Designing and Implementing Elastically Scalable Services
A State-of-the-art Technology Review

Kiyana Bahadori and Tullio Vardanega
Brain, Mind, Computer Science PhD Course, University of Padova, Italy

Keywords: Service, Service-Oriented Architecture, Microservice Architecture, Cloud Computing, Elastic Scalabiity, Orchestration.

Abstract: The prospect of fast and affordable on-demand service delivery over the Internet proceeds from the very notion of Cloud Computing. For service providers, the ability to afford those benefits to the user is contingent on attaining rapid elasticity in service design and implementation, which is a very open research goal as yet. With a view to this challenge, this paper draws a trajectory that, starting from a better understanding of the principal service design features, relates them to the microservice architectural style and its implications on elastic scalability, most notably dynamic orchestration, and concludes reviewing how well state-of-the-art technology fares for their implementation.

1 INTRODUCTION

Cloud computing has imposed itself as an attractive novel operational model for hosting and delivering IT services over the Internet (Zhang et al., 2010). Its central tenet of as-a-service provisioning, to the user (from the application perspective) and to the service provider (from the implementation perspective), meets the technology and economic requirements of todays acquisition pattern of IT infrastructure (Armbrust et al., 2009).

In fact, the promised benefits of cost-effectiveness favoring pay-as-you-go pricing model, scalability and reliability, appeal major IT companies to meet their business objectives in a Cloud environment (Andrikopoulos et al., 2013)(Buyya et al., 2009)(Khajeh-Hosseini et al., 2012)(Tran et al., 2011). However, from the service provider point of view, the cost is outweighed by economic benefits of *elasticity* and transference of risk especially in an occurrence of over and under-provisioning of resource (Armbrust et al., 2010). In that context, the goal of service providers is to design and implement service portfolios capable of satisfying given service level objectives (SLO), in the face of fluctuating user demand, while keeping operational costs at bay. Elasticity as the ability to conserve resources at a fine grain with as fast as possible lead time allows matching resource to workload much more closely which result in cost savings for service providers (Armbrust et al., 2010). To this end, service design and implementation must both seek and provide for elastic scalability (Herbst et al., 2013). In this paper, we address challenges related to that endeavour.

Contribution: This paper first attempt to clarify the notion of service and its requirements which is much overloaded and frequently misunderstood. In the quest for a service-based architecture that can guide service design, it justifies the adaptation of the microservice architecture style. Subsequently, we look into state-of-the-art technology to assist the implementation of elastic scalability, concentrating on dynamic orchestration, which is one particular facet of that general quality. This work addresses two specific research questions. (**RQ1**) What are the design requirements that most relate to elasticity? (**RQ2**) How far does state-of-the-art technology help achieve elasticity?

The remainder of this paper is organized as follows: We briefly explain the notion of service in Section 2. Section 3 discusses the design implications of service orientation. Section 4 shows how containerization is the response to the rapid horizontal scalability and relates containers to microservices, highlighting how the former provides best fitting support for the adoption of the later. Section 5 assesses how state-of-the-art technology solutions cover the needs for dynamic orchestration. Section 6 draws some conclusions.

Bahadori, K. and Vardanega, T.
Designing and Implementing Elastically Scalable Services.
DOI: 10.5220/0006781705570564
In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*, pages 557-564
ISBN: 978-989-758-295-0

2 UNDERSTANDING THE NOTION OF SERVICE

The first widespread use of the notion of service, in relation to software systems, arose as part of the Service Oriented Architecture (SOA) initiative. In the context of SOA, the term *service* is defined as an independent logical unit, which provides functionality for a specific business process and can be composed with other services to form an application (Erl, 2007).

Another connotation of service emerged in Cloud Computing as part of the X-as-a-service model, coined to evoke the manner of contract-based, pay-as-you-go utility delivering (Fehling et al., 2014).

The union of the two perspectives sets a most ambitious goal: building a software application by agile aggregation of service units in such a way that the exposed capabilities are realized with the least resource consumption, and then delivered and consumed as metered utilities with assured quality of service.

Several qualities are required of services to fit this bill. In this paper, we briefly review them, to give directions to service designers. We begin our review from the need for services to be *composable* and *independently deployable* (Serrano et al., 2011)(Stine, 2015)(Mazmanov et al., 2013).

2.1 Composability and Independent Deployability

Enterprise IT architectures have to leverage and improve the existing suite of applications to address changing the business requirements rapidly (Kim et al., 2016; Humble and Molesky, 2011). Other than for deprecated silo-ed monoliths (Richardson, 2017), the functionality provided by a single service is normally insufficient to respond alone to all user requests. For example, the powering of an online shopping site rests on a multiplicity of services, ranging displaying, carting, payment processing, monitoring and shipping, which may either be procured from third-parties or provided by specialized development units inside the business organization. In other words, a service normally needs to collaborate with other services to pursue a specific business goal, and the fabric of such collaboration may change with the goal. This collaborative trait requires services to warrant *composability* (Serrano et al., 2011), i.e., the ability to participate, unchanged, in different aggregates as business demand requires (Tao et al., 2013)(Rao and Su, 2004). Another essential trait of a service that helps meet the requirements of elastic scalability is being *independently deployable* (desirably, by fully automated machinery (Lewis and Fowler, 2014)). It provides an ability for a developer to deploy, update, test and investigate directly on a particular service without affecting the rest of the system. Therefore each service can be scaled independently of other services. Moreover, the isolation of each service addresses the challenge of the technology stack. This, in fact, is completely different from using monolithic application where components must be deployed together. Composability and independent deployability proceed from service design with the following characteristics.

Explicit Boundaries. A service possesses an interface specification that describes its functionality and exposes for the use of other services. That interface, separate from the corresponding implementation, forms the boundary of that service, and contains all that one needs to know to interact with it. The boundary for a service is defined by means of a contract (Cibraro et al., 2010). A contract contains a schema and associated service policies (for style of interaction, persistence, guarantee, etc.), and is published using different mechanisms, initially via WSDL (Christensen et al., 2001), and later through REST API (Masse, 2011) and GRPC (Wang et al., 1993). The schema defines the structure of the data message, agnostic to programming languages (hence intrinsically interoperable), needed to request a specific service functionality.

Policy-driven Interoperability. W3C's notion of web service was the first practical solution to assure network-based interoperability (David Booth, 2004; Cohen, 2002). Service policies provide a configurable set of semantic stipulations concerning service expectations or requirements expressed in machine-readable language.

For example, an online shopping service may require a security policy enforcing a specific service level (requiring an ID) and the users who do not comply are not allowed to continue. The security policy can be used with other related services such as shipping. Augmenting service interfaces with policy stipulations strengthens the assurance of service guarantees across technology boundaries.

Loose Coupling. A well-defined service boundary goes hand in hand with the self-contained-ness of the functionality set exposed in its service interface, and therefore with the degree of loose coupling and autonomy of its implementation. Loose coupling is had when you do what your service interface declares (without incurring chains of dependencies) and do not place arbitrary constraints on your context of use

(hence being fit for reuse in multiple compositions) (Fehling et al., 2014).

Self-containedness. A service interface is self-contained if its implementation can be independently managed (for instance, replicated or migrated) and versioned (so long as in a backward-compatible manner) without affecting the rest of the system (Erl, 2007).

Statelessness. Separation between service execution and state information enables replication, which is one of the essential dimensions of scalability (Abbott and Fisher, 2009). Statelessness (Erl, 2007) is the name that designates this design quality.

3 BREAKING DOWN THE MONOLITH

Accepted wisdom has it that the journey toward well-versed service orientation, which fully meets the requirements discussed in section 2, conventionally departs from its farthest opposite: the rightfully deprecated monolith (Lewis and Fowler, 2014). The monolith architecture assembles all that the application needs to offer into one single sticky bundle, which epitomizes the notion of single point of failure, is unable to scale efficiently, and suffers the worst pain of the dependency hell (Merkel, 2014).

What is much less understood is the pitfall that trips most incautious practitioners: ending up building a distributed monolith formed of ”microliths”, that is, a collection of single-instance (not scalable) mono-services (only with a cooler name) that communicate over blocking protocols (Steel et al., 2017).

On the solid footsteps of successful adopters such as Amazon (Kamer, 2011), LinkedIn (Ihde, 2015), Netflix (Mauro, 2015), the end point of that transformative journey should arguably be the microservice architecture (Dragoni et al., 2016). This is no easy journey, though. One of the biggest difficulties in getting there safely is with cutting the right boundary for individual microservices, neither too coarse, which would be just one tad smaller monolith, nor too fine-grained, as in a distributed-object system, with painful service latency and maddeningly complex orchestration.

One driving criterion that helps cut the service boundary right is to focus on the scalability concern. Scalability is the ability to deploy cost-effective strategies for extending one’s capacity (Weinstock and Goodenough, 2006). Scalability can be obtained in two ways: vertical or horizontal. Vertical scaling (aka scale-up) is the ability to increase, in quantity or capacity, the resources availed to a single instance of the service of interest. The extent of this strategy is evidently upper-bounded by the capacity of the hosting server (Varia, 2010; Vaquero et al., 2011). Horizontal scaling (aka scale-out) is the ability to aggregate multiple units, transparently, into a single logical entity to adapt to different workload profiles (Beaumont, 2017). Replication is one central dimension to this strategy.

In fact, to meet the goals stated earlier, we need to pursue *elastic* scalability (aka elasticity), that is, the combination of strategy and means that allow dynamic resource provisioning and releasing, while preserving service continuity, and that are amenable to full automation (Armbrust et al., 2010).

Speed and *precision* are the qualifying traits of elasticity (Herbst et al., 2013). As the number of requests for particular service increases (respectively, decreases), the speed of resource provisioning (respectively, releasing) should vary in accord with the speed of demand variation, fully transparent to the user. The latter quality, which one might call frugality, prevents wastefulness in the ratio of provisioning over need, by matching the footprint of resource deployment to the level of demand as exactly as the grain of service design allows. It is the balance of those two qualities that sanctions the goodness of the service boundary.

Numerous studies (e.g., (Namiot and Sneps-Sneppe, 2014; Balalaie et al., 2016; Balalaie et al., 2015; Dragoni et al., 2017; Villamizar et al., 2015; Krylovskiy et al., 2015)) show that adopting the microservice architecture helps pursue those goals.

To better understanding, we illustrate the methodology which monolithic and microservice architecture uses to handle the scalability in Figure 1.

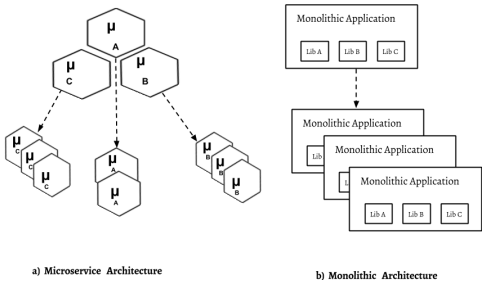

Figure 1: Scalability in monolithic and microservice architectures.

Having multiple independent units collaborate in providing a response to a user request most definitely incurs coordination overheads that must be addressed by *orchestration* (Venugopal, 2016).

Orchestration can be understood as the automated management and coordination of complex software system, constituted of collaborative parts, and their associated (computing) resources (Venugopal, 2016). The entry level of of orchestration entails automating the deployment of the application by means of *Continuous Integration and Continuous Delivery* (CI/CD) solutions such as Chef (Nelson-Smith, 2013), Ansible (Hall, 2013) and Jenkins (Jenkins, 2017).

Of course, to align with elastic scalability, orchestration must be dynamic, that is, able to support the adjustment of resource provisioning and service deployment required to either follow reactively or (better) adapt predictively to fluctuating user demand.

Having multiple independent units collaborate in providing a response to a user request most definitely incurs coordination overheads that must be addressed by *orchestration* (Venugopal, 2016).

Orchestration can be understood as the automated management and coordination of complex software system, constituted of collaborative parts, and their associated (computing) resources (Venugopal, 2016). The entry level of of orchestration entails automating the deployment of the application by means of *Continuous Integration and Continuous Delivery* (CI/CD) solutions such as Chef (Nelson-Smith, 2013), Ansible (Hall, 2013) and Jenkins (Jenkins, 2017).

Of course, to align with elastic scalability, orchestration must be dynamic, that is, able to support the adjustment of resource provisioning and service deployment required to either follow reactively or (better) adapt predictively to fluctuating user demand.

4 RAPID SERVICE SCALING USING CONTAINERIZATION

Virtualization is the fundamental techonology that utilize resources by creating virtual logical partition from a shared pool of resources to meet principle of elastic scalability for service (Armbrust et al., 2009)(Zhang et al., 2010). To this end, hypervised virtualization has been a major enabler for Cloud computing. Its primary bounty includes isolation (which it assures) and self-sufficiency, that is, no external dependency and perfect loose coupling (which it allows for), both being much desired qualities for service-oriented applications. In addition to, or perhaps as a reflection of, being too resource-costly, however, classic hypervised virtualization technology is fundamentally unable to respond to the rapidity requirements put forward by elastic horizontal scaling (Pahl and Xiong, 2013).

Container-based virtualization, which originates from the Linux Container project (LXC), is much better apt at the scaling, while still assuring excellent isolation (Pahl and Xiong, 2013). The container uses the kernel of the host operating system to run multiple root file systems. The name ”container” designates each such root file system (Docker, 2017) and holds the code and libraries that constitute the contained application, while sharing the host operating system with all other entities that run on it. This sharing is a blessing but also a weakness; the latter because it requires the application to actually run on the host OS, renouncing interoperability. Containers use `namespaces` to cater for isolation among processes and `cgroups` to limit and control resources usage in individual process.

Because of their leaner nature, containers consume less resources (so that one single host can accommodate many containers) and are lighter and easier to house and transport, faster to deploy, boot and shut down than virtual machines (Dua et al., 2014). It is not surprising therefore that containers were found to be up to 10 times faster to bootstrap than virtual machines (Felter et al., 2015).

Docker is an open-source container technology based on the LXC technique, which is built around a container engine (Docker, 2017). A Docker container is composed of two parts: a pile of images and one container. Images are a collection of different read-only file systems stacked on top of each other using the Advanced multi-layered Unification File-system (AuFS), which implements union mounting that allows separate file systems or directories to be overlaid into a single coherent file system. A container is the writable image that sits on the top layer of the said image stack. A microservice architecture is a collection of multiple independent deployable services written in a variety of language and frameworks, which communicate with high-level, application-independent protocols (Lewis and Fowler, 2014)(Newman, 2015). Each service needs to be provided with its required resources in a fast, reliable and cost-effective way, and then needs to be run, upgraded and replaced independently. These requirements speak of containers (Pahl, 2015). There are two approaches to deploying application using containers: one-to-one and one-to-many (Richardson, 2016). The classification criterion is the number of services housed in the container, which defines its *granularity*. The one app (service) per container approach allows each individual service to run on a dedicated container, so that one container hosts only one service and each service is packaged as a container image. This solution eases horizontal scaling, provides fast build/rebuild of the container, and allows

the same container to be reused for multiple different purposes in the same system. The major downside of this approach is that it may yield a large number of containers, and thus considerable overhead for interaction and management. The one-to-many solution, where each container houses multiple services, addresses this very concern, without however answering the question of which services to gather into which container.

5 REVIEWING TECHNOLOGY SUPPORT FOR DYNAMIC ORCHESTRATION

The complementary specification in the direction of designing and implementing elastic scalability for the definition of dynamic orchestration we discussed in Section 4 requires the technology solutions to support efficient resource allocation and coordination. The scheduler is an essential component of an ideal dynamic orchestration solution, as it is the entity in charge of generating a dynamic execution plan that responds to changes in user demand.

The scheduler must generate dynamic statistics on the state of the resources that it manages, to compute the best possible service-resource mapping, using either reactive or proactive strategies.

Reactive approaches respond to demand fluctuation when predictions are not available. Examples of this solution appear in most commercial solutions such as RightScale (RightScale, 2017) and AWS (Amazon, 2017).

Proactive approaches use predicted demand to allocate resources before they are needed (Gong et al., 2010; Nguyen et al., 2013; Dawoud et al., 2011). In using proactive approaches, the accuracy of demand prediction and provisioning is obviously critical to saving costs with utility computing (Tsai et al., 2010).

A modern scheduler should provide support for the following features:

- **Affinity / Anti-affinity**, to implement restrictions that enforce given entities to always be adjacent or separate to one another. An affinity rule simply means the ability to ensure that certain workload or virtual server always runs on the same host. Anti-affinity works in the opposite way, with the same impact as Affinity.
- **Taint and toleration**, to mark a particular (logical/physical) computing node as un-schedulable so that no container will be scheduled on it other than those that explicitly tolerate the taint. This requirement can also be used to keep away a node with a particular specification (e.g., a physical resource) from the others and dedicate it to given workload. It also allows rolling application upgrades of a cluster with almost no downtime.
- **Custom schedulers**, to delegate responsibility for scheduling an arbitrary subset of hosts to the user, alongside with the default scheduler.

To examine how current state-of-the-art technology meets the requirements for elasticity, we look in Kubernetes (Kubernetes, 2017)(Brewer, 2015), OpenShift (OpenShift, 2017), DockerSwarm (Docker, 2017), as the most widely used solutions for the dynamic orchestration.

Kubernetes. One of the first open source orchestration platform written in Google's Go programming language. Its architecture is based on a set of master and worker (Minion) nodes. Within each node, there are groups of containers called pods, which share resource and are deployed together. Pods play a significant role in placing workload across a cluster of nodes. Kubernetes uses architectural descriptions, written in JSON or YAML, to describe the desired state of services, which it feeds to the master node. The master node devolves all dynamic orchestration tasks onto worker nodes. Kubernetes provides `Required` and `Preferred` rules. Required rules need to be met before a pod can be scheduled. Preferred rules do not guarantee enforcement. Kubernetes also uses a replication controller with a reactive approach to instantiate pods as required.

Kubernetes provides pods and nodes with affinity /anti-affinity attributes. It defines rules as a set of labels for pods, which determines how nodes schedule them. Anti-affinity rules use negative operators.

Kubernetes marks nodes to avoid them to be scheduled. The `Kubectl` command creates a taint on nodes marks them un-schedulable by any pod that does not have a toleration for taint with the corresponding key-value.

Kuberenes at v1.6 supports customs schedulers in a way that allows users to provide a name for their custom scheduler and ignore the default one. Custom strategies are completely custom, meaning that the user must write an ad-hoc plugin to use them.

OpenShift. A dynamic orchestration platform built on top of Kubernetes, hence it inherits the capabilities of its ancestor. By default, its container platform scheduler is responsible for the placement of new pods onto nodes within the cluster. It reads data from the pod and tries to select the most appropriate node, based on its configuration policies. It does not

modify the pod, and simply creates a binding for the pod that ties it to the particular node.

Docker Swarm. It is a native orchestration tool for the Docker Engine, which supports deploying and running multi-container applications on different hosts. The term swarm refers to the cluster of physical/virtual machines (Docker engines), called nodes, where services are deployed. Its architecture uses one master and multiple worker nodes.

The key concepts in it are *services* and *tasks*. Service designates the tasks that need to execute on the worker nodes, as specified in the tasks' desired state. Tasks represent atomic scheduling units of work associated to a specific container. Docker Swarm uses a declarative description, called Compose file, written in YAML to describe services.

The swarm manager is responsible for scheduling the task (container) to worker nodes (hosts). Docker Swarm places a set of filters on nodes and containers to support the first two requirements mentioned earlier. Node filters `constraint` and `health` operate on Docker hosts to select a subset of nodes for scheduling. The node filter `containerslots` with a number value is used to prevent launching containers above a given threshold on the node.

Docker Swarm offers three strategies, called `spread`, `binpack`, `random`, to manage cluster assignment according to need. Under the spread strategy (which is the default one), it computes ranking according to the nodes available CPU and RAM, attempting to balance load across nodes. The binpack strategy attempts to fill up the most used hosts, leaving spare capacity in less used ones. The random strategy (which is primarily intended for debugging) assigns computation randomly among the nodes that can schedule it.

As mentioned earlier, scheduler is an essential component to achieve optimal provisioning that response to changes in user demand.

Concentrating on the scheduler to achieve optimal provisioning in the mentioned technologies, we observe that all are using predictive approaches to achieve elasticity. As, using resources have boot-times that vary, depending on the application, from a couple of minutes to even 30-40 minutes to load all the components needed. Even in the container world, spawning a new large-image container on a new node while the network is under stress, will easily take 5 to 10 minutes. The same holds for shut-down times. Therefore, the longer the boot time will result in consuming more resource and less efficiency. So, the more important it becomes to be proactive and provide required resources in advance to achieve effective elasticity. This justifies investigating proactive algorithms in place / along with reactive ones to achieve effective elasticity.

6 CONCLUSIONS

The technology exploration that we have made in this paper shows that there continues to be a gap between what state-of-the-art solutions have to offer in the way of support for elastic scalability and the full range of requirements that such a need entails. Accordingly, proactive approaches to resource scheduling are the most acute need that is not supported as yet other than in early research prototypes. To this end, our future research goal is to design, implement and comparatively evaluate experimental proactive resource allocation algorithms, which Cloud service provider could employ to achieve rapid elasticity and leverage it for a more proficient use of DevOps.

REFERENCES

Abbott, M. L. and Fisher, M. T. (2009). *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. Addison-Wesley Professional, 1st edition.

Amazon (2017). Amazon Web Service. https://aws.amazon.com/.

Andrikopoulos, V., Binz, T., Leymann, F., and Strauch, S. (2013). How to adapt applications for the cloud environment. *Computing*, 95(6):493–535.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. (2010). A view of cloud computing. *Commun. ACM*, 53(4):50–58.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., Lee, G., Patterson, D. A., Rabkin, A., Stoica, I., et al. (2009). Above the clouds: A berkeley view of cloud computing. Technical report, Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley.

Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2015). Migrating to cloud-native architectures using microservices: an experience report. In *European Conference on Service-Oriented and Cloud Computing*, pages 201–215. Springer.

Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2016). Microservices architecture enables devops: migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52.

Beaumont, D. (2017). How to explain vertical and horizontal scaling in the cloud. https://www.ibm.com/blogs/cloud-computing/2014/04/explain-vertical-horizontal-scaling-cloud/.

Brewer, E. A. (2015). Kubernetes and the path to cloud native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 167–167, New York, NY, USA. ACM.

Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., and Brandic, I. (2009). Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616.

Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. (2001). Web Service Description Language (WSDL) 1.1 W3C Note. Technical report, World Wide Web Consortium (W3C).

Cibraro, P., Claeys, K., Cozzolino, F., and Grabner, J. (2010). *Professional WCF 4: Windows communication foundation with. NET 4*. John Wiley & Sons.

Cohen, F. (2002). Understanding web service interoperability. *IBM Technical Library*.

David Booth, Hewlett-Packard, H. H. F. M. E. N. I. M. C. C. F. D. O. (11 February2004). Web services architecture, w3c working group note.

Dawoud, W., Takouna, I., and Meinel, C. (2011). Elastic vm for cloud resources provisioning optimization. *Advances in Computing and Communications*, pages 431–445.

Docker (2017). What is docker. https://www.docker.com/what-docker.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2016). Microservices: yesterday, today, and tomorrow. *arXiv preprint arXiv:1606.04036*.

Dragoni, N., Lanese, I., Larsen, S. T., Mazzara, M., Mustafin, R., and Safina, L. (2017). Microservices: How to make your application scale. *arXiv preprint arXiv:1702.07149*.

Dua, R., Raja, A. R., and Kakadia, D. (2014). Virtualization vs containerization to support paas. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 610–614. IEEE.

Erl, T. (2007). *Soa: principles of service design*. Prentice Hall Press.

Fehling, C., Leymann, F., Retter, R., Schupeck, W., and Arbitter, P. (2014). *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer Science & Business Media.

Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE.

Gong, Z., Gu, X., and Wilkes, J. (2010). Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16. Ieee.

Hall, D. (2013). *Ansible Configuration Management*. Packt Publishing.

Herbst, N. R., Kounev, S., and Reussner, R. (2013). Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 23–27.

Humble, J. and Molesky, J. (2011). Why enterprises must adopt devops to enable continuous delivery. *Cutter IT Journal*, 24(8):6.

Ihde, S. (2015). InfoQ — From a Monolith to Microservices + REST: the Evolution of LinkedIn's Service Architecture. [Online]. http://www.infoq.com/presentations/linkedin-microservices-urn/.

Jenkins (2017). Jenkins. https://jenkins.io/.

Kamer, S. (2011). GIGAOM, The Biggest Thing Amazon Got Right: The Platform. [Online]. https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/.

Khajeh-Hosseini, A., Greenwood, D., Smith, J. W., and Sommerville, I. (2012). The cloud adoption toolkit: supporting cloud adoption decisions in the enterprise. *Software: Practice and Experience*, 42(4):447–465.

Kim, G., Debois, P., Willis, J., and Humble, J. (2016). *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press.

Krylovskiy, A., Jahn, M., and Patti, E. (2015). Designing a smart city internet of things platform with microservice architecture. In *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*, pages 25–30. IEEE.

Kubernetes (2017). Kubernetes. https://kubernetes.io/docs/tutorials/kubernetes-basics/.

Lewis, J. and Fowler, M. (2014). Microservices: a definition of this new architectural term. *Mars*.

Masse, M. (2011). *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.".

Mauro, T. (2015). Nginx — Adopting Microservices at Netflix: Lessons for Architectural Design.[Online]. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

Mazmanov, D., Curescu, C., Olsson, H., Ton, A., and Kempf, J. (2013). Handling performance sensitive native cloud applications with distributed cloud computing and sla management. In *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, UCC '13, pages 470–475, Washington, DC, USA. IEEE Computer Society.

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2.

Namiot, D. and Sneps-Sneppe, M. (2014). On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27.

Nelson-Smith, S. (2013). *Chef: The Definitive Guide*. O'Reilly Media, Inc.

Newman, S. (2015). *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.".

Nguyen, H., Shen, Z., Gu, X., Subbiah, S., and Wilkes, J. (2013). Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *ICAC*, volume 13, pages 69–82.

OpenShift (2017). Container Application Platform by the Open Source Leader. https://www.openshift.com/.

Pahl, C. (2015). Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31.

Pahl, C. and Xiong, H. (2013). Migration to paas clouds-migration process and architectural concerns. In *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2013 IEEE 7th International Symposium on the*, pages 86–91. IEEE.

Rao, J. and Su, X. (2004). A survey of automated web service composition methods. In *SWSWPC*, volume 3387, pages 43–54. Springer.

Richardson, C. (2017). Monolithic architecture. http://microservices.io/patterns/monolithic.html.

Richardson, C. (February 10, 2016). Choosing a Microservices Deployment Strategy. https://www.nginx.com/blog/deploying-microservices/.

RightScale (2017). RightScale. https://www.rightscale.com/.

Serrano, M., Shi, L., Foghlú, M. Ó., and Donnelly, W. (2011). Cloud services composition support by using semantic annotation and linked data. In *International Joint Conference on Knowledge Discovery, Knowledge Engineering, and Knowledge Management*, pages 278–293. Springer.

Steel, E., Berube, Y., Bonér, J., Britton, K., and Coatta, T. (2017). Hootsuite: In pursuit of reactive systems. *ACM Queue*, 15(3):60.

Stine, M. (2015). Migrating to cloud-native application architectures.

Tao, F., LaiLi, Y., Xu, L., and Zhang, L. (2013). Fc-paco-rm: a parallel method for service composition optimal-selection in cloud manufacturing system. *IEEE Transactions on Industrial Informatics*, 9(4):2023–2033.

Tran, V., Keung, J., Liu, A., and Fekete, A. (2011). Application migration to cloud: a taxonomy of critical factors. In *Proceedings of the 2nd international workshop on software engineering for cloud computing*, pages 22–28. ACM.

Tsai, W.-T., Sun, X., and Balasooriya, J. (2010). Service-oriented cloud computing architecture. In *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pages 684–689. IEEE.

Vaquero, L. M., Rodero-Merino, L., and Buyya, R. (2011). Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Re*, 41(1):45–52.

Varia, J. (2010). Architecting for the cloud: Best practices. *Amazon Web Services*, 1:1–21.

Venugopal, S. (2016). Cloud orchestration technologies,ibm. https://www.ibm.com/developerworks/cloud/library/cl-cloud-orchestration-technologies-trs/index.html.

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., and Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Computing Colombian Conference (10CCC), 2015 10th*, pages 583–590. IEEE.

Wang, X., Zhao, H., and Zhu, J. (1993). Grpc: A communication cooperation mechanism in distributed systems. *SIGOPS Oper. Syst. Rev.*, 27(3):75–86.

Weinstock, C. B. and Goodenough, J. B. (2006). On system scalability. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

Zhang, Q., Cheng, L., and Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18.