UiO : Department of Informatics
University of Oslo

Joint Pre-Proceedings of the Workshops Associated with ESOCC 2017

Kyriakos Kritikos, Zoltán Ádám Mann, Claus Pahl, Volker Stolz (Eds.)

Research report 471, 27 September, 2017

ISBN 978-82-7368-436-3
ISSN 0806-3036

Contents

Preface

This volume contains the papers presented at the workshops associated with the 6th European Conference on Service-Oriented and Cloud Computing, ESOCC 2017. The workshops were held in Oslo, Norway, on 27th September 2017. The workshops covered specific topics in service-oriented and cloud computing-related domains:

- 3rd Intl. Workshop on Cloud Adoption and Migration (CloudWays 2017)
- 1st Intl. Workshop on Business Process Management in the Cloud (BPM@Cloud 2017)

All papers presented at the workshops were selected through a rigorous review process, in which each submission was reviewed by at least three members of the workshops' program committees.

We as the workshop chairs would like to thank all authors for their submissions, and the reviewers for their work.

For the organizers,
Volker Stolz (Høgskulen på Vestlandet / Universitetet i Oslo)

CloudWays Workshop Papers

Engineering Cloud-based Applications: Towards an Application Lifecycle

Vasilios Andrikopoulos

Johann Bernoulli Institute for Mathematics and Computer Science
University of Groningen, the Netherlands
v.andrikopoulos@rug.nl

Abstract. The adoption of cloud computing by organizations of all sizes and types in the recent years has created multiple opportunities and challenges for the development of software to be used in this environment. In this work-in-progress paper, the focus is on the latter part, providing a view on the main research challenges that are created for software engineering by cloud computing. These challenges stem from the inherent characteristics of the cloud computing paradigm, and require a multi-dimensional approach to address them. Towards this goal, a lifecycle for cloud-based applications is presented, as the foundation for further work in the area.

Keywords: cloud computing, software engineering, cloud-based applications, software lifecycle

1 Introduction

The adoption of cloud computing has increased dramatically since the introduction of the term only roughly ten years ago — despite the fact that the technologies underpinning the paradigm have been around for a while longer. It is not an exaggeration to claim that in one way or another cloud computing offerings and associated technologies are currently being used by the majority of software-intensive enterprises. A report of the Thoughtworks Technology Advisory Board back in May 2015[1], for example, claims that *"Organizations have accepted that "cloud" is the de-facto platform of the future, and the benefits and flexibility it brings have ushered in a renaissance in software architecture."* From the thousand professionals from across sectors participating to RightScale's annual survey in early 2017 [31], 95% are reporting that the organization they belong to is already using or experimenting with the use of cloud computing.

Under the umbrella of the same term, however, there are multiple service delivery and deployment models on offer, succinctly summarized by NIST's widely accepted definition of cloud computing [24]. The availability of these options,

[1] Thoughtworks Tech Radar, May 2015: https://assets.thoughtworks.com/assets/technology-radar-may-2015-en.pdf

in conjunction with the plethora of offerings by cloud providers like Amazon Web Services (AWS), Microsoft Azure (MSA), and Google Compute Platform (GCP), and software solutions for the deployment of private clouds such as the ones from VMWware and OpenStack, create both opportunities and challenges for software developers [4]. Even the process of selecting an appropriate provider to run software on is an open research subject, with many of the issues identified in [34] (e.g. lack of standardization in the QoS descriptions and lack of long term performance prediction) still valid today. As such, there are still many issues that need to be resolved with respect to how cloud computing is to be used for software development.

At the same time, in the recent years the discourse on the best practices and principles of software development, at least in the industry, has been affected significantly by the introduction of two movements that have a co-dependence relation with cloud computing. The first one is the use of DevOps technologies and processes in order to bridge the gap between development and operations of software [8] in order to streamline software delivery and maintenance. The adoption of Continuous Delivery/Integration (CD/CI) techniques with frameworks like Jenkins[2] used together with deployment automation tools like Chef[3] or Ansible[4] shortens the development cycle dramatically and produces synergy with agile-oriented software development practices. Allowing for the management of multiple software stacks running in partially isolated containers inside one operating system as made popular by Docker[5], is the logical extension of this approach: each architectural component is developed, deployed, managed, and updated in its own software stack, and therefore it can follow a life cycle that is loosely coupled with the overall system evolution. This principle is made even more prominent in the second of the movements relevant to the discussion, i.e. microservices [27]. While there is an ongoing discussion in the academic community related to the actual innovation of microservices in comparison to Software-Oriented Architecture, it is important to notice how the notion of microservices have integrated into practice the use of design patterns, that so far have been mostly adopted at a much lower level (e.g. the Gang of Four book). Entries on microservices in Martin Fowler's blog[6], a popular grey literature source for practitioners and researchers provides many instances of this phenomenon.

In summary, therefore, the virtualization of resources and their offering as services, in conjunction with the DevOps movement, the containerization of software stacks, and the use of microservices, have evolved the way that software is developed, deployed, and managed over time. The key message of this paper is that *engineering software, and in particular software architecture, should similarly evolve*. For the purposes of scoping, the discussion is focused on how

[2] Jenkinks: `https://jenkins.io/`
[3] Chef: `https://www.chef.io/`
[4] Ansible: `https://www.ansible.com/`
[5] Docker: `https://www.docker.com/`
[6] For example: Microservices, by James Lewis and Martin Fowler (March 2014): `https://martinfowler.com/articles/microservices.html`

software engineering can change to incorporate cloud-related concepts by means of introducing a *cloud-based application lifecycle*. In the absence of a widely accepted definition of what constitutes one, and following the definition of service-based applications discussed in [3], this paper uses a working definition of *Cloud-based applications (CBAs)* as *applications that rely on one or more cloud services in order to be able to deliver their functionality to their users*. CBAs therefore include both cloud-enabled through migration [2] and cloud-native applications [21].

The rest of this paper is structured as follows: Section 2 identifies and presents the most relevant challenges to cloud-based application engineering (definitely not an exhaustive list). Section 3 transforms these challenges into a set of requirements on lifecycle methodologies in this context. Consequently, Section 4 discusses a CBA lifecycle that aims to address these requirements as the basis for future research. Finally, Section 5 compares the proposed lifecycle with related approaches, and Section 6 concludes with a short summary and future work.

2 Major challenges

Following the NIST definition [24], cloud computing has the following essential characteristics: (i) *On-demand self-service:* appropriate interfaces are offered to consumers to access resources (computational, storage, network, etc.) in an automated manner. (ii) *Broad network access:* resources are accessed over the network by heterogeneous clients. (iii) *Resource pooling:* service providers are enforcing a multi-tenant model of sharing the offered resources. (iv) *Rapid elasticity:* the volume of accessed resources can be adjusted dynamically, by any quantity and at any time. (v) *Measured service:* a metering mechanism is used to ensure appropriate billing for the used resources in predefined periods of time.

The combinations of these characteristics has severe implications for the software that is being developed in this environment. In the following we identify four major challenges that arise due to these characteristics.

2.1 *aaS software model

The first major challenge stems from the fact that resources are offered in the *Everything as a Service (*aaS)* model, usually affiliated with the categorization of delivery models into Infrastructure (IaaS), Platform (PaaS), and Software as a Service (SaaS), also covered by the NIST definition. The *aaS model is a natural outcome of the first two characteristics (i.e. *on-demand self-service*, and *broad network access*) and in many cases manifests as sets of RESTful APIs that are exposing cloud resources through relatively simple CRUD operations. While there has been lots of work on the subject of engineering service-based applications in the last 15 or so years, see for example [3], the very nature of service orientation still poses particular difficulties when used as the model for accessing resources. These can be attributed to the following:

Information hiding behind interfaces: exposing only the amount of information that is absolutely necessary for clients to use a service is one of the fundamental premises of service orientation [12]. However, this means that software developers have to refer to documentation and help desks in order to understand the boundary conditions and assumptions of consuming each resource.

Lack of control and observability over resource implementation: while the on-demand self-service characteristic prescribes a degree of control over the consumed resources by removing the need for administration on the part of the provider, this control is in practice limited to the operations defined in the service API, that for all practical purposes act as black box endpoints.

Distributed and heterogeneous environment: distribution transparency [33] is an essential feature of offered services, creating an impression of homogeneity and opaqueness to software developers. Nevertheless, the operating environment is fundamentally distributed, irrespective of the type of software developed on it (distributed or not).

Evolution driven by 3rd parties: as with many other API publishers in the past, cloud providers reserve the right to change their supported APIs at any point in time — and they do so for various reasons. As such, therefore, the evolution of software developed on these solutions is at least partially driven by the cloud providers and beyond software developer control.

Lastly, it can also be argued that while it is indeed possible to build all kinds of systems on top of cloud resources, it is consistent with the model that it is offered to design and implement them as services themselves. Doing so, however, imposes its own challenges, as evidenced by the continuous research output of the SOA community in the last two decades. The most thorny issue to deal with is probably the design of the system as services itself; indicative of the complexity of this issue is the fact that service design is identified as a major research question in both the SOA research roadmap [30], and its revision ten years later [10]. Further work towards this direction is therefore required.

2.2 Multi-tenancy of resources

One of the most difficult challenges to address, especially for performance-sensitive systems, is that of the shared nature of cloud resources due to its *resource pooling* characteristic. In a sense it is exactly this characteristic which makes rapid elasticity possible, while allowing for resource prices to be offered at very low levels, as also discussed by the next challenge. In essence, multiple tenants sharing the same infrastructure enable economies of scale for service providers and allow for higher utilization on the provider side through smart scheduling of large volumes of work load.

This sharing of resources, however, leads at the same time to performance variability that is external to the application itself, and as such outside of the

control of the system developer. The inherent variance of cloud offerings has been documented in a series of publications: in [22], for example, large deviations are reported for similar in specification offerings across different providers, while significant variance can be observed in the same provider and offering within the same day and week [32] (and even more so across different availability zones), or even over the period of a year for the same offering [17]. Benchmarking cloud applications is faced with multiple challenges, see for example [9], and [14], and is not readily available as a tool for software developers to incorporate in their toolset. Cloud monitoring [1] is therefore the most common way to check and potentially address detrimental performance variation of the consumed resources.

2.3 Utility computing

One of the main reasons for the wide adoption of cloud computing is the transfer of costs from the capital to operating expenses through its "pay as you go" model [6], enabled by its *measured service* characteristic. In this sense, cloud computing can be seen as an implementation of the utility computing vision [39]. Access to computational resources in this context is enabled in a utility-oriented model, and results in the illusion of virtually infinite resources being available — assuming of course a sufficiently large budget [6]. At the same time, the use of economies of scale on behalf of the service providers, and the environment of intense competition for a very lucrative market, result into continuously decreasing prices for the offered resources. This creates the dynamics of a "race to zero" phenomenon, especially in storage offerings[7]. Even if the provider prices are not lower in comparison with operating one's own data center as e.g. in the (already outdated) analysis of [37], there are boundary conditions that still make the use of cloud solutions favorable to the alternative [38]. The key is in the *rapid elasticity* characteristic which allows for quick scaling to cope with dynamic demand, resulting in compensation of potentially incurred losses throughout relatively stable demand periods by means of serving requests that would otherwise be over capacity and therefore resulting in loss of revenue.

Nevertheless, cheap is not the same as free of charge, and costs for successful cloud-based companies might run so high that result in their profit margin shrinking to the point of necessitating the migration to their own data centers instead, as documented by the case of Dropbox[8], a company that was famous for running all their infrastructure on Amazon Web Services until that point. Rightscale's 2017 State of the Cloud survey [31] reports two stark findings that are relevant to this discussion: first, mature adopters of the technology are more concerned with cost management in comparison to beginners to it; second, only a minority of companies actually take measures to minimize unnecessary costs (e.g. VMs unnecessarily being active). Some notion of costs control is therefore clearly necessary.

[7] See for example: `http://www.computerweekly.com/microscope/news/4500271376/Whatever-the-cost-may-be-Cloud-price-war-continues`

[8] See `https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/`

2.4 Distributed topology

There is no escaping the fact that systems developed in the cloud environment are essentially distributed, and they need to be designed, implemented, and operated as such [11]. Distribution in this case is both spatial and logical, but distribution transparency [33] is partially violated when e.g. availability zones are used for the deployment of applications. On top of this, there are multiple offerings by service providers that can be used as alternatives to application components [4] taking advantage of the *on-demand self service* characteristic. For example, Database as a Services (DBaaS) offerings can replace completely the data layer of an application, providing native scaling mechanisms to cope with increasing demand. An illustration of the range of possibilities available to software architects is the case of Netflix, which combines AWS EC2, S3, EBS and other offerings to run in a cloud-only environment[9].

Adding to the size of the design space is the capability to use containers as the means for enabling portability of application components and work loads across cloud providers, essentially expanding on the characteristic of *resource pooling*. In conjunction with a cloud orchestration layer, containers allow for a series of benefits like reduced (infrastructure) complexity, automation of portability, better governance and security management, transparent geographical domain-aware distribution, and the ability to automate services that offer policy-based optimization and self-configuration [23]. As a result, there are many possible system configuration options that are optimal under different dimensions [4], e.g. cost versus performance, creating exceptional challenges to software architects in identifying the best solution for their needs.

3 Requirements on the Solution Space

From the discussion above it becomes quickly obvious that addressing these challenges is a multi-faceted undertaking, and that their nature requires them to be considered throughout the lifecycle of software systems operating in the Cloud. The following constraints are, as a result, imposed on possible solutions for engineering *cloud-based applications (CBAs)*:

1. Irrespective of the purpose and type of software under consideration, cloud-based application development should *understand and incorporate service-orientation concepts*. In practice, this means that resources are accessed through programmatic interfaces, which in turn favors the Infrastructure as a Code approach [16] that homogenizes the way that the software itself and its supporting infrastructure is managed. Across similar lines, cloud service composition [18], which deals with the selection and aggregation of cloud services in order to support software, needs to be considered on equal grounds with (software) service composition [30] which delivers functionality by combining independent services.

[9] Netflix Global Cloud Architecture: `https://www.slideshare.net/adrianco/netflix-global-cloud`, slide 26

2. System design should incorporate the notion of *dynamic topology*. Topology here refers to the software and infrastructure stack required to operate the software artifacts under consideration, including e.g. the middleware associated with them. The system topology is prone to change over time due to changes *a)* in the system architecture, and *b)* refactoring of the infrastructure that supports the system. This might also include incorporation of new services by the same cloud provider or migration to another provider and/or deployment model. In this sense, the resolution of system architecture into concrete deployment models should rely on the generation of viable topologies through e.g. graph transformations, as per [4], instead of explicit modeling of alternatives. This is a consequence of the very large amount of available alternatives during design when considering all the different configurations available for each service type.
3. *Self-* characteristics* (e.g. self-management, -adaptation, -healing, -configuration, etc.) are necessary to deal with the multi-tenancy induced performance variability and its impact to the QoS of cloud-based applications. The introduction of a MAPE-K (Monitor, Analyse, Plan, and Execute over a Knowledge base) feedback loop [20] is a necessary and very common solution at this level as the means to implement control [29], but the difficulty is in evaluating the impact of individual cloud services, e.g. a DBaaS solution, to overall performance. Furthermore, the connection between run-time observations and design-time predictions is not sufficiently covered by the state of the art [15], and further work is necessary towards this direction. End-to-end performance measurement is potentially more important — alternative viable topologies have to be evaluated after all against their actual effectiveness in generating revenue — and in case of software delivered as services relatively easy to implement.
4. An *awareness of consumed resources* on self-management level during both development and operation of the system is essential. Cost models that cover the various deployment models, e.g. an extension of the model for hybrid clouds discussed in [19], should be used for this purpose. However, such analysis cannot be only performed offline. Instead, design- and run-time cost analysis should complement each other [25], resulting in cost models that are dynamically updated by actual billing data received from the cloud provider.

In the following we introduce a lifecycle model for cloud-based applications that incorporates the constraints discussed above as the means for defining in the future a holistic framework for engineering cloud-based applications. For this purpose the lifecycle model of service-based applications as discussed in [3] is used as the inspiration for this work.

4 Cloud-based Applications Lifecyle

4.1 The Phases of the Lifecycle

Figure 1 illustrates the proposed lifecycle of CBAs. Before proceeding with explaining the stages of the lifecycle, it needs to be pointed out that for the

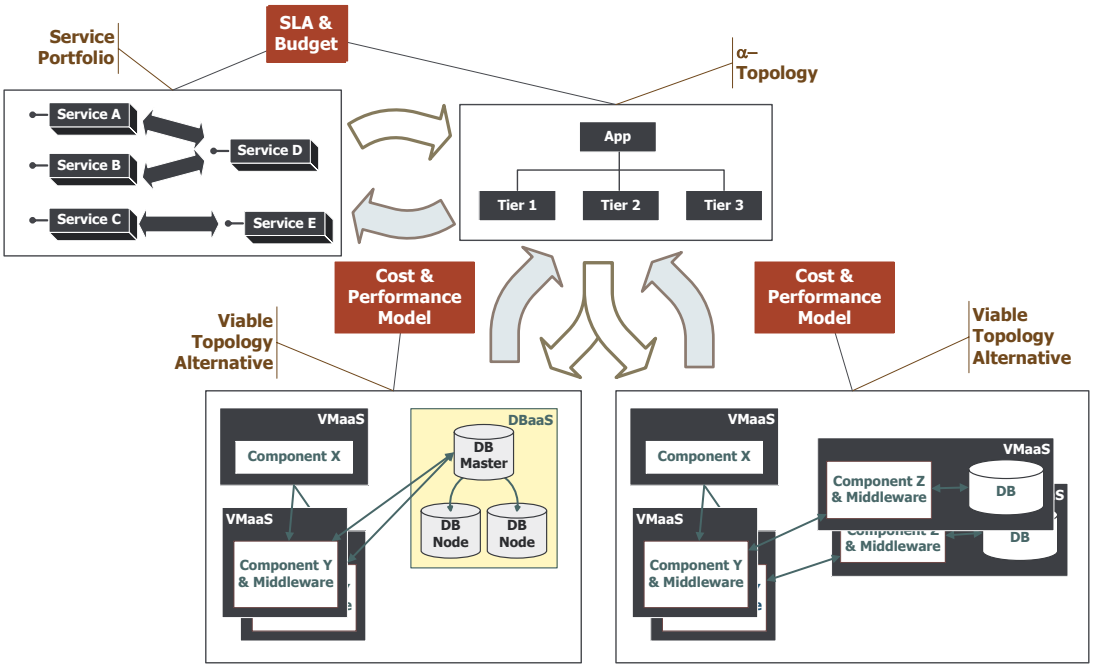


Fig. 1. The Lifecycle of Cloud-based Applications

purposes of this discussion, there is no clear design- and run-time (or development and operation, respectively) distinction, but more of a spectrum of activities spanning between them. The everything as a service and dynamic topology challenges affect more the one end (design), while performance variability and cost awareness more the other (run time). However it is impractical to attempt to assign them to specific stages of the lifecycle. The proposed CBA lifecycle (as shown in the figure) reflects this by intentionally not identifying when the transitions between stages are to take place, but only the transition relations between them. In this respect, the presented lifecycle is in accordance with the main principles of the DevOps movement [8] which unifies the different stages of software lifecycle.

Looking now at the figure, and starting from its top left part, the highest stage of the lifecycle consists of the *service portfolio* for the application, i.e. the collection of services that implement the functionalities offered by the application. Such services could be composed out of other services, belonging either to the same portfolio, or being external to it, as per the well established SOA practice [30]. Following the same principles, the service portfolio is the outcome of a *service identification* phase that connects higher level requirements and business operations into functionalities to be exposed by the application as services. In terms of how these services are mapped into software components and its supporting middleware, a *decomposition* into structural tiers can be applied using one of the methodologies discussed in [3]. In principle, non-application specific software components should be excluded from this process, resulting into a system architecture expressed as a set of *α-topologies* [4] (top center of Fig. 1).

The intentional exclusion of the underlying software stack from this stage (except where it cannot be avoided as e.g. in the case of customized middleware that needs to be rolled out together with the application) allows for flexibility in the transition to the next stage, that of *viable topology alternatives*, each one of which represent the whole software stack and its relation to the application components (bottom half of Fig. 1). Viable topology models encapsulate the various types of cloud services (e.g. VM or DB as a Service in the figure) that are part of the infrastructure supporting the software stack of the application.

As discussed above, and due to the numerous cloud service offerings currently available, a large number of viable topology alternatives potentially exist for each application. Selecting between them can be, and usually is interpreted as an optimization problem for which there are many techniques available (see [4] for further discussion). However, an alternative approach would be to look into this situation as an exploratory search problem instead. In this context, identifying a unique optimal solution in advance would be of not such interest as in transitioning between different alternative solutions in order to identify the optimal for the current conditions. For this purpose, the overall consumer utility and revenue generated by the viable topology currently used needs to be evaluated by comparing the continuously updated *cost and performance models* for each viable topology against the *Service Level Agreements (SLAs) and budget* associated with the service portfolio by the application owners. This approach requires, of course, that costs for the transition between viable topology models are negligible in comparison with the overall revenue generated by the application. Using a microservices-based approach for the decomposition of the service portfolio into isolated sub-systems before generating viable topologies would actually minimize such costs, since the finer granularity of each system tier would mean less components to consider (and potentially migrate) on the topology level. Alternatively, if this transition is deemed too costly and/or if the search space of viable topology alternatives has been exhausted then it is meaningful to revert to the previous stages of the lifecycle and either decompose the service portfolio as different α-topologies, or even refactor the service portfolio itself, repeating the cycle as necessary, as discussed in the following.

In order to add the necessary self-* mechanisms that regulate decision making during system operation a distributed MAPE-K model can be used [20], as discussed in Section 3. Considering the lifecycle of Fig. 1, however, it becomes clear that a hierarchical organization of controllers is better fitting. On the bottom level, it is possible to view the architectural components of each viable topology as its own autonomic element. However, all such elements need to coordinate with a controller on the level of the viable topology which is responsible for changes inside it. Another level of controllers is necessary to be added at the level of α-topologies when more than one viable topologies are active for a given decomposition. A similar process is repeated to the level of the service portfolio, and is used in order to trigger the transitions between stages of the lifecycle. Since the degree of automation that is feasible and available can vary among these transitions, it might become necessary to involve architects and system designers

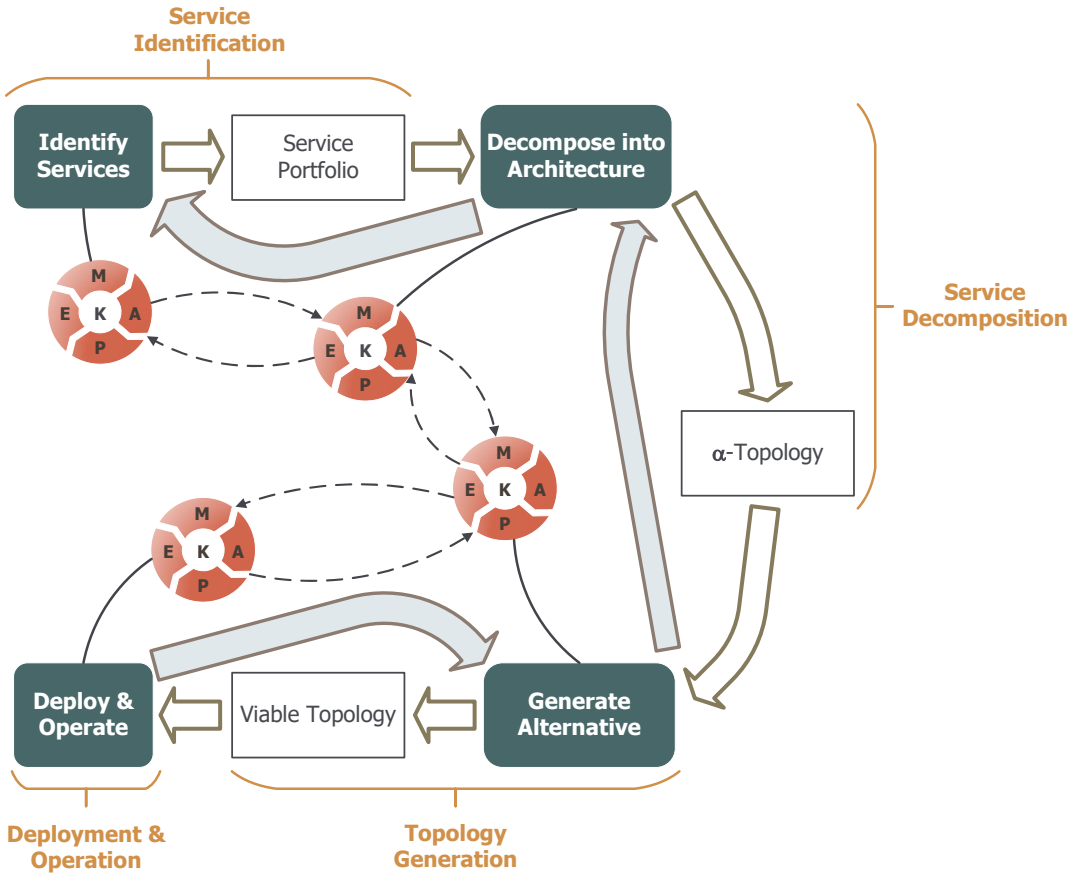


Fig. 2. The Phases of the CBA Lifecycle, with Activities Implemented as MAPE-K Loops and Information Flowing Between them

for this purpose. As such, design activities could be triggered by operations, as much as operational models could be derived during development.

Figure 2 summarizes and illustrates this discussion by identifying the concrete *phases* of the proposed lifecycle (Service Identification, Service Decomposition, Topology Generation, and Deployment & Operation) and the *activities* that take place in each phase (Identify Services, Decompose into Architecture, Generate Alternative, and Deploy & Operate, respectively). Each of the activities in the figure is implemented by a MAPE-K controller which is responsible for monitoring the situation at its level (e.g. α-topology), analyzing its behavior (is the application within its SLA and budget constraints?), planning for an action if necessary (deciding whether to transition into a new viable topology by moving into the Generate Alternative phase, or into a new *alpha*-topology by escalating the decision upwards into the Service Identification controller), and executing the decided action. Rules for the decision making, and the outcomes of past decisions are persisted in the knowledge base component of the controller at each level in order to learn over time about the effectiveness of each decision in a given context. Figure 2 shows the flow of information between the controllers of each level as

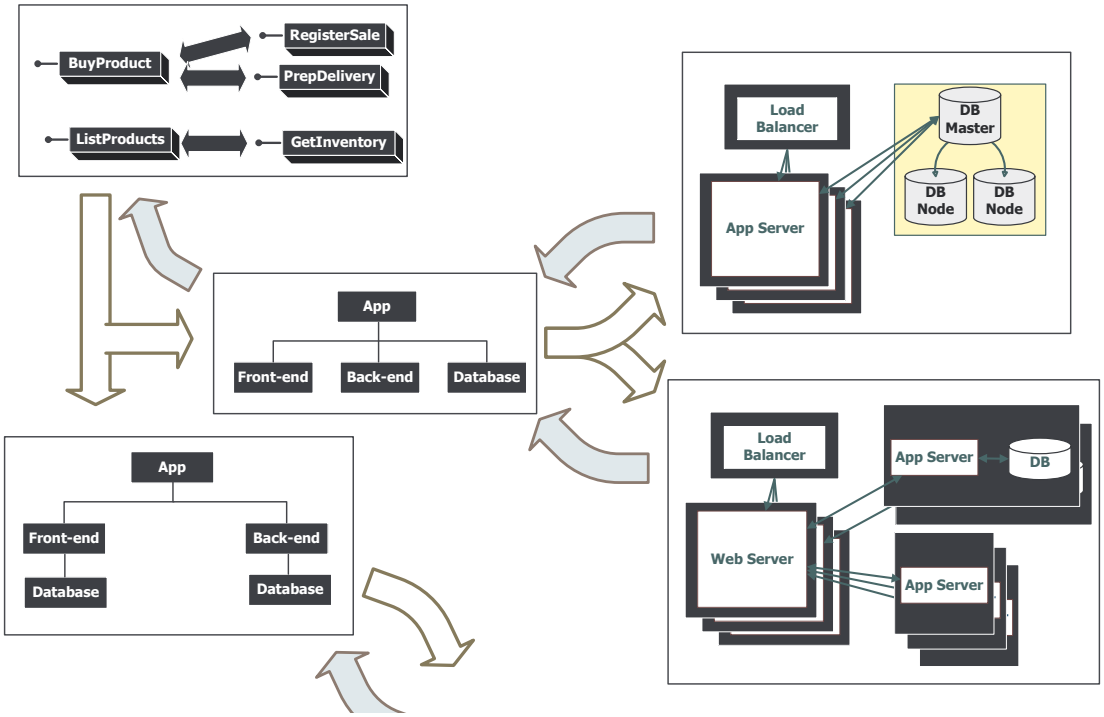


Fig. 3. The Lifecycle of an Example CBA (Web Shop)

dashed arrows between the loops. Bottom-level controllers (i.e. the controllers of the components in a viable topology in the Deployment & Operation phase) can only decide to escalate the need for an adaptive action upwards in the hierarchy, while top-level controllers (i.e. the controller at the level of Service Identification) can only trigger transitions into a lower level through the next phase.

4.2 An Example Instantiation

Figure 3 shows an example instantiation of the proposed lifecycle in the case of a Web Shop application. The service portfolio for the Web Shop consists (among others) of two client-facing services: BuyProduct and ListProducts, the former of which is composed out of services RegisterSale and PrepDelivery, while the latter one is using the internal service GetInventory. The BuyProduct service can be decomposed into a classic three-tier architecture, resulting in the top α-topology in the figure; for ListProducts a simpler two-tier architecture with separate (eventually synchronizing) databases is used. Staying with the first α-topology we can see that there are at least two alternative viable topologies to consider: in the first a DBaaS solution like AWS RDS[10] is used for the Database tier, operating in a cluster mode for scalability purposes. The front- and back-end are implemented as a web application deployed inside an *App Server* like JBoss[11]

[10] Amazon Relational Database Service: `https://aws.amazon.com/rds/`
[11] JBoss: `http://www.jboss.org/`

that is scaled horizontally by running inside multiple VMs in a service like AWS EC2[12]. A Load Balancer solution is deployed inside its own VM for traffic routing. An alternative viable topology for the Web Shop consists of deploying the front-end in its own dedicated VM cluster, decoupling the stateless functionalities of the back-end and deploying them separately in their own stack, and bundling the rest of the back-end into VMs combining application servers and database instances that replace the DBaaS solution (but which still need some logic to synchronize). Such transformations require of course much more detailed α-topologies than the examples in Fig. 3 that are kept to a minimum for illustration purposes, but are nevertheless possible to be largely automated given an appropriate knowledge base of reusable software stacks expressed e.g. as γ-topologies [4].

4.3 Evaluation & Discussion

Looking at the requirements identified in Section 3, it can be seen that the proposed lifecycle satisfies indeed them by: (i) seamlessly integrating service-orientation concepts both at the level of the artifacts that it deals with (applications as service portfolios), and at the level of cloud services used as the underlying resources for the deployment and operation of an application; (ii) building around the dynamic nature of application topologies by decoupling their α-topology from the actual viable topology and relying on the generation of the latter on demand to cope with changes in the perceived behavior of the application through the hierarchy of MAPE-K controllers; (iii) implementing the foreseen self-* characteristics by means of the same controllers; and finally, (iv) by introducing awareness of the consumed resources across the different phases of the lifecycle. However, validation of the lifecycle in more complex scenarios than the example presented in the previous through e.g. field studies, is the subject of future work since it is related with the development of the necessary tooling to support it (see Section 6). Furthermore, and in terms of limitations to the presented work there are two main issues not covered by the discussion: quality assurance for the developed software, and security and privacy. Both of these issues are in practice cross-cutting concerns running in parallel to the lifecycle, and while it can be argued that they could therefore be considered external to it, they nevertheless need to be examined further in future works.

5 Related Work

There are a number of mature works in the literature focusing on the complete lifecycle of cloud-based applications that are related to the lifecycle proposed here. In their majority however they address only parts of the requirements discussed in Section 3. For example, the Cloud Application Lifecycle Model (CALM) and its supporting framework is introduced in [35] without a provision for self-* characteristics or cost awareness. The same holds for [26] that discusses

[12] Amazon EC2: `https://aws.amazon.com/ec2/`

a cloud application lifecycle from a service governance perspective, and for the lifecycle presented in [28] which builds around the notion of blueprints as abstract templates for services to be published in application marketplaces. The work in [36] uses a centralized repository as the means to manage knowledge related to the phases of the lifecycle, but without the notion of cost awareness.

In further related work, the MODAClouds project relies on a Model-Driven Development-based approach to support the lifecycle of cloud-based applications [5]. The project builds on the models@runtime architectural pattern to connect run-time and design-time [13] and provides an IDE for the development, provisioning, deployment, and adaptation of CBAs. Nevertheless, the CBA lifecycle itself is only implicitly defined by this approach. The work in [7], part of the PaaSage project, discusses a service-based application lifecycle that emphasizes a multi-cloud deployment model. When compared to this work, the approach discussed in [7] attempts to (dynamically) optimize provider selection considering also monitoring data without however taking into account the possibility to re-distribute the application as part of this process.

6 Conclusions & Outlook

In summary, this work is based on the observation that the adoption of cloud computing, in conjunction with the advancements in software development in the form of DevOps, container-based software management, and microservices, requires an evoluationary step in software engineering practices, and especially in the area of software architecture. The challenges that drive this evolution are the everything as a service model in which cloud resources are offered, the multi-tenant environment created by resource pooling, the need to incorporate cost awareness due to the utility-based cost model for cloud computing, and the abundance of available offerings that can easily and efficiently replace parts of the software stack of each application. These challenges transform the lifecycle of cloud-based applications into a set of loops that transition between the set of application functionalities encapsulated as services, abstractly defined but application-specific architectural models, and software stack models that seamlessly incorporate cloud services. These transitions are triggered by controllers that coordinate within and across the various stages of the lifecycle.

Future work focuses on developing the methodologies and instrumentation necessary in order to support the proposed lifecycle, with a refinement of its various stages as an essential part of this process. A complete IDE in the manner discussed by the MODAClouds approach [5] is identified as the means to achieve this. Such an environment would further allow for field study-based validation of the lifecycle through collaboration with the industry. The development and integration with the IDE of the MAPE-K controllers as the implementation of the lifecycle phases-related activities is a critical component towards this effort.

References

1. Aceto, G., Botta, A., De Donato, W., Pescapè, A.: Cloud monitoring: A survey. Computer Networks 57(9), 2093–2115 (2013)
2. Andrikopoulos, V., Binz, T., Leymann, F., Strauch, S.: How to Adapt Applications for the Cloud Environment. Computing 95(6), 493–535 (2013)
3. Andrikopoulos, V., Bucchiarone, A., Di Nitto, E., Kazhamiakin, R., et al.: Service engineering, pp. 271–337. Springer (2010)
4. Andrikopoulos, V., Gómez Sáez, S., Leymann, F., Wettinger, J.: Optimal Distribution of Applications in the Cloud. In: Jarke, M., Mylopoulos, J., Quix, C. (eds.) Proceedings of the 26th Conference on Advanced Information Systems Engineering (CAiSE 2014). pp. 75–90. Lecture Notes in Computer Science (LNCS), Springer (2014)
5. Ardagna, D., Di Nitto, E., Casale, G., Petcu, D., et al.: Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds. In: Proceedings of the 4th International Workshop on Modeling in Software Engineering. pp. 50–56. MiSE '12, IEEE Press, Piscataway, NJ, USA (2012)
6. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., et al.: Above the clouds: A berkeley view of cloud computing. Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley (Feb 2009), `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html`
7. Baryannis, G., Garefalakis, P., Kritikos, K., Magoutis, K., et al.: Lifecycle management of service-based applications on multi-clouds: a research roadmap. In: Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds. pp. 13–20. ACM (2013)
8. Bass, L., Weber, I., Zhu, L.: DevOps: A Software Architect's Perspective. Addison-Wesley Professional (2015)
9. Binnig, C., Kossmann, D., Kraska, T., Loesing, S.: How is the weather tomorrow?: towards a benchmark for the cloud. In: Proceedings of the Second International Workshop on Testing Database Systems. p. 9. ACM (2009)
10. Bouguettaya, A., Singh, M., Huhns, M., Sheng, Q.Z., et al.: A service computing manifesto: the next 10 years. Communications of the ACM 60(4), 64–72 (2017)
11. Cavage, M.: There's just no getting around it: you're building a distributed system. Queue 11(4), 30 (2013)
12. Erl, T.: SOA: principles of service design. Prentice Hall Press (2007)
13. Ferry, N., Solberg, A.: Models@ runtime for continuous design and deployment. In: Model-Driven Development and Operation of Multi-Cloud Applications, pp. 81–94. Springer (2017)
14. Folkerts, E., Alexandrov, A., Sachs, K., Iosup, A., et al.: Benchmarking in the cloud: What it should, can, and cannot be. In: Technology Conference on Performance Evaluation and Benchmarking. pp. 173–188. Springer (2012)
15. Heinrich, R., Schmieders, E., Jung, R., Rostami, K., et al.: Integrating run-time observations and design component models for cloud system analysis. In: Proceedings of the 9th Workshop on Models@run.time. vol. 1270, pp. 41–46. CEUR (2014)
16. Hüttermann, M.: Infrastructure as Code, pp. 135–156. Apress (2012)
17. Iosup, A., Yigitbasi, N., Epema, D.: On the Performance Variability of Production Cloud Services. In: Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on. pp. 104–113. IEEE (2011)
18. Jula, A., Sundararajan, E., Othman, Z.: Cloud computing service composition: A systematic literature review. Expert Systems with Applications 41(8), 3809–3824 (2014)

19. Kashef, M.M., Altmann, J.: A cost model for hybrid clouds. In: International Workshop on Grid Economics and Business Models. pp. 46–60. Springer (2011)
20. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
21. Kratzke, N., Quint, P.C.: Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study. Journal of Systems and Software 126, 1–16 (2017)
22. Li, A., Yang, X., Kandula, S., Zhang, M.: CloudCmp: Comparing Public Cloud Providers. In: Proceedings of the 10th Annual Conference on Internet Measurement. pp. 1–14. IMC '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1879141.1879143
23. Linthicum, D.S.: Moving to autonomous and self-migrating containers for cloud applications. IEEE Cloud Computing 3(6), 6–9 (2016)
24. Mell, P., Grance, T., et al.: The NIST definition of cloud computing. NIST Special Publication 800-145 (2011), http://dx.doi.org/10.6028/NIST.SP.800-145
25. Moldovan, D., Truong, H.L., Dustdar, S.: Cost-aware scalability of applications in public clouds. In: Cloud Engineering (IC2E), 2016 IEEE International Conference on. pp. 79–88. IEEE (2016)
26. Munteanu, V.I., Fortis, T.F., Negru, V.: Service lifecycle in the cloud environment. In: Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on. pp. 457–464. IEEE (2012)
27. Newman, S.: Building microservices. O'Reilly Media, Inc. (2015)
28. Nguyen, D.K., Lelli, F., Papazoglou, M.P., Van Den Heuvel, W.J.: Blueprinting approach in support of cloud computing. Future Internet 4(1), 322–346 (2012)
29. Pahl, C., Jamshidi, P.: Software architecture for the cloud–a roadmap towards control-theoretic, model-based cloud architecture. In: European Conference on Software Architecture. pp. 212–220. Springer (2015)
30. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: State of the art and research challenges. Computer 40(11), 38–45 (2007)
31. RightScale: RightScale 2017 State of the Cloud Report (2017), https://www.rightscale.com/lp/state-of-the-cloud
32. Schad, J., Dittrich, J., Quiané-Ruiz, J.A.: Runtime measurements in the cloud: observing, analyzing, and reducing variance. Proceedings of the VLDB Endowment 3(1-2), 460–471 (2010)
33. van Steen, M., Tanenbaum, A.S.: A brief introduction to distributed systems. Computing 98(10), 967–1009 (2016)
34. Sun, L., Dong, H., Hussain, F.K., Hussain, O.K., Chang, E.: Cloud service selection: State-of-the-art and future research directions. Journal of Network and Computer Applications 45, 134–150 (2014)
35. Tang, K., Zhang, J.M., Feng, C.H.: Application centric lifecycle framework in cloud. In: e-Business Engineering (ICEBE), 2011 IEEE 8th International Conference on. pp. 329–334. IEEE (2011)
36. Tran, H.T., Feuerlicht, G.: Service repository for cloud service consumer life cycle management. In: European Conference on Service-Oriented and Cloud Computing. pp. 171–180. Springer (2015)
37. Walker, E.: The Real Cost of a CPU Hour. Computer 42(4), 35–41 (2009)
38. Weinman, J.: Cloudonomics: a rigorous approach to cloud benefit quantification. J. Software Technol 14(4), 10–18 (2011)
39. Zhang, Q., Cheng, L., Boutaba, R.: Cloud computing: state-of-the-art and research challenges. Journal of Internet Services and Applications 1(1), 7–18 (2010)

A Cloud Computing Workflow for managing Oceanographic Data

Salma Allam[1], Antonino Galletta[2], Lorenzo Carnevale[2], Moulay Ali Bekri[1], Rachid El Ouahbi[1] and Massimo Villari[2]

(1) Lab MIASH & Lab MACS, Department of Computer Science & Mathematics Faulty of Sciences, University Moulay Ismail, Meknes, Morocco
allam.salma@gmail.com, ali.bekri@gmail.com, elouahbi@yahoo.fr
(2) Department of Engineering, University of Messina, Messina, Italy -
{angalletta, lcarnevale, mvillari}@unime.it

Abstract. Ocean data management plays an important role in the oceanographic problems, such as ocean acidification. These data, having different physical, biological and chemical nature, are collected from all seas and oceans of the world, generating an international networks for standardizing data formats and facilitating global databases exchange. Cloud computing is therefore the best candidate for oceanographic data migration on a distributed and scalable platform, able to help researchers for performing future predictive analysis. In this paper, we propose a new Cloud based workflow solution for storing oceanographic data and ensuring a good user experience about the geographical data visualization. Experiments prove the goodness of the proposed system in terms of performance.

Key words: Oceanography, Cloud Computing, Data Collection, Data Management, Data Migration, NoSQL, Big Data

1.1 Introduction

Ocean Data management is a current challenge because both of ocean specific terminology diversity (physio-chemical parameters, sensor type, units of measures, conditions of measures, etc.) and of huge volume of ocean data collected from several international projects. The last aim to control the ocean acidification phenomena, an emerging global problem related to the seawater CO2 rate [1] that negatively affects the environment. Therefore, scientific community was thinking about software for calculating inorganic seawater carbon in order to track the evolution of CO2 in the oceans.

However, traditional desktop or web application can not provide the functionalities required by similar problem. Indeed, storing and processing a big volumes of data needs availability, reliability and scalability. For this purpose, the best choice for this kind of application is Cloud Computing, which delivers the resources for managing efficiently the collected data.

The goal of this scientific work follows the previous one [2]. More specifically, in this paper we planned to improve scalability and user experience of the Web Application, used for visualizing oceanographic data, already developed. For this purpose, here

we analyzed the oceanographic data contest in order to design a Cloud workflow for migrating data from online databases to a distributed system able to enable future predictive analysis. Thus, we designed a data acquisition and integration workflow through a Cloud Storage approach which use a more recommended NoSQL solution for successfully managing semi-structured data and retrieving them for future seawater's acidification predictive analysis.

The rest of the paper is organized as follows. Related Works are described in the section 1.2. In Section 1.3, we discussed the material used in this scientific work, from data source up to main oceanographic data issues. The Section 1.4 explains the Cloud approach used in order to migrate oceanographic data from sources to Cloud Storage, whereas in Section 1.5 we discussed the outcomes' experiments. Finally, the Section 1.6 concludes the paper with the lights for the future.

1.2 Related Work

The most popular oceanographic data visualization software is the Ocean Data View (ODV). According to R. Schlitzer [3], ODV is a software used for the interactive exploration, analysis and visualization of oceanographic and other geo-referenced profile, time-series, and trajectory or sequence data. It displays original data points or gridded fields based on the original data and supports different data formats. ODV displays data on different views representing it in a global map that integrates the gridding algorithms [4] based software, called DIVA [5], in order to grid elements in the map for performing interpolation. Moreover, it allows to select one data source by entering the outer coordinates, considering the result as a separate small collection. In addition, ODV allows to select features for drawing one or more specific diagrams in order to compare these.

A software for 3D visualization has been proposed by W. Ware & al. [6]. The representation of that requires a user visual stimulation and allows them to compare two or more locations [7].

On the other hand, in recent time, the scientific community has started an investigation about atmospheric and oceanographic research using the Cloud Computing paradigm [8]. In order to proof that, in [9], the author reported a survey for discussing the progress made by Cloud in the oceanographic challenges. These include effective discovery, organization, analysis and visualization of large amounts of data. In [10], the authors reported "*the outcomes of an NSF-funded project that developed a geospatial cyberinfrastructure to support atmospheric research*". Specifically, they provided several modules for covering the aforementioned challenges in order to "*develop an online, collaborative scientific analysis system for atmospheric science*". In [11], instead, the authors described the LiveOcean project, which aims to mitigate "*the financial impact of ocean acidification on the shellfish industry in the Pacific Northwest of the United States*". The authorts builded this system on Microsoft Azure Cloud Platform highlighting the modularity as most importante theme.

Other important aspect is the management of the sensors designated for gathering row data. Indeed, increasing the number of data also the oceanographic context entries in the Big Data problem and specific solutions, such as [12] and [13], are useful for be inspired.

Our approach aims to go over the [3], [4], [5], [6] and [7] solutions, proposing a Cloud Computing scenario in order to provide for Big Data coming into the oceanographic context.

Cloud Computing is a very hot topic in scientific community, it raises challenges in research fields as described in [14], [15], [16] and [17]. Most of the works are focused on the study and realization of innovative models that allows the collaboration among different Cloud providers focusing on various aspects of the federation. New trends in scientific work aim to adopt the Federation for new interesting scenarios: IoT, Edge, Cloud and Osmotic Computing [18].

1.3 Material

The following section describes the material used for this scientific work, highlighting the data structure and discussing the main challenges and issues.

1.3.1 Data Source

Data used in this scientific work came from the Carbon Dioxide Information Analysis Center (CDIAC), which is considered the first information analysis center for the oceanic parameters [19]. It provides an important climate-change database center organized as showed in the Figure 1.1.

In particular, with reference to the figure 1.1, it is possible to notice the ODVServer zone, that represents the data sources selected in our study. Data are stored into relational databases, it is possible to export them using the comma separated value (CSV) format. In particular the ODVServer Data System is composed by three databases:

1. the GLobal Ocean Data Analysis Project (GLODAP) which gathers unified dataset for determining the "*global distributions of both natural and anthropogenic inorganic carbon, such as radiocarbon*" [20];
2. the PACIFic ocean Interior CArbon (PACIFICA) database that gathers "*data synthesis of ocean interior carbon and its related parameters in the Pacific Ocean*" [21];
3. the CARbon dioxide IN the Atlantic Ocean (CARINA) database which gathers "*data set of open ocean subsurface measurements for biogeochemical investigations*" [22].

Other data sources (SOCAT, CORILIOS CORA, JGOFS, eWOCE, LDEO and CLIVAR) are out of the scope of this paper and will be treated in future works.

1.3.2 Data Structure

Data collected in the aforementioned databases have a common structure explained in the following:

– **Time data**: Month, Day, Year;
– **Location data**: Longitude, Latitude, Depth;

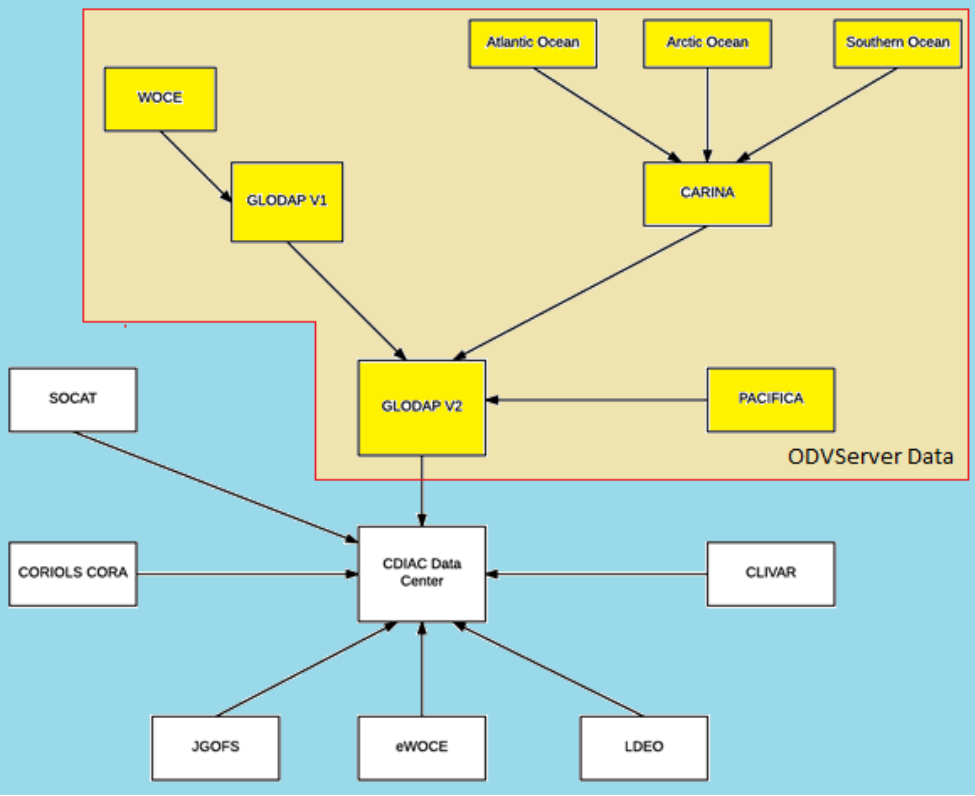


Fig. 1.1. CDIAC Data Center

– **Physical & Chemical data**: Section, Station, Cruise, BottomDepth, BottleNumber, Cast, Salinity, cdtSalinity, Oxygen, Nitrate, Nitrite, Silicate, Phosphate, CFC11, CFC12, CFC113, TCO2, Alkalinity, pCO2, pHSWS25, pHSWS25_Temp, AnthropogenicCO2, DOC, TOC, DeltaC14, DeltaC13, H3, DeltaH3, He, C14err, H3err, DeltaH3err, He_err, CCl4, SF6, AOU, pCFC11, CFC11Age, pCFC12, pCFC113, pCCl4, pSF6, CFC12Age, PotentialAlkalinity, ConventionalRadiocarbonAge, NaturalC14, bkgc14e, BombC14, BombC14atom, NaturalC14atom, PotTemperature, SigmaTheta, Sigma1, Sigma2, Sigma3, Sigma4, bf, sf, cdtsf, of, no3f, no2f, sif, po4f, cfc11f, cfc12f, cfc113f, tco2f, alkf, pco2f, phsws25f, aco2f, docf, tocf, c14f, c13f, h3f, dh3f, hef, ccl4f, sf6f, aouf, palkf, bkgc14f, bombc14f;
– **Environmental data**: Pressure, Temperature.

Data are collected according to the specific oceanographic region. Specifically, each region is composed by a set of sections that contain several stations geographically located. For each of them, there are more than one record depending on the depth variation. The Figure 1.2 represents a hierarchical overview of the entire dataset just described.

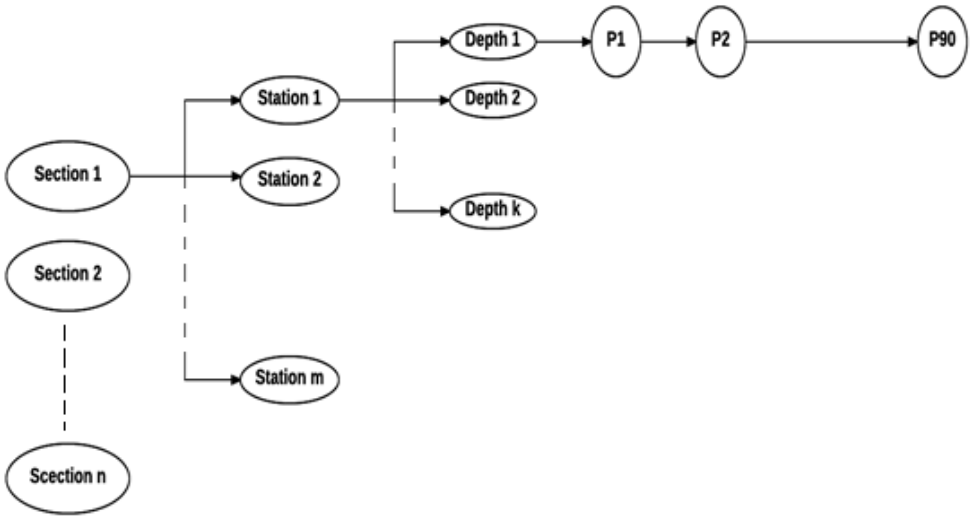


Fig. 1.2. Hierarchical multivalued attributes. Here P1, P2, ... , P90 denote Physical & chemical data.

1.3.3 Challenges and Issues of Oceanographic Data

The CDIAC oceanographic data have the 'Cruise/Section-Station-Depth' form. Such data are in relationships with time and space and archive all information about how, when and where data were stored, as well as type and nature of available data. Thus, it provides a conceptual overview of ocean data structure that should be useful in data management.

In ocean data, the dimensions are normally recorded as 'date/time, latitude, longitude, and depth' [23]. By associating latitude and longitude in location, the model will be 'date/time, location, and depth'. Unfortunately, data provided by the GLODAP, PACIFICA and CARINA databases do not have the data/time field in the 'dd/mm/yy hour:minute:second' form, but only in the 'dd/mm/yy' form. Therefore, the model becomes 'data, location, depth'. For this purpose, it is difficult to treat these data as time series.

Referring to the TLZ model [23], there are eight possible relationships, as reported in the table 1.1.

We also noted that the geographical distance and depth gauge is not periodic among samples, i.e the Figure 1.3 shows that the distance among stations is not the same and the stations depth is also different. More specifically, the data gauge on different depth is not uniform throughout the sample collection process.

Thus, the main challenge was to manage these data, in order to facilitate future predictive analysis and query them in a flexible way.

1.4 Methodology

As mentioned in the Section 1.1, this work aimed to improve the previous one [2] through the Cloud Computing utilization. Specifically, the Sun/Oracle's JEE-based

Table 1.1. DLH space relationship. Noted Date='D', Location='L' and Depth='H'.

D L H	Relationship
1 1 1	One date, one location, one depth
1 1 n	One date, one location, many depth
1 n 1	One date, many location, one depth
1 n n	One date, many location, many depth
n 1 1	many date, one location, one depth
n 1 n	many date, one location, many depth
n n 1	many date, many location, one depth
n n n	many date, many location, many depth

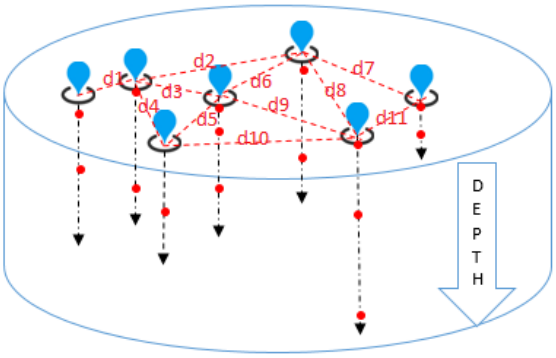


Fig. 1.3. A representation of data characteristic

cross-platform, called ODVServer, used to store data in a traditional SQL database and to visualize oceanographic data using Google Maps APIs. The figure 1.4 shows an overview of the previous platform.

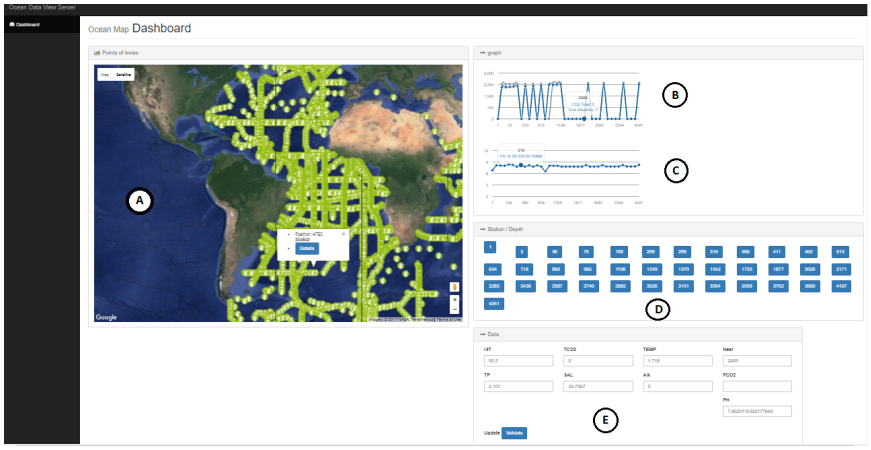


Fig. 1.4. Overview of the ODVServer platform. The layout was divided into two side. The first one (A) provided a geographical representation of the reading. Indeed, the right side (B, C, D, E) provided the insight view of the selected station.

Unfortunately, the visualization of all sections from one database was not easily distinguishable. Moreover, we were not able to analyze the oceanographic data on the basis of different geographical shapes. For this purpose, we propose the Cloud improvement reported in the following.

1.4.1 Workflow

Driven by the need to improve the viewing system of the different oceanographic sections, we have planned to replace web data processing, performed with the Google Maps APIs, with the native storage of a GeoJSON, a human and machine-readable format for encoding geographic data structures. Therefore, a NoSQL database was required in order to manage semi-structured and non structured data and for storing all the oceanographic databases.

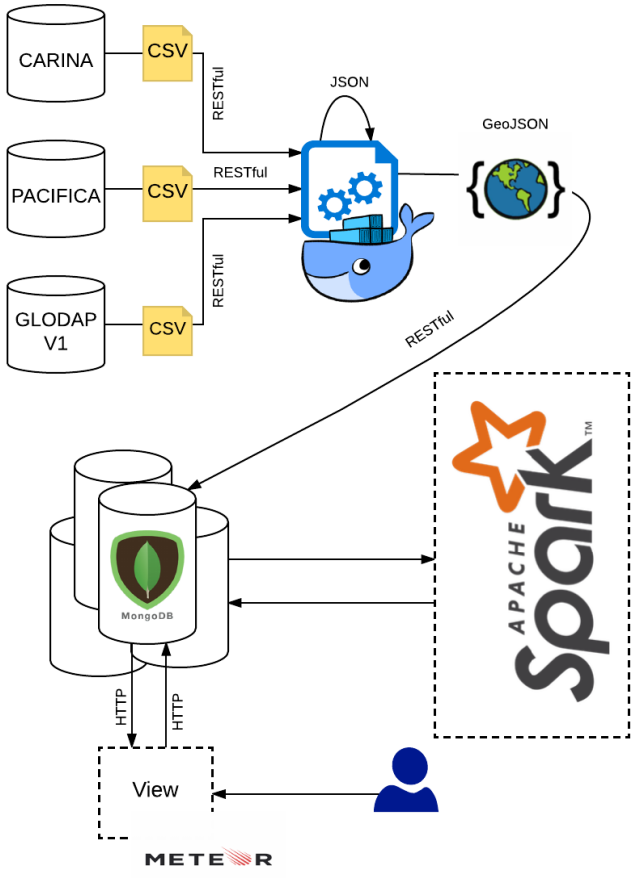


Fig. 1.5. The workflow includes the data acquisition and integration phases, considering the CSV format databases (CARINA; PACIFICA and GLODAP). A NoSQL solution stores all the GeoJSON information and integrates well with data analysis tool, such as Apache Spark, and MEAN stack web application, such as the Meteor JS framework. The dotted lines indicates guidelines for future works.

Referring to the Figure 1.5, data move from sources (CARINA, PACIFICA and GLODAP) to the Cloud Platform through RESTful APIs. Specifically, the listening microservice receives CSV data in order to implement the transformation into GeoJSON. Thus, this information moves to a sharded and replicated MongoDB distribution, a native JSON NoSQL database. The choice of MongoDB avoids further data transformations. Moreover, in order to scale the microservice workload, it is embedded inside a docker container, ensuring a lightweight and portable service virtualization.

The bottom side of the figure 1.5 shows a HTTP communication between the distributed storage and the front end, in order to view the query and future analyses results. At the same time, Apache Spark has been thought for performing future real-time predictive algorithms on oceanographic data. However, the dotted lines indicate guidelines for future works.

1.4.2 Oceanographic Data Visualization

Based on the previous description, we looked for two properties: interoperability and flexibility. The first one is guaranteed by the GeoJSON standard. Its fixed structure identifies two parts: **geometry** section contains geospatial information and type of GeoJSON element (see section 2 in the Figure 1.6); and **properties** section contains all parameters codified as key-value pairs (see section 3 in the Figure 1.6). We remark that section 1 in the Figure 1.6 represents the MongoDB's master key.

Fig. 1.6. Difference structure of JSON and GeoJSON

On the other hand, the flexibility is guaranteed by data management and visualization dynamism, which allow users to select any representation. Indeed, in our approach,

we decided to store all samples in a single MongoDB collection. Thus, we can create virtual representation based on users' demand. For instance, as showed in the Figure 1.7, by means of our approach users are able to create virtual representations per each zone of interest, starting from position of samples or other constrains.

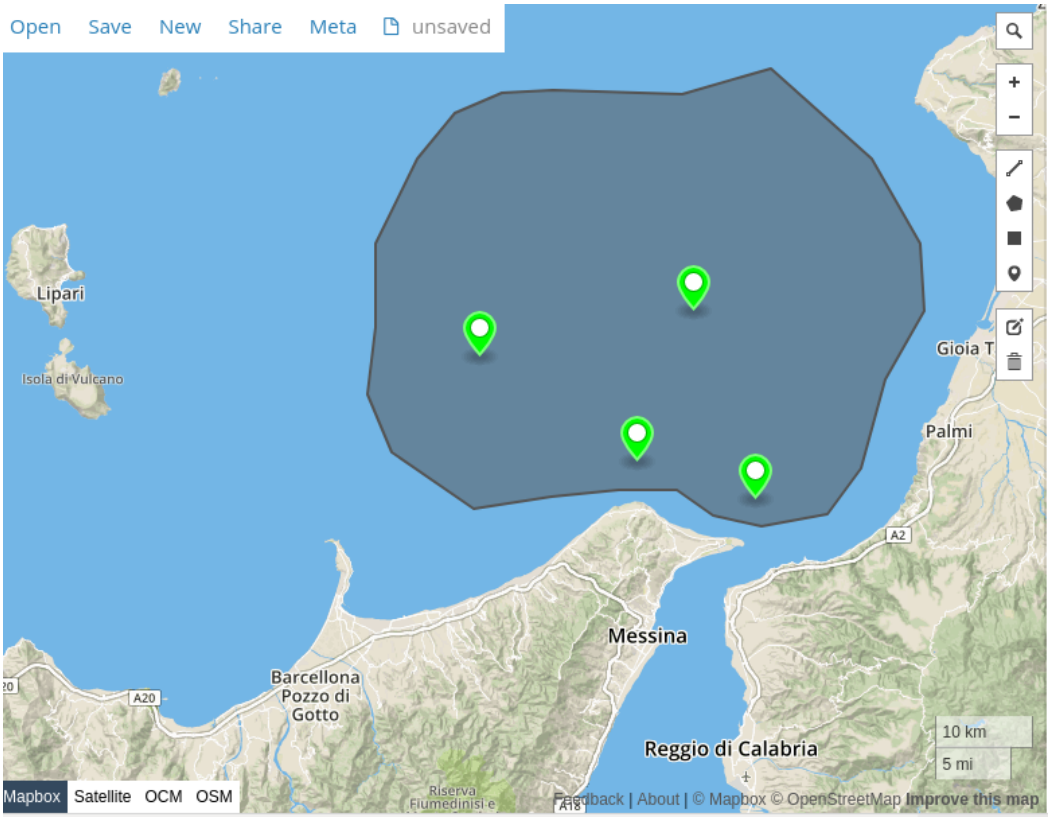


Fig. 1.7. User demand

1.5 Performance

In this section we discuss about of the performances of the system from a numerical point of view. In particular, we conducted two different kind of analysis: for populating and retrieve data of our system. Our testbed is composed of two different blades server. More specifically, we have 2 different type of machines one for the computation and the other one for the storage. Computation workstation, in which is running the data conversion module, is equipped by the Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz, RAM 16GB, OS: Ubuntu server 16.04 LTS 64 BIT. Storage workstation, in which our database system MongoDB is running on single node, is composed by the Intel(R) Core(TM) i3-6100 CPU @ 3.70GHz, RAM 32GB, and Ubuntu server 16.04 LTS 64 BIT. Unfortunately we did not find any solution to compare performances of our system.

We made scalability tests in different scenarios for both type of analysis. Experiments were repeated 30 subsequent times in order to consider confidence intervals at 95% and average values.

1.5.1 Insert data

Fig.1.8 shows the performances for parsing CSV data to GeoJSON and storing them into MongoDB. Its behavior is linear with the increasing of the dataset size. On the x-axis we reported the dataset size, whereas on the y-axis, we reported the response time expressed in msec. How we can observe, the response time for 100.000 samples is acceptable less than 10 sec.

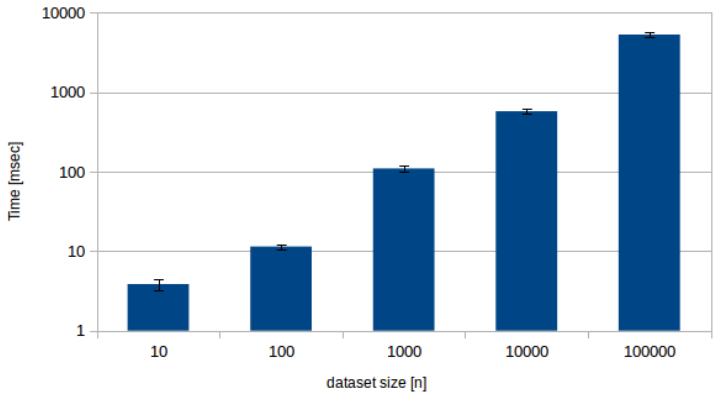


Fig. 1.8. Data insert performance

1.5.2 Retrieve data

Here we consider times for retrieving data from MongoDB in a specific geographic shapes, in order to understand if flexibility features are really implementable. More specifically we considered increasing concentric circles with different radius, starting from 10 meters up to 100 kilometers.

The behavior, as showed in Fig.1.9, is constant around 30 msec, variations are due to networks delay.

1.6 Conclusions and Future Work

In this scientific work, we investigated the management of oceanographic data through the utilization of a Cloud Computing workflow. First of all, three CSV format databases have been selected as data sources. Therefore, we explained the workflow necessary for migrating these data up to the Cloud Storage. This scientific work is the first initiative adopting Cloud for manage Ocean data, for this reason we did not find any solution

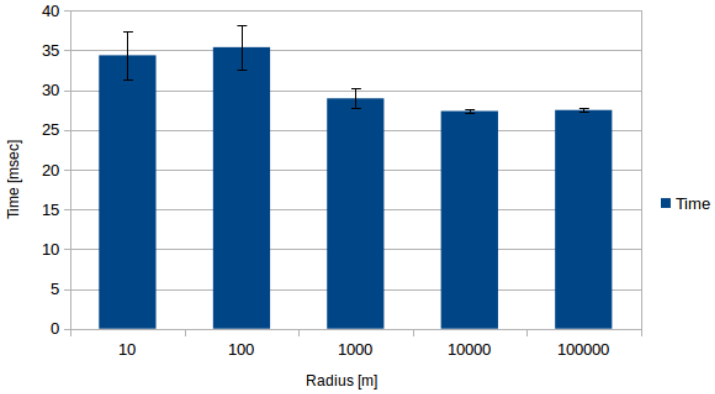


Fig. 1.9. Data retrieve performance

to compare performance of our system. However experiments showed that our system response time presents a linear trend. The execution time grows up with the increasing number of considered samples.

On the other hand, the dotted lines in the figure 1.5 shows our idea about the future perspective. In particular, Meteor JS framework will be the technology we aim to use for developing the new frontend version. This choice depends on the native MEAN stack adopted, which includes MongoDB as backend database; whereas Apache Spark will be useful for performing predictive oceanographic data analysis, such as the acidification prediction. About that, the figure 1.10 shows a possible future work about the selection of the features that best describe the reported behavior. Other future works are related to adopt the new Osmotic Computing paradigm.

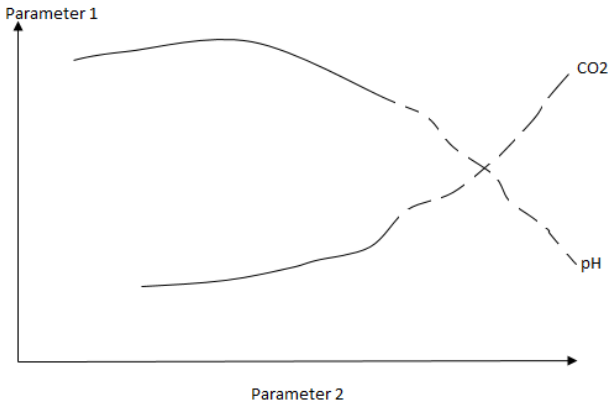


Fig. 1.10. A goal of the future work can be the acidification prediction.

ACKNOWLEDGMENT

This work has been supported by Cloud for Europe grant agreement number FP7-610650 (C4E) Tender: *REALIZATION OF A RESEARCH AND DEVELOPMENT PROJECT (PRE-COMMERCIAL PROCUREMENT) ON "CLOUD FOR EUROPE"*, Italy-Rome: Research and development services and related consultancy services Contract notice: 2014/S 241-424518. Directive: 2004/18/EC. (http://www.cloudforeurope.eu/).

References

1. S. C. Doney, W. M. Balch, V. J. Fabry, and R. A. Feely, "Ocean acidification A critical emerging problem," *Oceanography*, vol. 22, no. 4, pp. 16–25, 2009.
2. R. E. S. ALLAM and M. D. E. O. OUAHBI, "ADVANCES IN INFORMATION TECHNOLOGY : THEORY AND APPLICATION ISSN : 2489-1703 February 2016," vol. 1, no. February, pp. 163–166, 2016.
3. R. Schlitzer, "Ocean Data View," pp. 1–11, 2011.
4. W. H. F. Smith and P. Wessel, "Gridding with continuous curvature splines in tension," *Geophysics*, vol. 55, no. 3, pp. 293–305, 1990. [Online]. Available: http://library.seg.org/doi/10.1190/1.1442837
5. G. Started, "Ocean Data View," pp. 1–11, 2011.
6. C. Ware, M. Plumlee, R. Arsenault, L. A. Mayer, S. Smith, and D. House, "GeoZui3D: Data fusion for interpreting oceanographic data," *Oceans Conference Record (IEEE)*, vol. 3, pp. 1960–1964, 2001.
7. M. Plumlee and C. Ware, "An evaluation of methods for linking 3D views," *Proceedings of the Symposium on Interactive 3D Graphics*, pp. 193–201, 2003. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-0038642661{\&}partnerID=tZOtx3y1
8. K. Butler and N. Merati, "Analysis patterns for cloud-centric atmospheric and ocean research," in *Cloud Computing in Ocean and Atmospheric Sciences*. Elsevier, 2016, pp. 15–34. [Online]. Available: https://doi.org/10.1016/b978-0-12-803192-6.00002-5
9. R. Wigton, "Forces and patterns in the scientific cloud," in *Cloud Computing in Ocean and Atmospheric Sciences*. Elsevier, 2016, pp. 35–41. [Online]. Available: https://doi.org/10.1016/b978-0-12-803192-6.00003-7
10. W. Li, H. Shao, S. Wang, X. Zhou, and S. Wu, "A2ci," in *Cloud Computing in Ocean and Atmospheric Sciences*. Elsevier, 2016, pp. 137–161. [Online]. Available: https://doi.org/10.1016/b978-0-12-803192-6.00009-8
11. R. Fatland, P. MacCready, and N. Oscar, "LiveOcean," in *Cloud Computing in Ocean and Atmospheric Sciences*. Elsevier, 2016, pp. 277–296. [Online]. Available: https://doi.org/10.1016/b978-0-12-803192-6.00014-1
12. M. Fazio, A. Celesti, M. Villari, and A. Puliafito, "The need of a hybrid storage approach for iot in paas cloud federation," in *2014 28th International Conference on Advanced Information Networking and Applications Workshops*, 2014, pp. 779–784.
13. A. Celesti, N. Peditto, F. Verboso, M. Villari, and A. Puliafito, "Draco paas: A distributed resilient adaptable cloud oriented platform," in *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, 2013, pp. 1490–1497.
14. F. Tusa, A. Celesti, M. Villari, and A. Puliafito, "How to enhance cloud architectures to enable cross-federation," in *Proceedings of IEEE CLOUD '10*. IEEE, July 2010, pp. 337–345.

15. G. Vernik, A. Shulman-Peleg, S. Dippl, C. Formisano, M. Jaeger, E. Kolodner, and M. Villari, "Data on-boarding in federated storage clouds," in *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, June 2013, pp. 244–251.
16. I. Goiri, J. Guitart, and J. Torres, "Characterizing cloud federation for enhancing providers' profit," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, july 2010, pp. 123 –130.
17. S. Azodolmolky, P. Wieder, and R. Yahyapour, "Cloud computing networking: challenges and opportunities for innovations," *Communications Magazine, IEEE*, vol. 51, no. 7, pp. 54–62, July 2013.
18. M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan, "Osmotic computing: A new paradigm for edge/cloud integration," *IEEE Cloud Computing*, vol. 3, no. 6, pp. 76–83, Nov 2016.
19. "Catalog of Databases and Reports," no. May, 1999. [Online]. Available: http://cdiac.ornl.gov/oceans/
20. R. Key, A. Olsen, S. van Heuven, S. Lauvset, A. Velo, X. Lin, C. Schirnick, A. Kozyr, T. Tanhua, M. Hoppema, S. Jutterström, R. Steinfeldt, E. Jeansson, M. Ishi, F. Perez, and T. Suzuki, "Global Ocean Data Analysis Project, Version 2 (GLODAPv2)," *Ornl/Cdiac-162, Ndp-093*, vol. 2, 2015.
21. T. Suzuki, M. Ishii, M. Aoyama, J. R. Christian, K. Enyo, T. Kawano, R. M. Key, N. Kosugi, A. Kozyr, L. A. Miller, A. Murata, T. Nakano, T. Ono, T. Saino, K.-I. Sasaki, D. Sasano, Y. Takatani, M. Wakita, and C. L. Sabine, "Pacifica Data Synthesis Project," *Ornl/Cdiac-159, Ndp-092*, 2013.
22. M. Hoppema, A. Velo, S. van Heuven, T. Tanhua, R. M. Key, X. Lin, D. C. E. Bakker, F. F. Perez, a. F. Ríos, C. Lo Monaco, C. L. Sabine, M. Álvarez, and R. G. J. Bellerby, "Consistency of cruise data of the CARINA database in the Atlantic sector of the Southern Ocean," *Earth System Science Data*, vol. 1, pp. 63–75, 2009.
23. C. L. Hubbs, "University of Michigan, U. S. A." vol. III, pp. 1–6, 1930.

Pattern-driven Architecting of an Adaptable Ontology-driven Cloud Storage Broker

Divyaa Manimaran Elango[1], Frank Fowley[1], and Claus Pahl[2]

[1] IC4, Dublin City University, Dublin, Ireland
[2] Software and Systems Engineering Research Centre, Free University of Bozen-Bolzano, Bolzano, Italy

Abstract. Cloud service brokerage enables the cloud service ecosytem to become more interoperable and allows users to migrate between offerings easily. To this end, we developed a multi-cloud storage broker in the format of an API to allow objects to be stored and retrieved uniformly across a range of cloud-based storage providers, allowing for portability and easy migration of software systems. This multi-cloud storage abstraction is implemented as a Java-based multi-cloud storage API and supports GoogleDrive, DropBox, Microsoft Azure and Amazon Web Services as sample service providers. The library offers three services, namely a file service, blob service and table service. A test application was used to compare storage operations across different providers. The abstraction is based on a layered ontological framework. While many multi-cloud applications exist, we will focus on mapping the layered ontology onto a design pattern-based organisation of the architecture to demonstrate how this meets often required maintainability and extensibility properties.

Keywords: Cloud Service Brokerage, Cloud Storage, Data Migration, Ontology, API Performance.

1 Introduction

Cloud service brokerage (CSB) allows the cloud service ecosystem to become more interoperable, thus allowing for portability and easier migration between providers [22, 30, 29]. CSB enables this portability and migration through the integration and adaptation of different provided service into a uniform presentation [14]. Thus, we developed a multi-cloud storage broker in the format of an API to allow objects to be stored and retrieved uniformly across a range of storage providers. The abstraction is based on a layered ontological framework, which is organised around common design patterns to ensure maintainability and extensibility to adapt to different service providers [20].

We introduce the basic concepts involved in designing the brokerage framework and methodology behind the abstraction library. The multi-cloud storage abstraction outcome is a Java-based multi-cloud storage API. This is available as a jar file that supports GoogleDrive, DropBox, Microsoft Azure and Amazon

Web Services [36, 34, 35, 33]. The library offers three services, namely a file service, a blob service and a table service. An application of the library can also be used to compare storage operations across different providers [15, 26].

In our presentation, we focus in particular on the ontology framework and how it is mapped onto a layered, design pattern-based library architecture for the API we developed [27, 28] – an aspect that has not been sufficiently covered in the presentation of other multi-cloud brokers.

This document is structured as follows. Section 2 gives an outline of cloud service brokerage. Section 3 describes background and related work. In Section 4, the ontology-based interoperability framework is explained, and Section 5 looks at other architectural design aspects. Section 6 discusses the implementation effort and the learning outcome. Section 7 contains some conclusions.

2 Cloud Service Brokerage Use Cases

A cloud broker is an intermediary application between a client and cloud provider service [16, 10]. Brokerage reduces the time spent by a client in analyzing different types of services provided by different service providers. In our case, this enables a single platform to offer the client a common cloud storage service. This results in cost optimization and reduced level of back-end data management requirements, but also enables migration of data and files through the joint interface [7]. A multi-cloud storage abstraction API can act as the cloud broker library which facilitates the integration of different types of cloud services [17]. The abstraction library allows the broker to adapt to a rapidly changing marketplace [4]. Changeability and extensibility will therefore be requirements of the broker library [8, 9]. Fig. 1 illustrates the architecture.

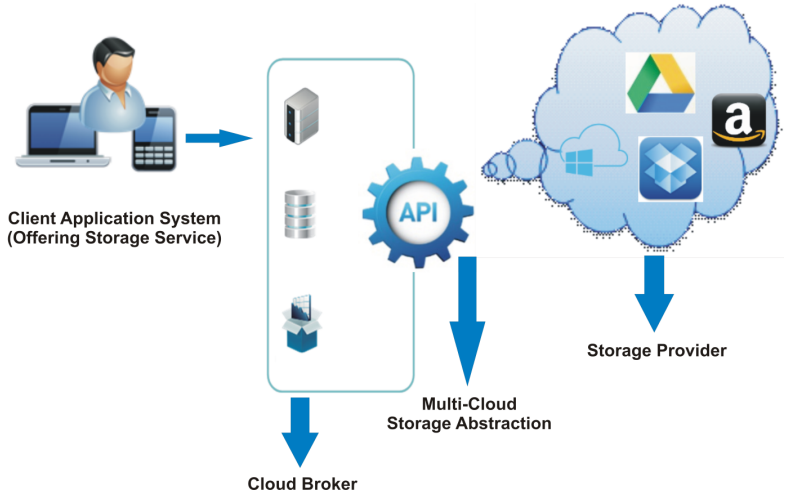


Fig. 1. Service brokerage architecture for cloud storage.

In order to illustrate this, we introduce a use case. Disaster recovery is a sample specific storage use case, used where there is an interruption of an action

or an event in an unpredictable time that causes the services to be unavailable to the end user. Cloud back-up storage is a way of protecting the online resources to make them available in the event of loss of data or any other disaster.

The multi-cloud storage abstraction library best suits this use case in terms of offering hassle-free back up, best time management in terms of restoring processes, increased scalability, security and compliance, redundancy and end to end recovery [2, 3]. The storage providers supported in this abstraction library, namely Microsoft Azure, Google, DropBox and Amazon Web Service, offer good bandwidth and low cost services that can be used for backup and recovery applications.

More generally, vendor lock-in is a barrier in cloud computing preventing users from migrating between providers. Vendor lock-in is a critical situation where the clients are dependent on a single cloud provider. The client is not given an option to migrate to other cloud providers. Issues can arise, such as legal constraints, increased cost and technical incompatibilities when choosing to move between cloud providers. Interoperability and portability are prominent characteristics of cloud computing, affected by vendor lock-in and the lack of standardisation [5, 6].

A multi-cloud storage API can play a crucial role in such cases, making it easier for the client to switch providers. This can be applied across different cloud type environments, like private or public environments which are more beneficial from a business perspective. Furthermore, the extensibility of the library to support new cloud providers gives the client a wide view of portability to many different new cloud providers.

3 Background and Related Work

3.1 Cloud Service Provider APIs

Several cloud storage provider APIs exist, of which we selected four providers with different numbers of individual services [36, 34, 35, 33]. We review some key observations.

- GoogleDrive offers a cloud file storage service. The API is built on OAuth2 authentication and is fairly easy to understand. The structure is clearly documented and the use of method calls is well explained. The GoogleDrive service includes access to a Google API client library. Failure to include http and OAuth client libraries will disable the authentication. The Google developer portal simplifies the way of implementing the API in a workspace and provides details for configuring the authentication.
- DropBox is a file hosting service. It uses SSL transfer for synchronization and AES 256 encryption for security. It also enables synchronised backup and web sharing. The DropBox API is very light-weight and easy for a new user to go through quickly. Code samples and method explanations are given in the developer's portal.

Table 1. Storage services and their providers.

Service	Azure	AWS	Google	DropBox
File	Azure Storage File	-	GoogleDrive	DropBox
Blob	Azure Storage Blob	AWS S3	-	-
Table	Azure Storage Table, Azure DocumentDB	Amazon DynamoDB, Amazon SimpleDB	-	-

- Azure Storage supports blob, file, queue and table services. The API is built on REST, HTTP and XML, and can be easily integrated with Microsoft Visual Studio, Eclipse and GIT. Azure is user friendly, and the classic portal interface can be used to set the storage account and document DB account parameters. The Azure SDK is available for all major development languages. The Azure SDK provides a separate API package for each service and has the same code flow across different service APIs.
- The Amazon Web Service S3 is a file storage service which is built on REST and SOAP. Their SDK is available in all major development languages. The developer portal includes documentation and a quick guide to get started with code development. However, it was considered that each service had too many classes. The library is heavy as it includes many packages for all services. Understanding the class naming was considered quite challenging as it has many services listed in the same SDK documentation. We also experienced the service be inconsistent, as there was an occasional delay in read and write requests.

This brief study of the main features of the providers resulted in a grouping of the cloud providers and their services as shown in Table 1.

3.2 Multi-Cloud Libraries

In order to provide a multi-cloud broker, we looked at existing multi-cloud libraries. Cloud providers publish specifications of their services, which are different style and which makes it hard to use them as a common joint interface. Several existing multi-cloud libraries were studied during this project, including Apache jclouds, DeltaCloud, Kloudless, SecureBlackBox, Temboo and SimpleCloud.

In order to achieve the flexibility requirement that would allow our library to be adapted to changing services or completely different services, we aimed to build the solution on a combination of proven patterns, adapted to the context here. It was decided to adopt an approach similar to that used in the Apache jclouds library for abstraction. Apache jclouds provides cloud-agnostic abstraction [23]. Use of a single instance context for the mapping of a user request in jclouds was used in our architecture. The purpose of having each class for each provider across different levels of service was adopted from a similar design in the SecureBlackBox library. Our concept of including a manager interface layer at each component level is adopted from the Apache LibCloud structure.

4 Ontological Framework

4.1 Abstraction, Interoperability and Extensibility

Abstraction serves to reduce complexity in software. The design of our multi-cloud storage API is built on multiple layers of abstraction. This provides for service-neutral functional logic which also realises extensibility, i.e., allows additional vendors to be supported without changing the underlying core functional logic of the API design. New multi-cloud services can be added to the API without any code change [24]. A programmable abstraction layer provides flexibility to connect and configure services, here in a fully secure way [25]. Thus, interoperability and portability can be achieved. Whilst our jclouds API-based architecture, that we partially adopted, only supports public cloud providers, in future this can be extended to include private clouds. Such APIs are used for developing cloud-based applications like content delivery platforms and back-up applications, as our earlier use case demonstrates.

The main objective of designing and developing an abstraction API for cloud storage is to produce an effective cloud delivery model, with a single portable view that supports enhanced business capabilities such as cloud brokerage [32]. The advantage of bringing these functionalities to an interoperable multi-cloud application provides 1) an easy way of importing and exporting data, 2) choice over price, 3) enhanced SLA, and 4) the elimination of vendor lock-in. There are existing standardisation frameworks being maintained in this area including the Cloud Infrastructure Management Interface (CIMI) and Open Cloud Computing Interface (OCCI) that aim at interoperability. Our integration broker provides interoperability based on a flexible, extensible API.

4.2 Storage Abstraction Ontology

A layered architecture will serve to provide interoperability and extensibility. At the core is a stroage abstraction ontology that describes the common naming and meaning of service concepts across the abstraction layers. This ontology model consists of four main layers, namely Service, Provider, (Level-2) Composite Object and (Level-1) Core Object.

- Service: The Service layer is the top layer and is directly integrated into the user interface layer. This layer basically describes the services that the multi-cloud storage abstraction API supports. There are three services currently supported. They are a blob, a table and a file service.
- Provider: The Provider layer is the second layer where the context object parameters are mapped to the service layer. The multi-cloud storage abstraction supports four main providers namely Microsoft Azure, Amazon Web Services, GoogleDrive and DropBox. The corresponding services supported by the providers are shown below:

Service	Provider
Blob	Azure storage Blob; AWS S3.
Table	Azure Storage Table; Azure DocumentDB; AWS DynamoDB; AWS SimpleDB.
File	Azure Storage File; DropBox and GoogleDrive.

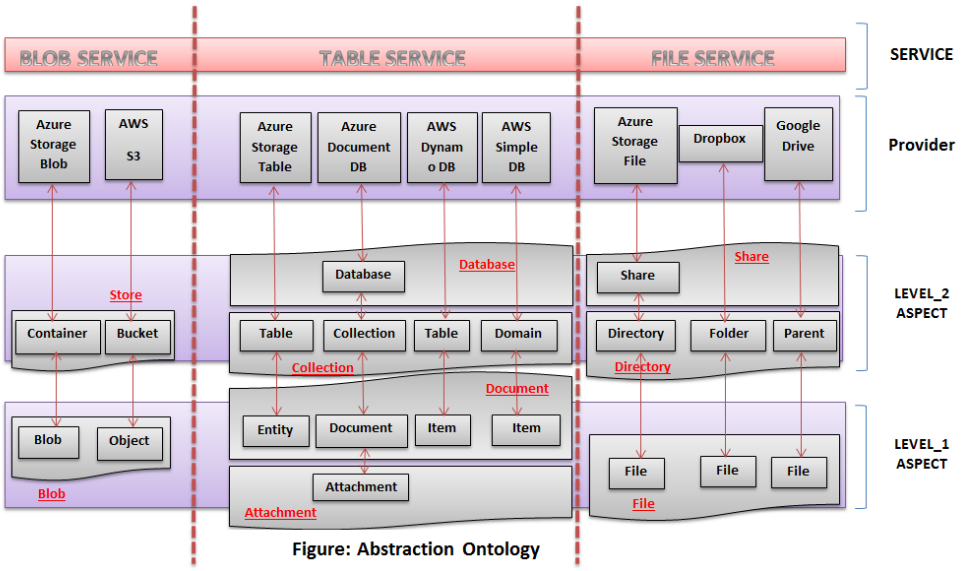


Fig. 2. Storage Abstraction Ontology based on 4 Layers.

– Level-2 (Composite Objects): The Composite Objects (object level-2) aspect is the third layer. This layer represents the first level or higher level of abstraction for the objects blob, table and file. This layer is service-neutral and brings out the common naming across the providers' specific functionalities. Each layer is abstracted based on the common operations and aspect of how the main function is applied in that particular service. Common naming is represented to easily categorise storage resources and group them to make the development of the coding easier.

  As shown in the Abstraction Ontology diagram (Fig. 2), the blob service has "Store" which groups 'Container' from Azure Storage Blob and 'Bucket' from AWS S3. The table service has two different sub-layers – where "Database" belongs to Azure DocumentDB Database, and where "Collection" groups 'Table' from Azure Storage Table, 'collection' from Azure DocumentDB Collection, 'Table' from AWS DynamoDB and 'Domain' from 'AWS SimpleDB'. The file service has two different sub-layers where "Share" belongs to Azure Storage File and "Directory" groups 'Directory' from Azure Storage File, 'Folder' from DropBox and 'Parent' from GoogleDrive.

– Level-1 (Core Object): The Core Object (object level-1) aspect represents the lower level of storage object abstraction. This layer contains the core functionalities of a particular service across different providers. The classes at this level are extended from an abstract class called AbstractConnector. The class implements the abstract methods defined in a AbstractConnector class. The mapping from Level-2 to Level-1 is performed by an interface class called Manager. This Manager identifies the provider class by its key. Basic CRUD operations on the storage resources are included as core methods. In order to implement these functions, each operation "request" should "pass

through" the Level-2 mappings and is then mapped across the service and providers.

The blob service has "Blob" which groups 'Blob' from Azure Storage Blob and 'Object' from AWS S3. The table service has two different sub layers. It has an "Item" which groups 'Entity' from Azure Storage Table, 'Document' from Azure DocumentDB Document, 'Item' from AWS DynamoDB and 'Item' from AWS SimpleDB. Also, the second sub layer "Attachment" belongs to the Azure DocumentDB Attachment. The file service has "File" which groups 'File' from Azure Storage File, 'File' from DropBox and 'File' from GoogleDrive.

4.3 Provider Functionality and Naming Analysis

The API should act as an adapter for accessing different cloud providers' services through a common interface. The main concern was having common naming for mapping the user's requests. A high degree of commonality existed between different cloud provider operations. However, some operations exists in one provider and do not exist in other providers. Moreover, the parameters in some of the methods also differ between providers. In the example below, the common Level-2 aspect of "Store" in the Blob service is defined to cater for providers with different specific names, namely Azure's storage blob container and AWS's S3 bucket. The table also shows the common createStore() method and the corresponding Azure-specific and AWS-specific underlying method calls.

The approach is based on identifying synonyms to common object names, such as container (azure) and bucket (S3) for 'store' as indicated below:

Common Name (Level-2)	Name in Azure Storage Blob	Name in AWS S3
STORE	CONTAINER	BUCKET

The same then applies to function names, as indicated below:

Common Method Name	Name in Azure Storage Blob	Name in AWS S3
createStore()	create() create container	createBucket() create bucket

The abstraction ontology maps similar service groupings together across different cloud providers. Selected services have similar or the same core functional logic – grouped into levels in the ontology. Based on this, the framework design includes the "service", "provider", "composite object" and "core object" for its implementation.

In the example below, the common level-2 composite "Collection" is defined for two providers. There are four corresponding service names, namely Azure storage table 'table', AWS document DB 'collection', AWS dynamo DB 'table' and AWS simple DB 'domain'. The table below shows the common getCollectionMetadata() method and its corresponding provider API method calls:

Common Ontology Name	Azure Storage Table	Azure Document DB	AWS Dynamo DB	AWS Simple DB
Composite: COLLECTION	TABLE	COLLECTION	TABLE	DOMAIN
Operation: getCollectionMetadata()	-	-	describeTable() returns information about the table.	domainMetadata() returns information about the domain.
Operation: listCollection()	listTables() Lists the table names in the account	readCollection() Reads a document collection by the collection link	listTables() Simplified method for invoking ListTables	listDomains() Lists all domains associated with the Access Key ID

The method name to retrieve the metadata of a collection in the Table service is not supported by Azure. However, the AWS API does support the metadata. So, a common operation can not be realised across the level-2 composite Collection. A similar problem exists with the blob and file services. This lack of consistency in the available provider API operations has led to the omission of valuable API method calls.

Another example is also in the table above, which shows the common listCollection() method and its corresponding provider API method calls. The method name to retrieve the list of collections in the table service is supported by both Azure and AWS. However, the API method names are different, although their description and logic is the same.

5 Storage API Design

As already indicated, we mapped an ontology-based conceptual framework onto a layered architecture, which in turn was structured by suitable design patterns. We also consider the security management of the different service providers in this context.

5.1 Ontology-Driven Design

Adopting best-practice in software design was important in order to reduce the development overhead and produce a quality library that could be extensible with new features. A "model-driven" software engineering technique was applied to simplify the process of design from concept modeling to implementation. This was applied to each level in the abstraction ontology by breaking entities into single components. There are two main types of modeling approaches that could have been used for the design. Firstly, there is a provider-specific model, in which the provisioning and deployment of the abstraction library is defined for each cloud provider. The second approach is a cloud provider-independent model, which defines the provisioning and deployment of the abstraction library in a cloud-agnostic way.

The approach outlined in the CloudML EU-funded research project was adapted here. It a domain-specific modelling language to reduce the design complexity for cloud systems. CloudML enables the provisioning and deployment

of an abstraction library. Its design includes level-1 core objects, which are assembled based on the CloudML internal component design. These are mapped to level-2 components using a model-driven approach. A client using the service does not necessarily know about the internal deployment, and there is no limitation on the design and evolution of the multi-cloud abstraction library. The storage library has been coded using the Javadoc framework. Thus, there is extensive commenting and documentation generated, which simplifies the understanding and purpose of each class, method implementation and relationship across classes. The library includes sample code to allow for better understanding of how each service functions.

5.2 Provider Security Analysis – Authentication

Apart from a mapping of concepts for core and composite storage objects used by the different providers into a common ontology, security is another concern that needs to be unified across the providers. It is important to note that authentication differs across each provider.

- Amazon Web Service authentication uses a secret key and an access key, which is common across all the target services supported by our storage abstraction API. An AWS user should have the specified role with the required access permissions to the resource. Identity Access Management allows the user to set the role and access privileges, and this provides each user with sufficient credentials. The account can be activated using phone verification and authentication. Credential auditing and usage reports can be used for review purposes.
- Microsoft Azure account subscription allows for the accessing of the resources available within an azure account. The Blob service is provided within an Azure storage account. The azure storage account name, also known as namespace, is the first level for processing authentication to the services within the storage account such as blob, file and table. It uses token-based authentication. The authentication of the Azure storage blob is done using a connection string which has the parameters of the storage account name and primary key. Similarly, the Table service is supported within an Azure storage account. The authentication of the Azure storage table is done using a connection string which has the parameters of the storage account name and primary key. An Azure Document DB account is required for accessing the Azure Document Db service and requires a master key and URI (also known as an end-point).
- DropBox authentication uses an access token. An access token is generated in the App console. An application is created under the app console and its permission is set to 'FULL'. Later, the authentication is set by linking the account using the access token when passing the instance of the DropBox API client.
- GoogleDrive authentication uses a client_secret json file. A project is created in the Google developer's console. The Drive API and OAuth protocol is enabled. The credentials are generated and saved as a client_secret json file. In

the coding, the authentication method should have the permission scope and drive scope set to 'GRANT'. When the browser opens for the authentication response, the client is permitted to allow for full read and write access.

All these authentication processes were considered when we developed the code for authentication method calls from the multi-cloud storage abstraction API. A credentials object is at the core of the solution:

- Credentials storage: The credentials are stored in a common config file. So, the user is not shown the authentication part as it happens in the back-end.
- Credentials update: If the credentials need to be changed, they are only changed in the configuration file, which reduces the difficulty for the user to set up the authentication process.

5.3 Design pattern

We use design patterns to organise the layered ontology-based architecture further in order to help us in achieving the required maintainability and extensibility requirements, but also in general the efficiency of the implementation.

Mapping using an Object Context for Maintainability. The design pattern in any multi-cloud library should attempt to reduce the need to have an object instantiation for each provider's class using the constructor. This was a problem with the jclouds library where there was a lack of code clarity and high level of complexity in the framework pattern. To overcome this in our multi-cloud storage abstraction, and also to provide a stable, maintainable code base, the context builder class is added to the architecture. This builder class includes a key and a value parameter pair. This key value pair together is called an item. This item adds the service, the provider, the aspect key, the operation key and the input parameters to the context. This context is passed as an object to execute the API method call. This mapping is common and is applied for all the services supported by the multi-cloud storage abstraction API. The table below explains the mapping of parameters into a single instance called context.

```
Context context = new Context();
context = addServiceContext(context);
context = addServiceProviderContext(context);
context = BlobService.addParameters(
   IConstants.ASPECT\_KEY,
   IConstants LEVEL-2\_STORE, context);
context.addItem(new Item(
   IConstants.OPERATION\_KEY,
   IConstants.OPERATION\_CREATE));
context = BlobService.addParameters(
   IConstants.STORE\_NAME,
   storeName, context);
```

A Plug-in Framework for Extensibility. Following API design principles, a developer should not have visibility of the underlying low-level abstraction classes, interfaces and methods. This means that a future extension can support

new features and services, but the framework should not have to be redesigned or its behavior changed. The framework should act like a plug-in for any new features, services or providers. This is achieved in the design by applying a strict coding rule where in Java the abstract class cannot be instantiated, it can only be inherited.

The design dictates that the level-1 layer, which implements the lower-level API methods, is extended from the AbstractConnector class. This abstract class must implement the interface IConnector and all of its associated methods. This is because if a class is abstract, then, by definition, it is required to create subclasses of its instance. The subclasses will be required by the compiler to implement any interface methods that the abstract class has left out. Less effort is required to extend the API because the framework remains unchanged.

Manager Interface Layer for Multi-Service Support. The limitation at the level-2 layer is the separation of the user level request to specifically distinguish between different API methods provided by the same provider. For example, Azure provides storage table and DocumentDB services. Similarly, AWS provides DynamoDB and SimpleDB. To overcome this, an interface component called manager has been implemented. The manager is responsible for identifying the corresponding "aspect-key" that is encapsulated within the context parameter.

There are two types of managers, namely, the store manager and the table manager. The composite object Level-2 aspect is the higher level of abstraction. Since Level-2 helps to identify the differences between the services, the store manager interface is added in this layer, which splits the request to either blob or file or table service at core object level-1. Table manager is used in a similar way. For example, the table service at Level-1 has two APIs, the DynamoDB and SimpleDB, supported by one provider, AWS. In order to differentiate between the services, an interface component called manager was added to identify the common method name.

The design dictates the relationship between the abstract class and the core logic of the level-2 aspect using the context parameter.

5.4 Apache jclouds and Design Patterns

Apache jclouds is an open source library available in Java and Clojure, which supports several major cloud providers. The jclouds library offers both a portable abstraction framework as well as cloud-specific features. The main aim of jclouds is to deal with errors, concurrency and cloud complexity. Our multi-cloud storage abstraction layer was designed using some of the design concepts and patterns of jclouds.

The jclouds Architecture. The jclouds framework consists of a portable abstraction layer called 'View', which is responsible for splitting the service type and cloud provider. A 'View' is connected to a provider-specific API or library

driven API. The Context Builder class maps the context object along with its parameters. The parameters include provider class object, view, API metadata and provider metadata. This object will be bound as a *singleton object* called Context and it is passed to the context builder. The API Metadata class populates friendly names for the key, which has two values – the type and the view information. The Service Registry acts like a manager, which is responsible for holding the key to connect to a provider's class. The framework implements a *builder pattern* for request and response, which connects to a backend API, along with authentication.

The jclouds library caters for blob services and compute services. An analysis was carried out to understand the features and purposes of each class in the jclouds storage functionality. The method calls were investigated to compare them across other multi-cloud libraries. The following code block outlines the jclouds library code for calling a context for an Azure blob. It uses the context builder class. The basic concept of abstraction used in the jclouds library is based on the builder pattern of software design. A context with service provider Azure that offers the portable BlobStore API:

```
BlobStoreContext context =
   ContextBuilder.newBuilder("azureblob")
   .credentials(storageAccountName, storageAccountKey)
   .buildView(BlobStoreContext.class);
```

6 Discussion

Maintainability and extensibility are objectives that need to be evaluated here. We actually have discussed these throughout the architecture discussion in the previous section. In a summarising discussion, we return here to a few important concerns such as establishing testability and maintainability through suitable design patterns, e.g., the dependency injection pattern, to point out benefits. Other aspects have already been discussed throughout Section 5.

6.1 Patterns and their Quality Implication.

From the above code in the previous section, it can be clearly seen that jclouds gets a separate instance for each provider's class and, in some cases, it makes direct REST calls to the underlying provider API. The programming style in the jclouds library follows the *dependency injection software design pattern*. It uses two *programming styles*: 1) Google Guice (a Google library alternative to Spring) and 2) Guava (which supports transformation, concatenation and aggregation for storage services).

Dependency Injection avoids code duplication, is unit-testable and modular. It essentially allows injecting of the service class instead of calling the API service method, by writing custom code and connecting them at run time, which avoids recompiling. Custom code instantiates an object for each service and provider.

6.2 Testability and Extensibility.

The jclouds framework uses dependency injection which makes reference to an object before it will proceed for execution. Implementing dependencies by constructors, using the 'new' constructor may result in difficulties for unit testing. Performing dependency injection using a *factory method* is a traditional solution to the testing problem. This process is also known as indirect dependency, where the factory method is realised by having an interaction class between the client class and the service class. It was considered that the use of too many interaction classes would make the code more complex and result in tight coupling between the abstraction layers. This would hide the definition of abstraction and furthermore, it would not facilitate the future extension of the library.

According to the principles of API design, there should be a small number of functionalities shared across the entire cloud provider API. This has been achieved in the abstraction design used.

7 Conclusions

Cloud service brokerage aims at customising or integrating existing services or making them interoperable. We have developed an integration broker following the classification schemes in [11, 12]:

- the main purpose is intermediation between cloud consumers and providers to provide advanced capabilities (interoperability and portability),
- it builds up on an intermediary/broker platform to provide a marketplace to bring providers and customers together,
- the broker system type is a multi-cloud API library.

We have focussed here on a broker solution for cloud storage service providers [1] to implement a joint interface to allow

- easy portability and migration for the user,
- easy extensibility for the broker provider.

This broker solution enables through the joint API also the opportunity for a cloud storage user to easily migrate between

While many multi-cloud APIs do exist, we have focussed here on the construction of a broker API. Again, ontologies have been used before, but we demonstrate here how a layered ontology and a corresponding layered architecture together with the use of appropriate design patterns can better help to achieve extensibility and efficiency of the implementation. The selection of design patterns has a significant impact on the testability, maintainability and extensibility of the layered architecture we have developed here.

We plan to extend the broker by adding further services by other providers to empirically verify the extensibility of the library. A more long-term usage beyond some performance testing on the provider services that we have conducted, should also help to better judge the maintainability in addition to the expected positive affect from the pattern application. More work could also go into more uniform specification of cloud services towards more standardisation [31, 21].

Acknowledgements

This work was partly supported by IC4 (Irish Centre for Cloud Computing and Commerce), funded by EI and the IDA.

References

1. S. Ried: Cloud Broker – A New Business Model Paradigm. Forrester. 2011.
2. D. Benslimane, S. Dustdar, A. Sheth: Services Mashups - The New Generation of Web Applications. Internet Computing, vol.12, no.5, pp.13-15, 2008.
3. D. Bernstein, E. Ludvigson, K.Sankar, S. Diamond, M. Morrow: Blueprint for the Inter-cloud: Protocols and Formats for Cloud Computing Interoperability. Intl Conf Internet and Web Appl and Services. 2009.
4. R. Buyya, R. Ranjan, R.N. Calheiros: Intercloud: Utility-Oriented Federation of Cloud Computing Environments For Scaling of Application Services. Intl Conf on Algorithms and Architectures for Parallel Processing , LNCS 6081. 2010.
5. Cloud Standards. http://cloud-standards.org/. 2017.
6. ETSI Cloud Standards. http://www.etsi.org/newsevents/ news/734-2013-12-press-release-report-on-cloudcomputing-standards. 2017.
7. C. Fehling, R. Mietzner: Composite as a Service: Cloud Application Structures, Provisioning, and Management. Information Technology 53:4, pp. 188-194. 2011.
8. C. Pahl, P. Jamshidi, D. Weyns: Cloud architecture continuity: Change models and change rules for sustainable cloud software architectures. Journal of Software: Evolution and Process 29(2). 2017.
9. C. Pahl. P. Jamshidi, O. Zimmermann: Architectural Principles for Cloud Software. ACM Transactions on Internet Technology. 2017.
10. Forrester Research: Cloud Brokers Will Reshape The Cloud. 2012. http://www.cordys.com/ufc/file2/cordyscms sites/download/09b57cd3eb6474f1fda 1cfd62ddf094d/pu/
11. F. Fowley, C. Pahl, L. Zhang: A Comparison Framework and Review of Service Brokerage Solutions for Cloud Architectures. 1st International Workshop on Cloud Service Brokerage. 2013.
12. F. Fowley, C. Pahl, P. Jamshidi, D. Fang, X. Liu: A Classification and Comparison Framework for Cloud Service Brokerage Architectures. IEEE Transactions on Cloud Computing. 2017.
13. M. Javed, Y.M. Abgaz, C. Pahl: Ontology change management and identification of change patterns. Journal on Data Semantics, 2(2-3): 119-143. 2013
14. S. Garcia-Gomez et al.: Challenges for the comprehensive management of Cloud Services in a PaaS framework. Scalable Computing: Practice and Experience 13(3). 2012.
15. D.M. Elango, F. Fowley, C. Pahl: Using a Cloud Broker API to Evaluate Cloud Service Provider Performance. Proceedings CloudWays'2017 Workshop. 2017.
16. Gartner - Cloud Services Brokerage. Gartner Research, 2013. http://www.gartner.com/it-glossary/cloud-servicesbrokerage- csb
17. N. Grozev, R. Buyya: InterCloud architectures and application brokering: taxonomy and survey. Software: Practice and Experience. 2012.
18. C. Pahl, P. Jamshidi: Microservices: A Systematic Mapping Study. Proceedings CLOSER Conference, Pages 137-146. 2016.

19. D. Taibi, V. Lenarduzzi, C. Pahl: Processes, Motivations and Issues for Migrating to Microservices Architectures: An Empirical Investigation. IEEE Cloud Computing. 2017 (accepted for publication).
20. C.N. Hofer, G. Karagiannis. Cloud computing services: taxonomy and comparison. Journal of Internet Services and Applications, 2(2), 81-94. 2011.
21. IEEE Cloud Standards. http://cloudcomputing.ieee.org/standards. 2015.
22. P. Jamshidi, A. Ahmad, C. Pahl: Cloud Migration Research: A Systematic Review. IEEE Transactions Cloud Computing. 2013.
23. jclouds. jclouds Java and Clojure Cloud API. http://www.jclouds.org/. 2015.
24. A. Juan Ferrer et al.: OPTIMIS: A holistic approach to cloud service provisioning. Future Gen Comp Syst, 28(1):66-77. 2012.
25. A.V. Konstantinou, T. Eilam, M. Kalantar, A.A. Totok, W. Arnold, E. Sniblel: An Architecture for Virtual Solution Composition and Deployment in Infrastructure Clouds. Intl Workshop on Virtualization Technologies in Distr Computing. 2009.
26. R. Mietzner, F. Leymann, M. Papazoglou: Defining Composite Configurable SaaS Application Packages Using SCA, Variability Descriptors and Multi-tenancy Patterns. Intl Conf on Internet and Web Applications and Services. 2008.
27. C. Pahl: Layered Ontological Modelling for Web Service-oriented Model-Driven Architecture. European Conf on Model-Driven Architecture ECMDA'05. 2005.
28. C. Pahl, S. Giesecke and W. Hasselbring: Ontology-based Modelling of Architectural Styles. Information and Software Technology (IST). 51(12): 1739-1749. 2009.
29. C. Pahl, H. Xiong: Migration to PaaS Clouds - Migration Process and Architectural Concerns. IEEE 7th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems MESOCA. 2013.
30. C. Pahl, H. Xiong, R. Walshe: A Comparison of On-premise to Cloud Migration Approaches. Europ Conf on Service-Oriented and Cloud Computing ESOCC. 2013.
31. M.P. Papazoglou, W.J. van den Heuvel: Blueprinting the Cloud. IEEE Internet Computing, November 2011.
32. D. Petcu et al.: Portable cloud applications - from theory to practice. Fut. Gen. Computer Systems 29(6):1417-1430. 2013.
33. Amazon Simple Storage Service (S3) Cloud Storage AWS https://aws.amazon.com/s3/
34. Dropbox https://www.dropbox.com/
35. Azure Storage - Secure cloud storage https://azure.microsoft.com/en-us/services/storage/
36. Google Drive - Cloud Storage & File Backup https://www.google.com/drive/
37. P. Jamshidi, C. Pahl, N.C. Mendonca: Pattern-based multi-cloud architecture migration. Software: Practice and Experience 47 (9), 1159-1184. 2017.
38. C. Pahl, A. Brogi, J. Soldani, P. Jamshidi: Cloud Container Technologies: a State-of-the-Art Review. IEEE Transactions on Cloud Computing. 2017.
39. C.M. Aderaldo, N.C. Mendonca, C. Pahl, P. Jamshidi: Benchmark requirements for microservices architecture research 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering. IEEE. 2017.
40. R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L.E. Lwakatare, C. Pahl, S. Schulte, J. Wettinger: Performance Engineering for Microservices: Research Challenges and Directions. Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion. 2017.

Cloud-Native Databases: An Application Perspective

Josef Spillner, Giovanni Toffetti, Manuel Ramírez López

Zurich University of Applied Sciences, School of Engineering
Service Prototyping Lab (blog.zhaw.ch/icclab/)
8401 Winterthur, Switzerland
{josef.spillner,toff,ramz}@zhaw.ch

Abstract. As cloud computing technologies evolve to better support hosted software applications, software development businesses are faced with a multitude of options to migrate to the cloud. A key concern is the management of data. Research on *cloud-native applications* has guided the construction of highly elastically scalable and resilient stateless applications, while there is no corresponding concept for *cloud-native databases* yet. In particular, it is not clear what the trade-offs between using self-managed database services as part of the application and provider-managed database services are. We contribute an overview about the available options, a testbed to compare the options in a systematic way, and an analysis of selected benchmark results produced during the cloud migration of a commercial document management application.

1 State Management in Cloud-Native Applications

Cloud-native applications (CNA) are software applications which pass down benefitial cloud computing characteristics. They use cloud platform and infrastructure services to become executable, offer their own functionality as software service interfaces, are resilient against dependency service unavailability and other incidents, scale elastically with user requests, are always available on demand and are billed with a pay-per-use utility scheme without upfront cost [2]. The inherent service orientation required for CNA favours a microservices model with explicitly stateful and stateless services. The handling of data is confined to the stateful services. These must in turn be highly available and resilient to prevent loss, corruption or delay of data operations. Databases, message queues, key-value stores, filesystems and other data access models have been analysed in prior works concerning these requirements [7, 13, 14]. The desired characteristics depend on near-instant service replication [10] which implies consistent data replication and sharding mechanisms.

Fig. 1 shows a typical topology of stateful and stateless microservices orchestrated to offer a single application as a service in a highly available and resilient manner on top of plain cloud infrastructure services. Almost all approaches rely on coordinated replication which brings self-awareness about its role (e.g., master or slave) to each microservice. Furthermore, they rely on fast-spawning service

implementations (e.g., containers or light-weight hypervisors) to achieve rapid elasticity upon request spikes and instance recovery after crashes.

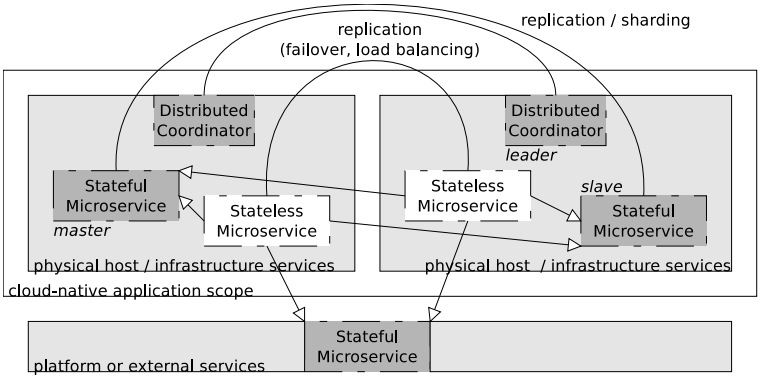


Fig. 1: Cloud-native application with internal and external data management

It is however not clear under which circumstances applications should manage data by themselves. The range of commercially offered platform-integrated stateful services is increasing. Their common value proposition can be expressed in a simplified way by *paying more to manage less*. But for a business decision, the value needs to be quantified. Due to the multitude of possible options, businesses need to obtain metrics on which such decisions can be performed. Apart from the pricing, such decisions need to account for risks and for end-to-end service provisioning quality and effort which from a software engineering perspective always includes the consideration of client-side bindings to the services.

To reduce the problem scope, we limit the research to applications which handle large structured documents in database systems. Hence, through this paper, empirical studies of how databases operated in a cloud-native context behave in commercial cloud environments are made possible. The main contribution is a testbed to measure and compare different database options from a vendor-neutral perspective. The resulting distinction between self-managed and provider-managed databases covered by the testbed is expressed in Fig. 2.

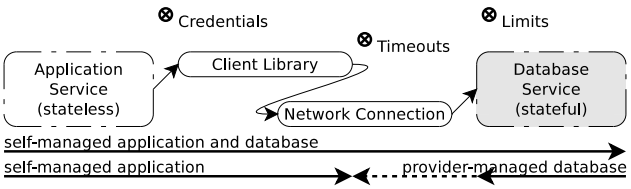


Fig. 2: Scopes of cloud-native database management

The paper is structured as follows: First, it presents the possible options for cloud-native databases, differentiating between fully managed and application-

controlled offers. Then, it defines a method to compare and rate cloud database services both on the technical and on the pricing level. The method translates into a design of a testbed whose architecture and implementation we present. Experiments we have conducted based on this method are then explained together with the obtained results. The findings from the experiments need to be interpreted in alignment with business strategies. For this reason the paper concludes with an open discussion about the strategic impact of using the testbed in a systematic way during the application engineering process.

2 Cloud-Native Database Options

Apart from conventional relational or document-centric databases, the migration of applications into the cloud and the associated new operational requirements have led to novel design choices and along with them new research challenges. Recent database models thus include encrypted, privacy-preserving and stealth databases, energy-efficient database operators and adaptive query systems over dynamically provisioned resources [8, 4]. Few of these designs have progressed beyond prototypical systems, but from an applied science perspective, we are interested in what recommendations can be given to application developers today. Hence, only conventional (relational and document-centric) database systems and services which are widely available on cloud platforms are considered along with systems commonly described as cloud-enabled or cloud-ready.

We distinguish between the choices of database hosting primarily by the responsibility. Database management systems can be managed by the application provider as part of the application (e.g. as application-controlled container), outside of the application scope itself (e.g. as a separate virtual machine whose autoscaling is determined by cloud facilities), and as fully cloud-managed service, typically named Database-as-a-Service (DBaaS). Our focus is on *cloud-native database* (CNDB) options which adhere to expectations from cloud application developers such as elastic scaling, resilience against unexpected issues, flexible multi-tenancy isolation and high performance at low price.

2.1 Self-Managed Database Systems and Microservices

The widespread proliferation of open source database management systems has led to the inclusion of these systems into software applications. The application logic then controls the lifecycle of the database, launches and terminates it as needed, and directly accesses it, often without authentication or through a single user account. Tenants in the application are in this case mapped to the database through identifiers or unprivileged separation such as tables or columns.

Cloud-native applications are often decomposed into horizontally scalable microservices where all instances are of equal importance in a peer structure. Only few database systems are currently mirroring this ability. Many still require a master-slave setup where the master instance needs to be launched before the

slave instances and must never fail, or variants thereof with multi-master replication. We analyse selected database systems concerning their use as disposable microservices in Table 1. Of these, only Crate fully conforms to this model, although a technology preview also exists for MongoDB (for master-slave replication).

Table 1: Available self-managed database microservices

Name	Relation of instances
CouchDB	master-slave and master-master replication, manual sharding
MongoDB	replica sets with master-slave replication, keyed sharding
Crate	set of peers with automated sharding upon scaling
PostgreSQL	master-slave replication, sharding through Citus
MySQL	master-slave replication, sharding through Fabric

2.2 Provider-Managed Database Services

From a cloud application perspective, it is desirable to maximise the flexibility by freely choosing among application-controllable software and managed services for the assumed database interface. Despite efforts to standardise the interfaces for database-as-a-service (DBaaS), the implementation differences are significant enough to warrant the propagation of information about the underlying database system. For instance, a developer may know how to write SQL statements but can optimise them and avoid pitfalls when knowing that the engine behind the SQL interface is in fact a MariaDB 10.2 with the XtraDB storage engine. This knowledge should be conveyed and flexibly interpreted using discoverable service descriptions, but in practice, it is often tightly coupled to the application. Furthermore, database interface and implementation options provided in the commercial cloud space vary significantly. Table 2 compares the availability of database interfaces at six public cloud (platform) providers from two countries, USA and Switzerland. Implementations marked with asterisk are available as open source and thus allocatable for local testing by application developers prior to paying for the cloud deployment.

Despite multi-database service offers by most providers, the table is sparse. This means that vendor lock-in risks need to be assessed. Furthermore, the pricing of DBaaS differs for offers with the same interface. For instance, MongoDB services are offered by Microsoft (as interface adapter to CosmosDB) and by the Swisscom Application Cloud (AC). The Swisscom offer excluding high availability starts at CHF 12 per month including 1 GB storage and 256 MB RAM. The equivalent Azure offer (hosted in Europe-West) starts at CHF 134 but includes 10 GB storage and 5 DTUs, a custom unit expressing the processing power. For an application engineer who wants to process a data volume of 1 GB, it is not clear if the cheaper offer would be performance-wise on par without further ex-

Table 2: Available provider-managed database services

Amazon Web Services	Google Cloud	Microsoft Azure	IBM Bluemix	APPUiO	Swisscom AC	Application Interface	Implementation
X[1]	X[2]					SQL	MySQL *
				X	X		MariaDB *
			X	X			PostgreSQL *
							Aurora
							Oracle DB
		X[3]					SQL Server
			X				DB2
		(X)[4]		X	X	JSON QL or similar (Mango etc.)	MongoDB *
			X[5]				CouchDB *
X							DynamoDB
		X					CosmosDB
		X				other	TableStorage
	X						Datastore
	X						BigTable (*)
X[6]	X[7]	X			X		Redis *
Notes: 1: RDS, 2: Cloud SQL, 3: Database Service, 4: CosmosDB adapter, 5: as Cloudant NoSQL, 6: as ElastiCache; 7: via external RedisLabs service							

periments. There are detailed studies on cloud database services in general [11]. In contrast, our focus is on their suitability for cloud-native applications.

3 Comparison Method and Testbed

The automatable comparison of databases is rooted in two main characteristics: performance and resilience. Other metrics such as price and isolation can be derived from trace data in conjunction with external information. Several queries and transactions are run to measure the performance through an application-specific benchmark. It includes the preparation of structures (tables, collections), individual inserts, bulk inserts, queries and deletions. Furthermore, the availability is measured and in the case of self-managed database services actively impeded by controlled interference and termination, leading to data about the resilience.

As our chosen approach is to provide a testbed to compare database options, its functional and non-functional requirements need to be defined first. The functional requirements are:

1. The testbed must run itself in the target environment of the cloud-native application to yield realistic metrics with simple queries and complex transactions.
2. Both self-managed and provider-managed database services need to be supported.
3. The testbed operator must be able to choose the dataset under test, either an existing one or a synthetic one which is generated as part of the operation.

The non-functional requirements are:

1. The scale of testing needs to be configurable to balance representative and timely results. Therefore, the runtime needs to be chosen to range from mere minutes to multi-day sampling.
2. All tests need to be idempotent to allow for repetitions and statistical detection of anomalies.

3.1 Testbed Architecture and Implementation

The testbed architecture is derived from the requirements. To correlate with cloud-native applications, a containerised approach is taken. Both the testbed itself, with its performance benchmark and resilience calculation parts, and all self-managed database services are launched as container compositions. Fig. 3 visualises the technique of how the experiments are conducted by using Docker Compose as orchestrator of containers. One container contains a performance benchmark application, another one a fault provocation application, two stateful containers serve as persistent input and output volumes for the reference dataset and the results respectively, and additional containers spawn the database systems. The testbed containers allow for parameterisation through environment variables to override any values in the internal configuration file. The most important properties include binding metadata and credentials. Furthermore, the testbed supports five configurable multi-tenancy isolation levels.

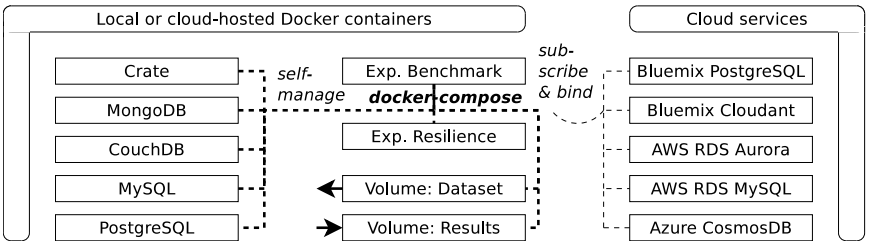


Fig. 3: Orchestrated containers and services as part of the experiment setup

Our implementation of this architecture is called CNDBbench, focusing on the benchmarking part while also containing the resilience part. It is consisting of Python classes for all supported database interfaces and the Docker image generation scripts, and is made available as open source software for use in other migration cases (see Repeatability).

3.2 Testbed Preparation: Document Management Scenario

Each instance of the testbed needs to be prepared according to application-specific needs. The guiding objective of our research has been to analyse database options for the class of cloud-native document management applications. Their requirement is storing millions of documents (e.g. scanned PDFs of dozens of MB in size) along with document metadata such as ownership, permissions, audit trails and searchable full text determined by OCR prior to insertion. From the application perspective, the design then involves stateful (database) components which are realised as bindings to database services or instances of application-controlled database microservices. Fig. 4 demonstrates a document management scenario and the possible realisation options.

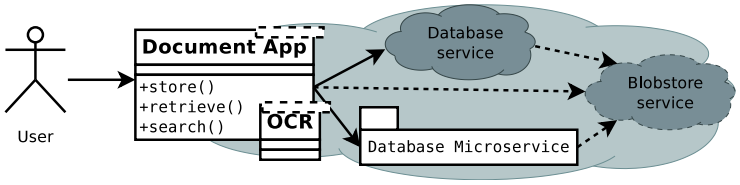


Fig. 4: Document management scenario

The reference dataset to evaluate the database choices consists of 100,000 generated entries which correspond to an actual domain-specific dataset with scanned newspaper articles. With associated metadata such as origin and access control lists, there are 1.4 million entries in total. The medium-sized data with large blob documents and structured metadata is representative for the domain of document management in the cloud through databases; alternative hybrid designs using blob storage are not considered in the present scenario. The following operations are performed to get both performance and deviation metrics: insertion of data, search and retrieval of partial data. This selection matches transactions in typical document management applications where updates and deletions happen rather sporadically.

3.3 Testbed Operation

Once the testbed is prepared, it needs to be operated in a way which most closely corresponds to the eventual operation of the application. Specifically, network delays and latencies as well as microservice execution technologies need to be properly reflected. Fig. 5 shows seven testbed configurations which correspond to all possible combinations of how to manage application data in the cloud. More variability is added by defining for the cases of application-managed databases where to physically store the data. Our research assumes attaching volume containers whereas provider-managed storage areas would be another option.

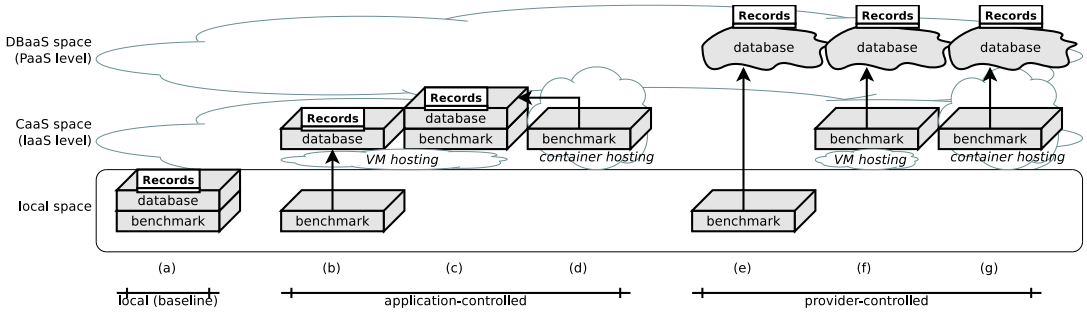


Fig. 5: Combinations of local, application-managed and provider-managed containers with application-managed and provider-managed databases

4 Selected Results

This section reports on results we have obtained from running the testbed in some of the explained operational combinations using the document management dataset. The research on the figurative *cloud-nativeness* of databases have been conducted with experiments targeting the desired technical properties of the specific application domain of document management. In total, 28 experiments have been performed and recorded, showing the versatility of CNDBbench. Selected results concerning performance, multi-tenancy flexibility and pricing will be reported. Apart from the results described here due to interesting observations, all experiments and results are analysed and described in a technical appendix to this paper (see Repeatability).

Five relational and document database systems from Table 2 have been selected for the study of the first group. They are briefly summarised in Table 3. Among those, PostgreSQL and MySQL are relational database systems (albeit with recently added JSON document processing capabilities) and have been available in early versions since the mid-1990s. CouchDB and MongoDB are often-cited representatives for document-centric systems which appeared in the late 2000s. Crate is the most recent system, created in 2014, whose focus on cloud deployments is stressed by masterless distributed operation and automatic node recovery in combination with a standard SQL-over-HTTP interface. It offers a mixed document/column store. All five systems have subtle differences in how they shard (and replicate) data.

For the second group, summarised in the bottom half of the table, three database service providers have been chosen: Amazon Web Service's Relational Database Service (RDS) with the Aurora implementation, which is a custom storage engine, in addition to the stock MySQL with its InnoDB, MyISAM and other default engines, IBM's Cloudant NoSQL and PostgreSQL service on its Bluemix platforms, which as the name suggests are a document store and a relational database, respectively, and Azure's CosmosDB née DocumentDB. An interesting observation is that even more sharding options are present which affect how well data can be managed by cloud-native applications. Interestingly, Aurora despite being a cloud service does not offer sharding for horizontal scal-

Table 3: Evaluated database system software and cloud services

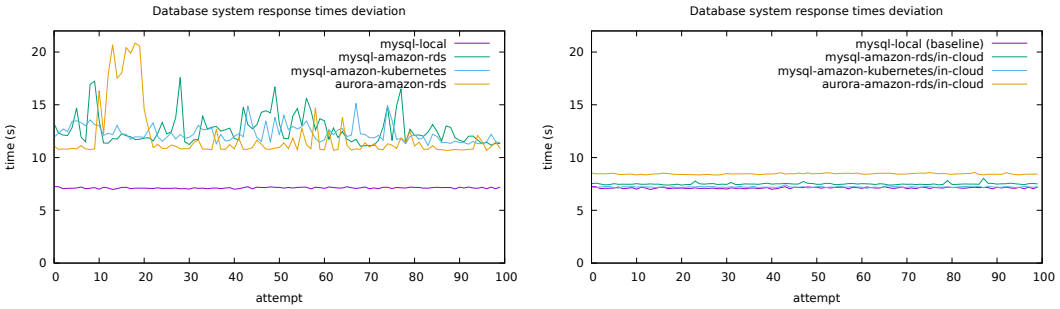
Software/Service	Data model	Runtime	Distribution
CouchDB	document	Erlang	create-sharding
MongoDB	document	C++	config-sharding
Crate	mixed-model	Java	auto-sharding
PostgreSQL	relational	C	master-sharding
MySQL	relational	C, C++	fabric-sharding
AWS RDS Aurora	relational	MySQL	read-replicas
AWS RDS MySQL	relational	MySQL	read-replicas
Azure CosmosDB	document	DocumentDB	key-sharding
Bluemix PostgreSQL	relational	PostgreSQL	failover-replicas
Bluemix Cloudant	document	CouchDB	none

ability. More variety is available at other providers, for instance Azure offering key-sharded data in CosmosDB which would otherwise resemble Cloudant.

4.1 Database Performance

The first experiment compares the deviation of response times as measure of instability between a local database system and a database system or service in the cloud, represented by AWS. A complex document management transaction consisting of six individual queries was performed with MySQL first as this system is reflected in the largest variety of cloud hosting options. The benchmark itself ran both on the local machine and as close as possible to the database, i.e. with high affinity in the cloud. Fig. 6a shows that the local queries are much faster and their response time more predictable than those of the cloud counterpart when the benchmark runs locally and thus all queries need to traverse the wide-area network. Fig. 6b contrasts the results with the affine benchmark. All such measurements are suffixed with */in-cloud*. The trivial comparison shows that a local benchmark with a local MySQL system performs equal to a Kubernetes-hosted benchmark and MySQL container pair, as both communicate via local link. As soon as the provider's services are involved, this translates into a local-area network transmission within one hosting region.

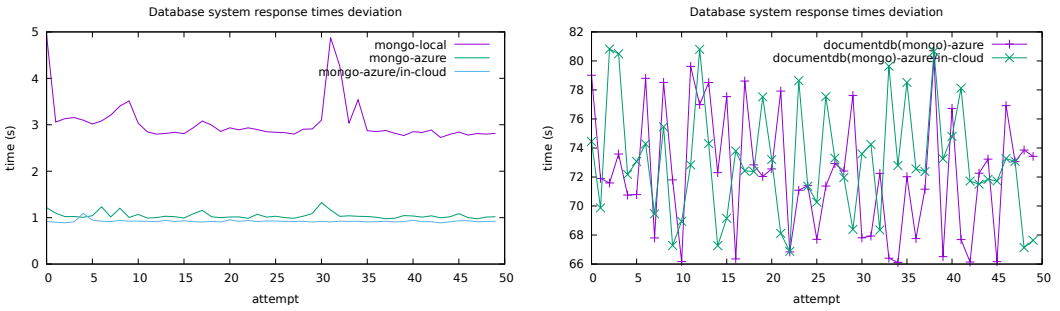
In Fig. 7a, a different set of queries was tested with MongoDB, hence the different absolute times and network delay effects. Nevertheless, the cloud-hosted database container shows a higher stability in response times with both local and cloud benchmark, while the latter also has a lower response time as expected from the observation of MySQL. The interesting difference is that the response time deviations are high for local MongoDB queries but low for local MySQL queries which suggests that not only the network influences the variation in response times. In contrast, Fig. 7b reports on the same experiments using the MongoDB adapter for CosmosDB which was conducted over two non-consecutive days. In both the local and cloud-hosted benchmark cases, the latter using an Azure VM, the performance is relatively stable within one day, vary-



(a) Local benchmark (b) Cloud benchmark

Fig. 6: Query times for MySQL

ing a lot between the days (about 33%), and extremely low compared to the native MongoDB counterparts. Additionally, Fig. 8 compares two database services from Bluemix to complete the variations in engines, providers, services and benchmark locations. The interesting observation is that not only are the absolute response times of PostgreSQL strictly below the ones of MySQL ($\bar{rt} = 0.92$ vs. 6.23), their deviation is also a lot smaller ($\sigma = 2.60$ vs. 28.32).



(a) MongoDB self-managed instances (b) CosmosDB with MongoDB interface

Fig. 7: Query times for MongoDB/CosmosDB, both local and cloud benchmarks

4.2 Database Multi-Tenancy

Data management is affected by the level of isolation between the tenants in a multi-tenant database service setup. Fig. 9 represents the model of matching isolation level to estimated performance and cost. For three out of the five different levels, we have measured the actual behaviour with three different implementations each.

Fig. 10 contains the corresponding results. The multi-threaded implementation (MT) takes longer per thread to return the results but all threads return close to each other, leading to a speedup of 22.5%, 41.9% and 59.6% over the

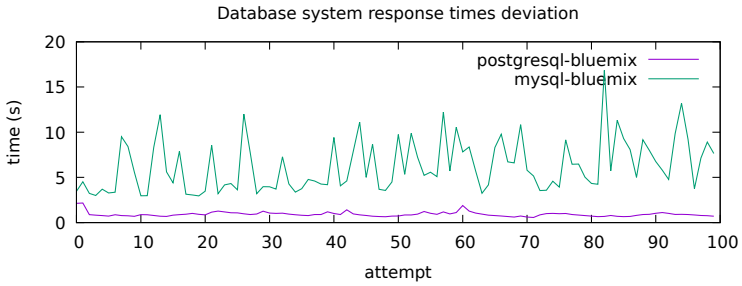


Fig. 8: Query times for MySQL and PostgreSQL, local benchmark

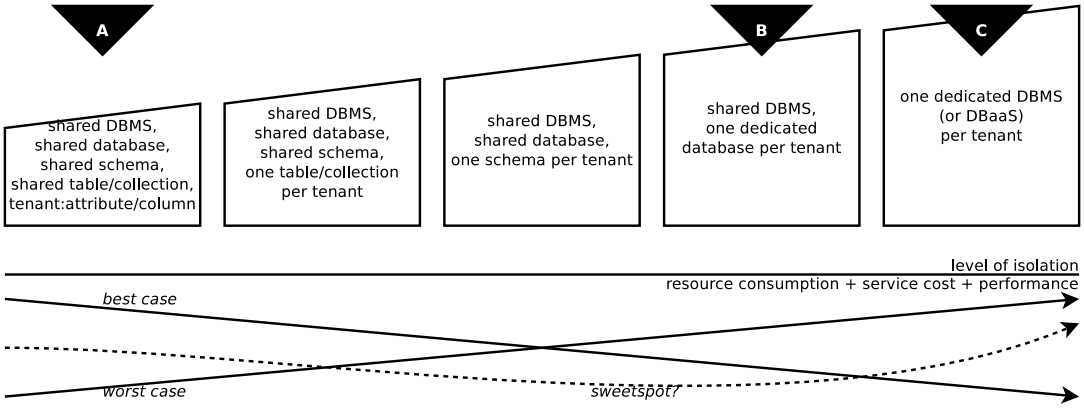


Fig. 9: Model of flexible multi-tenancy configurations for database services, with MongoDB

single-threaded implementation of A, B and C, respectively. Option C is the fastest and most isolated option, but does not represent an unconditional overall sweetspot due to also being the most expensive one.

4.3 Database Pricing

Of interest to the application provider is the total cost of provisioning in relation to a quality of experience which allows for a surplus-generating revenue. Our findings indicate that there is no clear price advantage of self-managed containers on the SaaS level versus a comparable DBaaS option, or vice-versa, when taking replicated containers for higher resilience into account. From a methodic point of view, we derive an unquantified graphical representation of pricing in relation to performance, availability/resilience, reliability, multi-tenancy and scalability as shown in Fig. 11 and propose to derive a comparison tool for application engineers.

5 Findings and Recommendations

As the selected results have shown, a general statement about a single best database option will not be possible, and a sharp definition of CNDB remains im-

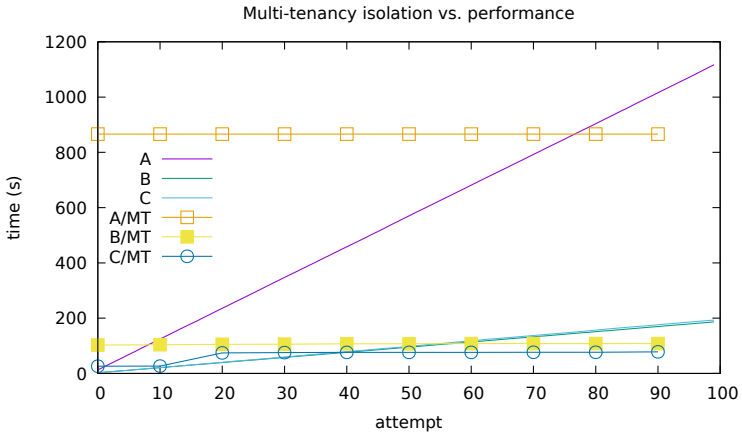


Fig. 10: Results for multi-tenancy options A, B and C with and without multi-threading

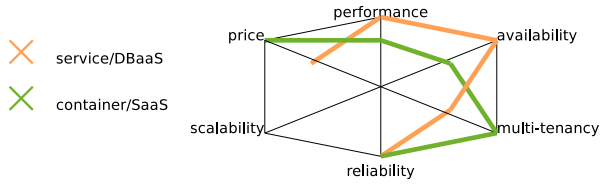


Fig. 11: Spider graph for pricing trade-offs, sampled for MySQL at Google; outside = best

possible. Our general recommendation is therefore that tools such as CNDBbench should be used in cloud application migration projects to produce metrics upon which selection decisions can be based.

Several systems and services have undocumented or undiscoverable limitations which can be revealed by systematic testing as is the case with CNDBbench. For instance, Crate only returns up to 10,000 rows by default and requires a `LIMIT` clause to return more. Azure CosmosDB limits the maximum requests to 1000 per second, which can be increased to 10,000, and requires the activation of further instances to grow beyond, despite low load on the database. Several protocols and client-side libraries are setting up timeouts. Some are merely difficult to deactivate, others even impossible, like the 20 second query timeout when inserting many records through PyMongo.

For the mentioned limitations, we recommend a discoverable description of these properties in addition to more complete documentation [6]. For the construction of future applications, assuming more maturity and choice in containerised database systems, we recommend auto-clustering microservices as currently implemented for Crate. In any case, the economics of self-managed instances depends to a large degree on the business background, including the skills and qualifications of the application engineers. In tech-savvy companies, self-managed

database containers running on top of virtual machines using container management frameworks are recommended.

6 Discussion and Conclusion

We discuss our findings in the context of recent publications about both cloud-native databases and database characteristics in the cloud in general.

Szczyrbowski and Myszor present a behaviour comparison between the Oracle Database Schema Service which offers an HTTP interface [13] and the local 11g equivalent. Their main focus is on performance stability, minimising deviations in query times for three operations: INSERT, UPDATE and SELECT. Their approach is comparable to ours apart from updates and technological choices. The findings suggest that the cloud service has a much lower deviation apart from also being (presumably due to opaque hardware differences) faster in the worst, average and best case. We were able to reproduce this for MongoDB but not for MySQL, and therefore assume that their findings cannot be generalised.

Another performance comparison is authored by Seriatos et al. [12]. The focus is on three database systems – MongoDB, Cassandra and HBase – in the BONFIRE cloud testbed. Cost and scaling are not discussed. The YCSB benchmark is used. The findings tell that each of the system performs differently depending on the workload which implies two future work directions: The first, mentioned by the authors, is the tuning of parameters; the second, added by us, is the design of adaptive multi-database connectivity as the next evolutionary step for CNDBs.

The focus on cost is set by Mian et al. in an analysis of resource configuration using the TPC-C/E/H benchmarks in three application scenarios [9]. While the authors focus on AWS EC2, the DBaaS services of the same provider are not considered. A similar aim is conveyed in the work by Floratou et al. albeit with a critical look at unpleasant surprises in terms of financial risks when using DBaaS [5]. The findings are that more expensive hourly services may turn out more cost-effective overall, which is substantiated with observations of MySQL and SQL Server running on local hardware. The authors propose a benchmark-as-a-service for application developers (as database users). To cover the scaling and resilience characteristics which are important in a cloud setting, Bagui et al. look at sharding techniques and propose an implementation [1]. The work is demonstrated with MySQL and extends to other engines. Costa et al. examined partial database migration to the cloud [3]. The migration path in this work is from local PostgreSQL to AWS DynamoDB without giving up the former by adding a transparent adapter to the application. The finding is that scalability bottlenecks can be circumvented by offloading data to DynamoDB. While we have not analysed the same system, our results with non-ACID confirm this observation.

Table 4 summarises which of the cloud database properties were covered by related works and whether our findings agree (☑) or disagree(✪) with them.

When the results are not clear, the need for future experimental research (⚙) is shown instead. The lack of a reusable testbed from the related work is evident.

Table 4: Related work comparison

Study	Performance	Scalability	Resilience	Tenancy	Price	Testbed
Szczyrbowski et al. [13]	✖					
Seriatos et al. [12]	⚙					
Mian et al. [9]					⚙	
Floratou et al. [5]					⚙	
Bagui et al. [1]		⚙	⚙			
Costa et al. [3]		☑				

We conclude that cloud-native databases are a challenging topic in need of more formal expressions concerning their configuration and characteristics and of more experiments. We suggest that future research should be directed towards a holistic approach of assessing flexible database options in the cloud which involve self-hosted data containers, blob storage services and DBaaS.

Repeatability

Our benchmark implementation, CNDBbench, is publicly available to repeat our experiments. For reference and reproducibility of the results, the experiment setup including hardware specifications and instructions is given in detail in a raw open science notebook which is made available together with a technical appendix due to the page number limitation. The notebook also contains reference results, additional experiments and findings concerning resilience, scalability and pricing[1,2]. We encourage the critical examination and re-use of the datasets.

Acknowledgements

This research has been funded by the Swiss Commission for Technology and Innovation (CTI) in project ARKIS/18992.1. It has also been supported by an AWS in Education Research Grant, an IBM Academic Initiative for Cloud offer, a Microsoft Azure Research Award and a Google Cloud credit, all of which helped us to conduct our experiments on public commercial cloud environments.

[1] CNDBbench: https://github.com/serviceprototypinglab/cndbbench
[2] CNDBresults: https://github.com/serviceprototypinglab/cndbresults

References

1. Bagui, S., Nguyen, L.T.: Database Sharding: To Provide Fault Tolerance and Scalability of Big Data on the Cloud. International Journal of Cloud Applications and Computing (IJCAC) 5(2), 36–52 (2015)
2. Brunner, S., Blöchlinger, M., Toffetti, G., Spillner, J., Bohnert, T.M.: Experimental Evaluation of the Cloud-Native Application Design. In: 4th International Workshop on Clouds and (eScience) Applications Management (CloudAM). Limassol, Cyprus (December 2015)
3. Costa, C.H., Maia, P.H., Mendonça, N.C., Rocha, L.S.: Supporting Partial Database Migration to the Cloud Using Non-intrusive Software Adaptations: An Experience Report. In: 4th ESOCC. CCIS, vol. 567. Taormina, Italy (September 2015)
4. Costa, C.M., Leite, C.R.M., Sousa, A.L.: Efficient SQL adaptive query processing in cloud databases systems. In: IEEE EAIS. pp. 114–121. Natal, Brazil (May 2016)
5. Floratou, A., Patel, J.M., Lang, W., Halverson, A.: When Free Is Not Really Free: What Does It Cost to Run a Database Workload in the Cloud? In: Topics in Performance Evaluation, Measurement and Characterization – Third TPC Technology Conference (TPCTC). LNCS, vol. 7144. Seattle, Washington, USA (August 2011)
6. Frey, S., Hasselbring, W., Schnoor, B.: Automatic Conformance Checking for Migrating Software Systems to Cloud Infrastructures and Platforms. J. Softw. Evol. and Proc. 25(10), 1089–1115 (October 2013)
7. Goldschmidt, T., Jansen, A., Koziolek, H., Doppelhamer, J., Breivold, H.P.: Scalability and Robustness of Time-Series Databases for Cloud-Native Monitoring of Industrial Processes. In: 7th IEEE International Conference on Cloud Computing (CLOUD). pp. 602–609. Anchorage, Alaska, USA (July 2014)
8. Götz, S., Ilsche, T., Cardoso, J., Spillner, J., Kissinger, T., Aßmann, U., Lehner, W., Nagel, W.E., Schill, A.: Energy-Efficient Databases using Sweet Spot Frequencies. In: 1st International Workshop on Green Cloud Computing (GCC). pp. 871–876. London, UK (December 2014)
9. Mian, R., Martin, P., Zulkernine, F.H., Vázquez-Poletti, J.L.: Cost-Effective Resource Configurations for Multi-Tenant Database Systems in Public Clouds. International Journal of Cloud Applications and Computing (IJCAC) 5(2), 1–22 (2015)
10. Nguyen, H., Shen, Z., Gu, X., Subbiah, S., Wilkes, J.: AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In: 10th International Conference on Autonomic Computing (ICAC). pp. 69–82. San Jose, California, USA (June 2013)
11. Sakr, S.: Cloud-hosted databases: technologies, challenges and opportunities. Cluster Computing 17(2), 487–502 (2014)
12. Seriatos, G., Kousiouris, G., Menychtas, A., Kyriazis, D., Varvarigou, T.A.: Comparison of Database and Workload Types Performance in Cloud Environments. In: First International Workshop on Algorithmic Aspects of Cloud Computing (ALGOCLOUD). LNCS, vol. 9511, pp. 138–150. Patras, Greece (September 2015)
13. Szczyrbowski, M., Myszor, D.: Comparison of the Behaviour of Local Databases and Databases Located in the Cloud. In: 12th International Conference on Beyond Databases, Architectures and Structures. Advanced Technologies for Data Mining and Knowledge Discovery. CCIS, vol. 613, pp. 253–261. Ustroń, Poland (May 2016)
14. Wiese, L.: Advanced Data Management for SQL, NoSQL, Cloud and Distributed Databases. DeGruyter/Oldenbourg (2015)

Using a Cloud Broker API to Evaluate Cloud Service Provider Performance

Divyaa Manimaran Elango[1], Frank Fowley[1], and Claus Pahl[2]

[1] IC4, Dublin City University, Dublin, Ireland
[2] SwSE, Free University of Bozen-Bolzano, Bolzano, Italy

Abstract. We introduce here a cloud service broker that implements a multi-cloud abstraction API – with the aim to use it for performance comparisons between different services. Our broker is a multi-cloud storage API that supports a number of provided storage services. The API library offers three services, namely File service, Blob service and Table service. While a broker normally addresses interoperability concerns, based on this architecture, a performance test application was developed here to compare between the different providers. This test configuration was used to compare a range storage operations of different services.

1 Introduction

A Cloud Broker is an intermediary application between a client and cloud provider service [10, 15]. This concept of brokerage reduces the time spent by a client in analyzing different types of services provided by different service providers [1]. This enables a single platform to offer the client a common cloud storage service. This results in cost optimization and reduced level of back-end data management requirements. We introduce here a cloud service broker that implements a multi-cloud abstraction API. Our broker is a multi-cloud storage API that supports GoogleDrive, DropBox, Microsoft Azure and Amazon Web Services as provided storage services. The API library offers three services, namely File service, Blob service and Table service. A multi-cloud storage abstraction API can facilitates the distribution of different types of cloud provider services [16]. The abstraction library allows the cloud broker to adapt to a rapidly changing marketplace. Vendor Lock-in is a major barrier in cloud computing. In order to avoid lock-in, a broker can help. A multi-cloud abstraction library can play a crucial role in such cases, where it makes it easier for the client to switch between cloud providers with different services available.

Switching or migrating between providers should be driven by quality [28, 29]. We use a broker implementation to compare the supported services [7, 13, 23] from a performance perspective [39] . While a broker normally addresses interoperability [3, 2, 4, 9], based on this architecture, a performance test application was developed here to compare between services [19]. The test app was used to compare a range storage operations across different providers.

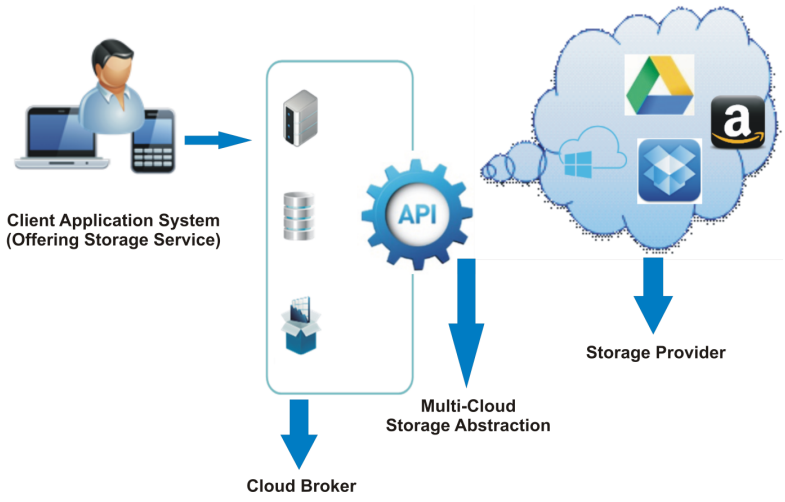


Fig. 1. Architecture of a Multi-cloud Storage Broker.

2 Selected Services and Architecture

GoogleDrive offers a cloud file storage service. The GoogleDrive service includes access to a Google API client library. *DropBox* is a file hosting service. It also enables synchronised backup and web sharing. The DropBox API is very light-weight and easy for a new user. *Azure Storage* supports blob, file, queue and table services. The API is built on REST, HTTP and xml, and can be integrated with Microsoft visual studio, eclipse and GIT. The Azure SDK provides a separate API package for each service and has the same code flow across different service APIs. *Amazon Web Service S3* is a file storage service which is built on REST and SOAP. Their SDK is available in all major development languages.

This study of providers resulted in a grouping of the cloud providers and their services as shown in the table below[3]:

Service	Azure	AWS	Google	DropBox
File	Storage File	-	GoogleDrive	DropBox
Blob	Storage Blob	AWS S3	-	-
Table	Storage Table, DocumentDB	DynamoDB, SimpleDB	-	-

Cloud services are generally provided with specifications which make it hard to use them as a common interface. Several existing multi-cloud libraries were studied during this project, including Apache Jclouds, DeltaCloud, Kloudless, SecureBlackBox, and SimpleCloud in order to adopt a successful solution template. It was decided to adopt an approach similar to the Apache Jclouds library for abstraction. Jclouds (http://www.jclouds.org/) provides cloud-agnostic abstraction. A single instance context approach for the mapping of a user request in jclouds was used in our implementation. The purpose of having each class for each provider across different levels of service was adopted from a similar design

[3] https://www.google.com/drive/; https://www.dropbox.com/; https://aws.amazon.com/s3/; https://azure.microsoft.com/en-us/services/storage/;

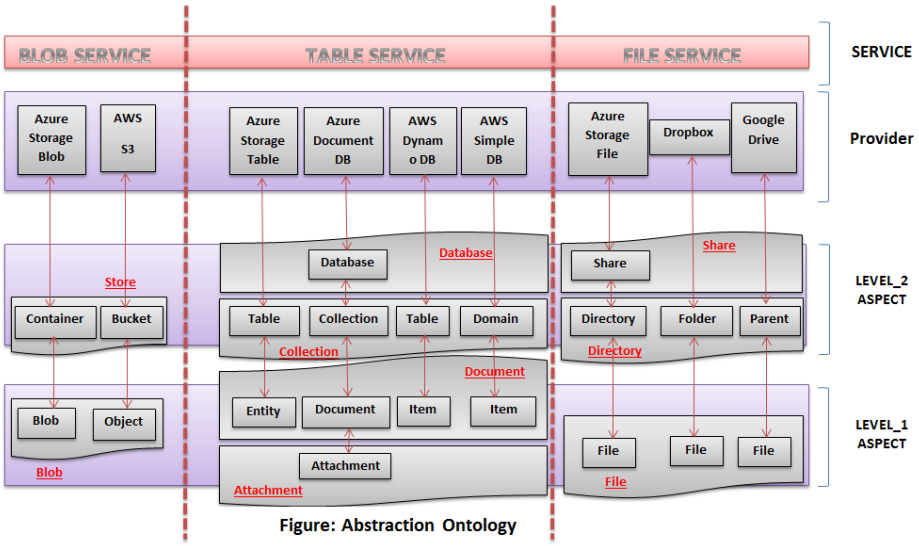


Fig. 2. Ontology-based layered broker architecture.

in the SecureBlackBox library. Our concept of including a manager interface layer at each component level is gleaned from LibCloud.

3 Broker Architecture

The main objective of designing and developing an abstraction API is to produce an effective cloud delivery model, enabling portability and interoperability as part of a cloud brokerage solution [14].

A Storage Abstraction Ontology describes the common naming and meaning approach of the abstraction API [26, 27]. The model consists of four main layers, namely Service, Provider, Level-2 (composite storage objects) and Level-1 (core storage objects).

Service: The Service layer is the top layer and is directly integrated to the user interface layer. This layer basically describes the services that the multi-cloud storage abstraction API supports. There are three services currently supported. They are Blob, Table and File service.

Provider: The Provider layer is the second layer, which is one of the context object parameters mapped to the service layer. The multi-cloud storage abstraction supports four main providers, namely Microsoft Azure, Amazon Web Services, GoogleDrive and DropBox. The corresponding services supported by the providers are shown below:

Service	Provider
Blob	Azure storage Blob; AWS S3.
Table	Azure Storage Table; Azure DocumentDB; AWS DynamoDB; AWS SimpleDB.
File	Azure Storage File; DropBox and GoogleDrive.

Level-2 Composite: Level-2 is the third layer. This layer represents the first level or higher level of composite object abstraction. This layer is service-neutral and

brings out the common naming across the providers specific functionalities. Each layer is abstracted based on the common operations and aspect of how the main function is applied in that particular service. Common naming is represented to easily categorise storage resources and group them to make the development of the coding easier.

As it is shown in the Abstraction Ontology Fig. 2, the Blob service has Store which groups Container from Azure Storage Blob and Bucket from AWS S3. The Table service has two different sub-layers - where Database belongs to Azure DocumentDB Database, and where Collection groups Table from Azure Storage Table, collection from Azure DocumentDB Collection, Table from AWS DynamoDB and Domain from AWS SimpleDB. The File service has two different sub-layers where Share belongs to Azure Storage File and Directory groups Directory from Azure Storage File, Folder from DropBox and Parent from the GoogleDrive service.

Level-1 Core: Level-1 represents the lower level of core object abstraction. This layer contains the core functionalities of a particular service across different providers. The classes in this level are extended from an abstract class called AbstractConnector. The class implements the abstract methods defined in the AbstractConnector class. The mapping from Level-2 to Level-1 is performed by an interface class called Manager. This Manager identifies the provider class by its key. Basic CRUD operations on the storage resources are included as core methods. In order to achieve these functions, each operation request should pass through the Level-2 mappings and are then mapped across the service and providers.

The Blob service has Blob which groups Blob from Azure Storage Blob and Object from AWS S3. The Table service has two different sub layers. It has Item which groups Entity from Azure Storage Table, Document from Azure DocumentDB Document, Item from AWS DynamoDB and Item from AWS SimpleDB. Also, the second sub layer Attachment belongs to Azure DocumentDB. The File service has File which groups File from Azure Storage File, File from DropBox and File from GoogleDrive.

4 Testing and Evaluation

We now desribe the test set-up and the results for the three service types blob, file and table.

The **Blob Service test** was performed on Azure Storage Blob and AWS S3. The test includes two object levels. The Level-2 represents Store (which includes container and Bucket). The Level-1 represents Blob (which includes Blob and Object). The total number of tests performed was 27. The performance test compares the operations across the service providers. Each operation was run 10 times, and the corresponding process time for each request from T1 to T10 was calculated. Each request was processed with the same blob size of 10.2MB. The result includes start time, end time, average time and total duration – Figs. 3 and 4.

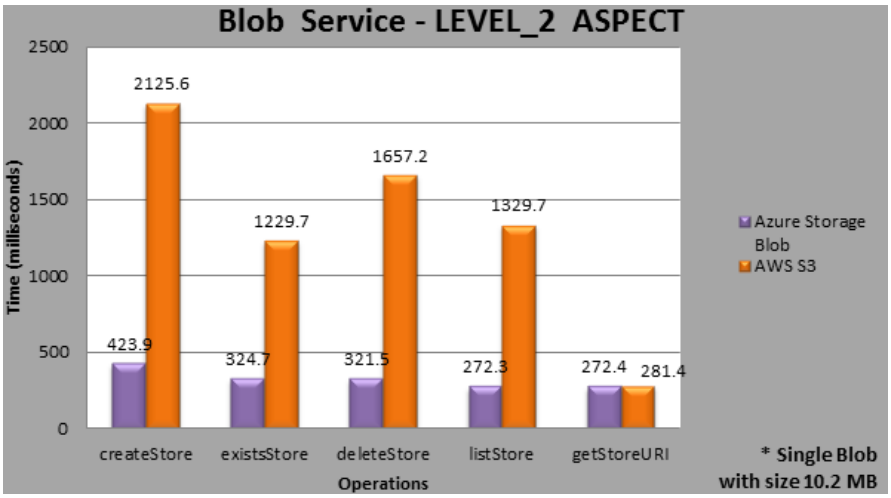


Fig. 3. Blob Service Level-2 Composite Object.

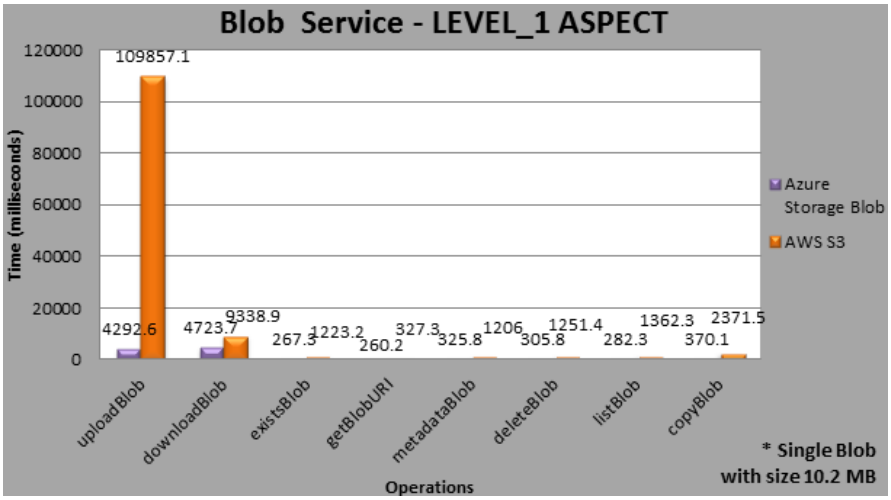


Fig. 4. Blob Service Level-1 Core Object.

The **File Service tests** were performed on Azure Storage File, GoogleDrive and DropBox. The tests include two object levels. The Level-2 represents Share and Directory. The Level-1 represents File. The total number of tests performed was 26. The performance tests compare the operations across the service providers. Each operation was run 10 times, and the corresponding process time for each request from T1 to T10 was calculated. Each request was processed with same file size of 10.2MB. The results include start time, end time, average time and total duration – Figs. 5 and 6.

The **Table Service tests** were performed on Azure Storage Table, Azure DocumentDB, AWS DynamoDB and AWS SimpleDB. The Tests include two object levels. The Level-2 represents Database and Collections (which includes Table, Collections, Table and Domain). The Level-1 represents Item (which in-

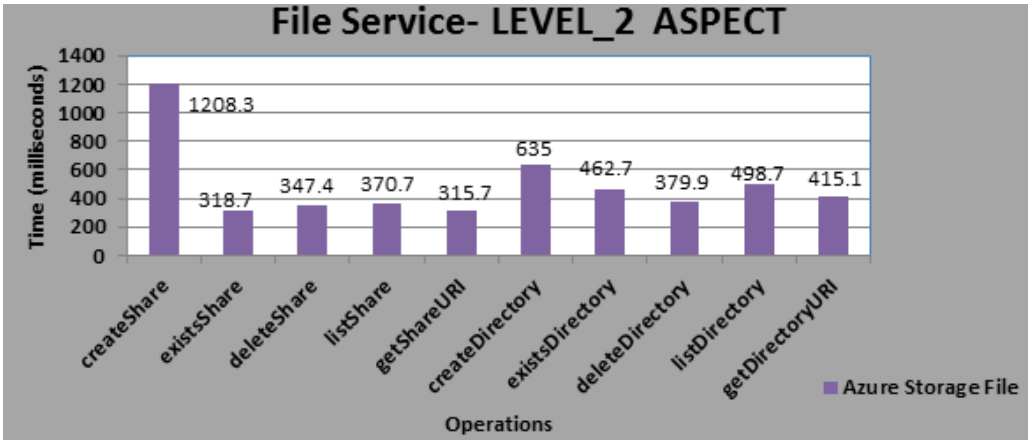


Fig. 5. File Service Level-2 Composite Object.

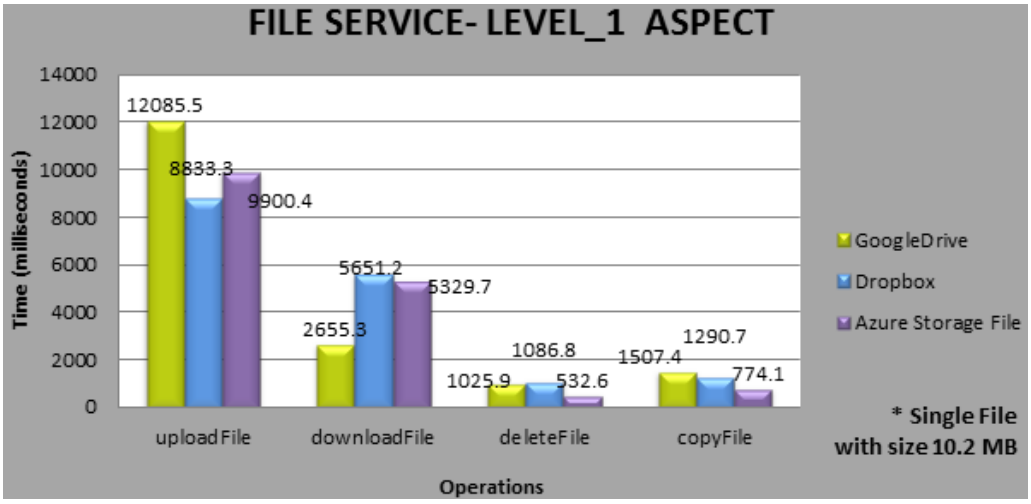


Fig. 6. File Service Level-1 Core Object.

cludes Entity, Document, Table Item and Domain Item) and Attachment. The total number of tests was 45. The performance tests compare the operations across the service providers. Each operation was run 10 times, and the corresponding process time for each request from T1 to T10 was calculated. Each request was processed with a single data record of approximately four columns. The results include start time, end time, average time and total duration – Figs. 7 and 8.

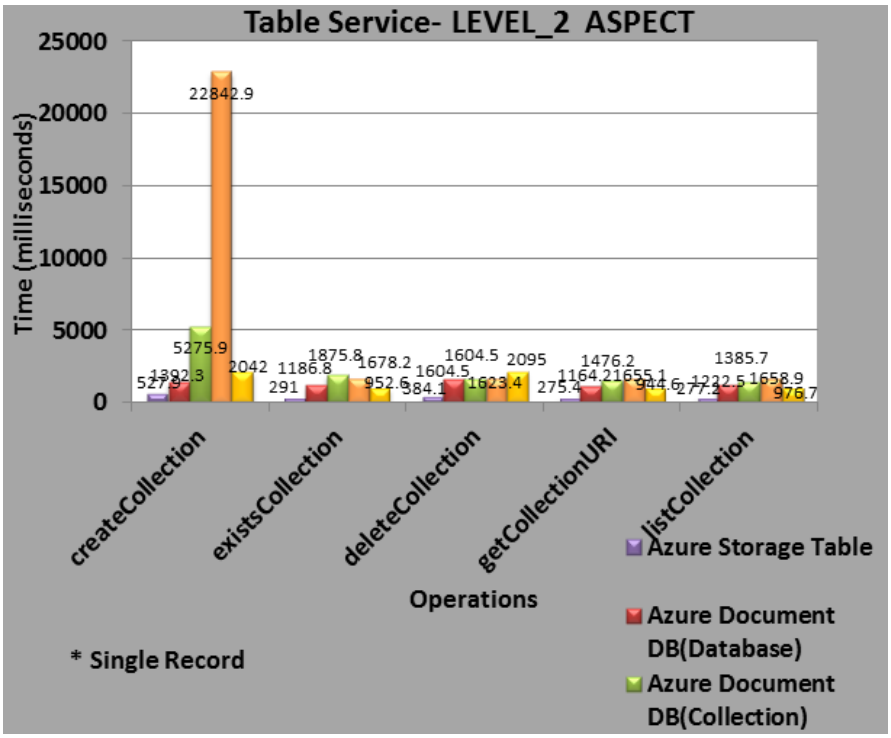


Fig. 7. Table Service Level-2 Composite Object.

5 Discussion of Results and Conclusions

Cloud service brokerage aims at customising or integrating existing services or making them interoperable. We have developed an integration broker following the classification schemes in [12, 11, 36]. The main purpose is intermediation between consumers and providers to provide advanced capabilities (interoperability and portability [30]) that builds up on an intermediary/broker platform to provide a marketplace to bring providers and customers together

We have focussed here on a broker solution for cloud storage service providers to implement a joint interface to allow

– easy portability for the user and

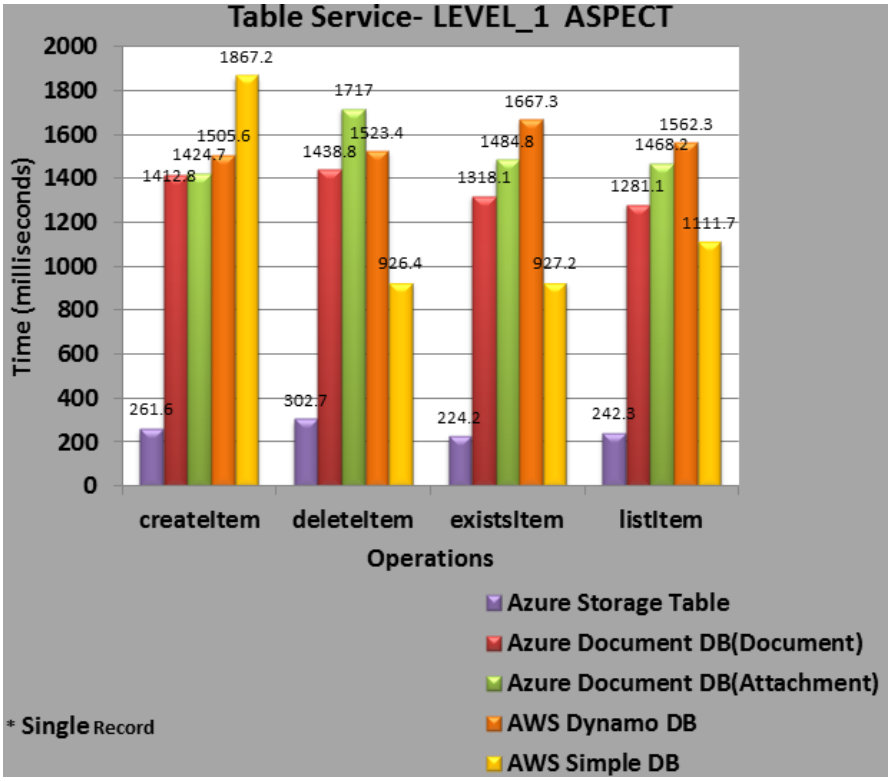


Fig. 8. Table Service Level-1 Core Object.

– easy extensibility for the broker provider.

This broker solution enables through the joint API also the opportunity for a cloud storage user to easily migrate between service providers and evolve the systems [21, 8], without having sufficient standards [20, 5, 6]. We looked here into using the broker to carry out performance tests across the providers in order to aid the decision which provider to choose.

The main results from the tests detailed in the paper are as follows:

– a) The performance of core object storage operations varies significantly across Cloud providers. Azure outperforms AWS S3 by a factor of between 4 and 5. For individual object operations, Azure is also up to 5 times faster in terms of access speed. For example, the common function of UploadBlob takes approximately 4 seconds on Azure and 10 seconds on AWS S3 for a 10.2 MB file.
– b) The tests of composite operations [25] that relate to collections show that Azure has significantly more access performance than other providers. In particular, AWS DynamoDB has a unusually long access time for its CollectionCreate operation. The tests on individual table entity operations show Azure to be the fastest by a considerable margin with over 5 to 6 times lesser access speeds on average.
– c) The average of the combined file upload and download speeds do not vary considerably across the 3 providers tested.

In the future, we plan to consider more storage services. Furthermore, the impact of different architectures in terms on IaaS or PaaS with and without the use of container technologies [37] shall be explored.

Acknowledgements

This work was partly supported by IC4 (Irish Centre for Cloud Computing and Commerce), funded by EI and the IDA.

References

1. S. Ried: Cloud Broker – A New Business Model Paradigm. Forrester. 2011.
2. D. Benslimane, S. Dustdar, A. Sheth: Services Mashups: The New Generation of Web Applications. Internet Computing, vol.12, no.5, pp.13-15, 2008.
3. D. Bernstein, E. Ludvigson, K.Sankar, S. Diamond, M. Morrow: Blueprint for the Inter-cloud: Protocols and Formats for Cloud Computing Interoperability. Intl Conf Internet and Web Appl and Services. 2009.
4. R. Buyya, R. Ranjan, R.N. Calheiros: Intercloud: Utility-Oriented Federation of Cloud Computing Environments For Scaling of Application Services. Intl Conf on Algorithms and Architectures for Parallel Processing , LNCS 6081. 2010.
5. Cloud Standards. http://cloud-standards.org/. 2017.
6. ETSI Cloud Standards. http://www.etsi.org/newsevents/ news/734-2013-12-press-release-report-on-cloudcomputing- standards. 2017.
7. C. Fehling, R. Mietzner: Composite as a Service: Cloud Application Structures, Provisioning, and Management. Information Technology 53:4, pp. 188-194. 2011.
8. C. Pahl, P. Jamshidi, D. Weyns: Cloud architecture continuity: Change models and change rules for sustainable cloud software architectures. Journal of Software: Evolution and Process 29(2). 2017.
9. C. Pahl. P. Jamshidi, O. Zimmermann: Architectural Principles for Cloud Software. ACM Transactions on Internet Technology. 2017.
10. Forrester Research: Cloud Brokers Will Reshape The Cloud. 2012. http://www.cordys.com/ufc/file2/cordyscms sites/download/ 09b57cd3eb6474f1fda 1cfd62ddf094d/pu/
11. F. Fowley, C. Pahl, L. Zhang: A Comparison Framework and Review of Service Brokerage Solutions for Cloud Architectures. 1st International Workshop on Cloud Service Brokerage. 2013.
12. F. Fowley, C. Pahl, P. Jamshidi, D. Fang, X. Liu: A Classification and Comparison Framework for Cloud Service Brokerage Architectures. IEEE Transactions on Cloud Computing. 2017.
13. S. Garcia-Gomez et al.: Challenges for the comprehensive management of Cloud Services in a PaaS framework. Scalable Computing: Practice and Experience 13(3). 2012.
14. D.M. Elango, F. Fowley, C. Pahl: Pattern-driven Architecting of an Adaptable Ontology-driven Cloud Storage Broker. Proceedings CloudWays'2017 Workshop. 2017.
15. Gartner - Cloud Services Brokerage. Gartner Research, 2013. http://www.gartner.com/it-glossary/cloud-servicesbrokerage- csb

16. N. Grozev, R. Buyya: InterCloud architectures and application brokering: taxonomy and survey. Software: Practice and Experience. 2012.
17. D. Taibi, V. Lenarduzzi, C. Pahl: Processes, Motivations and Issues for Migrating to Microservices Architectures: An Empirical Investigation. IEEE Cloud Computing. 2017 (accepted for publication).
18. C. Pahl, P. Jamshidi: Microservices: A Systematic Mapping Study. Proceedings CLOSER Conference, Pages 137-146. 2016.
19. C.N. Hofer, G. Karagiannis: Cloud computing services: taxonomy and comparison. Journal of Internet Services and Applications, 2(2), 81-94. 2011.
20. IEEE Cloud Standards. http://cloudcomputing.ieee.org/standards. 2015.
21. P. Jamshidi, A. Ahmad, C. Pahl: Cloud Migration Research: A Systematic Review. IEEE Transactions Cloud Computing. 2013.
22. jclouds. jclouds Java and Clojure Cloud API. http://www.jclouds.org/. 2015.
23. A. Juan Ferrer et al. OPTIMIS: A holistic approach to cloud service provisioning. Future Generation Computer Systems, 28(1):66-77. 2012.
24. A.V. Konstantinou, T. Eilam, M. Kalantar, A.A. Totok, W. Arnold, E. Sniblel: An Architecture for Virtual Solution Composition and Deployment in Infrastructure Clouds. Intl Workshop on Virtualization Technologies in Distr Computing. 2009.
25. R. Mietzner, F. Leymann, M. Papazoglou: Defining Composite Configurable SaaS Application Packages Using SCA, Variability Descriptors and Multi-tenancy Patterns. Intl Conf on Internet and Web Applications and Services. 2008.
26. C. Pahl: Layered Ontological Modelling for Web Service-oriented Model-Driven Architecture. European Conf on Model-Driven Architecture ECMDA'05. 2005.
27. C. Pahl, S. Giesecke, W. Hasselbring: Ontology-based Modelling of Architectural Styles. Information and Software Technology (IST). 51(12): 1739-1749. 2009.
28. C. Pahl, H. Xiong: Migration to PaaS Clouds - Migration Process and Architectural Concerns. IEEE 7th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems MESOCA. 2013.
29. C. Pahl, H. Xiong, R. Walshe: A Comparison of On-premise to Cloud Migration Approaches. Europ Conf on Service-Oriented and Cloud Computing ESOCC. 2013.
30. D. Petcu et al.: Portable cloud applications - from theory to practice. Fut. Gen. Computer Systems 29(6):1417-1430. 2013.
31. M. Javed, Y.M. Abgaz, C. Pahl: Ontology change management and identification of change patterns. Journal on Data Semantics, 2(2-3): 119-143. 2013
32. Amazon Simple Storage Service (S3) Cloud Storage AWS https://aws.amazon.com/s3/
33. Dropbox https://www.dropbox.com/
34. Azure Storage - Secure cloud storage https://azure.microsoft.com/en-us/services/storage/
35. Google Drive - Cloud Storage & File Backup https://www.google.com/drive/
36. P. Jamshidi, C. Pahl, N.C. Mendonca: Pattern-based multi-cloud architecture migration. Software: Practice and Experience 47 (9), 1159-1184. 2017.
37. C. Pahl, A. Brogi, J. Soldani, P. Jamshidi: Cloud Container Technologies: a State-of-the-Art Review. IEEE Transactions on Cloud Computing. 2017.
38. C.M. Aderaldo, N.C. Mendonca, C. Pahl, P. Jamshidi: Benchmark requirements for microservices architecture research Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering. IEEE. 2017.
39. R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L.E. Lwakatare, C. Pahl, S. Schulte, J. Wettinger: Performance Engineering for Microservices: Research Challenges and

Directions. Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion. 2017.

SERVICE	PROVIDER	LEVEL OF ASPECTS	Operation Type	Parameter:	Inputs	FileSize	Start Time	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	End Time	Avg Time	Duration	Avg
BlobService	Azure Storage Blot	LEVEL_2	createContainer	ntainerNa	stcontaine	10.2 MB	1478.612	3004	105	114	117	112	116	238	111	184	138	1478.612	423	00:04.0	423.9
BlobService	Azure Storage Blot	LEVEL_2	existsContainer	ntainerNa	stcontaine	10.2 MB	1478.613	2500	76	83	73	93	71	73	79	82	117	1478.613	324	00:03.0	324.7
BlobService	Azure Storage Blot	LEVEL_2	deleteContainer	ntainerNa	stcontaine	10.2 MB	1478.616	2575	78	71	70	69	68	69	71	70	74	1478.616	321	00:03.0	321.5
BlobService	Azure Storage Blot	LEVEL_2	listContainer			10.2 MB	1478.613	2153	73	60	56	65	60	56	59	76	65	1478.613	272	00:02.1	272.3
BlobService	Azure Storage Blot	LEVEL_2	getContainerURI	ntainerNa	stcontaine	10.2 MB	1478.613	2338	68	38	35	41	35	40	39	47	43	1478.613	272	00:02.1	272.4
BlobService	AWS S3	LEVEL_2	createContainer	ntainerNa	picasso-bu	10.2 MB	1478.699	6387	1587	1711	1370	1426	1618	2172	1647	1530	1808	1478.699	2125	00:21.0	2125.6
BlobService	AWS S3	LEVEL_2	existsContainer	ntainerNa	picasso-bu	10.2 MB	1478.699	5493	841	723	757	910	744	707	727	710	685	1478.699	1229	00:12.0	1229.7
BlobService	AWS S3	LEVEL_2	deleteContainer	ntainerNa	picasso-bu	10.2 MB	1478.706	5394	1186	1071	1173	1845	1152	1188	1350	1167	1046	1478.706	1657	00:16.1	1657.2
BlobService	AWS S3	LEVEL_2	listContainer			10.2 MB	1478.701	4580	941	836	1061	893	737	1280	833	1294	842	1478.701	1329	00:13.0	1329.7
BlobService	AWS S3	LEVEL_2	getContainerURI	ntainerNa	picasso-bu	10.2 MB	1478.7	2649	30	17	15	16	15	17	27	14	14	1478.7	281	00:02.1	281.4
BlobService	Azure Storage Blot	LEVEL_1	Upload	ilePath,Bl	d\\Project\	10.2 MB	1478.614	9781	3553	5137	3245	3110	3141	2914	4522	4103	3420	1478.614	4292	00:42.1	4292.6
BlobService	Azure Storage Blot	LEVEL_1	Download	oadFilePa	tject\\New	10.2 MB	1478.614	7225	5097	4984	4083	4576	4215	4351	4117	4139	4450	1478.614	4723	00:47.0	4723.7
BlobService	Azure Storage Blot	LEVEL_1	Exists	ne,BlobRe	ontainer",	10.2 MB	1478.614	2023	69	66	63	68	76	82	77	76	73	1478.614	267	00:02.1	267.3
BlobService	Azure Storage Blot	LEVEL_1	URI	ne,BlobRe	ontainer",	10.2 MB	1478.614	2277	37	35	33	34	33	41	32	46	34	1478.614	260	00:02.1	260.2
BlobService	Azure Storage Blot	LEVEL_1	Metadata	ne,BlobRe	ontainer",	10.2 MB	1478.615	2879	44	54	45	43	41	39	38	37	38	1478.615	325	00:03.0	325.8
BlobService	Azure Storage Blot	LEVEL_1	Delete	ne,BlobRe	ontainer",	10.2 MB	1478.616	2385	76	81	74	66	67	71	70	73	95	1478.616	305	00:03.0	305.8
BlobService	Azure Storage Blot	LEVEL_1	List	ntainerNa	stcontaine	10.2 MB	1478.615	2135	66	75	75	75	85	77	91	73	71	1478.615	282	00:02.1	282.3
BlobService	Azure Storage Blot	LEVEL_1	ListBatch	ntainerNa	stcontaine	10.2 MB	1478.615	2189	91	100	85	77	77	76	75	76	81	1478.615	292	00:02.1	292.7
BlobService	Azure Storage Blot	LEVEL_1	Copy	,Destinat	ontaine	10.2 MB	1478.616	2251	136	224	144	179	154	174	147	148	144	1478.616	370	00:03.1	370.1
BlobService	AWS S3	LEVEL_1	Upload	ePath,Blo	ard\\Proje	10.2 MB	1478.702	115172	113111	99692	108463	106346	114377	109462	112648	111216	108084	1478.703	109857	18:18.1	109857.1
BlobService	AWS S3	LEVEL_1	Download	adFilePat	Project\\N	10.2 MB	1478.704	18565	7761	10831	7672	7507	7502	8273	8285	9286	7707	1478.704	9338	01:33.0	9338.9
BlobService	AWS S3	LEVEL_1	Exists	e,BlobRe	fesso-buck	10.2 MB	1478.704	4402	814	784	775	1449	798	809	854	787	760	1478.704	1223	00:12.0	1223.2
BlobService	AWS S3	LEVEL_1	URI	e,BlobRe	fesso-buck	10.2 MB	1478.704	3061	25	20	21	30	30	21	25	21	19	1478.704	327	00:03.0	327.3
BlobService	AWS S3	LEVEL_1	Metadata	e,BlobRe	fesso-buck	10.2 MB	1478.705	4758	844	1043	818	813	787	722	816	755	704	1478.705	1206	00:12.0	1206
BlobService	AWS S3	LEVEL_1	Delete	e,BlobRe	fesso-buck	10.2 MB	1478.705	5307	740	697	730	707	841	1279	706	745	762	1478.705	1251	00:12.1	1251.4
BlobService	AWS S3	LEVEL_1	List	ucketNam	casso-bu	10.2 MB	1478.705	5284	1018	912	899	1013	888	880	897	893	939	1478.705	1362	00:13.1	1362.3
BlobService	AWS S3	LEVEL_1	Copy	,Destinat	ontaine	10.2 MB	1478.705	6883	1904	1756	1834	2032	2087	1829	1875	1781	1734	1478.705	2371	00:23.1	2371.5
FileService	Azure Storage File	LEVEL_2	createShare	ShareNam	"testshare'	10.2 MB	1478.523	5052	985	916	770	878	741	562	683	768	728	1478.523	1208	00:12.0	1208.3
FileService	Azure Storage File	LEVEL_2	existsShare	ShareNam	"testshare'	10.2 MB	1478.529	2670	80	39	64	64	77	63	37	38	55	1478.529	318	00:03.0	318.7
FileService	Azure Storage File	LEVEL_2	deleteShare	ShareNam	"testshare'	10.2 MB	1478.539	2744	72	71	69	70	70	94	78	100	106	1478.539	347	00:03.0	347.4
FileService	Azure Storage File	LEVEL_2	listShare			10.2 MB	1478.532	2990	123	112	54	63	84	75	77	53	76	1478.532	370	00:03.1	370.7
FileService	Azure Storage File	LEVEL_2	getShareURI	ShareNam	"testshare'	10.2 MB	1478.529	2252	86	113	92	111	93	111	94	114	91	1478.529	315	00:03.0	315.7
FileService	Azure Storage File	LEVEL_2	createDirectory	me,Directre	"testdi	10.2 MB	1478.532	4317	310	303	108	178	145	193	189	445	162	1478.532	635	00:06.0	635
FileService	Azure Storage File	LEVEL_2	existsDirectory	me,Directre	"testdi	10.2 MB	1478.532	3726	81	109	95	131	76	110	93	112	94	1478.532	462	00:04.1	462.7
FileService	Azure Storage File	LEVEL_2	deleteDirectory	me,Directre	"testdi	10.2 MB	1478.539	2838	105	107	105	106	123	103	103	101	108	1478.539	379	00:03.1	379.9
FileService	Azure Storage File	LEVEL_2	listDirectory			10.2 MB	1478.534	2779	369	410	190	209	202	209	201	201	217	1478.534	498	00:04.1	498.7
FileService	Azure Storage File	LEVEL_2	getDirectoryURI	me,Directre	"testdi	10.2 MB	1478.532	2481	205	204	205	205	205	362	88	87	109	1478.532	415	00:04.0	415.1
FileService	Azure Storage File	LEVEL_1	Upload	ame,FileP	rimard\\Pi	10.2 MB	1478.534	24474	4738	11320	14444	7463	12013	9090	6784	4711	3967	1478.534	9900	01:39.0	9900.4
FileService	Azure Storage File	LEVEL_1	Download	ame,FilePa	\\Project\	10.2 MB	1478.535	8569	4791	4844	4757	4990	4801	4992	5355	5557	4641	1478.535	5329	00:53.0	5329.7
FileService	Azure Storage File	LEVEL_1	Exists	oryName,F	"testdirec	10.2 MB	1478.536	2779	296	315	299	311	305	308	304	279	334	1478.536	553	00:05.1	553
FileService	Azure Storage File	LEVEL_1	Metadata	oryName,F	"testdirec	10.2 MB	1478.536	2728	157	131	110	157	134	146	103	110	131	1478.536	390	00:03.1	390.7
FileService	Azure Storage File	LEVEL_1	Delete	oryName,F	"testdirec	10.2 MB	1478.537	2871	408	476	447	256	179	187	197	159	146	1478.537	532	00:05.0	532.6
FileService	Azure Storage File	LEVEL_1	Copy	ShareNam	"testsh	10.2 MB	1478.537	4283	352	307	294	295	384	627	603	280	316	1478.537	774	00:07.1	774.1
FileService	DropBox	LEVEL_1	Upload	FileRefere	iew folder	10.2 MB	1478.283	10746	9155	7385	8656	9846	7310	9354	8973	8459	8449	1478.283	8833	01:28.0	8833.3
FileService	DropBox	LEVEL_1	Metadata	eferenceN	"test.pdf"	10.2 MB	1478.284	3330	501	503	557	510	482	482	471	506	520	1478.284	786	00:07.1	786.2
FileService	DropBox	LEVEL_1	Download	FileRefere	r/Dropbo	10.2 MB	1478.518	6933	4228	4904	4961	4890	5403	6672	5535	4693	8293	1478.518	5651	00:56.1	5651.2
FileService	DropBox	LEVEL_1	Delete	FilePath	"test.pdf"	10.2 MB	1478.519	3385	722	666	770	1634	777	756	725	726	707	1478.519	1086	00:10.1	1086.8
FileService	DropBox	LEVEL_1	Copy	Destinatio	pdf", "Cop	10.2 MB	1478.519	3957	810	864	818	820	1344	1114	767	859	1554	1478.519	1290	00:12.1	1290.7
FileService	GoogleDrive	LEVEL_1	Upload	Path,File	folder\\Go	10.2 MB	1478.268	17282	15613	11321	10637	11029	13292	10830	10418	10210	10223	1478.268	12085.5	02:00.0	12085.5
FileService	GoogleDrive	LEVEL_1	List			10.2 MB	1478.28	2788	314	329	294	280	359	317	379	292	275	1478.28	562	00:05.1	562.7
FileService	GoogleDrive	LEVEL_1	Download	Path,File	folder\\	10.2 MB	1478.272	5484	2332	2303	2524	2475	2287	2423	2091	2868	1766	1478.272	2655	00:26.1	2655.3
FileService	GoogleDrive	LEVEL_1	Delete	eferenceN	test.pdf	10.2 MB	1478.277	3892	673	854	745	661	696	666	692	745	635	1478.277	1025	00:10.0	1025.9
FileService	GoogleDrive	LEVEL_1	Copy	Destinatio	pdf" , "cop	10.2 MB	1478.279	4400	1083	1240	1800	1101	1016	1119	1083	1163	1069	1478.279	1507	00:15.0	1507.4
TableService	zure Storage Tabl	LEVEL_2	createTable	TableNam	"testtable"		1478.736	2634	402	306	280	275	268	324	252	275	263	1478.736	527	00:05.0	527.9
TableService	zure Storage Tabl	LEVEL_2	existsTable	TableNam	"testtable"		1478.736	2470	49	49	44	44	40	72	49	44	49	1478.736	291	00:02.1	291
TableService	zure Storage Tabl	LEVEL_2	deleteTable	TableNam	"testtable"		1478.739	2987	107	101	88	92	90	100	96	88	92	1478.739	384	00:03.1	384.1
TableService	zure Storage Tabl	LEVEL_2	getTableURI	TableNam	"testtable"		1478.736	2338	42	43	40	42	47	43	53	53	53	1478.736	275	00:02.1	275.4
TableService	zure Storage Tabl	LEVEL_2	listTable				1478.737	2372	48	43	51	45	42	42	42	43	44	1478.737	277	00:02.1	277.2
TableService	AzureDocumentDE	LEVEL_2	createDB	DBName	"TestDB"		1478.782	4091	1233	1321	1027	1115	938	917	918	1136	1227	1478.782	1392	00:13.1	1392.3
TableService	AzureDocumentDE	LEVEL_2	existsDB	DBName	"TestDB"		1478.782	3181	1195	851	1205	1024	116	1142	1176	1064	914	1478.782	1286	00:12.1	1186.8
TableService	AzureDocumentDE	LEVEL_2	deleteDB	DBName	"TestDB"		1478.805	4318	1491	1458	1376	1194	1207	1236	1284	1305	1176	1478.805	1604	00:16.0	1604.5
TableService	AzureDocumentDE	LEVEL_2	getDBURI	DBName	"TestDB"		1478.782	3807	886	800	830	744	803	1213	905	856	798	1478.782	1164	00:11.1	1164.2
TableService	AzureDocumentDE	LEVEL_2	listDB				1478.783	3340	947	922	918	1028	1015	1039	1023	1012	981	1478.783	1222	00:12.0	1222.5
TableService	AzureDocumentDE	LEVEL_2	createCollection	e,Collectic	"TestCollection"		1478.784	4636	1841	2042	1739	1633	1561	3183	12362	11889	11873	1478.784	5275	00:52.1	5275.9
TableService	AzureDocumentDE	LEVEL_2	existsCollection	e,Collectic	"TestCollection"		1478.784	6885	1215	1232	1609	1294	1126	1103	1393	1449	1452	1478.784	1875	00:18.1	1875.8
TableService	AzureDocumentDE	LEVEL_2	deleteCollection	e,Collectic	"TestCollection"		1478.805	4318	1491	1458	1376	1194	1207	1236	1284	1305	1176	1478.805	1604	00:16.0	1604.5
TableService	AzureDocumentDE	LEVEL_2	getCollectionURI	e,Collectic	"TestCollection"		1478.784	4233	1227	1023	1240	1208	1239	1320	1023	1131	1118	1478.784	1476	00:14.1	1476.2
TableService	AzureDocumentDE	LEVEL_2	listCollection	DBName	"TestDB"		1478.784	3522	1129	1254	1109	1112	1302	973	1082	1244	1130	1478.784	1385	00:13.1	1385.7
TableService	AWSDynamoDB	LEVEL_2	createTable	TableNam	"testtable"		1478.815	26592	22075	21986	21475	26317	21638	21683	21883	23070	21710	1478.815	22842	03:48.0	22842.9
TableService	AWSDynamoDB	LEVEL_2	existsTable	TableNam	"testtable"		1478.815	6235	1207	1047	1229	1133	1325	1115	1138	1021	1332	1478.815	1678	00:16.1	1678.2
TableService	AWSDynamoDB	LEVEL_2	deleteTable	TableNam	"testtable"		1478.824	5176	1348	1229	1228	1171	1491	1228	1332	1026	1005	1478.824	1623	00:16.0	1623.4
TableService	AWSDynamoDB	LEVEL_2	getTableURI	TableNam	"testtable"		1478.815	5484	1238	1213	1243	1211	1192	1184	1227	1208	1351	1478.815	1655	00:16.1	1655.1
TableService	AWSDynamoDB	LEVEL_2	listTable				1478.824	8127	966	1046	898	954	912	921	922	922	921	1478.824	1658	00:16.1	1658.9
TableService	AWSSimpleDB	LEVEL_2	createDomain	omainNam	estDomain"		1478.869	6269	1591	1561	1552	1638	1561	1579	1564	1552	1553	1478.869	2042	00:20.0	2042
TableService	AWSSimpleDB	LEVEL_2	existsDomain	omainNam	estDomain"		1478.869	4861	701	506	485	609	517	467	460	465	455	1478.869	952	00:09.1	952.6
TableService	AWSSimpleDB	LEVEL_2	deleteDomain	omainNam	estDomain"		1478.885	5779	1671	1656	1789	1703	1644	1677	1739	1630	1662	1478.885	2095	00:20.1	2095
TableService	AWSSimpleDB	LEVEL_2	getDomainURI	omainNam	estDomain"		1478.869	5114	510	484	475	553	455	461	474	463	457	1478.869	944	00:09.0	944.6
TableService	AWSSimpleDB	LEVEL_2	listDomain				1478.869	5380	513	475	500	541	473	468	472	459	486	1478.869	976	00:09.1	976.7
TableService	zure Storage Tabl	LEVEL_1	createEntity	nKey,Rown	"smith	1 Record	1478.776	2264	38	40	36	38	39	42	40	39	40	1478.776	261	00:02.1	261.6
TableService	zure Storage Tabl	LEVEL_1	deleteEntity	,Partition	"smith", "jonson"	1 Record	1478.777	2329	79	84	86	74	73	73	79	83	67	1478.777	302	00:03.0	302.7
TableService	zure Storage Tabl	LEVEL_1	existsEntity	,Partition	"smith", "smith"	1 Record	1479.301	1861	40	36	36	40	40	52	57	41	39	1479.301	224	00:02.0	224.2
TableService	zure Storage Tabl	LEVEL_1	listEntity	ame,Partit	table", "sr	1 Record	1479.301	2054	46	36	40	46	39	36	36	35	55	1479.301	242	00:02.0	242.3
TableService	AzureDocumentDE	LEVEL_1	createDocument	ne,Docume	ion", "doc	1 Record	1478.79	3659	1168	1180	1207	1132	1154	1161	1175	1153	1139	1478.79	1412	00:14.0	1412.8
TableService	AzureDocumentDE	LEVEL_1	existsDocument	ctionNam	ollection",	1 Record	1478.791	3738	1143	1077	1053	1089	1065	1008	998	999	1011	1478.791	1318	00:13.0	1318.1
TableService	AzureDocumentDE	LEVEL_1	listDocument	e,Collectic	"TestColl	1 Record	1478.791	3610	997	1092	1023	1031	1059	988	983	1007	1021	1478.791	1281	00:12.1	1281.1
TableService	AzureDocumentDE	LEVEL_1	deleteDocument	ctionNam	Collection'	1 Record	1478.795	3675	1300	1204	1231	1157	1183	1199	1137	1157	1145	1478.795	1438	00:14.0	1438.8
TableService	AzureDocumentDE	LEVEL_1	createAttachment	e,Docume	"docume	1 Record	1478.791	3762	1179	1256	1154	1136	1234	1130	1118	1121	1157	1478.791	1424	00:14.0	1424.7
TableService	AzureDocumentDE	LEVEL_1	existsAttachment	ame,Docum	"docu	1 Record	1478.793	3848	1168	1299	1177	1255	1222	1100	1264	1353	1162	1478.793	1484	00:14.1	1484.8
TableService	AzureDocumentDE	LEVEL_1	listAttachment	ctionNam	Collection'	1 Record	1478.792	4125	1327	1219	1228	1140	1112	1128	1142	1074	1187	1478.792	1468	00:14.1	1468.2
TableService	AzureDocumentDE	LEVEL_1	deleteAttachment	ame,Docum	"docu	1 Record	1478.794	4039	1584	1397	1459	1501	1246	1469	1456	1502	1517	1478.794	1717	00:17.0	1717
TableService	AWSDynamoDB	LEVEL_1	createItem	me,Attribu	ble","jsonstring"		1478.821	5130	1128	1221	1130	1127	1133	1021	1041	1010	1115	1478.821	1505	00:15.0	1505.6

Fig. 9. Detailed Performance Measurements.

TosKer: Orchestrating applications with TOSCA and Docker

Antonio Brogi, Luca Rinaldi, and Jacopo Soldani

Department of Computer Science, University of Pisa, Italy

Abstract. Docker is emerging as a simple yet effective solution for deploying and managing multi-component applications in virtualised cloud platforms. Application components can be shipped within portable and lightweight Docker containers, which can then be interconnected to allow components to interact each other. At the same time, the need for an enhanced support for orchestrating the management of the application components shipped within Docker containers is emerging.

In this paper we show how TOSCA can be exploited to provide such an enhanced support, by proposing a representation for describing the components forming an application, as well as the Docker containers used to ship such components. We also present TosKer, an engine for orchestrating the management of multi-component applications based on the proposed TOSCA representation and on Docker.

1 Introduction

Cloud computing has revolutionised IT, by allowing to run on-demand distributed applications at a fraction of the cost which was necessary just a few years ago [3]. This is possible as cloud providers exploit virtualisation techniques to achieve elasticity of large-scale shared resources [22]. Container-based virtualisation (where the operating system kernel permits running multiple isolated guest instances, called *containers*) can thus play an important role for cloud platforms, especially because it provides a lightweight virtualisation framework for PaaS/edge clouds [19,30] Applications can be packaged, along with all software dependencies they need to run, into portable and lightweight containers, which can then be managed on cloud platforms [28].

Containers are also an ideal solution for SOA-based architectural patterns (e.g., microservices [24]) that are emerging in the cloud community to decompose monolithic applications into suites of independently deployable, lightweight components. Application components can indeed be packaged in independently deployable, lightweight containers, which can then be interconnected to allow components to interact with each other (forming multi-container applications [31]).

Docker [14] is considered the de-facto standard for container-based virtualisation [29]. Docker permits packaging software components in Docker *images*, which are then exploited as read-only templates to create and run Docker *containers*. Docker containers can also mount external *volumes*, which ensure data persistence independently of the lifecycle of containers [23].

Docker permits orchestrating containers, by allowing to define multi-container Docker applications [31]. Given (the images of) the containers forming a multi-container application, the volumes they must mount, and the connections to set up among containers, Docker compose [15] is indeed capable of automatically deploying the corresponding application.

Docker containers are however treated as "black-boxes", and they constitute the minimum orchestration entity considered by currently existing approaches for orchestrating multi-component applications with Docker (e.g., [2,15,16,32]). Application components must be manually packaged, along with all their software dependencies, in (images of) Docker containers. Components are then strictly bound to their hosting containers, as it is not possible to orchestrate the management of the components forming an application independently of the Docker containers hosting them. For instance, it is not possible to run only some of the components hosted on a container, as whenever a container is started, all components it hosts are also started. Also, if we wish to change the container used to host a component, new Docker images must be manually developed (e.g., if a `maven` container is hosting the front-end and back-end of an application, and we wish to move the front-end to a `java` container, we must develop two new Docker images, one for hosting the front-end on a `java` container and one for hosting *only* the back-end on a `maven` container).

To fully exploit the potential of SOA, the current support for orchestrating multi-component applications with Docker should be enhanced. A concrete solution is to still rely on Docker containers as a portable and lightweight mean to deploy application components on cloud platforms, by also allowing to independently manage the components and containers forming a multi-component application [28]. In this paper we propose a solution precisely following this idea, which relies on the OASIS standard TOSCA [27] as the mean for orchestrating multi-component applications on top of Docker containers.

- We propose a TOSCA-based representation for multi-component applications, which permits modularly specifying the components forming an application, the Docker containers and Docker volumes needed to run them, as well as the relationships occurring among them (e.g., a component is hosted on a container, a component connects to another).
- We also present TosKer, an engine for orchestrating the management of multi-component applications based on the proposed TOSCA representation and on Docker.

Our approach enhances the current support for orchestrating the management of multi-component applications in Docker, as it considers application components as orchestration entities, which are independent from the Docker containers and Docker volumes used to build their runtime infrastructure. For instance, TosKer allows to independently manage the application components hosted on a Docker container, hence allowing to run only some of them (if needed). Our approach also eases the change of Docker containers used to host the components of an application, as this only requires to update the corre-

sponding TOSCA specification[1] (which will then be processed by TosKer to automatically deploy and manage the specified application).

The rest of the paper is organised as follows. Sect. 2 provides some background on TOSCA. Sect. 3 illustrates our proposal for specifying multi-container Docker applications in TOSCA, and Sect. 4 presents the TosKer engine for actually orchestrating such applications. Finally, Sects. 5 and 6 discuss related work and draw some concluding remarks, respectively.

2 Background

TOSCA (*Topology and Orchestration Specification for Cloud Applications* [27]) is an OASIS standard whose main goals are to enable (i) the specification of portable cloud applications and (ii) the automation of their deployment and management. TOSCA provides a YAML-based and machine-readable modelling language that permits describing cloud applications. Obtained specifications can then be processed to automate the deployment and management of the specified applications. We hereby report only those features of the TOSCA modelling language that are used in this paper[2].

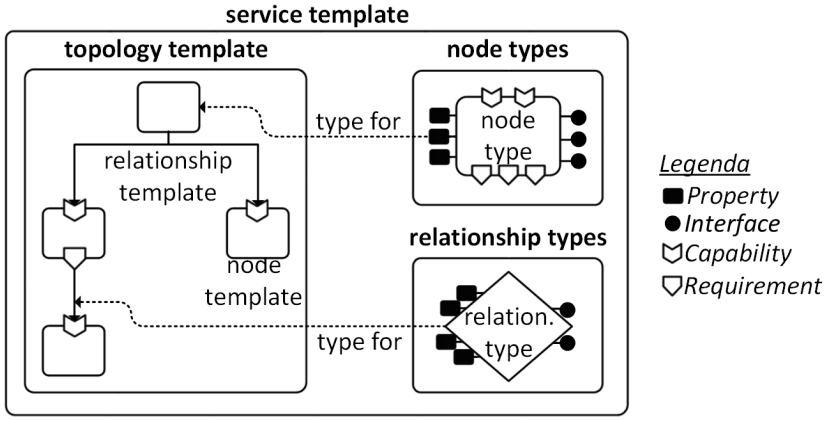


Fig. 1. The TOSCA metamodel [27].

TOSCA permits specifying a cloud application as a service template, that is in turn composed by a topology template, and by the types needed to build such a topology template (Fig. 1). The topology template is a typed directed graph that describes the topological structure of a multi-component application. Its nodes (called node templates) model the application components, while its edges (called relationship templates) model the relations occurring among such components.

[1] This can also be done automatically by exploiting TosKeriser [10]. Given a TOSCA application specification, TosKeriser can indeed automatically (discover and) include the Docker containers offering the software support needed by its components.

[2] A more detailed, self-contained introduction to TOSCA can be found in [5,12].

Node templates and relationship templates are typed by means of node types and relationship types, respectively. A node type defines the observable properties of a component, its possible requirements, the capabilities it may offer to satisfy other components' requirements, and the interfaces through which it offers its management operations. Requirements and capabilities are also typed, to permit specifying the properties characterising them. A relationship type instead describes the observable properties of a relationship occurring between two application components. As the TOSCA type system supports inheritance, a node/relationship type can be defined by extending another, thus permitting the former to inherit the latter's properties, requirements, capabilities, interfaces, and operations (if any).

Node templates and relationship templates also specify the artifacts needed to actually realise their deployment or to implement their management operations. As TOSCA allows artifacts to represent contents of any type (e.g., scripts, executables, images, configuration files, etc.), the metadata needed to properly access and process them is described by means of artifact types.

TOSCA applications are then packaged and distributed in CSARs (*Cloud Service ARchives*). A CSAR is essentially a zip archive containing an application specification along with the concrete artifacts realising the deployment and management operations of its components.

3 Specifying multi-component applications

Multi-component applications typically integrate various and heterogenous components [18]. We hereby define a TOSCA-based representation for such components, as well as for the Docker containers and Docker volumes that will be used to form their runtime infrastructure.

We first define three different TOSCA node types[3] to permit distinguishing the Docker containers, the Docker volumes, and the application components forming a multi-component application (Fig. 2).

– *tosker.nodes.Container* permits representing Docker containers, by indicating whether a container requires a *connection* (to another Docker container or to an application component), whether it has a generic *dependency* on another node in the topology, or whether it needs some persistent *storage* (hence requiring to be attached to a Docker volume). *tosker.nodes.Container* also permits indicating whether a container can *host* an application component, whether it offers an *endpoint* where to connect to, or whether it offers a generic *feature* (to satisfy a generic *dependency* requirement of another container/application component). To complete the description, *tosker.nodes.Container* provides placeholders (through the properties *ports*, *env_variables* and *command*, respectively) for specifying the port mappings, the

[3] The actual definition of all TOSCA types discussed in this section is publicly available on GitHub at `https://github.com/di-unipi-socc/tosker-types`.

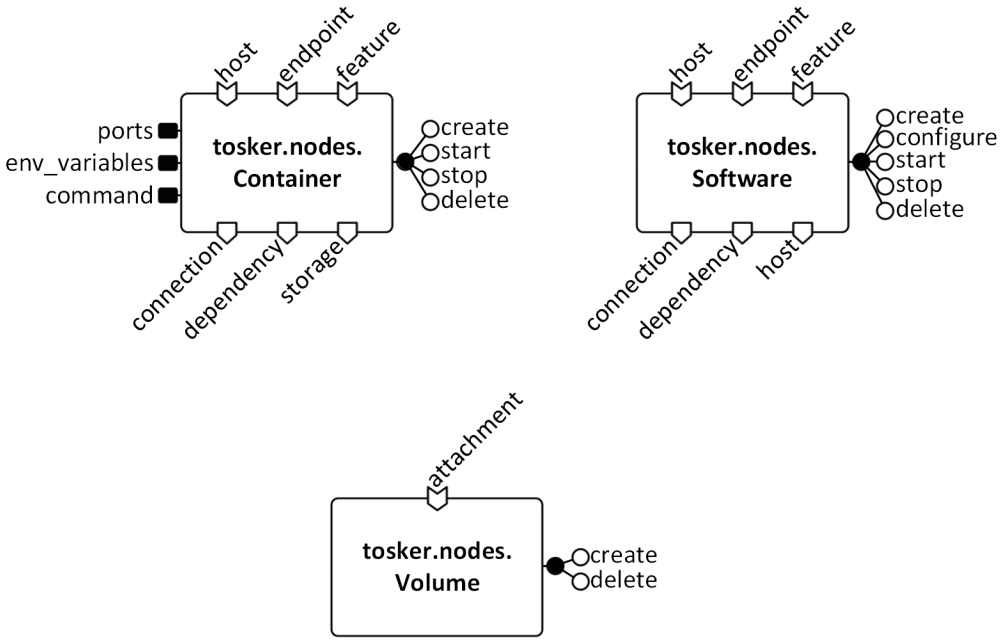


Fig. 2. TOSCA node types for multi-component, Docker-based applications, viz., *tosker.nodes.Container*, *tosker.nodes.Software*, and *tosker.nodes.Volume*.

environment variables, and the command to be executed when running the corresponding Docker container, and it lists the operations to manage a container (which correspond to the basic operations offered by the Docker platform [23]).

- *tosker.nodes.Volume* permits specifying Docker volumes, and it defines a capability *attachment* to indicate that a Docker volume can satisfy the *storage* requirements of Docker containers. It also lists the operations to manage a Docker volume (which corresponds to the basic operations offered by the Docker platform [23]).
- *tosker.nodes.Software* permits indicating the software components forming a multi-component application. It permits specifying whether an application component requires a *connection* (to a Docker container or to another application component), whether it has a generic *dependency* on another node in the topology, and that it has to be *host*ed on a Docker container or on another component[4]. *tosker.nodes.Software* also permits indicating whether an application component can *host* another application component, whether it provides an *endpoint* where to connect to, or whether it offers a generic *feature* (to satisfy a generic *dependency* requirement of a container/application component). Finally, *tosker.nodes.Software* indicates the operations to manage an application component (viz., *create*, *configure*, *start*, *stop*, *delete*).

[4] The *host* requirement is mandatory for nodes of type *tosker.nodes.Software*, as we assume that each application component must be installed in another component or in a Docker container.

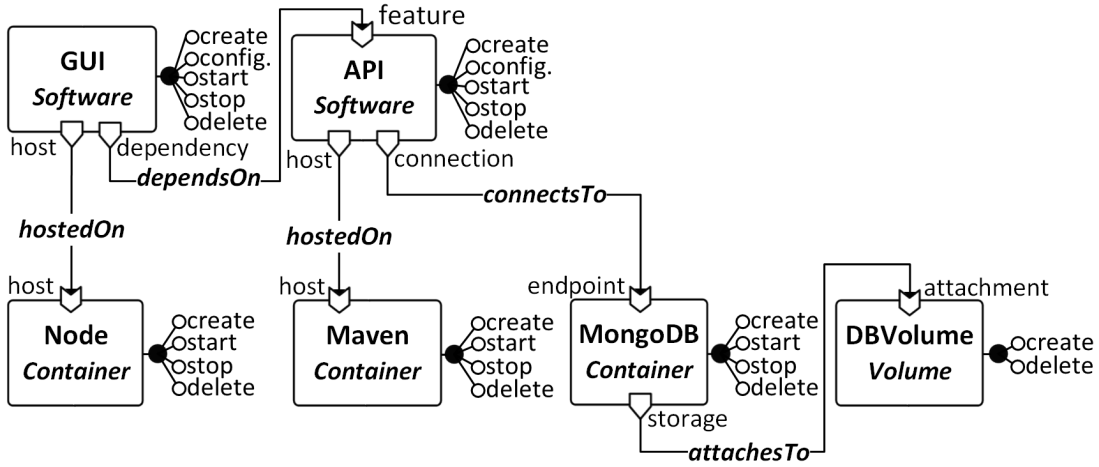


Fig. 3. An example of multi-component application specified in TOSCA (where nodes are typed with *tosker.nodes.Container*, *tosker.nodes.Volume*, or *tosker.nodes.Software*, while relationships are typed with TOSCA normative types [27]).

The interconnections and interdependencies among the nodes forming a multi-component application can be indicated by exploiting the TOSCA normative relationship types [27].

- *tosca.relationships.AttachesTo* can indeed be used to attach a Docker volume to a Docker container.
- *tosca.relationships.ConnectsTo* can indicate the network connections to establish between Docker containers and/or application components.
- *tosca.relationships.HostedOn* can be used to indicate that an application component is hosted on another component or on a Docker container (e.g., to indicate that a web service is hosted on a web server, which is in turn hosted on a Docker container).
- *tosca.relationships.DependsOn* can be used to indicate generic dependencies between the nodes of a multi-component application (e.g., to indicate that a component must be deployed before another, as the latter depends on the availability of the former to properly work).

Example 1. Consider *Thinking*, an open-source[5] web application that allows users to share their thoughts, so that all other users can read them. *Thinking* is composed by three main components, namely (i) a Mongo database storing the collection of thoughts shared by end-users, (ii) a Java-based REST API to remotely access the database of shared thoughts, and (iii) a web-based GUI visualising all shared thoughts and allowing to insert new thoughts into the database. Fig. 3 illustrates a representation of the *Thinking* application in TOSCA.

[5] The source code of *Thinking* is publicly available on GitHub at `https://github.com/di-unipi-socc/thinking`.

(i) The database is obtained by directly instantiating a *MongoDB* container, which needs to be attached to a volume where the shared thoughts will be persistently stored.
(ii) The *API* is hosted on a *Maven* Docker container, and it requires to be connected to the *MongoDB* container (for remotely accessing the database of shared thoughts).
(iii) The *GUI* is hosted on a *NodeJS* Docker container, and it depends on the availability of the *API* to properly work (as it sends GET/POST requests to the *API* to retrieve/add shared thoughts). □

Finally, also artifacts must be typed [27], as they are used to implement deployment and management operations of the nodes forming a multi-component application and they must specify the metadata needed to properly access and process them. We hence define *tosker.artifacts.Image* and *tosker.artifacts.Dockerfile* to permit indicating that an artifact is an actual image or a Dockerfile, which will then be used to create a Docker container. We also extend such artifact types by defining *tosker.artifacts.Image.Service* and *tosker.artifacts.Dockerfile.Service*, to permit distinguishing images that execute a service when started from those that "simply package" a runtime environment. We can instead rely on TOSCA normative artifact types [27] for all other kinds of artifacts linked by the nodes in a multi-container Docker application.

Example 1 (cont.). Consider again the application in Fig. 3. The image artifact associated to the *MongoDB* container is of type *tosker.artifacts.Image.Service*, as it links to an image offering a MongoDB server when executed. The image artifacts associated to the containers *Node* and *Maven* are instead of type *tosker.artifacts.Image*, as they link to images just offering runtime environments (for NodeJS-based and Maven-based applications, respectively). The management operations of *GUI* and *API* are instead implemented by ".sh" scripts[6]. □

4 TosKer

We hereby present TosKer, an orchestrator capable of automatically deploying and managing multi-component applications specified with the proposed TOSCA representation. We first illustrate the architecture of TosKer, and we then discuss its current prototype implementation.

4.1 The architecture of TosKer

Fig. 4 shows the architecture of TosKer, which is designed to be modular and easily extensible. The architecture of TosKer indeed partitions the functionalities of TosKer into lightweight modules that interact with each other, and new

[6] The resulting TOSCA application specification is publicly available at `https://github.com/di-unipi-socc/TosKer/blob/master/data/examples/thoughts-app/thoughts/thoughts.yaml`. A CSAR packaging such specification (together with all artifacts needed to deploy and manage the *Thinking* application) is avaialble at `https://github.com/di-unipi-socc/TosKer/blob/master/data/examples/thoughts-app/thoughts.csar`.

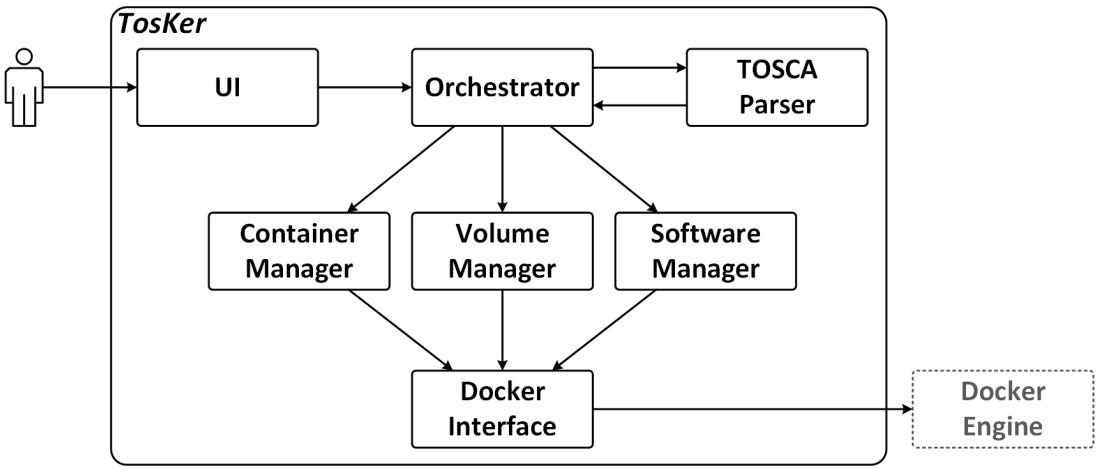


Fig. 4. The architecture of TosKer.

functionalities can be easily added to TosKer by developing and plugging-in new modules.

User interface. The UI allows to feed TosKer with the necessary input. The latter includes a CSAR (packaging the TOSCA specification of a multi-component application together with all artifacts needed to realise its management), a sequence of management operations to be executed, and (optionally) the subset of the application components on which to perform such a sequence of management operations.

TOSCA utilities. The TOSCA Parser is an utility module for parsing a CSAR and generating an internal representation of the application it packages. Such representation will then be exploited by the other modules in TosKer to deploy and manage the corresponding application.

Orchestration core. The Orchestrator is the core component of TosKer, as it is in charge of planning and orchestrating the management of multi-component applications. It first receives the input from the UI, and it exploits the TOSCA Parser to generate an internal representation of the multi-component application contained in the input CSAR.

The Orchestrator automatically determines which management operations have to be executed on which components, and in which order[7]. (to permit executing the input sequence of operations on the indicated subset of application components). The result is a (possibly expanded) sequence of management operations, each to be executed on a certain application component.

The Orchestrator then orchestrates the actual execution the above mentioned sequence of management operations by coordinating the Container Manager, Volume Manager and Software Manager. It indeed iterates over the sequence, and it

[7] The Orchestrator assumes that components are managed according to the TOSCA standard management lifecycle [27]. If such lifecycle is not respected (e.g., by requiring to *delete* a component that has not yet been created), then the Orchestrator will raise an error and stop orchestrating the application management.

dispatches the actual execution of an operation on a component to the corresponding manager (e.g., to *create* a component of type *tosker.nodes.Container*, the Orchestrator dispatches the actual execution of *create* on such component to the Container Manager). dispatched to the

Managers. The Container Manager, Volume Manager, and Software Manager implement the actual lifecycle for components of type *tosker.nodes.Container*, *tosker.nodes.Volume*, and *tosker.nodes.Software*, respectively.

- The Container Manager is in charge of implementing the operations to *create*, *start*, *stop* and *delete* Docker containers, by also taking into account the different types of artifacts from which they are generated (viz., Docker images or Dockerfiles — see Sect. 3).
- The Volume Manager has to implement the operations to *create* and *delete* Docker volumes (as volumes can only be created or deleted [23]).
- The Software Manager is in charge of implementing the operations to *create*, *configure*, *start*, *stop* and *delete* a component of type *tosker.nodes.Software*. Notice that, as such a kind of components will be hosted on Docker containers, the actual execution of a management operation on a component requires to issue commands to its container. For instance, to *create* a component, the Software Manager has to (i) copy all artifacts of the component inside a dedicated folder of its container, (ii) start the container by executing the script implementing the *create* operation of the component, (iii) commit the changes applied to the container as a new image, and (iv) re-create the container by exploiting the newly created image.

Notice that each manager implements management operations by instructing the Docker Interface on which Docker commands to execute.

Docker interface. The Docker Interface is in charge of interacting with the Docker engine installed on the host where TosKer is running. It is used by the managers to manage Docker containers and Docker volumes, and to execute operations inside running containers.

Notice that the Docker Interface decouples TosKer from the actual Docker engine used, meaning that it can issue commands to a classic Docker engine (as in the current implementation of TosKer— see Sect. 4.2), but it could also be used to issue commands to an engine capable of distributing containers in a cluster (e.g., Docker swarm [16] or Kubernetes [32]).

4.2 Prototype implementation

We have implemented a prototype of TosKer, which is open-source and publicly available on GitHub[8]. The prototype is written in Python[9], and it is composed

[8] `https://github.com/di-unipi-socc/TosKer`.

[9] The choice of Python was mainly motivated by the availability of two open-source Python libraries: *docker-py* (`https://github.com/docker/docker-py`) and *tosca-parser* (`https://github.com/openstack/tosca-parser/`). *docker-py* implements a Python interface for the Docker engine API. *tosca-parser* is instead a parser for TOSCA application specifications (developed by the OpenStack community).

by a main package (`tosker`) containing the set of Python modules implementing the various components forming the architecture of TosKer (viz., `ui.py`, `tosca_parser.py`, `orchestrator.py`, `container_manager.py`, `volume_manager.py`, `software_manager.py`, and `docker_interface.py`).

The current prototype of TosKer is also published on PyPI[10] (*Python Package index*), which permits installing it on a host by simply executing the command `pip install tosker`. It can then be used as a standard Python library, or as a command line software by executing:

```
$ tosker FILE [COMPONENTS] COMMANDS [INPUTS]
```

where `FILE` is a CSAR archive or a TOSCA YAML file (containing the specification of a multi-component application), `COMPONENTS` is optional and permits specifying the subset of application components to be managed, `COMMANDS` is the sequence of management operations to be executed, and `INPUTS` is an optional sequence of input parameters to be passed to the TOSCA application[11].

Example 2. Consider again the *Thinking* application in Example 1. Suppose, for instance, that we wish to *create* and *start* its *API* and *MongoDB*. We can instruct TosKer to do so, by executing:

```
$ tosker /usr/share/tosker/examples/thougthts.csar \
  API MongoDB create start
```

Notice that this will not only result in creating and starting *API* and *MongoDB*, but also the *Maven* container and *DBVolume* they require to properly work. *GUI* and *Node* will instead be ignored by TosKer, as they are not contained in set of components input to TosKer, nor they are needed by *API* or *MongoDB*. □

To test the current prototype of TosKer, we specified the open-source application *Thinking* in TOSCA, as well as three other existing applications, viz., (i) a Wordpress instance running on a PHP web server and connecting to a MySQL back-end, (ii) a NodeJS-based REST API connecting to a MongoDB back-end, and (iii) an application with three interacting servers written in NodeJS. All applications were effectively deployed by the current prototype of TosKer, and they constituted the basis for developing a battery of unit tests[12], which covered 96% of the source code of the Python modules we implemented (see Table 1).

5 Related work

We hereby position TosKer with respect to other currently available solutions for orchestrating the management of multi-component applications with Docker and/or TOSCA.

[10] `https://pypi.python.org/pypi/tosKer`.

[11] Details on how to process inputs for TOSCA applications can be found in [27].

[12] The TOSCA application specifications and the battery of unit tests that we implemented are publicly available on GitHub at `https://github.com/di-unipi-socc/TosKer/tree/master/data/examples` and `https://github.com/di-unipi-socc/TosKer/tree/master/tests`, respectively.

Table 1. Unit test coverage in the current prototype of TosKer (obtained by running the *coverage-py* tool — `https://coverage.readthedocs.io`).

Module	Total Statements	Missed Statements	Coverage
`ui.py`	75	18	76%
`docker_interface.py`	168	4	98%
`tosca_parser.py`	219	2	99%
`orchestrator.py`	105	2	98%
`container_manager.py`	26	0	100%
`volume_manager.py`	9	0	100%
`software_manager.py`	67	2	97%
Total	669	28	96%

Docker-based orchestration. Docker natively supports multi-container Docker applications with Docker compose [15]. Docker compose permits specifying the (images of) containers forming an application, the links/connections to be set between such containers, and the volumes to be mounted. Based on that, Docker compose is capable of deploying the specified application. However, Docker compose treats containers as black-boxes, meaning that there is no information on which components are hosted by a container, and that it is not possible to orchestrate the management of application components separately from that of their containers (as it is instead possible with TosKer).

Other approaches worth mentioning are Docker swarm [16], Kubernetes [32], and Mesos [2]. Docker swarm permits creating a cluster of replicas of a Docker container, and seamlessly managing it on a cluster of hosts. Kubernetes and Mesos instead permit automating the deployment, scaling, and management of containerised applications over clusters of hosts. Docker swarm, Kubernetes and Mesos differ from TosKer as they focus on how to schedule and manage containers on clusters of hosts, rather than on how to orchestrate the management of the components and containers forming multi-component applications.

TOSCA-based orchestration. OpenTOSCA [4] is an open-source engine for deploying and managing TOSCA applications. It is designed to work with a former, XML-based version of TOSCA [25], and to process applications "imperatively" (viz., by executing management plans defined by the application developer in the form of BPEL or BPMN workflows). TosKer instead works with the newer, YAML-based version of TOSCA [27], and it is designed to process applications "declaratively" (viz., by automatically determining the management plans to be executed from the topology of an application).

Other approaches worth mentioning are SeaClouds [8], Brooklyn [1], Alien4Cloud [17], and Cloudify [20]. SeaClouds [8] is a middleware solution for deploying and managing multi-component applications on heterogenous IaaS/PaaS clouds. SeaClouds fully supports TOSCA, but it lacks a support for Docker containers. The latter makes SeaClouds not suitable to orchestrate the management of multi-component applications including Docker containers.

Brooklyn [1], Alien4Cloud [17] and Cloudify [20] instead natively support Docker containers, and they permit orchestrating the management of the software components and Docker containers forming cloud applications. They however all differ from TosKer because they treat Docker containers as black-boxes (hence not permitting to orchestrate the management of application components separately from that of the containers hosting them).

Brooklyn [1] and Cloudify [20] also differ from TosKer as they require to specify applications in non-standard blueprint languages (inspired to, but not fully compliant with, the OASIS standards CAMP [26] and TOSCA [26], respectively). For instance, a relationship is specified in TOSCA by connecting a requirement of one component to a capability of another, and requirements/capabilities can be used to express interconnection constraints (which then permit validating TOSCA application topologies [9]). Cloudify blueprints instead do not include any notion of requirements or capabilities, as relationships just connect a source node to a target node.

Summary. To the best of our knowledge, ours is the first solution that permits specifying and orchestrating multi-component, Docker-based applications in TOSCA, and managing software components independently of the containers hosting them.

6 Conclusions

Container-based virtualisation is emerging as a simple yet effective solution for deploying and managing multi-component applications in cloud platforms [28]. Application components can be shipped within portable and lightweight Docker containers, which can then be interconnected to allow components to interact with each other. At the same time, the current support for orchestrating the management of the application components shipped within Docker containers is limited [29]. For instance, components must be manually packaged in Docker containers, and it is not possible to manage components independently of the containers hosting them (e.g., whenever a container is started/stopped, all components hosted on such container are also started/stopped).

In this paper we illustrated how TOSCA [27] can enhance the support for orchestrating multi-component applications with Docker. We indeed (i) proposed a TOSCA-based representation for multi-component applications, which permits distinguishing the Docker containers and software components in a multi-component application, as well as the relationships occurring among them. We also (ii) presented TosKer, an orchestration engine for automatically deploying and managing multi-component applications based on TOSCA and Docker.

Our approach enhances the current support for orchestrating the management of multi-component applications in Docker. TosKer can indeed automatically install application components within the containers hosting them (instead of requiring to manually package comoponents in images of Docker containers), and it permits independently orchestrating the management of components and

containers (instead of binding the management lifecycle of components to that of the containers hosting them).

We believe that our approach can also facilitate the widespread adoption of the TOSCA standard. TosKer indeed provides a lightweight, easy-to-use engine for deploying and managing TOSCA-based applications (exploiting Docker to host their components).

We tested the current prototype of TosKer by developing a battery of unit tests based on four existing applications. A more thorough evaluation of TosKer, based on concrete case studies and/or on datasets of multi-component applications (e.g., μset [6]), is in the scope of our immediate future work.

Additionally, the current prototype of TosKer permits orchestrating applications on single hosts and it does not yet support horizontal scaling of containers. TosKer can be adapted to include such features, for instance, by simply including a new version of the Docker Interface which interacts with Docker Swarm [16] or Kubernetes [32] (instead of with the Docker engine installed on a host). This is also in the scope of our future work.

It is finally worth noting that TosKer permits orchestrating the management of multi-component applications, by already offering some basic planning capabilities. For instance, when required to *start* a component of an application, TosKer automatically determines which other components have to be started, and it plans the sequence of operations that permits starting all such components. Such planning is however based on a fixed set of operations, whose behaviour is fixed by the TOSCA standard management lifecycle [27]. This is because our approach does not yet include a way to customise the management behaviour of application components. A solution can be to integrate our approach with models designed precisely to permit compositionally describing the management behaviour of the components forming an application (e.g., Aeolus [13] or fault-aware management protocols [7]), which would also permit improving the planning capabilities of TosKer (e.g, by exploiting the Aeolus-based planning algorithm in [21]). The integration of our approach with an existing solution for modelling, analysing and planning the management of multi-component applications is also in the scope of our future work.

Acknowledgments

The authors would like to thank Claus Pahl for all helpful and stimulating discussions on how to enhance the current support for orchestrating multi-component applications with Docker, which were reported in [11] and laid the foundations for the work presented in this paper.

References

1. Apache Software Foundation: Brooklyn. `http://brooklyn.apache.org`
2. Apache Software Foundation: Mesos. `http://mesos.apache.org/`

3. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. Communications of the ACM 53(4), 50–58 (2010)
4. Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., Wagner, S.: OpenTOSCA – a runtime for TOSCA-based cloud applications. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) Service-Oriented Computing: 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings. pp. 692–695. Springer, Berlin, Heidelberg (2013)
5. Binz, T., Breitenbücher, U., Kopp, O., Leymann, F.: TOSCA: Portable Automated Deployment and Management of Cloud Applications, pp. 527–549. Springer, New York, NY (2014)
6. Brogi, A., Canciani, A., Neri, D., Rinaldi, L., Soldani, J.: Towards a reference dataset of microservice-based applications. In: Proceedings of the 1th Workshop on Microservices: Science and Engineering (MSE 2017). Springer (2017), *[To appear]*
7. Brogi, A., Canciani, A., Soldani, J.: Fault-aware application management protocols. In: Aiello, M., Johnsen, B.E., Dustdar, S., Georgievski, I. (eds.) Service-Oriented and Cloud Computing: 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings. pp. 219–234. Springer (2016)
8. Brogi, A., Carrasco, J., Cubo, J., D'Andria, F., Ibrahim, A., Pimentel, E., Soldani, J.: EU Project SeaClouds - adaptive management of service-based applications across multiple clouds. In: Proceedings of the 4th International Conference on Cloud Computing and Services Science (CLOSER 2014). pp. 758–763 (2014)
9. Brogi, A., Di Tommaso, A., Soldani, J.: Validating TOSCA application topologies. In: Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,. pp. 667–678. SciTePress (2017)
10. Brogi, A., Neri, D., Rinaldi, L., Soldani, J.: From (incomplete) tosca specs to running apps, with docker. In: Software Engineering and Formal Methods - SEFM 2017 Collocated Workshops, Trento, Italy, September 4-5, 2017, Revised Selected Papers. Lecture Notes in Computer Science, Springer (2017), *[To appear]*
11. Brogi, A., Pahl, C., Soldani, J.: Enhancing the orchestration of multi-container docker applications, 2016. *[Submitted for publication]*
12. Brogi, A., Soldani, J., Wang, P.: TOSCA in a nutshell: Promises and perspectives. In: Villari, M., Zimmermann, W., Lau, K.K. (eds.) Service-Oriented and Cloud Computing: Third European Conference, ESOCC 2014, Manchester, UK, September 2-4, 2014. Proceedings. pp. 171–186. Springer Berlin Heidelberg (2014)
13. Di Cosmo, R., Mauro, J., Zacchiroli, S., Zavattaro, G.: Aeolus: A component model for the cloud. Information and Computation 239(0), 100 – 121 (2014)
14. Docker Inc.: Docker. `https://www.docker.com/`
15. Docker Inc.: Docker compose. `https://github.com/docker/compose`
16. Docker Inc.: Docker swarm. `https://github.com/docker/swarm`
17. FastConnect, Bull, Atos: Alien4cloud. `https://alien4cloud.github.io/`
18. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications. Springer (2014)
19. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and linux containers. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 171–172. IEEE Computer Society (2015)
20. GigaSpaces Technologies: Cloudify. `http://cloudify.co/`

21. Lascu, T.A., Mauro, J., Zavattaro, G.: A planning tool supporting the deployment of cloud applications. In: Proceedings of the 2013 IEEE 25th International Conference on Tools with Artificial Intelligence. pp. 213–220. ICTAI '13, IEEE Computer Society (2013)
22. Leymann, F.: Cloud computing. it — Information Technology, Methoden und innovative Anwendungen der Informatik und Informationstechnik 53(4), 163–164 (2011)
23. Matthias, K., Kane, S.P.: Docker: Up and Running. O'Reilly Media (2015)
24. Newman, S.: Building Microservices. O'Reilly Media, Inc. (2015)
25. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA), Version 1.0. `http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf` (2013)
26. OASIS: Cloud Application Management for Platforms (CAMP), Version 1.1. `http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.pdf` (2016)
27. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA) Simple Profile in YAML, Version 1.0. `http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.pdf` (2016)
28. Pahl, C.: Containerization and the paas cloud. IEEE Cloud Computing 2(3), 24–31 (2015)
29. Pahl, C., Brogi, A., Soldani, J., Jamshidi, P.: Cloud container technologies: A state-of-the-art review. IEEE Transactions on Cloud Computing `https://doi.org/10.1109/TCC.2017.2702586`, *[In press]*
30. Pahl, C., Lee, B.: Containers and clusters for edge cloud architectures – a technology review. In: Proceedings of the 2015 3rd International Conference on Future Internet of Things and Cloud. pp. 379–386. FICLOUD '15, IEEE Computer Society (2015)
31. Smith, R.: Docker Orchestration. Packt Publishing (2017)
32. The Kubernetes Authors: Kubernetes. `http://kubernetes.io/`

BPM@Cloud Workshop Papers

Towards PaaS Offering of BPMN 2.0 Engines: A Proposal for Service-level Tenant Isolation

Majid Makki, Dimitri Van Landuyt, Wouter Joosen

Abstract Business processes modeling and management solutions provide powerful abstraction mechanisms for the control flow of complex, task-driven applications, and as such allow for better alignment with business-related concerns. Despite the existence and wide adoption of standardized business process management languages such as WS-BPEL and BPMN 2.0, workflow engines in current Platform-as-a-Service (PaaS) offerings are in practice more restricted, in part for reasons such as vendor lock-in, but also due to restrictions of multi-tenant environments.

In this paper, we explore the main security-related problems caused by offering BPMN2-compliant workflow engines in a multi-tenant PaaS environment, particularly focusing on threats caused by misbehaving tenants and the lack of proper tenant isolation. In addition, we propose a service-level tenant isolation framework that allows PaaS offerings to support workflow engines which comply with the BPMN 2.0 standard, and we discuss the technical feasibility of implementing this framework using Java technologies such as OSGi and the Resource Consumption Management API (JSR-284).

1 Introduction

Platform-as-a-Service (PaaS) is a category of cloud computing services where the execution platform is offered to software teams for facilitating the development, deployment and maintenance of applications [29, 33]. When optimized resource utilization is among the main goals, different applications may share a single installation of the execution platform in a multi-tenant fashion. Workflow engines, in charge of executing business processes, can be part of a PaaS offering and, thus, shared among multiple tenant applications.

Majid Makki, Dimitri Van Landuyt, Wouter Joosen
imec-DistriNet, KU Leuven, 3001 Heverlee, Belgium, e-mail: firstname.lastname@cs.kuleuven.be

Renowned workflow engines in PaaS offerings, such as Amazon SWF [2] and Fantasm in Google App Engine [4], incur a high degree of vendor lock-in and are limited in functionality and suboptimal vis-à-vis utilization of resources. Since these engines have their own custom (i.e. non-standard) workflow modeling languages, the application will be, *de facto*, locked-in by the PaaS provider due to high cost of porting (cf. [24]). In addition, some features, such as human tasks or advanced event handling mechanisms which are commonly used in state-of-the-art business process automation (cf. [16]), are not supported out of the box. Supporting such features requires quite some ad-hoc engineering effort by the application developers. Furthermore, despite sharing the execution environment of the workflow engine between multiple tenant applications, these solutions require separate environments for executing workflow tasks of each distinct tenant application. The latter decreases resource efficiency whose maximization is a principal goal of cloud computing [11].

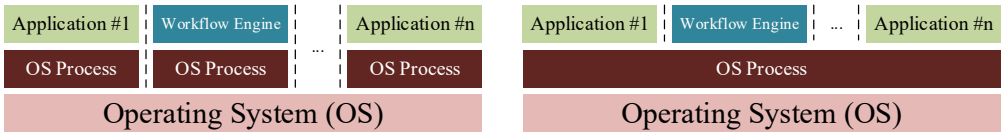
These problems can be solved by offering workflow engines that comply with the Business Process Modeling and Notation 2.0 (BPMN 2.0) [3] specification which, next to the Business Process Execution Language (BPEL) [8], is the standard increasingly being adopted in practice. The standardized nature of such engines increases the portability of applications developed using them. In addition, thanks to accumulated experience of decades which is behind BPMN 2.0, these engines do not lack necessary and mainstream functional features. Furthermore, these engines are capable of executing workflow tasks in the same execution environment as the engine itself. Thanks to this capability, resources are utilized more efficiently.

However, PaaS offering of BPMN2-compliant engines causes certain security threats which necessitates specific protection measures well beforehand. The principal source of threats is the untrusted tenant-provided code of workflow tasks that will be executed in an execution environment shared between the PaaS provider and multiple, possibly competing, tenants (cf. [30]). For instance, conflicting access to IO resources is possible. As an alternative example, tenants may exhaustively consume resources such as memory and bring down the service entirely.

The state-of-the-art protection mechanism against such threats is OS-level virtualization, i.e. hypervisors [21] or containers [12], where granularity-level of tenant isolation is, as shown in Figure 1(a), that of Operating System (OS) processes. This requires having at least one active OS process for each tenant which implies that quite some resources are reserved even if the tenant application does not impose any load. Moreover, OS-level virtualization is not sufficient for all functionalities of BPMN2-compliant engines because they execute some workflow tasks within the same OS process as the engine itself[1] which requires sharing a single OS process between the PaaS provider and tenants. For more efficient utilization of resources and being compatible with the nature of BPMN2-compliant engines, tenant isolation has to take place at a higher level in the computational stack. As depicted by Figure 1(b), this is the level where the service itself is implemented.

This work-in-progress paper proposes a *service-level* tenant isolation framework for enabling PaaS offering of BPMN 2.0 engines based on Java technologies. We

[1] This is required by the BPMN 2.0 specification for some types of tasks.



(a) Example of OS-level Virtualization (b) Service-level Isolation

Fig. 1 OS-level virtualization requires having separate OS processes for each tenant application and the workflow engine while service-level isolation allows running all code inside a single OS process.

formulate a concrete research problem by analyzing the BPMN 2.0 specification and widely-used BPMN2-compliant engines. Furthermore, given the fact that the *absolute* majority of BPMN2-compliant engines are Java-based [6], we present an initial outline of a solution based on Java-related technologies. The proposed solution takes threads as units of isolation for executing untrusted code of tenant applications to overcome the aforementioned insufficiency of the OS-level virtualization approach and the suboptimal resource utilization thereof. The technical feasibility of the solution is shown by explaining how existing technologies, such as OSGi [9] and the Java Resource Consumption Management API (JSR-284 [5]), enable its implementation.

The rest of this paper is structured as follows. Section 2 analyzes the research problem. Section 3 presents the solution outline along with remarks on its technical feasibility. Section 4 briefly contrasts this proposal with related work. Finally, Section 5 concludes the paper.

2 Problem Statement

This section analyzes the most compelling security threats caused by the PaaS offering of BPMN 2.0 engines and formulates the research problem as a number of concrete requirements.

2.1 Security Threat Analysis

We have systematically analyzed and prioritized the security threats using the **STRIDE**[2] threat model [23, 31]. The most problematic security threats, insofar as this paper is concerned, are related to two *core* features of BPMN 2.0 namely `Script Task` and `Service Task` activity types. The former is required by the standard to be "executed by a business process engine" [3]. This implies that the same OS process running the engine is responsible for executing the `Script`

[2] The acronym stands for six threat categories namely **S**poofing, **T**ampering with Data, **R**epudiation, **I**nformation Disclosure, **D**enial of Service and **E**levation of Privilege.

`Task`. While the standard does not require `Service Task` activities to be executed by the same Operating System (OS) process running the engine, most well-known and enterprise-ready BPMN 2.0 engines, such as jBPM [7] and Activiti [1], allow defining `Service Task` activities that are executed within the same OS process running the engine. Retaining this additional feature is essential for avoiding the overhead of remote service invocation, e.g. (de-)serialization and network delay.

In a multi-tenant context of a PaaS offering, the code of `Script Task` and `Service Task` activities belong to untrusted tenant applications using the engine. Executing untrusted code of tenants in the same OS process running the engine incurs different types of security threats (cf. [13, 15, 30]). Tampering with Data, Information Disclosure, Denial of Service and Elevation of Privilege are identified as the most important threat categories in this specific context and are discussed below.

Tampering with Data. Since tenant-provided code may access IO resources, one tenant application may modify another tenant's data stored on a shared device. In addition, a tenant application may modify the value of in-memory object references belonging to or shared with other tenants.

Information Disclosure. A tenant application may read another tenant's data by, e.g., listening on a network port belonging to the other tenant. Similarly, a tenant application may access in-memory object references belonging to or shared with other tenants.

Denial of Service. One tenant application may disrupt the PaaS offering entirely either by *using up* computational resources or by misusing part of the API shared among all tenants. The resources of concern are CPU cycles, memory space and IO bandwidth (both storage and network). Misuse of shared API can be either in form of killing the OS process hosting the service or in form of locking shared objects *indefinitely*.

Elevation of Privilege. By creating a thread which is not under control of the framework, one tenant application may increase its privileges and act without constraints imposed by the framework.

2.2 Requirements

Since these threats are caused by code running within a single OS process, protection against them has to take place inside that OS process as well. Therefore, a *service-level* tenant isolation framework is needed which fulfills the following functional requirements:

- FR1: The framework should guarantee that no tenant application may access objects or primitive values belonging to other tenants.
- FR2: The framework has to guarantee that shared system objects and references accessible for tenant applications can neither be locked indefinitely nor be modified by any of them.

- FR3: The creation of threads has to be entirely mediated by the framework.
- FR4: Permission of killing the OS process has to be denied for all tenant applications.
- FR5: The framework should check tenant permissions before granting access to any IO resource (e.g. a file on storage device or a network port).
- FR6: An upper limit has to be put on CPU usage, memory consumption and IO bandwidth (both storage and network) on a per-tenant basis.
- FR7: Upper limit imposition on resource consumption has to be so flexible that resource utilization maximizes. In other words, if there are unused resources that can be allocated safely, the framework should let tenant code exceed the limits to some extent.

In addition, fulfillment of the above functional requirements should respect the following quality requirements:

- QR1: All tenant isolation measurements should be enforced transparently by the framework. In other words, code of tenant applications has to be entirely decoupled from the tenant isolation framework.[3]
- QR2: The relative performance overhead of the framework compared to OS-level virtualization tactics (cf. [21, 12]) is required to be in an acceptable margin.

3 Service-level Tenant Isolation

This section outlines a service-level tenant isolation framework as a solution for fulfilling the above requirements and shows technical feasibility of the framework. Section 3.1 presents the framework architecture conceptually. Section 3.2 elaborates on partial fulfillment of FR1 while Section 3.3 supplements it and discusses static code restriction which is required for fulfillment of FR2 and FR3. Supplementary measures for fulfillment of FR2 and FR3 are presented in Section 3.4 while Section 3.5 deals with permission checking mechanism which is required for FR2, FR3, FR4 and FR5. Finally, Section 3.6 explains how the framework realizes FR6 and FR7. The qualitative requirements are taken into account orthogonally throughout this section.

3.1 Overall Architecture

Figure 2 shows the principal components running within a single OS process in three layers: (i) the bottom layer is the Java execution platform and the components it provides, (ii) the middle layer is where the service components of the PaaS offering,

[3] This is required for portability of tenant applications to other instances of the same BPMN 2.0 engine where the tenant isolation framework is not used.

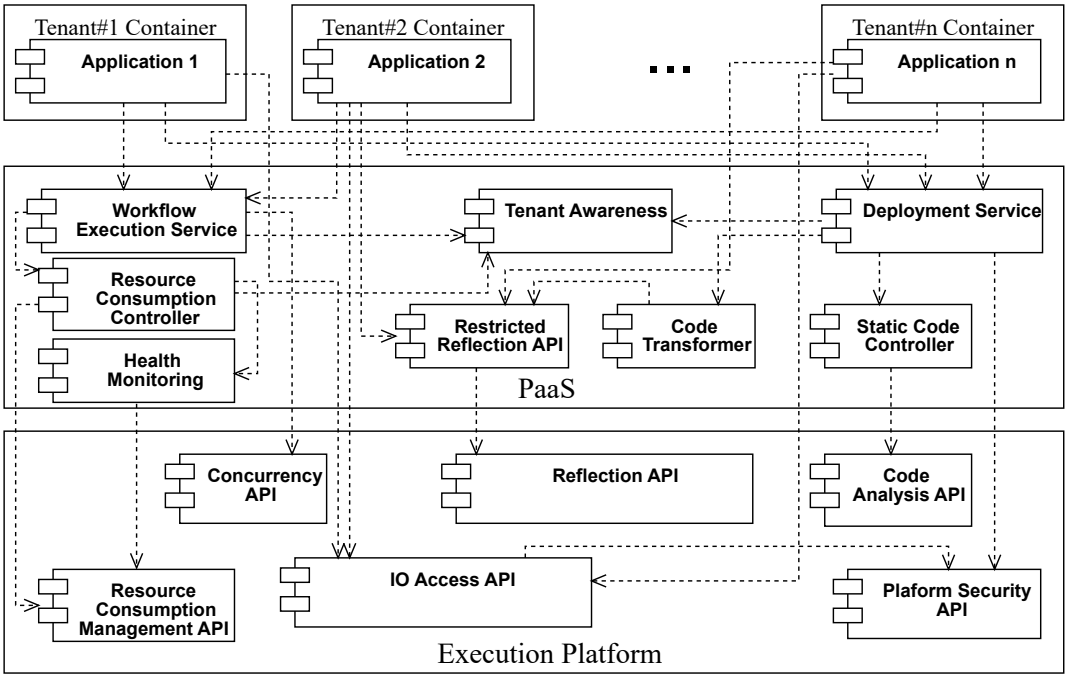


Fig. 2 Building Blocks of the Framework in Relation to Tenant Applications and Execution Platform

including components of the tenant isolation framework, sit, and (iii) the top layer consists of code of tenant applications built upon the PaaS offering.

The tenant isolation framework consists of seven main components which are introduced gradually throughout this section. The two front-end components of the framework which are directly used by tenants are `Deployment Service` and `Workflow Execution Service`. The former is responsible for deploying tenant applications into the PaaS environment. The latter is responsible for starting new workflow executions or continue/monitor/abort existing ones by running the code of tenant applications in isolation from other tenants.

For imposing isolation measures, the `Workflow Execution Service`, using an existing BPMN2-compliant engine such as jBPM [7], runs the untrusted code of tenant applications in separate threads and guarantees that each active thread is associated with only one tenant application at a time. Once each thread finishes its job, it can be reused for other tenant applications using a thread pool provided by the Java `Concurrency API`. The `Workflow Execution Service` leverages upon the `Concurrency API` of the Java execution platform for handling threads. Furthermore, it uses the `Tenant Awareness` component of the framework which is responsible for keeping track of associations between threads and tenants. Thus, the main cornerstone of the proposed solution is taking threads as units of tenant isolation just as OS-level virtualization tactics take OS processes as units of tenant isolation.

3.2 Tenant Containers

In order to dedicate a separate referencing space for objects of each tenant application (cf. FR1), the `Deployment Service` deploys each application in a distinct tenant container. As opposed to containers of OS-level virtualization approach, containers shown in Figure 2 are managed inside a single OS process. OSGi [9] bundles provide exactly this containerization functionality. OSGi loads each bundle using a distinct Java classloader and sets the bootstrap classloader, which is responsible for loading core Java classes, as the parent of each bundle classloader. Hence, the code of each bundle can access fields of its own classes and *static* fields of core Java classes.

By containing the code of each tenant application in a separate OSGi bundle, the FR1 requirement will be partially fulfilled. For complete fulfillment of FR1, cross-bundle communication between tenant application bundles has to be forbidden which is the topic of next section along with realization of FR2 and FR3.

3.3 Static Code Restriction

Access of tenant applications to other classes and interfaces has to be restricted for fulfillment of FR1, FR2 and FR3. API restriction is required both statically and dynamically. Static restriction takes place only once at deployment time by the `Static Code Controller` (cf. Figure 2). This is done in multiple stages using code analysis facilities provided by the Java platform and tools built upon it.

Cross-bundle Communication. Each OSGi bundle declares the list of classes it imports from other bundles. This is used for cross-bundle communication. By imposing limits on this list, the `Static Code Controller` guarantees that no cross-bundle communication is possible between two tenant applications and, thus, completes the realization of FR1.

Blacklist. One of the main elements of `Static Code Controller` is `BlacklistService` which maintains a list of classes, methods and fields that tenant applications are not allowed to use.

Given the structure of classloaders in OSGi, security vulnerabilities pertaining to shared object locks and modifications (cf. FR2) are caused by four Java programming constructs related to classes loaded by the bootstrap classloader: (i) *static* field declarations, (ii) reference updates on *static* fields, (iii) changing state of objects referenced by *static* fields, (iv) *static synchronized* methods, and (v) *synchronized* blocks locking *static* fields [13, 15, 30]. The `BlacklistService` searches for occurrences of these constructs in all classes loaded by the bootstrap classloader using the Java source code querying facilities provided by the Spoon library [28] as well as call graph construction and reference analysis (a.k.a. points-to analysis) mechanisms of the Soot framework [20]. A call graph consists of nodes and edges representing Java methods and invocation relationship between them respectively.

Reference analysis helps resolving non-static access to objects referenced by *static* fields of concern.

In addition, to further comply with FR3, the `BlacklistService` adds the `java.lang.Thread` class to the blacklist. Furthermore, using the Spoon library, it adds any method in the `java.util.concurrency` package capable of instantiating new threads or intervening in the life-cycle of an existing thread.

Acceptance Policy. The final stage of `Static Code Controller` involves accepting or rejecting the tenant application code. It checks whether the code of a tenant application directly or indirectly deals with blacklisted constructs. For instance, it checks whether a blacklisted `static synchronized` is invoked or the constructor of the `Thread` class is used. The first step for doing so is creating a call graph using the Soot framework. Afterwards, the call graph has to be traversed to see if any of the fields, methods or classes in the blacklist is used by the methods included in the call graph. Furthermore, using the reference analysis mechanism provided by Soot, it has to be verified whether objects referenced by *static* fields in the blacklist are modified indirectly (e.g. by means of an intermediate local reference).

3.4 Dynamic Code Restriction

Restricting code of tenant applications statically is not sufficient because all the malicious operations, which can be detected statically, can be done dynamically as well using the `Reflection API`. Therefore, `Restricted Reflection API` should be used by tenant applications instead of the original Java `Reflection API` (cf. Figure 2). This, however, can reintroduce the vendor lock-in problem that QR1 requires to avoid. Therefore, tenant applications are allowed to use the original `Reflection API` but at deployment-time, the `Deployment Service` asks the `Code Transformer` to transform tenant code such that the original `Reflection API` is replaced by `Restricted Reflection API`. This is feasible and straightforward because the `Restricted Reflection API` has exactly the same package structure and exposes exactly the same API as the original one but with different behavior in some cases. The `Code Transformer` component employs the transformation utilities provided by the Spoon framework [28].

The behavior difference is summarized by Listing 1 where `this` either refers to a `Field` object whose `get` method is called, a `Constructor` object whose `newInstance` method is invoked or a `Method` object whose `invoke` method is called. Determining whether the use of these class members are blacklisted for tenant applications, the same `BlacklistService`, which maintains the blacklist, is used. Since the blacklist is prepared once in the entire life-time of the application and kept in memory for subsequent uses, this does not impose a significant performance overhead (cf. QR2).

As shown, Java `SecurityManager` is used to check whether `evadeBlacklist` permission is granted to the OSGi bundle requesting the re-

```
SecurityManager sm = System.getSecurityManager();
if (BlacklistService.isBlacklisted(this))  {
  if (sm != null) {
    sm.checkPermission(new
      ReflectPermission("evadeBlacklist"));
  }
}
super.originalMethod(...); // pseudocode
```

Listing 1 Restricting access of tenant applications to Java Reflection API.

flective operation. This permission has to be granted only to trusted bundles, i.e. not to bundles containing code of tenant applications. The `checkPermission` method throws an `AccessControlException` if the required permission is not granted to the bundle requesting the reflective operation.

The next section elaborates on how the permission checking mechanism of the Java `SecurityManager` works and on how it is employed by the tenant isolation framework.

3.5 Permission Checks

In addition to the above permission checking, the tenant isolation framework enforces permission checks on invocations of `System.exit` method (cf. FR4) and on every IO access (cf. FR5). Permission checks are automatically done by the Java platform itself once the `SecurityManager` is enabled. The role of the framework, hence, is limited to enabling the `SecurityManager` and granting permissions properly.

The `SecurityManager` relies on a mechanism called *stackwalking* and associates a permission set to each *protection domain* (cf. [25, 14]). In OSGi, there exists one protection domain for each bundle. The set of permissions of each tenant application is granted to it by the `Deployment Service` at deployment time. Tenant permission sets are, in principle, a combination of `FilePermission` and `SocketPermission` for restricting their access to IO resources which is required by FR5. By denying the `RuntimePermission("exitVM")` to tenant bundles, FR4 is also fulfilled. All other bundles, which do not contain tenant-provided code, are granted `AllPermission`.

When permission p is required, the Java platform `SecurityManager` triggers stackwalking, i.e. tracing the entire method invocation stack which has led to the point of permission checking, and verifies that

$$\forall_{pd \in PD_s} p \in P_{pd} \tag{1}$$

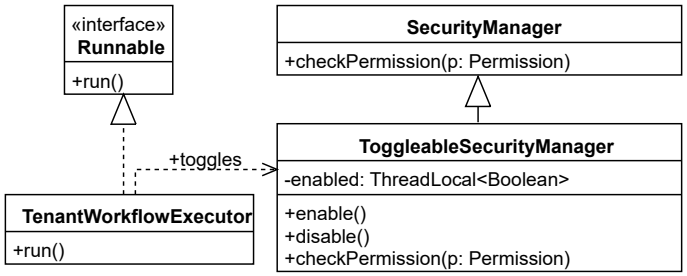


Fig. 3 The framework extends the `SecurityManager` such that it can be enabled only when tenant-provided code is executed. This is done by means of a `ThreadLocal` variable toggled before and after a tenant workflow executes.

```
if (enabled.get()) {
  super.checkPermission(p);
}
```

Listing 2 Evading permission check when it is not required.

where PD_s is the set of all protection domains involved in the scanned method invocation stack s and P_{pd} is the set of permissions granted to the protection domain pd. This way, the permissions of tenant application for whom permission p has to be checked are taken into account and retrieved from its corresponding protection domain.

Permission checks are not required when the running code is trusted, e.g. when PaaS management and monitoring components are executed. In order to avoid the performance overhead of the `SecurityManager` when it is not needed, the framework enables it only when tenant application code is executed and disables it otherwise. The `TenantWorkflowExecutor`, which is responsible for starting/resuming/aborting a tenant workflow, toggles the `ToggleableSecurityManager` shown in Figure 3 before and after workflow execution using `enable` and `disable` methods of the latter. These methods change the value of a `ThreadLocal` variable which is used in the `checkPermission` method according to Listing 2.

3.6 Resource Consumption Control

Fulfilling FR6 requires associating a service-level agreement (SLA) to each tenant application. Listing 3 shows the structure of a tenant SLA. The framework uses the Java Resource Consumption Management API (JSR-284) [5] for imposing limits on resource consumption of each tenant application. On a per-tenant basis, `ResourceMeter` instances are created for each element of the SLA. Meters are notified by the Java platform every time their corresponding resources are allocated or released. The `Resource Consumption Controller` creates tenant meters only once (the first time they are needed) by consulting the `Tenant Awareness` component which has access to SLAs. Once meters are created, they

```
public class TenantSLA {

  private long maxCpuUsage; // CPU nanoseconds per second
  private long maxMemoryUsage; // bytes

  private int maxOpenFiles;
  private long maxReadDiskRate; // bytes per second
  private long maxWriteDiskRate; // bytes per second

  private int maxOpenSockets; // TCP sockets
  private long maxReadSocketRate; // bytes per second
  private long maxWriteSocketRate; // bytes per second

  private int maxOpenDatagrams; // UDP datagrams
  private long maxReadDatagramRate; // bytes per second
  private long maxWriteDatagramRate; // bytes per second

  // getters and setters

}
```

Listing 3 Structure of Tenant SLA.

```
ResourceContextFactory factory = ResourceContextFactory.
                                                  getInstance();
ResourceContext rc = factory.lookup(tenantId);
rc.bindThreadContext(); // binds to the current thread
// workflow execution code
rc.unbindThreadContext(); // unbinds from the current thread
```

Listing 4 Binding tenant `ResourceContext` to threads before starting workflow execution.

will be associated with a tenant-specific `ResourceContext`. Each time a workflow execution thread starts for a tenant, the `TenantWorkflowExecutor` enforces resource control measures according to Listing 4.

The `BoundedMeter`, provided by the Java platform, is the meter type used for `maxMemoryUsage`, `maxOpenFiles`, `maxOpenSockets` and `maxOpenDatagrams`. This meter type, which imposes a fixed limit on usage amount, is used by the framework in a such way that no tenant can go beyond predefined limits of its SLA. This is because once these types of resources are allocated beyond SLA limits, there is no guarantee that they can be released. For instance, if a memory leak bug in the tenant application is the cause of reaching the memory limit, it is possible that the memory usage of that specific tenant application never falls into a decreasing trend (cf. [13, 15, 30]).

For the remaining SLA elements, the framework imposes a more flexible control (cf. FR7) by employing the `ThrottledMeter` of the Java platform. This meter type first consults a specified `ResourceApprover` to see if resource al-

location is possible before throttling the usage. If resource allocation beyond SLA limits is not safely possible, the usage rate will be throttled to the predefined rate in the tenant SLA. For checking whether resource allocation beyond SLA limits is safely possible, the framework provides a `SafeGreedyApprover` which approves any requested amount insofar as the system is in overall healthy condition. For approving the requested amount, the latter checks the following condition:

$$total_r + amount_r \leq l_r \times capacity_r \tag{2}$$

where r is a resource of concern, $total_r$ is the total consumption amount of all tenants before approving the new request, $amount_r$ is the requested amount, $capacity_r$ is the total capacity of the system on r and l_r is the leniency factor between 0 and 1. When leniency factor is set to zero, every tenant will be strictly restricted to its SLA regardless of resource availability, i.e. unallocated amounts. When it is set to one, the unallocated amounts will be used for letting more active tenants going beyond their SLA limits.

The `SafeGreedyApprover` consults the `System Health Monitoring` component for retrieving the total usage. Capacity is a set by system administrators while total usage is retrieved from the `ResourceContextFactory` provided by the Java platform. The latter does not calculate the total usage every time queried. Instead, it keeps track of total amounts on every resource allocation and release. Decisions of this approver are safe because they are made about resources measured on a per-second basis and surpassing their limits will have no effect beyond the measurement window which is two seconds according to the API. In other words, it is guaranteed that these resources can be throttled back to the limits defined in tenant SLAs once the load on the system increases.

4 Related Work

In this section, we briefly compare this paper with related work.

PaaS Offering of Workflow Engines. Pathirage et al. [27] proposes an architecture for PaaS offering of Apache ODE [10] which is a workflow engine complying with WS-BPEL. Since all activities involving code execution are remote web-services, the security threats that we covered in this paper are not relevant for that work. However, delegating execution of all untrusted activity code to remote web-services both reduces efficiency of resource utilization and imposes the overhead of remote invocation (e.g. network delay and serialization). Yu et al. [34] claims having enabled jBPM [7], a BPMN2-compliant engine, to be offered as a PaaS. However, the security threats discussed in this paper are overlooked altogether. Amazon SWF [2] and Fantasm on Google App Engine [4] are production ready PaaS offerings of workflow engines. However, applications developed using them highly suffer from vendor lock-in problem in terms of both

code and data. Furthermore, resource utilization is sub-optimal due to adoption of OS-level virtualization tactics.

OS-level Virtualization. Similar isolation measures can be imposed by means of containers (cf. [12]) or virtual machines (cf. [21]). However, in case of Java, a separate instance of the JVM should be started for each tenant which reduces efficiency of resource utilization. Furthermore, these solutions are totally insufficient for offering BPMN2-compliant engines as the latter runs `Script Task` and, in some cases, `Service Task` activities in the same OS process as the workflow engine itself.

Java Language Vulnerabilities. Security threats related to running untrusted code in Java threads are discussed in [13, 15, 30]. These problems are caused by the shared nature of *static* fields, blocking effect of *static synchronized* methods, reference leaks and shared nature of computational resources. Our threat analysis is based on these works and we have proposed ideas for solving some of them and workarounds for some others based on existing technologies.

Service-level Tenant Isolation. As opposed to OS-level virtualization, service-level tenant isolation for PaaS is rather new and not extensively adopted in practice. Krebs et al have proposed a framework for constraining resource consumption at service-level [19]. However, their solution relies on estimation of tenant resource consumption from external properties of the system such as response-time using statistical methods such as linear regression. Our solution, on the contrary, is based on accurate measurements of the execution platform instead of estimated consumption levels from external properties. Rodero et al. [30] has hinted at the possibility of using the Java Resource Consumption Management API (JSR-284) [5] for solving the isolation problem vis-á-vis resource consumption but *how* this API can be used for the purpose of tenant isolation is left open. Even though they state reference leaks as an unsolvable problem in Java environments, imposing strict memory limits for each tenant application can prevent reference leaks of one tenant application becoming problematic for other tenants. To the best of our knowledge, Oracle Weblogic Server Multitenant [26] is the only attempt in practice to enforce tenant isolation measures at service-level of a middleware. However, it only provides resource access control and *limited* resource consumption control. The resource consumption mechanism in this middleware does not support disk and network bandwidth isolation, meaning that one tenant application may take up the entire network bandwidth, for instance, and disrupt the service for other tenants. Furthermore, the static and dynamic code restriction mechanisms presented in this paper are not supported by this application server which leaves tenants unprotected against threats related to vulnerabilities of the Java language mentioned in the previous paragraph.

Application Performance Isolation. There are a number of works dealing with performance isolation for Software-as-a-Service (SaaS) applications [32, 17, 18, 22]. While these works deal with SLAs expressed in external properties of an application such as response-time and throughput, the SLAs are defined in system-level terms such as CPU usage. This is because these works do not deal with the issues of running untrusted code in a shared execution environment.

5 Conclusion

Business process automation is a common practice supported by a set of mature standards (e.g. BPMN 2.0 and WS-BPEL) and numerous workflow engines that implement these standards. Due to the specific deployment model of multi-tenancy in a Platform-as-a-Service (PaaS) context, full support of these standards requires additional attention to security threats caused by misbehaving tenants. We have presented an outline of a framework for tenant isolation in the context of co-existing business processes of different tenants, and we have discussed its practical feasibility for the Java environment.

Advancing this work fits into our ongoing research on the key trade-offs related to multi-tenancy between resource optimization, customization support (e.g. by means of tenant-provided tasks), security (tenant isolation) and portability of business processes across different cloud providers. We have implemented the permission checking and resource consumption mechanisms of the framework. In follow-up work, we will further implement the code restriction part and evaluate the proposed framework vis-à-vis performance overhead compared to the OS-level virtualization approach and dimension of tenant code restrictions (i.e. determining how limited tenant applications will be in using the Java API given the restrictions imposed by the framework).

References

1. Activiti User Guide. https://www.activiti.org/userguide/. Accessed: 2017-05-24
2. Amazon Simple Workflow Service (Amazon SWF). https://aws.amazon.com/documentation/swf/. Accessed: 2017-06-12
3. Business Process Model and Notation 2.0. http://www.omg.org/spec/BPMN/2.0/PDF/. Accessed: 2015-08-04
4. Google App Engine Fantasm. https://cloud.google.com/appengine/articles/fantasm. Accessed: 2017-06-12
5. JSR 284: Resource Consumption Management API. https://jcp.org/en/jsr/detail?id=284. Accessed: 2017-06-12
6. List of BPMN Engines. https://en.wikipedia.org/wiki/List_of_BPMN_2.0_engines. Accessed: 2017-07-05
7. RedHat JBoss jBPM. http://www.jbpm.org/. Accessed: 2017-06-04
8. Web Services Business Process Execution Language. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. Accessed: 2016-06-04
9. Alliance, O.: OSGi specification. https://osgi.org/download/r4v43/osgi.core-4.3.0.pdf (2012). Accessed: 2017-04-19
10. Apache: Apache ode. http://ode.apache.org/. Accessed: 2017-07-09
11. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al.: A view of cloud computing. Communications of the ACM **53**(4), 50–58 (2010)
12. Bernstein, D.: Containers and cloud: From lxc to docker to kubernetes. IEEE Cloud Computing **1**(3), 81–84 (2014)
13. Czajkowski, G., Daynés, L.: Multitasking without comprimise: a virtual machine evolution. In: ACM SIGPLAN Notices, vol. 36, pp. 125–138. ACM (2001)

14. Gong, L., Ellison, G.: Inside Java (TM) 2 Platform Security: Architecture, API Design, and Implementation. Pearson Education (2003)
15. Herzog, A., Shahmehri, N.: Problems running untrusted services as java threads. Certification and Security in Inter-Organizational E-Services **177**, 19–32 (2004)
16. Ko, R.K., Lee, S.S., Wah Lee, E.: Business process management (bpm) standards: a survey. Business Process Management Journal **15**(5), 744–791 (2009)
17. Krebs, R., Loesch, M., Kounev, S.: Platform-as-a-service architecture for performance isolated multi-tenant applications. In: Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on, pp. 914–921. IEEE (2014)
18. Krebs, R., Momm, C., Kounev, S.: Metrics and techniques for quantifying performance isolation in cloud environments. Science of Computer Programming **90**, 116–134 (2014)
19. Krebs, R., Spinner, S., Ahmed, N., Kounev, S.: Resource usage control in multi-tenant applications. In: Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on, pp. 122–131. IEEE (2014)
20. Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The soot framework for java program analysis: a retrospective. In: Cetus Users and Compiler Infastructure Workshop (CETUS 2011), vol. 15, p. 35 (2011)
21. Li, Y., Li, W., Jiang, C.: A survey of virtual machine system: Current technology and future trends. In: Electronic Commerce and Security (ISECS), 2010 Third International Symposium on, pp. 332–336. IEEE (2010)
22. Lin, H., Sun, K., Zhao, S., Han, Y.: Feedback-control-based performance regulation for multi-tenant applications. In: Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on, pp. 134–141. IEEE (2009)
23. Microsoft: The stride threat model. https://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx (2015). Accessed: 2017-04-19
24. Opara-Martins, J., Sahandi, R., Tian, F.: Critical review of vendor lock-in and its impact on adoption of cloud computing. In: Information Society (i-Society), 2014 International Conference on, pp. 92–97. IEEE (2014)
25. Oracle: Java 8 SE platform security. https://docs.oracle.com/javase/8/docs/technotes/guides/security/overview/jsoverview.html. Accessed: 2017-04-19
26. Oracle: Oracle weblogic server multitenant. https://docs.oracle.com/middleware/1221/wls/WLSMT/. Accessed: 2017-07-09
27. Pathirage, M., Perera, S., Kumara, I., Weerawarana, S.: A multi-tenant architecture for business process executions. In: Web services (icws), 2011 ieee international conference on, pp. 121–128. IEEE (2011)
28. Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., Seinturier, L.: Spoon: A library for implementing analyses and transformations of java source code. Software: Practice and Experience **46**(9), 1155–1179 (2016)
29. Rimal, B.P., Choi, E., Lumb, I.: A taxonomy and survey of cloud computing systems. INC, IMS and IDC pp. 44–51 (2009)
30. Rodero-Merino, L., Vaquero, L.M., Caron, E., Muresan, A., Desprez, F.: Building safe paas clouds: A survey on security in multitenant software platforms. computers & security **31**(1), 96–108 (2012)
31. Shostack, A.: Threat Modeling: Designing for Security. Wiley (2014)
32. Walraven, S., De Borger, W., Vanbrabant, B., Lagaisse, B., Van Landuyt, D., Joosen, W.: Adaptive performance isolation middleware for multi-tenant saas. In: Utility and Cloud Computing (UCC), 2015 IEEE/ACM 8th International Conference on, pp. 112–121. IEEE (2015)
33. Walraven, S., Truyen, E., Joosen, W.: Comparing paas offerings in light of saas development. Computing **96**(8), 669–724 (2014)
34. Yu, D., Zhu, Q., Guo, D., Huang, B., Su, J.: jbpm4s: A multi-tenant extension of jbpm to support bpaas. In: Asia-Pacific Conference on Business Process Management, pp. 43–56. Springer (2015)

CEP-based SLO Evaluation

Kyriakos Kritikos[1], Chrysostomos Zeginis[1]
Andreas Paraboliasis[2], and Dimitris Plexousakis[1]

[1] Institute of Computer Science - FORTH, Greece
{kritikos,zegchris,dp}@ics.forth.gr
[2] Computer Science Department, University of Crete, Greece
csd3031@csd.uoc.gr

Abstract. Modern service-based applications (SBAs) operate in highly dynamic environments where both underlying resources and the application demand can be constantly changing which external SBA components might fail. Thus, they need to be rapidly modified to address such changes. Such a rapid updating should be performed across multiple levels to better deal, in an orchestrated and globally-consistent manner, with the current problematic situation. First of all, this means that a fast and scalable event generation and detection mechanism should exist to rapidly trigger the adaptation workflow to be performed. Such a mechanism needs to handle all kinds of events occurring at different abstraction levels and to compose them so as to detect more advanced situations. To this end, this paper introduces a new complex event processing framework able to realise the respective features mentioned (processing speed, scalability) and have the flexibility to capture and sense any kind of event or event combination occurring in the SBA system. Such a framework is wrapped in the form of a REST service enabling to manage the event patterns that need to be rapidly detected. It is also well connected to other main components of the SBA management system, via a publish-subscribe mechanism, including monitoring and the adaptation engines.

Keywords: complex event processing, event pattern, detection, service

1 Introduction

Due to tough competition, organisations can survive if they can improve their services to exhibit better service levels with less cost. Such organisations need to also possess a smart infrastructure and a dedicated devops team to appropriately re-configure the services offered as well as manually intervene in unanticipated, problematic situations. As such, a lot of effort is spent in maintaining such an infrastructure while an increasing management and operational cost also incurs.

Fortunately, the advent of cloud computing has revolutionarised the way resource management is performed. Nowadays, organisations can outsource their infrastructure management to cloud providers that promise to offer infinite, cheap commodity resources on an on-demand basis. Due to flexible resource management and the capability to scale a cloud-based system, organisations

can now optimise their services at the infrastructure level. However, still effort is needed at higher-levels of abstractions. In particular, external SaaS services need to be dynamically selected to realise part of the required functionality while the whole system needs to be adapted.

In the literature, it has been advocated [8] that dynamic SBA adaptation should be performed in a cross-layer manner by also putting in place, as a pre-requisite, a suitable monitoring framework. Cross-layer adaptation is needed for various reasons. First, as the service system itself includes multiple levels that must be appropriately controlled. Second, as the individual adaptation at one level can influence, impact or even negate the adaptation results at adjucent levels, leading to a vicious re-adaptation cycle. Cross-layer monitoring is also needed to propagate and aggregate up to higher-levels measurements produced in lower levels so as to cover measurability gaps.

As the glue between monitoring and adaptation, there is a need for a rapid and scalable Service Level Objective (SLO) evaluation framework able to transform measurements to events and subsequently detect event patterns that can lead to performing adaptation actions in the context of adaptation rules. Such a framework should also exhibit suitable accuracy levels by correctly correlating the events occurring based on their metrics and measured objects. It should also be able to detect and correlate events which should map to both the type and instance level in the managed SBA system.

In this work, such a framework has been carefully designed and realised, by conforming to all the aforementioned requirements. In particular, the framework architecture was initially designed by considering principles, such as service-orientation, and by carefully decoupling framework parts subject to scaling. Based on this architecture and the appropriate selection of the right, existing components and tools, a respective framework was then implemented and integrated in our existing SBA monitoring and adaptation framework []. Such an integration is loosely coupled as our SLO evaluation framework can be in principle connected to any monitoring and adaptation engine.

The developed framework relies on the CAMEL domain-specific language (DSL), able to capture various aspects in the cloud-based application lifecycle management, including the monitoring and adaptation ones. In particular, this DSL is expressive enough to specify complex event patterns, where each event maps to a metric condition, and associate them with respective sets of adaptation actions that must be triggered to adapt the SBA in a cross-layer manner. CAMEL also covers well the monitoring aspect via its capability to specify how composite metrics are aggregated and to associate metrics with the (e.g., service) component that they measure. As it will be shown, such information is essential to have the ability to correlate events in the context of event pattern detection.

The proposed framework relies on the Esper Complex Event Processing (CEP) engine. This engine is quite scalable with the capability to process thousands or even millions of events. Due to the way our architecture has been designed, this engine can be scaled when its processing limits are reached, enabling our framework to really scale at those parts where most of the load is directed.

To evaluate the proposed framework scalability, a series of synthetic experiments have been performed. Such experiments show some limitations of the currently centralised framework deployment and thus the exact points where such a framework should scale.

The rest of the paper is structured as follows. The next section reviews the related work. Section 3 provides background information necessary for the comprehension of this paper contribution. Section 4 analyses the proposed framework architecture and supplies some implementation details. Section 5 explains the way the event pattern specification is generated by accounting also on how the events of the pattern should be correlated. Section 6 supplies and discusses the experimental framework evaluation results. Finally, the last section concludes the paper and draws directions for further research.

2 Related Work

Various approaches have been proposed in complex event processing and event pattern detection. Most rely on CEP engines that detect complex events continuously and build correlations and relationships between them, such as causality and timing ones. The detection of complex patterns is based on various techniques applied either over event streams [16][15] or in an offline [10][7] manner.

Statistical event detection approaches mainly exploit a user-defined minimum frequency or support (*minsup*). The springboard of all these approaches is the *Apriori* algorithm [1]. This algorithm produces the set of all significant association rules (rules relating a set of variables) between items in a large transactions database with a *minsup*. In [13], the authors introduce a method for discovering frequent event patterns, as well as their spatial and temporal properties in sensor networks, exploiting data mining techniques. Provided that events are put into a spatial and temporal context, the authors correlate certain event types on a sensor node with context events in a confined neighborhood in the recent past. Thus, a pattern of events is discovered whenever this pattern's frequency surpasses a *minsup*. In [11], the authors propose the *Lossy Counting* widely used algorithm. This is an one-pass algorithm that computes approximate frequency counts of elements in a data stream and involves grouping the row items into blocks or chunks and counting within each chunk.

Temporal event processing approaches exploit the temporal relations among an input stream's events. Such approaches can be very useful for deriving implicit information for the temporal ordering of raw data and predicting the future behavior of the monitored application. In [3] the authors introduce a formal framework for expressing data mining tasks involving time granularities, as well as algorithms for performing these tasks. Time constraints are injected into the system to bound the distance between an event pair in terms of time granularity. For instance, event e_2 must happen within two minutes after the occurrence of event e_1 so as to consider e_1, e_2 an event pattern. In [12] a temporal data mining approach is presented for data that cannot fit in memory or are processed at a

faster rate than the generation one. The proposed sliding window model slides forward in hops of batches, while only a single batch is available for processing.

Finally, logic-based approaches exploit inferencing to discover patterns defining respective association rules. In [14] a pattern discovery approach is proposed mapping logical equivalences based on propositional logic. In particular, a rule mining framework is introduced, generating coherent application domain independent rules for a given dataset that do not require setting an arbitrary *minsup*. The logic-based approach in [2] proposes an event calculus (EC) dialect, called *RTEC*, for efficient run-time recognition that is scalable to large data streams and exploits main EC predicates to discover specific activities. Finally, our previous work [17] has introduced a logic-based algorithm for discovering valid event patterns causing specific SLO violations that can be further exploited to enrich the adaptation rules defined by experts.

3 Background

3.1 Esper

Esper[3] is a stream-oriented CEP engine that provides the SQL-like and rich Event Processing Language (EPL). EPL enables expressing complex (event) matching conditions that include temporal windows, joining of different event streams, as well as filtering, aggregation, sorting and pattern detection. The proposed framework exploits it for the event pattern detection.

3.2 CAMEL

CAMEL is a multi-DSL, developed in the context of the PaaSage[4] project to deal with the specification of multiple aspects in the multi-cloud applications lifecycle. It integrates already existing languages, like CloudML[6], as well of new languages developed with that project, like the Scalability Rule Language (SRL)[9]. The aspects covered by CAMEL mainly include: deployment, requirement, metric, scalability, provider and organisation aspects.

This paper focuses mainly on the metric and scalability aspects covered by the SRL sub-DSL of CAMEL. The metric package attempts to cover all measurement details that need to be specified for a non-functional metric, like formulas, functions, units of measurement plus measurement schedules and windows. This package is also able to specify conditions on metrics that can be exploited to specify SLOs as well as non-functional events in scalability rules.

The scalability aspect is covered via specifying scalability rules that map single events or event patterns to one or more scaling actions. Scaling actions can be either horizontal or vertical. Horizontal scaling actions include scale-out and scale-in actions while vertical actions include scale-up and scale-down.

[3] `http://www.espertech.com/esper/`

[4] `https://paasage.ercim.eu/`

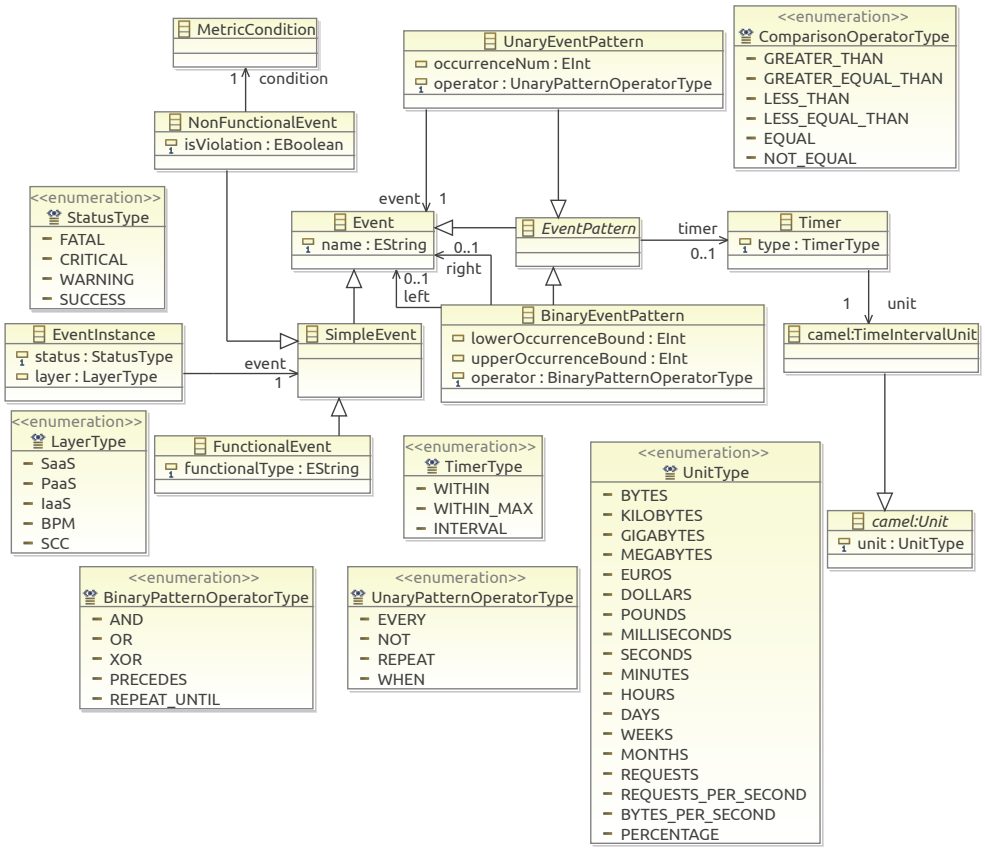


Fig. 1: The event pattern part of the SRL meta-model.

The conceptualisation of events and event patterns is depicted in Figure 1. Events can be single or composite. Single events can be further distinguished in functional and non-functional. Functional events map to a certain functional fault, like an application component failure. Non-functional events are associated to a metric condition violation. A composite event maps to a logical or time-based combination of one or more events in the form of an event pattern. As such, such a combination is associated with respective logical and time-based operators. Both binary and unary operators can be defined which leads to producing unary and binary event patterns, respectively. Logical operators include AND, OR, NOT and XOR. Time-based operators have been inspired by Esper's EPL and include many of the operators defined in that language (e.g., REPEAT).

As an event pattern is also a kind of event, patterns can be recursively defined. This means that, for example, when applying a binary logical operator (e.g., AND) over a certain binary event pattern, the first event could be single and the second could be another event pattern. For instance, suppose that the event pattern EP_1: $A \wedge (B \vee C)$ must be defined. To specify EP_1, we need to define that the first event is A, the second event maps to the event pattern EP_2 and that the logical operator applied is $\wedge$. The second event pattern EP_2 would then be specified as the application of the $\vee$ operator over two events, B and C.

Via the recursive definition of events, more complex and advanced situations can be captured in respective rules. This should not stop to the case of scalability rules, but could cover any adaptation rule kind. This has been performed by the CloudSocket project[5] [5] via an SRL extension. This extension can specify any adaptation rule kind at different abstraction levels. The event part of the rule specification was left as is, but the action part was extended to specify a workflow of adaptation actions that can be performed at the levels of infrastructure, platform, service and business process. As such, this extension fits well to the latest research trends in service computing that require specifying, executing and managing cross-layer rules to more effectively deal with the adaptation of cross-level SBAs in both simple and more advanced problematic situations.

In the context of this work, only the event part of an adaptation rule is considered due to intended functionality to be delivered. The CEP engine developed just detects the need to trigger a rule and then informs the rule execution component, e.g., an *Adaptation Engine*, to enact that execution of that rule.

4 SLO Evaluation Framework

4.1 Framework Analysis

The proposed SLO Evaluation Framework relies on the modular architecture depicted in Figure 2. This architecture comprises three main levels: (a) interface; (b) core logic; (c) database (DB). At the interface level, the main actions (add, update, delete) that can be performed over an event pattern (EP) have been wrapped into the form of a REST service, called, *EP Service*, able to parse CAMEL/SRL fragments mapping to the specification of these patterns. Each action, when called, then has an impact over the core logic level of the framework.

At this second level, there is a main component, called *EP Parser*, which is responsible for processing the EPs obtained from the *EP Service*. Depending then on the action requested, different interactions take place at this level.

EP Addition. In case of adding a new EP, the *EP Parser* transforms it into an EP, specified in the EP language of the CEP framework, which is then registered in the server of that CEP framework, called *CEP Server*, so that it can be immediately detected. The names of metrics referenced by the EP, i.e., directly involved in the conditions of the EP's events, are also sent to the *Metric Subscriber* which not only informs its local metric list but also registers for subscribing to such metrics, when they are new, in the *Metric Publisher*. In parallel to this registration, the updated metric list of the *Metric Subscriber* is stored in the *EP DB* for fault-tolerance and rapid recovery reasons. The *Metric Publisher* is responsible for publishing the values of metrics monitored to potential subscribers. As such, it can well map to a *Monitoring Engine* of a SBA management system. Once both the new EP and its respective metrics are registered in the corresponding system parts, the EP addition has been successful. So, the *EP Parser* stores the new EP in the *EP DB* not only for recovery reasons but also

[5] `www.cloudsocket.eu`

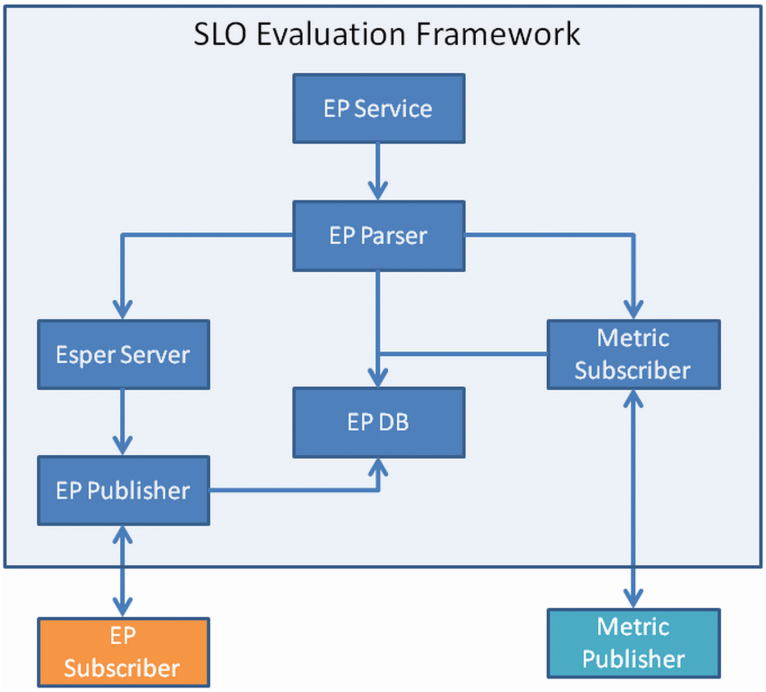


Fig. 2: The architecture of the SLO Evaluation Framework.

to gather statistics about EPs, while being detected by the *Esper Server*. The *EP DB* has been realised in the form of a model repository able to store, query and manipulate models of CAMEL, especially EPs, along with their statistics.

EP Deletion. In case of EP deletion, the EP is first fetched from the *EP DB*. Then, in parallel, the *EP Parser* informs both the *CEP Server* and the *Metric Subscriber* to update their structures and take further actions. The *CEP Server* just deregisters the EP's EPL specification. On the other hand, after checking that the EP metrics to be removed are not exploited in other EPs, the *Metric Subscriber* is informed to unsubscribe to these metrics to reduce the system load.

EP Update. In case of EP update, the produced EPL statement by the *EP Processor* is used to update the previous one. In addition, the *Metric Subscriber* is informed for adding or removing metrics which are or not needed any more (by any EP), respectively.

While the above actions can take place through the interaction of an external agent / user with the proposed Framework, we highlight that, in principle, the same interactions could be differently achieved, e.g., via a publish-subscribe mechanism. As the respective functionality has been realised, we could easily switch from one to another mechanism or have both available at the same time.

As there are internally performed actions inside the framework, while it is running, these are now explicated in detail below.

As the *Metric Subscriber* subscribes to metrics, it can asynchronously receive measurements for such metrics from the *Metric Publisher*. Such measurements are then transformed into events which are fed into the *CEP Server*. Once all

suitable events are received by the latter component, it can detect one or more EPs. When this occurs, this component will inform the *Event Publisher*.

The *EP Publisher* is responsible for publishing events to interested subscribers, named as *EP Subscriber*s. Such subscribers could be adaptation engines responsible for executng the respective adaptation rule triggered, as, e.g., specified in CAMEL. Apart from this publication, the *EP Publisher* also updates the entries in the *EP DB* to modify the respective statistics of the EP(s) concerned.

The proposed architecture exploits publish-subscribe mechanisms to both receive some events/measurements and publish other kinds of events (e.g., EPs). In this respect, it can actually interact with multiple components that might be willing to obtain information from or feed information to this framework. For instance, adaptation responsibility for an SBA management system could be split into multiple instances of an *Adaptation Engine* to balance the respective load. All these instances could then subscribe to the *EP Publisher* to manage their own part of the adaptation space, i.e., only those EPs that concern them.

The presented architecture is logical. This means that it can be flexibly distributed at the physical level. For instance, we could have multiple instances of the framework part that involves the *CEP Server* and the *Event Publisher* to load balance the event workload entering the framework. Alternatively, we could scale out the whole framework into parts that focus on different EP partitions. For example, the SBA management system could be split similarly into different parts, where each part could be devoted to a subset of all SBAs managed. Each system part could be then associated to one instance of the SLO Evaluation Framework, thus mapping only to the EPs of the SBAs that need to be handled.

4.2 Implementation

All framework components have been implemented in Java. The CEP engine exploited is Esper, version 5.3.0. For the publish-subscribe mechanism, the 0-MQ[6] messaging middleware has been exploited that incurs less overhead with respect to other messaging middleware realisations. The *EP DB* has been realised as a model repository implemented via the CDO technology[7] which provides suitable and robust mechanisms for model persistence and lazy loading as well as the HQL language to enable posing queries at a higher abstraction level than pure SQL. The *EP Service* has been implemented via the Jersey[8] java library.

5 Event Pattern Generation & Detection

While it could be considered as straightforward to transform an event pattern in CAMEL into an EPL statement in Esper, this is by far not trivial as the events in an EP need to be correctly correlated. Correlation means that the events should be associated with either the same measured objects or with objects that

[6] `zeromq.org`
[7] `https://eclipse.org/cdo/`
[8] `http://jersey.github.io/`

are connected in the SBA dependency hierarchy. This also has an impact on the way measurements are represented as the information concerning the measured object should be already present and be then copied accordingly in the internal representation of the event in Esper.

Concerning the metric measurements, we have actually assumed the following: (a) the *Metric Subscriber* subscribes only to metrics based on their name; (b) the *Metric Publisher* publishes measurements for metrics that might be named equivalently. The latter means that the measurement information published should include sufficient information to enable the framework to identify exactly what object is being measured.

To decouple the proposed framework from the dependency knowledge it should possess, we assume that such dependency information is provided within the measurement information published. While this leads to some published information duplication, it translates to a loose integration of this framework with the SBA management system. Otherwise, the framework would need to connect to a models@runtime component [4] in that system to be informed constantly about both the type and instance level in the SBA dependency hierarchy.

The measurement information published includes: (a) the metric's name (e.g., MeanResponseTime); (b) the metric value; (c) the measurement timestamp; (d) the name of the application/service concerned; (e) the name of the component measured; (f) the name of the instance of the component measured; (g) the name of the VM measured; (h) the name of the instance of the VM measured.

Values for fields (a)-(d) are always present. Depending on the level and kind of component measured, only some of the values of the other fields need to be supplied. For instance, when a measurement at the type level has been produced (i.e., mapping to an aggregation), then, if it concerns a certain component, we also need to specify the VM in which that component should have been deployed. This is required to distinguish for which deployment of a component, the measurement should hold. If the VM is being measured at the type level, then we just need the name of the VM without referring to any application component name. Similar logic applies to the instance level with the sole exception that we also need to cover the type level as the framework does not have the knowledge about which is the type of a particular instance. For example, if a measurement for an instance of an application component is obtained, then we need to indicate also the name of that component. In this case, we also need to specify the instance of the VM on which the component instance has been deployed as well as that instance's type. This is the case where all fields should be filled in.

Now, we are going to explain the way EPs in CAMEL are transformed to EPL statements. We distinguish between two cases: (a) all events in the EP refer to the instance level; (b) all events in the EP refer to the type level. We do not consider a mixture of events from different levels as this does not make sense.

In case that the instance level is covered, there are actually two cases: (i) all events refer to the same component; (ii) all events refer to different but related components. In sub-case (i), suppose that we need to correlate two events that refer to the same VM instance. Further suppose that event E_1 highlights that

the raw CPU utilisation of this VM instance is above 80%, and the other, E_2 indicates that the raw memory utilisation for the same VM instance is above 90%. Finally, suppose that an EP EP_1 needs to be detected which requires the logical conjunction of these two events, i.e., $E_1 \wedge E_2$. The respective EPL statement to be created would be the following:

```
every(ev1=Event(metric='CPUUtilisation' and value >= 80 and application='APP1' and vm='VM1') and Event(metric='MemoryUtilisation' and value >= 90 and application='APP1' and vmInstance=ev1.vmInstance and vm='VM1'))
```

In this statement, we join these two events as streams based on their application, VM and VM instance fields. Via this join, we impose that the EP should hold for a specific application and VM but we do not care about which matched vm instance is concerned (as any instance needs to be matched here). Moreover, the presence of EVERY indicates that the pattern should be repeatedly inspected and not just once.

In sub-case (ii), we need to correlate the different kinds of components together. As such, suppose that a slighly different EP EP_2 needs to be detected mapping to the logical conjunction of the first event E_1 (see above) and a new event E_3 signifying that the raw response time of any instance of component $C1$ is greater than 20 seconds. The EPL to be constructed will be as follows:

```
every(ev1=Event(metric='CPUUtilisation' and value >= 80 and application='APP1' and vm='VM1') and Event(metric='ResponseTime' and value > 20 and vmInstance=ev1.vmInstance and application='APP1' and vm='VM1' and component='C1'))
```

This EPL statement is more complicated as it needs to join two events for which we need to guarantee that they refer to the same application, VM and VM instance, where the first two fields are mapped to specific values. We also need to guarantee that the second event refers to component $C1$ and we do not care about the instance of that component.

For the type level, we have two similar kinds of cases: (i) simple and (ii) complex. However, as we do not care about the instance level, the EPL statements to be constructed are simpler.

Suppose that in the (i) sub-case, we need to detect an EP EP_3 mapping to the logical conjunction of two events: E_4 associated with a mean response time of above 20 seconds for component $C1$ and E_5 associated with a mean availability of less than 99.99 for $C1$. The EPL statement produced will be the following:

```
every(ev1=Event(metric='MeanResponseTime' and value > 20 and application='APP1' and component='C1') and Event(metric='MeanAvailability' and value < 99.99 and application='APP1' and component='C1'))
```

In this statement we just join two event streams based on their application and component which are clearly identified.

For the (ii) sub-case, suppose that we need to detect an EP EP_4 mapping to the logical conjunction of two events: E_4 as above and E_6 highlighting that the average CPU utilisation is greater than 80% for the $VM1$ hosting the $C1$ component. The EPL statement produced will be the following:

```
every(ev1=Event(metric='MeanResponseTime' and value > 20 and application='APP1' and component='C1' and vm='V1') and Event(metric='MeanCPUUtilisation' and value > 80 and component='C1' and application='APP1' and vm='VM1'))
```

So, we actually join again the two events but now the fields concerned are the application, component and VM ones with specific values provided to them.

The above 2 cases with their 2 sub-cases have been exemplified with certain examples. In reality, our framework is able to go beyond the capabilities shown in these examples. It can process any kind of complex EP with an arbitrary nesting and any kind of operator from those captured by CAMEL. However, showing such a complex case needs substantial space and thus, it has been left out from the analysis in this paper.

6 Evaluation

This section describes an experimental evaluation of our proposed framework's performance based on two main experiments. The setup of the machine where the experiments were conducted was a PC with quad-core Intel(R) CPU at 2,7GHz and 8GB RAM running the Microsoft Windows 10 (64-bit) operating system.

The input of the experiments relies on an *Event (Pattern) Generator* component which, based on a certain simplified deployment model of an SBA, attempts to generate both a set of EPs as well as events. The number of these two sets is configurable via respective control parameters. The EPs are created in different levels of complexity by exploiting all the event composition operators offered by SRL/CAMEL. The events (actual measurements) are generated such that they map to the leaf part of the EP (trees) constructed.

6.1 First Experiment

The first experiment (See Figs. 3 and 4) evaluates the proposed framework's performance, memory usage and CPU load, while increasing the total amount

of events (with a 10000 step) produced by the *Event (Pattern) Generator*. The number of EPs in Esper is predefined (1000). Each measurement is calculated by the average value of 10 iterations while the events are sent in 10-second intervals.

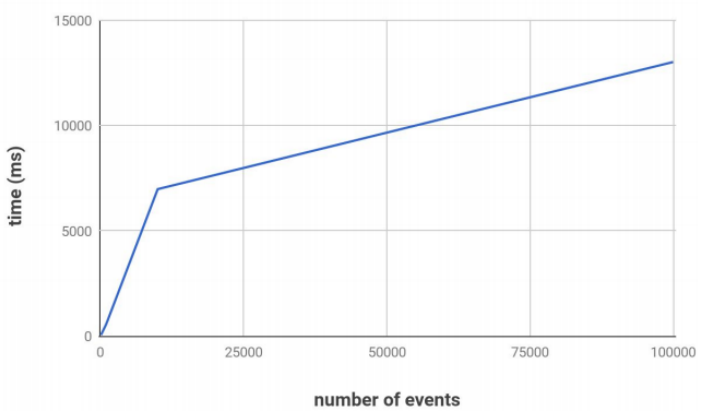
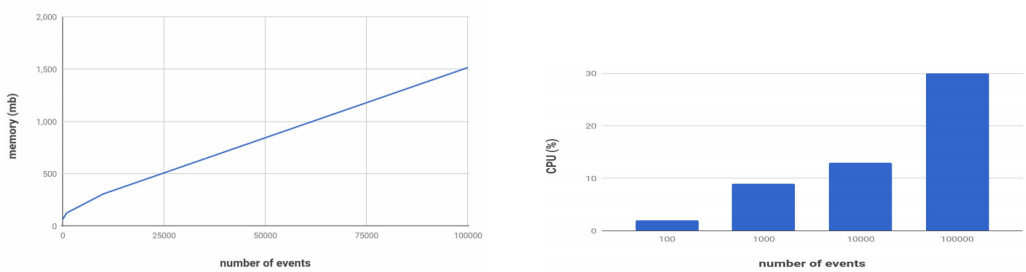


Fig. 3: Performance of the CEP System while increasing the no of events



(a) Memory consumption

(b) CPU usage

Fig. 4: Overall resource utilisation while increasing the no of events

Fig. 3 shows that the processing time increases linearly with the number of events and reaches the bound of almost 13 seconds. The latter seems to be a point for further improvement, possibly by considering the configuration of the Esper CEP. It also indicates that possibly Esper needs to be load-balanced by physically distributing it in two or more nodes.

Concerning memory consumption, see Fig. 4a, the behaviour is also linear. The memory consumption reaches the bound of 1.5 GB, which looks normal and acceptable if we consider the final load imposed to the framework.

CPU utilisation, see Fig. 4b seems to not exhibit a linear behaviour. On the other hand, it reaches only 30% which means that there is still a great potential for processing additional events.

In overall, this experiment signifies that Esper scales well, especially with respect to resource usage, but the event processing time needs to be improved.

6.2 Second Experiment

The second experiment (See Figures 5 and 6) similarly evaluates the event processing time, the memory usage and the CPU load of the proposed framework, while increasing the total amount of EPs stored in the Esper engine to stress its detection functionality. The number of events produced is predefined (100.000). Each measurement is again calculated by the average value of 10 iterations while the events are sent in 10-second intervals.

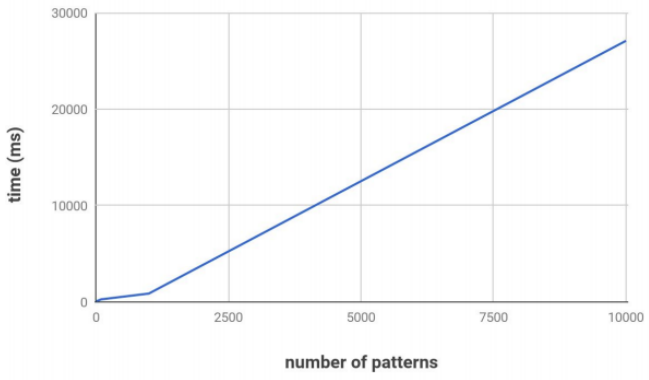
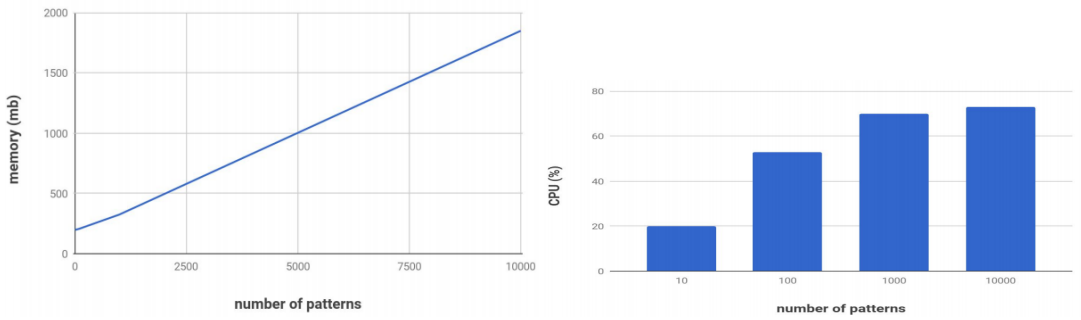


Fig. 5: Performance of the CEP System while increasing the no of stored patterns



(a) Memory Consumption (b) CPU usage

Fig. 6: Overall resource utilisation while increasing the no of patterns

The experiment results are quite similar to those of the previous experiment. The event processing time, see Fig. 5, is again linearly increasing but it is now larger than the corresponding last processing time value in the first experiment,

where the overall number of events is the same. This is quite normal due to the fact that the amount of EPs to be detected is far greater (x10), which maps to 10 times greater detection load. However, the final processing time value is not 10 times greater than the last value in the previous experiment, which signifies that Esper seems to scale better in this aspect.

With respect to memory consumption, see Fig. 6a, the behaviour is again linear and we just reach a slighly higher final value (around 1.8 GB) than in the first experiment. This again means that Esper is able to handle better event patterns with respect to normal events.

Finally, Fig. 6b depicts that now CPU has reached around a 70% utilisation compared to the final value of 30% in previous experiment. This shows that Esper is stressed and starts to reach its limits with respect to event detection and processing. We need to further investigate what should be the final limit over which Esper will have to be distributed to better handle the incoming load.

In overall, we can see that Esper scales well and could be exploited in a system able to handle the monitoring and evaluation of multiple SBAs. Especially, we need to highlight its nice behaviour with respect to detecting a huge amount of EPs. However, it seems that Esper needs some improvement with respect to the pure event processing time, especially as events seem to increase in a different scale when the number of SBAs grows (in contrast to the number of EPs).

7 Conclusions and Future Work

This paper has proposed a new SLO evaluation framework for SBAs that relies on a rich EP expression language, namely SRL, and on the well-known Esper CEP engine. This system has been designed based on a modular architecture where many of its parts can scale on demand. This system is also loosely coupled with the respective monitoring and adaptation engines that might be employed in a SBA management system. The management of EPs is wrapped into the form of a REST service enabling a respective SBA management system to be decoupled from underlying implementation peculiarities and manage the generation and handling of adaptation rules that contain such EPs.

The proposed framework was evaluated according to its processing time and resource usage. The results show that the framework scales linearly when the number of events or the number of EPs that need to be detected increases. This holds mainly for the processing time and memory consumption. However, CPU utilisation seems to start reaching its limits when both the number of events and EPs become too high which actually represents a point for CEP distribution.

Concerning future work, we plan to further evaluate the SLO evaluation framework and especially investigate its exact distribution points. We also plan to compare Esper with other CEP engines in order to reach an informed decision about which CEP engine is more suitable in our context. In fact, it can be interesting to create a system which can be configured to exploit different CEP engines by incorporating the appropriate abstraction mechanisms.

8 Acknowledgments

This work is supported by CloudSocket project that has been funded within the European Commissions H2020 Program under contract number 644690.

References

1. Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.
2. Alexander Artikis, Marek J. Sergot, and Georgios Paliouras. Run-time composite event recognition. In *DEBS*, pages 69–80. ACM, 2012.
3. Claudio Bettini, Xiaoyang Sean Wang, Sushil Jajodia, and Jia-Ling Lin. Discovering frequent event patterns with multiple granularities in time sequences. *IEEE Trans. Knowl. Data Eng.*, 10(2):222–237, 1998.
4. Gordon Blair, Nelly Bencomo, and Robert B. France. Models@ Run.Time. *Computer*, 42(10):22–27, October 2009.
5. Daniel Seybold, Frank Griesinger, Kyriakos Kritikos, Antonio Gallo, Simone Cacciatore, Andreea Popovici, Joaquin Iranzo, Roman Sosa, Wilfrid Utz, and Damiano Falcioni. Explanatory Notes: Final BPaaS Prototype. CloudSocket Project Deliverable D4.6-D4.8, June 2017.
6. Nicolas Ferry, Franck Chauvel, Alessandro Rossini, Brice Morin, and Arnor Solberg. Managing multi-cloud systems with CloudMF. In *NordiCloud*, pages 38–45. ACM, 2013.
7. Joseph L. Hellerstein, Sheng Ma, and Chang-Shing Perng. Discovering actionable patterns in event data. 41(3):475–493, 2002.
8. Raman Kazhamiakin, Marco Pistore, and Asli Zengin. Cross-layer adaptation and monitoring of service-based applications. volume 6275 of *Lecture Notes in Computer Science*, pages 325–334, 2009.
9. Kyriakos Kritikos, J̈org Domaschka, and Alessandro Rossini. SRL: A Scalability Rule Language for Multi-cloud Environments. In *CloudCom*. IEEE, 2014.
10. Magnus S. Magnusson. Discovering hidden time patterns in behavior: T-patterns and their detection. *Behavior Research Methods, Instruments, & Computers*, 32(1):93–110, Mar 2000.
11. Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. pages 346–357, 2002.
12. Debprakash Patnaik, Naren Ramakrishnan, Srivatsan Laxman, and Badrish Chandramouli. Streaming algorithms for pattern discovery over dynamically changing event sequences. *CoRR*, abs/1205.4477, 2012.
13. Kay Römer. Distributed mining of spatio-temporal event patterns in sensor networks. In *EAWMS Workshop at DCOSS*, pages 103–116, 2006.
14. Alex Tze Hiang Sim, Maria Indrawan, Samar Zutshi, and Bala Srinivasan. Logic-based pattern discovery. *IEEE Trans. Knowl. Data Eng.*, 22(6):798–811, 2010.
15. Di Wang, Elke A. Rundensteiner, and Richard T. Ellison. Active complex event processing over event streams. *PVLDB*, 4(10):634–645, 2011.
16. Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *SIGMOD Conference*, pages 407–418. ACM, 2006.
17. Chrysostomos Zeginis, Kyriakos Kritikos, and Dimitris Plexousakis. Event pattern discovery for cross-layer adaptation of multi-cloud applications. In *ESOCC*, 2014.

Towards Business-to-IT Alignment in the Cloud

Kyriakos Kritikos[1], Emanuele Laurenzi[2], and Knut Hinkelmann[2]

[1] ICS-FORTH, Heraklion, Greece
kritikos@ics.forth.gr
[2] FHNW University of Applied Sciences and Arts Northwestern Switzerland
{emanuele.laurenzi, knut.hinkelmann}@fhnw.ch

Abstract. Cloud computing offers a great opportunity for business process (BP) flexibility, adaptability and reduced costs. This leads to realising the notion of business process as a service (BPaaS), i.e., BPs offered on-demand in the cloud. This paper introduces a novel architecture focusing on the design of BPaaS that includes the integration of existing state-of-the-art components as well as new ones taking the form of a business and a syntactic matchmaker. The end result is an environment enabling to transform domain-specific BPs into executable workflows which can then be made deployable in the cloud so as to become real BPaaSes.

Keywords: BPaaS, service, design, discovery, selection, alignment, mediation

1 Introduction

Due to intense market competition, organisations can survive only if they offer services that are either innovative or exhibit a better quality than their competitors. However, by owning a limited infrastructure and continuously requiring to improve the existing business processes (BPs) leads to reaching certain impassable limits. Moreover, the infrastructure maintenance, operation and management costs can be quite prohibiting, especially for small or medium enterprises.

Fortunately, nowadays, cloud computing can become the medium via which organisations can acquire cheap, commodity resources on-demand while also being able to achieve certain benefits, including: outsourcing infrastructure management with reduced cost, flexible resource management, and elasticity. Such benefits can certainly enable improving BP performance.

However, as cloud computing deals only with the infrastructure level, an organisation now faces the problem of how to align the business with the IT level. This is a widely known and hard to solve problem for which no complete solution exists. Moreover, many organisations do not have the expertise to use and combine the cloud services offered.

The above problems can be solved by combining BP management with cloud computing to realise the BP as a service (BPaaS) paradigm to enable migrating BPs in the cloud. However, such a combination is by far not trivial as: (a) there are multiple levels involved from a BP down to IaaS services incorporating many

entities that need to be properly managed; (b) there are multiple realisation opportunities; (c) it is difficult to map business to technical requirements and match both requirement kinds to respective cloud service capabilities; (d) the use of different services at different abstraction levels requires employing a cross-level BP adaptation approach to handle the various issues that might be involved.

Such generic challenges also map to specific ones in the BP lifecycle. In BP design, the latter challenges are translated to: (a) how to map a BP to a technical workflow with a suitable automation level; (b) how to map business terms and requirements into technical ones to drive the selection of the most suitable services to be then integrated into the workflow; (c) how to deal with the service incompatibility problem effectively to guarantee the correct execution of the designed workflow. Such a problem relates to checking the syntactic compatibility of messages exchanged between two or more selected workflow services.

To realise the vision of BPaaS, the CloudSocket project [3] is delivering a platform that unifies together environments that support different BP lifecycle activities. This paper focuses on our contribution towards enabling the BPaaS Design Environment to deal successfully with the 3 aforementioned issues. This translates to introducing an innovative architecture including suitable components that support smart and semantic service discovery at both the business and technical levels, the optimal service selection, the mapping between business and technical requirements and the capability to mediate between the execution of two or more services to achieve message-level compatibility. In result, the developed environment enables the BPaaS provider to transform the initial business functional and non-functional requirements that match the necessities of potential BPaaS customers into an executable workflow that can then be made deployable into the cloud by exploiting other CloudSocket environments.

The BPaaS Design Environment comprises two new components that constitute another contribution of this paper. The business matchmaker enables to find services that support the user functional and non-functional requirements at the business level via following a novel questionnaire-based approach. Such services are then filtered and selected by state-of-the-art technical matchmaker and service selection components. The latter component relies on a novel syntactic matchmaking component able to infer the message-based compatibility between two or more selected services and produce a respective mapping specification. Such a specification can then be exploited by a service mediation service to support the compatible message transformation between services and guarantee the smooth operation of the BPaaS workflow in which this service is integrated.

This paper is structured as follows. Section 2 shortly analyses some existing research results, exploited in the production of the BPaaS Design Environment. Section 3 analyses the main architecture of this environment by also explaining the main functionality and role of each component involved. Sections 4 and 5 detail the two main novel components involved in the architecture, the business and syntactic matchmakers. Section 6 introduces a use case to showcase the main

[3] `www.cloudsocket.eu`

benefits of the proposed environment as well as to validate it. Finally, the last section concludes the paper and draws directions for further research.

2 Background

2.1 Business-to-IT Alignment

The Business-to-IT alignment issue in the Cloud typically refers to the gap between business requirements and technical solutions. Cloud offerings are described technically making it hard for business people, usually understanding a higher-level business language, to properly assess the best fitting cloud solution. Achieving to identify suitable cloud solutions in this context requires specifying requirements for and capabilities of a service in both a business and IT language. To ensure understanding and transparency of knowledge, it is a common practice to represent it in models [5] [20]. Models abstract away from complex realities and become fit to their purpose, i.e., to precisely model the respective intended domain. In [6] we already adopted a model-driven approach where an extension of the BPMN language allows modelling business process requirements (in a business language) and workflows/cloud services descriptions (in a technical language). The approach includes the translation of the business language into the technical one enabling the comparison between cloud service requirements and technical specifications. In result, the matching cloud services are identified. The translation and the comparison are performed by semantically enriching models with ontologies. Hence, models become also machine-interpretable. In modeling research, this practice is well-known as *Semantic Lifting*. [1] defines it as "the process of associating content items with suitable semantic objects as metadata to turn unstructured content items into semantic knowledge resources". Semantic Lifting shifts the purpose of modelling beyond transparency and communication [7]. The interpretable knowledge base (i.e., ontology) allows for model processing operations, realised in modelling mechanisms and algorithms [6].

In this work, we build on the findings in [6] but adopt a different perspective of Business-to-IT alignment in the Cloud. Namely, we shift from the language translation to the mapping of values between requirements and specifications on both the business and IT levels, separately. Hence, the Business-IT alignment paradigm is applied sequentially by further refining the results from the business layer in the IT layer. Namely, we propose three matchmaking components: (a) the business and (b) technical matchmakers enhanced with formal semantics for the machine-interpretation plus (c) the syntactic matchmaking component. The combination of these three components allows identifying the most suitable cloud services that will eventually form a workflow.

2.2 Technical Service Matchmaking

Technical service matchmaking involves functional and QoS matching. Functional matchmaking has been traditionally focused on I/O-based matching [10,

17] while QoS matching takes the view of QoS as conformance [12] and employs different kinds of techniques (semantic reasoning, constraint solving & mixed) [13] to infer if the solution space of the service is included in the solution space of the request. While most of the proposed work focus on one aspect individually, some approaches have attempted to consider both aspects simultaneously [2, 9]. However, they usually consider only the sequential pattern to combine the matching in both aspects and do not employ semantic techniques, thus not exhibiting the right performance and leading to results of medium or low accuracy.

As such, our previous work [15] has explored different ways the 2 matchmaking types can be jointly performed: (a) sequential combination (with two different variations); (b) parallel combination; (c) subsumes combination (construction of an hierarchy of service advertisements that are connected with the subsumes relation to exploit the fact that if a parent is matched by a request, then also its descendants will be matched). After an experimental evaluation of these combinations, it was shown that the parallel combination of aspect-specific matching approaches leads to the best possible results with respect to performance, as matchmaking accuracy is exactly the same in both approaches.

Our approach exploits two aspect-specific matchmakers, a functional and a non-functional. The functional matchmaker relies on the combination of I/O-based and IR-based matching and has been developed in the Alive project [3]. This matchmaker exploits a smart graph-based structure able to dynamically tolerate changes in domain ontologies (i.e., the ontologies via service I/O is annotated) while supplies almost constant-in-time query operations over the graph (e.g., to discover all ancestors in the ontology for a specific service input / output parameter). I/O based matching is performed in a two-step manner. First, the output of the service and its request are considered for the matching which results in different kinds of match categories that rely on the relations between the semantic concepts of the the service and the request output parameters. Then, input-based matching is performed on the fly to produce the final matchmaking results which can then be ranked according to different criteria.

The unary matchmaker [13] follows a hybrid approach in QoS service matchmaking. First, it considers ontological-based specifications of services to align them based on their QoS terms (their QoS metrics in particular). Then, it performs the respective filtering in a step-wise manner by considering each QoS term individually each time. As unary constraints are assumed to be involved in the service offer and demand, the matchmaker employs smart structures to support parameter-based filtering which results in ultra fast matching time. In fact, by being compared with other QoS matchmakers, this matchmaker was shown to be the fastest in both service matchmaking and registration and the most scalable without affecting in any case the accuracy of the produced matchmaking results.

2.3 Service Selection

Various service selection algorithms have been already proposed. However, they usually consider only one abstraction level by also neglecting semantics, thus

producing results of imperfect accuracy. The accuracy is further reduced by considering that some algorithms employ smart but non-optimal solving techniques, like Genetic Algorithms or heuristics to accelerate the service selection time.

By considering now that the service selection for a BPaaS includes different abstraction levels along with design choices (e.g., either to select an external SaaS or deploy an internal software component in an IaaS to realise the functionality of a BPaaS workflow task), we have developed an constraint-satisfaction-based algorithm [14] which resolves these two issues while also catering for other important features including: (a) the consideration of multiple optimisation objectives by employing the Analytic Hierarchy Process (AHP) [18] and Simple Additive Weighting (SAW) [8] techniques; (b) the addressing of non-linear constraints; (c) the capability to bridge the gap between the two levels (SaaS and IaaS) via the insertion of functions that derive the QoS at the SaaS level based on the respective capabilities selected at the IaaS level; (d) the ability to deal with overconstrained user requirements by employing smart utility functions that allow the slight violation of the user requirements so as to produce at least one solution; (e) the capability to consider all possible execution paths in the BPaaS workflow and not just one (e.g., the critical one); (f) the capability to consider ranges of possible values for the service offerings (suitable when not a single value can be guaranteed for a certain QoS parameter); (g) the capability to consider dependencies between QoS parameters at the same level which will enable a more accurate evaluation of the respective solutions; (h) the capability [11] to accelerate the solving time by fixing parts of the problem to particular partial solutions by relying on the BPaaS execution history.

3 Architecture

The BPaaS Design Environment follows a model-driven and semantics-aware approach to support the business-to-IT alignment in the cloud. Such a approach comprises 3 main transformation steps: (a) BP-to-business-services; (b) business-services-to-technical-services; (c) BP&technical-services-to-executable-workflow. As such, the approach guarantees the technical feasability of the solution produced by employing a two-step service matchmaking process at both the business and technical level as well as a service selection approach that is syntactic-compatibility-aware at the technical level.

To achieve its main goal, the environment exhibits an architecture, depicted in Figure 1, comprising 8 main components that are now analysed in detail. Some of the components correspond directly to some of the aforementioned steps while others play a supporting or an orchestration role. The architecture also spans the well-known 3 levels of user interface (UI), business logic and persistence.

BPaaS Designer (BD). It represents the main point of interaction with the user in the context of BPaaS design. It enables specifying both domain-specific BPs and respective executable workflows. It also guides users in providing suitable input to support the alignment and transformation of BPs into workflows.

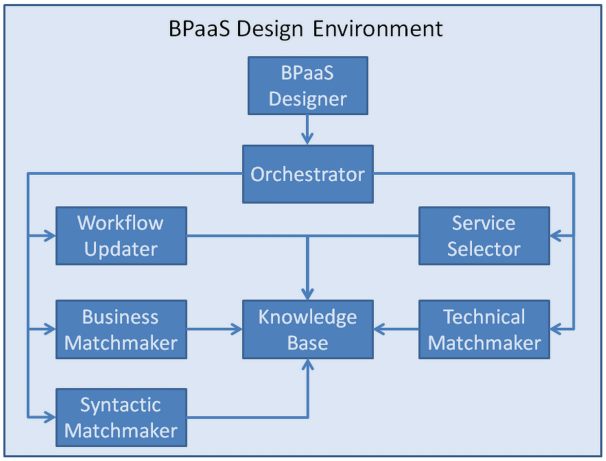


Fig. 1: The architecture of the BPaaS Design Environment

Orchestrator (Orch). It is responsible for orchestrating its underlying components to handle the requests issued by the BPaaS Designer.

Business Matchmaker (BM). It is responsible for matchmaking the cloud services registered in the Knowledge Base based on business requirements derived from a questionnaire-based approach which is explained in the next section.

Technical Matchmaker (TM). A technical, functional and non-functional service matchmaker. It exploits state-of-the-art aspect-specific matchmakers in a parallelised fashion according to the approach in [15].

Service Selector (SS). It [14] attempts to produce a concrete optimal solution for the service-based workflow at hand by considering the users technical non-functional requirements while also attempting to maximise the message compatibility between services. The latter is derived through using the next component.

Syntactic Matchmaker (SM). It is called dynamically by the SS while solving the service selection problem for the BPaaS workflow to find the compatibility between the next and all previously selected services in each execution path where such a service participates. Such a compatibility maps to the message compatibility [16] between the output parameters of the already selected services and the input parameters of the next service. When an incompatible solution is constructed, SS can backtrack and check another solution. To smartly deal with cases where the same call is issued, e.g., due to deep backtracking, SM stores the call results so as to immediately answer it. The mapping of the output parameters to the input ones of the next service is also recorded to enable updating the BPaaS workflow via a mediation service, as performed by the next component.

Workflow Updater (WU). It is responsible for updating the BPaaS workflow by performing the following actions for each workflows execution path: (a) we replay the solution construction in each path to obtain the respective mapping of the current service in the path from the SM; (b) we introduce a mediation service within the workflow, immediately before the current service, which takes as input the current output parameter set and the mapping specification and produces as output the input parameters of the current service.

Knowledge Base (KB). It includes all necessary and sufficient information to support all reasoning tasks executed in the system, including the business and technical service matching and selection for the BPaaS workflow at hand.

4 Business Matchmaking

The *Business Matchmaker* allows specifying requirements in a more user centric approach than the one in [6]. It relies on a context-adaptive questionnaire that guides the user via a set of questions that reflect BP functional and non-functional requirements. Follow-up questions are displayed based on the result of the context-adaptive algorithm that considers: (a) the user preferences in terms of categories (e.g., *Performance* rather than *Data Security*)); (b) the information value of semantic attributes reflecting cloud service specifications at the business level, e.g., 10 minutes of *monthly downtime*. Questions are prioritised such that the relevant ones are displayed first to discover services as quickly as possible.

The idea is that the questionnaire can be applied on the whole BP first. If no service can be found, we can then move down to groups of activities, until the level of single activities.

4.1 The Context-Adaptive Questionnaire

The Context-Adaptive Questionnaire relies on the BPaaS ontology in [5]. Questions focus first on functional requirements and then on non-functional ones. The questionnaire enables to specify functional requirements in two ways:

- by asking the user to insert an action and an object from a predefined taxonomy in the BPaaS ontology. This corresponds to the convention of BPMN to name activities by a verb (i.e. action) and a noun (object) [19] whose combination provides the "what-is-about" knowledge.
- by asking the user to insert the most suitable category from the APQC Process Classification Framework.

Next, the user can choose one of the 5 non-functional (NF) categories: *Data Security*, *Payment*, *Performance*, *Service support*, and *Target Market* .

The NF categories were derived from the Cloud Service Agreement Standardisation Guidelines [4], an outcome of the European 2020 initiative "Digital Agenda for Europe" published to standardize and streamline the terminologies and understanding of cloud services. The NF categories were subsequently discussed and validated within the CloudSocket consortium. In result, a respective set of questions and sub-questions were derived out of them. For instance, the *Performance* category contains the following questions:

- Whats your preferred monthly downtime in minutes?
  Possible answer: 30 minutes
- What would you like to upload?
  Possible answers: Audio MP3, Video MP4, PDF and/or Microsoft Office documents

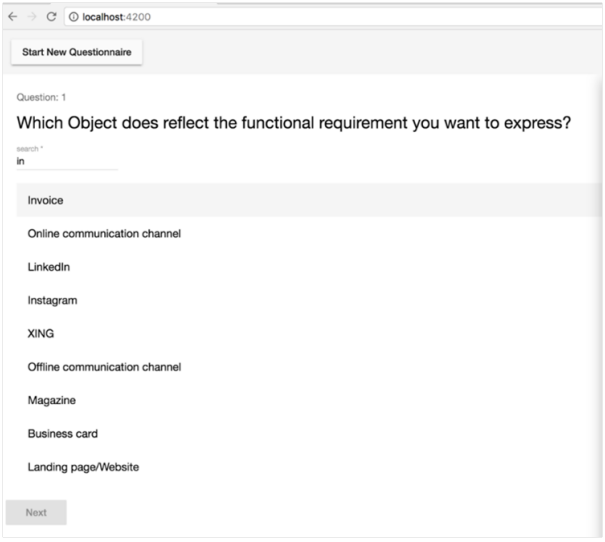


Fig. 2: The object selection for the functional requirements posing

– Should the process be executed on a daily, weekly, monthly or yearly basis?
  Possible answer: On a weekly basis
  - *Sub-Question*: How many times should the process be executed?
    Possible answer: at least 10 times
– What is your favorite response time level?
  Possible answer: High, Medium or Low
– How many simultaneous users should the cloud service support?
  Possible answers: at most 10

For each question, we distinguished among four types of answers as: (1) single-answer selection; (2) multi-answer selection; (3) search-insert; (4) value-insert. Value- and search-insert require input from the user. While the former enables the user to insert values (e.g., the aforementioned downtime in minutes), the latter provides the possibility to crawl predefined values from the ontology and select the suitable one. For instance, answers related to the first 3 functional requirement questions (i.e., Action, Object and APQC category) are of search-insert type. Namely, users can insert keywords for the BP they are looking for, and the ontology returns the concepts matching these keywords. Fig. 2 shows the implementation result of this functionality.

Each time a question is answered, semantic rules are applied to convert implicit knowledge reflecting the business requirements into an explicit one. This prepares the ground to identify matching cloud services by applying a semantic query. For example, assume we have the following:
Specifications from the KB as follows:

– A cloud service with the execution constraint of 20 times per day.

Requirements from the questionnaire as follows:

– Should the process be executed on a daily, weekly, monthly or yearly basis?
  Answer: At least on a weekly basis.

- How many times should the process be executed?
  Answer: At least 10 times

Running a process at least on a weekly basis implies that can also run on a daily basis. The semantic rule, therefore, would infer the answer "On a daily basis" and insert it in the KB. The semantic query then compares the derived fact with the cloud service fact related to the execution constraint. In result, the cloud service specification matches with the requirement.

4.2 Question Prioritisation Algorithm

The NFR questions follow a question prioritisation algorithm. This enables to identify the matching cloud services by asking as few questions as possible. Answers to the questions, along with previous ones, are used to display the follow-up question. The algorithm considers the following:

– Grouping among non-functional attributes. For instance, assuming that a cloud service has the *availability* and *response time* attributes of the *Performance* category. If the user selects to answer one of the two, the follow-up question will be on the remaining attribute in the same category.
– Entropy expressing the variation degree in the values of each non-functional attribute (e.g., *availability*). Entropy of an attribute is "0" when every cloud service stored in the *KB* contains the same attribute value, while "1" in the opposite case. For example, if all cloud services in the *KB* have their *data location* in Switzerland, the entropy of this attribute is 0. As such, the question related to the preferred data location will not be asked as it will not filter out any services from the matching set.

The entropy formula is expressed as follows:

$$Entropy\left(attr_i\right) = -\sum_{j=1}^{J}\left(p_{ij} \cdot log_2\left(p_{ij}\right)\right)$$

where J is the total number of attribute values and p_{ij} is the probability that a certain attribute value val_{ij} of attribute $attr_i$ appears in a specific cloud service. By considering that this probability is independent and uniform across all attribute values, then p_{ij} can be expressed as: $p_{ij} = \frac{[CS]_{csval_{ik}=val_{ij}}}{[CS]}$ where the nominator denotes the number of cloud services that exhibit the respective attribute value ($csvalik$ denotes the value of $attr_i$ for cloud service k) and the denominator the number of all cloud services.

The prioritisation algorithm's signature and main logic is as follows.
Input.

– already stated variables: $attr, CS, val, csval$.
– The set of non-functional categories C =*Data Security*, *Payment*, *Performance*, *Service support*, *Target Market*.

– Set of tuples $< attr_i, Q_l >$ where Q is the set of questions and Q_l is a certain question where $1 \leq l \leq [Q]$. So, each tuple maps 1 attribute to 1 question.

Output. The filtered set of cloud services CS that match with the content of the questionnaire, i.e., questions and answers.
Business Logic.

1. IF the number of categories left is positive ($[C] > 0$), select a category c_n, ELSE exit.
2. IF c_n has a positive number of semantic attributes left, i.e., $[attr|]_{attr_i.cat=c_n} > 0$, THEN calculate the entropy of all the selected category's attributes, ELSE remove the current category c_n from C and go to (1).
3. Select attribute $attr_i$ with highest entropy.
4. Display question Q_l that is mapped with the $attr_i$.
5. Get user answer mapping to a value val_{ij} of attribute $attr_i$
6. Filter services in CS which do not satisfy the condition: $csval_{ik} = val_{ij}$.
7. Remove the semantic attribute $attr_i$ from the category c_n and go to (2).
8. Exit.

5 Syntactic Matchmaking

Business and technical matchmakers cannot guarantee the message compatibility between selected services in a BPaaS workflow. Such a compatibility is thus an obligatory, hard constraint in service selection for producing optimal, message-compatible solutions that can be safely executed. So, a component is needed able to derive such compatibility and supply it as a function to the *Service Selector*.

The main idea is that this component should first find which output messages of previously selected services match to which input messages of the currently selected service (based on the *Service selector*'s solution generation process) for each execution path in the BPaaS workflow. Then, it should check for each message-to-message match if the first message conveys less information than that required by the second message. If this checking succeeds, there is no compatibility between the execution path's considered services. When all message pair matches are compatible, the considered services are message-compatible.

Message Matching. The first message compatibility step can rely on existing semantic service annotations to easily and rapidly discover matching message pairs, as the messages involved in these pairs should correspond to semantically compatible concepts. However, even in the presence of such knowledge, message matching is by far trivial and follows a two-step process: (a) semantic message matching; (b) syntactic message matching. This process is exemplified via the example of a certain service pair which involves service S_1 with 2 output parameters mapped to ontology concepts A & B and service S_2 with 2 input parameters mapped to ontology concepts C & D.

At the semantic level, we follow a bipartite matching approach which checks whether every parameter of the current service has a respective mapping to

one parameter of the previously selected services (or the initial user input) in a certain execution path and attempts to discover a solution with the highest overall match degree. As such, we first define a local match degree between two parameters to be the distance between the parameters' annotation concepts in the ontology subsumption hierarchy, provided that the second parameter's concept subsumes the first parameter's one. If the latter does not hold, the distance is infinite. This guarantees that no information loss occurs as in the opposite case, the more concrete concept in the S_2 input will require the specification of additional pieces of information than those exhibited in the concept in the S_1 output. A mapping solution's overall distance is then the sum of the distances of the matches found. As such, the matching problem can be defined as follows:

$$min \left\{ \begin{array}{l} \frac{1}{[J]} \cdot \left(\sum_{i \in I}^{j \in J} \left(\frac{dist(M_i, N_j)}{maxPSize} \cdot x_{ij} \right) + \sum_{j \in J} \left(1 - \sum_{i \in I} x_{ij} \right) \right) \\ \sum_{j \in J} x_{ij} \leq 1 \\ \sum_{i \in I} x_{ij} \leq 1 \\ i = [1, \ldots, [I]], j = [1, \ldots, [J]] \end{array} \right\}$$

where I and J are the sets of input and output parameters, respectively, x_{ij} is a decision variable whether the output parameter i matches the input one j, $dist\,(M_i, N_j)$ is the distance between annotation concepts M_i and N_j of the two parameters pair while $maxPSize$ represents the maximum subsumption path length in the respective domain ontology used.

Suppose that the following relations hold in the running example: A subsumes B, C & B, C subsume D. In this respect, the best possible matching is $\{A \rightarrow C, B \rightarrow D\}$ with overall distance of 2. The other matching solution $\{A \rightarrow D, B \rightarrow C\}$ is not selected as the local distance between B & C is infinite so the overall distance is also infinite.

The algorithm then proceeds at the syntactic level by considering only those message pairs with a finite local degree of match. For each message pair filtered, we note the information items for the output parameter and the information items of the input parameter and then we see whether the former include the latter. As the matching of information items to the matching ontology concept has been already identified, we perform this checking by replacing the information items with the attributes of the ontology concept. Even if the concepts matched are not identical, as they are related with a subsumption relation, they will certainly have common attributes. So, the problem then is mapped to checking whether the concept attributes of the output parameter are a superset of the attributes of the input parameter.

Message types might also convey information not included in an ontology requiring to perform a different matching kind for them. The logic of this matching is similar to that for the semantic level. In particular, bipartite matching is performed with a sole difference in the way the distance is calculated at the local level. At that level, we need to consider both how similar the field names are and how close are their types. Name similarity can rely on well-known string distance measures (e.g., Levenshtein) while type similarity can rely on the approach

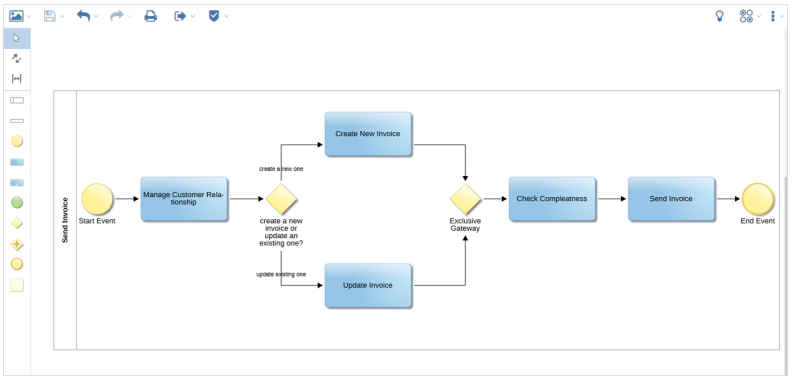


Fig. 3: The Send Invoice business process in BPMN 2.0

in [16] mapping to the compatibility level between types. The overall distance at the local level would equal the weighted sum of the two different distances.

When there is no match for an input parameter part, this means that the compared services are semantically incompatible. Otherwise, if all input parameter parts are matched, then the compared messages are semantically compatible.

Let us now continue the running example to explain syntactic matchmaking. Suppose that A & C have been found equivalent. A maps to message type MT_1 containing 4 information pieces MT_{11}, MT_{12}, MT_{13} and MT_{14}. C maps to message type MT_2 which contains 3 information pieces MT_{21}, MT_{22}, and MT_{23}. Based on matching message types to ontology concepts, we have that MT_{11} and MT_{21} map to $A.A1$ while MT_{12} and MT_{22} map to $A.A2$. As such, the information pieces are transformed into $\{A.A1, A.A2, MT_{13}, MT_{14}\}$ for first message type and $\{A.A1, A.A2, MT_{23}\}$ for the second. For those pieces that do not map to ontology attributes, we need to solve a bipartite matching problem again. Suppose that $dist\,(MT_{13}, MT_{23}) = 0.8$ and $dist\,(MT_{14}, MT_{23}) = 0.2$. Then, obviously the sole mapping to be selected will be $\{MT_{13} \rightarrow MT_{23}\}$. If we replace MT_{23} with MT_{13}, we then need to check whether $\{A.A1, A.A2, MT_{13}\}$ is subset of $\{A.A1, A.A2, MT_{13}, MT_{14}\}$ which holds.

6 Validation

Our approach was validated based on a use case developed within the CloudSocket consortium by the industrial partners. We focused on one of the most common BPs among SMEs - the Send Invoice one. This BP is modelled in BPMN, see Fig. 3, via our integrated BPaaS Design environment. It starts with the "Manage Customer Relationship" activity; next an exclusive gateway splits the BP flow between either creating a new invoice or updating an existing one. Then, the invoice completeness is checked, and finally the invoice is sent.

The next services were inserted in the *KB* as instances of *CloudService* class:

– YMENS, Zoho and Sugar CRM were inserted as CRM systems which were annotated with the action *Manage*, the object *Customer* and the APQC category *3.5.2.4 Manage Customer Relationship*

– Mathema Document Generator, Open Source Billing, Simple Invoice and InvoiceNinja as invoicing systems annotated with the action *Generate*, the object *Invoice* and the APQC category *9.2.2.2 Generate Customer Billing Data*
– Gmail, Ninja_email and Mailjet were inserted as e-mail systems which were annotated with the action *Manage*, the object *Invoice* and the APQC category *9.2.2.3 Transmitting Billing Data to Customers*

Table 1 shows a part of the non-functional profiles of the considered services.

Service	Montly Downtime	Response Time Level	File Type	No of Simul. Users	Execution Constraint
YMENS CRM	4 min	High	Office doc, PDF, audio, Video	500	none
Zoho CRM	4 min	High	Office doc, PDF, audio, Video	500	none
Sugar CRM	10 min	Medium	Office doc, PDF, audio, Video	200	500 (monthly basis)
InvoiceNinja	4 min	High	Office doc, PDF, audio, Video	600	none
Ninja Email	4 min	High	Office doc, PDF, audio, Video	400	none
Simple Invoices	10 min	Medium	Office doc, PDF, audio, Video	300	none
Mailjet	4 min	High	Office doc, PDF, audio, Video	100	1K (monthly basis)
Open Source Billing	4 min	Medium	Office doc, PDF, audio, Video	200	none
Gmail	4 min	High	Office doc, PDF, audio, Video	100	None

Table 1: Functional requirements for each group and single activity

6.1 Business Matchmaking

The *Business Matchmaker* was used to identify the most suitable cloud services. In particular, as a first step, we applied the questionnaire on the whole BP. Fig. 4 shows the notebook from which the questionnaire was started.

We specified functional requirements in the first 3 questions - object *Send*, action *Invoice* and APQC category *9.2.2 Invoice Customer* - and none of the cloud services matched; Fig. 5 shows the empty list at the right-hand side.

Next, the questionnaire was applied on two single activities (i.e., Manage Customer Relationship and Send Invoice) as well as on a group of activities (i.e., Create Invoice, Update Invoice and Check Invoice Completeness).

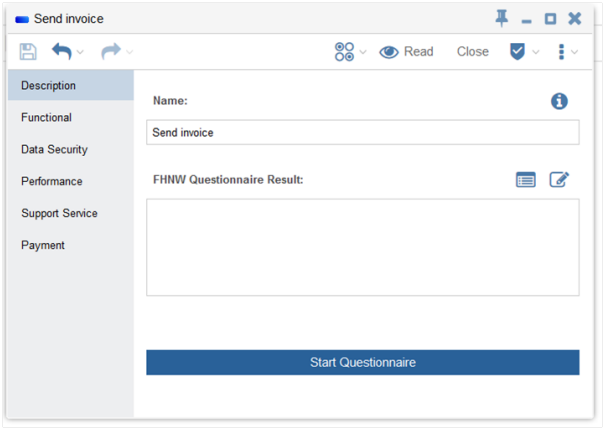


Fig. 4: The starting notebook for the whole process

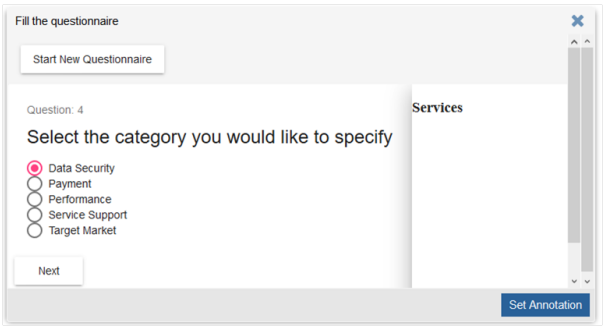


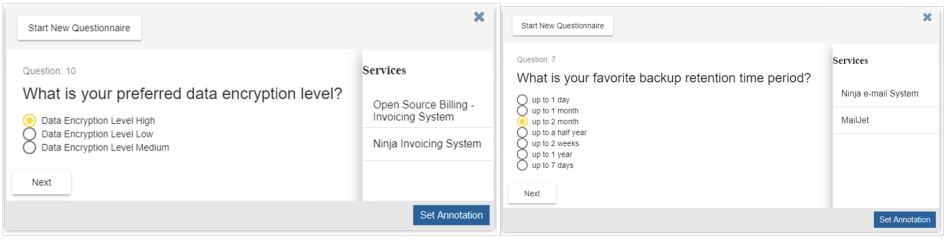
Fig. 5: Functional requirements failure and NFR category selection

Table 2 shows the functional requirements for each activity/group. In the first case, after specifying action, object and APQC category, the questionnaire showed the three matching cloud services: YMENS, Zoho and SugarCRM. In the 4th question, we chose the *Performance* category, and the question prioritisation algorithm kicked in. The question regarding the number of simultaneous users was asked (attribute with highest entropy) and a value of 500 was entered. This filtered out SugarCRM as it has the capability of max 200 simultaneous users.

BPMN Activity	Action	Object	APQC Category
Manage Customer Relationship *(Single activity)*	Manage	Customer Relationship	3.5.2.4 Manage Customer Relationship
Send Invoice *(Single activity)*	Send	Invoice	9.2.2.3 Transmit Billing Data to Customers
Create New Invoice, Update Invoice, Check Invoice Completeness *(Group of activities)*	Generate	Invoice	9.2.2.2 Generate Customer Billing Data

Table 2: Functional requirements for each group and single activity

Similarly, we applied the questionnaire on the designated group of activities. The matching services were InvoiceNinja and Open Source Billing, see Fig. 6a.



(a) The selected invoice management services (b) The selected email services

Fig. 6: The selected services for last two activity groups

Finally, we applied the questionnaire on the last activity of the BP: Send Invoice. The matching cloud services were Ninja e-mail and Mailjet (see Fig. 6b).

6.2 Technical Matchmaking & Selection

As the final result maps to two services per each activity (group), we now proceed with the technical matchmaking and selection. Suppose that the user provides the following global requirements for the whole process: $cost < 100$ euros per month, $cycletime < 1$ minute and $VPM < 16$ (#vulnerabilities per month). Further, suppose that the user imposes that for the *Manage Customer Relationship* activity, the following constraints should hold: $responsetime < 30$ seconds and $VPM < 10$. Finally, Table 3 depicts the non-functional profiles of the remaining services.

Service	Cost	Response Time	VPM
ZOHO CRM	30 euros	35 seconds	10
YMENS CRM	35 euros	20 seconds	05
Mailjet	25 euros	10 seconds	02
Ninja_Email	10 euros	15 seconds	03
Open Source Billing	35 euros	25 seconds	08
Invoice Ninja	45 euros	10 seconds	05

Table 3: The technical non-functional offerings of the 6 services

Technical non-functional matchmaking would then filter Zoho CRM as it does not conform to the local constraints posed for the CRM-based activity. This leads to performing the selection over 4 solutions as we have one candidate for the first (group) of activities and 2 candidates for the rest two activity groups. However, while running service selection, it is detected that the Ninja_Email and Open Source Billing are incompatible, which leaves us with 3 solutions. Moreover, the

solution mapping to selecting YMENS, Open Source Billing and Ninja_Email has an MTBF equal to 1 hour which violates the respective global constraint. So, in the end, we need to select only between two solutions.

The combinations to be checked for selecting the best one are depicted in Table 4. As the broker requires to optimise all non-functional terms (cost, cycle time and MTBF), it gives equal preference over them. By also considering that the activities are sequentially executed in the BPaaS workflow, the final result would map to selecting the services YMENS, InvoiceNinja, and Ninja_Email. While there is perfect syntactic compatibility between InvoiceNinja and Ninja_Email as they are offered by the same organisation, in the case of YMENS CRM and InvoiceNinja the respective message types are compatible but still need to be aligned (e.g., attributes *accountid* and *id_number* which map to the same attribute *id* of concept *Client*). As such, the MS service was included between these 2 services resulting in a workflow with 4 services sequentially executed (YMENS CRM → MS → InvoiceNinja → Ninja_Email).

Solution	Cost	Cycle Time	VPM	Utility
YMENS + InvoiceNinja + Ninja_Email	90 euros	45 seconds	13	0.144
YMENS + Open Source Billing + Mailjet	95 euros	50 seconds	15	0.099

Table 4: The final ordered solutions produced

7 Conclusions and Future Work

This paper has introduced a novel architecture for the design of business process as a service products which is able to effectively deal with the business-to-IT alignment problem in order to map an initial domain-specific BP into an executable BPaaS workflow. Such an architecture has been carefully designed and implemented to include suitable components which focus on different parts of the business-to-IT alignment problem, including business and technical matchmakers, a service selection as well as an automatic workflow update component to enable the effective addressing of the message compatibility problem in service-based workflow execution.

Our future work will focus on more advanced research challenges which include: (a) the automatic production of a more complete and more close to production workflow; (b) the automatic population of the KB; (c) the coverage of additional cases in business-to-technical-requirement alignment.

Acknowledgments This research has received funding from the European Community's Framework Programme for Research and Innovation HORIZON 2020 (ICT-07-2014) under grant agreement number 644690 (CloudSocket).

References

1. Azzini, A., Braghin, C., Damiani, E., Zavatarelli, F.: Using Semantic Lifting for improving Process Mining: a Data Loss Prevention System case study

2. Benaboud, R., Maamri, R., Sahnoun, Z.: Agents and owl-s based semantic web service discovery with user preference support. International Journal of Web & Semantic Technology 4(2), 57–75 (2013)
3. Cliffe, O., Andreou, D.: Service Matchmaking Framework. Public Deliverable D5.2a, Alive EU Project Consortium (10 September 2009), available at: `http://www.ist-alive.eu/index.php?option=com_docman&task=doc_download&gid=28&Itemid=49`
4. Cloud Select Industry Group (C-SIG): Cloud Service Level Agreement Standardization Guidelines. Technical Report, EC (2014), `http://ec.europa.eu/information_society/newsroom/cf/dae/document.cfm?action=display&doc_id=6138`
5. Hinkelmann, K., Gerber, A., Karagiannis, D., Thoenssen, B., van der Merwe, A., Woitsch, R.: A new paradigm for the continuous alignment of business and IT: Combining enterprise architecture modelling and enterprise ontology. Computers in Industry 79, 77–86 (2016)
6. Hinkelmann, K., Kurjakovic, S., Lammel, B., Laurenzi, E., Woitsch, R.: A Semantically-Enhanced Modelling Environment for Business Process as a Service. In: ES. Melbourne, Australia (2016)
7. Hrgovcic, V., Karagiannis, D., Woitsch, R.: Conceptual Modeling of the Organisational Aspects for Distributed Applications: The Semantic Lifting Approach. In: Compsac Workshops. pp. 145–150. IEEE (jul 2013)
8. Hwang, C., Yoon, K.: Multiple Criteria Decision Making. Lecture Notes in Economics and Mathematical Systems (1981)
9. Jiang, S., Aagesen, F.A.: An Approach to Integrated Semantic Service Discovery. In: First International IFIP TC6 Conference. pp. 159–171. Springer (2006)
10. Klusch, M.: Semantic Web Service Coordination. In: CASCOM: Intelligent Service Coordination in the Semantic Web. pp. 59–104. Springer (2008)
11. Kritikos, K., Magoutis, K., Plexousakis, D.: Towards knowledge-based assisted iaas selection. In: CloudCom. pp. 431–439. IEEE Computer Society, Luxembourg (2016)
12. Kritikos, K., Plexousakis, D.: Requirements for QoS-Based Web Service Description and Discovery. IEEE Transactions on Services Computing 2(4), 320–337 (2009)
13. Kritikos, K., Plexousakis, D.: Novel Optimal and Scalable Nonfunctional Service Matchmaking Techniques. IEEE T. Services Computing 7(4), 614–627 (2014)
14. Kritikos, K., Plexousakis, D.: Multi-cloud Application Design through Cloud Service Composition. In: CLOUD. pp. 686–693. IEEE, New, NY, USA (2015)
15. Kritikos, K., Plexousakis, D.: Towards Combined Functional and Non-functional Semantic Service Discovery. In: ESOCC. pp. 102–117. Springer, Vienna, Austria (2016)
16. Kritikos, K., Plexousakis, D., Paternò, F.: Task model-driven realization of interactive application functionality through services. TiiS 3(4), 25 (2014)
17. Plebani, P., Pernici, B.: URBE: Web Service Retrieval Based on Similarity Evaluation. IEEE Transactions on Knowledge and Data Engineering 21(11), 1629–1642 (2009)
18. Saati, T.: The Analytic Hierarchy Process. McGraw-Hill (1980)
19. Silver, B.: BPMN Method and Style, Second Edition. Cody-Cassidy Press, Aptos, second edi edn. (2011)
20. Uschold, M., King, M., Morale, S., Zorgios, Y.: The Enterprise Ontology. The Knowledge Engineering Review 13(01), 31–89 (1998)