1.1 Problem Statement

The job that comes into the system can be split into multiple tasks. Each one of these tasks can be given a different priority. There can be dependencies between tasks and this facilitates the need for some tasks to wait on the result of other tasks from the same job. The tasks are then executed based on an allocation policy. The allocation policy must take the aforementioned criteria into account.

At the same time the system has to take into account that the infrastructure available may not be sufficient to execute the given number of tasks within a reasonable amount of time. The fixed budget must be used to provision resources in order to dampen the varying influx of tasks.

Formally stated, the question at hand is: “Can budget constrained cloud bursting of business workloads be efficiently handled by a combination of traditional allocation and provisioning policies?”. These business workloads are furthermore varied as well as priority and deadline-based. The workload is presumed to be unpredictable, the periodical budget is fixed and can be set by the company.

6.1.2 Approach

Allocation Policies

The system handles jobs which may consist of tasks with interdependencies. To handle this, whenever a task result arrives, all tasks that depend on this result are ‘released’ to the allocation policy. Three allocation policies are investigated, each using one internal queue to store unallocated jobs:

1. First-Come, First-Served (FCFS): this policy assigns tasks in the order in which they arrive.
2. Greedy Priority-based (Greedy): as tasks come in with a pre-assigned priority (which is updated to handle aging), a greedy policy that orders the tasks by priority and assigns the best slots for each task in order is evaluated.
3. Shortest-Job First (SJF): Pre-assigned priorities and an aging factor are processed into regular SJF to prevent tasks from staying in the system indefinitely.
4. Heterogeneous Earliest Finish Time Scheduling (HEFT): A critical works method aims to give preference to those jobs after which most computing time is still needed due to dependencies. A priori runtime estimates are used to find the critical path for each task, the length of which is combined with the pre-assigned priorities and an aging factor to prioritize the tasks.

In the allocation policies, jobs are ordered by an overall priority who’s calculation depends on the policy and consequently allocated to the node of their choosing in this order. In the allocation policies, the tasks are ordered by an overall priority, which is computed as

\[ p_{\text{overall}}(t_i) = p_{\text{initial}}(t_i) \ast f_{\text{policy}}(t_i) \]

where \( f_{\text{policy}}(t_i) \) depends on the variables the policies use. The implementation of \( p_{\text{initial}}(\cdot) \) can be found in section 5.6.

Provisioning Policies

The company that requires this system predicts that there will be trends in the usage pattern of the system. These trends are believed to be daily trends, thus implying that if the previous Tuesday a spike in the number of users occurred, then this Tuesday, in the same time...
frame there may be a spike in the number of users in the system. This provisioning furthermore has a cost limit attached to it, thus it has to be carefully selected when to lease a node or number of nodes and when to release them. The following policies are compared and evaluated:

1. On demand - When the number of tasks exceeds the number of nodes/slots that they can be assigned to, the policy decides to provision a new node.
2. On demand geometric - On successive lease cycles a geometrically increasing number of instances is provisioned.
3. On demand AIMD - The regular AIMD algorithm is used with slow start and fast recovery.

6.1.3 Previous Work

The first three allocation policies are traditional allocation policies, with small modifications to account for aging and pre-assigned priorities where needed. Critical Work Scheduling is based on the notion of critical paths, the longest path in a graph, and was implemented for frameworks with homogeneous executors in [39]. Toporkov et. al extended this idea to scalable systems with heterogeneous computing resources in [40] and follow up papers, notably [41]. Cost based estimation for provisioning has been dealt with in [42]. On demand policies and on demand geometric policies can be found in [43]. Finally, work on predictive policies can be found in [44].

6.2 Experimentation plan

6.2.1 Setup

The testing environment consists of the head node, a Cassandra database, an installation of Condor with a set of persistent cores and finally two cloud environments. The head node is run off a machine with the following specifications:

- Intel(R) Core (TM) 17-2760QM CPU @ 2.40GHz
- 8 GB DDR2 RAM
- Windows 7 64bit operating system

The cassandra installation, which is used as a database, is given a single-node cassandra server. Condor is run off the same machine running the head node. The condor client can claim all 8 cores for allocation. For the experiments, two types of cloud environments were used. The Windows Azure cloud environment was used, running Ubuntu 12.04 as the operating system. Amazon EC2 cloud environment was used, running Ubuntu 12.10 as the operating system.

Apache JMeter is an open source application that was used to simulate job submission workloads for the purposes of testing. JMeter can be used to simulate multiple users submitting jobs to the system at varying intensities. The scheduling policies were be tested for a constant number of available cloud nodes. The workload was linearly increased over a timespan of five minutes in order to stress-test the policies. This workload can be seen in figure [6]. Consequently, the different provisioning policies were tested while using a HEFT scheduling policy. The workload used can be seen in figure [7]. For the sake of comparison, one workload will be tested using the EC2 Amazon cloud service.

The workflows consist of computational tasks and are represented by directed acyclic graphs, where edges between the nodes represent a dependency between the tasks. A dependency between two tasks implies that the output of the first task is send as input to the dependent task. The workflows contain four (small), six (medium) or twenty (large) simple tasks, with up to three layers of dependencies. Test results are given for experiments run with a random combination of one small job of medium priority and two medium sized jobs; one of low and one of high priority. Ideal makespan values on the test nodes are approximately 0.75 seconds for the small jobs and 1.05 seconds for the medium sized jobs.

The metrics that will be used to evaluate the results of the experiments are:
• Makespan: the makespan metric is defined as the time it takes for a job to be executed. This is measured from the point in time that the job is submitted in the system to the point in time the job result is available.

• Queue length: the length of the queue in the allocation manager, which changes after every allocation cycle.

• Number of slots: the number of slots available in the system for the scheduler to allocate to.

6.2.2 Expected Results

It is expected that the HEFT allocation policy will perform best, as it takes the expected runtimes of a tasks dependent tasks into account. The SJF allocation policy is similar to the Greedy allocation policy, but takes runtime prediction of tasks into account as well as the pre-assigned priorities. It is therefore expected to perform better than Greedy. Finally, the first-come-first-serve allocation policy is most simple in nature, not taking pre-assigned priorities or aging into account. This simplicity predicates fast allocation and hence the queue length and the makespan are expected to be lower here than for the greedy policy. However, a larger variation in queue length is expected here.

The on-demand geometric provisioning policy is expected to perform best given the unpredictable workloads because it is expected to handle peaks swiftly. The AIMD policy uses slow start up to a threshold after which additive increase is used. This policy is expected to handle normal varying traffic well, but perform poorly at sudden peaks. The regular on-demand provisioning policy use a system load threshold to determine whether to scale up or down and is therefore stateless. It is therefore expected to perform worse than the two earlier mentioned
6.3 Results

6.3.1 Allocation policies

Two metrics were used to evaluate the allocation policies: the makespan metric, time passed spend by the job in the system, and queue length metric, number of tasks waiting to be allocated. Tests were conducted with a fixed number of nodes, seven Azure medium instances, and the workload as depicted in figure 6.

Queue Length  Figure 8 depicts the variation of queue length over time for the various allocation policies. It can be seen that the Greedy policy has the lowest maximum total queue size. However, the queue size increases very strongly, only to reach its peak because no more jobs are submitted to the system after five minutes. Other experiments show that the greedy policy is less suited for longer runs, performing slightly worse than regular FCFS and SJF overall. First come first serve allocation and shortest job first allocation behave similar to each other. Both show a more steady increase in queue size, after approximately three minutes of testing. The peaks are less steep, showing that both policies handle increasing queue sizes quite well. The result for the HEFT policy is similar, though queue size grows slower at first and then peaks higher. This can be explained by noting that this policy calculates the critical path for each task; a computationally intensive task. As queue size increases, it becomes unfeasible for the policy to keep up on the testing machine. Other experiments show that HEFT excels when small jobs are present only, but performs far worse with larger jobs. Its performance relies heavily on the amount of processing power available. Hence, though HEFT is an intelligent policy, it performs worse in practice than the SJF or FCFS policies.

![Graphs showing queue length for different allocation policies](image)

(a) First come first serve allocation  
(b) Greedy allocation  
(c) HEFT allocation  
(d) Shortest job first allocation

Fig. 8: Total queue size

Makespan  In figure 9, the results for the four allocation policies can be seen in terms of mean and standard deviation. It can be seen that the FCFS policy performs slightly better than the SJF policy and significantly better than the Greedy policy. For reasons mentioned earlier, the HEFT allocation policy performs just slightly better than greedy and worse than FCFS and SJF.
It is important to note that although FCFS performs best in this test, it does not do so in a test where pre-assigned priorities are not all the same. This can be seen in figure [10]. It can be seen that the FCFS policy yields a poorer runtime for the jobs with medium priority than for jobs with low priority, because it does not take these priorities into account. The greedy policy on the other hand, is ‘fairer’ in that the ordering for runtimes corresponds to the ordering of priorities. The SJF and HEFT policies react more or less evenly to all priorities, as they rely on other variables as well. The pre-assigned priorities can be given a higher impact for these policies.

Fig. 9: Allocation results for the four tested policies. Results expressed in mean job runtime and standard deviation of job runtimes.

Fig. 10: Performance of the different policies on the distinct test jobs, having different sizes and priorities. The job types are to be read as JOB SIZE - JOB PRIORITY. Results are normalized with respect to job size.

6.3.2 Provisioning policies

In order to evaluate the different provisioning policies one main metric was used that is number of slots. All the policies have been tested on Microsoft Azure. Furthermore, the geometric provisioning policy was tested on Amazon’s EC2 cloud service as well.
**Number of slots** The increase in slots and nodes can be seen in figure 12. The regular on-demand policy scales in a linear manner on Azure. However, due to limitations on Azure, we see that the number of slots does not increase very fast, as azure has a mechanism wherein only one provision order can be issued at a time. The other provision orders are issued as well, however these are queued up and executed when the previous order is completed. From this behaviour, it can be seen that system does not have enough of resources to handle the influx of jobs. This behaviour can be seen in the on-demand AIMD and on-demand geometric policies as well. However, in the case of the AIMD (Additive Increase Multiplicative Decrease) policy, it can be seen see that the number of nodes increases faster that the on-demand policy, then slows down and behaves like the regular on demand policy. However the on-demand geometric policy scales much faster. All policies scale only instance one at a time, due to Azure’s locking mechanism. Futhermore, all tests hit an upper barrier of 19 virtual machines as the maximum limit on azure is 20 virtual machines (one machine is always in use as a persistent machine).

![On-demand provisioning policy](image1)

![On-demand AIMD provisioning policy](image2)

![On-demand geometric provisioning policy](image3)

![On-demand Geometric provisioning policy on EC2](image4)

**Fig. 11: Number of slots**

**Comparison of Azure vs EC2** From the above section, it can be seen that the limitations of azure have a large negative impact on the performance of the system and the various policies. Also azure does not scale down in time as it is still busy leasing new nodes and thus has a lock on the cloud service preventing any other action. From these factors, it can be seen that Azure is not compatible for scalability, as it delays the action of the policy, defeating the purpose of dynamic scaling. In order to show that the system can scale down, further tests were carried out on Amazon EC2. We see here that EC2 is able to keep up with scaling up of the desired resources. We also see that the policy behaves exactly as described. This can be seen in figure 11(d).

### 6.4 Conclusion

From the tests that have been run, we see that FCFS and SJF perform the best for the test workload. The Greedy allocation policy performs slightly worse than the aforementioned two. However when it comes to handling pre-assigned priorities correspondingly, the greedy policy performs best.
The HEFT allocation policy, which was presumed to be the best for these kind of jobs, performs quite poorly. However, this is mainly due to a combination of HEFT’s demands and the system that the test is being performed on. HEFT performed even worse in tests with larger jobs, making it unwieldy except in the presence of much computing power.

The tests of the provisioning policies show that the on-demand geometric policy on Microsoft Azure scales up the fastest and is the most suitable when it comes to dealing with sudden peaks in the workload. The AIMD policy scales slower and the regular on-demand policy lags behind even more. Neither can provide enough resources in time for the system to handle large workloads. The on-demand geometric policy performs best in the EC2 cloud environment, as it is allowed to scale up and down as required, without any constraints from the cloud service itself. From this we conclude that Microsoft Azure is not suitable for dynamic scaling. EC2, however, does support scalability.

6.5 Discussion

Future experiments may be conducted into allocation and provisioning policies in the current context. Allocation policies have been tested with jobs of different sizes and priorities. In these experiments, the prioritization and runtime prediction policies were fixed. Future work may include different aging algorithms and runtime prediction algorithms different from linear regression.

Provisioning policies were tested with the Azure cloud service, with the exception of the geometric policy, which was tested on the EC2 cloud service as well. Future work may include testing more policies on both cloud service. Furthermore, the policies were tested on short workloads of approximately 25 minutes, with different patterns of traffic. For accurate testing of provisioning cost, tests of at least several hours are necessary, preferably 24 hours or more.
References


“Doxygen,” http://www.stack.nl/~dimitri/doxygen/


“Google Guice,” https://code.google.com/p/google-guice/


“Amazon EC2 API for Java,” http://aws.amazon.com/sdkforjava/


“Project Jersey,” https://jersey.java.net/

“Flot charts,” http://www.flotcharts.org/

“Bootstrap,” http://twitter.github.io/bootstrap/


“Jboss as 7,” http://www.jboss.org/jbossas


A Orientation Report

Preface

Early 2013, the four members of this team, Reinier Hartog, Richard van Heest, Vincent Hellendoorn en Anand Sawant, have begun setting up a Bachelor’s final project at Coas software systems, following their request to one of the team members. We have since discussed with the company representative, Janco Tanis, and the TU Delft supervisor, Alexandru Iosup, and settled on a project. This project, which is described in this report, has the objective to scale one of Coas systems into the cloud.

Abstract

Scalability has risen as a viable and effective alternative to owning computing resources when delivering services. Especially when the workloads vary rapidly or increase strongly overall, the ability of scaling dynamically has many advantages. In a special case, a company may have some computing resources available and desire to scale out when traffic peaks. A monthly budget may then be allocated, which needs to be used efficiently. This can be achieved through a combination of scheduling and provisioning policies. This document details the orientation report of a Bachelor of Science final project in which the above described situation is the topic of research and engineering.

A.1 Introduction

Preceding the start of the Bachelor final project, one of our group members was approach by the software development company COAS. Initially, the subject of the proposed project was workflow management. This was later changed, due to the need for research to be involved in the project, in favour of a project involving scalability and scheduling.

This document details the project assignment in chapter two, after which the approach and planning is set forth in chapter three. Chapter four describes the organization of this project and chapter five details the measures taken to assure quality.

A.2 Project assignment

A.2.1 Project environment

The company provides services in terms of both software and hardware. In one case, the company has a grid of computing resources available to which a client can submit jobs. These jobs are received by a headnode and distributed over the grid. Each node has access to a database to execute the jobs allocated to it. There are several problems with this setup. Primarily, the grid is a constant resource that must handle strongly varying traffic. Furthermore, the connection between all nodes and the central database is not desired, even less so for a scalable system.

The company also lacks insight in the traffic to and from the computing grid. This becomes problematic as it is hard to determine whether the computing grid reaches its maximum capacity and when. These are some of the main reasons for changing the current architecture. The project environment consists of the current framework, that is, the computing grid and corresponding software, which functions within a larger system.

A.2.2 Project goals

Having set forth the problems with the current situation, several changes must be made. The primary goal of this project is for the system to dynamically scale into the cloud when traffic increases. A fixed, periodical budget is available for this. The system must therefore handle its resources well, which can be achieved by both provisioning and allocating efficiently, in terms of time and cost. The corresponding quantitative goal is that the system must not spend more than the fixed budget in a certain period. Research is needed to use those policies that will use the budget efficiently.
This scalability requirement furthermore predicates the use of a scalable database. This change will be combined with the exchange of remote access (RA) from nodes to the database, to active replication (AR) of data, which is retrieved and sent by the headnode.

The company furthermore wants insight into a number of variables, such as traffic, number of nodes and remaining budget. For this, a frontend must be provided, as well as a collection of datastores in which current information about the system can be stored. Most notably, the company wants prediction for the submitted tasks and, if possible, predictability for the task runtime. In order to fulfill this requirement, the system must be able to poll jobs for their status and provide accurate predictions.

A.2.3 Assignment

The project assignment is, in short, to create a framework that will replace the current framework as described in section 2.1. The new framework must meet the requirements as set forth in section 2.2. The goals set there are the responsibility of the group.

The following are not the responsibility of the group. The framework will function within the old system. The exact specifications of for example communication or the jobs that will be send was given, hence the framework provides a general interface and the possibility to add required functionality to the system. It follows that writing job splitting policies and other specific logic is not a responsibility of the team.

The framework operates within a secure environment. It is therefore the responsibility of the team not to compromise this security. However, it is not the responsibility of the team to add a layer of security to the framework.

The framework will provide implementations for multiple cloud services as well as interfaces for tools that are used, to prevent the system from being too specific. It is not the responsibility of the team to supply implementations for specific tools, frameworks or libraries. The team may supply an implementation for a certain framework, tool or library for testing purposes.

A.2.4 Products and Services

The team is to write a framework that will replace the current framework. The product must satisfy the requirements set forth in the preceding sections. Furthermore, research will be conducted in scheduling and allocation policies to satisfy the efficiency criterion. The product may not compromise the security of the surrounding environment. The product must be able to use present computing resources and scale into the cloud, without exceeding a specified budget. The product must supply sufficient insight, as well as the possibility of submitting jobs, through a frontend. The product is not allowed to lose any jobs without reporting the loss to the frontend.

A.2.5 Requirement and limitation

Section 2.1 has detailed the project goals. Sections 2.2 and 2.4 have detailed the requirements imposed on the team and the requirements imposed on the team, the product and the project. The limitations of this project can be found in section 2.3.

A.2.6 Conditions

For the project to be completed, approximately ten weeks are needed for the current team of four. During this time, office space must be available between 9am and 17pm Monday through Friday. To accommodate travel times across the team, the team may work from the university in Delft for several days each week.

Furthermore, a registration for the Azure cloud service is needed for testing. Requirements will be established by communication with representatives from the company and a requirements analysis and architectural design report will be submitted for which feedback is expected. See section three for further details.
A.3 Approach and planning

As was stipulated in section 2.6, a period of ten weeks is needed to complete this project. In this period, the requirements will be gathered, the different components of the system will be designed, implemented and tested. The system will be divided into five main components. These components will be implemented in order, rather than iteratively.

The project will start with the requirements analysis. A period of one to two weeks is allocated to this. Conversations with representatives from the company are required as well as feedback on the report when it is completed. Consequently, an architectural design document will be written in which the implementation plan for the system is detailed. For this report too, feedback from the company is required, as the architectural design aims to prove that the requirements will be realised in the implementation of the system. For this document, a period between two and three weeks is allocated.

The next five to seven weeks are reserved for system implementation and testing. Furthermore, research will be conducted during this period after different allocation and provisioning policies in the context of this project. During the implementation of the system, which, as was said earlier, will be done on a subsystem by subsystem base, test-driven development will be applied when possible. All code written by the team must be paired with unit tests and, where complex logic is involved, acceptance tests. Finally, the last week of the project is reserved for integration testing, in which amongst other the system will be stress tested. Tests using different policies and combinations thereof will be conducted here as well.

A.4 Organization

The project team consists of four members. Each of these members will be held to the same responsibility to the project, barring unforeseen circumstances. These responsibilities include: working on the project at least eight hours a day, every week day for the duration of the project, with the exception of up to five days of leave. Group members are assumed to make up for lost time. All group members must have at least regular Java programming experience, as this is the main programming language in this project. Furthermore, second year computer science level is expected from all team members.

A continuous integration tool called Jenkins will be used to monitor code quality and progress. This tool will be connected to an SVN server on which the project code will be stored. Each group member has a laptop on which they can work on the project. Required frameworks, tools and libraries will be run from one or more of these machines. A free trial budget is available for leasing cloud nodes. When needed, the company can supply in additional funds to continue testing. As was described in section 2.6, office space will be used both at the site of the company and at Delft university of Technology.

A.5 Quality Assurance

Several measure will be taken to guarantee code quality and minimize risk. Firstly, the reports written will be discussed with the company and elaborate feedback is required. Furthermore, during implementation, regular interaction with company representatives is necessary to guarantee correctness of the system. Finally, documentation is made for every part of the system to guarantee a correct delivery of the product.

The product itself will be tested during and after implementation. Furthermore, comments will be added to clarify the codes use and purpose. Extensive logging will be added to the system to increase its maintainability.

To decrease the risks involved with this project, several measures are taken. The research involved in this project may be dropped or decreased in intensity when it appears this becomes unreachable given the workload. Furthermore, the requirements are given priorities that indicate to what extend they must be realized in the final system. Lower priority requirements may be dropped when necessary. The company furthermore limits its investment to decrease their risk.
B Architectural Design

B.1 Sequence Diagrams

B.1.1 Submit job sequence

Related system features: 5.1, 5.2
B.1.2 Receive job in scheduler sequence

Related system features : 5.2
B.1.3 Preprocess job sequence

Related system features: 5.3, 5.4, 5.5, 5.6
B.1.4 Allocation of tasks sequence

Related system features: 5.7, 5.8, 5.9, 5.10, 5.11
B.1.5 Execution of task
B.1.6 Prediction of runtime sequence

Related system features: 5.9
B.1.7 Data collection sequence

Related system features: 5.12, 5.13
B.1.8 Provisioning sequence

Related system features: 5.14, 5.15, 5.16, 5.17
B.1.9 Deployment of node sequence

Related system features : 5.18
B.1.10 Job status sequence

Related system features: 5.19
B.1.11 Retrieve result sequence

Related system features : 5.20
B.1.12 Job monitoring sequence

Related system features: 5.21
Garbage collection sequence

Related system features: 5.22, 5.23
B.1.14 Setting variables sequence

Related system features: 5.24
B.2 Interface View

The interface view diagram is depicted here.
B.3 Design View

The design view is split into six subsystems. In the interface view, the division between five subsystems can be found, as the garbage collector subsystem is not depicted in this view. This appendix furthermore includes a diagram containing all 'common' classes, which are utilized across multiple subsystems.

B.3.1 Frontend Class Diagram
B.3.3 Managers Class Diagram
B.3.7 Garbage Collection Class Diagram
B.4 Infrastructure View

The infrastructure view diagram is depicted here.
B.5 Logical View

The logical view is depicted here.