Dynamic Tailoring and Cloud-based Deployment of Containerized Service Middleware

Santiago Gómez Sáez, Vasilios Andrikopoulos, Roberto Jiménez Sánchez, Frank Leymann, Johannes Wettinger
Institute of Architecture of Application Systems (IAAS),
University of Stuttgart, Stuttgart, Germany
{gomez-saez, andrikopoulos, leymann, wettinger}@iaas.uni-stuttgart.de

***Abstract*—The emergence and consolidation of container-based virtualization techniques has simplified and accelerated the development, provisioning, and deployment of applications for the Cloud. When considering the case of composite service-based applications that rely on service middleware solutions for their operation, container-based virtualization offers the opportunity for rapid and efficient building and deployment of lightweight, optimally configured middleware instances. As such, it provides an ideal tool for the purposes of cloudifying existing middleware solutions and offering them as part of larger PaaS offerings. As part of this effort, our investigation focuses on leveraging and evaluating a container-based virtualization environment towards enabling the assembly, provisioning, and execution of dynamically tailored instances to satisfy service middleware communication requirements of specific applications. For these purposes we scope the discussion on one particular type of messaging middleware for composite service applications, the Enterprise Service Bus (ESB) technology.**

Keywords*—*Platform-as-a-Service (PaaS); ESB Service Middleware; Container Virtualization; SOA; Performance Analysis; Cost Analysis

I. Introduction

Many modern applications are designed and implemented as a combination of multiple services, leveraging existing functionalities offered in the *Everything-as-a-Service* (*aaS) model. Application developers have a plethora of SaaS, PaaS and IaaS offerings to choose from in hosting, deploying, and running their applications, completely or partially (e.g. in a cloud bursting scenario). Looking into the PaaS delivery model in particular, a fundamental building block of PaaS solutions is the capability to offer *service middleware as a service* [1] which enables the realization of composite service applications following the service-oriented computing paradigm [2]. Service middleware allows for the deployment and execution of service-based applications typically defined as sets of executable process models. Previous works [1], [3], [4] have demonstrated that the concept of middleware as a service is both feasible and efficient, focusing however on resource sharing expressed as multi-tenancy of the offered middleware.

In this work we start with the same premise but focus on one particular type of service middleware, and more specifically the *Enterprise Service Bus* (ESB) technology, which essentially acts as the messaging hub between services ensuring their integration. Our previous experiences in cloudifying an ESB solution [5], [6] showed that while scaling an ESB solution horizontally by adding more VMs is easy to implement, it results in underutilization of resources inside each VM. Furthermore, we have also concluded that a significant amount of resources are overutilized due to the provisioning and installation tasks of the ESB in the form of retrieving and deploying a number of features like different queuing mechanisms, messaging format transformers, etc., that are not actually used in conjunction in practice.

The advent of mature container-based virtualization techniques like Docker, as discussed for example by [7], offer us the opportunity to address both these issues in a comprehensive solution. Our work is motivated by the possibility to efficiently and rapidly provision and manage lightweight containers, each one of which hosts a custom configuration of a ESB solution that is dynamically tailored to the needs of a particular application. While we focus on a particular ESB solution for the purposes of this discussion, the approach followed is equally applicable to any service middleware solution. The main contributions of this work can be summarized by the following:

1) A process-based approach aimed at building and deploying dynamically tailored service middleware components for satisfying application requirements.
2) A categorization of the different features offered by well known ESB solutions, e.g. Apache ServiceMix, WSO2, Fuse ESB etc. that is used by this process.
3) A performance evaluation of our proposal considering both dimensions, i.e. the effectiveness of bulding and executing tailored ESB instances, and the impact of dynamic provisioning of tailored ESB instances using the container-based virtualization technology.

The rest of this paper is structured as follows. Section II provides a deeper background on both the ESB technology, as well as additional motivation for the use of container-based virtualization. Section III presents the core of our proposal, including the tailoring and deployment process, and the categorization of ESB solutions into features that allow their dynamic tailoring to specific application requirements. Section IV discusses the experimental evaluation of our proposal and presents our findings. Finally, Sections V and VI close the paper with related work, and conclusions and future work, respectively.

II. Background & Motivation

A. Enterprise Service Bus (ESB)

The Enterprise Service Bus (ESB) technology has been introduced as the fundamental service middleware component in service-oriented architecture (SOA), as it enables the interoperability among applications in a loosely coupled manner [8]. As such, in the last years it has become ubiquitous in service-oriented enterprise computing environments. ESBs typically provide the means for an abstract decoupling between connected applications by creating logical endpoints which are exposed as services and conform a multi-protocol environment, where routing and data transformation are transparent to the service connected to it [8].

A strong connection between Service Oriented Computing (SOC) and Cloud Computing has been forced in the last years towards enabling a rapid adaptation to business changes by using different Cloud services as the underlying resources to provision and host SOA-based applications [9]. The efficient usage of the ESB as the main service middleware component in PaaS Cloud infrastructures to enable the deployment and execution of service-based applications has been widely investigated in [1], [4], [6]. More specifically, in the context of the 4CaaSt[1] EU Project the ESB technology has been adopted as the main building block towards enabling the communication among multi-tier applications in an elastic manner [5]. However, as a first step towards this challenge, multiple works focused on cloudifying the ESB, i.e. enhancing the service middleware with multi-tenant aware management and communication capabilities, e.g. [4], [6].

Provisioning and executing multi-tenant aware service middleware instances, however, introduces fundamental challenges related to how and where to manage and allocate, respectively, ESB instances in order to maximize the resources utilization while satisfying the end user QoS requirements. Previous works have targeted such investigations by evaluating the performance under different VM-based deployment scenarios [4], [6]. Such approaches focus on provisioning and executing full blown ESBs instances shared among multiple users, where most of the resources required for their execution are spent on overhead, resulting essentially in underutilization of the available resources. Typical ESB solutions are packaged and shipped in different flavors. For example, the Apache ServiceMix[2] ESB solution is built and shipped as a *full*, *JBI*, or *minimal assembly*, with different numbers of features in each option. We aim in this work to leverage the existence of minimal assembly ESB packages towards building and tailoring ESB instances to satisfy specific application requirements.

B. Hypervisor-based vs. Container-based Virtualization

Virtualization is one of the key enablers for Cloud computing. Different kinds of resources such as storage, servers, and networks are virtualized to improve utilization of the underlying physical resources and to keep workload distribution flexible, e.g., by migrating virtual server instances on demand. In terms of server virtualization, hypervisor-based approaches are predominant: a physical server runs a host operation system plus or including the hypervisor, a special piece of software to manage virtual machine instances running on top of the physical machine. Each virtual machine (VM) runs a separate operating system, independent of the operating system on the physical server. This is technically enabled by the hypervisor that provides a complete set of virtualized hardware (CPU, memory, disk, etc.) to each guest operating system. The hypervisor-based approach provides a lot of freedom in building application stacks (including the operating system, middleware, and application components) as well as hosting and moving them between physical machines. VMs can also be packaged as VM images in a portable way, so multiple hypervisor implementations can instantiate such an image to run VMs. Xen[3] and VMware[4] are two prominent examples for hypervisor-based virtualization solutions. Today, such solutions are used by Cloud providers at large scale to offer virtual servers with a huge variety of guest operating systems.

However, hypervisor-based virtualization is not the only way of providing virtualized environments to host applications independently of the underlying physical resources. Whereas hypervisors virtualize all hardware components required to run a completely independent guest operating system, *container virtualization* follows the idea of sharing and virtualizing resources on the level of the operating system. Each isolated and virtualized environment in the operating system is a container instead of a full-blown VM. Technically, the kernel of the operating system is shared among containers, so hardware is not emulated as in the hypervisor-based approach. To ensure isolation of containers, the system processes, filesystem, and network devices are strictly bound to a particular container. Consequently, one container cannot access other containers in an uncontrolled manner. Moreover, there are mechanisms to ensure performance isolation to prevent a single container from offensively consuming all available resources. Container virtualization is not new, but Docker [10], [11] established a strong and open ecosystem, so several Cloud providers seriously support and promote this kind of virtualization, too. As a result, a number of solutions like AWS Beanstalk[5], Google App Engine[6], or Microsoft Azure[7] provide an ecosystem capable of deploying containerized components and applications.

The conceptual differences [12]–[14] between hypervisor-based and container-based virtualization has impact on the performance when using the one or the other. In general, the overhead of container-based virtualization is lower because (i) hardware is not emulated and (ii) there is not a separate guest operating system for each virtualized environment. Consequently, container virtualization tends to perform better. This effect combined with portable packaging and distribution to move containers as proposed by Docker

[1] 4CaaSt EU Project: http://www.4caast.eu/
[2] Apache ServiceMix: http://servicemix.apache.org/
[3] Xen: http://www.xenproject.org
[4] VMware: http://www.vmware.com
[5] AWS Elastic Beanstalk: http://aws.amazon.com/elasticbeanstalk/
[6] Google App Engine: https://appengine.google.com/
[7] Microsoft Azure: http://azure.microsoft.com/

leads to better utilization of available resources. However, as shown in a recent comparison [7], performance benefits may depend on the application profile. Thus, there is a need for minimizing the overhead caused by the components deployed within each container to fully leverage the benefits of container virtualization. Therefore, a major goal of our work is to dynamically build tailored middleware components in the form of containerized ESB instances.

III. Dynamic Tailoring & Deployment

In the following section we present our proposal for an approach to dynamically create *tailored service middleware instances* that can efficiently share and utilize the resources of a VM. This approach aims at building towards a *middleware as a service* solution in the PaaS model supporting and assisting in the constitution, provisioning, and deployment of service-based composite applications. Moreover, it tackles the performance issues and limited flexibility that occur when using a monolithic, general-purpose middleware, before focusing specifically on ESB solutions as a case study for its realization.

A. Tailoring & Deployment Process

The process outlined in Figure 1 shows how tailored service middlewares can be built and deployed. The entry point of the process is to *identify the communication requirements* of a given application. Such a task is typically performed by *Application Developers* responsible for analyzing the application-specific communication requirements and developing new or provisioning existing components to satisfy them. As an example, a Web application may use an HTTP endpoint (typically listening on TCP port 80) to enable the interaction of users through a Web-based user interface and/or a Web API. Moreover, an application typically connects to further back-end systems such as databases, messaging systems, and caching infrastructures. The communication between the application and such back-end systems is covered by additional communication channels based on various kinds of protocols. All these aspects have to be captured as communication requirements to enable the *selection of a service middleware solution* as provided, for instance, by an ESB implementation. Such service middleware solutions typically provide a huge variety of features. The *selection of features* is driven by the communication requirements that have been identified in the previous step. The selection of the middleware solution itself can be revised or refined, e.g., in case it turns out that some required features are not provided by the selected solution. Optionally, *custom communication components can be selected* (or developed, if necessary) to be used as plug-ins for the service middleware solution. This is required in order to support any custom communication requirements of an application (e.g., proprietary communication protocols) that cannot be met by the features of the selected service middleware. In case it turns out that one of the custom components is not compatible with the selected service middleware solution or the selected set of features, the previous selections can be revised and refined correspondingly.

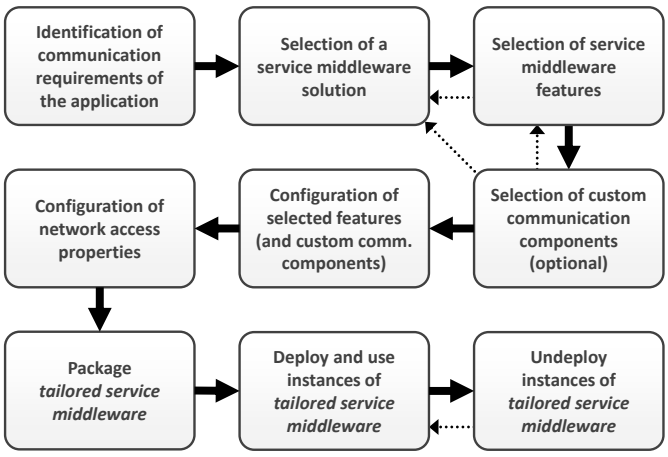

Figure 1: Service middleware tailoring & deployment process

After identifying the communication requirements and selecting the means to resolve these requirements, the selected *features are configured* by defining, e.g., which transport protocols are used and how communication channels are secured. In addition, if custom communication components have been selected previously they are typically configured, too. After configuring the features (and potentially custom components), *network access properties are configured*. This includes which ports are required and how they are provided to the application to communicate with the service middleware. Based on the previously performed configurations and selections, a *tailored service middleware is packaged*, satisfying the application-specific requirements identified in the beginning. Because we are utilizing container virtualization in the scope of this work, containers are used as packaging technique. Such a container can finally be *deployed and used* as tailored service middleware for the given application by creating corresponding service middleware instances. Additional instances are deployed and existing instances are *undeployed* on demand, for instance, depending on the load the application receives.

The presented process to build and deploy tailored service middleware instances is generic enough to be applied to any kind of service middleware. Our work focuses on ESB instances as a prominent representative of service middleware. Therefore, in the following we are looking into how ESB features can be categorized and selected to meet application-specific communication requirements.

B. ESB Features Categorization & Selection

To enable the expression of application-specific communication requirements that can be potentially resolved by ESB features, an initial categorization of such features is required. Figure 2 provides an overview of such an ESB features categorization. Figure 2 is the result of analysis of prominent ESB implementations such as

Table I: Selected ESB features provided by Apache ServiceMix 4

Feature	Category	Description	Functionality
webconsole	Management & Orchestration	Web UI to manage and monitor the ESB	- Install/uninstall features - Change configuration - View logs
camel-core	Management & Orchestration	Run Apache Camel[8] in ServiceMix	- Camel in OSGi integration
camel-xmljson	Message Transformation	Convert message data between formats	- Convert from XML to JSON - Convert from JSON to XML
servicemix-validation	Validation	Schema validation using JAXP 1.3	- Supports XML schema definitions - Supports RelaxNG definitions
camel-sql	Storage	Interact with SQL databases	- Provides JDBC interface
camel-http	Communication	Interact with external systems through HTTP	- Consumes HTTP endpoints - Serves HTTP endpoints
camel-jetty	Communication	Simple, embedded Web server	- Handles HTTP sessions - Supports CORS - Supports data streams
servicemix-http	Communication	Serve HTTP/SOAP endpoints	- Supports SOAP 1.1 and 1.2 - Supports WS-Security

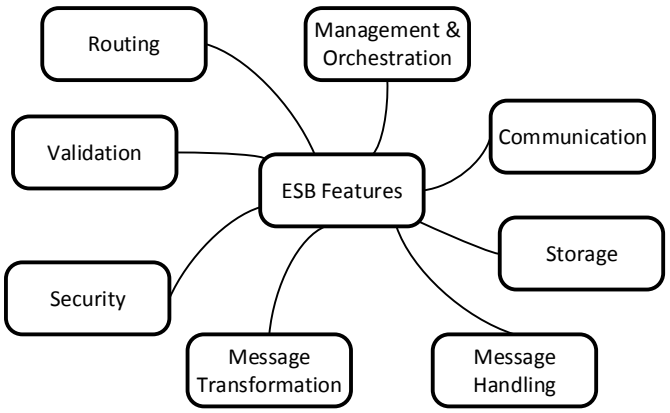

Figure 2: Overview of ESB features

JBoss ESB[9], Apache ServiceMix[10], and Mule ESB[11], with respect to which features they offer, and how they can be grouped. More specifically, the *validation* category comprises all features related to ensuring the correctness of data processed by an ESB. For instance, XML data may be validated against an XML schema definition. An ESB also offers features to define and control *routing* behavior for incoming and outgoing messages. This includes decisions where to send, and how to forward messages, e.g., according to routing rules. Moreover, the *management & orchestration* category covers all features related to managing the ESB and its components (e.g., dynamically changing configuration parameters) as well as orchestrating external and additional functionality provided by scripts, workflows, and other kinds of executables. All features related to the interaction between the ESB and external networking endpoints (not only service endpoints) fall in the *communication* category. These may include features to interact with Web service endpoints such as RESTful APIs as well as to connect to a server using, for example, SSH.

Storage is another class of ESB features, comprising various functionalities provided in order to manage and interact with storage systems such as database management systems. The category *message handling* consists of features to provide tools for dealing with messages in various ways beside routing. For example, the scheduled delivery of messages can be performed by corresponding message handlers. In addition, the *message transformation* category covers features that provide tooling to convert message data between different formats (JSON, XML, etc.) and to re-structure the message payload, e.g., using XSLT. Finally, all security-related features are covered by the *security* category. This includes, for instance, providing secure communication channels (e.g., using HTTPS) as well as signing and encrypting message data.

Table I presents a selected set of features provided by Apache ServiceMix 4, an open-source ESB that we use for evaluating our concepts. Depending on the functionality required by a particular application (e.g., an HTTP/SOAP endpoint), the corresponding feature (e.g., servicemix-http) needs to be selected and configured according to the tailoring process, as discussed in Section III-A.

IV. Evaluation

In the following, we focus on presenting an evaluation and comparative analysis of the benefits and drawbacks introduced by our proposed approach.

A. Methodology & Setup

The experiments driven as part of this work aim at investigating and analyzing three fundamental aspects concerned with the performance of the proposed Cloud-based middleware system: (i) the variation when providing support for tailoring, deploying, and executing minimal

[9]JBoss ESB: http://jbossesb.jboss.org
[10]Apache ServiceMix: http://servicemix.apache.org
[11]Mule ESB: http://www.mulesoft.com/platform/soa/mule-esb-open-source-esb

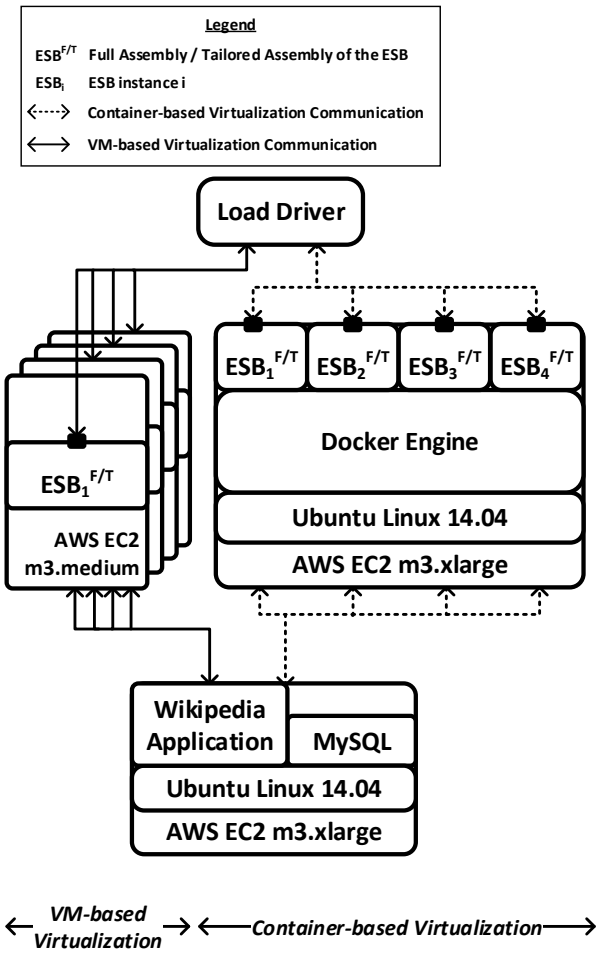

Figure 3: Experimental Setup

ESB assemblies, (ii) the impact of utilizing a container-based virtualization approach and infrastructure as the basis for the provisioning and deployment of the service middleware instances, and (iii) the trade-off between performance and incurred monetary cost for provisioning and executing the different ESB assemblies in the various deployment approaches. For this purpose, the experiments must consider as a first step the impact on the performance when provisioning, deploying and executing multiple service middleware instances in VM- vs. container-based environments. The existence of a wide spectrum of VM configurations introduces a further challenge in the design of the experiments, as these must be selected accordingly to the number of service middleware instances, and in an equivalent manner to the selected deployment approach.

The experimental setup depicted in Fig. 3 comprises different deployment and execution scenarios of the service middleware solution Apache ServiceMix ESB 4.5.3[12]. Apache ServiceMix 4.5.3 is currently shipped in different assembly flavors like *full*, *JBI*, or *minimal* assemblies, among others. For the purposes of this work, we selected the *full* and *minimal* assemblies for the creation of full

[12]Apache ServiceMix 4.5.3:http://servicemix.apache.org/downloads/servicemix-4.5.3.html

Table II: Workload Characteristics

Operation Type	#Requests	Ratio
Read	199925	99.96%
Write	75	0.04%
Image Retrieval	8971	4.49%
Skins Retrieval	66926	33.46%
Wiki Pages Retrieval	106347	53.17%
Others	7634	3.81%

(ESB^F) and tailored (ESB^T) service middleware clusters, respectively (see Fig. 3). The evaluation scenarios comprise a cluster of four ESB instances deployed in the following manner: (i) a VM-based deployment of one ESB instance per VM, and (ii) the container-based deployment of one ESB instance per container, constituting a cluster of four containers deployed in one VM. All scenarios were run in the AWS EC2[13] infrastructure and the provisioned VM were selected to suit the following configuration equivalently:

- four AWS EC2 *m3.medium* VM instances with 1 vCPU and 3.75GB RAM per VM, running Ubuntu 14.04 LTS, and with a usage cost of 0.077U$/h, and
- one AWS EC2 *m3.xlarge* VM instance with 4 vCPUs and 15GB RAM, running Ubuntu 14.04 LTS and Docker[14] version 1.4.1, and with a usage cost of 0.308U$/h.

These configurations of EC2 were equivalently selected, as one m3.xlarge image has roughly four times the computing capacity of one m3.medium. As such, the comparison between the container-based instances with the VM-based ones is fair in terms of their available computational resources.

For all the previously depicted scenarios, a back-end MediaWiki[15] service was deployed in an AWS EC2 *m3.xlarge*, hosting the Wikipedia 2008 content [15]. The communication with the MediaWiki application is wired through the deployment of Apache Camel Jetty[16] endpoints deployed in each ESB instance. The deployment of such endpoints requires the previous installation of the Apache Camel[17] and Apache Camel Jetty bundles in the ESB. For the ESB^F assembly scenarios, such packages already contains such bundles. However, in the ESB^T assembly scenarios, such bundles are dynamically installed during the ESB tailoring process. In terms of workload, we randomly extracted and generated a realistic workload of 200K requests from the Wikipedia access traces, which is then distributed among (concurrent) users and ESB instances. The load driver depicted in Fig. 3 acts as the services' consumer emulator and generates a set of 100 concurrent users per ESB instance in an uniformly distributed manner with an interval creation of 50ms. In all scenarios, the

[13]AWS EC2: http://aws.amazon.com/ec2/

[14]Docker: https://www.docker.com/

[15]MediaWiki: https://www.mediawiki.org/wiki/MediaWiki

[16]Apache Camel - Jetty Endpoint: http://camel.apache.org/jetty.html

[17]Apache Camel: http://camel.apache.org/

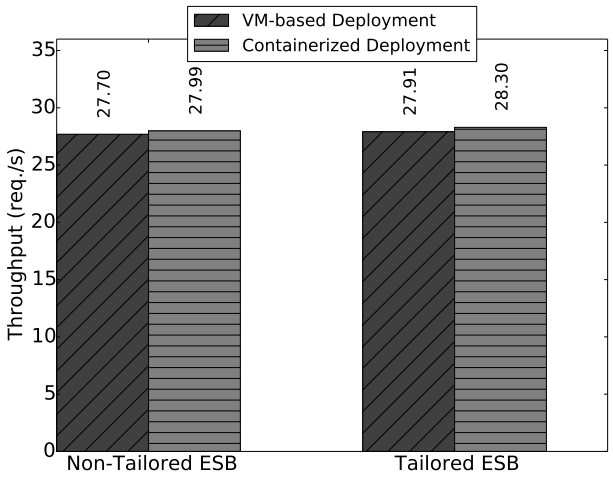

Figure 4: Average Throughput per Scenario

load driver is deployed on-premise in order to emulate the network latency introduced in the requests. The load driver profile, generated workload sample, and Camel Jetty endpoint configurations are publicly available[18]. The workload characteristics are depicted in Table II. As it can be observed, most of the Wikipedia accesses consist of read operations which retrieve different kinds of data, e.g. images, pages, style skins, etc. The workload analysis, however, does not yet measure popularity aspects, e.g. which are the most accessed Wikipedia pages. Moreover, due to the *read intensive* nature of the MediaWiki (Wikipedia) application, such workload only comprises a minimum set of write operations. Sampling and generating workload focusing on increasing the number write operations as well as incorporating data popularity aspects is part of ongoing work and will be included when required in future investigations.

Our experimental evaluation offers a two dimensional analysis. On the one dimension, we aim at analyzing the impact on the performance in terms of the amount of users that the system is able to cope with. For this purpose, we measure the throughput in terms of requests per second for the different deployment scenarios. The second dimension consists of analyzing the monetary expenses incurred when hosting the ESB instances in different VM instances types and deployment scenarios. The monetary costs are calculated by first calculating the elapsed time of the experiments, and by subsequently using the hourly usage cost established by Amazon AWS.

B. Results

The following results can be observed with respect to the performance variation of the system (see Fig. 4):

- the overall system's capacity is increased by an average of 1% when building and running tailored ESB instances.

[18]Tailored ESB MediaWiki Evaluation: https://github.com/sgomezsaez/Tailored_ESB_MediaWiki_Eval

Table III: Monetary Cost Evaluation (in AWS EC2 February 2015 Usage Prices)

Scenario	Elapsed Time (h)	Cost (US$)
VM Depl. / ESB^F	1.976	0.608
Containerized Depl. / ESB^F	2.041	0.628
VM Depl. / ESB^T	1.942	0.598
Containerized Depl. / ESB^T	1.953	0.601

- Such a performance improvement breaks down into an increase of 0.75% and 1.1% for the VM-based and container-based deployment scenarios, respectively.
- Focusing on the performance variation introduced by the usage of container-based technologies, the usage of container-based virtualization provides a performance improvement of 1% and 1.37% for the non-tailored and tailored ESB scenarios, respectively.

With respect to the monetary costs incurred by running the previous experiments in an off-premise public Cloud infrastructure, Table III summarizes the most relevant findings:

- Building and executing tailored ESB instances reduces the average monetary costs by approximately 3%.
- More specifically, the deployment of tailored ESBs in VMs reduces the incurred monetary costs by approximately 1.6%. However, when utilizing a container-based technology, such a cost increases by approximately 4.3%.
- In line with the previous observation, when provisioning and deploying ESB instances in VMs, the usage of container-based virtualization technologies increases the operational cost by approximately 10%.

C. Discussion

The experimental results of the previous section enable us to perform a deeper bi-dimensional analysis, taking into consideration the impact on the performance and monetary costs when applying the approach presented in this work. The generated Wikipedia-based workload used as part of this experiments describes a read intensive application, which is mostly accessed to retrieve Wikipedia pages and skin styles.

The performance variation results presented in Section IV-B can drive the following conclusions:

- Building, provisioning and executing tailored ESB instances as part of the fundamental PaaS service middleware towards satisfying the application's requirements has a beneficial impact in the performance. Such improvement is due to the usage of ESB minimal assembly packages which can be dynamically tailored and provisioned by our framework.
- The usage of container-based virtualization technologies as part of the underlying infrastructure has indeed improved in all scenarios the performance of service middleware infrastructre.

- The derived monetary costs is indeed reduced when provisioning and executing tailored ESB instances. However, the costs of running the ESB instances increase when utilizing a container-based underlying infrastructure.

The approach presented in this work showed that building and provisioning ESB instances in a tailored manner as part of a custom service middleware for deploying PaaS-based applications has a beneficial impact on the system's performance. Going a step further and evaluating the usage of container-based technologies demonstrated a further gain in the performance of the service middleware execution. Moreover, the incurred costs due to applying the proposed tailoring to minimal ESB instances can be significantly reduced when using a VM-based underlying infrastructure, but can be increased when using container-based underlying technologies. The provisioning time and monetary cost were not included in this experiments. Future experiments are planned to tackle such life cycle step by including the provisioning and management cost associated with the usage of the proposed virtualization technologies as the underlying infrastructure of the service middleware. For instance, we believe that there can exist a significant positive impact on the provisioning time when combining tailoring and container-based deployment mechanisms. The proof for such hypothesis is, however, part of ongoing empirical investigations and part of future reports.

V. Related Work

In the following we present our investigations on existing approaches in the fields of service middleware and the tailoring of these components, the usage and evaluation of container virtualization technologies, and the efforts in the field of service middleware towards adapting such components to the Cloud environment.

Service middleware solutions have been widely used as key integration building blocks among components or devices. The specialization of service middleware components as an approach for optimizing middleware components for domain specific environments has been proposed already in [16], [17]. For example, [16] proposes a process-based approach towards building specialized middleware components for real-time systems. In the scope of SOA-based systems and enterprise application integration, there is currently no approach for building tailored middleware components on an on-demand basis to satisfy application's functional and non-functional requirements. Focusing on the elasticity features of message-based systems, in [18] a generic and elastic message queue architecture adapted to Cloud environments and capable of providing elasticity features is proposed. In this work, message queuing capabilities are natively supported in the ESB, and elasticity features are planned to be supported by means of leveraging the distribution of containerized ESB instances in a containers cluster, such as CoreOS[19].

In the scope of PaaS platforms, there have been several efforts in the direction of provisioning and executing virtualized middleware components as fundamental components for enabling the deployment of custom applications, e.g. taking into account performance isolation aspects [19]. For example, in [4] and [6] multi-tenancy techniques have been applied in ESB solutions towards enabling the sharing of the underlying middleware resources. Such components can be built and deployed as part of pre-configured VM instances on an on-demand basis following, for example, middleware-oriented deployment approaches [20]. However, the complexity and management overhead can be significantly reduced when leveraging the usage of lightweight pre-built and reusable container images for tailoring the different ESB components. Moreover, the advantages brought by container-based virtualization technologies enable the provisioning and execution of multiple containers in the same VM with small adaptations overhead, e.g. port forwarding, memory allocation, etc.

Container virtualization techniques have emerged in the last years as an alternative to hypervisor-based virtualization [13]. The reduction in resources usage together with the rapid provisioning features of lightweight containers have contributed to both application developers and Cloud providers to adopt and provide offerings based on such technology [14]. Several works, such as [7] and [12], have investigated the impact of virtualizing the OS rather than the hardware. Results showed that containers result in an equal or better performance than virtual machine-based approaches in most of the cases. Our work leverages such virtualization technique and combines it with a tailoring process towards provisioning custom ESB instances on an on-demand basis. The performance evaluation driven as part of this work also showed an improvement when running the ESB instances in container-based virtualized environments. The analysis of provisioning and management tasks performance overhead is part of future work.

VI. Conclusions and Future Work

Service middleware is an essential component for the realization of composite service-based applications, and a fundamental building block for PaaS solutions that support such applications. Cloudifying and offering (service) middleware as a service as part of PaaS solutions has already been demonstrated to be an efficient solution, especially when considering the resource sharing capabilities among service providers through multi-tenancy. In this work we reap the benefits of the ongoing developments in container-based virtualization techniques and technologies to provide more fine-grained control over the configuration, deployment and execution of a service middleware solution in the form of an Enterprise Service Bus (ESB).

More specifically, in the previous sections we proposed a dynamic tailoring and deployment process of service middleware components geared towards the cloud environment. We focused the discussion on how this process can be applied in practice on the ESB technology and showed how minimal flavors of an ESB can be rapidly packaged and deployed using Docker. We evaluated our proposal using the MediaWiki application and a load generated from Wikipedia traces, with the application using different configuration and deployment scenarios of the same ESB technology. The results show a small but

[19]CoreOS: https://coreos.com/

promising overall improvement due to the combination of container-based virtualization with application-specific ESB configuration, with a significantly smaller amount of VMs to be managed. Future work focuses on investigating and developing the potentially required tool chain to support the previously presented process. With respect to the evaluation, we plan to incorporate the provisioning and management costs of the different deployment scenarios. Furthermore, we are already working towards allowing the cluster-based management of containers in an automated manner for further performance improvements with less effort on resource provisioning on behalf of application developers.

ACKNOWLEDGEMENTS

This work is funded by the FP7 EU-FET project 600792 ALLOW Ensembles.

References

[1] M. Hahn, S. Gómez Sáez, V. Andrikopoulos, D. Karastoyanova, and F. Leymann, "Development and Evaluation of a Multi-tenant Service Middleware PaaS Solution," in *Proceedings of the 7th International Conference on Utility and Cloud Computing (UCC 2014)*. IEEE Computer Society, December 2014, pp. 278–287.

[2] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: a research roadmap," *International Journal of Cooperative Information Systems*, vol. 17, no. 02, pp. 223–255, 2008.

[3] M. Pathirage, S. Perera, I. Kumara, and S. Weerawarana, "A Multi-tenant Architecture for Business Process Executions," in *2011 IEEE International Conference on Web Services (ICWS 2011)*, 2011, pp. 121–128.

[4] A. Azeez, S. Perera, D. Gamage, R. Linton, P. Siriwardana, D. Leelaratne, S. Weerawarana, and P. Fremantle, "Multi-tenant soa middleware for cloud computing," in *2010 IEEE 3rd International Conference on Cloud Computing (CLOUD 2010)*. IEEE, 2010, pp. 458–465.

[5] S. Garcia-Gomez, M. Jiménez-Ganán, Y. Taher, C. Momm, F. Junker, J. Biro, A. Menychtas, V. Andrikopoulos, and S. Strauch, "Challenges for the comprehensive management of cloud services in a paas framework," *Scalable Computing: Practice and Experience*, vol. 13, no. 3, 2012.

[6] S. Strauch, V. Andrikopoulos, S. Gómez Sáez, and F. Leymann, "ESB^{MT}: A Multi-tenant Aware Enterprise Service Bus," *International Journal of Next-Generation Computing*, vol. 4, no. 3, pp. 230–249, November 2013.

[7] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," *Technology*, vol. 28, p. 32, 2014.

[8] D. Chappell, *Enterprise service bus*. O'Reilly Media, Inc., 2004.

[9] Y. Wei and M. Brian Blake, "Service-oriented computing and cloud computing: Challenges and opportunities," *IEEE Internet Computing*, vol. 14, no. 6, pp. 72–75, 2010.

[10] J. Fink, "Docker: a Software as a Service, Operating System-Level Virtualization Framework," *Code4Lib Journal*, vol. 25, 2014.

[11] J. Turnbull, *The Docker Book*. James Turnbull, 2014.

[12] M. J. Scheepers, "Virtualization and Containerization of Application Infrastructure: A Comparison," 2014.

[13] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, 2007, pp. 275–287.

[14] S. J. Vaughan-Nichols, "New Approach to Virtualization is a Lightweight," *Computer*, vol. 39, no. 11, pp. 12–14, 2006.

[15] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Elsevier Computer Networks*, vol. 53, no. 11, pp. 1830–1845, July 2009.

[16] A. Dabholkar and A. Gokhale, "A generative middleware specialization process for distributed real-time and embedded systems," in *14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC 2011)*. IEEE, 2011, pp. 197–204.

[17] V. Subramonian, G. Xing, C. Gill, C. Lu, and R. Cytron, "Middleware specialization for memory-constrained networked embedded systems," in *Proceedings of 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004)*. IEEE, 2004, pp. 306–313.

[18] N.-L. Tran, S. Skhiri, and E. Zimányi, "Eqs: An elastic and scalable message queue for the cloud," in *IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom 2011)*. IEEE, 2011, pp. 391–398.

[19] R. Krebs, M. Loesch, and S. Kounev, "Platform-as-a-service architecture for performance isolated multi-tenant applications," in *IEEE 7th International Conference on Cloud Computing (CLOUD 2014)*. IEEE, 2014, pp. 914–921.

[20] J. Wettinger, V. Andrikopoulos, S. Strauch, and F. Leymann, "Characterizing and Evaluating Different Deployment Approaches for Cloud Applications," in *Proceedings of 2nd IEEE International Conference on Cloud Computing Engineering (IC2E 2014)*. IEEE Computer Society, 2014.

All links were last followed on April 24, 2015.