DYNAMIC TENANT PROVISIONING AND SERVICE ORCHESTRATION IN HYBRID CLOUD

Dmitry Vasilenko and Mahesh Kurapati

IBM Watson and Cloud Platform, IBM, Chicago, USA

ABSTRACT

The advent of container orchestration and cloud computing, as well as associated security and compliance complexities, make it challenging for the enterprises to develop robust, secure, manageable and extendable architectures which would be applicable to the public and private cloud. The main challenges stem from the fact that on-premises, private cloud and third-party, public cloud services often have seemingly different and sometimes conflicting requirements to tenant provisioning, service deployment, security and compliance and that can lead to rather different architectures which still have a lot of commonalities but evolve independently. Understanding and bridging the functionality gaps between such architectures is highly desirable in terms of common approaches, API/SPI as well as maintainability and extendibility. The authors discuss and propose common architectural approaches to the dynamic tenant provisioning and service orchestration in public, private and hybrid clouds focusing on deployment, security, compliance, scalability and extendibility of stateful Kubernetes runtimes.

KEYWORDS

Kubernetes, Tenant Provisioning, Service Orchestration, Hybrid Cloud.

1. INTRODUCTION

Cloud computing has become one of the key components of enterprise IT systems. The benefits of cloud computing include reduced IT costs, scalability, shorter time to market, improved business continuity and collaboration efficiency. In cloud computing, the Kubernetes platform has become de facto standard for automating deployment, scaling, and management of containerized applications. The Kubernetes Control and Data Planes provide sound architectural foundation for modern information systems. A number of cloud providers such as Amazon, Red Hat, Google, Microsoft and IBM offer Kubernetes-based solutions for public and private cloud. Despite the wide acceptance and aforementioned benefits there are still a few functionality, security and compliance gaps which should be carefully considered and addressed before any service offering utilizing the containerized frameworks will be ready for the prime time. This is especially true for so-called runtimes or stateful applications which require per-tenant persistent storage or non-trivial life cycle management. The analytical frameworks and applications such as Jupyter Notebooks, RStudio/Shiny, OpenCPU and other allow the cloud users to deploy and run code of arbitrary complexity. Such applications should be sandboxed to reduce the risk of security breaches by restricting the running environment and allowing for the private view of the globally shared kernel resources, such as the network stack, process and mount tables. Strict adherence to the ISO27002 [1], GDPR [2], HIPAA [3] and other standards and legislations is often required or mandatory. In this paper the authors discuss and propose approaches to the consolidated Runtimes framework which extends the existing Kubernetes architecture and enables for secure and complaint stateful applications deployed and managed in public, private and hybrid cloud environments.

DOI: 10.5121/ijccsa.2019.9301

This paper is organized as follows. The section 2 provides an overview of the related works. The section 3 describes designs approaches, API, SPI and core components of the proposed architecture. The section 4 offers an overview of the persistent storage management for the cloud. The last section provides conclusions and describes our next steps.

2. RELATED WORKS

Dynamic tenant and service provisioning in the cloud require an integrated solution across the technology stack, including software, platform, and infrastructure, and combining functional, non-functional and resource allocation requirements [4]. It was shown [5], that much of the service development is still done in an ad hoc manner, without standardized API/SPI, thus resulting in poor reuse of software assets. The resource provisioning problem is considered in [6] as a Mixed Integer Linear Program (MILP). The authors assume that the underlying infrastructure is based on a set of end-to-end connections with guaranteed sustainable bandwidth such as Carrier-Grade Ethernet (CGE) circuits and investigate the impact of services on resource allocation carried out by a Cloud Service Provider (CSP).

As discussed in [7], the service provisioning and deployment often refers to service selection and service composition to satisfy functional and QoS requirements. In contrast to this, [8] focuses on the pattern-based deployment, configuration and management of complex services including their supporting middleware. Work that addresses this depth of deployment and configuration often resorts to a simplified model such as common middleware already deployed [9] or knowledge of the specific target environment so that provisioning steps are known in advance [10]. In [11] paper the authors identified challenges in fulfilling the promise of simplification of distributed software service composition and deployment and proposed an approach and architecture for composition and deployment of virtual software services in cloud environments.

A generic cloud computing stack that classifies technologies and services into different layers was proposed in [12]. The authors showed how the cloud computing stack facilitates communication between different technologies and services. As indicated by a number of surveys on cloud adoption, security, privacy and compliance remain the number one concerns for cloud providers. Also, compute isolation between tenants is important when multiple tenants are served by the same application instance. These concerns become even more stringent for GDPR and HIPAA complaint applications. Cloud applications increasingly rely on complex middleware solutions to support the above requirements [13, 14, 15]. These solutions can involve complex policy engines [14, 15].

Embracing *Privacy by Design* principles [17] will ensure that tenant data in-transit and at-rest is protected and at the same time provide appropriate access and integrity. Addressing the identified tenant provisioning, service orchestration, security and compliance challenges will ensure that the proposed API/SPI for the participating service components are sound and devise better architectural approaches that observe these design principles. Through this, the authors of this paper propose an efficient way of architecting the service components to support cloud requirements for stateful Kubernetes runtimes.

In [18] authors indicate that there is a clear lack of documented best practices on how to orchestrate cloud environments, either public, private or hybrid. The paper targets DevOps practitioners and defines solutions for cloud orchestration, describing them as three core patterns of software containerization, local reverse proxy and orchestration by resource offering. The approach enables effective resource sharing with minimal virtualization overhead and allows clients to access any service in a cluster abstracting its placement. In [19] the authors present an orchestrator for the Software-Defined Data Center, and the strategies used to allocate and

provision resources, basing decisions on the current state of the system. The comprehensive survey of cloud resource orchestration techniques is provided in [20].

3. RUNTIMES ARCHITECTURE

The proposed architecture defines two Kubernetes clusters depicted in Figure 1. The Runtimes UI microservice cluster incorporates stateless services to handle UI rendering and user interactions for the runtimes. The Runtimes cluster hosts instances of the stateful runtimes allocated for the given tenant.

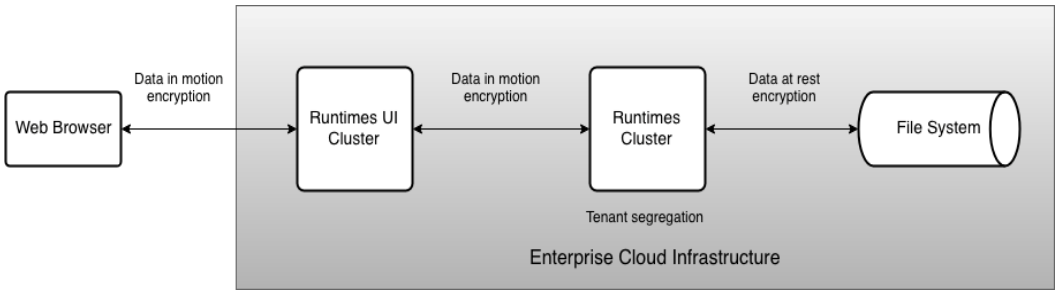

Figure 1. Runtimes and UI microservices clusters

The core services for the Runtimes cluster are defined in the *runtimes-admin* namespace as shown in Figure 2.

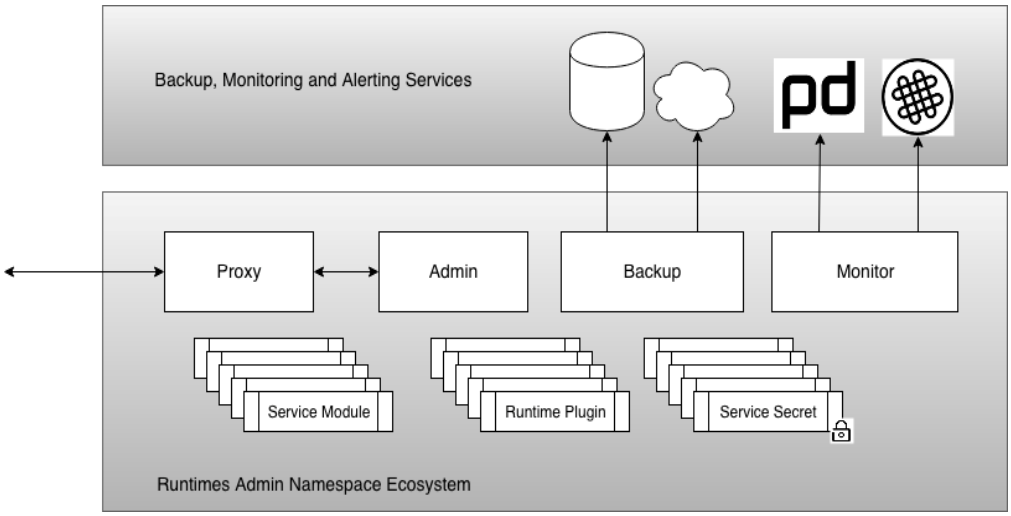

Figure 2. Runtimes admin namespace ecosystem

The *runtimes-admin* namespace hosts the generic Proxy, Admin, Backup and Monitoring components as well as runtime specific plugins, modules and secrets. The following sections of this paper provide further architectural details for each component.

The Figure 3 depicts the simplified network topology diagram for the Runtimes cluster. As shown below, the tenant namespaces are segregated to ensure compute isolation. In multizone clusters at least 2 instances of the Proxy and Admin components should be deployed per availability zone.

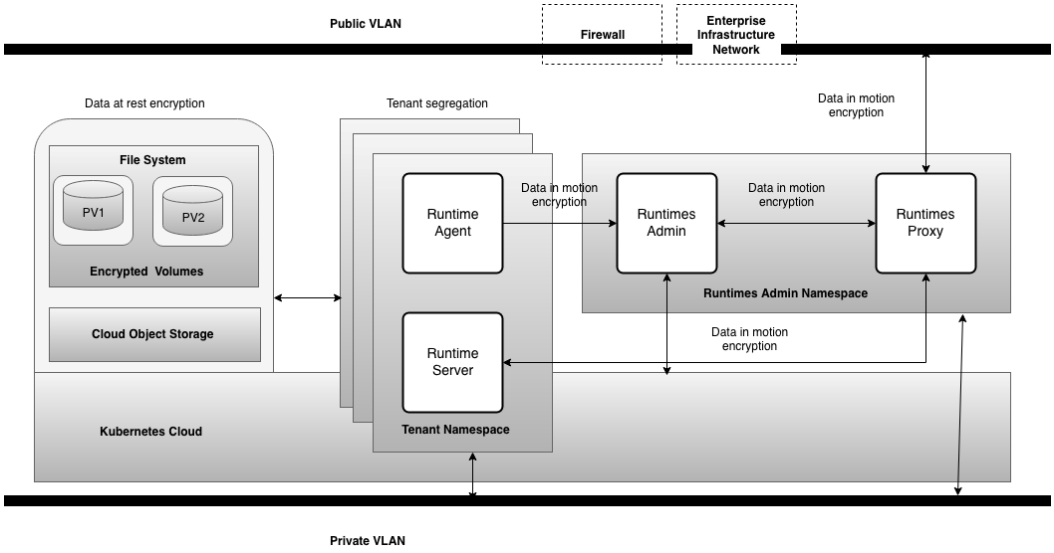

Figure 3. Network topology diagram for Kubernetes Runtimes clusters

3.1. RUNTIMES REVERSE PROXY

As shown in Figure 2, the Proxy component expose a single HTTP endpoint to the Runtimes cluster. The Proxy handles requests to the Admin as well as to the tenant containers. To inject service specific behaviour to the request processing workflow the Proxy supports Service Module SPI shown in Table 1.

Table 1. Runtimes service module SPI

Function	Parameters	Description
init	1. Initialization parameters	Initializes the service module. Called by the Proxy during startup and on service module secret changes.
http	1. HTTP request 2. HTTP response 3. Proxy options	Optional. Called by the Proxy to allow the service module to participate in the HTTP processing.
wss	1. HTTP request 2. Network socket between the server and client 3. The first packet of the upgraded stream (may be empty) 4. Proxy options	Optional. Called by the Proxy to allow the service module to participate in the Web Socket processing.

The service modules for the Proxy are optional and can be used for service-specific authorization and authentication as well as for handling web socket requests. The parts of the HTTP routes for the Proxy should be encrypted and obfuscated for security reasons.

3.2. RUNTIMES ADMIN

The Admin component handles the dynamic tenant provisioning and service lifecycle management. The Admin allocates a Kubernetes namespace for a tenant as needed and applies namespace isolation policies to allow intra-namespace and disable inter-namespace communication for the tenants.

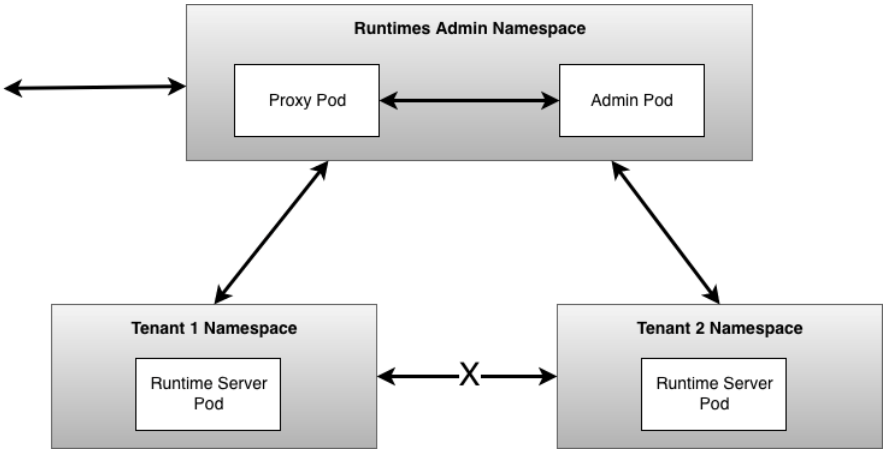

Figure 4. Multi-tenant namespace compute isolation

The requirements for the tenant namespace isolation are mandatory for the public clouds but can be relaxed for the private clouds. In some cases, for the private cloud deployments the Admin and tenants can use the same namespace. If the tenant namespaces are segregated, the Kubernetes DNS records for the services will have the form of *service-instance.tenant-namespace.svc.cluster.local*. In the case of combined Runtime namespace for the private cloud the records will have the form of *tenant-service-instance.runtimes-namespace.svc.cluster.local*. The Admin provides container lifecycle management REST API summarized in Table 2. The endpoints are only accessible to the Proxy and Agents running in the tenant pods. While the Proxy can invoke any REST API methods the Agent calls are limited to the session invalidation and service removal for the same tenant. The Proxy can also support *authority delegation* when the particular REST API call is performed on behalf of another user, typically an administrator. The common use cases include stopping a runtime for the user by the administrator from the web interface.

Table 2. Runtimes container lifecycle management REST API

Method	Endpoint	Description
POST	/services/<user>/<service>/<instance>	Starts the runtime pod for the given user, service and specified service instance. The JSON payload of the POST request should include parameters used to generate all-in-one deployment YAML for the service.
DELETE	/services/<user>/<service>/<instance>/stop	Invalidates the runtime session and removes Kubernetes pod deployment and service. This REST API call is typically performed by the agent running in the runtime pod.
DELETE	/services/<user>/<service>/<instance>	Deletes the service instance. This includes the Kubernetes pod deployment, service, persistent volume claims and secrets.
GET	/services/<user>/<service>/<instance>	Returns the status of the runtime: HTTP 200 if the service is up and running and HTTP 503 otherwise.

The Admin component works in concert with the service plugins. The service plugin design for the Runtimes was inspired by the Kubernetes flex volume drivers [16] and defines a set of

callouts to be implemented by the plugins. The plugins are implemented and deployed as Kubernetes daemon sets. The summary of the plugin callout SPI is shown in Table 3.

Table 3. Runtimes plugin callout SPI

Callout	Parameters	Description
init	None	Initializes the service plugin. Called by the Admin during plugin registration and initialization. On success, the function returns a capabilities map showing capabilities supported by the plugin.
deploy-service	1. User identifier 2. Service name 3. Service instance 4. Runtime registration information 5. Runtime configuration information	Called by the Admin during the service startup sequence. The function returns all-in-one deployment YAML for the service including deployment, service, persistent volume claims and secrets.
check-service	1. User identifier 2. Service name 3. Service instance 4. Runtime configuration information	Optional. Called by the Admin during the service startup sequence to identify discrepancies between existing and proposed configuration. The function returns failure status if the configurations are incompatible.
deregister-service	1. Runtime registration information	Optional. Called by the Admin during session invalidation or service deletion to allow for service specific deregistration actions.
image-pull	1. Image pull specification	Optional. Called from the service plugin daemon set init containers during deployment to prepull container images to the worker nodes.
image-cleanup	1. Image pull specification	Optional. Called from the service plugin daemon set process to check and remove unused images from the worker node image cache. It is important to note that the Kubernetes manages lifecycle of all images through *imageManager*, with the cooperation of *cadvisor* which should be sufficient for most applications. Authors found, however, that proactive image cleanup can improve overall stability of the cluster.

3.3. RUNTIMES BACKUP SERVER

The Backup Server component handles regular user metadata backups as well as backups of the persistent volumes allocated for tenants. The frequency of the backups and other characteristics are defined by the service level agreement. As the Kubernetes persistent volumes claims are namespace-bound the Backup Server spawns an Agent in the tenant namespace to perform the data backup. The backups can be stored in the cloud object storage such as Amazon S3. The Backup Server also performs tenant namespace backups if the services deployed into the namespaces were not accessed during defined period of time. The tenant namespace as well as associated services and persistent volumes can be recreated when the user logs back in again.

3.4. RUNTIMES MONITOR SERVER

The Monitor Service handles alerts, notifications and remediation actions for the Runtimes cluster. While existing application performance management (APM) frameworks such as Prometheus and New Relic can be used to handle variety of alert conditions authors found that in

some cases the custom, dedicated monitoring service is still beneficial. The Runtimes Monitor Service works in concert with the existing APM frameworks and handles pod, persistent volume claim (PVC) and node conditions listed in Table 4.

Table 4. Runtimes container lifecycle management REST API

Object	Condition	Parameters	Description
Pod	long-running	cut-off-minutes	The alert is triggered if the pod was running longer than the *cut-off-minutes*.
	long-pending	cut-off-minutes	The alert is triggered if the pod was in the pending state longer than the *cut-off-minutes.*
	long-terminating	cut-off-minutes	The alert is triggered if the pod was in the termination state longer than the *cut-off-minutes*.
	last-restart	cut-off-minutes	The pod has been restarted within the last the *cut-off-minutes*.
	endpoint-probe	endpoint-url timeout-seconds retries status-code	The alert is triggered when the HTTP endpoint fails to respond to the GET request. The value for the *endpoint-url* can be a template using the pod specific parameters in run-time.
PVC	long-pending	cut-off-minutes	The alert is triggered if the persistent volume claim was in the pending state longer than the *cut-off-minutes*.
Node	not-ready	cut-off-minutes	The alert is triggered if the node did not become ready within *cut-off-minutes*.

4. PERSISTENT STORAGE MANAGEMENT

Kubernetes offers a wide breadth of persistent volume options for storage integration. One of the important requirements for the public clouds is to have transparent, reliable and high performant storage support across availability zones in the multizone cluster. Kubernetes virtualization infrastructure also lends itself particularly well to bare metal server deployments when setting up persistent storage. A number of software defined storage solutions such as Amazon Elastic File System, Google Regional Persistent Discs, Portworx and other are readily available. For the private and hybrid clouds, however, there are still situations when the persistent storage, such as NFS, for the given service should be allocated in the particular availability zone due to the volume affinity. As a result, the services pods will have node affinity to that particular availability zone even if deployed into the multizone cluster. Another important aspect is the observed latency of the persistent volume provisioning for the given tenant. While the storage provisioning process is typically automated, the dynamic tenant provisioning can still be visibly delayed if the volumes are not pre-allocated based on anticipated demand. To achieve low-latency for the NFS persistent volume provisioning the authors designed and implemented Kubernetes flex volume provisioner and driver for sparse files. The sparse files approach allows for per-tenant encryption at rest using Linux Unified Key Setup (LUKS). The simplified diagram for dynamic storage provisioning is shown in Figure 4.

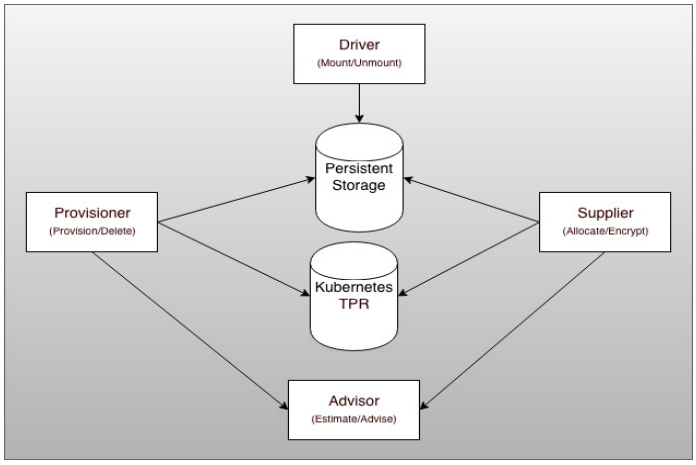

Figure 4. Dynamic persistent storage provisioning

There are four main actors that support NFS flex volume sparse file functionality: Advisor, Supplier, Provisioner and Driver. The Advisor forecasts the sparse file demand using exponentially weighted moving average (EWMA) technique and provides the estimation for the Supplier. The Supplier uses the estimation for the anticipated sparse file demand from the Advisor and allocates and encrypts sparse files using large provisioned NFS mount. If the current mount is exhausted the Supplier will pre-allocate new NFS mounts as needed to meet demand. The Provisioner implements the Kubernetes Provisioner interface and is responsible for creating persistent volumes (PV) for the given claim. The Provisioner looks up the available, pre-allocated and encrypted sparse file and creates PV accordingly. The PV also includes the flex volume specification used by the Driver. The Driver implements Kubernetes flex volume mount/unmount call-outs for the encrypted sparse files. Additionally, the Driver mounts the NFS mount point if it was not yet mounted on the worker node. The sparse mount and file metadata are stored in Kubernetes Third Party Resource tables. The described technique was tested internally and can be beneficial for private as well as for public clouds. For the public cloud scenarios, the approach can be used to support large number of trial, free or limited plan users with relatively small persistent volumes. There could be hundreds of thousands of accounts for such users in public clouds and direct NFS storage provisioning for these accounts can be rather time consuming and inefficient.

5. CONCLUSIONS

In this paper the authors discussed architectural approaches to the dynamic tenant provisioning and service orchestration in public and private clouds with the focus on bridging the gaps related to the deployment, security, compliance, scalability and extendibility of stateful Kubernetes runtimes applications. We introduce a notion of clear separation between stateless UI microservices and stateful Runtimes clusters as well as defined REST API for the generic Proxy and Admin services. We also discussed the SPI for the Runtimes service modules and plugins to enable efficient collaboration between core and application-specific services. As the stateful runtimes require high-performant, reliable and secure persistent storage, just-in-time provisioning and per-tenant encryption at rest as well as transparent support for cross-AZ deployments for multi-zone clusters become increasingly important. We showed that the proposed architecture enforce security, compliance, usability, availability and the overall management of Kubernetes runtimes for internal, hybrid or public clouds.

The authors plan to continue this research focusing on architecture of secure, distributed persistent storage to support multi-region deployments in hybrid cloud.

ACKNOWLEDGEMENTS

The authors would like to thank Thanh Pham, Thuan D Ngo and Wei Huang from the IBM Silicon Valley Lab (SVL) and Mehboob Alam from the IBM Rochester Lab for their support, encouragement and invaluable insights.

REFERENCES

[1] ISO/IEC 27002:2013 Information technology. Security techniques. Code of practice for information security controls. Retrieved May 17, 2019, from https://www.iso.org/standard/54533.html.

[2] The EU General Data Protection Regulation (GDPR). Retrieved May 17, 2019, from https://eugdpr.org/.

[3] The Health Insurance Portability and Accountability Act of 1996. Pub. L. 104-191. Stat. 1936.

[4] Ramachandran, L., Narendra, N.C. & Ponnalagu, K., Dynamic provisioning in multi-tenant service clouds, SOCA (2012) Vol. 6, pp 283–302.

[5] Agarwal V, Dasgupta K, Karnik NM, Kumar A, Kundu A, Mittal S, Srivastava B (2005) A service creation environment based on end to end composition of web services. In: WWW, pp 128–137.

[6] Aoun R, Doumith EA, Gagnaire M (2010) Resource provisioning for enriched services in cloud environment. In: CloudCom, pp 296–303.

[7] Su, X., Rao, J.: A survey of automated web service composition methods. In: SWSWPC (2004).

[8] Arnold W, Eilam T, Kalantar MH, Alexander V (2007) Konstantinou, and Alexander Totok. Pattern based soa deployment. In: ICSOC, pp 1–12.

[9] Kichkaylo, T., Karamcheti, V.: Optimal resource-aware deployment planning for component-based distributed applications. In: HPDC, Washington, DC, USA, pp. 150–159. IEEE Computer Society Press, Los Alamitos (2004).

[10] Ludwig, H., Gimpel, H., Dan, A., Kearney, B.: Template based automated service provisioning supporting the agreement driven service life-cycle. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 283–295. Springer, Heidelberg (2005).

[11] Konstantinou AV, Eilam T, Kalantar M, Totok AA, Arnold W, Snible E (2009) An architecture for virtual solution composition and deployment in infrastructure clouds. In: VTDC ’09: Proceedings of the 3rd international workshop on virtualization technologies in distributed computing. ACM, New York, NY, USA, pp 9–18.

[12] Lenk A, Klems M, Jens N, Tai S, Sandholm T (2009) What’s inside the cloud? an architectural map of the cloud landscape. In: ICSE cloud 09: First international workshop on software engineering challenges for cloud computing, pp 23–31.

[13] Walraven S., Van Landuyt D., Rafique A., Lagaisse B., and Joosen W. Paashopper: Policy-driven middleware for multi-paas environments. Journal of Internet Services and Applications, 6(1), January 2015.

[14] Walraven S, De Borger W., Vanbrabant B., et. al. Adaptive performance isolation middleware for multi-tenant saas. In 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC), pp 112–121, December 2015.

[15] Rafique A., Van Landuyt D., et. al. Policy-driven data management middleware for multi-cloud storage in multi-tenant SaaS. In 2nd IEEE/ACM International Symposium on Big Data Computing, pp 78–84. IEEE, December 2015.

[16] Kubernetes Flex Volume Driver. Retrieved May 3, 2019, from https://github.com/kubernetes/community/blob/master/contributors/devel/sig-storage/flexvolume.md

[17] Cavoukian A., Privacy by Design, The 7 Foundational Principles, Information and Privacy Commissioner, Ontario, Canada, 2009.

[18] Sousa, T.B., Correia, F.F. and Ferreira, H.S. 2015. DevOps Patterns for Software Orchestration on Public and Private Clouds. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 22, October 2015.

[19] D. Adami et al., "Cloud and Network Service Orchestration in Software Defined Data Centers", SPECTS 2015.

[20] Weerasiri D, Barukh MC, Benatallah B, Sheng QZ, Ranjan R. A Taxonomy and Survey of Cloud Resource Orchestration Techniques. ACM Computing Surveys 2017, 50(2), 26.

AUTHORS

Dmitry Vasilenko is an Architect and Senior Software Engineer working on the IBM Watson and Cloud Platform. He received a M.S. degree in Electrical Engineering from Novosibirsk State Technical University, Russian Federation, in 1986. Before joining IBM SPSS in 1997 Mr. Vasilenko led Computer Aided Design projects in the area of Electrical Engineering at the Institute of Electric Power System and Electric Transmission Net works. His current research interests include communication engineering, network security, computer engineering and cloud computing.

Mahesh Kurapati is an Advisory Software Engineer in the Business Analytics Department of the IBM Software Group. He received a B.E. degree in Electronics Engineering from Bangalore University, India, in 1993 and Specialization on P.C. Based Instrumentation from Indian Institute of Sciences, Bangalore, India. Before joining IBM in 2006 Mr. Kurapati was involved in various telecommunications and data mining projects. At IBM, Mr. Kurapati's primary focus is on the development and operations for the Watson Studio line of products.