A Qualitative Discussion of Different Approaches for Implementing Multi-Tenant SaaS Offerings

Christof Momm, Rouven Krebs

SAP Research
Vincenz-Priessnitz-Str. 1
76131 Karlsruhe, Germany
christof.momm@sap.com
rouven.krebs@sap.com

Abstract: The upcoming business model of providing software as a service (SaaS) not only creates new challenges for service providers but also for software engineers. To enable a cost-efficient service management, the hosted application should support multi-tenancy. For implementing multi-tenancy, several options are available, whereas all of them to a certain degree require a reengineering of the application at hand. Since for many applications this represents the next big evolution step, this paper is devoted to discussing different options for implementing multi-tenancy regarding the initial reengineering effort required for meeting the customer expectations versus cost-saving potential.

Introduction

Today, we observe a paradigm shift in the way software is delivered to customers. While it used to be the common practice to ship software packages to the customer and operate the deployed software on premise, nowadays the consumption of certain software as on demand services (i.e. Software-as-a Service, short SaaS) becomes more popular [DW07]. Customers expect from such a SaaS offering flexibility in case of changing requirements, a contractually guaranteed quality of service based on service level agreements (SLAs) [TYB08] and a demand-oriented billing. Major SLAs for SaaS offerings include responsiveness, unplanned downtime (availability), planned downtime and data safety (in particular backup) guarantees.

Acknowledgement: This work is supported by the German Federal Ministry of Education and Research under promotional reference 01jG09004 (ValueGrids) and received funding from the European Union's Seventh Framework Programme FP7/2007-2013 (4CaaSt) under grant agreement n° 258862.

For service providers, the key success factor is to offer scalable demand-oriented services (also referred to as elastic cloud services) [BYV08] with high SLA compliance, while achieving low total cost of ownership (TCO), in particular low incremental costs. The TCO for customization, provisioning and operation of software can be significantly reduced by introducing multi-tenancy, where "user requests from different organizations and companies (*tenants*) are served concurrently by one or more hosted application instances based on the shared hardware and software infrastructure" [GS+07].

For implementing multi-tenancy, several options are available, reaching from shared single parts of the application stack, such as infrastructure or middleware, to shared application instances. The incremental operating costs for a tenant thereby highly depend on the level where multi-tenancy is introduced in the application stack. This is because each layer involves a common, non-tenant-specific overhead, which can be saved by introducing resource sharing. Thus, the shared application instances in the long term always represent the TCO-minimal solutions. However, at the same time each of these options to a certain degree requires the reengineering of existing applications, in particular if the resulting solution should meet customer expectations regarding quality of service and scalability. Hence, which option the software engineer / provider should decide on not only depends on the expected incremental operation costs, but also on the complexity of the necessary modifications for the given application in combination with the expected service lifetime as well as number of customers.

For many existing applications, multi-tenancy enablement (as part of a general SaaS enablement) represents the next big evolution step. Therefore, this paper aims at helping decision makers in comparing different multi-tenancy implementation options for a given SaaS scenario regarding the resulting TCO. To this end, we first point out major customer requirements as well as the most prominent options for multi-tenancy-enabling an existing application. Then, we introduce a simple cost model explaining the major cost drivers, both fixed and variable. Finally, we provide a qualitative discussion of the different implementation alternatives regarding the fixed initial reengineering effort (fixed costs) required for meeting customer expectations and the variable costs for operation. This discussion is based on our experiences with large-scale business applications and considers customer requirements such as (dynamic) scalability as well as guarantees regarding responsiveness, planned downtime and provisioning time. Availability, data safety, security requirements are not in scope.

Customer Requirements and Multi-Tenancy Implementation Options

This section introduces the considered customer requirements on SaaS offerings along with different feasible options for implementing a corresponding multi-tenant solution on the provider side.

Looking at existing SaaS offerings and talking to customers, we identified the following major requirements and multi-tenancy challenges:

- *Provisioning or System-ready-to-use time:* Defines the time for provisioning a new service instance to a customer. Customers thereby expect getting a fully-operational instance offered within minutes to hours, not hours to days.

- *Responsiveness:* For a given customer workload threshold, the application must not drop below a certain lower response time and/or throughput bound, e.g. sub-second average response times. In a multi-tenant setting different tenants share the same resources, which creates the risk that tenant workloads interfere with each other. An overload created by one tenant may therefore negatively impact the performance of another tenant. Accordingly, adequate performance isolation is required on the provider side.

- *(Dynamic) Scalability:* Customers expect from SaaS offerings that they adapt to changing workload requirements in a very flexible way. Ideally, such an adaptation happens fully dynamic (within seconds), but even a more static approach (within hours to days) in most cases is already sufficient.

- *Planned Downtime:* Since the software still evolves over time, the provider requires time for regular maintenance and upgrade activities. Customers expect guaranteed upper limits for downtimes and time frames, e.g. weekend only. Furthermore, it might happen that different customers expect different time frames, with only little or even without any overlap. A multi-tenant setting may introduce additional dependencies between tenant upgrade or patch activities leading to significantly higher time demands.

- *Availability (Unplanned Downtime):* Refers to the service uptime without planned downtimes. A customer-friendly definition of available ("up") would be that the client is able to use the processes of his scoping, which implies that all involved components are online. Due to the shared usage of resources a failure at a single point may cause the violation of several tenant SLAs at once.

- *Data safety:* Adequate mechanisms must be in place to avoid data loss. SLAs mostly include statements about backup mechanisms, in particular recovery point and recovery time objectives. Special backup mechanisms must be in place to support a tenant-specific rollback of data without overwriting non-affected data while ensuring data isolation.

- *Security/Privacy:* Data security and legal requirements regarding storing locations etc. must be ensured. In a multi-tenant setting the provider has to ensure a strict logic data isolation between tenants. Since tenants might share the same data base or even the same table the application logic's complexity might increase.

It is worth to mention, that the requirements themselves as well as their importance strongly depend on the customers size [SS+10].

In [KM08] and [OG+09] several options for implementing a cost-efficient multi-tenant solution are presented. In the following, we briefly introduce the option we think are generally suitable for meeting these requirements. Figure 1 illustrates the different options. All of these options have in common that still several instances of the shared part may exist, e.g. several physical servers with a VM installed or several shared middleware instances. Hence, the allocated elements may be moved to another instance of the shared resources, which represents one way for (dynamically) scaling of resources.

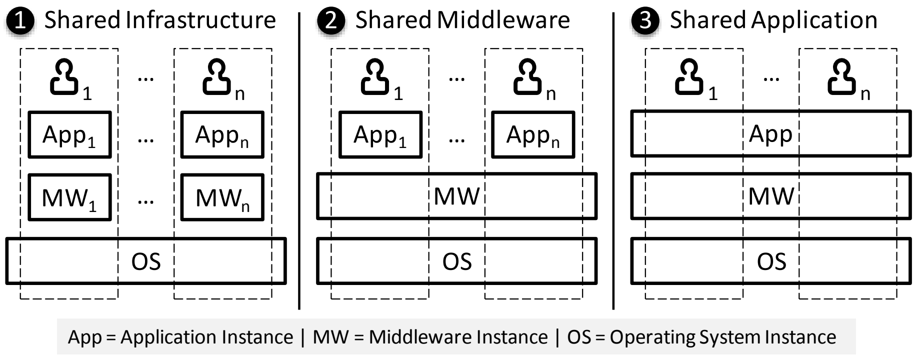

Figure 1: Considered Multi-Tenancy Implemenation Options

In the following, we briefly introduce the specifics of the different options.

- *(1) Shared Infrastructure:* In this case, only the infrastructure is shared amongst tenants using virtualization technologies. To enable a convenient provisioning of tenant instances, virtual appliances comprising application and middleware instances have to be created. So data isolation is per se given, while performance isolation has to be ensured by the VM technology. Scalability can either be achieved by increasing resources assigned to a VM (scale up) or by moving a VM to another shared hardware (scale out). All application lifecycle management (LM) procedures (upgrade/patch) may remain as they are.

- *(2) Shared Middleware:* This option involves a shared middleware instance on top of a shared OS, which may be either placed on a physical or virtualized hardware. Data isolation in this case is also per se given and the LM procedures on the application level do not have to be changed. Fast provisioning, performance isolation and scalability by contrast have to be natively supported by the middleware. Regarding the scalability, both a scale up of the shared middleware instance (e.g. using VM technology) and a scale out of tenants to another shared middleware instance is feasible.

- *(3) Shared Application:* This option involves least overhead per tenant since already the application instance shared amongst tenants. However, at the same this option probably involves most reengineering activities. Data and performance isolation have to be added on the application level. Similarly, scalability and LM procedures might require significant changes of the application.

Simple Cost Model for Evaluation

Depending on the application that should be offered as service and the option used for implementing multi-tenancy, different reengineering measures of varying complexity are necessary for fulfilling this requirement. The goal of decision makers (i.e. service providers in cooperation with the responsible software engineers) is to find the cost-minimal solution for the considered application. This section therefore introduces a coarse-grained cost model that helps comparing different alternatives.

This cost model includes two major components:

- *Initial reengineering costs:* Initial costs for reengineering activities required once for implementing the chosen multi-tenancy option for a given application.

- *Continuous operating costs:* Monthly costs for operation. This includes fixed costs per instance (application, middleware or hardware) as well as costs per tenant.

Accordingly, every reengineering activity that reducing the monthly operating costs will sooner or later be amortized. This breakeven point is calculated as follows

$$NumMonthToBreakEven = \frac{Intial\ Reeng. Costs}{\Delta Operating\ Costs}$$

If the service is offered forever, the variant with the lowest incremental cost is always the best. However, we rather assume that a service sooner or later gets replaced. Thus, the question is if this time span is sufficiently long to justify huge investments (i.e. service usage time > time to break even). To determine the breakeven point the operating costs have to be estimated. These operating costs comprise monthly costs for basic tenant-independent efforts produced by the operated instances of the application stack instance (application, middleware or operating system) as well as tenant-specific application efforts. Hence, the operating cost function depends on the selected implementation option and the number of expected number of tenants n:

$$OpCost_{sharedInfra} = CostOS_{Base} + n * (CostMW_{Base} + CostApp_{Base} + Cost\_App_{Tenant})$$

$$OpCost_{sharedMW} = CostOS_{Base} + CostMW_{Base} + n * (CostApp_{Base} + Cost\_App_{Tenant})$$

$$OpCost_{ShareApp} = CostOS_{Base} + CostMW_{Base} + CostApp_{Base} + n * Cost\_App_{Tenant}$$

Accordingly, introducing an additional shared resource in the stack saves $\Delta Operating\ Costs$ of (n-1) times the base costs for the shared resource. Factors influencing these base costs are resources demanded by the produced base load, maintenance efforts as well as lifecycle management efforts.

Note that these cost functions in reality might not be linear. However, it is probably not feasible to determine the exact cost functions and they are still suitable for comparing different options.

The single components of this cost function have to be estimated for the different options for multi-tenancy-enabling the application at hand. More precisely, the effort for the different required modifications (reengineering activities) *as well as* the expected maintenance/operations costs has to be estimated for each alternative. Some feasible approaches for estimating these efforts are discussed in section "Related Work". To support this effort estimation, in the following section we point out typical modifications required for multi-tenancy-enabling existing applications and discuss selected aspects regarding the resulting operating costs for the different implementation options.

Discussion of Reengineering Effort per Implementation Options

In this section, we discuss the potential reengineering effort for each multi-tenancy implementation as well as some aspects regarding the operating costs, in particular the expected resource consumption. The discussion is based on our experiences with large-scale business information systems and considers the application and middleware layer. The reason for this is that the middleware might be a proprietary development, which might have to be enhanced as well. The scope of the discussion is limited to certain customer requirements pointed out in section 2. Availability (Unplanned Downtime), data safety and security/privacy are not further discussed in view of the reengineering measures.

Shared Infrastructure

Initial Reengineering Effort:

Provisioning or System-Ready-to-Use Time
It should be possible to create new instances of the stack from master images. To this end, it might be necessary to relax existing copyright protection mechanisms, in particular hardware keys. Moreover, it must be possible to change addressing information dynamically after installation. Static addressing information has to be removed.
Responsiveness
Responsiveness in this case depends on amount of virtualized resources and performance isolation is handled by VM technology. It should be checked beforehand that the used VM technology does not cause additional performance problems. If so, it

might be necessary to change parts of the implementation, e.g. avoid certain memory operations.
(Dynamic) Scalability
To support a growing customer workload, the resources assigned to the respective VM can be increased up the limits of the physical hardware. Performing this VM-based scaling dynamically, i.e. without having to restart/reboot the system, has to be supported by the used middleware and operating system. Both have to detect additional resources (in particular CPU and memory) on the fly. Extending an existing middleware might be very complex. The application in turn has to be able to deal with multiple cores. Otherwise, this might cause significant reengineering effort.
Planned Downtime
Since each customer gets its own middleware and application instance, there are no side effects with other tenants. Thus, the upgrade procedures may remain the same as in an on premise setting as long as the upper limit regarding the planned downtime is sufficient. Otherwise, the corresponding LM procedures have to be revised as well.

Operating Costs (Resource Demand): "Shared Infrastructure" involves the lowest degree of resource sharing. Thus, providers per se have to deal with a significant overhead (VM, middleware and application base load). Furthermore, information about application internals is not available, which limits the potential of optimizing resource utilization and probably results in higher demands.

Shared Middleware

Provisioning or System-Ready-to-Use Time
The middleware has to support running multiple deployments of the same application components. Regarding the application level, just like in case of the shared infrastructure, it might be necessary to enable a dynamic addressing. Moreover, it is likely that configuration, monitoring and logging have to be revised in order to support multiple application instances on one middleware. Quite often, there is only one configuration/monitoring/logging file per application and multiple deployments of the same application instance are not supported.
Responsiveness
Concerning the responsiveness, the multiple application deployments may interfere with each other and by this create critical overload situation. To avoid this, the middleware has to provide adequate performance isolation mechanisms, e.g. fixed thread or memory quota per application instance (i.e. tenant) or tenant-aware request queuing. This should ideally be transparent to the application.
(Dynamic) Scalability
If no VM technology is used, one way for reacting on changing workload requirements is moving an application instance to another less occupied middleware instance. In case application caches exist, performance might drop significantly if these caches are not moved as well. A dynamic scaling would then require a live migration of application

instances including the complete application cache, just like it is already supported by some VM technology on the infrastructure level [CF+05]. Another option for managing workloads is to introduce a load balancing between application instances by actively managing their resource quotas.
Planned Downtime
In case of application upgrades which do not affect the middleware the upgrade procedures may remain the same as in an on premise setting as long as the upper limit regarding the planned downtime is sufficient. However, a middleware upgrade probably requires upgrading *all* application instances (tenants). To ensure that all of them finish within the given time limit, the existing upgrade procedures might have to be adapted, e.g. introduce parallel processing and avoid synchronization points.

Operating Costs (Resource Demand): In this scenario, hardware, operating system and middleware are shared and only the multiple application instance cause additional overhead. The resource optimization potential is better than in the "Shared Infrastructure" scenario, but still limited since application-specific information is missing. Moreover, the resource demand heavily depends on the available middleware capabilities. If live migration of application instances without cache loss is not supported, the demand is much higher.

Shared Application

Provisioning or System-Ready-to-Use Time
Tenants in this case are solely represented as a set of tenant-specific data, meta-data and configurations within an application instance. The application has to be modified in a way that new tenants may be created automatically, e.g. using pre-defined templates. As prerequisite, this tenant-specific information has to be isolated, which might not be trivial [OG+09]. More work on how to enable existing applications to support multi-tenancy is presented in section "Related Work".
Responsiveness
Tenant workloads necessarily interfere with each other when sharing a single application instance. Performance-isolation is strongly required, but implementing it solely on the application level is probably not feasible. Instead, the middleware has to expose an API for effectively assigning resources to tenant-specific transactions, whereas the application has to implement adequate mechanisms for controlling the tenant/resource allocation.
(Dynamic) Scalability
If a VM technology is used, it is possible to scale up the resources assigned to the application instance (given that changes in the infrastructure are recognized). In addition to this, tenants could be moved to another less busy application instance (scale out). Just like in case of the shared middleware, a tenant-move should then consider all tenant-specific cache data. The application has to be extended by adequate management interfaces, which allow easily retrieving this tenant data. For supporting a live migration

does probably not work without changing the middleware as well. At least the message dispatcher has to be aware of the two running application instances and perform the final switch.
Planned Downtime
For multi-tenancy-enabled applications, it is very likely that the LM procedures have to be adapted to distinguish between common and tenant-specific steps. In addition to this, developers have to consider the fact that all LM procedures have to be performed for the whole application instance in a transactional manner, but still involve tenant-specific activities. Ensuring adherence to the given planned downtime constraints might therefore require major revision to the existing LM procedures. Parallel processing of tenant upgrades and avoidance of synchronization points across tenants is crucial. Also, adequate roll-back mechanisms should be provided in case it is not possible to finish an upgrade/patch in time. Supporting tenant-specific timeframes in this case might not be feasible any more. The only option we can think of is putting together tenants with similar time frame requirements, meaning that upgrade constraints have to be considered within scope of the tenant placement.

Operating Costs (Resource Demand): This scenario involves the highest degree of resource sharing and therefore introduces the minimum possible overhead in terms of additional base load. Moreover, it offers the best resource optimization potential since very fine-grained application-level monitoring is possible. This particularly allows a better prediction of resource demands based on given customer workloads. However, again the optimization potential heavily depends on the maturity of the tenant lifecycle management operations, in particular the resource demand for tenant moves.

Related Work

To apply the simple cost model, adequate effort estimation techniques are required. In the following we point out some suitable approaches. The reengineering effort for evolution steps can for instance be predicted by using top-down effort estimation approaches such as *Function Point Analysis* (FPA) [IFP99] and evolution-specific extensions to the *Comprehensive Cost Model* (COCOMO) II [BD+03] or bottom-up effort estimation techniques like *Architecture-Centric Project Management* (ACPM) [PB01]. Predicting the maintenance and operation costs is probably more difficult. Besides a bottom-up estimation involving the operations team, a general evaluation of the application's maintainability might be helpful, such as the *Architecture-Level Prediction of Software Maintenance* (ALPSM) [BB99]. In addition to this, systematic measurements as for instance proposed in [WH10] could be used to determine the middleware / application base load.

A major component of the cost model we introduced represents the initial reengineering effort required for introducing multi-tenancy. Looking at approaches, which address the problem of multi-tenancy-enabling existing applications helps getting a better feeling regarding the required measures. Guo et al. [GS+07] explore the requirements for multi-tenant applications and present a framework with a set of services to help people designing and implementing multi-tenant applications in an efficient manner. They provide a solution on architectural level to achieve isolation, in particular regarding performance and availability.

Bezemer and Zaidman [BZ10] additionally consider maintenance risks created by multi-tenant applications. Besides costs for future evolution steps, which are not explicitly addressed by the cost model, the discussion also covers recurring maintenance activities such as patches, which are part of the second component of the cost model, the operating costs. The authors describe the need to separate multi-tenant components from single-tenant logic and provide a conceptual architecture blueprint to reduce the risk of high maintenance efforts. This blueprint distinguishes authentication-, configuration- and database-components. In [BZ+10] the results from [BZ10] be applied to evolve an existing single-tenant application to a multi-tenant one with low efforts.

A major factor influencing the operating costs is the actual resource consumption. Kwok and Mohindra [KM08] provide an approach for calculating the overall resource demand for a multi-tenant application instance. Based on this, they propose a tool for optimizing tenant placements, which may consider various constraints defined by SLAs. This helps to significantly reduce the overall resource consumption and therefore has to be taken into consideration when calculating the operating costs. In a similar way, Westermann and Momm [WM10] placed tenants in a cost-minimal way with help of genetic algorithms, but using more detailed resource consumption / performance models based on the software performance curve approach described in [WH10]. Fehling et al. [FLM10] identify several optimization opportunities originating from an intelligent distribution of users among functionally equal resources with varying QoS.

An inappropriate tenant placement / resource optimization not only results in resource wasting, but may also lead to SLA violations causing additional operating costs. In a similar way, missing performance isolation between tenants can easily lead to costly SLA violations. To avoid these situations, Cheng et al. [CSL09] introduce a framework for monitoring and controlling tenant performance using an SLA aware scheduling. In this way it is possible to verify tenant allocations at runtime. This helps to achieve performance isolation between tenants.

Conclusion

In this paper, we introduced a simple cost model for evaluating different options for implementing multi-tenancy depending on the application at hand, the cost for operation / lifecycle management, the expected number of tenants and the expected period of use. Most of the components of this cost function can be estimated using standard effort estimation approaches used in software engineering. To support the estimation of the initial reengineering effort we furthermore pointed out common modifications necessary to implement die different multi-tenancy options while fulfilling the customer requirements.

However, the complexity of such a multi-tenancy or in general SaaS enablement can probably be significantly reduced by developing adequate migration and reengineering techniques, just like in case of service-oriented architectures. At the same time, existing middleware and programming models should be enhanced to natively support multi-tenancy. This would considerably improve the development of new applications for the SaaS model and create new options for migrating existing applications.

References

[BB99] Bengtsson, P.; Bosch, J.: Architecture level prediction of software maintenance. In: Proc. of the Third European Conference on Software Maintenance and Reengineering, 1999, pp. 139–147

[BL+04] Bengtsson, P.; Lassing, N.; Bosch, J.; van Vliet,H.: Architecture-level modifiability analysis (ALMA), Journal of Systems and Software 69(1-2), 2004, pp. 129 – 147.

[BD+03] Benediktsson, O.; Dalcher, D.; Reed, K.; Woodman, M.: COCOMO-Based Effort Estimation for Iterative and Incremental Software Development, Software Quality Control 11(4),2003, pp. 265-281..

[BYV08] Buyya, R.; Yeo, C. S.; Venugopal, S.: Market-oriented cloud computing: Vision, hype, and reality for delivering IT services as computing utilities. In: Proc. of the 10th IEEE International Conference on High Performance Computing and Communications, 2008.

[BZ10] Bezemer C.; Zaidman, A.: Multi-Tenant SaaS Applications: Maintance Dream or Nightmare. In: Technical Report Series of Delft University of Technology (Software Engineering Group), 2010, Report TUD-SERG-2010-031

[BZ+10] Bezemer, C.; Zaidman, A.; Platzbeecke B.; Hurkmans T.; Hart A.: Enabling Multi-Tenancy: An Industrial Experience Report. In: Technical Report Series of Delft University of Technology (Software Engineering Group), 2010, Report TUD-SERG-2010-030

[CSL09] Cheng, X.; Shi, Y.; Li Q.: A multi-tenant oriented performance monitoring, detecting and scheduling architecture based on SLA. In: Proc. of Joint Conferences on Pervasive Computing (JCPC) 2009, 2009, pp. 599-604.

[DW07] Dubey,A.; Wagle, D.: Delivering software as a service. In: The McKinsey Quarterly, 2007, pp. 1-12

[FLM10] Fehling, C.; Leymann, F.; Mietzner, R.: A Framework for Optimized Distribution of Tenants in Cloud Applications. In: 2010 IEEE 3rd International Conference on Cloud Computing (CLOUD), 2010, pp. 252-259

[GS+07] Guo, C.; Sun W.; Huang Y.; Wang, Z.; Gao B.: A Framework for Native Multi-Tenancy Application Development and Management. In: 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007.

[IFP99] IFPUG. Function Point Counting Practices Manual. International Function Points Users Group: Mequon WI, 1999.

[KM08] Kwok, T.; Mohindra, A.: Resource Calculations with Constraints, and Placement of Tenants and Instances for Multi-tenant SaaS Applications. In: Proc. of Service-Oriented Computing - ICSOC 2008, 2008, pp. 633-648

[OG+09] Osipov, C.; Goldszmidt, G.; Taylor M.; Poddar, I.: Develop and Deploy Multi-Tenant Web-delivered Solutions using IBM middleware: Part 2: Approaches for enabling multi-tenancy. In: IBM's technical Library, 2009, http://www.ibm.com/developerworks/webservices/library/ws-multitenantpart2/index.html

[PB01] Daniel J. Paulish and Len Bass. Architecture-Centric Software Project Management: A Practical Guide. Addison-Wesley Longman Publishing Co., Inc., USA, 2001.

[SS+10] Sripanidkulchai, K.; Sahu,S.; Ruan, Y.; Shaikh, A.; Dorai, C.: Are clouds ready for large distributed applications? In: ACM SIGOPS Operating Systems Review, vol. 44 issue 2, 2010, pp. 18-23

[TYB08] Theilmann, W.; Yahyapour, R.; Butler, J.: Multi-level SLA Management for Service-Oriented Infrastructures. In: Towards a Service-Based Internet, 2008, pp. 324-335

[WH10] Westermann, D.; Happe, J.: Towards performance prediction of large enterprise applications based on systematic measurements. In: Proc. of the Fifteenth International Workshop on Component-Oriented Programming (WCOP) 2010, 2010, pp. 71-78.

[WM10] Westermann, D.; Momm, C.: Using Software Performance Curves for Dependable and Cost-Efficient Service Hosting, In: Proc. of the 2nd International Workshop on the Quality of Service-Oriented Software Systems (QUASOSS '10), 2010.