Design and Performance Analysis of Load Balancing Strategies for Cloud-Based Business Process Management Systems

Michael Adams[1](✉), Chun Ouyang[1], Arthur H. M. ter Hofstede[1], and Yang Yu[2]

[1] Queensland University of Technology, Brisbane, Australia
{mj.adams,c.ouyang,a.terhofstede}@qut.edu.au
[2] Sun Yat-sen University, Guangzhou, China
yuy@mail.sysu.edu.cn

Abstract. Business Process Management Systems (BPMS) provide automated support for the execution of business processes in modern organisations. With the advent of cloud computing, the deployment of BPMS is shifting from traditional on-premise models to the Software-as-a-Service (SaaS) paradigm with the aim of delivering *Business Process Automation as a Service* on the cloud. To cope with the impact of numerous simultaneous requests from multiple tenants, a typical SaaS approach will launch multiple instances of its core applications and distribute workload to these application instances via *load balancing* strategies that operate under the assumption that tenant requests are stateless. However, since business process executions are *stateful* and often long-running, strategies that assume statelessness are inadequate for ensuring a uniform distribution of system load. In this paper, we propose several new load balancing strategies that support the deployment of BPMS in the cloud by taking into account (a) the workload imposed by the execution of stateful process instances from multiple tenants and (b) the capacity and availability of BPMS workflow engines at runtime. We have developed a prototypical implementation built upon an open-source BPMS and used it to evaluate the performance of the proposed load balancing strategies within the context of diverse load scenarios with models of varying complexity.

Keywords: Business Process Management System
Software-as-a-Service · Load balancing · Workflow engine · Scalability

1 Introduction

As a leading exemplar of process-aware information systems, Business Process Management Systems (BPMS) are dedicated to providing automated support for the execution of business processes in modern organisations. In recent times, the advantages offered by cloud computing have triggered an increased demand for the deployment of BPMSs to shift from traditional on-premise models to the Software-as-a-Service (SaaS) paradigm.

Traditionally, BPMS are installed on-site to serve a single organisation. In a multi-tenant cloud environment, scaling up a discrete BPMS to serve simultaneous demands from multiple organisations is challenging given the limited capacity of a single deployment. Thus it has become necessary to reshape the architectures of BPMS so that better support can be offered to clients seeking the benefits offered by cloud-based implementations. A review of the existing efforts towards a generic architecture for a scalable BPMS in the cloud reveals that this aim has not yet been achieved.

In cloud computing generally, to deal with the increasing load from multiple clients, a SaaS approach often deploys multiple instances of its core applications and distributes workload (of tenant requests) to these application instances via a load balancing strategy [4]. This approach assumes that the tenant requests are stateless, but a business process execution is *stateful* and often long-running, and so existing load balancing strategies designed for handling stateless requests are not suitable for handling the execution of process instances. Rather, load balancing strategies need to take into account the work currently being performed statefully on each application instance.

In this paper, we propose a design for a multi-tiered approach to developing load balancing strategies that can better support the stateful requests and responses associated with the execution of process instances for a multi-tenant cloud environment, with a focus on supporting scalability. We define a dual-faceted metric to measure current process engine load, the first component based on the complexity of a process definition *from the perspective of the computational execution engine*, and the second on measures taken in real time from the operational environment. To validate our approach, a prototype has been developed that deploys multiple instances of a traditional open-source BPMS in a cloud environment, using a stateful load balancing middleware component that implements a number of load balancing strategies at various levels of complexity and suitability. A realistic process load is generated via a simulation tool to demonstrate and validate the applicability of our approach, and demonstrates improved capabilities for supporting large volumes of work in a multi-tenanted cloud environment.

The rest of the paper is organised as follows. Section 2 provides the background and sets out the research problem to address. Section 3 proposes the design of a suite of load balancing strategies. Section 4 discusses a prototypical implementation of a scalable BPMS and the load balancing component, and presents a validation of the approach. Section 5 reviews related work. Finally, Sect. 6 concludes the paper and provides an outline of future work.

2 Background: A Cloud-Based BPMS Architecture

Business Process Management Systems (BPMS) are systems that aim to support business process execution by coordinating the right work to the right person (or application) at the right time. A generic architecture of *traditional* BPMS, as depicted in Fig. 1(a), follows the *workflow reference model* [10] proposed by the

Workflow Management Coalition. The core component is the *workflow engine*, which is responsible for creating, running, and completing execution instances of business processes. A process instance is also called a *case*. Each case consists of a sequence of *work items* and these are managed by the *worklist handler* through which the end users (e.g. staff in an organisation) interact with their own list of work items, e.g. to check out a work item to start the work, or to check in a work item when the work is completed. Other operational matters in a BPMS such as resource administration (e.g. managing the access control of individual end users) and process monitoring (e.g. tracking the progress of each running case) are taken care of by the *administration and monitoring tools*.

In addition, there are also two important data repositories. One is the *process model repository*. Before a process can be executed in the workflow engine, it needs to be defined in the form of a process model using a proper modelling language. A process modelling tool is used to create process models. It is also part of a BPMS but is not further considered in this paper, since we are interested in business process execution (rather than process modelling). Hence, we assume that a process model has always already been deployed to the workflow engine from the process model depository. Next, the workflow engine records the (step-by-step) execution of a process and exports the relevant data in the form of *execution logs*. Such data are valuable for process analysis and monitoring.

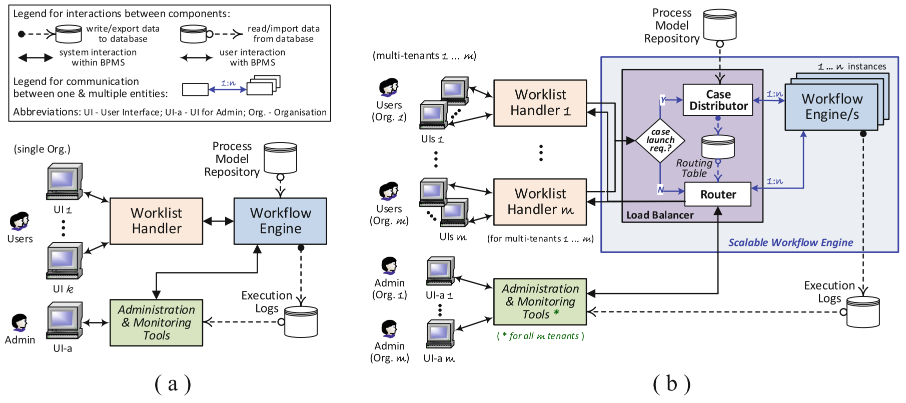


Fig. 1. BPMS architectures: (a) a traditional BPMS deployed on-premise for a single organisation *vs.* (b) a cloud-based BPMS for multiple tenant organisations

However, a traditional BPMS is usually deployed on-premise to serve a single organisation, and as such it is not designed to address simultaneous demands from multiple organisations. When deploying a BPMS in a cloud environment to provide a process automation service to multiple clients, it is necessary to redesign its underlying architecture so that the resulting system is capable of scaling up to cope with large volumes of work from these organisations. Figure 1(b) depicts a generic architecture for cloud-based BPMS. It is a redesign of the traditional BPMS architecture shown in Fig. 1(a) following the principle of a well-established SaaS maturity model [4]. A detailed proposal of the design of this architecture is presented in our previous work [14].

As Fig. 1(b) depicts, the key component to handle simultaneous requests from multiple organisations is the so-called *scalable workflow engine*, which consists of *multiple instances* of workflow engines coordinated via a *load balancer*. The load balancer further comprises a *case distributor*, a *router* and a *routing table*. The case distributor is responsible for handling a request for launching a case. It allocates the case to one of the workflow engines based on a *load balancing strategy* (e.g. to identify the least occupied engine) (see Sect. 3.4). Once a case is distributed to a workflow engine, the case will be executed in a *stateful* manner, meaning that all the work items belonging to the case will be handled by the *same* workflow engine until the case is completed. After a case is launched, the router takes over the responsibility of communication to direct work items between worklist handlers and workflow engines. The routing table records the necessary information in coordinating each work item between the right workflow engine and the right worklist handler.

Figure 2 shows a UML sequence diagram capturing the sequence of interactions between a worklist handler, a case distributor and the workflow engines for launching a case. Upon receiving a request to launch a case, the case distributor will (1) ask each workflow engine for its current busyness status in a certain time period, (2) invoke a load balancing algorithm to decide on an appropriate engine to allocate the case, (3) communicate with that engine to launch the case, and (4) record in the routing table the information necessary for correctly routing the work items belonging to the case.

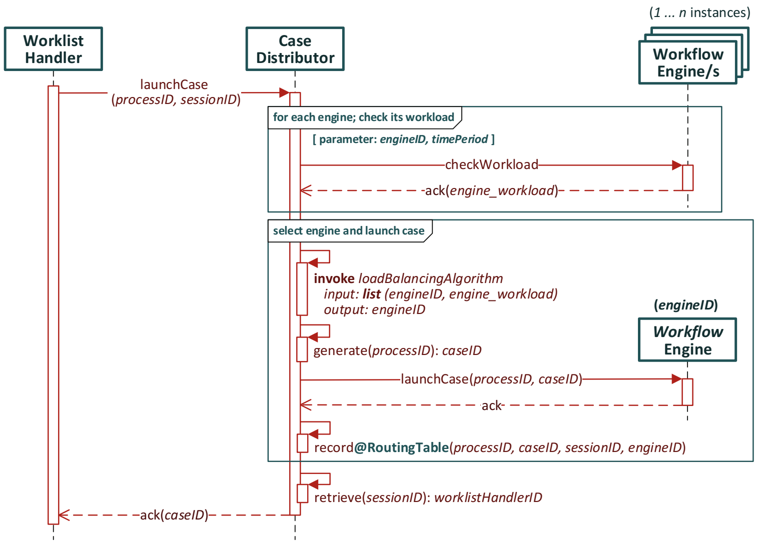

Worklist Handler
Case Distributor
(1 ... n instances)
Workflow Engine/s
launchCase (processID, sessionID)
for each engine; check its workload
[parameter: engineID, timePeriod]
checkWorkload
ack(engine_workload)
select engine and launch case
invoke loadBalancingAlgorithm
input: list (engineID, engine_workload)
output: engineID
(engineID)
Workflow Engine
generate(processID): caseID
launchCase(processID, caseID)
ack
record@RoutingTable(processID, caseID, sessionID, engineID)
retrieve(sessionID): worklistHandlerID
ack(caseID)

Fig. 2. A UML sequence diagram capturing the interactions of launching a case

3 Design of Load Balancing Strategies

This section focusses on the design of several load balancing strategies that may be applied to handle stateful management of process instances and to achieve an even distribution of the load across an array of process engines in a cloud-based BPMS. It covers a definition of the load balancing problem to address within the scope of this study, an initial proposal for measuring load complexity and resource occupation, and a specification of different levels of capabilities to be achieved for a complete load balancing strategy.

3.1 Problem Definition

A key objective of our load balancing strategies is to ensure a uniform distribution of system load among multiple instances of workflow engines deployed in a cloud-based BPMS. We assume that the number of workflow engines in the system is fixed and known *a priori*, and also that all these engines are of similar (computational) efficiency and capacity. Below, we define the problem of load balancing among multiple workflow engines.

Definition 1 (Balanced workload distribution among engines). *For a cloud-based BPMS, $\mathcal{G}$ denotes a set of workflow engines deployed in the system, $\mathcal{C}$ a set of cases executed in the system, $\mathcal{T}$ a set of timestamps, and $\mathcal{TD}$ a set of time durations. Let $\mathcal{W} \subseteq \mathbb{R}^+ \cup \{0\}$ be a set of values representing the amount of workload.*

- *$\forall g \in \mathcal{G}$, $\omega_g : \mathcal{T} \rightarrow \mathcal{W}$ specifies the amount of workload carried by a given workflow engine g at any point of time.*
- *$\gamma : \mathcal{C} \rightarrow \mathcal{G}$ specifies the workflow engine to which a case is allocated, i.e. each case $c \in \mathcal{C}$ is allocated to only one workflow engine to support the stateful execution of the case.*

Given a point of time $t \in \mathcal{T}$ and a time duration $d \in \mathcal{TD}$, $[t, t+d]$ defines a time interval, and $\int_t^{t+d} \omega_g(t)$ represents the accumulated amount of workload of a workflow engine g during the time interval $[t, t+d]$. Hence, let $\mathcal{G} = \{g_1, ..., g_n\}$, given any time period $[t, t+d]$, $\int_t^{t+d} \omega_{g_1}(t) \approx ... \approx \int_t^{t+d} \omega_{g_n}(t)$ demonstrates a balanced distribution of work load among the group of engines (specified by $\mathcal{G}$).

3.2 Complexity of Process Instances Measure

One important fact to consider is that process instance definitions may have varying computational complexity, thus imposing workloads of varying capacities on process engines. In light of this, we introduce a new case attribute, namely the *Case Complexity Indicator* (*CCI*), as a quantifiable measure to capture the complexity of a process specification.

Interestingly, while there have been many complexity metrics proposed for process models (see [16] for a survey), every one of them is aimed at measuring complexity from the perspective of a human reader, i.e. as a tool for communication and/or manual analysis. There have been no studies or definitions of complexity metrics of process model instances at runtime from the perspective of the execution engine, to our knowledge. Nevertheless, for this work we have made an initial attempt to consider what may be useful ways to measure how complex a process engine may find a particular case during its execution, taking into account each of the three main process perspectives: control-flow, data and resourcing.

As a starting point, we use the *Extended Cardoso Metric (ECaM)* [11] to inform the value of the CCI. ECaM can be used to measure the structural complexity of a process model and it is built on the classic Cardoso metric [3] which is applicable to all processes (or process languages) that support XOR-, OR-, and AND-splits. ECaM generalises and improves the classic Cardoso metrics so that they can be applied to all Petri net aligned languages.

However, the ECaM on its own is an insufficient measure for execution complexity. We added consideration for the number of tasks defined within a model, since that frequency has a direct effect on the frequent calculation of process state space during execution. These two measures are a first attempt to define control-flow complexity at runtime.

For the data perspective, we consider the number of user-defined data types, and the number of data assignments to input and output variables of tasks, to be indicators of runtime complexity. These attributes are considered to be useful indicators of data load, because each value assignment invokes a transposition that often requires a (re)construction of an explicit data structure to store instance values. Finally, for the resource perspective, the number of roles assigned to tasks is considered, since role groups have to be unpacked at runtime to the contained set of its individual participants, so that work item allocation to individuals can be achieved.

Using these five input measures, we define a runtime complexity metric below.

Definition 2 (Runtime complexity metric). *A runtime complexity metric RCM of a process model is a tuple* $(C, T, U, V, R, W_C, W_T, W_U, W_V, W_R)$ *such that:*

- C *is the extended Cardoso metric*
- T *is the number of tasks*
- U *is the number of user-defined types*
- V *is the number of input and output variables*
- R *is the number of resource roles assigned to tasks*
- W_C *is a weighting factor for the extended Cardoso metric*
- W_T *is a weighting factor for the number of tasks metric*
- W_U *is a weighting factor for the number of user-defined data types metric*
- W_V *is a weighting factor for the number of input and output variables metric*
- W_R *is a weighting factor for the number of roles metric*

A weighting applied to each measure provides the capacity for an overall, balanced complexity metric calculation that can be effectively used to contribute to the measure of load placed on a process engine by a process instance definition at runtime.

3.3 Engine Busyness Measure

The second component of our measure of engine load is derived for four key runtime data points related to the operational environment. We define this component below.

Definition 3 (Operational busyness). *A process engine's operational busyness measure is a tuple* $(R, P, I, T, W_R, W_P, W_I, W_T, L_R, L_P, L_I, L_T)$ *such that:*

- R *the number of requests processed per second*
- P *the average processing time per processed request (in milliseconds)*
- I *the number of work item starts and completions per second*
- T *the number of worker threads currently executing in the engine's container*
- W_R *is a weighting factor for the number of requests processed per second metric*
- W_P *is a weighting factor for the average processing time per processed request metric*
- W_I *is a weighting factor for the number of work item starts and completions per second metric*
- W_T *is a weighting factor for the number of worker threads metric*
- L_R *is an upper limit on the number of requests processed per second, representing a maximum comfortable load for an engine*
- L_P *is an upper limit on the average processing time per processed request, representing a maximum comfortable load for an engine*
- L_I *is an upper limit on the number of work item starts and completions per second, representing a maximum comfortable load for an engine*
- L_T *is an upper limit on the number of worker threads currently executing, representing a maximum comfortable load for an engine*

The upper limit attributes should be configurable, to account for different hardware infrastructures, while the weightings can be set for each so that an accurate factor for any platform can be established. The final busyness factor calculation (B) for an engine is given as:

$$B = ((((R/L_R) \cdot W_R) + ((P/L_P) \cdot W_P) + \\ ((I/L_I) \cdot W_I) + ((T/L_T) \cdot W_T))/4) \cdot 100$$

A final metric for the execution load a process engine is currently experiencing can be defined as the total of all process complexity metrics for currently executing cases combined with the total operational busyness factor for the engine.

3.4 Capability Requirements Analysis

We define a set of specific requirements for a load balancer so that it can reach different levels of capability in distributing the load of process instances to the appropriate process execution engine at run-time. The even distribution of load across all active engines in an array can be supported through the dynamic calculation of the *busyness* of each engine using different strategies, leading to the following four capability levels.

– *Level 0:* Applying a general work scheduling or resource allocation mechanism (e.g. random choice) *without* considering how busy each engine is. This is essentially equivalent to a default stateless load distribution.
– *Level 1:* Considering each engine's busyness *at the exact instant of time* when a new case is required to be distributed to an engine for execution.
– *Level 2:* Considering each engine's busyness *within the time period* of a sliding window looking *backward* (i.e. a number of time units in the past) from the time of case distribution.
– *Level 3:* Considering each engine's busyness *within the time period* of a sliding window looking *forward* (i.e. a number of time units in the future) from the time of case distribution.

4 Performance Analysis and Validation

4.1 Proof of Concept

A prototype that implements the conceptual design shown in Fig. 1(b) has been realised in the YAWL environment [9]. YAWL was selected as the implementation platform because it is open-source, stable, and offers a service-oriented architecture, allowing the new load balancing component to be implemented independent to the existing components. Importantly, absolutely no changes were required to be made to the YAWL environment itself in enabling support for an array of YAWL engines in a cloud environment.

The load balancer component manages a set of available engines. It is situated as a middleware layer that captures all interface calls from external worklist endpoints and redirects them to the appropriate process engine for processing. This redirection is achieved by a trivial configuration change in each worklist so that rather than having a single engine as its remote endpoint, that endpoint instead refers to the load balancing component. The prototype component currently supports all four load distribution capability levels (0–3) described in the previous section. The component is extensively configurable, and changes to configuration values are applied in real time.

Each time the load balancing component receives a request to launch a new process instance, it will poll each engine for its current busyness factor, in terms of the currently configured capability level, and then pass the request to the least busy engine, except when configured to distribution capability level 0 when a random choice is made. For all other (non-launch case) requests, the routing table is queried and the request directed to the engine handling the relevant process instance.

4.2 Implementation

Whenever the load balancing component receives a request to launch a new case, it uses a selection algorithm to select the *idlest* (least busy) engine, based on the currently configured capability level. The method for engine selection on case launch is shown in Algorithm 1. The procedure begins with the current capability level (*mode*) and the set of active engines (*eSet*) (line 1). A reference to the current least busy engine is stored in the *idlest* attribute, and the lowest busyness value is stored in the *lowb* attribute (primed on line 6). Each engine's busyness factor is derived, depending on the current capability level (line 8). If it is less than the current value of *lowb* attribute, then that value is stored in *lowb* (line 10) and a reference to that engine is stored in the *idlest* attribute (line 11). Once all engines have been processed, the idlest engine reference is returned.

Algorithm 1. Selection of Idlest Engine

```
 1: procedure IdlestEngine($mode, eSet$)
 2:     if $mode.random$ then                          ▷ Baseline random distribution
 3:         return RandomItem(eSet)
 4:     end if
 5:     $idlest \leftarrow nil$
 6:     $lowb \leftarrow max$                          ▷ Prime lowest load attribute
 7:     for all $e$ in $eSet$ do                       ▷ For each engine
 8:         $b \leftarrow$GetBusyness($mode, e$)       ▷ Get engine's load
 9:         if $b < lowb$ then
10:             $b \leftarrow lowb$
11:             $idlest \leftarrow e$                  ▷ This $e$ is current idlest
12:         end if
13:     end for
14:     return $idlest$
15: end procedure
```

A different method is used to find the idlest engine at each capability level on a case launch request, as detailed below.

Capability Level 0. At level 0, a random number, limited by the number of available engines, is generated in the programming environment and then used to select an engine from the set.

Capability Level 1. At level 1, each engine is immediately polled for a busyness factor, comprising the various measures described in Sect. 3.3 that exist for the engine at that instant, the total complexity metric of all the cases currently running on the engine, and the configured weightings for each.

Capability Level 2. When the load balancer is set to distribution capability level 2, a backward sliding window to smooth factor readings is applied. Level 2 uses an exponentially weighted moving average to calculate engine busyness, which allows for a configurable weighting factor to be applied in an exponentially decreasing manner to older readings.

It is calculated recursively:

$$S_t = \begin{cases} Y_1, & t = 1 \\ \alpha \cdot Y_t + (1-\alpha) \cdot S_{t-1}, & t > 1 \end{cases}$$

where:

- α is the weighting factor (a.k.a. the forget factor), between 0 and 1, representing the degree of decrease for older readings, which diminish more quickly as values of alpha approach 1.
- Y_t is the engine busyness reading at time instance t.
- S_t is the weighted moving average at time instance t.

Capability Level 3. Level 3 dynamically invokes one of three algorithms, depending on the current busyness values stored as a time series, which attempt to predict near-term future values of the series. The three algorithms used are: single variable regression, single variable polynomial regression, and multiple variable linear regression[1]. Each algorithm seeks to extrapolate from the previous set of busyness measures for each engine a trend line or curve that best fits the data, then uses that trend to forecast upcoming busyness values. If a threshold of time series data values has not yet been met, this strategy reverts to capability level 2.

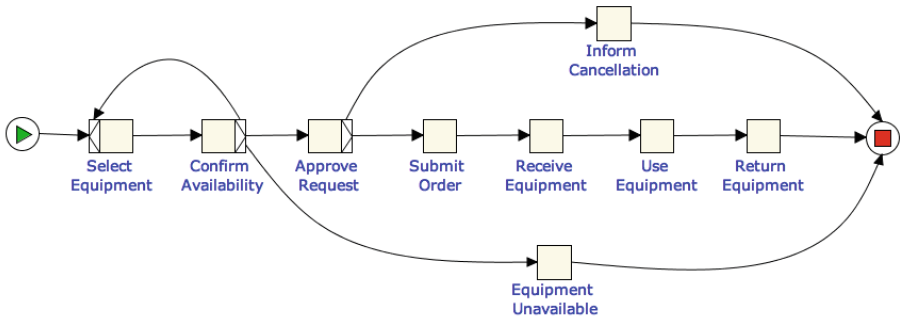


Fig. 3. *Rental Equipment* model used for testing

[1] The java-based OpenForecast libraries are used to dynamically select the best-fit algorithm and to perform the calculations (www.stevengould.org/software/openforecast/).

4.3 Validation Environment

We installed the load balancing component and four YAWL engines on the QUT High Performance Computing platform, using a discrete virtual machine for each engine, and another for the load balancing component. Each VM ran on a single core of an Intel Xeon CPU E5-2680 @ 2.70 GHz, and 4 Gb of RAM was available to each.

A simulation tool was used to launch a new instance of a given process specification every 3 s for a total of 200 instances. The tool supports the entire execution of each case by processing work items with resource ‘robots’, i.e. automated agents that mimic the role of human resources. These agents were configured to simulate processing of each work item for a randomised period within upper limit of between 100 and 3000 ms.

Two process specifications were used for testing. The first is a mostly linear model of an equipment rental process, with a process loop included between the *Select Equipment* and *Confirm Availability* tasks (Fig. 3) to add some variation between cases. An XQuery function was applied to each XOR-split predicate to simulate actual behaviour, with the loop actuating on approximately 40% of *Confirm Availability* task completions, and each cancellation task occurring in approximately 10% of cases.

The second test specification was based on the example found in [1], and models a travel booking process (Fig. 4). Again, an XQuery function was applied to the OR-split predicates following the *register itinerary* task so that for each instance of the specification, one, two or all of the subsequent tasks were executed.

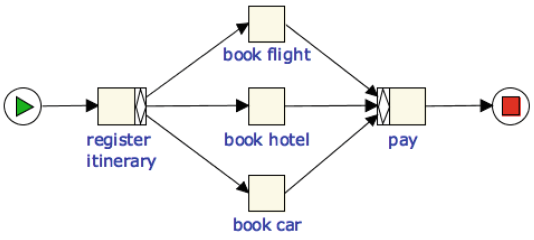
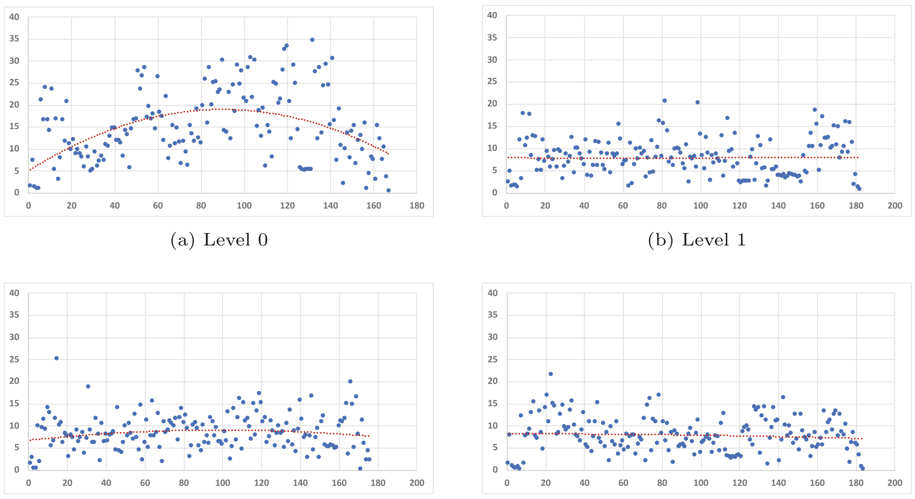


Fig. 4. Travel booking model used for testing (from [1])

4.4 Validation Results

The scatter charts in Fig. 5 show the degree to which the load balancer was able to spread the load across the four engines in our test, for each of the load distribution capability levels, when running simulated cases of the *Rental Equipment* specification. Each chart is represented as a time series graph, where the x-axis specifies a series of equal time slices and the y-axis specifies the number of standard deviations between the set of busyness factors reported for each engine at



(a) Level 0

(b) Level 1

(c) Level 2

(d) Level 3

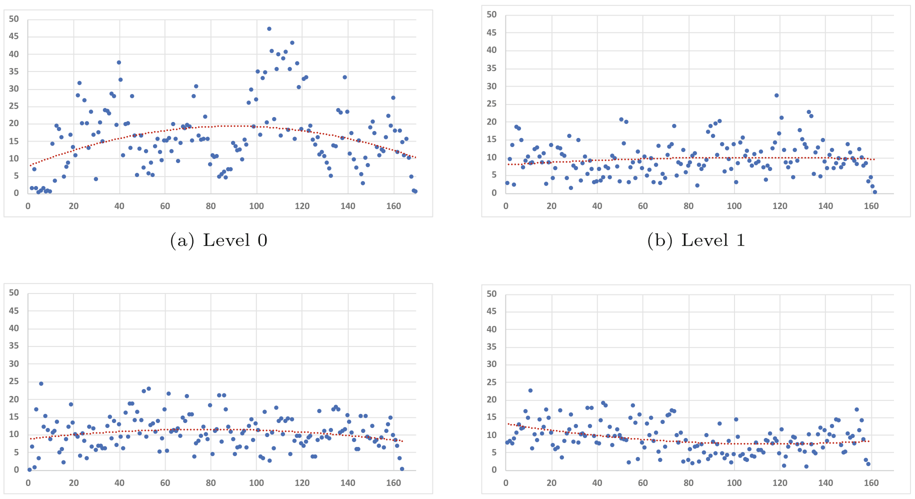
Fig. 5. Scatter charts showing simulation results using *Rental Equipment* model at each load distribution capability level

each time slice. The number of standard deviations is a sound measure of the uniformity of load distribution, where a lower value represents a better balancing of engine loads across all engines.

At distribution capability level 0 (i.e. random choice), as expected there is a wide variance in the busyness values of each engine at any particular time slice (Fig. 5(a)). Almost half of the values exceed 15 std. devs. and 14% exceed 25 std. devs. These results illustrate the inappropriateness of typical stateless load balancing algorithms for supporting stateful, possibly long lived process executions.

Setting the load balancer to distribution capability level 1, which calculates a busyness value for each engine at the moment a new case start is requested, can be seen in Fig. 5(b). It is evident that there is an immediate strong improvement over the level 0 random distribution when each engine's current busyness is taken into account, with a narrowing of the range of engine busyness factors at each time slice. Almost all values are under 20 std. devs., and over 92% are less than 15 std. devs. However, since snapshot values are used for level 1 distribution, there is still some variance between engines at times, as should be expected.

When the load balancer is set to distribution capability level 2, which uses a backward sliding window to smooth each engine's busyness factor readings, the narrowing of busyness distributions per time unit becomes even more marked, as can be seen in Fig. 5(c). Capability level 3, which uses a prediction algorithm to envisage future busyness movements, shows a further, but less marked, refinement (Fig. 5(d)).


(a) Level 0

(b) Level 1

(c) Level 2

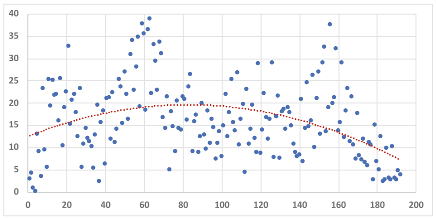
(d) Level 3

Fig. 6. Scatter charts showing simulation results using *Travel Booking* model at each load distribution capability level

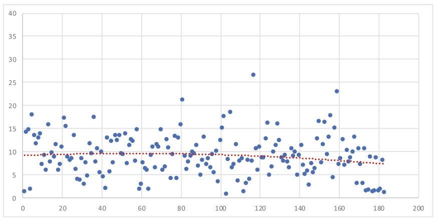
A similar set of outcomes for the *Travel Booking* specification can be seen in Fig. 6, which shows that the load balancing component is able to ensure a reasonably even load distribution for models with differing complexity metrics, especially at the higher capability levels.

Finally, Fig. 7 shows the results of a simulation at each level when a random combination of the two test specifications were used, that is, when engines executed a number of each kind of specification simultaneously. While the overall shape of each scatter chart is similar to those when only one specification was used, there is some widening of data points, especially at the lower capability levels, which may indicate more work needs to be done to fine tune the busyness measures and weights to better manage concurrent specifications of different complexities.

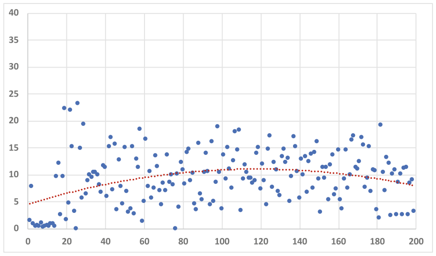
It can be seen that the algorithms at each subsequent capability level refine the ability of the load balancing component to maintain a relatively even load across all available engines, with levels 1–3 strongly outperforming the typical stateless load balancing strategy and level 3 with look-ahead prediction performing best of all. By tuning how the busyness of a stateful process engine can be measured, real gains can be achieved in ensuring engines avoid being overloaded to the point where client response times start to suffer. This in turn can help to guarantee Quality of Service thresholds. From our testing, it is clear that implementing an intelligent load algorithm for stateful process management provides improved load balancing outcomes over stateless or random-choice approaches.



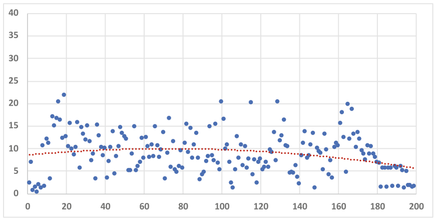
(a) Level 0



(b) Level 1



(c) Level 2



(d) Level 3

Fig. 7. Scatter charts showing simulation results when both test specifications are combined

5 Related Work

An overview of the related literature shows that most of the existing studies focus on the provision of cloud-based SaaS that can be executed for multiple tenants on demand, while the overall stateful process execution is still deployed, operated and managed locally. For example, in [15] the authors discuss a multi-tenant BPM architecture, but utilising only a single-instance Apache ODE system in the cloud and allowing multiple tenants to run their process instances on it. There is no discussion on performance management, or running more than a single instance.

There are a few approaches that require a new BPMS to be developed from the ground up, rather than leveraging existing systems in their entirety in the cloud as discussed in this paper. For example, in [8] the authors propose a hybrid architecture that offers distributed process executions across both client and server sides, although the cloud-based executions are limited to invoked web services. As another example, the research in [13] focusses on an event-based distributed process execution environment coupled with SLA models.

Some of the relevant research emphasizes Quality-of-Service (QoS), such as the generic QoS framework for cloud-based BPMS presented in [12], which comprises QoS specification, service selection, monitoring and violation handling. The authors also present a QoS framework that provides a hierarchical scheduling strategy for costing purposes [20]. Such frameworks guide the design of new support systems based in the cloud, rather than primarily informing the cloud-based hosting of existing BPMS.

Benchmarking BPM systems is an important requirement to determine the optimal method for load balancing and costing. The work reported in [5–7] makes use of containers such as *Docker* to provide a test bed for producing repeatable and trusted benchmarking results. The authors propose the grouping of metrics into three levels: engine (overall throughput, latency, hardware utilisation), process (utilisation per instance) and feature (language impacts). Their *BenchFlow* framework simulates users and interactions to enable meaningful comparisons between BPM systems. The same group of authors have also performed micro-benchmarking of some of the more common workflow patterns and have discussed the effects that pattern complexity may have on performance metrics [18]. While not specifically geared towards cloud-based BPM, this work will inform the future direction of our research in developing more precise load balancing capabilities and costing strategies.

Security and privacy issues associated with BPMS solutions in a multi-tenant cloud is another interesting topic, which will be addressed formally in our future work. In this space, an overview of the security and privacy challenges when dealing with multi-tenanted data in the cloud is presented in [2]. The authors also provide a very broad description of different cloud architectures, but there is no discussion about load balancing in a multi-tenanted, multi-engined environment. In [19] the authors put forward an approach that separates process execution and data management through the use of so-called ‘self-guided artifacts’, thereby providing data security in a multi-tenanted environment.

Finally, an approach relevant to that presented in this paper is described in [17], where the authors define an architecture for hosting multiple YAWL engines in the cloud for multiple tenants, but only offer a basic load balancing strategy.

6 Conclusion

In this paper, we have detailed a scalable system architecture, with a focus on the design of load balancing strategies, for deploying BPMS in a multi-tenanted cloud environment. Further, we have developed and validated a prototype implementation of a load balancing component that is able to distribute system load over an array of process engines at different capability levels. We have also demonstrated that such a load balancing component can successfully handle a large number of concurrent process instances while balancing system resources, in particular process engines, in a cloud-based BPMS. Given that our design is generic and independent of any specific system or environment, it can be used to serve as a reference for the implementation of specific BPMS in the cloud.

There are several clear directions for future work. One is to develop more precise load balancing capabilities, by considering more complex and accurate measures for runtime process model complexity metrics, and indeed for process engine busyness measures too. Another direction is to develop a further capability level, one which interrogates process logs to determine actual experiential measures of past specification executions and uses that data as input into forecasting methods. A third future direction is to add support for dynamically increasing and/or reducing the number of active process engines at run-time to achieve better scalability and economy of resources. Finally, consideration of machine learning techniques to gradually tune the load balancing algorithms for maximal load smoothing across multiple workflow engines will be another interesting focus of future work in this area.

Acknowledgments. This work is supported by the Research Foundation of Science and Technology Plan Project in Guangdong Province (2016B050502006) and the Research Foundation of Science and Technology Plan Project in Guang-zhou City (2016201604030001).

References

1. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: yet another workflow language. Inf. Syst. **30**(4), 245–275 (2005)
2. Anstett, T., Leymann, F., Mietzner, R., Strauch, S.: Towards BPEL in the cloud: exploiting different delivery models for the execution of business processes. In: Proceedings of the International Workshop on Cloud Services (IWCS 2009), pp. 670–677. IEEE Computer Society (2009)
3. Cardoso, J.: Control-flow complexity measurement of processes and Weyuker's properties. In: 6th International Enformatika Conference, vol. 8, pp. 213–218 (2005)
4. Chong, F., Carraro, G.: Architecture strategies for catching the long tail. MSDN Library, April 2006
5. Ferme, V., Ivanckikj, A., Pautasso, C., Skouradaki, M., Leymann, F.: A container-centric methodology for benchmarking workflow management systems. In: Proceedings of the 6th International Conference on Cloud Computing and Service Science, pp. 74–84. SciTePress (2016)
6. Ferme, V., Ivanchikj, A., Pautasso, C.: A framework for benchmarking BPMN 2.0 workflow management systems. In: Motahari-Nezhad, H.R., Recker, J., Weidlich, M. (eds.) BPM 2015. LNCS, vol. 9253, pp. 251–259. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23063-4_18
7. Ferme, V., Ivanchikj, A., Pautasso, C.: Estimating the cost for executing business processes in the cloud. In: La Rosa, M., Loos, P., Pastor, O. (eds.) BPM 2016. LNBIP, vol. 260, pp. 72–88. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45468-9_5
8. Han, Y.-B., Sun, J.-Y., Wang, G.-L., Li, H.-F.: A cloud-based BPM architecture with user-end distribution of non-compute-intensive activities and sensitive data. J. Comput. Sci. Technol. **25**(6), 1157–1167 (2010)
9. ter Hofstede, A.H.M., van der Alast, W.M.P., Adams, M., Russell, N. (eds.): Modern Business Process Automation: YAWL and its Support Environment, 1st edn. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-03121-2
10. Hollingsworth, D.: Workflow management coalition: the workflow reference model. Technical report, TC00-1003, January 1995
11. Lassen, K.B., van der Alast, W.M.: Complexity metrics for workflow nets. Inf. Softw. Technol. **51**(3), 610–626 (2009)
12. Liu, X., Yang, Y., Yuan, D., Zhang, G., Li, W., Cao, D.: A generic QoS framework for cloud workflow systems. In: 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing, pp. 713–720. IEEE (2011)

13. Muthusamy, V., Jacobsen, H.-A.: BPM in cloud architectures: business process management with SLAs and events. In: Hull, R., Mendling, J., Tai, S. (eds.) BPM 2010. LNCS, vol. 6336, pp. 5–10. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15618-2_2
14. Ouyang, C., Adams, M., ter Hofstede, A.H., Yu, Y.: Towards the design of a scalable business process management system architecture in the cloud. In: Trujillo, J. (ed.) ER 2018. LNCS, vol. 11157. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00847-5_24
15. Pathirage, M., Perera, S., Kumara, I., Weerawarana, S.: A multi-tenant architecture for business process executions. In: 2011 IEEE International Conference on Web services, pp. 121–128. IEEE (2011)
16. Polančič, G., Cegnar, B.: Complexity metrics for process models-a systematic literature review. Comput. Stand. Interfaces **51**, 104–117 (2017)
17. Schunselaar, D.M.M., Verbeek, H.M.W., Reijers, H.A., van der Aalst, W.M.P.: YAWL in the cloud: supporting process sharing and variability. In: Fournier, F., Mendling, J. (eds.) BPM 2014. LNBIP, vol. 202, pp. 367–379. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15895-2_31
18. Skouradaki, M., Ferme, V., Pautasso, C., Leymann, F., van Hoorn, A.: Micro-benchmarking BPMN 2.0 workflow management systems with workflow patterns. In: Nurcan, S., Soffer, P., Bajec, M., Eder, J. (eds.) CAiSE 2016. LNCS, vol. 9694, pp. 67–82. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39696-5_5
19. Sun, Y., Su, J., Yang, J.: Separating execution and data management: a key to business-process-as-a-service (BPaaS). In: Sadiq, S., Soffer, P., Völzer, H. (eds.) BPM 2014. LNCS, vol. 8659, pp. 374–382. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10172-9_25
20. Wu, Z., Liu, X., Ni, Z., Yuan, D., Yang, Y.: A market-oriented hierarchical scheduling strategy in cloud workflow systems. J. Supercomput. **63**(1), 256–293 (2013)