YAWL in the Cloud

D.M.M. Schunselaar*, T.F. van der Avoort, H.M.W. Verbeek*, and W.M.P. van der Aalst*

Eindhoven University of Technology,
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
{d.m.m.schunselaar, h.m.w.verbeek, w.m.p.v.d.aalst}@tue.nl

Abstract. In the context of the CoSeLoG project (which involves 10 Dutch municipalities), we realised a proof-of-concept implementation based on YAWL. The municipalities want to share a common IT infrastructure and learn from one another, but also allow for local differences. Therefore, we extended YAWL to run in a cloud-based environment leveraging on existing configuration possibilities. To support "YAWL in the Cloud" we developed load-balancing capabilities that allow for the distribution of work over multiple YAWL engines. Moreover, we extended YAWL with multi-tenancy capabilities: one municipality may effectively use multiple engines without knowing it and one engine may safely run the processes of multiple municipalities.

1 Introduction

Within the Netherlands, more and more municipalities seek cooperation to cut costs. One of the directions to cut costs is to share infrastructures supporting the processes of municipalities. Every municipality has a server/ business process management system (BPM system)/ case handling system/ etc. to support them. However, these systems are not used to their fullest capacity, and also capacity might change, e.g., decrease, during the year. Municipalities can cut costs by sharing infrastructure and have an adaptive system to handle the peaks. Within the CoSeLoG project, we are cooperating with 10 municipalities who want to cooperate with each other and learn from each other[1] [1]. One of the elements of the project is a shared infrastructure where the municipalities share IT-solutions and processes.

In recent years, we created many (configurable) YAWL models [2–4] for the processes of the municipalities. Therefore, we want to use YAWL to create a proof-of-concept implementation, showing that municipalities can share a common infrastructure and still allow for the necessary "colour locale". Unfortunately, YAWL has been designed to be used on a single machine. If we are using a single machine, this requires the owner of this machine to share (part of) her infrastructure which might not always be desirable. Furthermore, we are dealing

* This research has been carried out as part of the Configurable Services for Local Governments (CoSeLoG) project (http://www.win.tue.nl/coselog/).

[1] http://www.win.tue.nl/coselog/

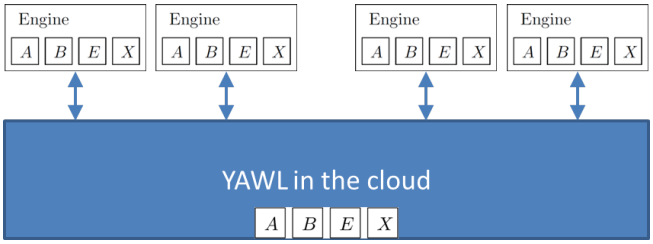


Fig. 1: The positioning of YAWL in the cloud.

with different organisations which do not want to expose some or all of their information to the other parties involved. Finally, by combining different organisations onto a single machine, this machine might become overloaded by a shared peak caused by the organisations.

In order to overcome the mentioned problems, we propose *YAWL in the cloud*. The cloud is tailored towards scalable resources to increase the computing power when necessary and to decrease the computing power when possible. Since the cloud can be maintained by a third party, there is no need to employ a person to maintain the systems. However, in order to make YAWL run in the cloud, we have to extend the standard YAWL architecture to deal with problems introduced by having multiple YAWL engines running simultaneous (e.g., non-unique case numbers amongst different engines). Therefore, we first introduce the general framework built around YAWL. Afterwards, we present the implementation and show some innovative features of YAWL in the cloud. Finally, we discuss the limitations and future work.

2 Architecture

An early design decision was to not change the YAWL system itself. By not changing YAWL itself, we are not bound to a specific (modified) version of YAWL. Moreover, to avert having to learn a new system, the end-user should not notice that she is working in the cloud. Finally, there has to be an additional component to control and observe the current state of YAWL in the cloud. This resulted in the high level architecture as depicted in Fig. 1, note that A, B, E, X are the interfaces of a YAWL engine. For each of the constraints, we list part of the effect it had on the architecture. Please note (as Fig. 1 shows), we are running multiple YAWL engines on multiple machines.

Instead of running a single YAWL server on a single machine, we assume that we are running multiple YAWL servers on multiple machines. The combination of these servers/machines is YAWL in the cloud.

No change to YAWL: By having multiple YAWL engines, it is no longer apparent to which engine to connect. Therefore, we introduce a *router* component. This router component routes the requests to the correct YAWL engine.

By not changing YAWL, each YAWL engine is unaware of the other YAWL engines. Therefore, it can no longer be guaranteed that every case identifier is unique. In order to overcome this, we introduce unique global cloud identifiers and maintain a mapping between global (cloud assigned) identifiers and local (engine specific) identifiers. This mapping is stored in a central database, and the transformations of the global/local identifiers to local/global identifiers (and vice versa) is done by the router.

We can now receive a request and route it to the correct engine(s). However, when multiple engines have to be consulted, we also obtain multiple responses. Therefore, we need to merge these responses in a single response before sending it back to the requestor. The merging of the responses is handled by the router.

Finally, it might be the case that not all the information in a response is intended for the requestor. This can be the case when an engine is running multiple cases for different tenants. Since the YAWL engine does not know the notion of multiple tenants, it sends information about all the tenants back as a response. Therefore, we also include filtering functionality in the router.

Invisible cloud infrastructure: YAWL is decomposed into components, e.g., engine, resource service, process designer. This decomposition yields that we only have to take care that the resource service is directed to *YAWL in the cloud* instead of an actual engine. This change is only a configuration of the resource service. By using the resource service as is, we do not change the front-end view of the end-user.

Management component: For the management component, we require a number of views on the cloud. One of these views is a functional view denoting per tenant which specifications and cases are loaded. Apart from a functional view, we also want a software view denoting the hierarchy of servers, engines, tenants, specifications, and cases. Furthermore, this management view should be usable to create new tenants, and bring new engines into the cloud.

Complete architecture: Taking all the constraints into account, we obtain the more detailed architecture depicted in Fig. 2. At the back, the communication with the different engines is situated. At the front we have a single point of entry for the resource services. Inside of the architecture, we have the routers for routing, translating, merging and filtering of requests and responses. These routers use the central database (DB) for the lookups of identifiers. There is a management component at the centre communicating with the engines and the central database. Finally, we have included a load balancer for both the incoming and outgoing requests. This load balancer is cloud-based and offers the option to enable/disable routers if necessarily. Furthermore, if a router cannot handle the requests, then the load balancer can initiate an extra router.

Within cloud computing there are different deployment models: public, private, community, and hybrid. In our architecture, we do not pose any restrictions on the used deployment model, i.e., our architecture is applicable to all of these

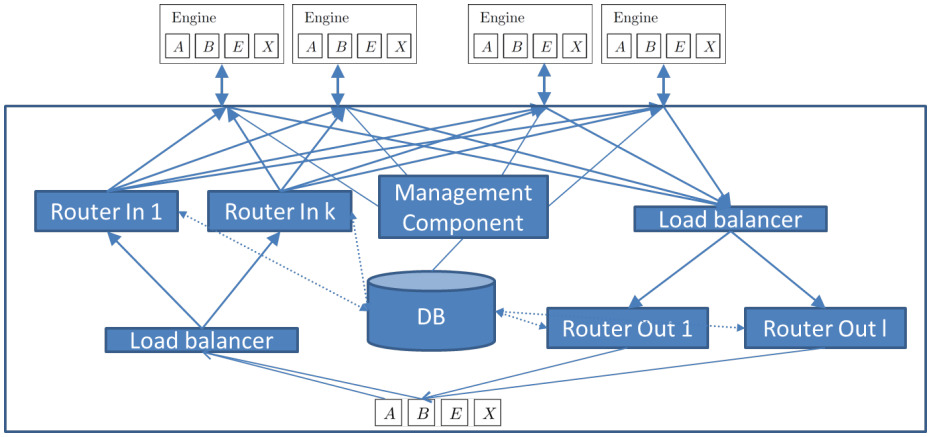


Fig. 2: The architecture for YAWL in the cloud

deployment models. Apart from different deployment models, there exist different service models, i.e., abstraction levels from the underlying hardware. These service models come in 4 different flavours (in ascending abstraction level): Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS), and Business Process as a Service (BPaaS). Our architecture has been designed to be used in a PaaS service model. For a more extensive discussion on service models and deployment models, see [5].

Having presented our architecture, we now present the YAWL in the cloud implementation. For each of the newly introduced components, we show some of the implementation details.

3 Implementation

Using the architecture shown in Fig. 2, we implemented the various components connecting to standard YAWL. Due to space restriction, not all implementation details can be covered. See [5], for a complete overview of the implementation of YAWL in the cloud.

The router: As mentioned, we had to introduce a routing component for routing, translating, merging and filtering of requests and responses before sending them to the requestor. In Fig. 3, the communication between the different components is depicted. First a request is sent to the router, then the router consults the database for (amongst others) translating global identifiers to engine specific identifiers. Afterwards, the router contacts the engines of interest for this request. After the engines have sent their responses, these responses are merged, filtered, and the database is consulted for translation of engine specific identifiers to global identifiers. Finally, the created response is forwarded to the client.

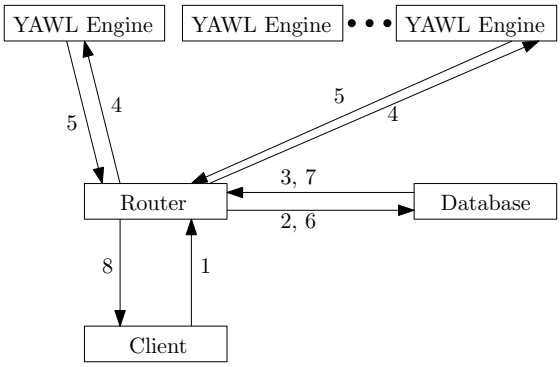


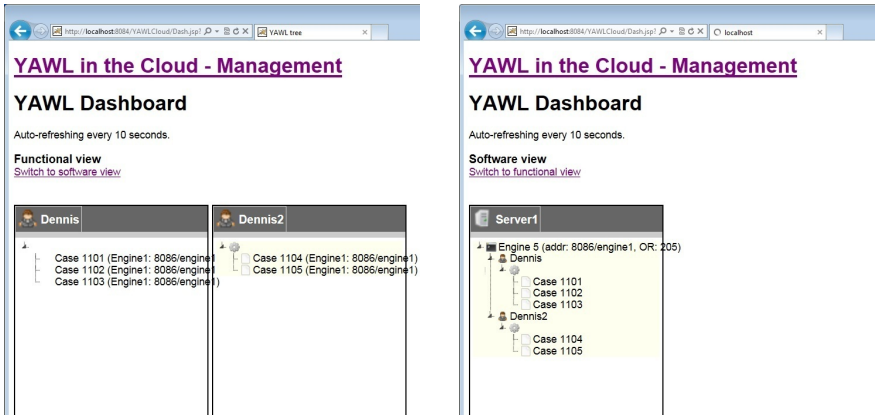
Fig. 3: The communication between the different components when a request is made.

Based on the type of request, different types of merges had to be introduced and rules for forwarding the request to specific engines. For instance, for the action *getAllRunningCases*, which gives all the running cases for a particular tenant, we have to merge the results per specification (i.e., multiple engines can have the same specification and we do not want to duplicate the specification to the user). Furthermore, this request has to be forwarded to all engines running specifications for this tenant. Finally, the cases for specifications not owned by the tenant have to be filtered out.

If we consider the action *getCasesForSpecification*, which gives all the running cases for a particular specification, we only need to forward this request to the engines running this specification. The engines return all the cases for that specification, the cases in the responses have to be merged into a single response. However, the filtering step is not required as this specification belongs exclusively to a specific tenant. Other tenants may use the same specification, but the identifier of this specification is different for different tenants.

The database: In the central database, we store the different local YAWL identifiers and the global cloud identifiers. We have local YAWL identifiers for specifications, cases, and work-items. Apart from storing the identifiers, we also maintain the different tenants and which specification a tenant has at her disposal. Finally, the database stores the settings for YAWL in the cloud, e.g., which constraints are present within YAWL in the cloud. A constraint can be that a tenant can have at most 5 cases per specification.

The management component: The management component is a view on the database and on the engines. In Fig. 4a and Fig. 4b two different views are presented on the systems. The first shows a separation of specifications and cases per tenant. The second shows a hierarchical view of servers, engines, tenants, specifications, and cases.



(a) Functional view, showing which cases and specifications are loaded per tenant.

(b) Software view, showing the servers, engines, tenants, specifications, and cases, and the hierarchy of them.

Fig. 4: Two different views on the cloud in the management component; per tenant, and hierarchical per server.

Apart from providing a view on the engines and database, the management component also allows to add/remove engines, and to add/remove tenants. Note that this functionality is currently semi-automated as the configuration of the different components still requires human involvement.

The front-end: Figure 4 shows the administrator view on two tenants. The views the different tenants have are depicted in Fig. 5 and Fig. 6. We have changed the name and colour of the YAWL admin view in order to stress that both tenants have indeed different views. Figures 5 and 6 also show the filtering step of the routers as a tenant only sees her cases, and not the cases of the other tenants.

4 Limitations and Future Work

We have presented our architecture for bringing YAWL in the cloud using the following design decisions: no changes to YAWL, and the end-user should be unaware to the cloud back-end. Furthermore, there had to be a management component to control the cloud. Along with our architecture, we also have shown some of our implementation details.

The implementation of bringing YAWL in the cloud is not complete. For every action on every interface, specific transformations had to be made. Therefore, we have focussed on the main interfaces: interface A inbound, and interface B in- and outbound. In future implementations, we want to extend the support to also include interface E and interface X.

YAWL in the cloud now mainly works in a semi-automated fashion, i.e., some actions require human involvement. In the ideal implementation, we want to have

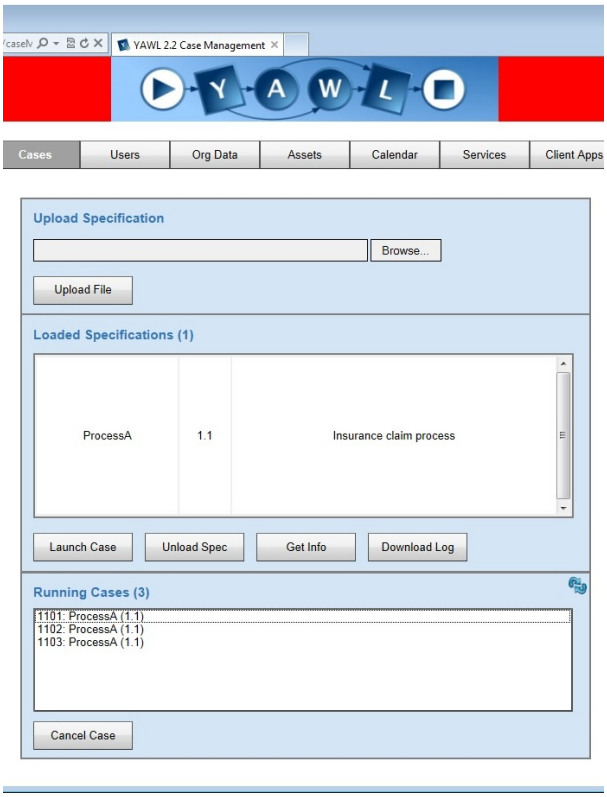


Fig. 5: The management view of tenant 4.

full automated support for all the features. Using this full automated support, it also allows for automated increases and decreases in computing power. In our current implementation, we have added functionality to optimise the computing power. Unfortunately, the added overhead of YAWL in the cloud is not compensated by the scalability. In the future, we want to profile our implementation and solve the bottlenecks currently in the implementation.

With bringing YAWL in the cloud, we mainly focussed on the engine. The next step would be to also bring the resource service in the cloud. By bringing the resource service in the cloud, we allow for an extra layer of flexibility and a clearer separation between organisations. Thanks to the decomposition of YAWL into different components, one can employ a similar approach to bringing the resource service in the cloud as we have done for bringing the engine in the cloud.

Finally, this research was started with collaboration and knowledge sharing amongst municipalities in mind. In the next step, we plan to use configurable YAWL as the base for the different process models in use by the municipalities. With configurable YAWL, the best practises are captured in a single model,

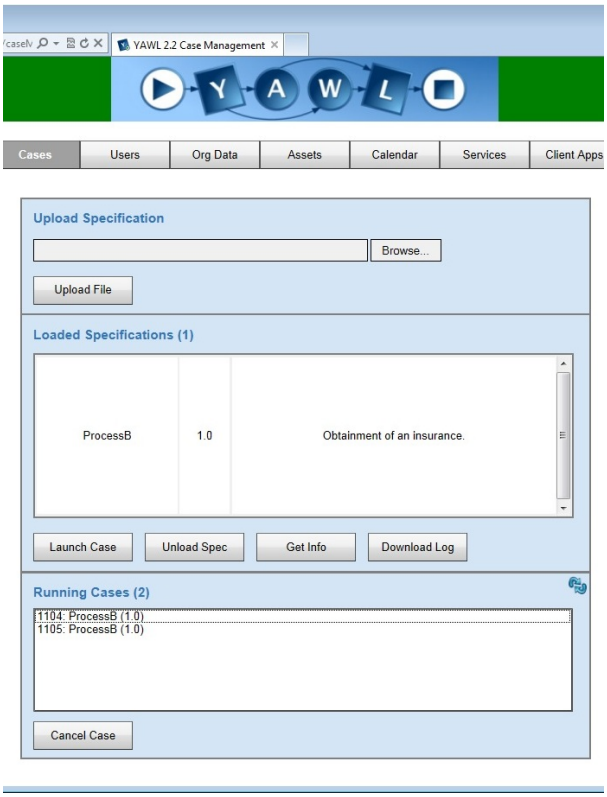


Fig. 6: The management view of tenant 5.

allowing the municipalities to cherry-pick the practises best fit for their organisation.

References

1. van der Aalst, W.M.P.: Business process configuration in the cloud: How to support and analyze multi-tenant processes? In Zavattaro, G., Schreier, U., Pautasso, C., eds.: ECOWS, IEEE (2011) 3–10
2. ter Hofstede, A.H.M., van der Aalst, W.M.P., Adams, M., Russell, N., eds.: Modern Business Process Automation: YAWL and its Support Environment. Springer (2010)
3. La Rosa, M.: Managing variability in process-aware information systems. PhD thesis, Queensland University of Technology (2009)
4. Gottschalk, F.: Configurable Process Models. PhD thesis, Eindhoven University of Technology, The Netherlands (December 2009)
5. Avoort, T.F.v.d.: BPM in the Cloud. Master's thesis, Eindhoven University of Technology, The Netherlands (2013)