BPM in the cloud

van der Avoort, T.F.

Award date:
2013

BPM in the Cloud

Master's Thesis by Tim van der Avoort BSc
Eindhoven, February 2013

Eindhoven University of Technology
Master's programme Business Information Systems
Department Mathematics and Computer Science

Supervisors

prof. dr. ir. W.M.P. van der Aalst — Eindhoven University of Technology, Department of Mathematics and Computer Science, Architecture of Information Systems

ir. D.M.M. Schunselaar — Eindhoven University of Technology, Department of Mathematics and Computer Science, Architecture of Information Systems

dr. R.H. Mak — Eindhoven University of Technology, Department of Mathematics and Computer Science, System Architecture and Networking

Abstract

The CoSeLoG project performs a case study on the subject of configurable process models, for a group of Dutch municipalities. CoSeLoG is in need of a BPMS (Business Process Management System) that supports *multi-tenancy*, is *automatically scalable*, and supports *configurable process models*. Currently there is no BPMS available that meets these requirements. Since the requirements have strong resemblance of the typical characteristics of cloud computing, we investigated a solution in that direction. We derived an architecture to combine multiple engines of YAWL, an open source BPMS, into one cloud-based BPMS called *YAWL in the cloud*. We did so by devising algorithms for request routing, request translation, request forwarding, response translation, and response filtering. Furthermore, an approach was designed to automatically scale the cloud-based BPMS. A proof of concept implementation of this architecture was made and tested. The implementation is a working cloud-based BPMS and comes with a dashboard to monitor system status. It supports multi-tenancy, is automatically scalable and supports configurable process models. Because of the overhead of CPU and network introduced by the architecture, it does not per se perform better than a classic YAWL engine, but the YAWL cloud can use the benefit of scale. We conclude that the architecture and proof of concept implementation achieved the goals of multi-tenancy, automatic scalability and configurable process model support, and that it can be used to investigate many new and interesting follow-up research topics.

Keywords

BPM, BPMS, YAWL, cloud computing, business process, process model, multi-tenancy

Table of Contents

List of Abbreviations

In alphabetical order:

API	Application Programming Interface
AWS	Amazon Web Services
BPaaS	Business Process as a Service
BPM	Business Process Management
BPMS	Business Process Management System
BPO	Business Process Outsourcing
COBIT	Control OBjectives for Information and related Technology
CoSeLoG	Configurable Services for Local Governments
CPU	Central Processing Unit
CRM	Customer Relationship Management
c-YAWL	Configurable YAWL
GAE	Google App Engine
GUID	Globally Unique IDentifier
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IaaS	Infrastructure as a Service
IDE	Integrated Development Environment
ITIL	Information Technology Infrastructure Library
JDOM	Java Document Object Model
JSF	Java Server Faces
JSON	JavaScript Object Notation
JSP	Java Server Pages
NIST	(American) National Institute of Standards and Technology
ODBC	Open Database Connectivity
ORM	Object-Relational Mapping
PaaS	Platform as a Service
RDP	Remote Desktop Protocol
REST	Representational State Transfer
SaaS	Software as a Service
SLA	Service Level Agreement
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSH	Secure Shell
TOGAF	The Open Group Architecture Framework
URL	Uniform Resource Locator
WfMC RM	Workflow Management Coalition Reference Model
XML	eXtensible Markup Language
XPath	XML Path Language
YAWL	Yet Another Workflow Language

1 Introduction

The field of Business Process Management, or BPM for short, is aimed at modelling, analysing and enacting business processes. With the help of new modelling paradigms and powerful process engines, BPM is gaining momentum. Various BPM systems, or BPMS for short, exist to support the enactment of business processes. Often such systems include extra tools like a process model definition editor.

Many organisations have processes that are similar among a group of them. Configurable process models allow to capture models of such processes, including the variability among organisations, in a single shared configurable process model. Main benefits of using configurable process models are: it eliminates redundancies in the group of similar process models, it fosters standardisation and reuse of proven practices, and it enables clear distinction between commonalities and variability [1].

1.1 Problem Description

The CoSeLoG project [2] is a research project that, among others, performs a case study on the subject of configurable process models. The target group of organisations investigated by the CoSeLoG team are Dutch municipalities. Frequent processes within the municipalities, such as handling permits and taxes are often, to some extent, similar among different municipalities. Configurable process models have been composed to describe both the commonalities and the variability of these classes of frequent processes. When selecting a BPMS for enacting the process models derived during the case study, the following difficulties were experienced by the CoSeLoG team:

- Most currently available BPMS are not *multi-tenant*. Single-tenant BPMS typically are provided as installable software for the on-premise or external co-located data centre of the customer. As a result, every organisation then has to host or co-locate their own installation of the BPMS. This requires knowledge and costs a significant upfront investment. Software updates, patches or hardware maintenance all have to be carried out for each individual installation.

- Most currently available BPMS are not scalable or not *automatically scalable*. BPMS that are not scalable put scale limits on what the customer can do using the system. BPMS that are scalable but not automatically scalable do not have these limits, but have a fixed amount of available resources, that does not match the variability of the actual usage of those resources.

- Most currently available BPMS do not support *configurable process models*.

The goal of this project is to overcome the difficulties experienced by the CoSeLoG team, by developing a proof of concept BPMS architecture and implementation that supports multi-tenancy, is automatically scalable, and supports configurable process models. The CoSeLoG team set the constraint that the internal working of the BPMS used in the architecture and implementation may not be changed, thus may only be used as-is.

1.2 Proposed Solution

There are, as concluded in the previous Section, three major requirements for the solution: supporting multi-tenancy, supporting automatic scalability, and supporting configurable process models. Both the requirements of multi-tenancy and automatic scalability are typical properties of cloud computing based solutions. Because of this strong resemblance, we search for a solution in the direction of cloud computing.

Context of Cloud Computing

The relatively recent concept of cloud computing aims at delivering computing as a utility service, just like water and electricity. Whether it be hardware infrastructure, computing platform or software, the customer can consume them without having to worry about the underlying foundations, since they will be taken care of by the provider. Some other benefits of using cloud based solutions are: one can save on upfront investment and move to a pay-per-use model, these solutions tend to scale up and down very well, resources can be used more efficiently because they are shared, and changes in circumstances or demand can often easily be acted upon. The major obstacle for cloud computing lies in fear for security and control issues.

Interest in cloud computing and usage of software provided from the cloud is on the rise. Google Apps [3] for e-mail and Salesforce [4] for CRM are just two examples of such products that are widely used and accepted as full alternative for classic in-house software. Although for many applications cloud solutions are commonplace, this is not the case for BPM yet. Some vendors are developing cloud based versions of their software. However, their use is not yet widespread. By conducting a survey, Gartner found that “40% of organizations with BPM initiatives use cloud services to support at least 10% of their business processes within those BPM projects” [5].

Requirement Coverage of Cloud Computing

Choosing to develop a cloud based solution covers the following requirements:

- Cloud solutions support *multi-tenancy* (by definition, see 2.2.1 Section Resource pooling).
- Cloud solutions are *automatically scalable* (by definition, see 2.2.1 Section Rapid elasticity).

Besides covering these requirements, using cloud computing has the following positive impact:

- When consuming a service from the cloud, the cloud provider is managing the service. Thus, when providing a BPMS from the cloud, an individual organisation has no need to have its own private installation. Since cloud services are typically provided as pay-per-use, no significant upfront investment is needed.

- Cloud solutions are not be bound by the resources of a single server. The architectures include both the sharing of a server by multiple tenants, and the spreading of multiple or a single tenant over multiple servers.

Mechanisms have to be developed to spread the load over multiple systems. These requirements are elaborated on, in Chapter 3.

Figure 1 shows, on a very high level, a before and after situation when moving BPM to the cloud. No local BPMS installation per organisation is needed anymore.

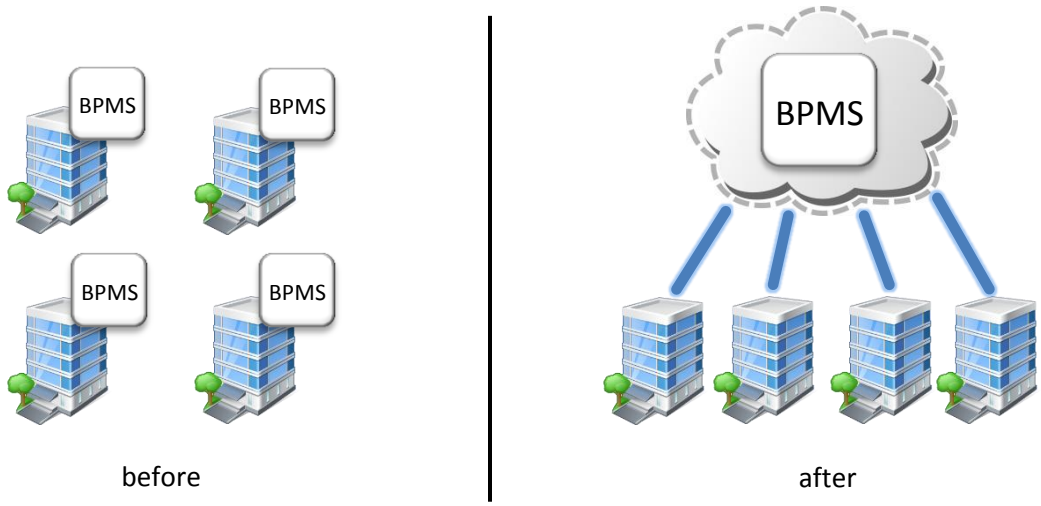


Figure 1: BPM in the cloud, before and after

Context of Configurable Process Models

The third requirement that has to be met is that the solution has to support configurable process models. Some BPMS have support for configurable process models built-in or by standalone tools. By choosing to make a cloud based version of a BPMS that already supports configurable models, we can cover the third requirement.

YAWL (Yet Another Workflow Language) is a workflow modelling language and an accompanying support environment, in the form of an open source BPMS. Unless stated otherwise, in this report the acronym YAWL will always refer to software, i.e. the YAWL BPMS. A configurable version of YAWL models exists, called c-YAWL. When combining a c-YAWL model with a valid configuration, a normal YAWL model can be generated. The configurable process models created for CoSeLoG are already modelled in the c-YAWL format. Therefore, we choose YAWL as the BPMS to be used for this project. We recall the constraint set in Section 1.1: the YAWL engine must be used as-is; its internal workings may not be changed.

Requirement Coverage of YAWL as chosen BPMS

YAWL has configurable process model support. Configurable process model support can strengthen the multi-tenancy benefits of cloud computing, because it has the following impact:

- When a group of organisations share a single configurable process model, both the generic model and the individual configuration can be stored. Groups of organisations that share processes are called communities. When an organisation wants to implement a specific configurable process model that is shared in its community, that organisation only needs to provide its individual model configuration.

Overall Project Goal

This leads to the following concrete goal for this project: Develop a cloud based version of YAWL, to be called *YAWL in the cloud*. YAWL in the cloud must be multi-tenant, automatically scalable, and facilitate the sharing of configurable process models among organisations.

1.3 Chapter Structure

The next Chapter (*Preliminaries*) describes BPM, YAWL and cloud computing for readers who are not yet familiar with one or more of these concepts or who want to refresh their knowledge.

Whereas the concise *Problem description* and *Proposed solution* Sections give a general high-level goal of this project, Chapter 3 (*Towards a YAWL Cloud*) sets specific goals for multi-tenancy when bringing the YAWL environment into the cloud.

In Chapter 4 (*Architecture*) an architecture is developed that enables the goals set in Chapter 3 by describing the build-up of the architecture step by step. Starting with a high-level system architecture, we go towards a more fine-grained component description of all the components included in the architecture.

An implementation of the architecture is made as a proof of concept. Chapter 5 (*Proof of Concept Implementation*) describes this implementation and shows some exemplary screenshots.

Finally, Chapter 6 (*Evaluation*) will compare the new cloud-based YAWL environment with its non-cloud-based predecessor. Did the cloud-based YAWL environment bring us what it was supposed to bring? Is *YAWL in the cloud* the BPMS that the CoSeLoG team is in need of?

2 Preliminaries

The project *BPM in the Cloud* combines two topics: *BPM* and *cloud computing*. To gain a good understanding of the project and this report, it is essential that the reader has some knowledge of on the one hand *BPM* and *BPM systems*, in particular *YAWL*, and on the other hand *cloud computing*.

Some familiarity of the reader with BPM is assumed. The acronym BPM can stand for both *Business Process Management* and Business Process Modelling. In the context of this report, the third letter of the acronym always stands for *Management*. We choose to follow the definition of BPM, as published in [6]:

"Supporting business processes using methods, techniques, and software to design, enact, control, and analyse operational processes involving humans, organizations, applications, documents and other sources of information." (2003)

The following Sections explain the aspects of YAWL and cloud computing that are important in this project.

2.1 Introduction to YAWL

Various systems exist to support BPM, also called BPM systems or BPMS for short. *YAWL (Yet Another Workflow Language)* is one of these systems. It consists of a process modelling language, and an accompanying support environment. Unless stated otherwise, in this report the acronym YAWL will always refer to software, i.e. the YAWL BPMS (the support environment of the YAWL modelling language).

According to [7], YAWL was developed without the pressures of vested interests. This led to some interesting distinguishing features. The YAWL modelling language is powerful: it has comprehensive support for the workflow patterns identified by the Workflow Patterns initiative [8]. Furthermore, it has a formal foundation, which by construction allows for automated verification and results in unambiguous specifications.

2.1.1 Architecture

YAWL is composed of various components. Figure 2 shows the internal architecture of YAWL. The most important part of this architecture, especially for this project, is the YAWL engine highlighted in blue. The depicted interfaces (A, B, E, X, R, O and W) are all REST (REpresentational State Transfer [9]) interfaces. All interfaces have several *actions* that can be performed by them. Some more details on the several interfaces can be found in Section 2.1.4.

The *YAWL engine* handles process enactment. The *process designer* is a standalone application for creating YAWL process models. The *resource service* adds the human resource perspective, and is the default web interface a user will typically see. The *web service invoker* can, as the name suggests, be used to invoke other web services, for example using SOAP (Simple Object Access Protocol). The *worklet service* adds sub-process and process exception handling functionality. More details on the YAWL architecture can be found in various Chapters of [10].

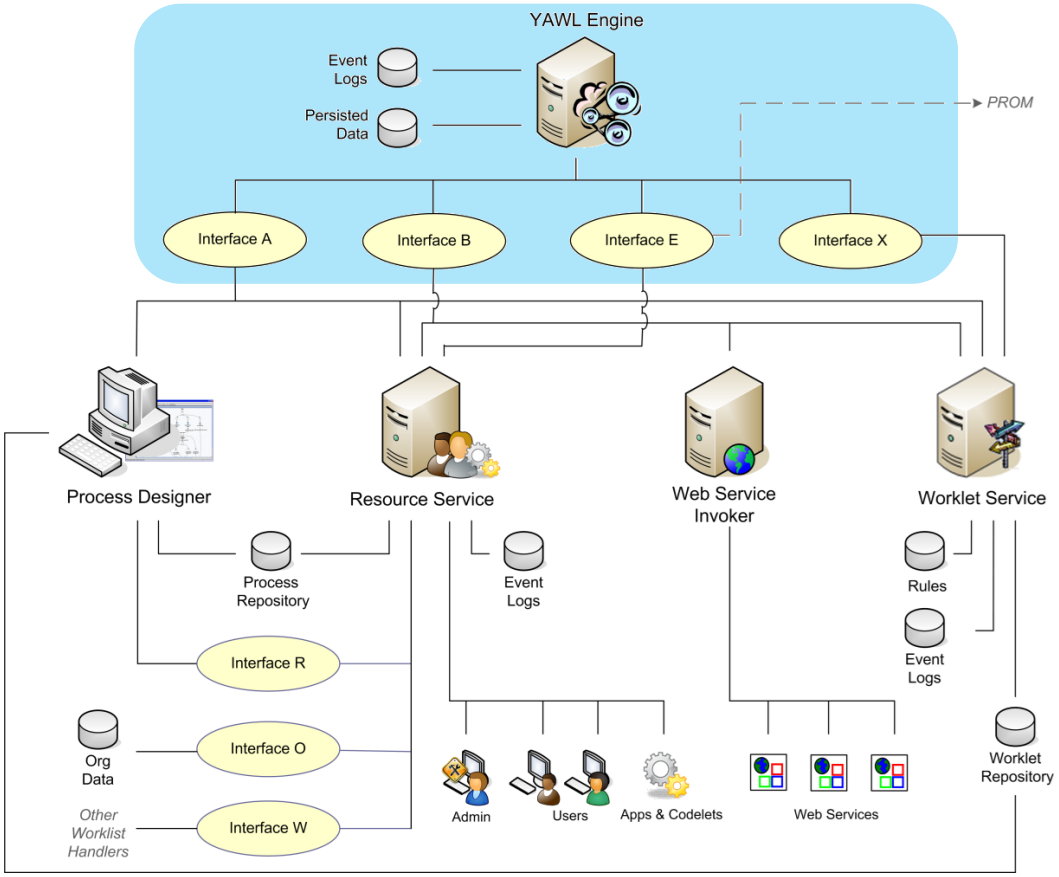


Figure 2: Architecture of YAWL

Because the YAWL engine is frequently depicted in this report, we use the simplified graphical representation of the YAWL engine as depicted in Figure 3.

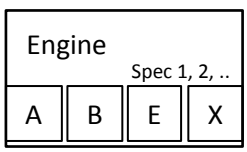


Figure 3: Simplified graphical representation of the YAWL engine

2.1.2 Web Interface

A normal end-user does not manually interact with the REST interfaces, but accesses a web interface instead. Figure 4 shows a screenshot of the default web interface an end-user of YAWL can see. It is important to note, referring to Figure 2, that this web interface is part of the *resource service* and is outside of the YAWL engine.

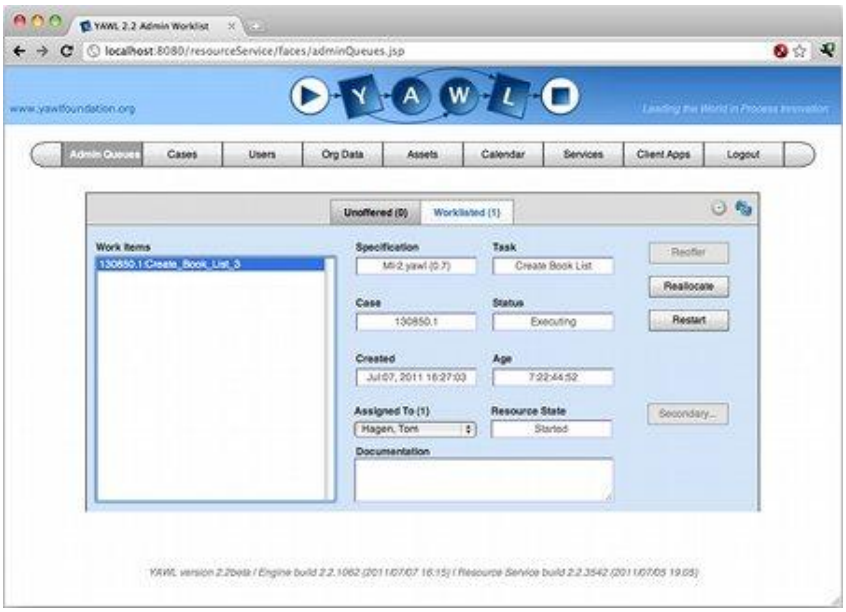


Figure 4: YAWL web interface screenshot (Resource Service) (source http://www.yawlfoundation.org/)

2.1.3 Relation to the BPM Life-Cycle

A common way to describe BPM is by using the cycle of *diagnosis*, *process (re)design and analysis*, *system configuration* and *process enactment and monitoring* called the *BPM life-cycle*. Slightly different variants exist, including more detailed steps. The essence of all of the variants is that BPM projects should never follow a single-execution approach, but rather be seen as a cycle.

Figure 5 shows a graphical representation of the BPM life-cycle. To put YAWL into perspective, its place in the model is highlighted in blue: YAWL is dealing with process enactment and monitoring.

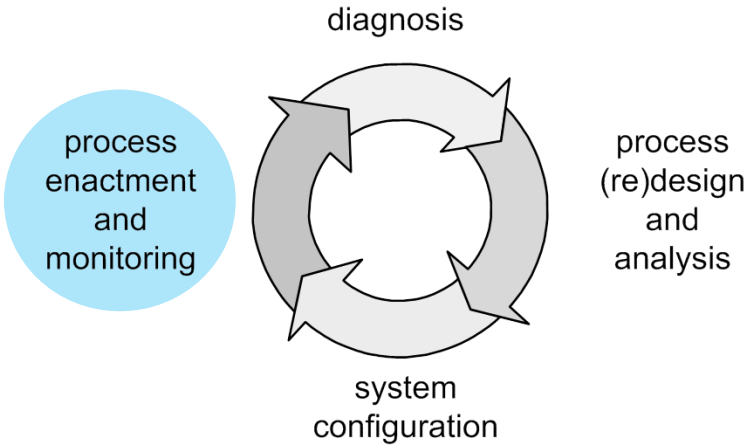


Figure 5: BPM life-cycle (source [7])

2.1.4 Relation to the WfMC Reference Model

A common way for describing interfaces of a BPMS is relating them to the Workflow Management Coalition reference model, or WfMC RM for short [11]. This model defines 5 types of interfaces, all used for different kinds of interaction. Details about the actions on the various interfaces can be found in Appendix B.

To put the interfaces of the YAWL engine into context, we relate them to the interfaces in the WfMC RM. The interfaces of YAWL do not directly match the interfaces in the WfMC RM. Figure 6 shows a mapping of the interfaces of the YAWL engine onto the WfMC RM. The mapping is as follows:

- *Interface A* of the YAWL engine is used for uploading and unloading specifications, and primarily maps to *Interface 1*, and for a small portion to *Interface 5* of the WfMC RM.
- *Interface B* of the YAWL engine is used for performing many actions related to process execution, such as actions on process instances and work items. Interface B has, by far, the most actions of all interfaces of the YAWL engine. Due to its comprehensiveness, it maps to *Interface 2*, *Interface 3*, and *Interface 4* of the WfMC RM.
- *Interface E* of the YAWL engine is used for process log interaction. It therefore maps to *Interface 5* of the WfMC RM.
- *Interface X* of the YAWL engine is used for interaction on process-level exceptions. There is no equivalent for this in the WfMC RM.

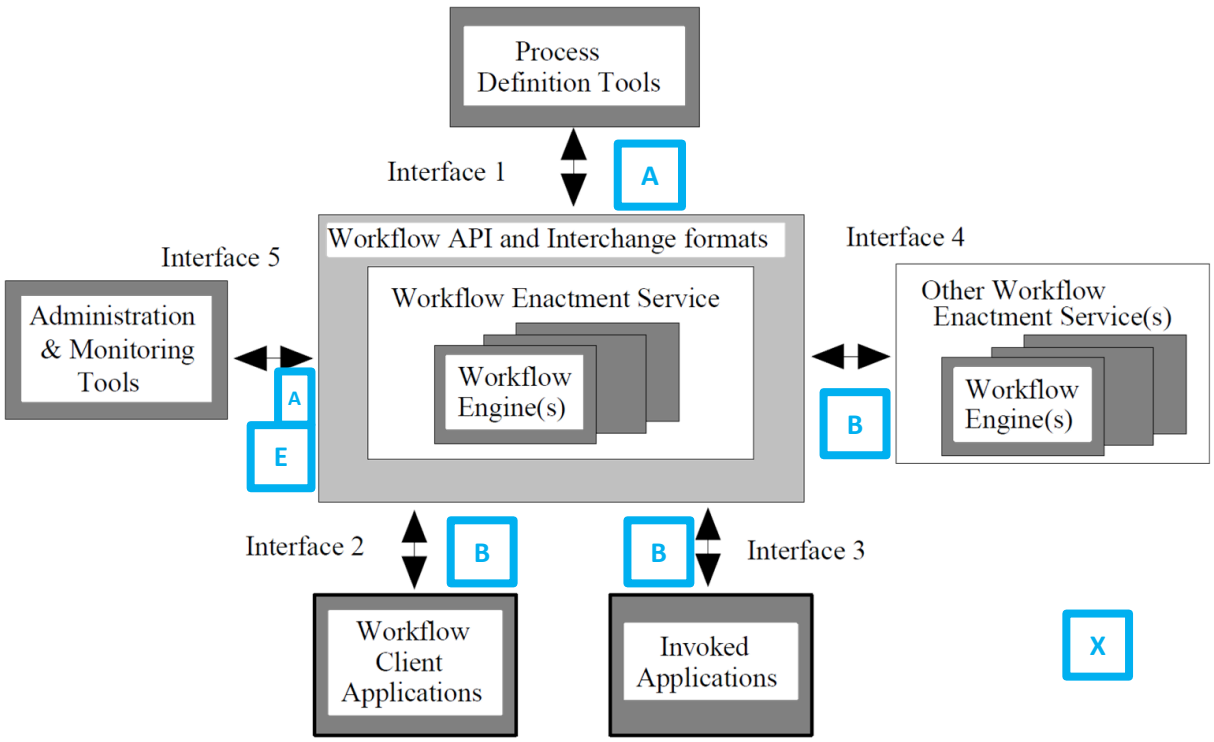


Figure 6: WfMC reference model (source [11]), enriched with the YAWL interfaces

2.2 Introduction to Cloud Computing

Various definitions of *cloud computing* exist. Using many definitions proposed by experts, "A Break in the Clouds: Towards a Cloud Definition" [12] proposed the following encompassing definition of the cloud:

"Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs." (2009)

Another definition of cloud computing is published by the American *National Institute of Standards and Technology* (NIST) [13]:

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models." (2011)

These definitions, and their more detailed descriptions, have a significant overlap between them. The NIST definition is more recent, and appears more complete. Furthermore, the NIST definition comes closest to how we experienced cloud computing, how providers appear to describe their services, and how consumers express their demands. Therefore, the NIST definition will be used as guidance for the first three subsections of this introduction: *Essential Characteristics*, *Service Models* and *Deployment Models*. These Sections are followed by Sections on *Benefits and drawbacks* of the cloud, the cloud *Ecosystem*, and the *History and future* of the cloud.

Figure 7 shows a graphical summary of the NIST definition:

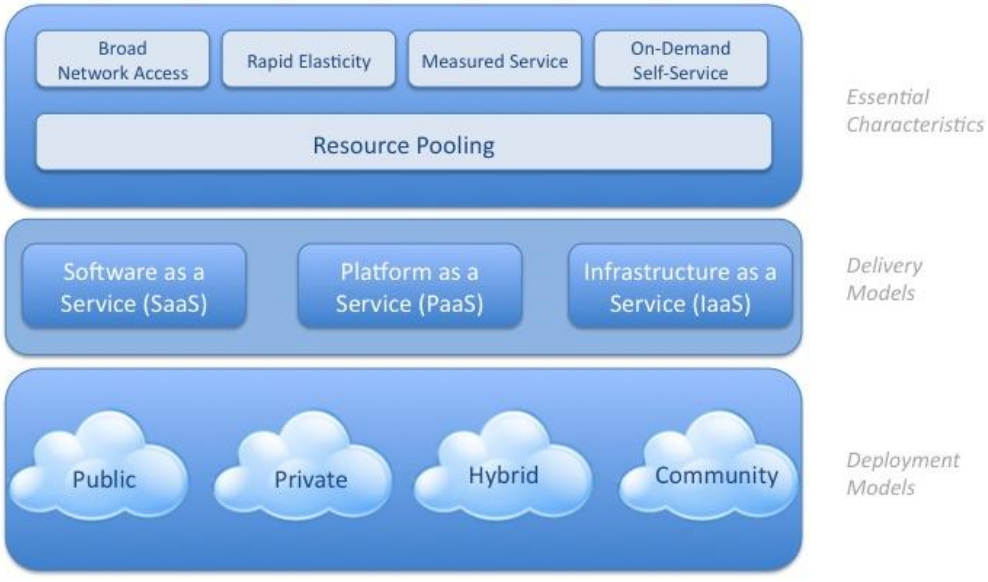


Figure 7: Visual Model of the NIST definition of cloud computing [13], as depicted in [14]

2.2.1 Essential Characteristics

NIST defines five essential characteristics of cloud computing, which are: *on-demand self-service*, *broad network access*, *resource pooling*, *rapid elasticity* and *measured service* [13]. Following these names, the following Sections explain their meaning. The individual characteristics are not unique for cloud computing. It is the combination of characteristics that is typical for cloud computing.

On-demand Self-service

The consumed services, such as servers and storage, can be increased or decreased on-demand. Consumers can do so themselves, without any required human interaction from the side of the provider. Human interaction from the consumers side is possible using web interfaces. Additional services can be put into action at a click of a button. However, human interaction on the consumers side is not required, since most providers offer *APIs* (Application Programming Interfaces) to automate the self-service process.

Broad Network Access

The consumed resources are accessible over a network. Communication with these resources follows standard mechanisms, or, protocols. Frequently used protocols to access resources in a cloud computing environment include: basic *HTTP* (HyperText Transfer Protocol), *REST* (Representational State Transfer), *SOAP* (Simple Object Access Protocol), *SSH* (Secure Shell) and *RDP* (Remote Desktop Protocol). Depending on the *deployment model* (see 2.2.3) the network can be the internet, or any other network, such as a company's internal network.

The NIST definition states that broad network access using defined standards promotes use by heterogeneous thin or thick client platforms, such as mobile phones, tablets, laptops and workstations. Although their phrasing is very specific for the *SaaS service model*, one can generally say that the use of defined standards increases interoperability.

Resource Pooling

In cloud computing, resources are consumed by multiple consumers. In other words, the provided service is *multi-tenant*. The offered resources are very diverse, and can range from (relatively) low level resources such as computing time, data storage and network load balancers to high level resources such as managed database hosting and fully fledged CRM (Customer Relationship Management) systems.

The provider's resources are pooled, i.e. they are used to serve multiple consumers. Depending on the *deployment model* (see 2.2.3) the group of consumers served from the same pool of resources can be smaller or larger. When resources are requested by consumer demand, vacant resources in the pool are dynamically assigned to the consumers.

The consumer has no control and knowledge over which physical machine or location they will be served from. Typically, consumers do have some control over higher level characteristics. Common options for this are: continent, country, state, nearby city, or datacentre.

Rapid Elasticity

The consumed resources can easily be scaled up as well as down. Depending on the type of resource, scaling can be done on one individual resource (e.g. increasing the processor speed of one virtual server), by adding additional resources (e.g. increasing the number of virtual servers), or by both. In some cases this scaling can be performed automatically, e.g. depending on time of day or defined usage metrics.

From the consumers viewpoint, the amount of resources available for consumption can be assumed to be virtually unlimited. Without any prior announcement any resource can be consumed in any quantity at any time. Some providers have discounted price plans for long term committed consumption of a specific resource (e.g. a one-year reserved virtual server).

Measured Service

Providers measure the usage of resources by individual consumers. Consumers are charged based on these measurements. Depending on the type of resource, appropriate metrics such as storage size, processing time, bandwidth usage, number of registered users, and combinations of such metrics are used.

Usually providers offer reporting and monitoring functionality, so that a consumer can see and analyse its resource consumption. Most consumers try to optimise their resource usage, in order to optimise cost-effectiveness. As mentioned before, with some providers, consumers get discounted prices for resources they commit to consume long-term.

2.2.2 Service Models

The cloud computing service models define what type of service is provided to the consumer. Different types of service have different levels of abstraction. Many providers offer services from multiple service models.

NIST defines three service models for cloud computing, which are *Infrastructure as a Service*, *Platform as a Service* and *Software as a Service*. These are also the three service models that are ubiquitous in cloud computing terminology, and are adhered to by many sources [12] [15] [16] [17].

Some providers name new models, such as *Business Process as a Service*. The various service models are not necessarily disjoint; whereas some resources are clearly in one of the service models, others can be debated to be in two adjacent service models. For example, managed database hosting can be seen as *Software as a Service* as well as *Platform as a Service*, depending on the consumers viewpoint.

All service models can be combined with all deployment models.

IaaS (Infrastructure as a Service)

The resources offered in the IaaS service model are mostly virtual versions of physical resources, such as storage space, virtual servers, virtual load balancers and network connections.

Consumers can abstract from having a physical infrastructure and instead can consume it as resources. The consumer has great freedom of what she can do with the resources. Considering e.g. *virtual servers*, one can use them to run almost any type of operating system, on which on its turn one can install any desired software.

The consumer cannot control the physical cloud infrastructure. Whereas it is often possible to select, up to some granularity, the location of the datacentre, it is not possible to control or demand any specific physical device.

PaaS (Platform as a Service)

The level of abstraction of PaaS is higher than that of IaaS. PaaS abstracts from software such as operating systems up to the level that is needed to run application software. The resources offered in this service model are platforms that can be used to deploy software onto.

Consumers can deploy compatible software to the platform. The software needs to be written in a format that is supported by the provider, e.g. be written for a specific runtime environment, programming language or library. In general, the platform will automatically scale up and down the needed cloud infrastructure, with no need of consumer interaction.

The consumer can neither control the cloud infrastructure, nor the operating system software. Although the cloud infrastructure cannot be controlled directly, often some settings are provided for e.g. the level of over-provisioning (performance) or the version number of the platform it is intended to run on.

SaaS (Software as a Service)

The level of abstraction of SaaS is higher than that of PaaS and IaaS. SaaS abstracts from software, in the sense that one doesn't have to buy, install or maintain it. The resources offered in this service model are complete software packages, including their deployment on a platform on the cloud infrastructure.

Consumers use software from the cloud. Examples are e-mail suites, office suites, CRM systems, book keeping systems and online storage synchronisation services. The software that can be consumed is not limited to end-user software, but can also be intermediate software products such as an API providing a service.

The consumer cannot control the platform the software runs on and the underlying infrastructure. Control is typically limited to application configuration settings.

BPaaS? (Business Process as a Service)

BPaaS combines the concepts of *Business Process Outsourcing* (BPO) and *SaaS*. Some reports, such as "The Evolution Of Cloud Computing Markets" by Forrester Research [18], suggest that BPaaS should be seen as a fourth cloud computing service model, with a level of abstraction that is higher than that of SaaS. However, the concept of BPaaS is not widespread practice, nor scientifically well documented yet.

2.2.3 Deployment Models

Deployment models are another dimension to describe a cloud service. Where service models describe what kind of cloud service you provide or consume, deployment models describe the level of ownership, and thus control, you have over a cloud service.

All deployment models can be combined with all service models.

Public Cloud

The public cloud is a cloud infrastructure to be consumed by the general public. Any organisation or private person can consume resources from a public cloud. In general, a public cloud is available over a public network: the internet. Most well-known cloud providers, such as Amazon AWS, Google App Engine, Microsoft Windows Azure and Rackspace, primarily offer public cloud services. The cloud infrastructure resides off-premises, in the providers datacentre.

Private Cloud

The private cloud is a cloud infrastructure that is consumed by a single organisation. The organisation can build the cloud infrastructure by itself, or let a third party do so. Using a private cloud can be considered when the benefits of cloud computing are desired within an organisation, but for some reason it is not desirable to use a public cloud. Reasons can be, for example, laws that prohibit the use of public clouds for certain usage, or organisations of such large scale that a private cloud is more cost-effective.

Community Cloud

The community cloud is a cloud that is consumed by a specific group of organisations that have something in common: a community. A community can be e.g. a group of organisations of the same type or a group of organisations that have the same security policies.

For example, the municipalities considered in the CoSeLoG project can be seen as a community. They are the same type of organisations, have to adhere to the same laws and have the same goals. If, for some reason, it is more desirable for them to have a cloud that is more private than the public cloud, they can consider to start their own community cloud.

Hybrid Cloud

The hybrid cloud is a combination of any two or more distinct cloud providers. These providers can be of the same or a different deployment model. Resources are consumed from the distinct cloud providers and are combined into a cloud infrastructure by using technology that enables data and application portability.

Usage scenarios include using a second cloud provider to handle overload of a first (e.g. private) cloud provider, or using multiple cloud providers to reduce the dependence on any single one of them.

One of the technologies that enables hybrid clouds is known by the name *intercloud*. Initiatives exist to unify the APIs of several cloud providers, in order to create one standardised API. When developing for that API, one can easily choose the cloud provider(s) to deploy to on-the-fly.

2.2.4 Benefits and Drawbacks

It are the benefits of cloud computing that contributed to major business interest in cloud computing. Using the cloud services makes room for new or improved business opportunities. The following benefits are frequently linked to cloud computing:

- Technical flexibility: on demand scalability and appearance of infinite resources. [17] [19]
- Financial flexibility: pay per use, little upfront investment. [19] [20]
- Short time-to-market. [17]
- Cost savings. [17]
- Some strengths for the IT-Governance frameworks COBIT, TOGAF and ITIL. [21]

There also are some drawbacks of cloud computing. The international character, the confidentiality of data, and limited control about the consumed services leads to fear for the risks, threats and vulnerabilities that may come with it. The following drawbacks are frequently linked to cloud computing:

- Technical cloud service provider related risks: e.g. data security, data leakage, resource sharing isolation problems. [14] [15] [22]
- Data lock-in. [14] [19] [20]
- Confidentiality and auditability. [14] [17] [19] [20] [22]
- Business continuity and service availability. [19] [20] [22]
- Data transfer bottlenecks. [19] [20]
- Performance unpredictability, due to virtual shared resources. [19] [20]
- Social cloud service provider and consumer related, e.g. malicious insiders. [15]
- Legal risks, because of country specific rules and regulations for e.g. privacy. [14] [15] [22]
- Some weaknesses for the IT-Governance frameworks COBIT and ITIL. [21]

Although a (business) assessment on the benefits and drawbacks of using the cloud for BPM is an interesting topic, it is outside the scope of the current project to make up this balance. Interested readers are referred to [21], which concludes that, from an IT management point of view, cloud computing is likely to have more benefits than drawbacks. Furthermore, it contains a cost estimation approach.

2.2.5 Ecosystem

With the rise of cloud computing came, of course, the rise of cloud providers. Some big names that cannot be overlooked are:

- *Amazon Web Services* (AWS), providing IaaS and PaaS services, of which Amazon EC2 for computing time, and Amazon S3 for storage, are most well-known.
- *Google*, providing PaaS services with Google App Engine. With Google Apps it provides a SaaS product; a full e-mail, calendar and documents office suite.
- *Microsoft*, with Windows Azure, offering PaaS services.
- Many other IaaS and PaaS providers exist, such as *Rackspace*, and *Go Grid*.
- Many other SaaS providers exist, such as *Salesforce.com* [4].
- Some intercloud initiatives exist, whose goal is to make cloud computing IaaS-provider independent, such as *RedHat* with its *DeltaCloud*.

For a larger list of providers and more details on the services they offer, see Appendix A.

2.2.6 History and Future

One can say that the essence of cloud computing is providing computing as a public utility. Some of the properties that cloud computing and public utility services have in common are: providing a service that abstracts from the technology behind it, measuring the service to analyse and charge, consuming from a shared pool, and self-servicing. The idea of providing computing as a public utility is not new. Even in the early 1960s, way before the internet era, scientists expressed these desires [19] [23] [24].

The word *cloud* refers to the internet. It is derived from past diagram notation for telephone and computer networks, in which clouds were drawn for parts of the network whose details were unknown.

Figure 8 shows the Google trends for the search term “cloud computing”. Google trends diagrams show the relative frequency of search terms over time. The horizontal axis shows the years, the vertical axis shows the Search Volume index (the relative frequency of searches for “cloud computing” over time, with the average over the entire timespan normalized to value 1.00). From this picture we can see that interest in cloud computing has been on the rise significantly since late 2007.

The adoption of cloud computing is expected to increase significantly over the coming years, as is its maturity [18] [20] [25].

Figure 8: Google trends of cloud computing

3 Towards a YAWL Cloud

In Chapter 1 (Introduction) we've seen Figure 1, showing a before and after picture of BPM in the cloud. However, for achieving hands-on experience with brining BPM to the cloud, we cannot avoid choosing a specific system to experiment with, for which we chose YAWL. This gives us Figure 9, which is, seeing Figure 1 as the generic goal, an instantiation of this goal for YAWL.

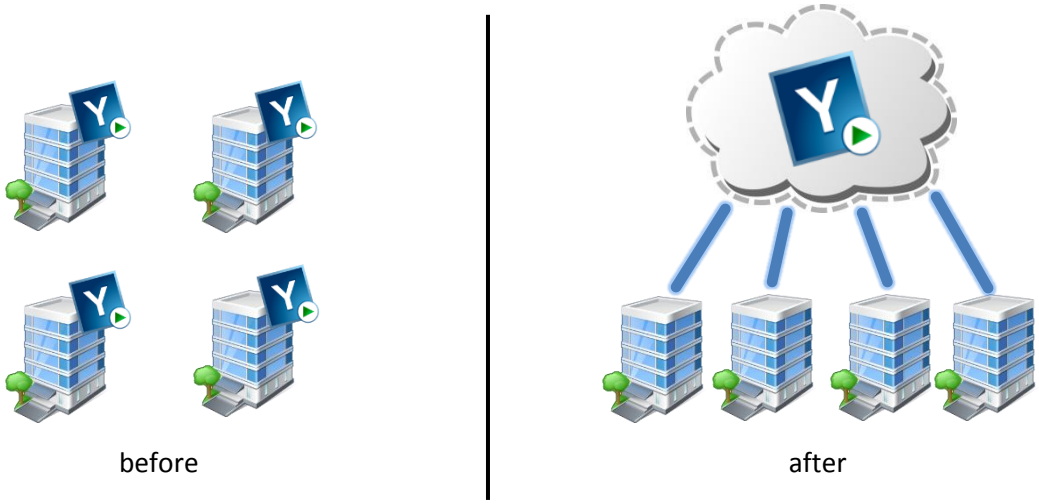


Figure 9: YAWL in the Cloud, before and after

Moving from the generic goal to a specific goal comes with a risk. Will the solution, specifically designed for YAWL, be of any use for the generic situation? During the design, assumptions under which the YAWL-specific solution is valid for other systems will be stated explicitly, to mitigate this risk. The outcome of this question can be found in Section 4.5.

3.1 Engines and Servers

Figure 10 shows some combinations of engines and servers. The YAWL engine runs on a server. To be precise, it runs as a Java Web Application in an Apache Tomcat container.

From left to right, we first see a server running one engine, which is the default deployment for classic YAWL. Next, we see a server running two separate engines. When configured correctly (the engines should run on different ports) this also is a working deployment using classic YAWL. In practice, the two engines will be completely independent deployments.

The situation on the far right, however, is impossible. One single engine, keeping YAWL as-is, cannot be run spread over two servers. We accept this limitation, and continue to work under the assumption that one engine runs on one server, and one server can run multiple engines.

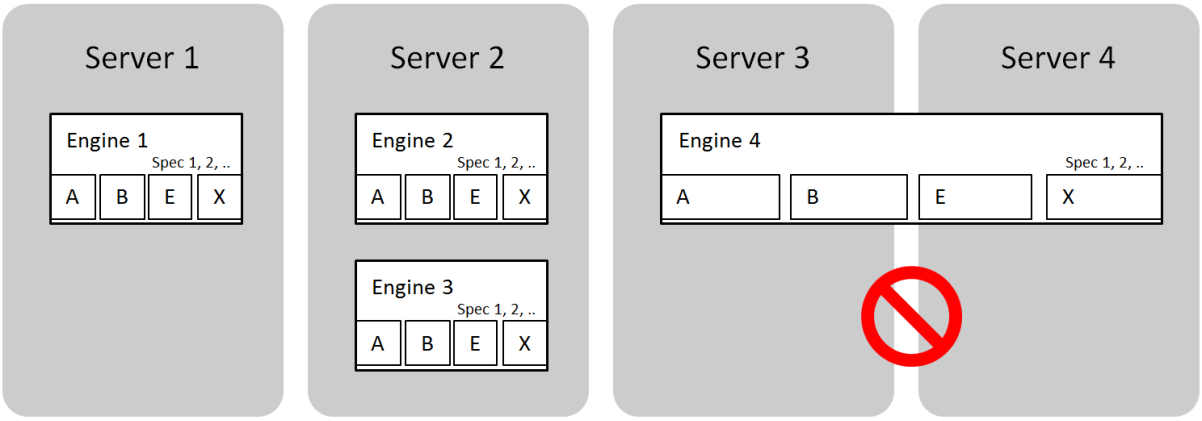


Figure 10: Engine placement on servers

3.2 Multi-tenancy - Current Situation

Multi-tenancy is the term used when one architecture (the YAWL cloud) is shared by multiple tenants (the YAWL cloud consumers). Classic YAWL deployments typically have one YAWL engine and one tenant. Figure 11 depicts tenancy for classic YAWL deployments in three ways. From left to right, we can first of all say that the tenant owns the server on which the engine runs. Alternatively, we can say that the tenant owns an engine, which is run on a server. Even more fine-grained, we can say that an engine runs on a server, and the engine serves one tenant.

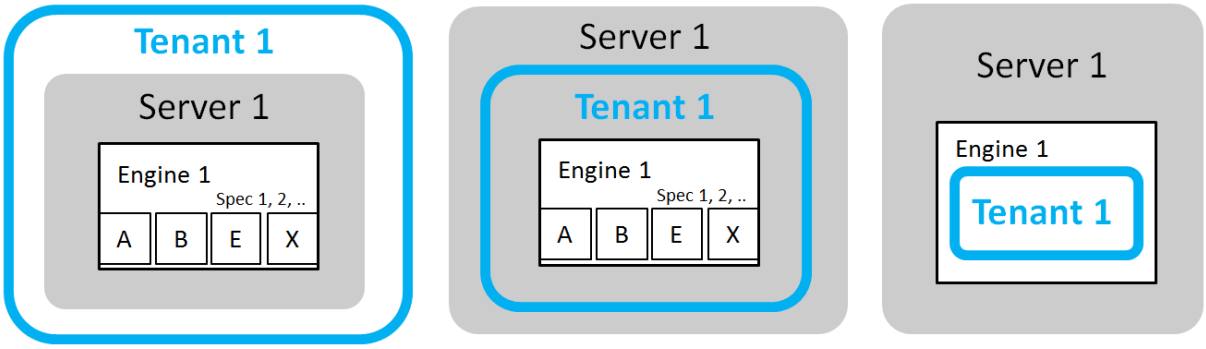


Figure 11: Three ways of depicting tenancy in classic YAWL deployments

If multiple servers are available, one can implement multi-tenancy of the combined architecture by deploying one server per tenant. Using the possibilities set in Figure 10, plus the tenancy notation of Figure 11, there is one only slightly more flexible form of multi-tenancy, using YAWL as-is, as depicted in Figure 12. Compared to the one server per tenant situation, the only extra degree of freedom for multi-tenancy in this situation is that, when configured correctly, it does not matter if you run one or more tenants on one server.

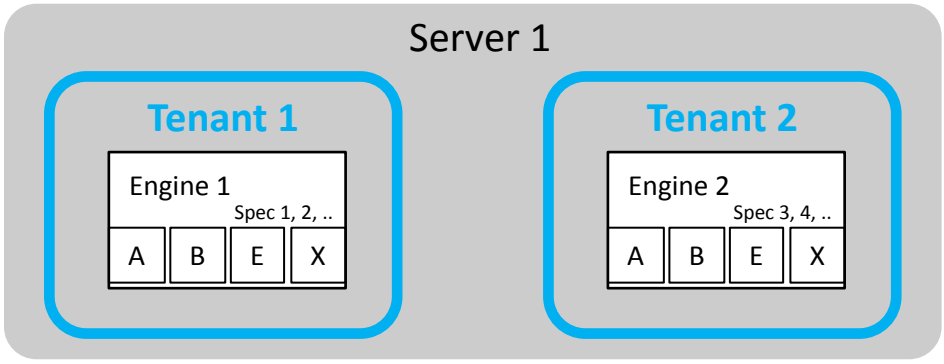


Figure 12: The only form of multi-tenancy using classic YAWL

3.3 Multi-tenancy - Desired Situation

The forms of multi-tenancy that are currently possible are, as seen in Section 3.2, very restrictive. For YAWL in the cloud, we want to provide more flexible combinations of tenants and engines.

In particular, there are two additional forms we want to support. Figure 13 shows these two forms. First of all, as seen on the left, we want to support serving more than one tenant from a single engine. Second, as seen on the right, we want to support serving one tenant from multiple engines. If both these forms can be achieved, more flexible combinations of tenants and engines are possible.

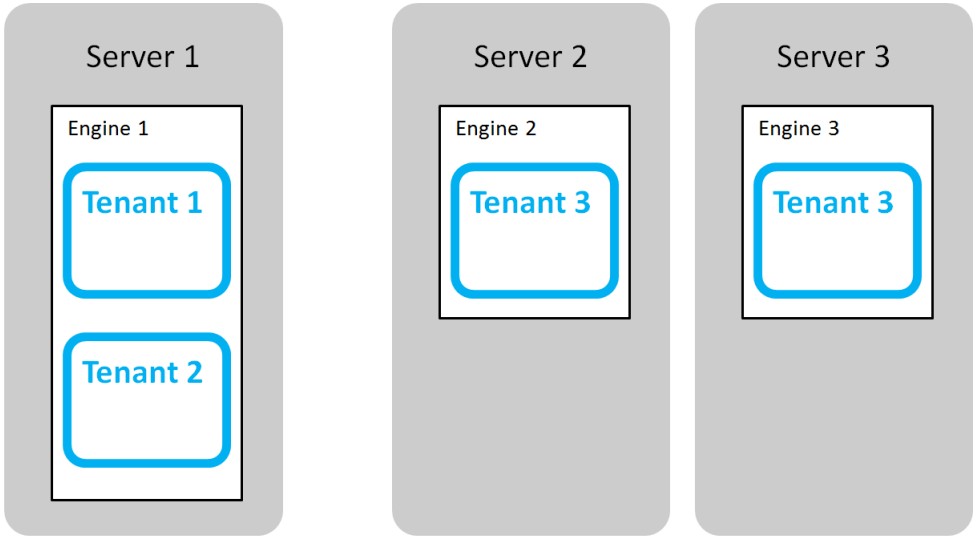


Figure 13: Two new types of combining engines and tenants

3.4 Allocation of Specifications and Cases

When looking at the distribution of tenants over YAWL engines, we can add two more levels of detail towards a more fine-grained model. The first step is including the *specifications* that run on an engine, the second step is including the *cases* of a specification that run on an engine. As we know from Section 3.2, in the current situation all cases of all specifications of one tenant are all running in one and the same engine. We introduce two forms of spreading the work when using more than one engine, that are not possible using classic YAWL.

First of all, we introduce functionality to spread *specifications* of one tenant over more than one engine. If *tenant 1* has *specification 1* and *specification 2*, we can then spread work as shown in Figure 14: one specification allocated to each of the engines.

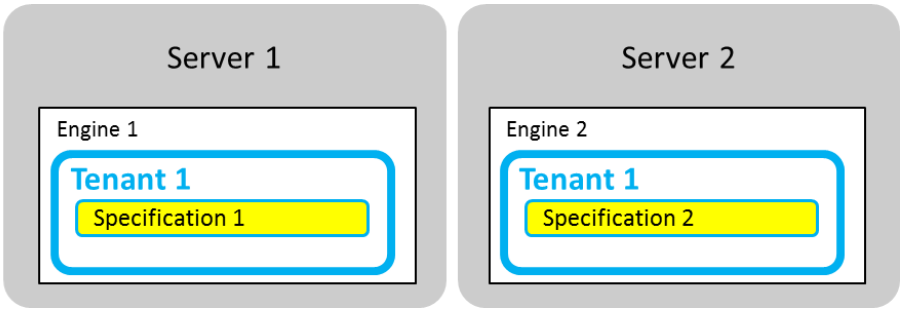


Figure 14: New type of spreading work over engines: specifications

Second, we introduce functionality to be able to spread *cases* of the same specification over multiple engines, each taking a portion of the cases running for that specification. If *tenant 1* has cases *c1, c2, c3* for *specification 1*, Figure 15 shows how these cases can then be spread over engines.

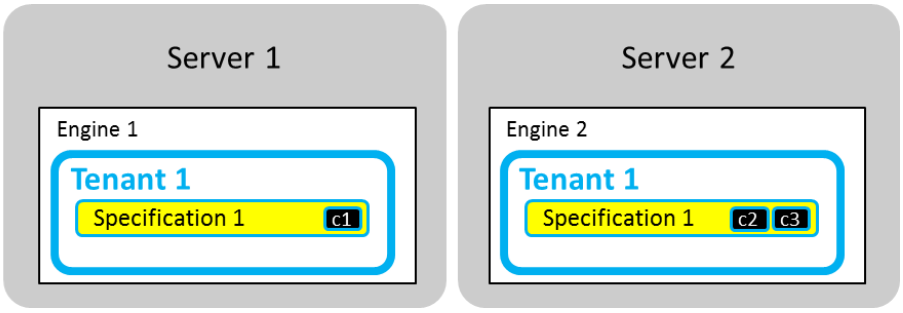


Figure 15: New type of spreading work over engines: cases

Combining the new forms of spreading work we introduced, all kinds of mixtures of engines, tenants, specifications and cases can be made. Figure 16 shows an example such a mixture, including all of the new forms. It shows multiple tenants running in *engine 1,* spreading specifications of *tenant 2* over *engine 1* and *engine* 3, and spreading cases of *specification 1* of *tenant 1* over *engine 1* and *engine 2.*

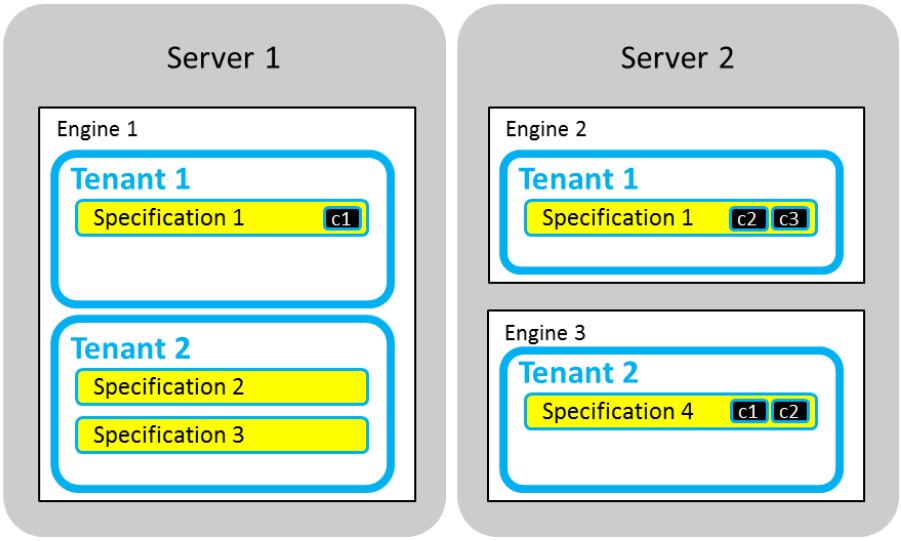


Figure 16: Example of spreading work, including two desired new ways to spread specifications and cases over engines

4 Architecture

The architecture of YAWL in the Cloud is described in a top-down manner in the following Sections. We start with a high level architecture, identifying which components shall be used or developed. The Sections that follow elaborate on these components, describing them in detail.

4.1 High Level (System) Architecture

The high level architecture of YAWL in the cloud is derived step by step, from the fact that the YAWL cloud will combine multiple classic YAWL engines into a larger system. The individual YAWL engines are used as-is, i.e. no changes will be made to them other than changing their configuration options.

We use the simplified graphical representation of the YAWL engine as introduced in Figure 3. Recall that all interfaces (A, B, E, X) accept inbound requests. Only interfaces B and E also send outbound requests themselves, mostly for announcements to the outside world. Blue arrows are used for inbound requests, whereas red arrows are used for outbound requests. The direction of the arrow is the direction of the request.

When using a single engine, the outside world can easily contact that engine at a single address. Every request is sent to that same engine. Usage of the YAWL cloud may not differ significantly from using a classic YAWL installation. So, when using two engines as depicted in Figure 17, it is unacceptable to have two separate points for incoming requests, because the outside world would then have to be adapted, to be able to determine to which engine a request should be sent.

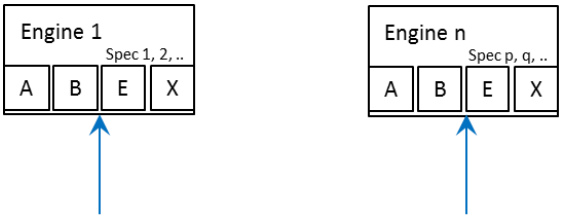


Figure 17: Two engines, two request destinations

To have a single point of entrance for incoming requests, a component must be placed in front of the engines. This component receives all requests, and routes them to the corresponding engine(s). Figure 18 shows this component, which we call *Router In*. To determine which engines to address, a mapping must be kept that links work items, cases, and specifications to engines. The mapping is stored in a database.

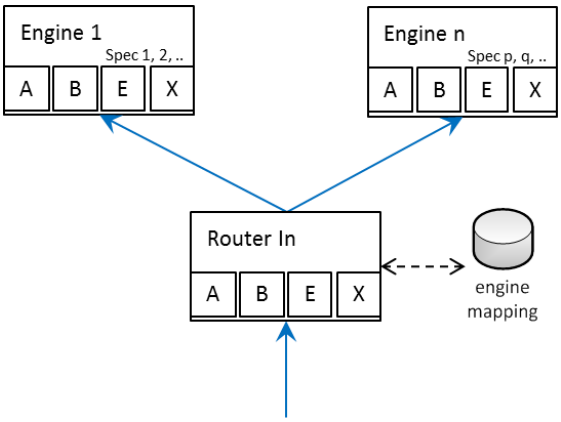


Figure 18: A router determines which engine serves a request

Since now all incoming requests of all engines go through Router In, it can easily become a bottleneck or single point of failure. This can be overcome by using multiple routers, as depicted in Figure 19. Any router should be able to route any request. The individual routers should therefore not keep individual state, but instead keep a global state in the database. In order to, again, accomplish a single point of entry for all incoming requests; we now use a load balancer to spread traffic over the routers. Preferably, this is a cloud based load balancer. In contrary to e.g. hardware load balancers, most cloud based load balancers are distributed systems themselves, which scale automatically. Scaling of the database can be simplified by using managed database hosting in the cloud, which often provides controls for scaling database performance and capacity up and down.

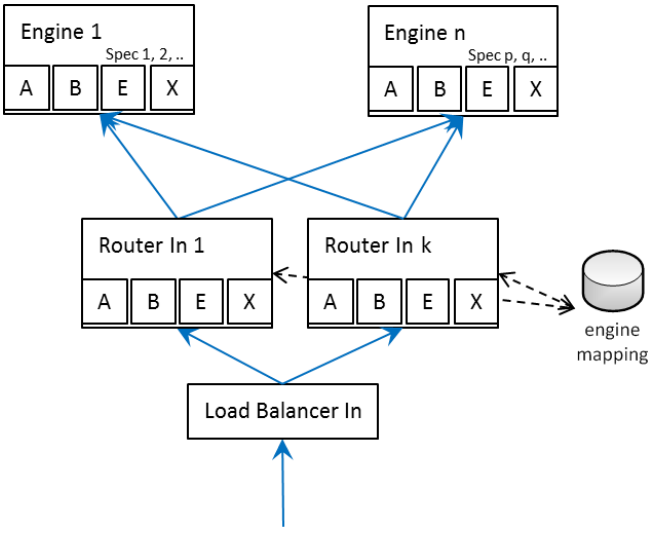


Figure 19: Load is spread over multiple routers by a load balancer

Up to now, we only considered incoming requests. YAWL also sends outgoing requests, mainly for making announcements to the outside world. The same strategy can be applied to these outgoing requests. This leads to the architecture depicted in Figure 20. For outgoing requests we again have routers, this time called *Router Out*, and again we have a load balancer on the incoming side of the routers. The fact that we have one outgoing arrow per Router Out is not a problem, since the HTTP requests are sent to the destination looked up in the engine mapping database. The source address of these requests is not of any importance.

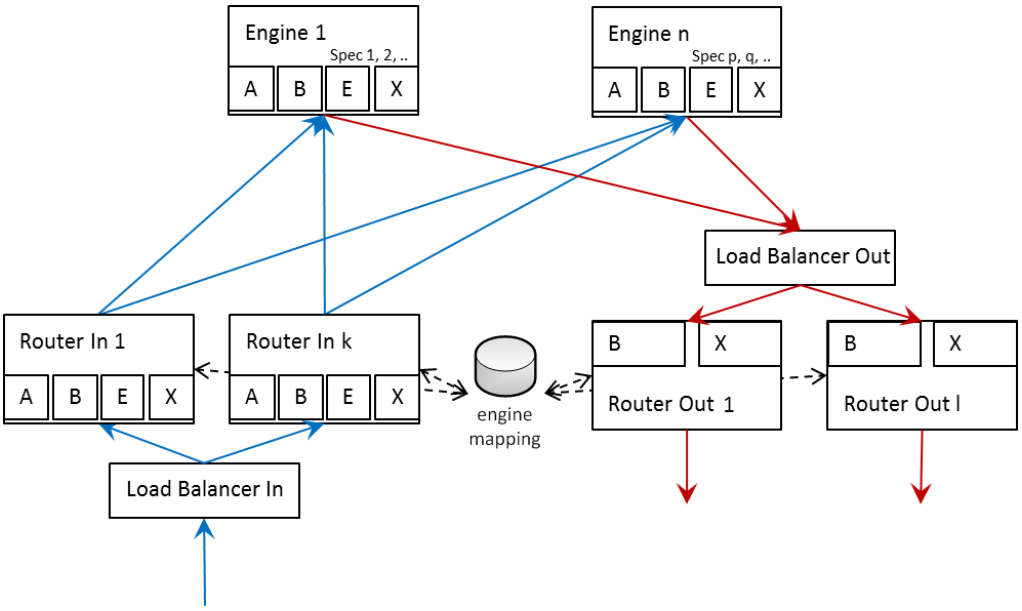


Figure 20: Architecture including both incoming and outgoing requests

When working with YAWL in the cloud through the provided interfaces, the user would have no insight at all in what happens in the internals of the cloud. Such insight is important, for example for monitoring and to be able to demonstrate the workings of this proof of concept. Therefore there is a dashboard component, which is described in Section 4.4. This is the same component that deals with re-allocation, which is described in Section 4.3.2. This component interacts with both the database and the individual engines. Figure 21 shows the architecture including this component.

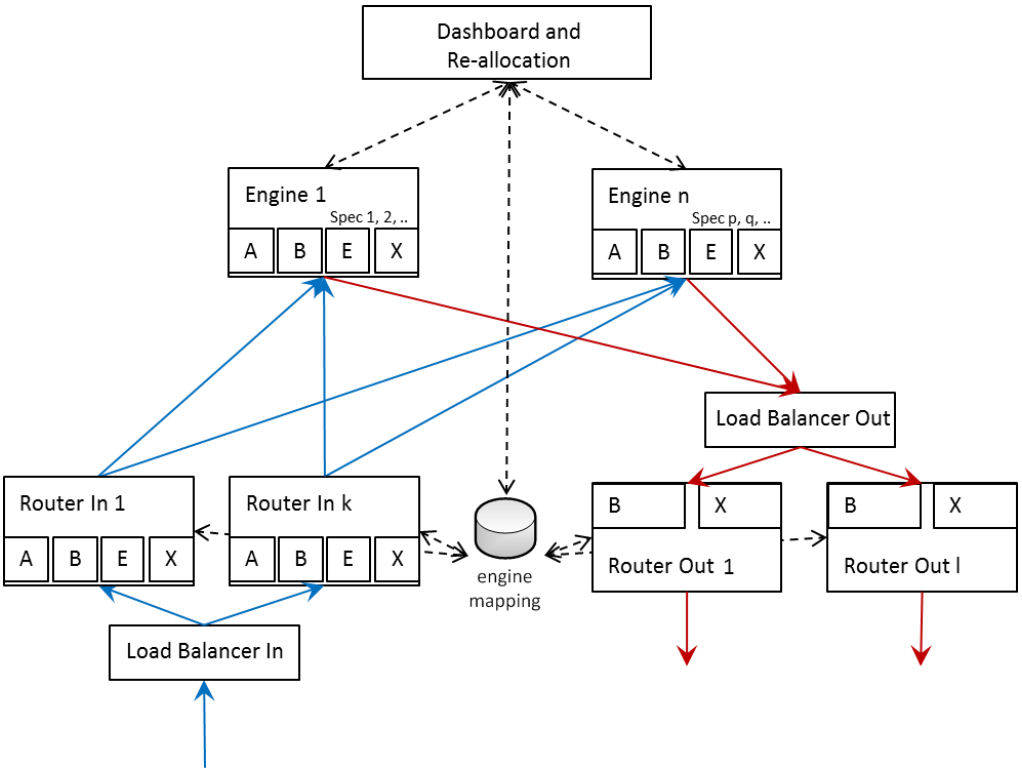


Figure 21: Architecture including dashboard

4.1.1 Maintaining the Engine Mapping

To route incoming requests, their mapping to the corresponding engines must be known. The mapping can be looked up based on the content of the request. More specific, this mapping can be looked up based on *specification id*, *case id*, or *work item id*. The mapping is stored in the database according to the database diagram depicted in Figure 22.

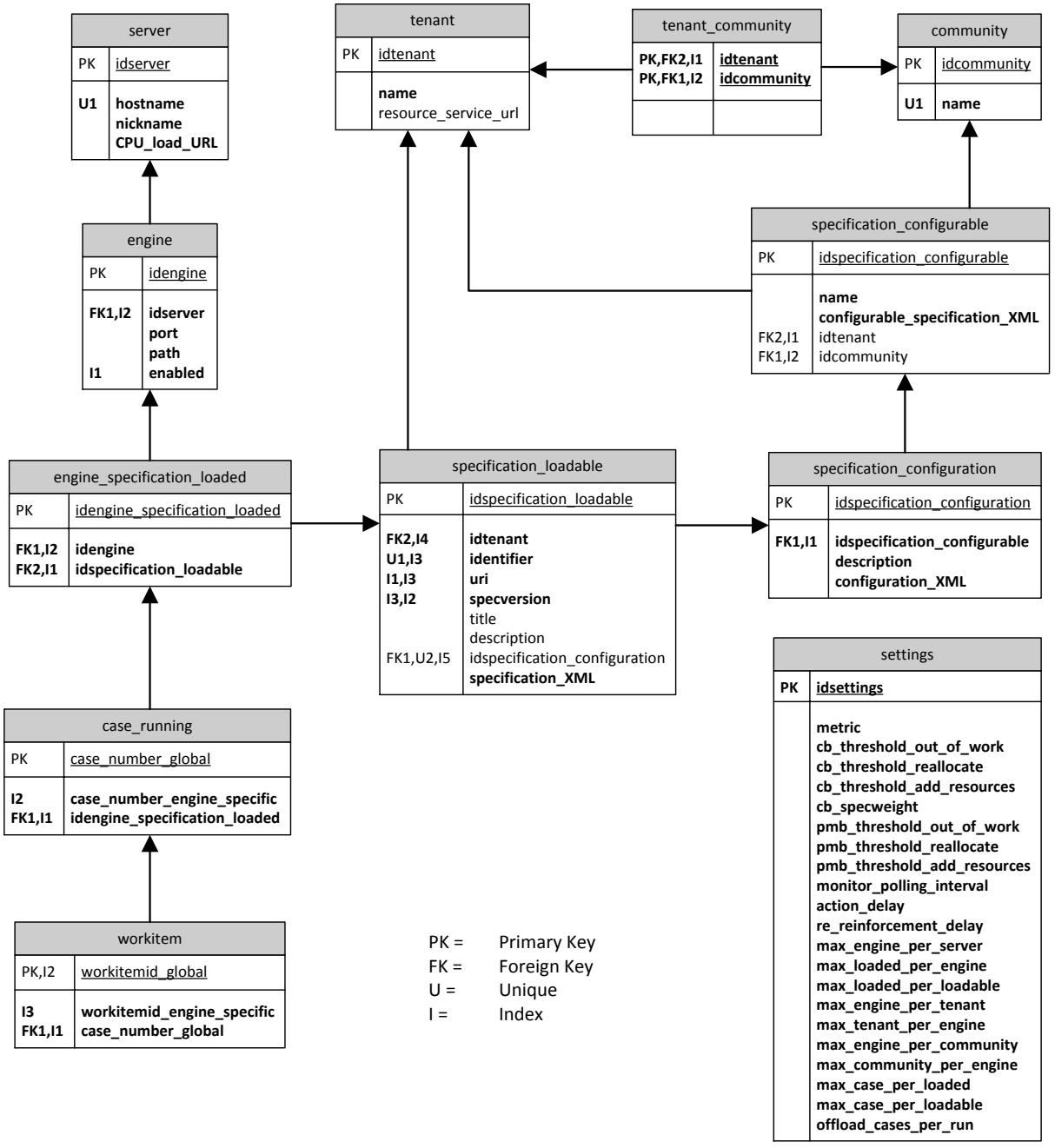


Figure 22: Database diagram of YAWL in the cloud

The left vertical chain in the model shows the administration that is a direct consequence of how YAWL works: server, engine, specifications loaded onto an engine, cases, and work items. There are four tables that contain information related to specifications:

- *specification_configurable* stores specifications that are not yet configured.
- *specification_configuration* stores configurations for stored configurable specifications.
- *specification_loadable* stores specifications that are directly loadable onto an engine, thus not containing any configurability anymore.
- *engine_specification_loaded* administrates which specifications are loaded onto which engines.

Whenever a new id is observed, it is added to the database by the router that observed it. New specification ids are discovered on interface A inbound. New case IDs are discovered in the response of an engine to the launch case action on interface B inbound. New work item ids are discovered when they are announced by the engines on interface B outbound.

Besides the engine mapping, the database is also used to store configuration settings that are applicable to all routers. This way they only have to be configured once. All these settings are related to the allocation strategies that are described in Section 4.3.

4.2 Router In and Router Out

Essential parts in the architecture of YAWL in the cloud are the routers. They are the crucial layer that makes the multiple as-is engines into a coherent whole. We identified the following minimal functionality these routers should support, starting with the inbound router:

Step 1: Receive Incoming Request

The inbound routers must provide all four inbound interfaces (A, B, E, X) to the outside world. They must listen to any incoming request arriving at any of these interfaces.

Step 2: Determine Target Engine(s)

After receiving the incoming request, the actual routing takes place. Using the request parameters, among which is the requested interface *action*, the router determines to which single engine, or group of engines, the request should be routed. For each of the target engines, the following steps must be performed:

1. The incoming *external request* must the translated into an internal *engine-specific request*. This step is required, because engines hand out identifiers. Such an identifier is unique for that single engine, but not necessarily unique over all engines. For all external identifiers that need translation, their engine-specific counterpart is looked up in the database.
2. The *engine-specific request* generated in step 1 is sent to the engine.
3. The *engine-specific response* from the engine is received.
4. The *engine-specific response* must be translated into an *external response*. Engine-specific identifiers are translated into external identifiers, by lookups in the database.

Step 3: Merge Responses

The outside world expects to receive one response for one request. All requests that had to be handled by multiple engines result in one response per engine, and thus have to be merged before they can be sent out. For requests that have been sent to a single engine, this step can be skipped. Depending on the interface action, different merging takes place. Appendix B shows, for each of the implemented interfaces, on which elements of the XML responses merging happens, and which merge step is performed. Some responses just need to be appended to each other, others have to be complemented.

Step 4: Filter Response

An engine is potentially doing work for multiple tenants. The engines are used as-is, and thus have no knowledge of tenants. As a result, responses of an engine potentially contain data belonging to another tenant than the tenant sending the request. For example, when requesting a list of all running cases, the routing in step 2 will forward the request to all engines that do work for the requesting tenant. The engines potentially run cases for other tenants too; these cases are filtered out in this step.

Note that, instead of filtering the merged response right after step 3, it is as well possible to filter the individual responses just before step 3. Depending on the performance and cost of the two steps, which also depends on the relative frequency of the various interface actions, either the one or the other way around can be resulting in better performance. For this proof of concept implementation we choose to first merge and then filter.

Step 5: Send Response

The response is now fully merged and filtered, resulting in a single response with data about only the requesting tenant. As a final step, this response to the original incoming request is sent out.

The *Router Out* is somewhat less complex. All outbound requests target only one external URL, so there is no need to merge responses. Furthermore, outbound requests contain data for one tenant only, so the filtering step can be skipped as well. Request and response translation still has to be performed; directions of translation are reversed compared to the direction of translation for *Router In*.

4.3 Allocation Strategies

We already introduced spreading work over engines, but we did not yet specify how this work should be distributed among these engines. For this we introduce *allocation strategies*, which consist of three parts:

$$\text{Allocation strategy} = (\ \text{occupancy rate},\ \ \text{thresholds},\ \ \text{restrictions}\ )$$

The following three Sections describe these three parts in detail. Section 4.3.4 gives some concrete examples of allocation strategies, and their configuration settings.

4.3.1 Occupancy Rate

We need a measure that describes the load of an engine, in order to be able to direct load to the least busy engine. We call this measure the *occupancy rate*. We introduce two basic options for this: a *count based metric*, and a *measurement based metric*. For the proof of concept implementation, we stick with these two basic metrics. However, if needed, the proof of concept implementation can be used to test more advanced and complex metrics afterwards.

Count Based Metric

The first metric we introduce is based on the assumption that the number of specifications and cases that is running on an engine are an indication of how busy that engine is. These numbers are combined into a formula. For this occupancy rate of a single engine, we use the following formula:

$$\text{Occupancy rate}\ (\ \text{engine}\ e\ ) = \sum_{\text{all specifications}\ i\ \text{running on}\ e} (\ sw(i)\ +\ cw(i) * cn\ (i, e)\ )$$

$$sw(i) = \text{specification weight of specification } i$$
$$cw(i) = \text{case weight of cases of specification } i$$
$$cn(i, e) = \text{number of cases of specification } i \text{ running on } e$$

The weight that the number of specifications puts on this metric, set by $sw(i)$, should be higher than that of the number of cases, set by $cw(i)$, because of the overhead it costs to load specifications to an engine, and because of the overhead it costs to consult unnecessarily many engines for interface requests regarding a single specification that have to be sent to all engines running that specification.

We did not develop a way to accurately determine the weight of individual specifications and cases, and we do not implement this for the proof of concept. Therefore we use a fixed value for $sw(i)$ and $cw(i)$. The implementation uses 1 for $cw(i)$ and a set fixed value for $sw(i)$, predefined and stored in the settings table. The lower bound of this metric is 0. It has no upper bound.

The actual number of specifications and cases that is running on an engine is known in the database. Thus, the individual engines do not have to be contacted to determine the value of this metric. It can be answered by a database query instead, which is an advantage.

A weakness of this metric is that the number of specifications and cases that is running on an engine is not necessarily a good measure for its load, e.g. when no actions are performed on these cases for a long time.

Measurement Based Metric

The second metric we introduce is based on the assumption that the CPU load of the server an engine is running on is an indication of how busy the engine is. The CPU load is a percentage, measured by the operating system of the server. Being a percentage, its lower bound is 0, and its upper bound is 100. The CPU load is calculated as a moving average over the last 20 observations: 10 seconds of two measurements per second.

A property of this metric is that it is a server metric, not a single engine metric. Its strength is that an engine, however low its load is, should not be loaded more heavy, if the server on which it runs is already heavy loaded with another engine. However, it gives no insight in which of the engines is causing the high CPU load. Furthermore, neither the current CPU load, nor its average over a timespan, give any assurance over CPU load that is required a minute later. However, this is similar to the count based metric giving no assurance over CPU load at any point in time.

There is room to improve the metric, by e.g. including measurements for memory usage, disk usage, or network interface metrics. We will not do so for the proof of concept implementation.

Both basic metrics have their strengths and weaknesses. For the proof of concept implementation we implement and experiment with both of them.

4.3.2 Thresholds

The defined occupancy rate metric can be used to make judgements based on the current state of YAWL in the cloud. When the value of the metric is above or below a certain threshold, we can draw conclusions from that. For this proof of concept, we identified three basic thresholds: one for scaling up, one for scaling down, and one for optimizing within the current scale.

All thresholds are configurable and are stored in the settings table of the database depicted in Figure 22. The configured values of the thresholds must adhere to the ordering depicted in Figure 23; the value of a threshold must be higher than the threshold depicted below it.

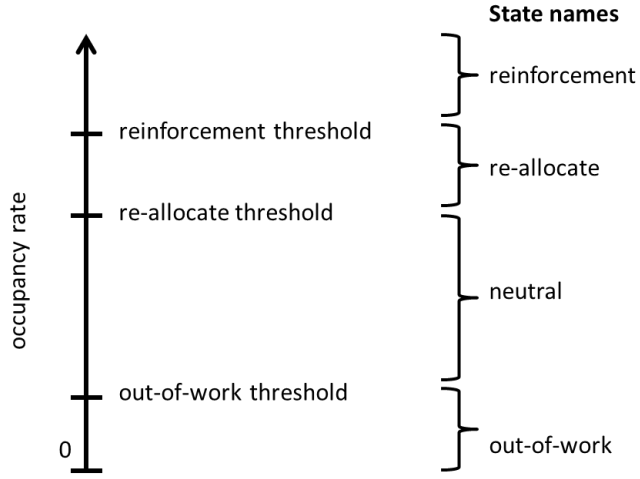


Figure 23: Occupancy rate thresholds and corresponding states

Out-of-work Threshold

When the occupancy rate of an engine falls below the out-of-work threshold, it will be gradually put out of work. The work of the engine will be moved to other engines, one case at a time. One must be aware that setting a high out-of-work threshold can cause yo-yoing of the number of engines, since the work that will be moved off the engine can cause other engines' occupancy rate to increase beyond the reinforcement threshold.

Re-allocate Threshold

When the occupancy rate of an engine grows above the re-allocate threshold, efforts will be made to try to relieve the engine of some work. This is done by moving work off the engine, by one or more cases at a time. Cases will only be moved to engines that have a lower occupancy rate. Moving of cases continues until the occupancy rate falls below the re-allocate threshold. The re-allocate threshold should not be too high, especially when using a measurement based metric, since there has to be some processing power left to move cases off the engine.

Reinforcement Threshold

Once the occupancy rate of an engine rises beyond the reinforcement threshold, the engine is considered to run at almost its full load. Reinforcement is needed, in the form of bringing an additional engine online. Since the reinforcement threshold lies above the re-allocate threshold, load will automatically be shifted off the engine once the extra engine comes online. If, at the next inspection of the occupancy rate, the occupancy rate is still higher than the reinforcement threshold, this must not automatically lead to adding another extra engine. Since starting the extra engine and shifting work of the original engine takes some time, this state must be ignored for a set time span. This time span is configurable and is called *re-reinforcement delay*.

For easy reference, we give state names to the various occupancy rate values. If an engine's occupancy rate is above the reinforcement threshold, we say that engine is in the reinforcement state. Similarly, if the value is above the re-allocate threshold, but below the reinforcement threshold, we say the engine is in re-allocate state. Engines that have occupancy rate values below the out-of-work threshold are in the out-of-work state. We say an engine is in a neutral state if its occupancy rate is higher than the out-of-work threshold, and below the re-allocate threshold.

A state will only be acted upon if an engine is in that state for long enough, or to be more precise, for longer than a predefined *action delay*.

4.3.3 Restrictions

When purely following the occupancy rate metric and its thresholds, any case can be allocated to any engine. Or, phrased differently, an engine can contain a mixture of cases of various specifications from various tenants. In some cases, grounds exist to put restrictions on the mixture. Restrictions can e.g. be set by governmental rules and legislation on data protection and segregation, or by the (un)willingness to share resources with another tenant. Adding restrictions also enables experimenting with different setups, such as one engine per specification. By default there are no restrictions.

Restrictions limit the cardinalities of the database. The restrictions have to be checked before every change of the current allocation. The maximum cardinalities that can be lowered are (the numbers correspond to Figure 24):

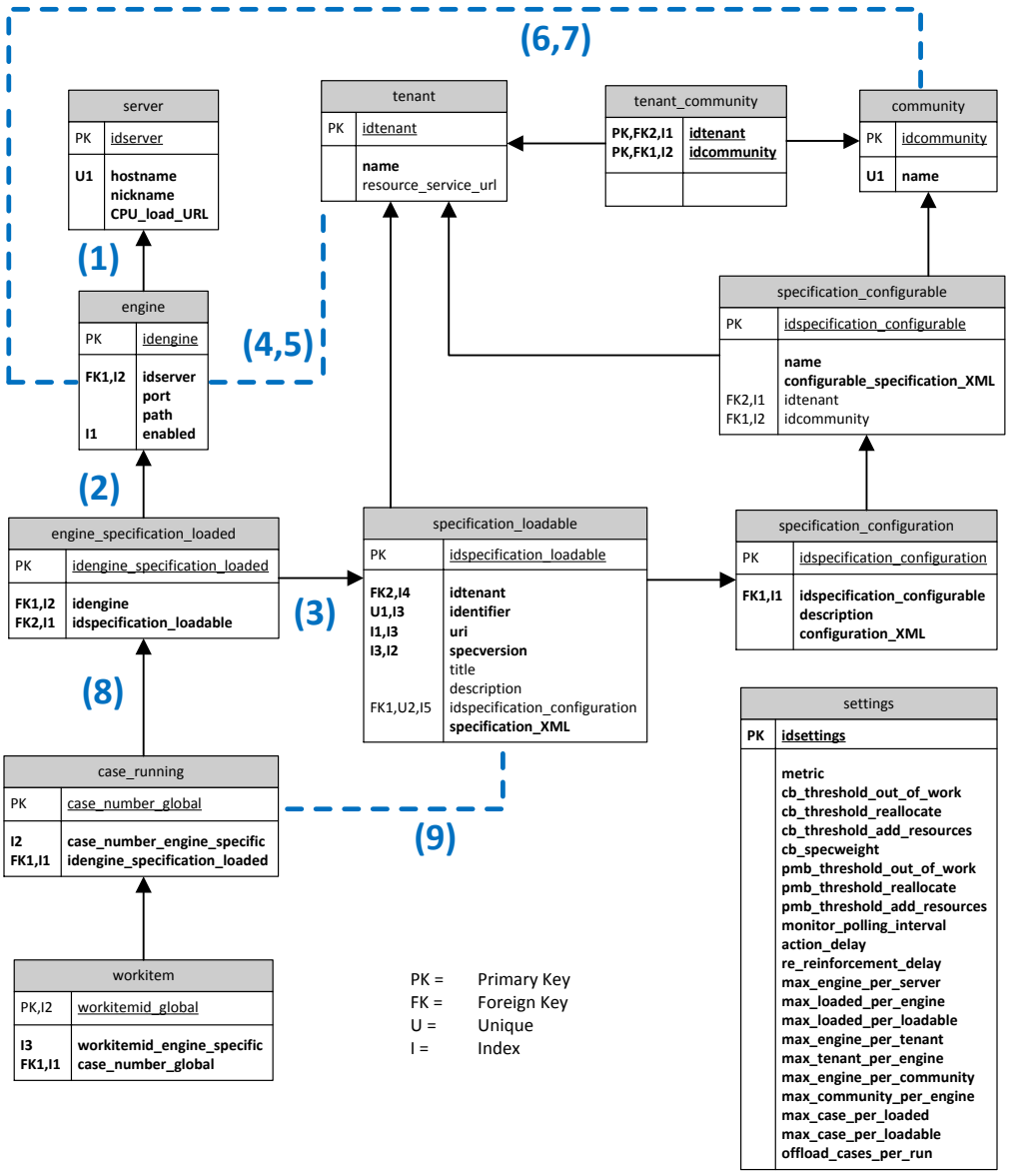


Figure 24: Cardinality restrictions visualised in database diagram

1. Maximum number of engines per server (*engine* per *server*).
2. Maximum number of specifications, running on one engine (*engine_specification_loaded* per *engine*).
3. Maximum number of engines running the same specification of one tenant (*engine_specification_loaded* per *specification_loadable*).
4. Maximum number of engines working for one tenant (*engine* per *tenant*, indirect via *engine_specification_loaded*, *specification_loadable*).

5. Maximum number of tenants sharing one engine (*tenant* per *engine*, indirect via *engine_specification_loaded*, *engine_specification_loadable*).
6. Maximum number of engines working for one community (*engine* per *community*, indirect via *engine_specification_loaded*, *engine_specification_loadable*, *tenant*, *tenant_community*).
7. Maximum number of communities sharing one engine (*community* per *engine*, indirect via *engine_specification_loaded*, *engine_specification_loadable*, *tenant*, *tenant_community*).
8. Maximum number of cases of the same specification, running on one engine (*case_running* per *engine_specification_loaded*).
9. Maximum number of cases for the same specification, as a total over all engines running that specification (*case_running* per *specification_loadable*)

We do not support lowering other cardinalities.

Cardinality restrictions are checked at any allocation, or change of allocation. Restrictions 1, 2, 3, 4, 5, 6, and 7 are all checked at the allocation of specifications to engines. Restrictions 8 and 9 are checked on case allocations.

Restrictions are enforced strongly: ultimately, given a finite set of engines, there are no valid allocations possible anymore. Both case and specification allocations are rejected if no valid allocations are available.

4.3.4 Examples

To get acquainted with allocation strategies, we provide two examples. The examples are arbitrarily chosen from the many combinations of parameters values that are possible.

Measurement Based: fast growing, slow shrinking

For this example we use the CPU measurement based metric. We define the thresholds on this metric such that increased load will easily lead to increasing the number of engines, long before engines get overloaded, thus the cloud is fast growing. To accomplish this, we need to choose a low reinforcement threshold. Furthermore we choose the thresholds such that engines that have been put into service, will not easily be put down when load decreases slightly. To accomplish this, we choose a low out-of-work threshold. Therefore we choose:

- Out-of-work threshold: 2% CPU
- Re-allocate threshold: 50% CPU
- Reinforcement threshold: 60% CPU

We choose to put one restriction on which engines can be used: we want every engine to run at most one specification at a time. Therefore we set the maximum number of *engine_specification_loaded* per *engine* to 1.

Count Based: sharing community

For this example we use the count based metric. We choose that the weight of deploying a specification to an engine, is 100 times the weight of a single case (i.e. $sw(i) = 100$, when $cw(i) = 1$). We define the following thresholds:

- Out-of-work threshold: 200
- Re-allocate threshold: 800
- Reinforcement threshold: 1200

Note that, since there is no known upper bound for the count based metric, it is hard to tell if the threshold value is low or high. It depends on the actual (virtual) hardware capabilities, what the upper bound of this metric on a specific system will be.

We restrict engines to only run work for one community of tenants at a time. Therefore we set the maximum number of *community* per *engine* to 1.

4.4 Dashboard

Via normal usage of YAWL in the cloud, it is practically invisible what goes on behind the scenes. To provide useful insight into the current status and workings of YAWL in the cloud, we introduce a dashboard. The current status can be observed from different perspectives. We choose the *what* and *how* perspectives. The functional view is the *what* view, which shows what the system is doing. The software view is the *how* view, which shows how the system is doing that. Both these views are available on the dashboard. To keep the dashboard data up to date, the dashboard must refresh itself at a set frequency.

Software View

The software view shows the current status of YAWL in the cloud from a technical perspective. As its primary dimension, it shows the servers that are in operation. For each of the servers it shows a tree-view of what is running on it: engines, tenants, specifications and cases. A schematic representation is depicted in Figure 25.

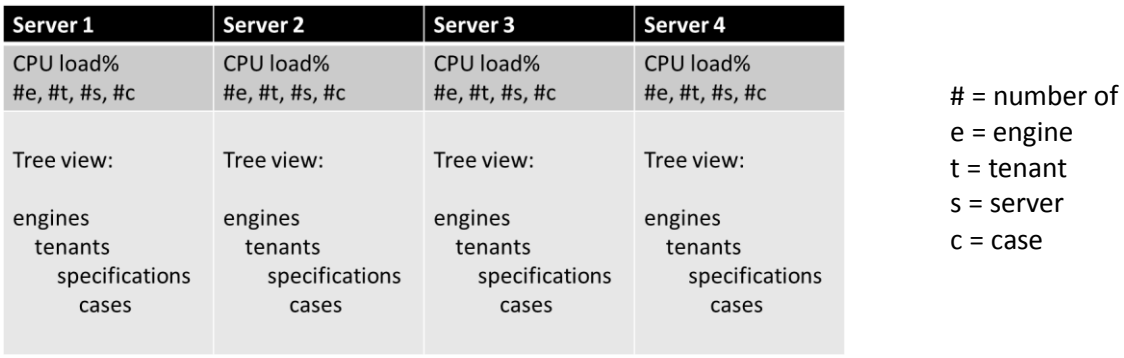


Figure 25: Schematic view of management component - software view

Functional View

The functional view shows the current status of YAWL in the cloud from a functional perspective. As its primary dimension, it shows the tenants. For each of the tenants it shows a tree-view of the specifications and cases running for that tenant. Servers and engines are only shown as informative side-note to the cases. A schematic representation is depicted in Figure 26.

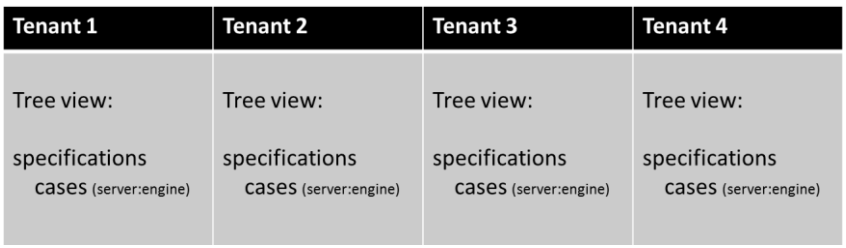


Figure 26: Schematic view of management component - functional view

Manual Actions

YAWL in the cloud automatically handles, among others, allocations, re-allocations and scaling of the system. Only if the system supports manual actions, we can influence the system state at will. Such manual interference can be useful, for example for maintenance, testing and analysis.

To be able to manually influence the system state, the dashboard must provide basic means to influence the current status of YAWL in the cloud. By hand, you can force some state changes. For this proof of concept the following manual actions are supported:

- Enabling an engine.
- Disabling an engine. Disabling is only allowed when the engine is not running any cases and specifications anymore.

Many other useful actions can be thought of, such as (not included in the proof of concept implementation):

- Adding a server to the YAWL cloud.
- Removing a server from the YAWL cloud. A server can only be removed when it is not running any engine.
- Adding an engine to a server.
- Removing an engine from a server. An engine can only be removed when it is not running any cases and specifications anymore.
- Manually moving a case to another engine. In case that engine is not running the corresponding specification yet, it must be ingested to that engine automatically.
- Manually moving a loaded specification to another engine. All cases running for that specification must be moved too. In case the target engine is already running the specification, cases will be moved there.
- Merging an engine into another engine. All specifications and their cases will be moved to the other engine. The source engine is empty afterwards. Merging into an empty engine on another server is interpretable as moving the engine into that server.

4.5 Applicability to Other BPMS

Although this architecture was devised with YAWL in mind, it is applicable to other BPMS too, if those BPMS match the following criteria:

- The BPMS has an engine that is accessible via an API, such as a REST or SOAP API.
- The API must allow for a man-in-the-middle approach.
- All actions performed on the API either in itself contain enough information to route and handle the request, or contain enough information to look up all information needed to route and handle the request.
- The results provided by the API are in a format that can be merged by some algorithm.
- The results provided by the API are in a format that can be filtered by some algorithm.

Figure 27 shows the architecture of YAWL in the cloud. Given by region, the following list states what must be changed to support another BPMS that meets above requirements:

1. The engine of the BPMS must be replaced, by definition.
2. The routers must be altered to mimic the interface endpoints and actions of the BPMS. Internal structure of the routers stays the same.
3. The engine mapping database must be altered to support storage of the specification, case and work item identifiers used by the BPMS (e.g. natural numbers, hexadecimal, etc.)
4. The dashboard must be altered to reflect any mapping database changes, and BPMS specific names (e.g. rename of specification, to match the BPMS's standard wording).

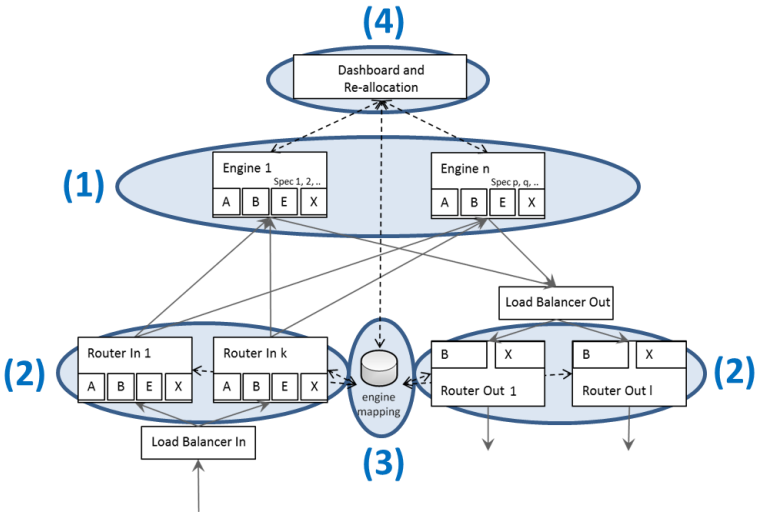


Figure 27: BPMS specific architecture regions

We conclude that the approach of this architecture is not YAWL specific, but the proof of concept implementation is, since the individual components contain YAWL specific implementation details. A proof of concept implementation for another BPMS that matches the criteria can be made in an analogue way to the implementation for YAWL as described in Chapter 5.

5 Proof of Concept Implementation

To prove that the proposed architecture devised in Chapter 4 actually works, we created a proof of concept implementation of this architecture. We start by describing the environment in which the proof of concept implementation was made in Section 5.1. Next, we describe every component of the implementation in Section 5.2, except for user facing dashboard component, which is described in Section 5.3.

5.1 Environment

Decisions about the implementation have been made in line with the original YAWL development. YAWL in the cloud is implemented in Java, using the integrated development environment of NetBeans IDE 7.1.1 [26]. Interaction with the database is handled by the Hibernate [27] Object/Relational Mapper framework for Java. Hibernate supports many database engines; we use MySQL [28] in our deployment. The deployment runs in Apache Tomcat [29].

Hibernate

Hibernate is an ORM (Object-Relational Mapping) framework library for Java. It can map an object oriented domain model onto a relational database and vice versa. Hibernate is database platform independent, and can be configured to connect to a wide range of different relational database servers. For this proof of concept we connect it to MySQL using the MySQL JDBC driver. The default built-in database connection pool of Hibernate cannot be used for production scenarios. Instead we configure to use the C3P0 connection pooling library [30], which can augment any standard JDBC driver. C3P0 has its own configuration settings, among which are specifications of the minimum and maximum number of simultaneous database connections in the pool, as well as the used connection timeout.

YAWL Library

Where advantageous, we make use of existing YAWL sources, as-is, by linking the library *yawl-lib-2.2.jar*. For example, the request forwarder inherits from the YAWL interface client class. Furthermore the YAWL utilities for JDOM and Java Servlets are used.

Namespaces

The namespace *com.yawlcloud* contains all implemented components. Its sub-namespace *com.yawlcloud.data* contains classes for all data types, corresponding to tables in the database diagram as depicted in Figure 22.

Interface Endpoints and URLs

As established in previous Chapters, YAWL in the cloud must present the same interfaces to the outside world as conventional YAWL does. This would result in an implementation of interfaces A, B, E, and X. However, given the limited time available for this project, it was impossible to fully implement all of these interfaces.

Interfaces A and B are the essential interfaces for a proper working of the YAWL engine (see Section 2.1.4). Interfaces E and X are not strictly necessary to interact with an engine. Therefore we did not focus on getting some implementation done for all interfaces, but rather focus on getting proper functioning interfaces A and B. Or, to be more precise, we implement interface A inbound, interface B inbound, and interface B outbound.

The base URL for YAWL in the cloud, which is a prefix for all the following URLs, is:

http://*router-hostname*:*router-port*/YAWLCloud

The YAWL resource service is not multi-tenant. Therefore, for every tenant, one resource service must be hosted. For filtering purposes, it is necessary to know which tenant is sending in a request to YAWL in the cloud, so all data belonging to other tenants can be filtered out. To identify this tenant, every tenant gets its own virtual endpoint of YAWL in the cloud interfaces. This results in the following URLs:

Interface A	/ia/*tenant-id*/ia/
e.g.	/ia/7/ia/
Interface B inbound	/ib/*tenant-id*/ib/
e.g.	/ib/7/ib/

The engines in the YAWL cloud send their outbound requests, mostly announcements, to the outbound interface of the cloud. There the engine-specific request must be translated to become an external request. The HTTP request does not contain enough information to identify which engine sent it, if multiple engines are running on one hostname. Therefore, each engine is configured to point its outbound requests to its own virtual endpoint of YAWL in the cloud. This results in the following URLs:

Interface B outbound	/resourceService/ib/*engine-hostname*/*engine-port*/
e.g.	/resourceService/ib/server1/8081/

Note that, although there is no resourceService running at this endpoint, but requests will be forwarded to the resourceService of the individual tenant the request belongs to. The URL of that tenant-specific resourceService is retrieved from the database.

YAWL in the cloud also listens to the following URLs, which are however not implemented in this proof of concept:

Interface E	/logGateway
Interface X inbound	/ix/
Interface X outbound	/workletService/ix/

5.2 Component Implementation

The implementation is split into several components. Some components, like the Data Access Layer, provide services to multiple other components. Other components, like the Request Router, are part of the flow a requests follows through the components. The following Sections describe the individual components in more detail; first the supporting components, then describe the request flow components.

Classes that merely represent entries in tables of the database are not described, as they are generated using Hibernate and directly relate to their database table structure.

5.2.1 Supporting Components

Some components are not in the normal request flow themselves, but are used by multiple components in that flow. These components are the Data Access Layer, and the Interface Cloud Based Server.

Data Access Layer

The Data Access Layer forms the link between the database and the implemented components. It adds a layer of abstraction around the database, such that Hibernate-specific code is only to be found in the Data Access Layer. The Data Access Layer contains functions for retrieving, updating and deleting data from the database. It works directly on, and returns objects from the *com.yawlcloud.data* namespace.

Interface Cloud Based Server

The Interface Cloud Based Server extends the *HttpServlet* from the standard Java servlet library. All the interfaces provided by YAWL in the cloud on their turn extend the Interface Cloud Based Server. It implements some basic functionality that is used by all of the interfaces, such as setting the correct character encoding and providing functions for logging and debugging.

HTTP responses allow for extra non-standard headers to be added to them, normally prefixed with "X-". We use such headers to provide information on how the request was handled. We always add information about the router that handled the request and a powered-by watermark. Depending on the interface (A, B, E, X), more headers specific to the working of that interface are can be added. Table 1 lists some example values for the headers we add.

Table 1: Additional HTTP response headers

Usage	Header	Example value
All interfaces	X-Powered-By	YAWL in the Cloud
	X-Served-By-Router	router1 (10.0.0.1:8080)
Interface specific (example: B inbound)	X-Processed-By-Engines	[server1:8080/engine1, server1:8080/engine2]
	X-Processed-Action	getLiveItems

5.2.2 Request Flow Components

The architecture of a router includes a flow through five steps, as described in Section 4.2, which all must be included in the proof of concept implementation. For every step or sub-step we introduce one component that implements it. The following listing shows all (sub-)steps, and the ordering to which they adhere:

1. Request router
2. Request translator
3. Request forwarder
4. Response translator
5. Result merger
6. Result filter

Figure 28 depicts the flow of a request through these components, using a three-server request example of interface B inbound. Furthermore, it shows the relationship between the individual components and the global architecture.

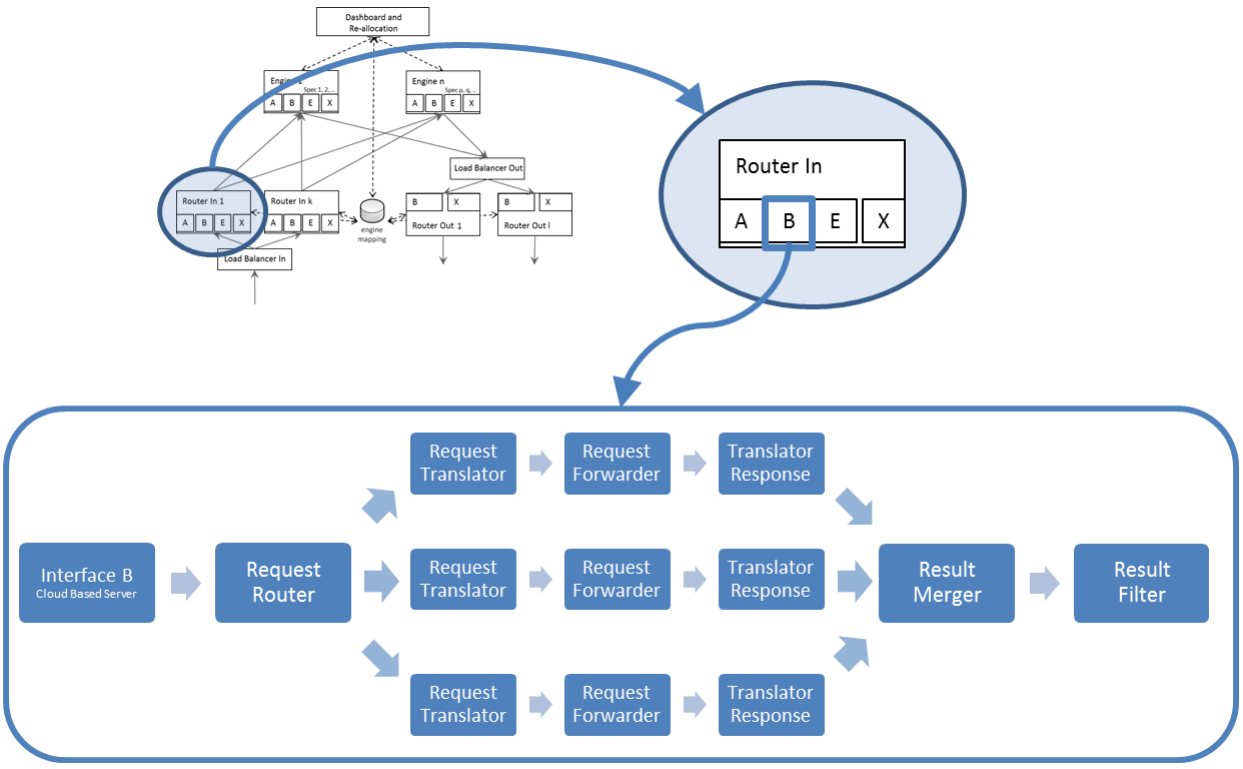


Figure 28: An example flow through the components, for a three-engine request to interface B

Running Example: the Flow of one Request

The role and function of each of the individual components is best understood when we follow the flow of a single request through all of the components. In this running example we will give input and output for each of the components. The example request we choose to follow is the action *getAllRunningCases* on interface B inbound. This request is sent by the *resourceService* of a tenant every time a user browses to the *Cases* page in the navigation bar. As the name of the action suggests, the response to this request must be a list of all the running cases for the tenant that is making the request. Table 2 shows the request parameters of this running example: action contains the action performed on the interface, and sessionHandle is used for session authentication.

Table 2: Running example input

Request parameter	Parameter value
action	getAllRunningCases
sessionHandle	7079ad1e-7477-4fac-ace6-26c195234253

Request Router

The role of the Request Router is to take a routing construct, a tenant, and a set of parameters, and return a list of all the engines that should be consulted to fulfil the request. The routing construct is determined by the interface implementation; the set of parameters is the set of request headers. Which of the parameters are used, if any, depends on the routing construct. Table 3 lists the implemented routing constructs and their semantics. The Request Router calls functions of the Data Access Layer to retrieve the (list of) engine(s).

Table 3: Routing constructs and their semantics

Routing construct	Semantics
AllDestinations	Returns all engines in operation.
AllDestinationsByTenant	Returns all engines in operation for this tenant.
AllDestinationsBySpecificationTenant	Returns all engines in operation for the specific specification of this tenant.
DestinationBySpecification	Returns all engines in operation for the specific specification.
DestinationByCase	Returns the engine running the specific case.
DestinationByWorkItem	Returns the engine running the specific work-item.

Request Router – Running Example

The Request Router looks up the action getAllRunningCases in its routing table. It finds that it must be routed using the AllDestinationsByTenant routing construct. A lookup in the database, through the Data Access Layer, learns that the current tenant has two engines that do work for this tenant: engine1 and engine2.

Request Translator

The purpose of the Request Translator is to translate the parameter map of a request. It takes a parameter map, and an engine, and returns the translated equivalent of the original parameter map. Depending on if the interface is inbound or outbound, the values in the parameter map are either translated from public to engine-specific or from engine-specific to public.

A parameter map is a map with string keys, and string values. The Request Translator walks the original map key by key, meanwhile filling a translated map with the translated values. The key determines if and how the value is translated. If the key equals *workItem*, the value will be treated differently: since the value of that parameter is an XML fragment, a Response Translator is created to recursively walk and translate the XML value before adding the result to the translated map.

Request Translator – Running Example

The Request Translator is invoked two times: one time for engine1, and one time for engine2. A lookup in the routing table learns that the parameter action does not need translation. By default, the sessionHandle is replaced by a sessionHandle for the specific engine. This leads to the following results:

Table 4: Translated request for engine 1

Request parameter	Parameter value
action	getAllRunningCases
sessionHandle	ef359e13-d234-4756-865e-49d59c3f36d2

Table 5: Translated request for engine 2

Request parameter	Parameter value
action	getAllRunningCases
sessionHandle	c61c773a-5a13-4bec-acc6-99656cdc8ffa

Request Forwarder

The Request Forwarder forwards requests that have been translated. To do so, it receives the destination URL for the request, and the request parameters that have to be included. As its result, it returns the response body it receives, as a string. Response headers are ignored, alike they are by the classic YAWL engines. The Request Forwarder extends the *Interface_Client* class from the YAWL library, which contains the code to make the actual HTTP requests.

Request Forwarder – Running Example

The translated requests that resulted from the Request Translator are sent to the engines. The engines respond with the following data:

Engine1:

```
<response>
 <AllRunningCases>
  <specificationID identifier="UID_aaaaaaaa-7fca-4ead-bb3d-485b721aee29" version="0.1" uri="CreditAppProcess">
   <caseID>82</caseID>
   <caseID>83</caseID>
   <caseID>87</caseID>
  </specificationID>
 </AllRunningCases>
</response>
```

Engine2:

```
<response>
 <AllRunningCases>
  <specificationID identifier="UID_bbbbbbbb-7fca-4ead-bb3d-485b721aee29" version="2.3" uri="DebitCheck">
   <caseID>4</caseID>
   <caseID>7</caseID>
  </specificationID>
  <specificationID identifier="UID_cccccccc-7fca-4ead-bb3d-485b721aee29" version="2.3" uri="DebitCheck">
   <caseID>82</caseID>
  </specificationID>
 </AllRunningCases>
</response>
```

Response Translator

The purpose of the Response Translator is to translate response XML documents. It takes the XML response as a string, and an engine, and returns the translated equivalent of the original XML document. Like with the Request Translator, the translation direction of the Response Translator depends on if the interface is inbound or outbound.

The XML document that is received as a string is first parsed into an XML Document object. Recursively all elements in the XML Document are walked and translated, as a hierarchy walk that starts with the root element. Upon walking an element, first its attribute values are translated. The attribute name determines if and how the value is translated. Next, if the element has no children, the value of the element is translated, using the name of the element as the key to determine if and how its value should be translated. If the element has children, all child elements of the element will be walked recursively.

Response Translator – Running Example

A lookup in the routing table for the action getAllRunningCases learns that the specificationID elements need translation. It specifically states that its identifier attribute should be translated as a specification GUID. Furthermore, it states that the caseID elements' data must be translated as a case number. Running the Response Translator gives the following output:

Engine1:

```
<response>
 <AllRunningCases>
  <specificationID identifier="UID_aaaaaaaa-7fca-4ead-bb3d-485b721aee29" version="0.1" uri="CreditAppProcess">
   <caseID>100</caseID>
   <caseID>102</caseID>
   <caseID>103</caseID>
  </specificationID>
 </AllRunningCases>
</response>
```

Engine2:

```
<response>
 <AllRunningCases>
  <specificationID identifier="UID_bbbbbbbb-7fca-4ead-bb3d-485b721aee29" version="2.3" uri="DebitCheck">
   <caseID>101</caseID>
   <caseID>108</caseID>
  </specificationID>
  <specificationID identifier="UID_cccccccc-7fca-4ead-bb3d-485b721aee29" version="2.3" uri="DeclineNotifications">
   <caseID>107</caseID>
  </specificationID>
 </AllRunningCases>
</response>
```

Result Merger

When a request has been forwarded to multiple engines, the results these engines sent back must be merged into a single result, before it can be sent out. Merging takes place based on a set of merge rules. The rules that should be applied are fed to the Result Merger by the individual interfaces' Cloud Based Server, based on the called interface action. The implementation of the Result Merger is specific for XML formatted results.

The set of merge rules is stored in a map. The keys of the map are the element names that must trigger the execution of a certain merge rule. The values of the map are the merge rules themselves. A single merge rule is a combination of a merge criteria, and a merge action. Table 6 lists the supported merge *criteria* and their semantics. Table 7 lists the supported merge *actions* and their semantics.

Table 6: Result Merger - Merge criteria and their semantics

Merge criteria	Semantics
Name	The merge action will be performed on any two elements at the same path, that have the same given name.
Name_and_Attributes	The merge action will be performed on any two elements at the same path, that have the same given name, that have equal attribute values.
Name_and_Attributes_and_Content	The merge action will be performed on any two elements at the same path, that have the same given name, that have equal attribute values, and have equal (child) content.

Table 7: Result Merger - Merge actions and their semantics

Merge action	Semantics, when merging matching elements ***element1*** from document ***A*** and ***element2*** from document ***B***
Combine elements	The resulting document contains both *element1* and *element2*. Used e.g. to join the list of running case numbers provided by the engines.
Complement children	The resulting document contains *element1*, whose child content is complemented with the child content of *element2* that is not present in *element1* yet.
Append children	The resulting document contains *element1*, whose child content is appended with all child content of *element2*.

Result Merger – Running Example

Looking up the merge rules for the action getAllRunningCases in the routing table learns that the response sections must be Combined on Name. The AllRunningCases element must also be combined with merge criteria Name. The elements specificationID must be Combined with criteria Name and Attributes. This leads to the following result:

```
<response>
 <AllRunningCases>
  <specificationID identifier="UID_aaaaaaaa-7fca-4ead-bb3d-485b721aee29" version="0.1" uri="CreditAppProcess">
   <caseID>100</caseID>
   <caseID>102</caseID>
   <caseID>103</caseID>
  </specificationID>
  <specificationID identifier="UID_bbbbbbbb-7fca-4ead-bb3d-485b721aee29" version="2.3" uri="DebitCheck">
   <caseID>101</caseID>
   <caseID>108</caseID>
  </specificationID>
  <specificationID identifier="UID_cccccccc-7fca-4ead-bb3d-485b721aee29" version="2.3" uri=" DeclineNotifications">
   <caseID>107</caseID>
  </specificationID>
 </AllRunningCases>
</response>
```

Result Filter

If work for a single tenant is executed by multiple engines, some actions require to forward the request to all of these engines. The classic YAWL engine itself has no notion of tenants, and may also return data belonging to other tenants. The Result Filter filters out all of the data belonging to another tenant. The implementation of the Result Filter is specific for XML formatted results.

Filtering takes place based on a set of filtering rules. The set of rules is stored in a map. The keys of the map are the names of the elements that should trigger execution of a filter rule. The values of the map are the actual filter rules themselves. One filter rule is the combination of a filter *value location*, and a filter *value type*. For two of the three filter value locations the value location needs a string name. The filter value location determines which part of the XML element should be inspected. The filter value type determines how the value should be interpreted, i.e. against which database table it must be tested. Table 8 lists the supported filter *value locations* and their semantics. Table 9 lists the supported filter *value types* and their semantics.

The individual interfaces' Cloud Based Server determine which set of filter rules is imposed on the provided result.

Table 8: Result Filter - Filter value locations and their semantics

Filter value location	Semantics
NodeText	A string containing the value of the element itself.
ChildNodeText (valueName)	A string containing the value of the first child-element of the element, that has the given name.
AttributeValue (valueName)	A string containing the value of the attribute of the element, that has the given name.

Table 9: Result Filter - Filter value types and their semantics

Filter value type	Semantics
CaseNumberGlobal	If the value found in the filter value location is not a global case number of the tenant making the request, the element is removed from the document.
SpecificationGUID	If the value found in the filter value location is not a specification GUID of the tenant making the request, the element is removed from the document.

Result Filter – Running Example

A lookup of the filter rules for the action getAllRunningCases learns that the attribute value specificationID must be filtered using the type SpecificationGUID. The caseID elements must be filtered based on its node text, using the value type CaseNumberGlobal. The specification DeclineNotifications belongs to another tenant. The ResultFilter filters it out completely, resulting in the following end result that is sent back as a response to the original request:

```
<response>
 <AllRunningCases>
  <specificationID identifier="UID_aaaaaaaa-7fca-4ead-bb3d-485b721aee29" version="0.1" uri="CreditAppProcess">
   <caseID>100</caseID>
   <caseID>102</caseID>
   <caseID>103</caseID>
  </specificationID>
  <specificationID identifier="UID_bbbbbbbb-7fca-4ead-bb3d-485b721aee29" version="2.3" uri="DebitCheck">
   <caseID>101</caseID>
   <caseID>108</caseID>
  </specificationID>
 </AllRunningCases>
</response>
```

5.3 Dashboard

The YAWL engines have to be installed on (virtual) servers by hand. By adding them to, or removing them from the YAWL in the cloud database, they can put into or out of service. In the future, these steps can be automated as well, to take full advantage of the scaling capabilities of cloud infrastructure. Almost all IaaS providers provide APIs to start and stop servers, which can be used to achieve this.

Routers also have to be installed on (virtual) servers by hand. To put a router into service, it has to be added to the configuration of the load balancer. Most cloud based load balancers have APIs that allow automating this step. To take a router out of service, it has to be removed from the load balancer configuration. Many cloud based load balancers are able to provide basic health checks on the servers it load-balances to, such as verifying an HTTP response code on a specified URL. Then, unhealthy servers can automatically be taken out of action.

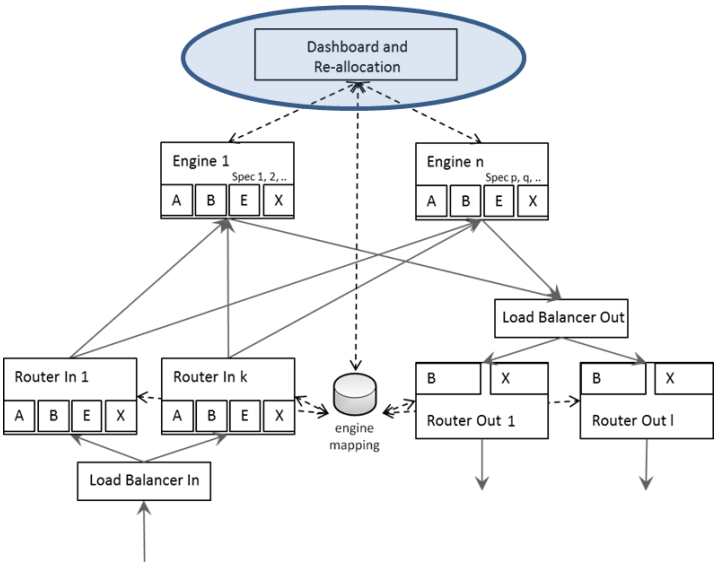


Figure 29: Place of dashboard component in architecture

5.3.1 Web User Interface

To administrate the YAWL in the cloud database, a web user interface was created using JSF (Java Server Faces). Figure 30 shows the available menu options from the interface:

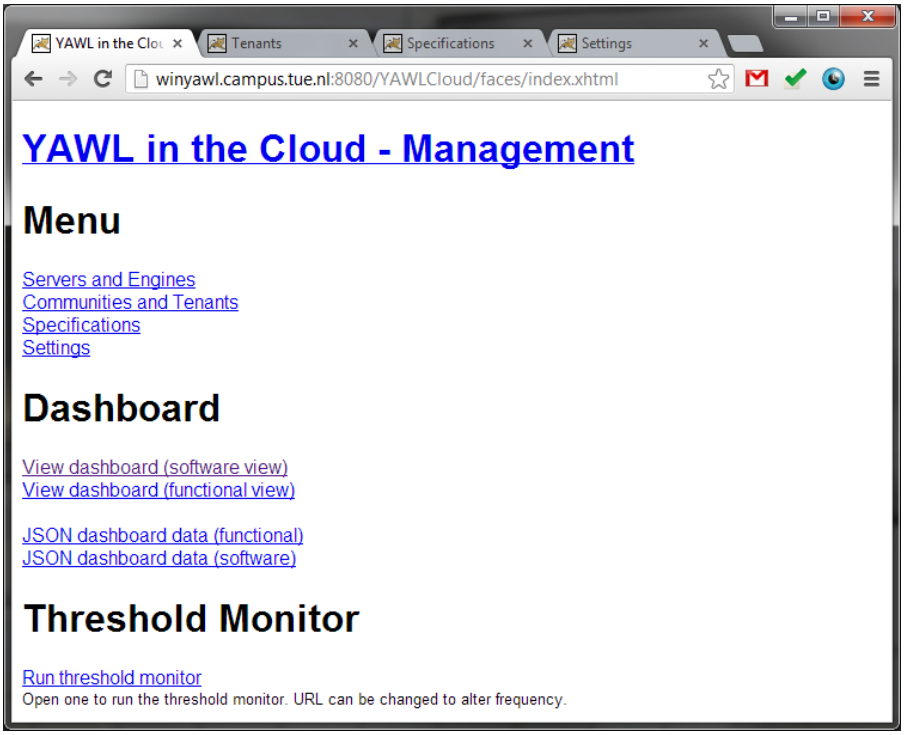


Figure 30: YAWL in the Cloud – Management user interface menu

Figure 31 shows the screen you get when clicking on 'Servers and Engines' from the menu. Similar screens exist for communities and tenants, specifications, and YAWL in the cloud settings.

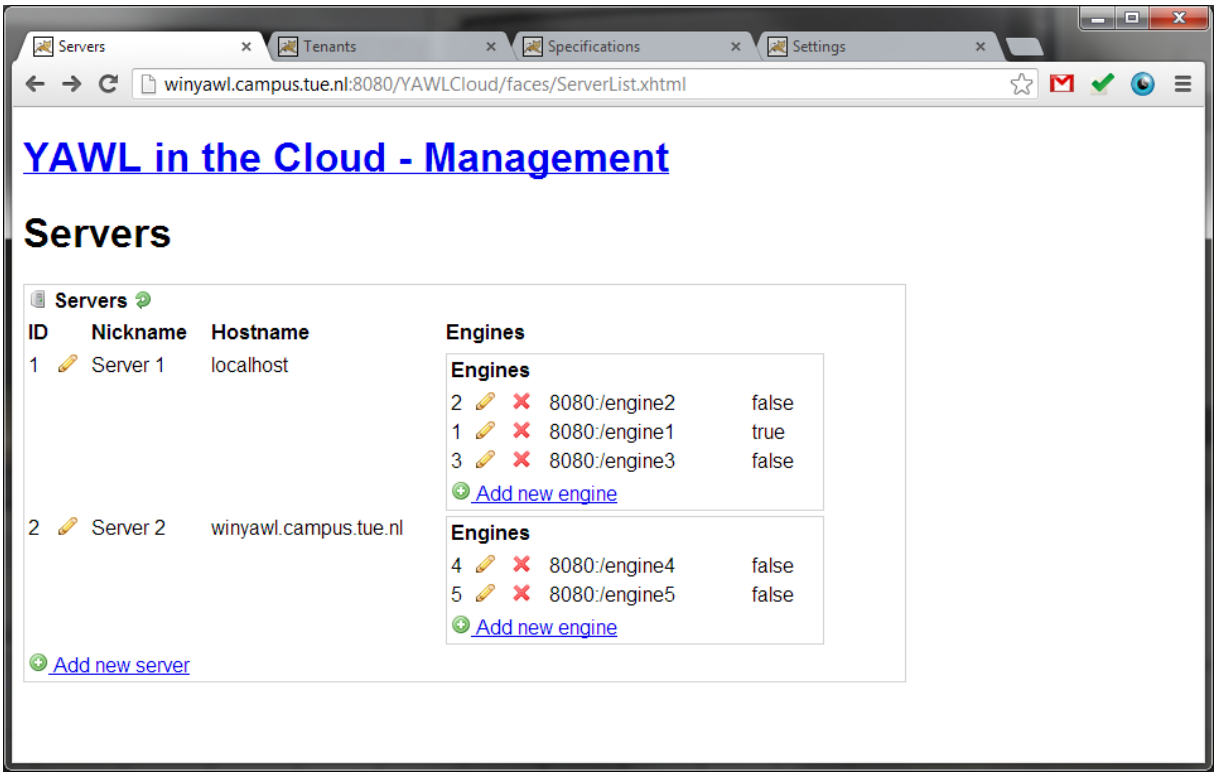


Figure 31: YAWL in the Cloud user interface - servers and engines

Another part of the web user interface is the dashboard, which we describe in the following Section.

5.3.2 Software View & Functional View

The dashboard shows the current status of the system. It is an HTML page containing a tree-views. The format of the tree-views depends on the chosen view. The tree-views are made using jsTree for jQuery [31]. The tree is populated from JSON data that is produced by the Interface Dashboard servlet. By clicking on the nodes in the tree, they can be collapsed and expanded. The dashboard refreshes itself every 5 seconds.

The dashboard is generated using JSP (Java Server Pages).

Figure 32 shows an example state of the system on the dashboard in functional view, Figure 33 shows the same state in software view.

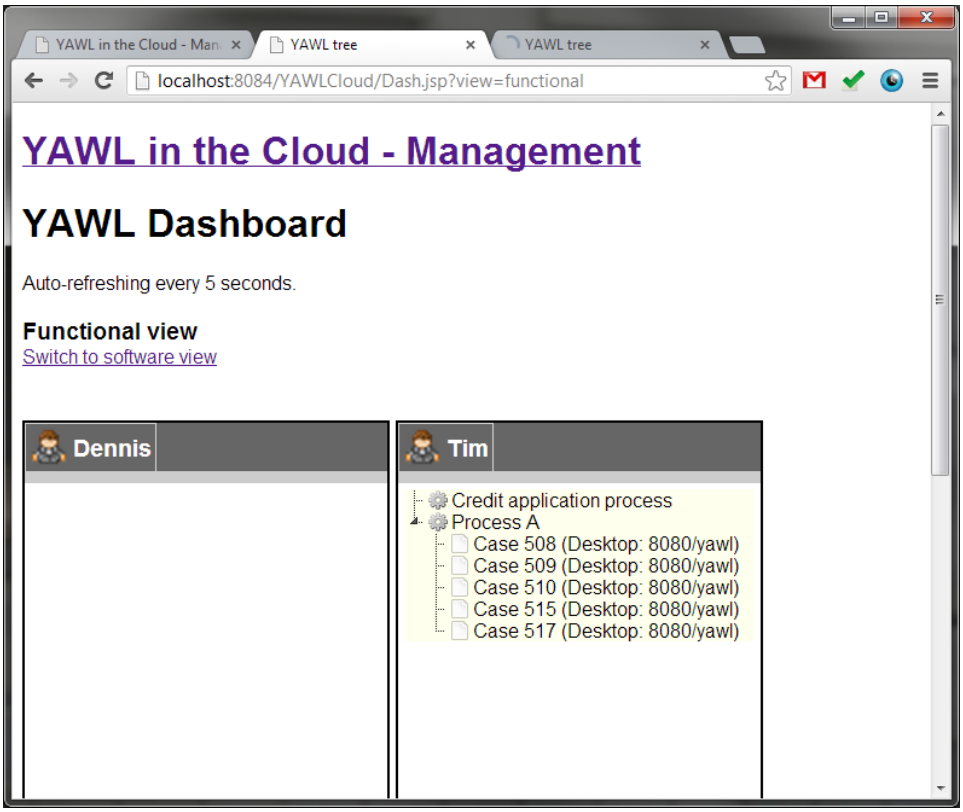


Figure 32: Dashboard in functional view

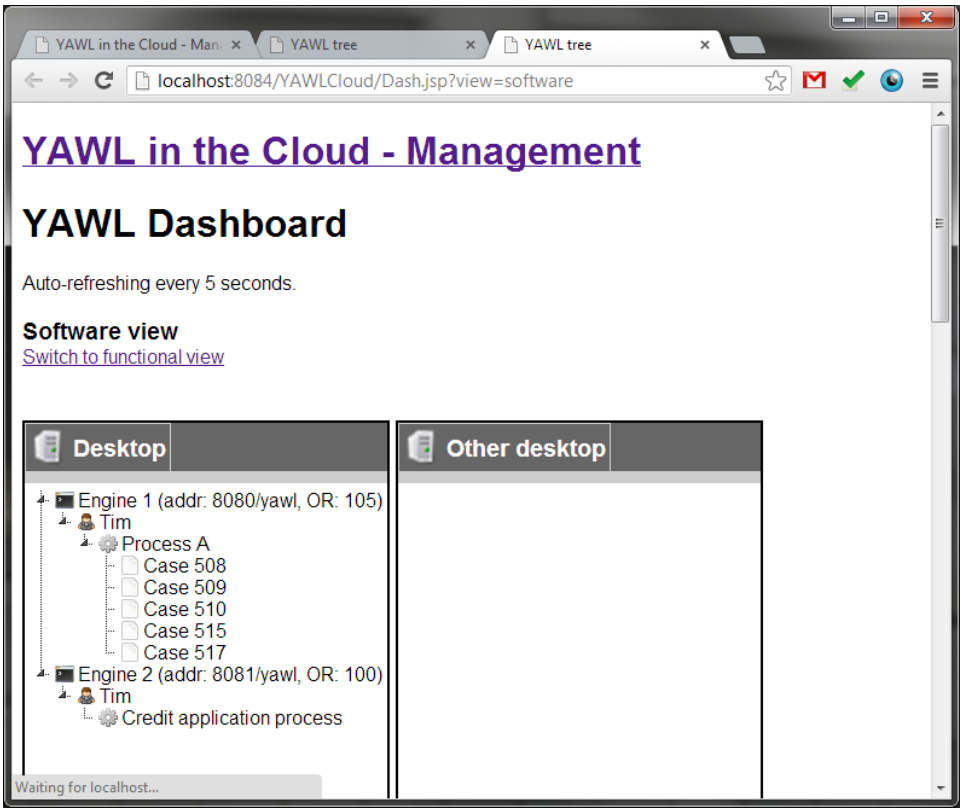


Figure 33: Dashboard in software view

6 Evaluation

In Chapter 1 we identified the goals of this project: a proof of concept architecture and implementation of a BPMS that supports multi-tenancy, is automatically scalable, and supports configurable process models. In Chapter 3, Chapter 4, and Chapter 5 we elaborated on these goals and designed a solution that should lead to achieving them. To verify if the designed solution is *indeed* achieving the set goals, we run tests against the proof-of-concept implementation.

After describing the environment we use to test in, in Section 6.1 we first show that normal usage of the YAWL Engine is working correctly. Then we verify the goal of multi-tenancy in Section 6.2, the goal of automatic scalability in Section 6.3, and the goal of support for configurable process models in Section 6.4. We conclude with an evaluation of the performance in Section 6.5.

Test Environment

For the tests described in this Chapter we deploy four engines. Each engine is deployed to its own virtual server. For this test environment we host all these four virtual servers on one physical server. The engine mapping database, as well as the databases of the engines and resource services, are all hosted in one database server hosted on the same physical machine. All machines can communicate over a virtual network provided by the physical host machine.

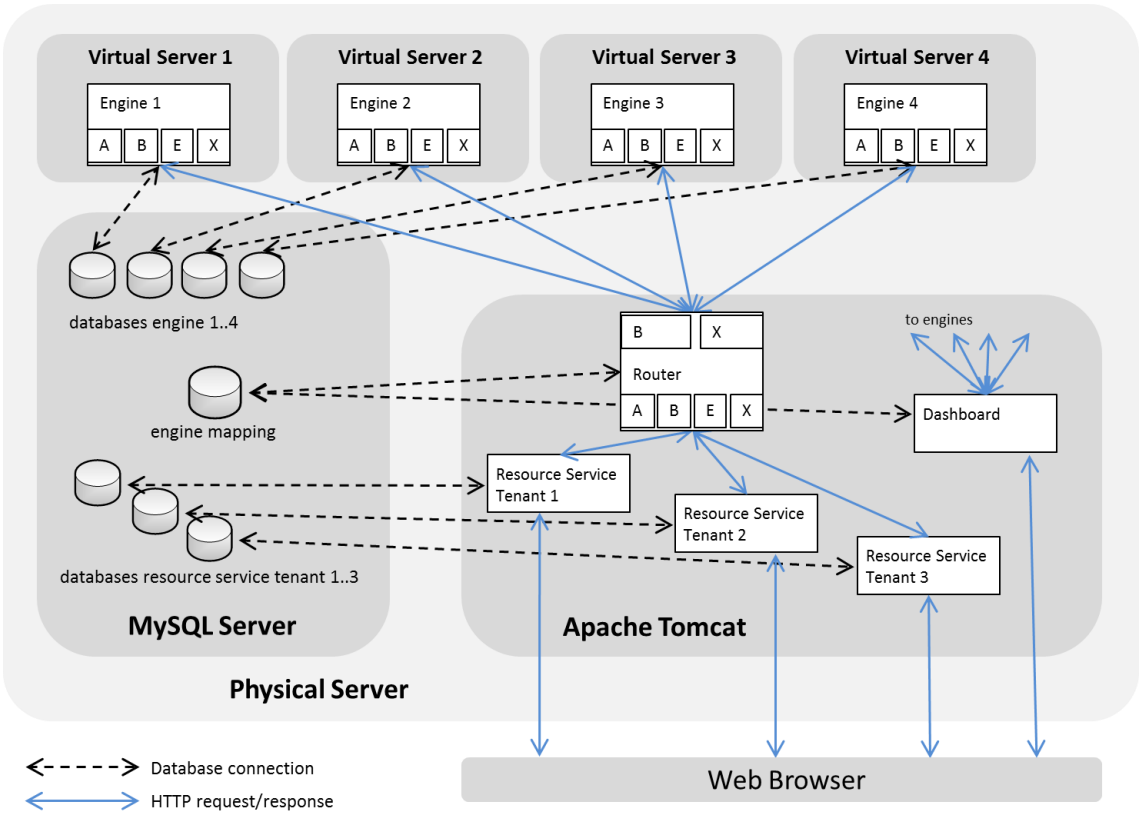


Figure 34: Test Environment setup

On the physical machine we run the Apache Tomcat webserver. The webserver hosts both the Router In, Router Out, and YAWL resource service for tenant 1 (Tim), tenant 2 (Dennis), and tenant 3 (Eric). Figure 34 shows a graphical representation of the test environment.

Test Environment to Cloud Production Environment

The test environment setup can easily be transformed into a real cloud production environment setup. The following transformation steps are required:

- Move all web applications, as highlighted in blue in Figure 35, to their own virtual server. This server can either be an IaaS cloud-based server, on which an operating system and Tomcat would have to be installed, or a PaaS server, where only the web application itself has to be deployed. (For IaaS and PaaS see Section 2.2.2.)
- Move all databases, as highlighted in blue in Figure 35, to cloud-based infrastructure. This can be done by setting up an IaaS cloud server, on which an operating system and MySQL Server would have to be installed, or by using a SaaS managed cloud-based database service. (For IaaS and SaaS see Section 2.2.2.)
- Whereas in our test environment setup we have to create and configure all (virtual) servers by hand, in a cloud production environment this can be automated. The Dashboard component would then call the API of the IaaS, PaaS, or SaaS service to obtain or dispose resources. Without this conversion step the setup will still work, but can then only scale up to the number of engines that was configured in advance by hand.

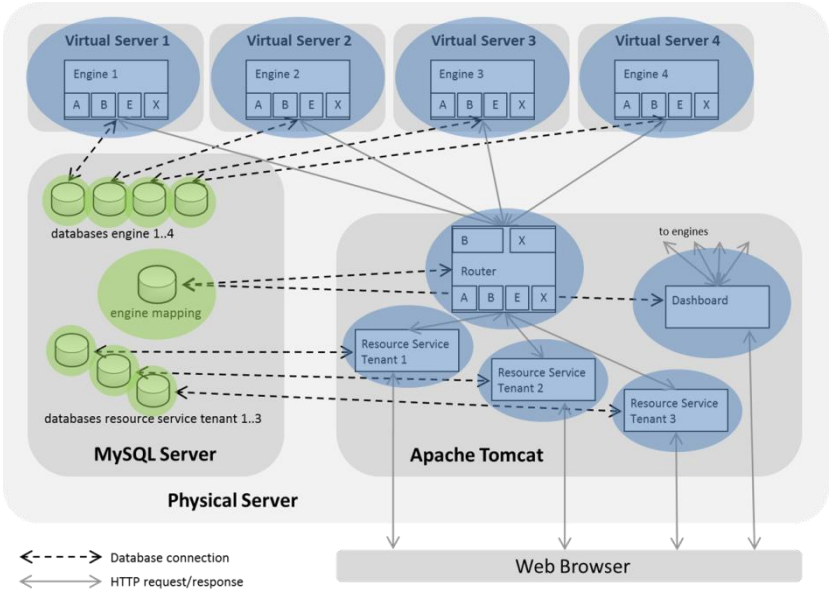


Figure 35: Test environment to cloud production environment transformation

For Section 6.1, and Section 6.2 we use the following settings:

Table 10: Configuration settings for tests

Parameter	Value
Metric	Count Based
Specification weight ($sw(i)$)	100
Max. Loaded per Engine	2
Max. Loaded per Loadable	2
Max. Engine per Tenant	2
Max. Case per Loaded	3

6.1 Existing functionality

YAWL in the Cloud must, besides achieving its three main goals, still function correctly in normal single-tenant one-engine situations. In this Section we show that using YAWL in the Cloud, we can still *upload a specification*, *start a case* for that specification, *complete a work-item* for the started case, can *cancel a case*, and can *unload the specification*.

We look at the resource service web interface of tenant 1. Before uploading a specification, the list of specifications is empty. After uploading the specification CreditAppProcess2.0 this specification is correctly shown in the list. (For the process model of CreditAppProcess2.0 see Section C.1.) Next we click Launch Case, and we see that the new case is now shown in the list of cases, which previously was empty.

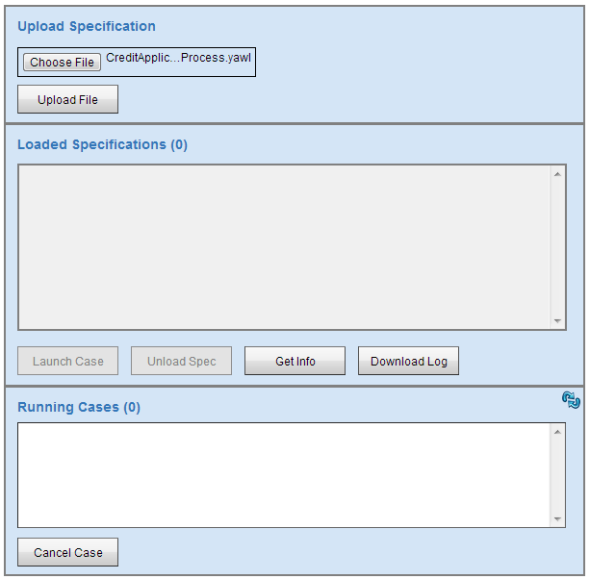


Figure 36: Before upload specification

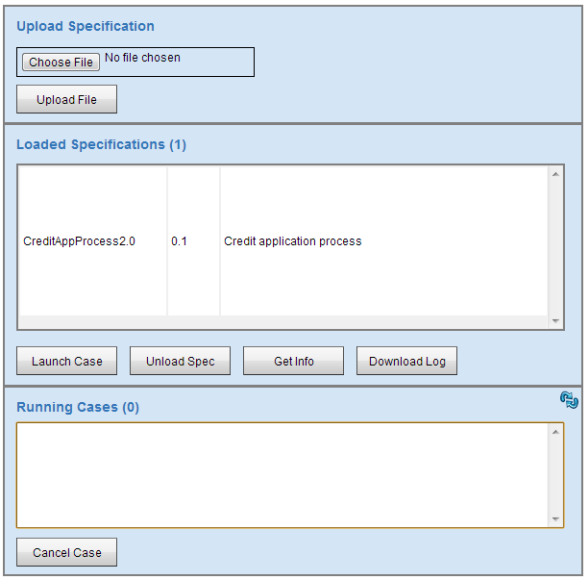


Figure 37: After upload specification

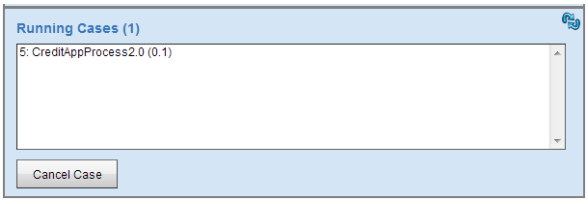


Figure 38: After launch case

When we navigate back to the Admin Queues page, we see the work-item receive application on the unoffered tab. We start the work-item 'receive application' for User One. When we now log in as User One, we see the work-item on the started tab. We click View/Edit, fill in an amount, and click Complete. We log back in as administrator, and now see the next work-item 'check for completeness' listed on the Unoffered tab. Next we click Cancel Case and successfully see an empty list of cases again.

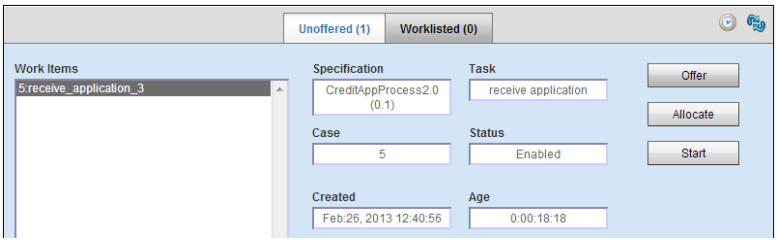


Figure 39: Unoffered work-item

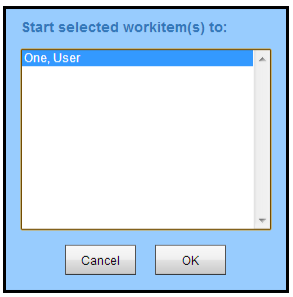


Figure 40: Start work-item for User One

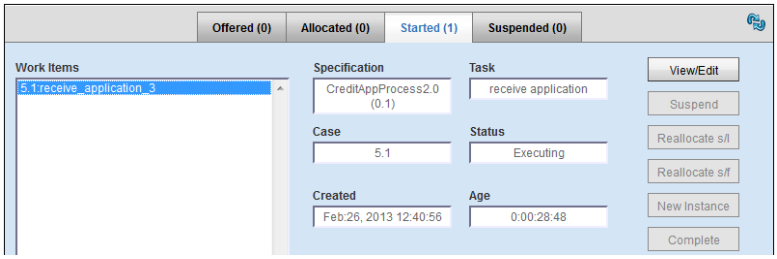


Figure 41: User One sees started work-item

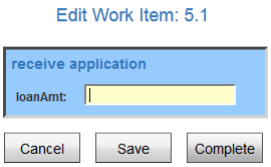


Figure 42: User One completes work-item

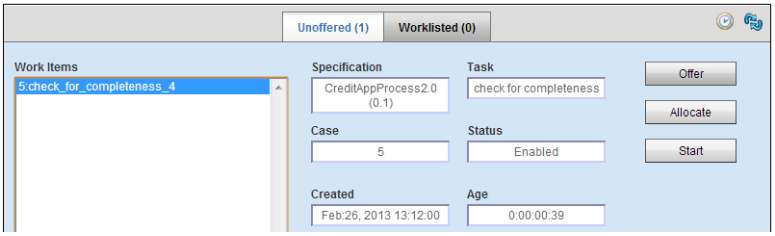


Figure 43: Next work-item is shown in unoffered list

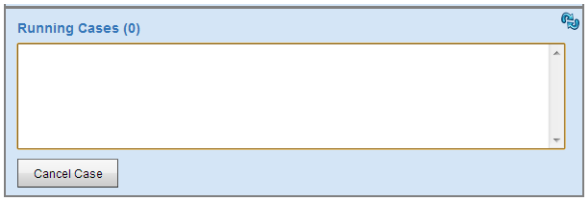


Figure 44: No running cases after cancel case

Finally, we unload the specification, and see Figure 36 again.

6.2 Multi-tenancy and Cardinality Restrictions

In Chapter 3 we identified three new ways of spreading tenants, specifications and cases over multiple engines. In this Section we evaluate if, for all three, this works as intended. Next, we test if the cardinality restrictions, introduced in Section 4.3.3, which define which workload distributions are valid and which not, are enforced.

We start in the empty system state depicted in Figure 45, and have three resource services configured for the tenants shown in Figure 46.

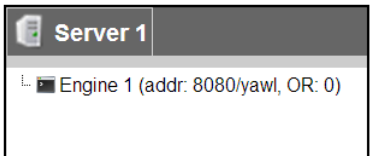


Figure 45: System state before multi-tenancy tests

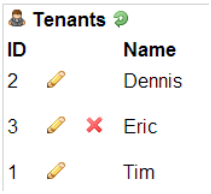


Figure 46: Tenant IDs and names

Multiple Tenants on One Engine

Figure 13 depicted the requirement of serving multiple tenants from one engine. Using the resource service of Tenant 1, we upload the credit application process model, and start one case. Similarly, using the resource service of Tenant 2, we upload the film production process model, and again start one case.

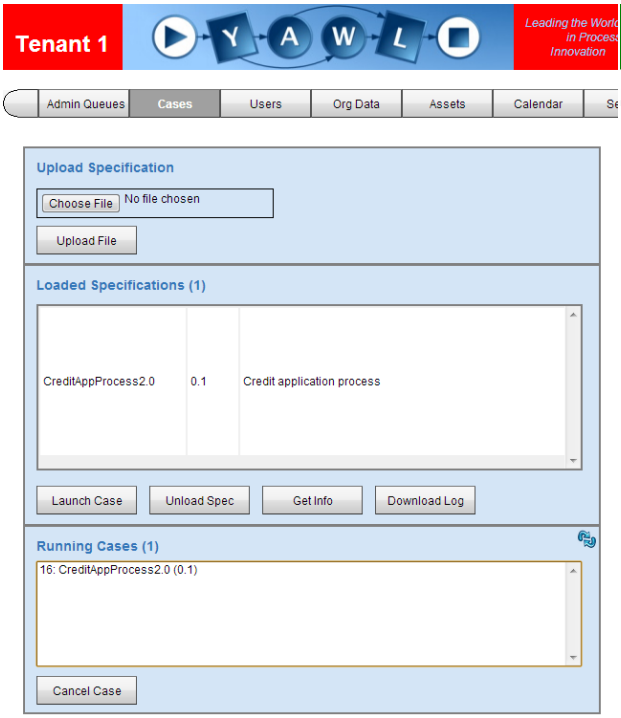


Figure 47: Resource service of tenant one, after uploading CreditAppProcess2.0 and starting one case

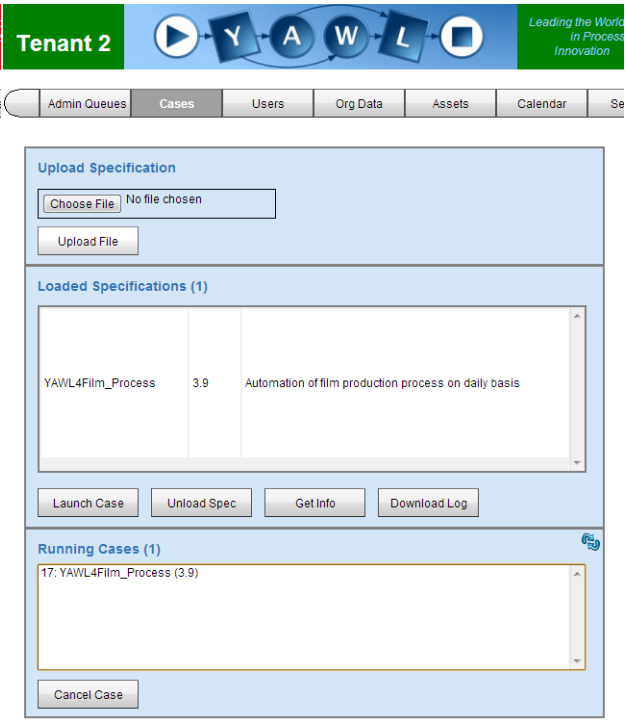


Figure 48: Resource service of tenant two, after uploading YAWL4Film_Process and starting one case

Afterwards we look at the dashboard in software view, and indeed see that both specifications and cases are in fact running on the same engine.

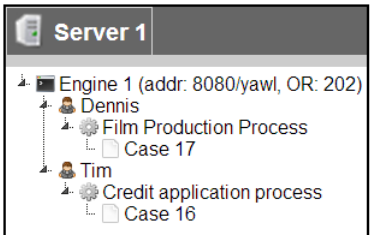


Figure 49: Two tenants running on one engine

One Tenant on Multiple Engines – Specifications

Figure 14 depicted the requirement of spreading specifications of one tenant over multiple engines. To evaluate if this requirement is accomplished, we start by enabling the empty Engine 2. If we now upload the specification called Process A for Tenant 1, it must be allocated to Engine 2, since Engine 2 has the lowest occupancy rate. We upload the specification, and start a case. In the specification and cases list of the resource service of Tenant 1, we see the additional specification and case, as depicted in . On the dashboard we see that Process A indeed was allocated to Engine 2, as depicted in . Thus, we conclude that specifications of one tenant can indeed be spread over multiple engines.

Figure 50: Resource service of Tenant 1, showing two specifications with each one case

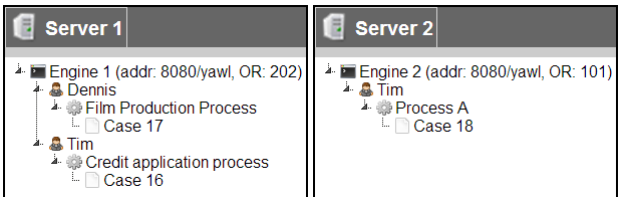


Figure 51: Dashboard showing that the two specifications of Tenant 1 are allocated to different engines

One Tenant on Multiple Engines – Cases of One Specification

Figure 15 depicted the requirement of spreading cases of the same specification over multiple engines. We recall that, as stated in Table 10, we set the maximum number of cases per specification loaded on an engine to three.

First, we launch two more cases of the credit application process. When inspecting the dashboard, we see that these cases are allocated to Engine 1.

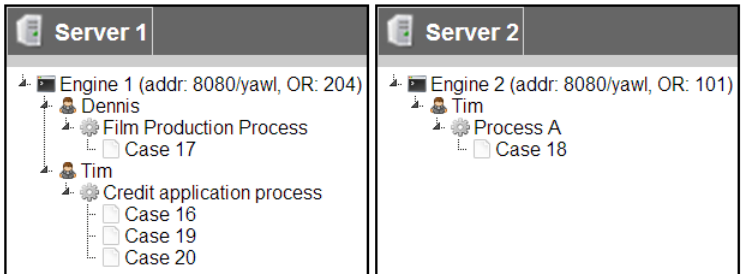


Figure 52: All cases of the credit application process run on the same engine

Next, we launch two more case of the credit application process. When inspecting the dashboard again, we see that the new cases are allocated to Engine 2. We conclude that cases of the same specification can indeed be spread over multiple engines.

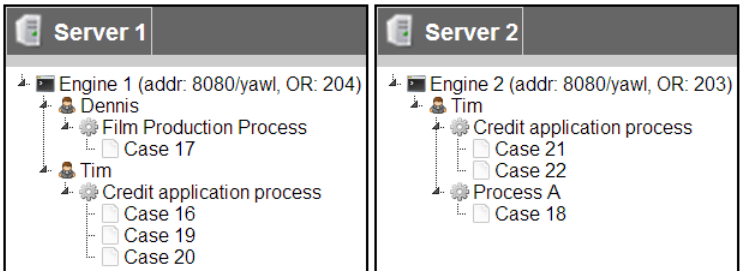


Figure 53: Cases of the credit application process run spread over two engines

Cardinality Restriction Enforcement

Figure 24 depicted the requirement of being able to enforce cardinality restrictions. In the previous Section we already saw that cardinality restrictions forced the new cases to be allocated to another engine. Ultimately, no more valid allocations are available, and the launch of a new case is rejected.

We recall that, as stated in Table 10, we configured that an engine may serve a maximum of two specifications, and that a specification may only be loaded onto two engines. We can successfully add one extra case of the credit application process, which is the last valid allocation. When we then click Launch Case again, we get the following error message, stating that the case could not be started:

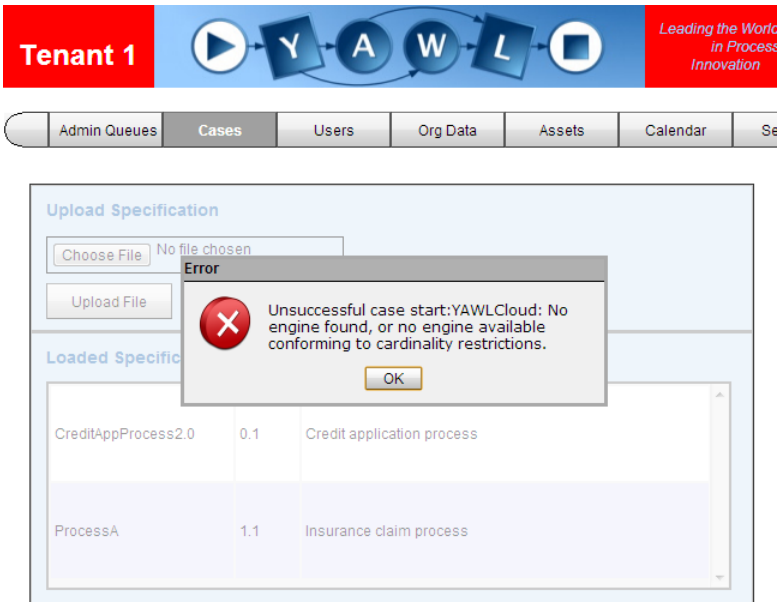


Figure 54: Launching a case is rejected if no more valid allocations exist

We conclude that cardinality restrictions on case allocation are indeed ultimately rejected, when no valid allocations are available anymore.

Also we recall that, as stated in Table 10, we set the maximum number of engines per tenant to two. Combined with the maximum of two specifications loaded on an engine, this makes that we already used all valid specification allocations for Tenant 1. We now try to upload the process model Process B, and see the following:

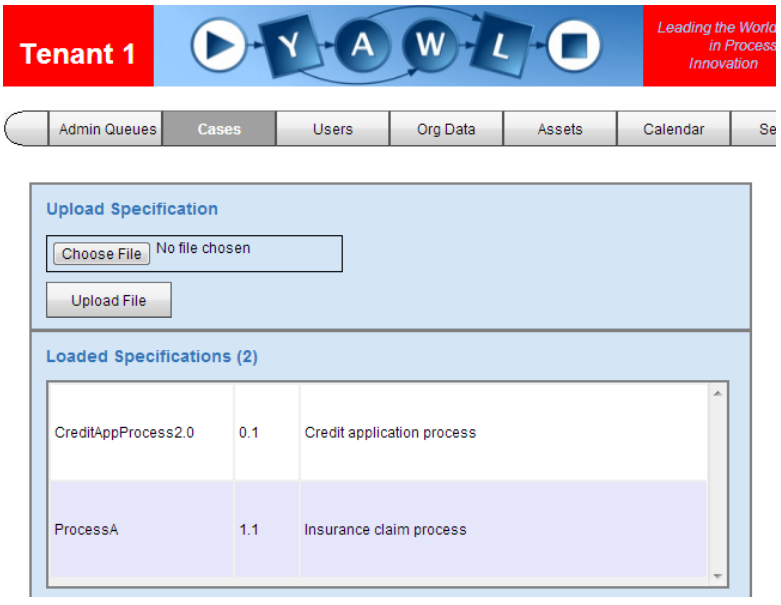


Figure 55: Uploading a specification is rejected if no more valid allocations exist

We see that Process B was not loaded. The resource service has received an error message, just like the error message for rejected case allocations. However, the as-is YAWL resource service, does not show such messages on screen. We conclude that cardinality restrictions on specification allocations are indeed ultimately rejected, when no valid allocations are available anymore.

6.3 Automatic Scalability

Automatic scalability uses the allocation strategies as introduced in Section 4.3. The first element of an allocation strategy is the choice of metric. We start this Section with verifying the functionality of the two metrics we introduced: the count based metric, and the measurement based CPU metric. The second element of an allocation strategy are the threshold values, which we test next.

Occupancy Rate – Count Based Metric

We start with taking a look at the system state depicted in Figure 53, and recalling that sw(i) was set to 100 according to Table 10. As a postfix of the engine name, we see that Engine 1 has occupancy rate 204, and that Engine 2 has occupancy rate 203. Given the fact that Engine 1 runs two specifications and four cases, and that Engine 2 runs two specifications and three cases, we conclude that this is correct, according to the definition of the count based metric.

Occupancy Rate – Measurement Based CPU Metric

Next, we switch the occupancy rate metric setting to the measurement CPU based metric. We compare the reported 4% average over the last 10 seconds, with the CPU usage history shown on the server. We conclude that the reported 4% is an accurate reflection of the shown history.

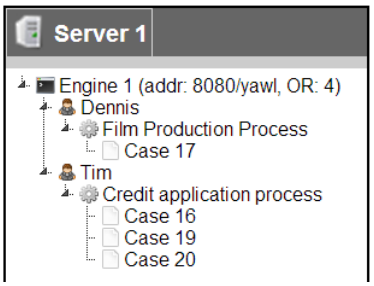


Figure 56: CPU based metric reports 4%

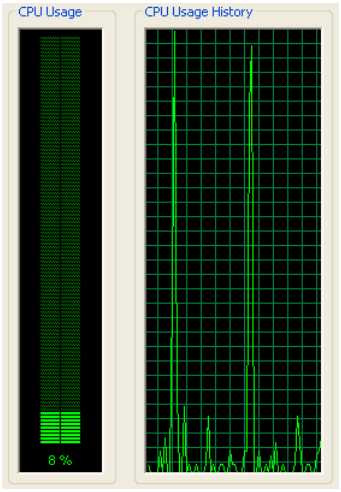


Figure 57: CPU meter and history on (virtual) server

We now switch back to the count based metric.

Threshold State Actions

As the value of the occupancy rate metric goes through the different threshold states, the system must take the corresponding actions. In this Section we test the *out-of-work*, *re-allocate*, and *reinforcement* actions. We remove all cardinality restrictions set for the previous Sections. We set the threshold values as depicted in Figure 58.

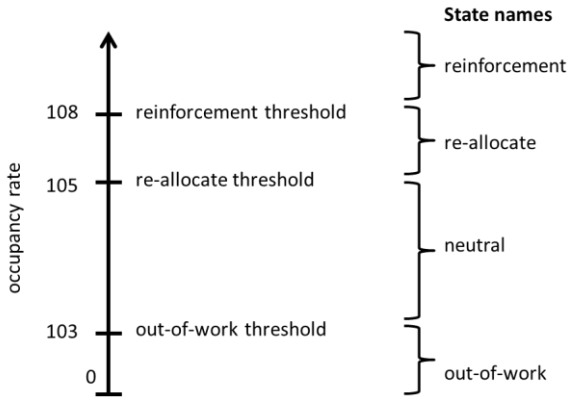
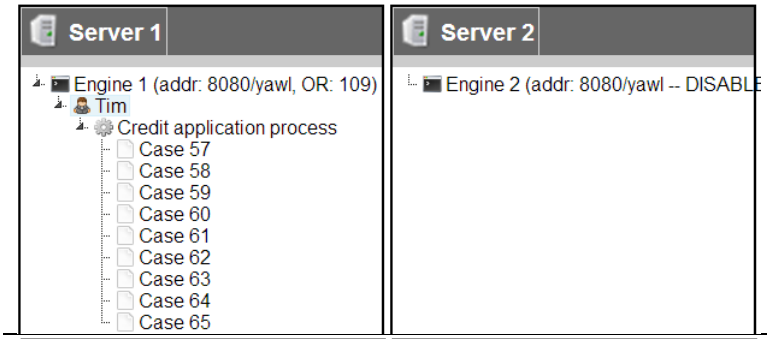
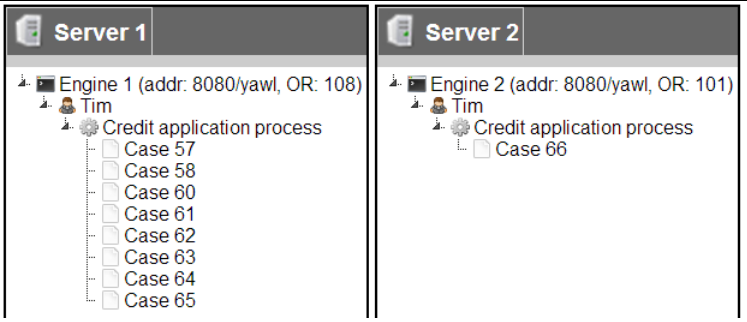
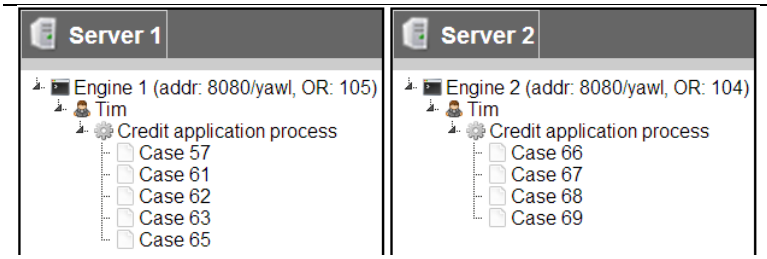
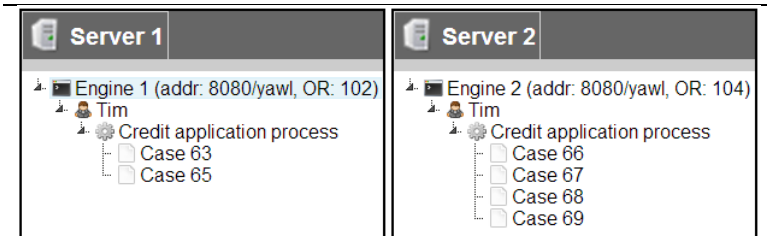
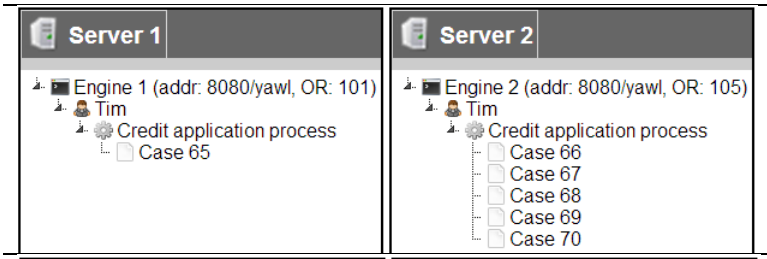
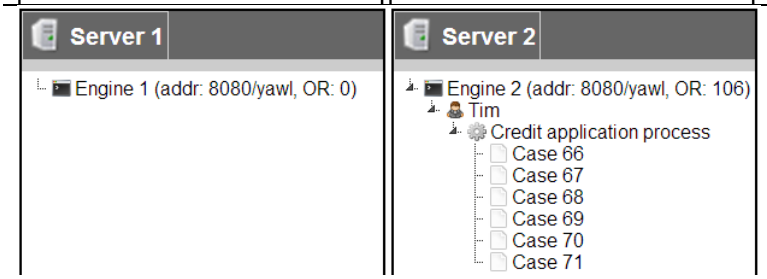


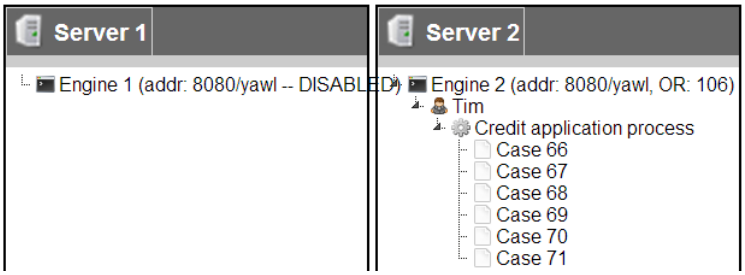
Figure 58: Threshold values for test

Note that transferred cases receive new case numbers. Since the as-is YAWL engine does not support transferring cases, we simulate transferal of cases by programmatically starting a new case on the engine the case is transferred to, and cancelling the case on the engine the case came from.

We now follow a scenario that brings us through all threshold states, and all according actions:

Dashboard state	**Note,** **Acting upon state** (read from logs)
Server 1 Engine 1 (addr: 8080/yawl, OR: 100) Tim Credit application process Server 2 Engine 2 (addr: 8080/yawl -- DISABLE	We start with Engine 1 enabled and Engine 2 disabled. Engine 1 runs the credit application specification. Engine 1: out-of-work Engine 2: disabled
Server 1 Engine 1 (addr: 8080/yawl, OR: 102) Tim Credit application process Case 57 Case 58 Server 2 Engine 2 (addr: 8080/yawl -- DISABLE	Using the resource service we launch two cases. Engine 1: out-of-work Engine 2: disabled
Server 1 Engine 1 (addr: 8080/yawl, OR: 103) Tim Credit application process Case 57 Case 58 Case 59 Server 2 Engine 2 (addr: 8080/yawl -- DISABLE	We launch one more case. Engine 1 is not below the out-of-work threshold anymore. Engine 1: neutral Engine 2: disabled
Server 1 Engine 1 (addr: 8080/yawl, OR: 106) Tim Credit application process Case 57 Case 58 Case 59 Case 60 Case 61 Case 62 Server 2 Engine 2 (addr: 8080/yawl -- DISABLE	We launch three more cases. The last case brings Engine 1 above the re-allocate threshold. Before the action delay is expired, Engine 1 stays in neutral state. After, the Engine 1 is in re-allocate state, but since no engines with a lower occupancy rate are available, no cases are offloaded. Engine 1: re-allocate Engine 2: disabled

	After launching another three cases, Engine 1 is above the reinforcement threshold. The action taken is re-allocate, until the action-delay timer has expired. Engine 1: reinforce Engine 2: disabled
	The action-delay timer expired. The reinforcement action is taken: Engine 2 gets enabled. A case gets offloaded to Engine 2. For at least the re-reinforcement delay, the state of Engine 1 cannot exceed re-allocate. Engine 1: re-allocate Engine 2: out-of-work
	During the next inspections of the occupancy rate, more cases are offloaded to Engine 2, until the occupancy rate of Engine 1 is not above the re-allocate threshold anymore. Engine 1: neutral Engine 2: neutral
	We finish three cases that were running on Engine 1, which then falls below the out-of-work threshold. No action is taken as long as the action delay has not expired. Engine 1: neutral Engine 2: neutral
	The action delay has expired. A case gets offloaded onto Engine 2. Engine 1: out-of-work Engine 2: neutral
	Engine 1 is still out-of-work; its last case gets offloaded to Engine 2. Engine 2 gets above the re-allocation threshold, but the action delay has not expired yet. Engine 1: out-of-work Engine 2: neutral



Engine 1 was empty and gets disabled. Engine 2 is now in re-allocate state. Since Engine 2 is not above the reinforcement threshold, Engine 1 is not enabled again.

Engine 1: disabled
Engine 2: re-allocate

Figure 59: Scenario through all threshold states and actions

We conclude that we have correctly seen the following actions:

- Threshold state classification updates.
- Reinforcement with an extra engine.
- Offloading cases to another engine in the re-allocation state.
- Offloading cases to another engine in the out-of-work state.
- Unloading an empty specification from an engine, when another engine is still running that specification.
- Disabling an empty engine.

6.4 Configurable Process Model Support

YAWL in the Cloud must support configurable process models. YAWL models can be configurable, so called, c-YAWL models. However, the as-is YAWL engine cannot execute c-YAWL models. Fully configured c-YAWL models can be converted to executable YAWL models. All three models are XML files.

YAWL in the Cloud can administrate the link between these three models: configurable specifications, specification configurations, and loadable specifications. In the dashboard the XML content can be added, and the links can be selected using a dropdown selection.

A disadvantage of the current YAWL in the Cloud implementation is that it is not integrated with the c-YAWL configuration tool. As a result, the user has to run the tool, and administrate the corresponding XML files by hand, using the web user interface. Figure 60 shows the list of configurable specifications, together with the stored configurations for those configurable specifications. Figure 61 shows how a loadable specification can be linked to one of these configurations.

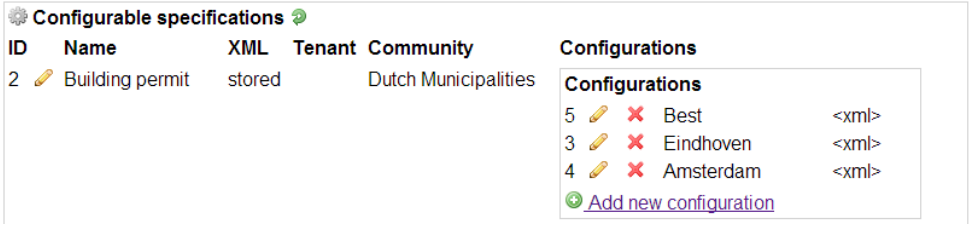


Figure 60: The list of configurable specifications, together with the available configurations

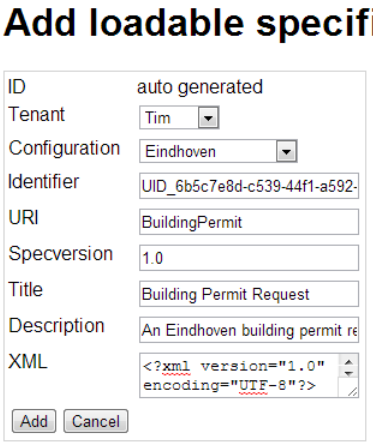


Figure 61: Configuration is selectable during administration of loadable specification

6.5 Performance Evaluation

We conclude this Chapter with a performance comparison of classic YAWL with YAWL in the Cloud. A test was constructed, which was run both against classic YAWL and YAWL in the Cloud, in the four engine setup depicted in Figure 34.

Test Setup

The test is executed using Apache JMeter; see Section C.2 for a description of JMeter. The test definition mimics the actions sent to the engine by the resource service, when the user launches a case, starts and completes a work item, and cancels a case. This results in a number of actions performed on interface B (some of which occur multiple times):

- checkConnection
- getAllRunningCases
- getSpecificationPrototypesList
- launchCase
- checkout (of a work item)
- getChildren (of a work item)
- getResourcingSpecs (of a work item)
- taskInformation
- getSpecificationDataSchema
- checkin (of a work item)
- getWorkItemsWithIdentifier
- cancelCase

There is a Poisson random timer between all actions performed on the interface, with a delay of 1000ms and lambda 500ms.

This scenario is run in parallel for 10 simultaneous users, each of which repeat the scenario for 20 times. There is a 30 seconds ramp-up from 0 to 10 simultaneous users to make sure the different requests are interleaved from the start. Section C.3 shows a screenshot of the test definition.

Test Execution Results

We start by comparing the overall response times:

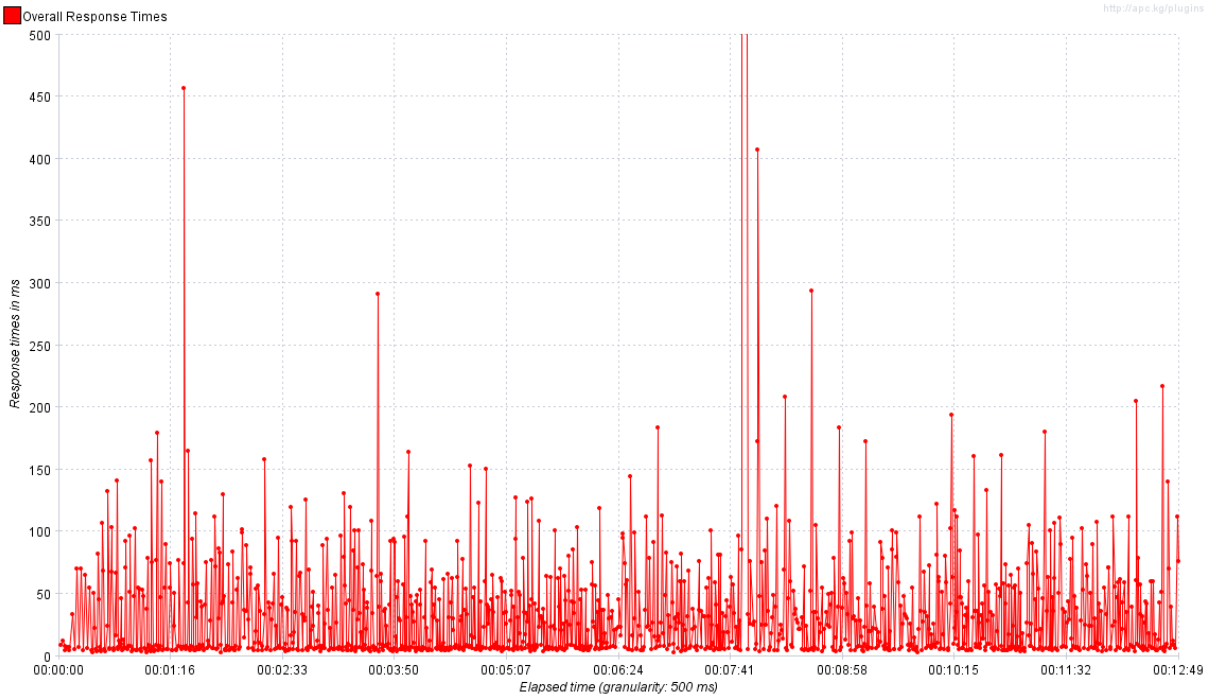


Figure 62: Overall response times of classic YAWL

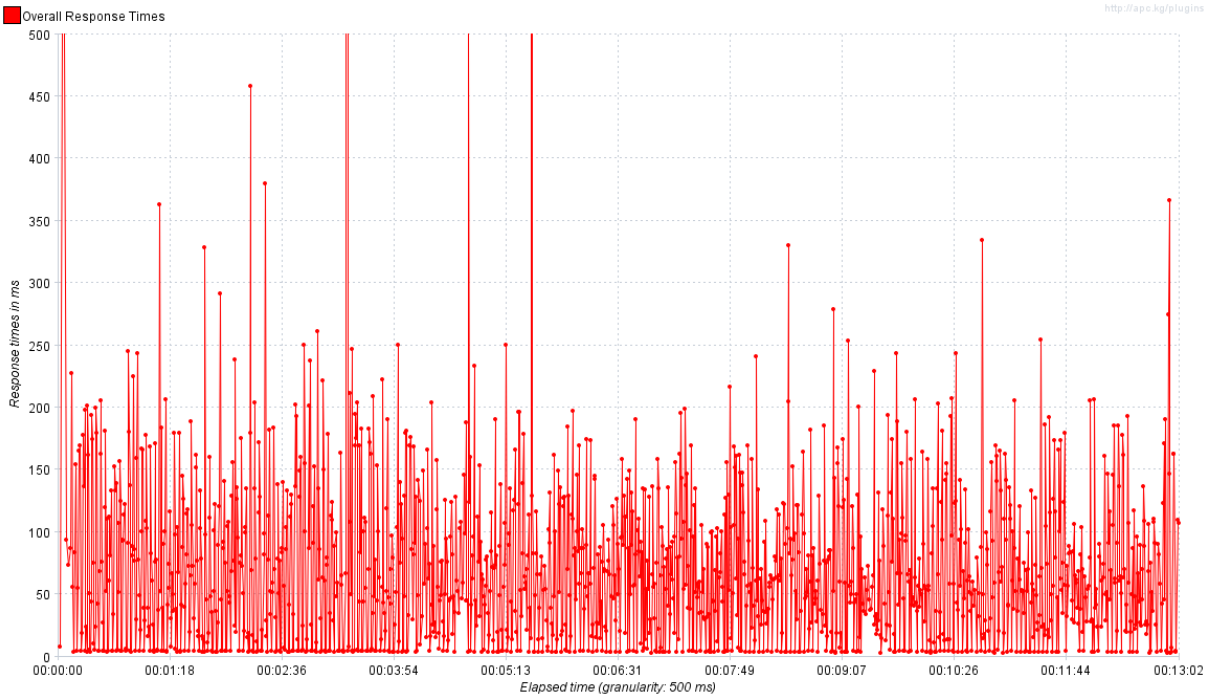


Figure 63: Overall response times of YAWL in the Cloud

Both graphs use the same scale. We see that YAWL in the Cloud performs different from classic YAWL: overall response times are higher.

To see where the differences can be found, we drill down on the individual actions performed on the interface using a percentile graph:

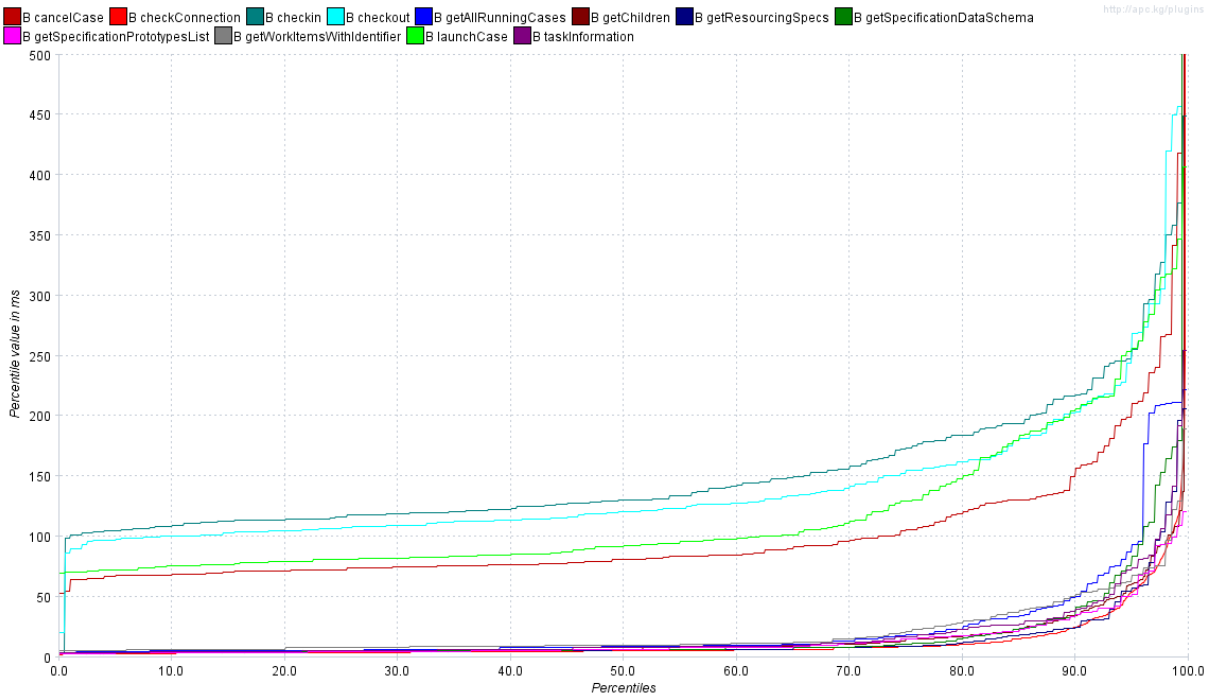


Figure 64: Percentile graph of classic YAWL

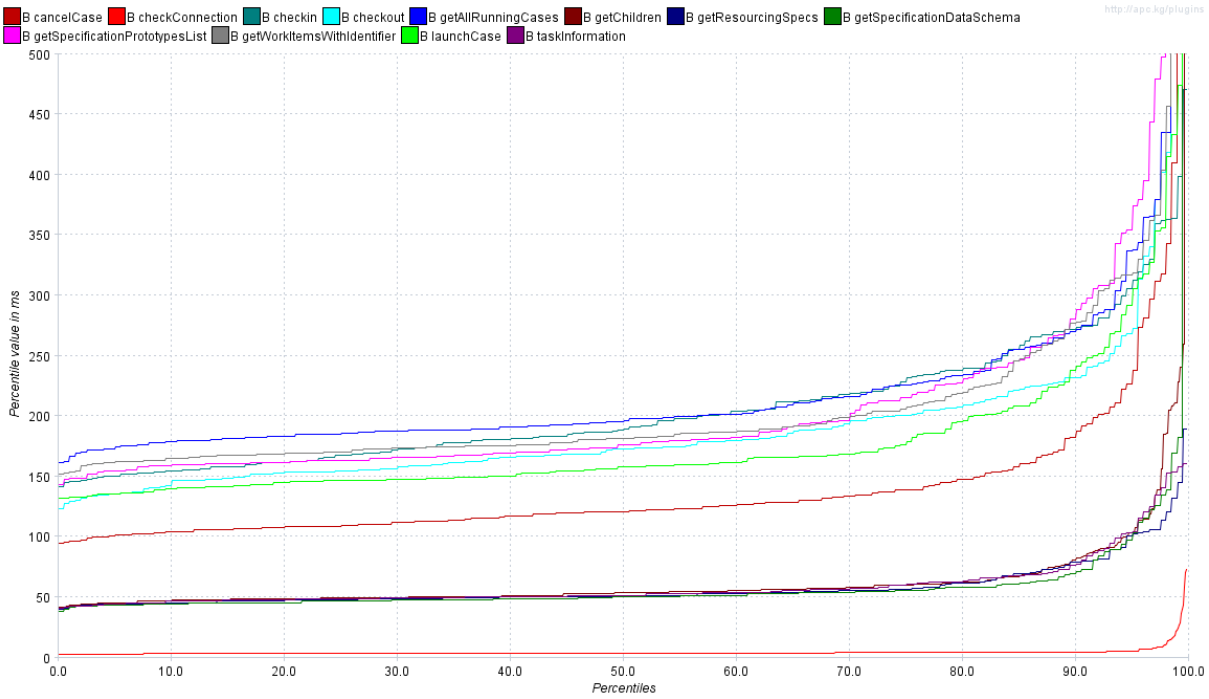


Figure 65: Percentile graph of YAWL in the Cloud

Again, both graphs use the same scale. First of all we see that response times of YAWL in the Cloud are higher. This is caused by the overhead in CPU and network communication of the architecture. In this setup, this disadvantage not overcome by the benefit of scale the four engines of YAWL in the Cloud provide.

More interestingly, we see that the individual interface actions have changed characteristics. Whereas in classic YAWL the actions *checkin*, *checkout*, *launchCase* and *cancelCase* are significantly slower than all other actions, we see that for YAWL in the Cloud this list is supplemented with the actions *getAllRunningCases* and *getWorkItemsWithIdentifier*. The cause is clear however: these are the two actions that have to be sent to multiple engines; the other requests are only sent to one engine.

The fact that YAWL in the Cloud has higher response times is of no surprise, since we introduced overhead. However, besides the fact that we introduced overhead, the proof of concept implementation was focussed on functional requirements, and can likely be optimized for performance. We conclude that the performance of the current proof of concept implementation of YAWL in the Cloud is worse than of classic YAWL. Furthermore we conclude that, due to the different routing constructs, the ratio of response times of the various interface actions has changed.

7 Conclusion

In this Chapter we summarize and conclude on the project BPM in the Cloud. Since this project is a first step into a new field of interest, in the next Section we give a list of possible future work, both in research and in implementation.

7.1 Accomplished Results

The field of BPM is not broadly represented in cloud-based technology offerings. During this project, we devised steps of a non-cloud-based BPMS, into the new world of cloud-based opportunities. We did so because the typical characteristics of cloud computing perfectly matched the requirements set for a BPMS to support the CoSeLoG project. Three goals were set: achieving multi-tenancy, automatic scalability, and configurable process model support.

To achieve multi-tenancy we introduced a mapping database, and devised multiple processing steps to be able to link multiple tenants and engines together. Three desired ways of distributing work over engines were defined: serving multiple tenants from one engine, serving different specifications of a tenant from different engines, and spreading cases of a specification over multiple engines running that specification. The approach was successful, as all three desired ways of distributing work are present and functioning in our proof of concept.

An architecture intended for cloud-based deployment would not be worth the label cloud-based if it could not scale automatically. We showed that our mechanism, which determines engine status based on a defined metric, can be used for automatic up-scaling, down-scaling, and re-allocation within the current scale. For the sake of resource usage optimization, scaling down can be as interesting as scaling up to an enormous system. A community of tenants can share the automatically scalable infrastructure.

The goal of configurable process model support was mostly covered by selecting YAWL, which has configurable process model syntax and an accompanying toolset. We only had to provide some administrative features, so the relationships between configurable models, configurations, and executable models can be documented.

Looking back, the pre-set constraint of having to use the BPM engine as-is, may be reconsidered, as for performance, as well as for simplicity of the architecture, making small changes to the engine can have multiple benefits, such as the as making translation unneeded.

Performance, in the sense of response times, is not the strongest point of our proof of concept. We introduced a significant amount of overhead in processing time and communication latencies over the network. The implementation did however include many different facets of bringing BPM to the cloud; we constructed: a mapping database, routers components, administrative web pages, a monitoring dashboard, and workload (re)distribution.

Altogether, we conclude that with the project BPM in the Cloud, and the proof of concept implementation of YAWL in the Cloud in particular, we made many initial steps into the relatively new field of cloud-based BPM.

7.2 Future Work

The project of bringing YAWL to the cloud is a first step into a new field of interest. As a first exploration it gives answers to some questions, such as “can it be done?”. But even more, it leads to new challenges, research opportunities, and questions. In this Section we propose multiple directions for possible future work.

Proof-of-Concept Implementation

The current proof-of-concept implementation is focussed on functionality and conceptual structure. Although none of its parts were consciously implemented inefficiently, additional effort can be spent on profiling its performance, and improving it where profitable.

After optimizing performance of the current design, additional performance can be found by investigating the balance between performance and genericness. The current design lets the engines do all the work, to minimize the level of domain knowledge that has to be implemented. Some of the actions on the interfaces can theoretically be answered by consulting the database of YAWL in the cloud. We did not do so to keep the solution more generic, but if performance is more important, this choice can be reconsidered.

The dashboard can be improved by implementing more functions. More administrator functionality can be added, for example to automate the creation of new tenant resource services, which is now done manually in Apache Tomcat. Furthermore, the visual appearance can be stylized.

Where servers and engines have to be deployed by hand now, after which YAWL in the cloud can decide to use them or not, this process can be automated by programming against the APIs provided by cloud providers, when run on real cloud-based infrastructure.

YAWL Engine

Extracting a case from an engine is currently not supported by the interfaces of the YAWL engine. When we currently transfer a running case from one engine to another engine, we effectively cancel the case on the one engine, and start a new case on the other engine. The YAWL engine has an internal mechanism to persist cases and reload them later on. This mechanism can be extended to provide this functionality over one of the interfaces.

In this project we use the YAWL engine as-is. This was a predefined constraint, and results in our design that builds around classic YAWL engines. It would be interesting to do a similar project without this constraint. If the internal workings of the engine can be changed, some of the overhead we introduced can be removed. For example, request and response translation can be omitted if the engines internally work with global identifiers, instead of engine-specific identifiers.

Strategy Optimization

The introduced allocation strategies and the accompanying thresholds introduce a whole new optimization problem. Questions that can be investigated are: Which occupancy rate metric works best? What are the optimal parameter values (e.g. case weight) of the occupancy rate metrics? What are the advised threshold values, and what are the optimal threshold values for specific deployments? Or, can better new metrics be devised altogether? These optimization questions alone can already take up a significant amount of work.

Extension

For now we made a design for the backbone in the YAWL architecture: the YAWL engine. The YAWL *resource service* is another important part of this architecture. It incorporates, for example, the entire web interface that end-users see, and administration for the organisational models. We did not yet include this service in the design, and as such we are currently using one resource service per tenant. This resource service is pointed at the virtual interface endpoints for that single tenant.

The implementation can be extended to include the resource service, either as-is by repeating the same procedure of building around it, or by changing its internals to be able to cope with multiple tenants or distribute workload. When such a design is finished and implemented, the tenants do not have to navigate to their own private resource service URL anymore, but browse to a general login page and work from there; just like people are used to when using e.g. cloud based webmail.

Other BPMS Engine

The architecture devised for YAWL in the Cloud is, apart from implementation details, not strictly coupled to YAWL only. It would be interesting to try to apply the same architecture to another BPMS engine. Doing so can strengthen and refine the architecture.

References

[1] M. La Rosa, “Process Configuration,” Queensland University of Technology, [Online]. Available: http://www.processconfiguration.com/. [Accessed 27 April 2012].

[2] “CoSeLoG,” Eindhoven University of Technology, [Online]. Available: http://www.win.tue.nl/coselog/. [Accessed 21 April 2012].

[3] Google, “Join the 5 million businesses using Google Apps,” Google Apps for Business, [Online]. Available: http://www.google.com/enterprise/apps/business/. [Accessed 10 November 2012].

[4] Salesforce.com Europe, “CRM - The Enterprise Cloud Computing Company,” Salesforce, [Online]. Available: http://www.salesforce.com/eu/. [Accessed 10 November 2012].

[5] “Gartner Survey Shows 40 Percent of Respondents with BPM Initiatives use Cloud Computing to Support At Least 10 Percent of BPM Business Processes,” Gartner, 17 February 2011. [Online]. Available: http://www.gartner.com/it/page.jsp?id=1550514. [Accessed 21 April 2012].

[6] W. M. P. v. d. Aalst, A. H. M. t. Hofstede and M. Weske, “Business Process Management: A Survey,” in *Proceedings of the 2003 international conference on Business Process Management (BPM'03)*, Springer-Verlag, Berlin, 2003.

[7] W. M. P. v. d. Aalst, M. Adams, A. H. M. t. Hofstede and N. Russell, “Introduction,” in *Modern Business Process Automation - YAWL and its Support Environment*, A. H. M. t. Hofstede, W. M. P. v. d. Aalst, M. Adams and N. Russell, Eds., Springer-Verlag, 2010, pp. 3-19.

[8] “Workflow Patterns Home Page,” Eindhoven University of Technology & Queensland University of Technology, [Online]. Available: http://www.workflowpatterns.com/. [Accessed 20 June 2012].

[9] R. T. Fielding, “Representational State Transfer (REST),” University of California, Irvine, 2000. [Online]. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. [Accessed 20 June 2012].

[10] A. H. M. t. Hofstede, W. M. P. v. d. Aalst, M. Adams and N. Russell, Eds., Modern Business Process Automation: YAWL and its Support Environment, Springer-Verlag, 2010.

[11] “Workflow Reference Model,” Workflow Management Coalition, [Online]. Available: http://www.wfmc.org/reference-model.html. [Accessed 20 June 2012].

[12] L. M. Vaquero, L. Rodero-Merino, J. Caceres and M. Lindner, “A break in the clouds: towards a cloud definition,” *ACM SIGCOMM Computer Communication Review,* vol. 39, no. 1, pp. 50-55,

January 2009.

[13] P. Mell and T. Grance, *The NIST Definition of Cloud Computing (SP 800-145),* NIST, National Institute of Standards and Technology, U.S. Department of Commerce, 2011.

[14] Cloud Security Alliance, "Security Guidance for Critical Areas of Focus in Cloud Computing V3.0," 2011. [Online]. Available: https://cloudsecurityalliance.org/guidance/csaguide.v3.0.pdf. [Accessed 2 April 2012].

[15] K. Dahbur, B. Mohammad and A. B. Tarakji, "A survey of risks, threats and vulnerabilities in cloud computing," in *Proceedings of the 2011 International Conference on Intelligent Semantic Web-Services and Applications*, New York, NY, USA, 2011.

[16] L. Wang, G. v. Laszewski, A. Younge, X. He, M. Kunze, J. Tao and C. Fu, "Cloud Computing: a Perspective Study," *New Generation Computing, Springer,* vol. 28, no. 2, pp. 137-146, 2010.

[17] K. Julisch and M. Hall, "Security and Control in the Cloud," *Information Security Journal: A Global Perspective, Taylor & Francis,* vol. 19, no. 6, pp. 299-309, 19 November 2010.

[18] S. Ried, H. Kisker and P. Matzke, "The Evolution Of Cloud Computing Markets," Forrester Research, Inc., Cambridge, USA, July 2010.

[19] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," EECS Department, University of California, Berkeley, 2009.

[20] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica and M. Zaharia, "A view of cloud computing," *Communications of the ACM,* vol. 53, no. 4, pp. 50-58, April 2010.

[21] N. Kratzke, "Cloud Computing Costs and Benefits - An IT Management Point of View," in *Cloud computing and Services Science*, I. Ivanov, M. v. Sinderen and B. Shishkov, Eds., New York, Springer Science + Business Media, 2012, pp. 185-203.

[22] W. Jansen and T. Grance, *Guidelines on Security and Privacy in Public Cloud Computing (SP 800-144),* NIST, National Institute of Standards and Technology, U.S. Department of Commerce, December 2011.

[23] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems, Elsevier,* vol. 25, no. 6, pp. 599-616, June 2009.

[24] W. M. P. v. d. Aalst, "Configurable Services in the Cloud: Supporting Variability While Enabling Cross-Organizational Process Mining," in *Lecture Notes in Computer Science - On the Move to Meaningful Internet Systems: OTM 2010*, vol. 6426, Springer-Verlag, 2010, pp. 8-25.

[25] S. Ried and H. Kisker, “Sizing The Cloud,” Forrester Research, Inc., Cambridge, USA, April, 2011.

[26] “NetBeans,” Oracle Corporation, [Online]. Available: http://netbeans.org/. [Accessed 10 September 2012].

[27] “Hibernate,” Red Hat, Inc., [Online]. Available: http://hibernate.org/. [Accessed 10 September 2012].

[28] “MySQL,” Oracle Corporation, [Online]. Available: http://www.mysql.com/. [Accessed 10 September 2012].

[29] “Apache Tomcat,” The Apache Software Foundation, [Online]. Available: http://tomcat.apache.org/. [Accessed 10 September 2012].

[30] “C3P0 JDBC DataSources/Resource Pools,” SourceForge, [Online]. Available: http://sourceforge.net/projects/c3p0/. [Accessed 10 November 2012].

[31] I. Bozhanov, “jsTree, jQuery tree plug,” [Online]. Available: http://www.jstree.com/. [Accessed 10 September 2012].

[32] “Icons by Lokas Software,” Lokas Software, [Online]. Available: http://www.awicons.com. [Accessed 21 April 2012].

[33] “Apache JMeter,” The Apache Software Foundation, [Online]. Available: http://jmeter.apache.org/. [Accessed 25 September 2012].

[34] “JMeter Plugins at Google Code,” [Online]. Available: http://code.google.com/p/jmeter-plugins/. [Accessed 25 September 2012].

[35] W. M. P. v. d. Aalst, “Business Process Configuration in the Cloud: How to Support and Analyze Multi-tenant Processes?,” in *Web Services (ECOWS), 2011 Ninth IEEE European Conference on*, Lugano, 2011.

[36] W. Voorsluys, J. Broberg and R. Buyya, “Introduction to Cloud Computing,” in *Cloud Computing: Principles and Paradigms*, R. Buyya, J. Broberg and A. Goscinski, Eds., Hoboken, NJ, USA, John Wiley & Sons, Inc., 2011, pp. 3-41.

[37] F. Gottschalk and M. La Rosa, “Process Configuration,” in *Modern Business Process Automation - YAWL and its Support Environment*, A. H. M. t. Hofstede, W. M. P. v. d. Aalst, M. Adams and N. Russell, Eds., Springer-Verlag, 2010, pp. 459-487.

[38] “Google Chart Tools - Google Developers,” Google Inc., [Online]. Available: https://developers.google.com/chart/. [Accessed 10 September 2012].

Some icons in this report are courtesy of [32].

A Ecosystem of Cloud Computing

To give an impression of the ecosystem of available cloud services, a summary was made. The table on the following pages shows this summary. Please note that it is intended to give a quick impression, and that it includes the biggest players in the market, but that the list is not intended to be complete. Examples of services in all three service models have been included. Although effort is spent on getting the details in the table correct, for most providers they are based on their own public (commercial) writings. The table was devised in February 2012. Offerings change very frequently, it is possible that some details did change since.

The Section “Test Environment to Cloud Production Environment” in the introduction of Chapter 6 describes how the offered cloud-based services can be used to deploy the proof of concept implementation in a cloud-based environment.

	IaaS					
Provider	**Virtual Servers**	**API**	**Load Balancer**	**Storage**	**Zones**	**Private/Hybrid cloud**
Amazon Web Services (AWS)	**EC2** Elastic Cloud Compute On-demand instances Reserved instances Spot instances Instance specs can be changed Instantiate using an AMI (Amazon Machine Image) Many Operation Systems available	yes, e.g. for creating instances	**ELB** Elastic Load Balancer, only works with EC2 instances	**EBS** Elastic Block Storage Volumes to attach to EC2 Instances	**Regions** 7 Regions across the continents (e.g. EU-West Ireland) **Availability zones** At least 2 zones per region (3 in EU)	**VPC** Virtual Private Cloud Can extend on-premise data center
Google App Engine (GAE)						**Secure Data Connector** To make behind-the-firewall data accessible to applications within GAE
Microsoft Windows Azure Services Platform	Windows Azure Operating System only.					
Oracle Public Cloud						
Rackspace	**Cloud Servers** Delivers Virtual Machines Various Operating Systems available Various instance sizes available	yes, e.g. for creating instances	**Cloud Load Balancer**	Virtual Server harddisk	**Data centers** 6 in USA 2 in UK 1 in Hong Kong	**Cloud Private Edition** In your own datacenter
Go Grid	Xen-virtualized.	yes, e.g. for creating instances	**F5 Hardware Load Balancers**	Virtual Server harddisk	**Data centers** 2 in USA 1 planned in Amsterdam	
OpenCirrus	Cloud Computing Testbed For research into aspects of service and datacenter management only Resources must be specified in advance Research must be approved first					
OpenNebula	Open Source Toolkit for Data Center Virtualization Management of virtualized data centers to enable on-premise IaaS clouds based on Vmware, Xen and KVM.					Can be extended with external resources (hybrid cloud), can be Amazon EC2 instances
VMware vCloud	Vmware doesn't supply public cloud services directly. Partners build public clouds based on Vmware vSphere virtualization, e.g. (example Vmware vCloud Datacenter Services): Terremark(Verizon), Bluelock, Colt, Dell Services, Optus, SingTel, Softbank.					
RedHat RHEV-M	Red Hat Enterprise Virtualization Manager Like Vmware vCloud this is a virtualization platform that can be used to build IaaS					
RedHat DeltaCloud	Unifies APIs of different cloud providers. Allows to build a multi-provider IaaS	yes, unified				
SalesForce.com						
Google						
Microsoft						

PaaS				SaaS	Conclusion and remarks
Storage	**Database**	**Application Execution**	**Connectivity**	**Applications**	
S3 Simple Storage Service **CloudFront** Local endpoints CDN (26 in total)	**RDS** Managed Relational Database Service for MySQL or Oracle. **SimpleDB** NoSQL data store for smaller data sets (max 10gb) **DynamoDB** Highly scalable NoSQL data store	**Elastic Beanstalk** Java using Apache Tomcat only available in region US East (Virginia)	**SQS** Message Queue Service **SES** Email Sending Service **SNS** Push Notification Service		Suitable for hosting YAWL as-is. Free usage tier in first 12 months, includes some EC2 hours and data transfer.
Datastore High-replication variant Master/slave variant	Datastore only, no relational database system available.	**Google App Engine** Supports languages: Java, Python, Go Can only use supported languages, APIs and frameworks, no OS access	**Task Queue** Perform background work outside of user request. **Mail service**		Not really suitable for hosting YAWL since it would require significant changes. Focus on application only. Free resources up to certain limit.
BLOB storage files **Storage Queue** Persistent messaging between applications **Windows Azure Drive** Virtual Hard Drive attached to a virtual machine **CDN** 24 nodes Content Delivery Network	**SQL Azure** Web Edition (up to 5 gb) Business Edition (up to 150 gb) Like SQL Server technology, but auto-scale **Table Storage** NoSQL database	**Web Roles** IIS7 with ASP.NET, PHP or Node.js **Worker Roles** Host any application, including Apache Tomcat and Java Virtual Machines	**Service Bus Queue** Messaging between systems More functionality than Storage Queue		Might be suitable for hosting YAWL as-is. Real PaaS, but some underlying IaaS visible (e.g. virtual machine access) 3 month free trial
	Oracle database 11g Relational database system	**Java Application Execution** JSP, JSF, Servlet, EJB, JPA, JAX-RS, JAX-WS. WAR and EAR Deployment		e.g. Fusion CRM, Fusion HCM	Focussed on a starting point that already includes oracle software/database. This is not the case in the testbed.
Cloud Files file storage Can be published to Akamai CDN		**Cloud Sites** Web Hosting platform			IaaS from hosting perspective.
Cloud storage CDN partnership with EdgeCast		**Managed hosting**			much like Rackspace Gartner: relatively small company, might not be able to keep up with the innovation speed set by bigger companies
					Purely research based. Limited functionality provided.
					Layer on top of public and private clouds to combine them. (or: for who wants to sell IaaS) This layer is not needed for the testbed.
					Virtualization platform that can be used to build IaaS. Not what the testbed needs.
					Virtualization platform that can be used to build IaaS. Not what the testbed needs.
					Layer on top of IaaS clouds. This layer is not needed for the testbed.
				e.g. SalesForce.com CRM	SaaS only
				e.g. GMail	SaaS only
				e.g. Office 365	SaaS only

B YAWL Interface Routing

For routing constructs, see Table 3.

For merge criteria, see Table 6.

For merge actions, see Table 7.

For filter value locations, see Table 8.

For filter value types, see Table 9.

B.1 Interface A Inbound

connect	new Rule(RoutingConstruct.None)
checkConnection	new Rule(RoutingConstruct.None)
upload	Handled in code
unload	new Rule(RoutingConstruct.AllDestinationsBySpecificationTenant, new HashMap<String, MergeRule>() {{ put("response", new MergeRule(MergeAction.Combine, MergeCriteria.Name) put("success", new MergeRule(MergeAction.Combine, MergeCriteria.Name) }})
getAccounts	new Rule(RoutingConstruct.AllDestinations, new HashMap<String, MergeRule>() {{ put("response", new MergeRule(MergeAction.Combine, MergeCriteria.Name) put("client", new MergeRule(MergeAction.Complement, MergeCriteria.Name_and_Attributes_and_Content) }})
getAccount	new Rule(RoutingConstruct.AllDestinationsByTenant, new HashMap<String, MergeRule>() {{ put("response", new MergeRule(MergeAction.Combine, MergeCriteria.Name) put("client", new MergeRule(MergeAction.Complement, MergeCriteria.Name_and_Attributes_and_Content) }})

getList	new Rule(
	RoutingConstruct.AllDestinationsByTenant,
	new HashMap<String, MergeRule>() {{
	put("response", new MergeRule(MergeAction.Combine, MergeCriteria.Name)
	put("specificationData", new MergeRule(MergeAction.Complement, MergeCriteria.Name_and_Attributes_and_Content)
	}},
	new HashMap<String, FilterRule>() {{
	put("specificationData", new FilterRule(FilterValueLocation.ChildNodeText, "id", FilterValueType.SpecificationGUID)
	}}
	)

getYAWLServices	new Rule(RoutingConstruct.AllDestinationsByTenant, new HashMap<String, MergeRule>() {{ put("response", new MergeRule(MergeAction.Combine, MergeCriteria.Name) put("yawlService", new MergeRule(MergeAction.Complement,

	MergeCriteria.Name_and_Attributes_and_Content) }})
createAccount, updateAccount, deleteAccount, newPassword, getPassword	new Rule(RoutingConstruct.AllDestinations)

B.2 Interface B Inbound

connect	new Rule(RoutingConstruct.None)
checkConnection	new Rule(RoutingConstruct.None)
checkIsAdmin	new Rule(RoutingConstruct.AllDestinations)
startOne	new Rule(RoutingConstruct.AllDestinations)
getCasesForSpecification	new Rule(RoutingConstruct.AllDestinationsBySpecificationTenant, new HashMap<String, MergeRule>() {{ put("response", new MergeRule(MergeAction.Combine, MergeCriteria.Name) }})

getLiveItems	new Rule(
	RoutingConstruct.AllDestinationsByTenant,
	new HashMap<String, MergeRule>() {{
	put("response", new MergeRule(MergeAction.Combine, MergeCriteria.Name)
	}},
	new HashMap<String, FilterRule>() {{
	put("workItem", new FilterRule(FilterValueLocation.ChildNodeText, "specidentifier", FilterValueType.SpecificationGUID)
	}}
	)

getAllRunningCases	new Rule(
	RoutingConstruct.AllDestinationsByTenant,
	new HashMap<String, MergeRule>() {{
	put("response", new MergeRule(MergeAction.Combine, MergeCriteria.Name)
	put("AllRunningCases", new MergeRule(MergeAction.Combine, MergeCriteria.Name)
	put("specificationID", new MergeRule(MergeAction.Combine, MergeCriteria.Name_and_Attributes)
	}},
	new HashMap<String, FilterRule>() {{
	put("specificationID", new FilterRule(FilterValueLocation.AttributeValue, "identifier", FilterValueType.SpecificationGUID)
	put("caseID", new FilterRule(FilterValueLocation.NodeText, null, FilterValueType.CaseNumberGlobal)
	}}
	)

getWorkItemsWithIdentifier	new Rule(
	RoutingConstruct.AllDestinationsByTenant,
	new HashMap<String, MergeRule>() {{
	put("response", new MergeRule(MergeAction.Combine, MergeCriteria.Name)
	}},
	new HashMap<String, FilterRule>() {{
	put("workItem", new FilterRule(FilterValueLocation.ChildNodeText, "specidentifier", FilterValueType.SpecificationGUID)
	}}
	)

getWorkItemsForService	new Rule(

	RoutingConstruct.AllDestinationsByTenant,
	new HashMap<String, MergeRule>() {{
	put("response", new MergeRule(MergeAction.Combine, MergeCriteria.Name)
	}},
	new HashMap<String, FilterRule>() {{
	put("workItem", new FilterRule(FilterValueLocation.ChildNodeText, "specidentifier", FilterValueType.SpecificationGUID)
	}}
	)

getCaseInstanceSummary	new Rule(
	RoutingConstruct.AllDestinationsByTenant,
	new HashMap<String, MergeRule>() {{
	put("response", new MergeRule(MergeAction.Combine, MergeCriteria.Name)
	put("caseInstances", new MergeRule(MergeAction.Combine, MergeCriteria.Name)
	}},
	new HashMap<String, FilterRule>() {{
	put("caseInstance", new FilterRule(FilterValueLocation.ChildNodeText, "caseid", FilterValueType.CaseNumberGlobal)
	}}
	)

getSpecificationPrototypesList	new Rule(
	RoutingConstruct.AllDestinationsByTenant,
	new HashMap<String, MergeRule>() {{
	put("response", new MergeRule(MergeAction.Combine, MergeCriteria.Name)
	put("specificationData", new MergeRule(MergeAction.Complement, MergeCriteria.Name_and_Attributes_and_Content)
	}},
	new HashMap<String, FilterRule>() {{
	put("specificationData", new FilterRule(FilterValueLocation.ChildNodeText, "id", FilterValueType.SpecificationGUID)
	}}
	)

url-ib	new Rule(RoutingConstruct.AllDestinationsByTenant)
cancelCase, getCaseState, getCaseData, getWorkItemInstanceSummary	new Rule(RoutingConstruct.DestinationByCase)
launchCase	new Rule(RoutingConstruct.NewAllocationBySpecification)
taskInformation, getMITaskAttributes, getResourcingSpecs, getSpecification, getSpecificationDataSchema	new Rule(RoutingConstruct.DestinationBySpecification)
checkout, checkin, rejectAnnouncedEnabledTask, getWorkItem, checkAddInstanceEligible, getChildren, getParameterInstanceSummary, createInstance, suspend, rollback, unsuspend, skip, url-workitem	new Rule(RoutingConstruct.DestinationByWorkItem)

B.3 Interface B Outbound

All action requests are directly forwarded to the resource service of the tenant. The available actions on interface B outbound are:

announceItemStatus
announceCaseCompleted
announceItemCancelled
announceCaseCancelled
announceTimerExpiry
announceEngineInitialised
announceCaseSuspending
announceCaseSuspended
announceCaseResumed
ParameterInfoRequest

B.4 Interface E

Interface E is not supported by the proof-of-concept implementation of YAWL in the Cloud. Interface E is normally used for process log interactions. For completeness, we do list all actions available on Interface E:

connect,
checkConnection,
getAllSpecifications,
getNetInstancesOfSpecification,
getCompleteCaseLogsForSpecification,
getSpecificationStatistics,
getCaseEvents,
getDataForEvent,
getDataTypeForDataItem,
getTaskInstancesForCase,
getTaskInstancesForTask,
getCaseEvent,
getAllCasesStartedByService,
getAllCasesCancelledByService,
getInstanceEvents,
getServiceName,
getCompleteCaseLog,
getEventsForTaskInstance,
getTaskInstancesForCaseTask,
getSpecificationXESLog

B.5 Interface X

Interface X is not supported by the proof-of-concept implementation of YAWL in the Cloud. Interface X is normally used for handling process level exceptions. For completeness, we do list all actions available on Interface X:

addInterfaceXListener,
removeInterfaceXListener,
updateWorkItemData,
updateCaseData,
completeWorkItem,
continueWorkItem,
unsuspendWorkItem,
restartWorkItem,
startWorkItem,
cancelWorkItem

C Evaluation Details

This Appendix contains more detailed information about the credit application process model, and the tool used for performance testing.

C.1 Process Model: Credit Application

Figure 66 shows the process model of the credit application process, which is frequently used in multiple Sections of this report.

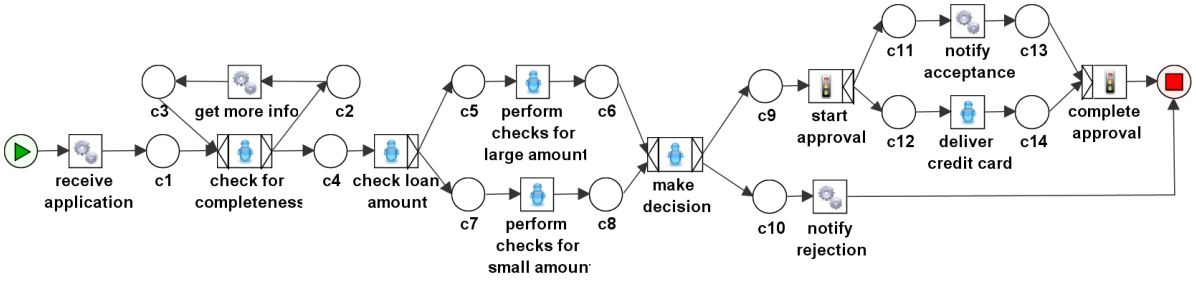


Figure 66: Credit Application Process Model

C.2 Test Tool for Performance Evaluation

Testing a web application for correctness, as well as testing how it behaves under load, is not a new problem specific to this project. Various comprehensive tools exist to both define and run such tests. We choose to use the open source tool Apache JMeter [33]. The part of YAWL in the cloud that faces the outside world is the set of REST interfaces it provides. Interaction with them is effectively a series of HTTP requests to the interfaces. It is this series of HTTP requests that we put in the test definitions.

JMeter has capabilities to test samples from multiple web protocols, out of which we use the HTTP request sampler. It also provides timers to put in between samplers, to simulate more time-realistic usage. Sequences of request samplers and timers can be put in so called test fragments, which can be re-used, and form the building blocks of larger tests. To make one test fragment cover multiple similar, but slightly different request sequences, one can use variables in the test fragment definitions. Global variables can accommodate variability over the entire test, such as against which interface endpoint URL (cloud or classic) it must be run. Other variables can accommodate local variability, such as reusing a fragment to launch a case, using the variables to define for which specification a case should be launched. If assertions are added to the HTTP samplers, these assertions will be verified against the received response. Global assertions will be checked against all responses, but more specific assertions can also be added to one individual HTTP sampler. If any of the assertions does not hold, or the request times out, the sample is considered as a failure. Using

post-processors, such as the XPath extractor, data in the response of a request can be used as a value for a variable, e.g. to pass along the case number to cancel a case that was started during that test.

Test fragments are combined into a scenario in a so called thread group. One thread corresponds to the interface interaction of one user. The thread group configuration defines how many simultaneous users should be simulated, and how many times the scenario should be repeated per user. Furthermore, a duration can be set during which the number of users is linearly increased from zero to the defined number of simultaneous users, the so called ramp-up time. A test definition is composed of one or more thread groups, which start execution at the same time. JMeter can remotely run its tests from multiple computers at the same time, in case more load is needed than one computer can generate and log.

The result of running a test depends on the type of listeners that have been added to the test definition. Information about individual samples, as well as statistics over all of the samples, is logged and can be presented in a tabular or graphical representation. For additional visualisation listeners we use the JMeter plugin package called *JMeter Plugins at Google Code* [34] which, for example, provides a visualisation to show response times in a percentiles graph.

Figure 67 shows the user interface of Apache JMeter. The left pane contains the test definition tree. The right pane contains configuration options or test results. The depicted test definition tree contains samplers, a thread group, global and local variables, global and local assertions, timers and a graphical listener.

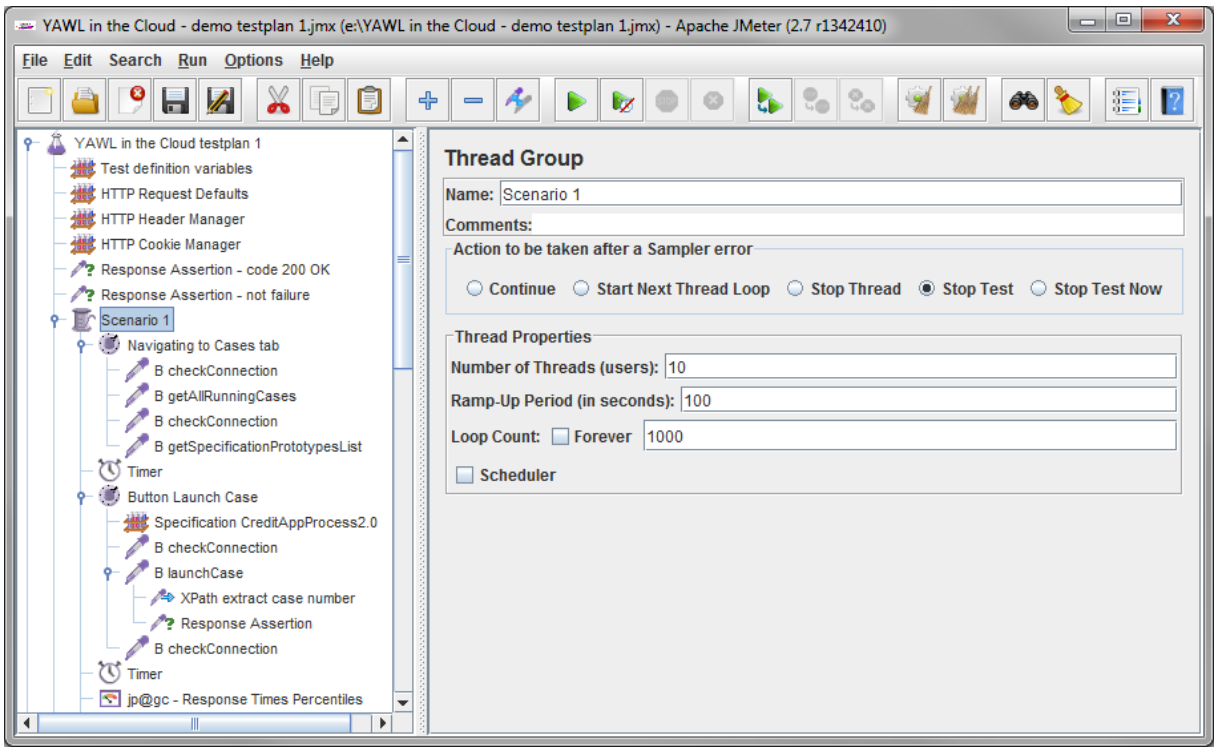


Figure 67: The user interface of Apache JMeter

Example: Functional and Performance Test

Figure 68 shows a section of a test definition for a functional and performance test. It is the equivalent of browsing to the cases tab of the resource service, and then clicking launch case. Figure 70 shows the result of a test run. The green icons indicate that all requests passed the test, i.e. the requests did not result in errors, failures, or failed assertions. Since timing does play a role for performance tests, in this test section you see timers as part of the definition. Figure 69 shows one of the possible graphical representations of test results: the percentiles view. The percentiles view gives insight in SLA related questions, such as “can I guarantee that 95% of all requests is handled within 5 seconds?”.

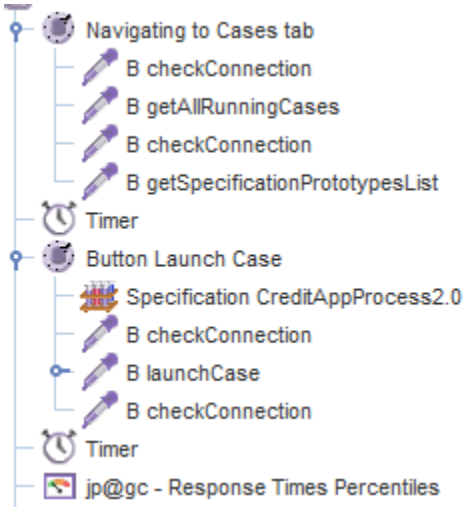


Figure 68: A section of a test definition

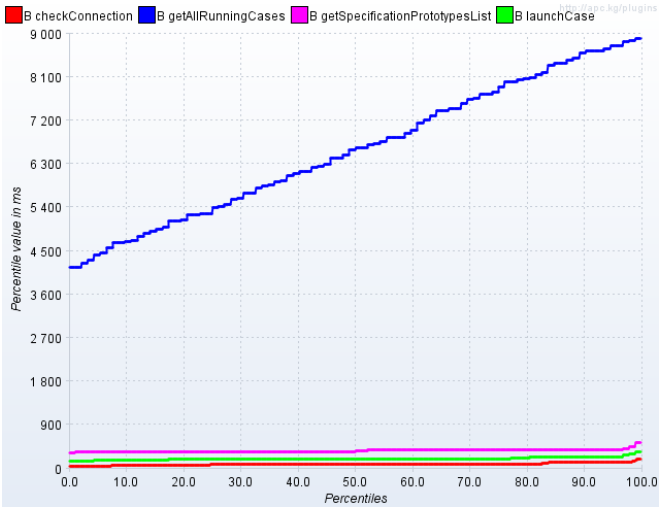


Figure 69: Test result: performance view

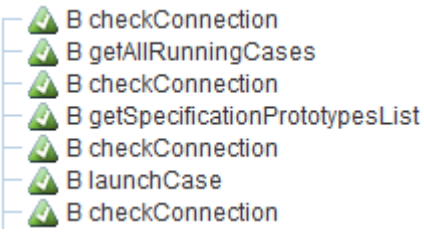


Figure 70: Test result: functional view

C.3 Test Definition

Figure 71 shows the performance test definition used to compare classic YAWL and YAWL in the Cloud. Note that the Poisson Random Timer will be applied to every single HTTP sampler.

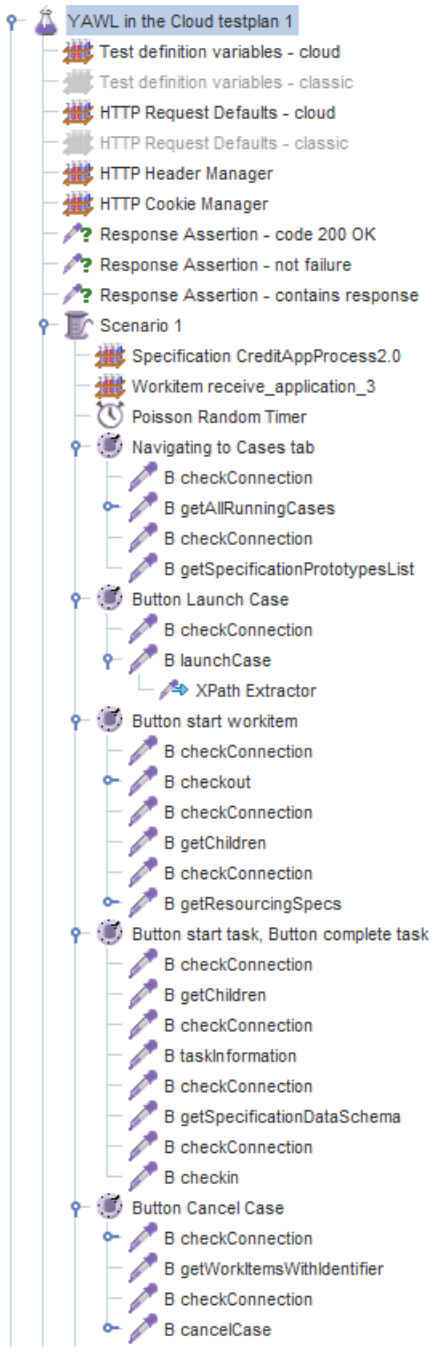


Figure 71: Performance comparison test definition

D Installation Manual

This installation manual is intended for those who want to install YAWL in the cloud themselves. We assume the reader of this installation manual has experience in setting up and configuring servers, and only needs to be provided with the correct configuration settings, rather than a detailed step by step description with screenshots.

YAWL in the cloud was tested on Microsoft Windows 7 and Microsoft Windows Server 2008 R2 enterprise in January 2013. Both had all Windows updates installed.

Java SE

YAWL in the cloud is written in Java. Install the Java SE 7u7 JRE runtime environment, which is available for download online.

NetBeans IDE 7.1.1

NetBeans was used to develop and build YAWL in the cloud. NetBeans IDE 7.1.1 is online available for download. NetBeans is not needed to run YAWL in the cloud; the compiled binaries suffice for that.

Tools

Suggested tools are:

- Google Chrome web browser, including developers tools.
- Notepad++ to edit any text-based file type, such as the configuration files in this installation.
- BareTail log viewer free, to show the log of YAWL in the cloud. BareTail automatically follows the end of the log, and is capable of colour-highlighting based on regular expression matches. E.g. make lines containing ‘error’, ‘warning’, or ‘exeption’ red, and lines containing ‘success’ green.
- To extract any WAR archive to disk by hand, use 7zip.

MySQL Server

As the database server we use MySQL Server 5.5.20.0 which can be downloaded for free. Follow the following steps during the installation:

- Choose install MySQL products
- Accept the license terms
- Choose the 'developer default' setup, which includes MySQL Server, MySQL Workbench and the MySQL ODBC connector.
    - This installation requires the Microsoft Visual C++ 2010 runtime, which will automatically be installed if missing on the target system.
- Configuration options: choose 'server machine' if you're installing MySQL on a server, which is not a dedicated database server.
- Enable TCP/IP port 3306 for remote connections, and tick the checkbox 'create firewall rule'.
- Choose to create a Windows Service to run MySQL Server
- Set the root password to 'Yitc!12demo'

Apache Tomcat

As the webserver we use Apache Tomcat 7.0.30, which is available as a 32-bit/64-bit Windows Service installer for free. Use the following configuration options:

- During the installation, choose to include (only) the following components:
    - Tomcat (core + service start-up + native)
    - Start menu items
    - Documentation
    - Manager application
- Set the username 'admin' and the password 'YAWL'
- In the Tomcat configuration add the following Java option:
    - --XXMaxPermSize256m
- Copy, from the MySQL\connector j\ directory the .jar file and put in in the Tomcat/Lib directory.

Firewall

Open up all ports that should be accessible from other machines:

- Open TCP port 3306 for connections to MySQL
- For Tomcat open TCP port range 8000-9000, or only (set of) port(s) hosted by Tomcat.

Engine Configuration

After installing an engine, do the following:

- Set the correct database name and password in *hibernate.properties*
- Set any logging preferences in *log4j.properties*
- Set the DefaultWorklist URL and the display name of the engine (for the Tomcat manager application) in *web.xml*

Resource Service Configuration

After installing a resource service, do the following:

- Set the correct database name and password in *hibernate.properties*
- Set any logging preferences in *log4j.properties*
- Set the InterfaceB_BackEnd URL and the display name of the engine (for the Tomcat manager application) in *web.xml*
- Set the name and colour to show in the header of the web pages in *pfHeader.jspf*

YAWL in the Cloud Configuration

To install YAWL in the cloud, put the WAR file the 'webapps' directory of Tomcat. It will automatically unpack and install. Then do the following:

- Set the correct database name and password in *hibernate.cfg.xml*
- Set the router name in *web.xml*
- Import the empty database structure into MySQL. This can be done using the export and import functions of MySQL Workbench CE.

E Index

E.1 List of Figures

E.2 List of Tables