Security Considerations for Microservice Architectures

Daniel Richter, Tim Neumann and Andreas Polze
Hasso Plattner Institute at University of Potsdam, P.O.Box 90 04 60, D-14440 Potsdam, Germany

Keywords: Security, Dependability, Cloud Infrastructure, Microservices.

Abstract: Security is an important and difficult topic in today's complex computer systems. Cloud-based systems adopting microservice architectures complicate that analysis by introducing additional layers. In the test system analyzed, base layers are combined into three groups (compute provider, encapsulation technology, and deployment) and possible security risks introduced by technologies used in these layers are analyzed. The application layer focuses on security concerns that concern authorization and authentication. The analysis is based on a microservice-based rewritten version of the seat reservation system of the Deutsche Bahn using technologies such as Amazon Web Services, Docker, and Kubernetes. The comparison concludes that the security of communication in the test system could be significantly improved with little effort. If security is not considered as an integral part from the beginning of a project, it can easily be neglected and be expensive to add later on.

1 INTRODUCTION

In microservice architectures, a complex system is split into multiple, small and mostly independently operating components, which communicate only via well-defined interfaces. This allows each component to be developed, tested, and scaled independently ((Richardson, 2017; Newman, 2015; Horsdal, 2016; Fowler, 2016)). While microservice architectures can reduce the complexity of a given system, it usually introduces – in comparison to monolithic applications – additional complexity through dependencies to supporting technology e.g. for deployment, scaling and management of containerized applications. In addition, the use of additional technologies increases the surface attack area ((Dragoni et al., 2017)).

To get an overview of technologies dependencies that a introduced in cloud-based applications that adopt a microservice architecture, we built an application based on Amazon Web Services, Docker, and Kubernetes, which is experimental, microservice-enabled reimplementation of the electronic seat reservation system of the Deutsche Bahn. It consists of a customer component (responsible for managing login data), a seat component (providing queryable train schedules and available seats) and a booking component (managing all booking data). Additionally, each of these components is backed by a separate database. The front-ends were developed for two display devices (single-page web/mobile application and ticket machine) for which four additional services were introduced (two front-end services and two Backend-for-Frontend services).

Our test system is deployed to Amazon Web Services (AWS). AWS introduces a variety of additional layers into the system: Firstly, the actual physical computers in an AWS data center. This is followed by three core AWS compute resources: *Elastic Compute Cloud* (EC2)[1], which provides and manages virtual machines; *Elastic Block Storage* (EBS)[2], which provides networked data storage volumes to EC2 instances; and *Virtual Private Cloud* (VPC)[3], which offers isolated networks for EC2 instances. Inside AWS, the test system consists of several EC2 instances. All EC2 Kubernetes nodes run the Kubernetes node administration software, responsible for further running other software on the node. The other important piece of software running on Kubernetes nodes is Docker, which manages the individual containers used to run the actual software deployed by Kubernetes.

To simplify the analysis, we split our testbed into three base layer groups: *compute provider*, *encapsulation technology* and *deployment*. The highest layer, the *application layer*, is the most complex layer in the

[1] https://aws.amazon.com/ec2/details/
[2] https://aws.amazon.com/ebs/details/
[3] https://aws.amazon.com/vpc/details/

Richter, D., Neumann, T. and Polze, A.
Security Considerations for Microservice Architectures.
DOI: 10.5220/0006791006080615
In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*, pages 608-615
ISBN: 978-989-758-295-0

system. Our focus was to secure the communication between individual application components (*authentication and authorization*).

2 THE BASE LAYERS

In this section, the base layers – all layers except the application layer – are analyzed. Since multiple layers often work together to provide one function, the layers have been organized into the three groups *compute provider*, *encapsulation technology* and *deployment*. Following that comparison, the security of the chosen technologies is analyzed.

2.1 Technologies for Layer Groups

As the technology used in each layer can directly impact its security analysis, we first compare multiple alternative technologies for each layer group. Our test system's layers are grouped as follows:

The **compute provider** group consists of all AWS related layers and generally provides some kind of computing infrastructure consisting of either physical or virtual machines, some networking solution, and some file storage system.

The **encapsulation technology** group mainly consists of the Docker layer and the Weave layer. Both can be used independently of each other; here, they work together to provide a distributed runtime environment for containers. This group is responsible for isolating services from each other so they cannot interfere with each other (except by predefined communication).

The **deployment** group contains the Kubernetes layers and is responsible for taking software in source or binary format and ensuring its execution and configuration.

2.1.1 Compute Provider

A compute provider is required to provide the infrastructure to run some software. The core functionalities are: Starting a new machine (based on some template) and connect it to some network. This usually involves assigning some kind of computing capacity to the new machine, configuring it and starting it. There are two types of machines which can be distinguished:

To start a **physical machine**, some hardware is allocated and configured. In some situations, this may involve purchasing and installing the hardware beforehand.

A physical machine itself can run multiple **virtual machine**s. Starting a virtual machine usually involves allocating some capacity on an existing physical machine and then starting it from some predefined image.

Another important classification is the type of provision: A data center owned by the company planning to use the compute provider, a data center operated by a third party, or a cloud provider (provider of mostly virtual machines with the additional restriction that new machines can be requested in an automated fashion, using an API).

Since cloud providers are much more modern than the data center-based approaches, they were the technology of choice for our testbed. The most commonly known commercial cloud providers are AWS, *Google Cloud Platform* (GCP)[4] and *Microsoft Azure*[5] ((Coles, 2017)). The most popular self-hosted cloud provider is *OpenStack*[6] ((Buest, 2014)), although it is also offered in hosted form by various third parties. Even though the cloud providers are mostly equal in functionality, AWS was chosen for two reasons: AWS was by far the largest cloud provider ((Coles, 2017)), and it was also the cloud provider of choice of Deutsche Bahn, our project partner.

2.1.2 Encapsulation Technology

Encapsulation could be achieved by running each service on a separate machine. The decision to use a separate encapsulation layer was made to achieve a higher degree of flexibility. In microservice architectures services usually are very lightweight and may only run for a short period of time. This makes for example the use of machines provided by AWS uneconomical: AWS bills at least one hour for a started instance and even the smallest EC2 instance type is was too large for a single service. Therefore, an encapsulation technology which allows running multiple services on one EC2 instance was needed.

On modern operating systems, there are generally two different encapsulation technologies:

VM-based encapsulation Each encapsulated process runs in its own virtual machine. This involves some overhead, since hardware such as storage devices needs to be simulated. Since a virtual machine requires an entire operating system running inside it, they are generally rather heavy-weight.

Container-based encapsulation Operating system-provided methods are used to isolate processes

[4]https://cloud.google.com/

[5]https://azure.microsoft.com/en-us/

[6]https://www.openstack.org/

on the host system. This imposes a smaller overhead than a VM-based approach and also allows resources to be easily shared between encapsulated processes or with the host. As a limitation, only encapsulating software for the same operating system as the host is supported.

Each technology has several advantages and disadvantages; the most important arguments are listed below:

(a) VM-based solutions provide greater isolation than container-based solutions. A vulnerability in the encapsulated software could thus have a greater impact when using containers.

(b) Container-based solutions have a lower overhead, which allows for more efficient usage of computing resources ((Felter et al., 2015)).

(c) VM-based solutions can run software independently from the host operating system, whereas container-based solutions only support software written for the host operating system.

(d) There is an abundance of tools, infrastructure, and pre-built software available for mainstream container-encapsulation technology. This is not necessarily applicable for VM-based solutions.

For our testbed, the choice fell on Docker – a container-based approach –, which has been widely used in IT projects in recent years and for which a lot of tools and pre-built software is available.

One further layer is part of the encapsulation group: The **networking layer**. If containers or virtual machines are used, multiple network addresses (one for each encapsulated piece of software) are needed. Additionally, it may is desirable to allow encapsulated applications to communicate with each other but nit with the machines they are running on. As such, a separate network is usually required for encapsulated applications. Some technologies require special support from the host's networking hardware, whereas others build so-called *overlay networks* ((Galuba and Girdzijauskas, 2009)) where each machine runs a special software which wraps network packets destined for another machine with some metadata and sends the wrapped packets to the other machine using a physical network.

Our testbed uses Weave Net that provides a virtual network, available on all nodes, which is used by the software deployed by Kubernetes to communicate.

2.1.3 Deployment

Deployment refers to the action of taking a piece of software, configuring it, and ensuring it is running on some machine. While this can be done "by hand", an automated system was required to allow smooth operation of the testbed and avoid an error-prone, manual process: The project's continuous integration infrastructure required setting up entirely new environments, each with about half a dozen services.

There are several technologies which can be used to distribute containers among multiple nodes, with popular choices being *Docker Swarm*[7] and Kubernetes.

2.2 Security Evaluation of Base Layer Technologies

Given our testbed, we shortly discuss selected security aspects of the chosen base layer technologies.

2.2.1 Compute Provider

As the data center is managed by Amazon, the security there cannot be influenced by its customers. However, Amazon states that its data centers comply with various commercial and governmental security guidelines ((Amazon Web Services, 2017)) such as *PCI DSS Level 1* (Payment Card Industry Data Security Standard). Among others, PCI DSS requires a) "Restricting physical access to cardholder data" (which, in the context of AWS, means that physical access to the actual hardware must be restricted), b) "Track and monitor all access to network resources and cardholder data" and c) "Regularly test security systems and processes" ((PCI Security Standards Council, 2016)). Given this certification and others, for the scope of this analysis it can be assumed that a AWS data center and hardware is set-up and managed in a secure manner.

AWS allows the modification of resources by using either a web interface (*AWS Management Console*) or an API. Access to the Management Console is secured using a username and password and, depending on the configuration, a two-factor authentication token. Accessing the API requires an access key. AWS offers a fine-grained permission system and the *CloudTrail*[8] service which, if configured, records all access to AWS resources along with which user accessed the resource and how they authenticated to do so.

AWS allows the creation of detailed rules for communication between EC2 instances. This feature is used extensively in the test system. Figure 1 shows inbound network rules for an ingress node – which ports are open is tightly restricted with only those ab-

[7] https://docs.docker.com/engine/swarm/

[8] https://aws.amazon.com/de/cloudtrail/

Security Groups associated with i-01158862e30a7a323

Ports	Protocol	Source	HPI-Kube	HPI-Kube-Ingress	HPI-Internal-SSH
6783-6784	udp	sg-0234f469	✔		
6781	tcp	sg-0234f469	✔		
6443	tcp	sg-0234f469, sg-61408f0a	✔		
10255	tcp	sg-0234f469	✔		
6783	tcp	sg-0234f469	✔		
10250	tcp	sg-0234f469	✔		
9898	tcp	sg-0234f469	✔		
6782	tcp	sg-0234f469	✔		
4194	tcp	sg-0234f469	✔		
80	tcp	sg-63995708, sg-ac06c6c7		✔	
1194	udp	sg-ac06c6c7		✔	
53	udp	sg-63995708		✔	
443	tcp	sg-63995708, sg-ac06c6c7		✔	
22	tcp	sg-61408f0a, sg-ac06c6c7			✔

Figure 1: **Screenshot of the inbound network rules of an ingress node:** The ingress node is part of three security groups: The HPI-Internal-SSH security group (`sg-63995708`) allows SSH access from the bastion server (`sg-ac06c6c7`) and any ingress nodes. The HPI-Kube security group (`sg-0234f469`) allows communication between all Kubernetes nodes. The HPI-Kube-Ingress security group (`sg-61408f0a`) allows ingress nodes to receive network requests from the bastion server as well as members of the HPI-Internal-SSH security group.

solutely necessary to operate the test system being made available.

The EC2 instances should be secured as any other Linux server, however for simplicity we limited ourselves to just a small number of steps: a) we relied on Amazon VPC to act as a firewall instead installing one on-machine, b) we logged-in into a non-root account using the `sudo` program for privileged operations, and c) we have disabled password-based *SSH* access.

2.2.2 Encapsulation Technology

The encapsulation technology group contains two layers, Docker and Weave Net. Both were used in their default configuration: Docker allowed certain users full access to the computer on which it is installed, as it is required by Kubernetes. Weave Net was configured and managed by Kubernetes. Security-wise, the Weave Net default configuration could be improved by specifying a password to encrypt communication between the Weave Net instances running on each node.

2.2.3 Deployment

Kubernetes and Weave Net provide one network to all applications running in Kubernetes, allowing them to communicate without restrictions by default. By employing so-called Network Policies, communication can be limited to specific applications (similar to the inbound network rules of AWS VPC).

The Kubernetes API server allows the creation and modification of resources in the Kubernetes cluster. Kubernetes 1.5 – the current version when the testbed was initially set-up – provides very coarse-grained access control mechanisms (essentially either full or no access to the cluster, the API server even provided an unauthenticated and unencrypted endpoint); This changed with Kubernetes 1.6, which introduced *Role-Base Access Control* (RBAC), a fine-grained permission system.

3 THE APPLICATION LAYER

The application layer contains the individual application components and is the most complex layer in the test system. The security analysis of this layer will focus on securing the communication between those individual components. To simplify that analysis, the application components are grouped together into another set of layers, based on how information flows through the system.

The most important aspect of securing the communication between application components is preventing unauthorized access, which usually involves the processes of *authentication* and *authorization*.

3.1 Authentication and Authorization Methods

Various methods exist to implement authentication and authorization in IT systems, some common ones are listed below:

Trust The implemented service trusts that it is only accessed by those parties who should access it.

Network policy A network policy which prevents all but authorized parties from communicating with the service is enforced.

IP-based The service itself makes a decision based on the IP address where the request to the service originates from.

Key/token-based An access key, or access token, is transmitted with each request and only if a known and correct key is passed, access is granted to the service.

MAC-based (*Message Authentication Code*) The contents of the request, as well as the access key, are passed through a cryptographic hash function and then transmitted.

Signing-based & Certificate-based Asymmetric cryptography is used to sign the request.

Session-based & Password-based The first request to a service is unauthenticated and initiates the session. Subsequent requests identify the session they belong to by, for example, using one of the previous methods.

Table 1: Authentication Method Summary.

Method	Fine-grained access control	Secret-based	Session-based	Network-based	Stack Level
Trust	No	No	No	No	N/A
Network policy	No	No	No	Yes	Network
IP-based	Yes	No	No	Yes	Network/ Application
Key/token-based	Yes	Yes, pre-shared	No	No	Application
MAC-based	Yes	Yes, pre-shared	No	No	Application
Signing-based	Yes	Yes, asymmetric	No	No	Application
Certificate-based	Yes	Yes, asymmetric	No	No	Transport
Session-based	Yes, within a session	Yes, after session start	Yes	No	Application
Password-based	Yes	Yes, pre-shared and after session start	Yes	No	Application

To properly compare authentication and authorization methods and to analyze their applicability for the different communication channels in the next section, Table 1 gives a summary of all the methods based on various criteria. Each column corresponds to one of the criteria listed below:

Support of fine-grained access control classifies whether a method supports more granular permissions than either no access or unrestricted access. The addition "within a session" means that users can gain (exclusive) access to additional resources valid for the duration of their session.

Secret-based classifies whether a method requires clients to store and manage some kind of secret. The following types of secrets are distinguished:

- **pre-shared** A secret which must be known to the client and server before the initial request.
- **asymmetric** A secret for use with asymmetric cryptography – the client stores a secret key and the server recognizes the associated public key (based on a list of known public keys or using another level of asymmetric cryptography)
- **after session start** During the initiation of a session, a secret is sent to or generated on the client. This secret is used during subsequent requests.

Session-based classifies whether a method makes use of, or requires, sessions.

Network-based classifies whether a method is network-based.

Stack level classifies at which level in the technology stack a method operates. Three values are used:

- **Network** The method is implemented as part of, or relies on, the network.
- **Application** The method is implemented in the application itself.
- **Transport** The method is implemented in the transport layer (somewhere between the network and the actual application).

The following investigation of the individual communication channels is simplified by ordering the different authentication and authorization methods based on the level of security they provide.[9] To avoid duplicating that analysis, a conditional ordering is defined and given below:

Trust vs. network-based: Since trust provides no authentication and authorization, network-based methods are more secure than the trust method.

Network policy vs. IP-based The network policy-based method is generally preferable, as it is independent of the application. The IP-based method however, has the advantage that it supports fine-grained access control, so if that is needed, the IP-based method is the only possible network-based method.

Network-based vs. secret-based For both methods, the application itself is vulnerable: If an attacker is able to compromise the application, they can gain access to secrets available to the application and act on behalf of the application, using the available network interfaces. For the network-based methods, the network is additionally vulnerable: If an attacker gains sufficient access to

[9] Often multiple methods will be equally applicable to a channel, in which case the method of choice should be the most secure authentication and authorization method.

the network, they can impersonate the application. For the secret-based method, the secret distribution mechanism is an additional vulnerability. Thus, which method is more secure depends on whether a compromise of the network or the secret distribution mechanism is more likely.

Token-based vs. MAC-based Both methods work similarly, however for the MAC-based methods the token is never transmitted over the network, which decreases the risk of it being intercepted.

Mac-based vs. signing-based A signing-based method uses different keys on the client and server, reducing the risk of compromise. Additionally, depending on the exact implementation, keys can be generated independently of the server, which decreases the complexity and attack surface of the server.

Signing-based vs. certificate-based While they should offer the same security as signing-based methods, certificate-based methods have the advantage of standardization. This standardization makes it easier to replace an application using the certificate-based methods and also means a lower likelihood of introducing security vulnerabilities compared to implementing custom signing-based methods.

Session-based A session-based method is usually used in different situations: It can be used when using a secret is impossible and identifies one user over several consecutive requests, but not between sessions.

Password-based This method is mostly the same as the token-based method. However, instead of sending the token on every request, it is sent only on the first request; afterwards, some kind of session identifier is sent with each request. This makes it more secure than the token-based method, as the risk of leaking the token is reduced.

3.2 Evaluation of Authentication and Authorization in our Testbed

Our testbed is a simplified reimplementation of the *Elektronische Platzbuchungsanlage* (EPA, "electronic seat reservation and booking system") of Deutsche Bahn, that is responsible for managing seat reservations in trains all across Germany. It consists of:

Customer component This component is responsible for managing login data. It is mostly unused in the current project, as it focused on the ticket purchase and seat reservation process.

Seat component "Seat & schedule component" would probably be a more appropriate name for this component, however the name assigned by the previous project was kept for consistency. This component provides a queryable schedule of all trains, as well as the ability to access which seats are available on a given train.

Booking component This component manages all booking data: which routes were booked and which seats are reserved on which trains.

Additionally, each of those three components is backed by a separate database. The front-ends for those components were developed for two display devices: A single-page web/mobile application and a ticket machine user interface, which was also based on web technologies and built as a single-page web application.

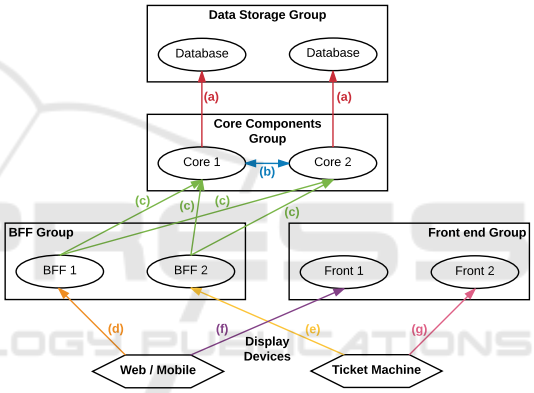

Figure 2: **Overview of Communication Groups:** The four rectangles represent the four communication groups, containing example services illustrated as ellipses. Each colored/labelled arrow represents one communication channel, with the arrowhead indicating the receiver of the communication request. The display devices are not part of any group and are represented as hexagons.

The components of our testbed have been grouped together in the following groups/layers to reduce the amount of communication channels which need to be considered:

Data Storage Group contains only the database backing the customer, seat and booking components. It is only accessed by the core components group, and there is no inter-group communication in the test set-up, although that is certainly possible in other situations.

Core Components Group contains the three core components, customer, seat and booking. These are accessed by the BFF group and do access the data storage group. Additionally, some requests

trigger inter-group communication between the components in the group.

Backend-for-Frontend (BFF) Group consists of the two BFFs,(Newman, 2015) one for each display device. These are accessed directly by the display devices and communicate with the core components group, if necessary. No inter-group communication happens between the different BFFs.

Front-End Group consists of the static web servers for the two front-ends. They are accessed by the display devices as well, however, they perform no other communication.

As shown in Figure 2, there are seven different communication channels between, within, and with the four groups. This number is further increased by the fact that the two display devices have not been combined, as they have different communication patterns depending on the version: The web/mobile front-end and BFF are open to the public. In this respect, no assumptions can be made about the requests sent to these services. It cannot be assumed that communication with these services will only take place from the official application and in the manner intended by the application developers. The BFF and the frontend must be able to handle all requests, including those made directly by a malicious third party, correctly. The opposite is the case with ticket vending machines: The team controls the hardware and possibly the network that is used for communication with the frontend and the BFF. Therefore, if necessary, it can be assumed that the communication from the ticket machine will only take place in the way intended. But even without this assumption, the hardware of the ticket vending machine is still controlled by the team and can be regarded as trustworthy – at least with a sufficiently secure hardware design – which enables additional security-relevant operations such as cryptography with pre-shared keys.

The communication channels distinguish themselves as follows:

(a) This communication channel interacts with third-party software, therefore the team did not have full control over the authentication and authorization methods used.

(b) Communication between different core components can usually be assumed to happen over a trusted network.

(c) Communication between the BFFs and core components is very similar to (b) except that they may reside on separate networks and that the BFFs may be considered untrusted since they are directly accessible from a public network.

(d) Communication takes place over a public network and originates from an untrusted device.

(e) Following the defense-in-depth approach, it was assumed, that communication takes place over a public network here as well. However, contrary to (d), communication originates from a trusted device.

(f) Once again communication happens over a public network from an untrusted device. Since the front-end services only offer static resources which must all be publicly accessible due to the nature of a web application, no authorization or authentication is required or possible here.

(g) As opposed to (f), resources accessed using this channel do not have to be publicly accessible. As such, some form of authorization and authentication can be implemented if the resources should remain inaccessible to the public. Similar to (e) it was again assumed that communication takes place over a public network.

In total, two authentication and authorization methods were used: Token-based authentication and authorization was used to connect to the database servers; session-based authentication and authorization was used for connections between the display devices and BFFs.

4 CONCLUSION AND FUTURE WORK

This paper evaluated the security of a microservice architecture: It first analyzed the security of the base layers, before focusing on authentication and authorization in the application layer. The practicality of multiple authentication and authorization methods was analyzed in the context of a reimplementation of the Elektronische Platzbuchungsanlage of Deutsche Bahn.

In comparison to monolithic applications, the use of cloud-infrastructure (compute provider layer) introduces additional complexity as well as additional attack vectors. Compared to classic VM-based cloud applications, technologies introduced in the encapsulation technology layer lead to the fact that more safety requirements have to be met. Currently, the analysis of additional security concerns is only limited to aspects regarding authorization and authentication (A2:2017; number two of ((Open Web Application Security Project, 2017)). But an increasing number of used technologies affects other risks, too: *security misconfiguration (A6:2017)*, *vulnerable compo-*

nents (A9:2017) or *insufficient logging and monitoring* (A10:2017). Also, Dev-ops, a software engineering culture and practice aimed at unifying development and operation that is often used in conjunction with microservices, introduces *non-production environment exposure* as a microservice-specific risk.

The main conclusions of this paper are that 1) modern computer systems are very complex, due to the many layers they are made up from, and 2) security is hard, takes effort, and should be an important consideration from the beginning of a project instead of an afterthought. At many points, security measures were not taken "for simplicity" or because "(human) resources were unavailable". While this may have been acceptable in the test system, a real-world product should never be launched with this many issues or areas of improvement.

We believe this shows very clearly why security is such a difficult topic: The benefits are hidden and the costs are high. The implementation of security in several microservices and in all system levels requires effort and careful planning. Once a project has started, security can easily be neglected for more immediately pressing concerns and may be difficult and even more expensive to add later. Even if security is a consideration from the beginning, there is often a choice between complexity and practicality. For example, to increase security, it would be possible to implement not only certificate-based authentication and authorization, but network policy-based authentication and authorization as a second layer of security. However, this would increase costs and complexity. Although the certificate-based method is clearly more secure than the certificate-based method, it is also more complex to implement than a token-based method because an additional infrastructure is required to manage all cryptographic keys.

As a final summary, we conclude that security should be a consideration from the very beginning of planning a system, to be able to implement effective and comprehensive security measures throughout the project – especially if monolithic applications are to be realized based on microservice applications.

ACKNOWLEDGEMENTS

The authors would like to thank Lena Feinbube, Leonard Marschke, Cornelius Pohl, Robert Beilich, Tim Basel, Timo Traulsen, Henry Hübler, Dr. Stephan Gerberding, Wolfgang Schwab, and Ingo Schwarzer for their support and assistance with this project.

REFERENCES

Amazon Web Services (2017). AWS Cloud Compliance. https://aws.amazon.com/compliance/. (visited on 2017/07/16).

Buest, R. (2014). Top 15 open source cloud computing technologies 2014. http://analystpov.com/cloud-computing/top-15-open-source-cloud-computing-technologies-2014-24727. (visited on 2017/07/16).

Coles, C. (2017). AWS vs Azure vs Google Cloud Market Share 2017. https://www.skyhighnetworks.com/cloud-security-blog/microsoft-azure-closes-iaas-adoption-gap-with-amazon-aws/. (visited on 2017/07/16).

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, Cham. DOI: 10.1007/978-3-319-67425-4_12.

Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172. DOI: 10.1109/ISPASS.2015.7095802.

Fowler, S. J. (2016). *Production-Ready Microservices: Building Stable, Reliable, Fault-Tolerant Systems*. O'Reilly Media. ISBN: 978-1-4919-6597-9.

Galuba, W. and Girdzijauskas, S. (2009). *Overlay Network*, pages 2008–2008. Springer US, Boston, MA. DOI: 10.1007/978-0-387-39940-9_1231.

Horsdal, C. (2016). *Microservices in .NET Core: With C#, the Nancy Framework, and Owin Middleware*. Manning Publications. ISBN: 978-1-61729-337-5.

Newman, S. (2015). *Building Microservices*. O'Reilly Media. ISBN: 978-1-4919-5035-7.

Open Web Application Security Project (2017). OWASP top 10 security risks 2017. https://www.owasp.org/index.php/ Top_10-2017_Top_10. (visited on 2018/01/24).

PCI Security Standards Council (2016). Payment card industry (pci) data security standard, v3.2. https://www.pcisecuritystandards.org/document_library?category=pcidss&document=pci_dss. (visited on 2017/07/16).

Richardson, C. (2017). *Microservice Patterns*. Manning Publications Co. ISBN: 978-1-61729-454-9.