From Serverful to Serverless: A Spectrum of Patterns for Hosting Application Components

Vladimir Yussupov[1], Jacopo Soldani[2], Uwe Breitenbücher[1], Antonio Brogi[2] and Frank Leymann[1]
[1]*Institute of Architecture of Application Systems, University of Stuttgart, Germany*
[2]*Department of Computer Science, University of Pisa, Italy*

Keywords: Application Deployment, Hosting, Patterns, Serverful, Serverless, Cloud Computing.

Abstract: The diversity of available cloud service models yields multiple hosting variants for application components. Moreover, the overall trend of reducing control over the infrastructure and scaling configuration makes it non-trivial to decide which hosting variant suits more a certain software component. In this work, we introduce a spectrum of component hosting patterns that covers various combinations of management responsibilities related to (i) the deployment stack required by a given component as well as (ii) required infrastructure resources and component's scaling rules. We validate the presented patterns by identifying and showing at least three real world occurrences of each pattern following the well-known Rule of Three.

1 INTRODUCTION

A plethora of available cloud service models enables building cloud applications using components hosted on various targets that require different management efforts. For example, cloud consumers can control lower-level infrastructure layers when hosting components using Infrastructure-as-a-Service (IaaS). Another option is to simply select an appropriate runtime when hosting components on Platform-as-a-Service (PaaS) or Function-as-a-Service (FaaS). In-between, Container-as-a-Service (CaaS) offerings facilitate hosting containerized components.

Such a diversity of hosting options allows flexibly outsourcing management responsibilities, e.g., hosting certain components on provider-managed offerings to minimize management efforts. Further extending this idea, the *serverless computing* paradigm (Baldini et al., 2017) focuses on developing applications comprising provider-managed components. The term *serverless* emphasizes a weaker role of processing, storage, and network resources for cloud consumers, as providers manage the underlying infrastructure, e.g., a FaaS platform and components it depends on. Serverless computing differs from its *serverful* counterpart in several aspects, namely (i) decoupling of computation and storage, (ii) code is executed without requiring consumers to allocate resources, and (iii) finer-grained pricing which focuses on the resources actually consumed (Jonas et al., 2019).

As a result, cloud consumers need to choose from component hosting variants with different management responsibilities. Typical questions that arise are "How to host a component requiring custom runtime and scaling rules?", or "How to host a periodically-invoked code snippet?". Unfortunately, currently there exist no well-structured guidelines for deciding on component hosting options. One instrument that can help in such cases is a *pattern*, since it describes a proven and abstract solution to a specific and frequently reoccurring problem (Alexander, 1977). Multiple *pattern languages* group solutions for certain domains, e.g., cloud computing (Fehling et al., 2014a), or enterprise integration patterns (Hohpe and Woolf, 2004). Hence, hosting-specific patterns could help making informed decisions about component hosting.

In this work, we introduce component hosting patterns which describe proven solutions for hosting application components. They cover different combinations of management responsibilities related to two aspects: (i) deployment stack and (ii) scaling configuration management. Here, the former describes to what extent cloud consumers can manage the underlying infrastructure and required software dependencies, whereas the latter is related to who is responsible for allocating infrastructure resources and defining scaling rules. Furthermore, we show how these patterns form a spectrum of hosting options, ranging from mainly consumer-managed, "serverful" to mainly provider-managed, "serverless" variant.

Yussupov, V., Soldani, J., Breitenbücher, U., Brogi, A. and Leymann, F.
From Serverful to Serverless: A Spectrum of Patterns for Hosting Application Components.
DOI: 10.5220/0010481002680279
In *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*, pages 268-279
ISBN: 978-989-758-510-4

2 PATTERN PRIMITIVES

In this section, we introduce the *pattern primitives* related to the domain of application deployment.
Application. An *application* is a collection of *software components* interacting with each other and external clients to provide a certain business functionality (Lau and Wang, 2007). Software components are typically installed and run on a specific *infrastructure* including processing, storage, and network resources (Messerschmitt, 2007). Depending on the chosen architectural style, software components can be coupled differently, e.g., remote procedure calls or messaging-based integration, which also influences to what extent the infrastructure can be shared among software components of the application.
Software Component. A *software component* is an application building block implementing functionalities required by an application (Councill and Heineman, 2001; Lau and Wang, 2007). Depending on their purpose, software components can be general-purpose or application-specific. General-purpose components provide functionalities not specific to a given application, e.g., a web server or a message-oriented middleware. Application-specific components implement part of the application's business logic, e.g., a Java-based e-commerce component.
Deployment Artifact. A *deployment artifact* is a software component packaged in a format required by a chosen deployment stack (OASIS, 2015). Examples of deployment artifacts are Docker container images packaging software components to host them on the Docker Engine, or packaging Java applications as Java Web application ARchives (WARs) to host them on application servers like JBoss.
Deployment Stack. A *deployment stack* is a valid combination of software components and infrastructure components (i.e., processing, storage, and network resources) that allows hosting a given deployment artifact. It is important to note that the same deployment artifact can be hosted on different deployment stacks. For example, a deployment artifact of type "Java ARchive" (JAR) can be hosted on a FaaS platform by selecting a suitable Java runtime, or by manually installing the Java runtime on a virtual machine provisioned using IaaS offerings. Figure 1 shows three valid deployment stacks for JAR files using service offerings from Amazon. The Stack #1 relies on AWS EC2, Amazon's IaaS offering, which allows cloud consumers to customize the deployment stack by installing any software components on the chosen virtual machine. Stack #2 shifts some management responsibilities to the provider, since the underlying container engine is provider-managed. However, cloud consumers are still able to customize the runtime environment by installing necessary software components as a part of the container. Finally, Stack #3 is managed mainly by Amazon, since cloud consumers only select a compatible provider-defined deployment stack, i.e., enabling Java8 in this case. Since deployment stacks might comprise *provider-managed* or *consumer-managed* components, the management efforts needed for the same deployment artifact differ depending on the chosen deployment stack. Furthermore, a *scaling configuration* for a given component that defines infrastructure resources and scaling rules needs to be managed too – either by cloud consumers or cloud providers.

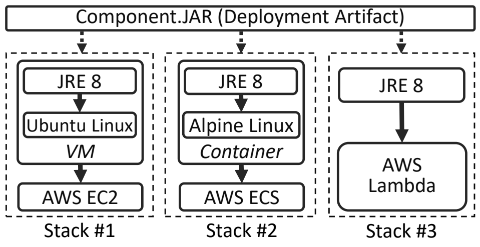

Figure 1: Different stacks for hosting a JAR on Amazon.

Consumer-managed Component. A component in deployment stack is *consumer-managed* if cloud consumers, e.g., application developers or operators, are responsible for managing it, e.g., installation, configuration, and dependency management. For example, in the Stack #2 from Figure 1, a Java runtime is installed as a part of a container image hosted on AWS ECS: while the underlying container engine is managed by the provider, the Java runtime is installed and configured by the cloud consumer. Similarly, if an operating system is consumer-managed as in the Stack #1 in Figure 1, cloud consumers are responsible for installing and configuring software components hosted on it, e.g., a NoSQL database.
Provider-managed Component. A *provider-managed stack component* is managed by providers. For example, a Java runtime selected in PaaS and FaaS offerings is provider-managed since the provider is responsible for installing and configuring the deployment stack required to run this runtime.
Scaling Configuration. A *scaling configuration* is a combination of component's scaling rules, e.g., horizontal vs. vertical scaling, and the amount of infrastructure resources that is required for hosting a given software component. As mentioned previously, a scaling configuration can be consumer-managed or provider-managed. For example, in the Stack #1 cloud consumers need to define the size of a virtual machine and the corresponding scaling rules, whereas in the Stack #3 the provider is responsible for allocating resources and scaling hosted functions.

3 APPLICATION COMPONENT HOSTING PATTERNS

In this section, we introduce the application component hosting patterns. Before discussing each pattern, we describe how patterns are aligned with each other, forming a spectrum of different hosting variants. Next, we describe how patterns are structured, and provide the in-detail definitions for each pattern.

3.1 A Spectrum of Hosting Patterns

Depending on the chosen cloud offering, both cloud providers and cloud consumers can be responsible for managing the underlying deployment stack, e.g., cloud consumers have more deployment stack management responsibilities when using IaaS in comparison with PaaS or FaaS. Moreover, the cloud offering choice also defines who manages the scaling configuration, e.g., providers are responsible for allocating infrastructure resources and scaling component instances in FaaS and certain CaaS offerings. Figure 2 shows a spectrum of component hosting patterns covering combinations of deployment stack and scaling configuration management responsibilities. To mark the spectrum's extremes, we use the terms *serverful* and *serverless* which emphasize the stronger or weaker role of processing, storage, and network resources for cloud consumers, respectively.

The most consumer-managed hosting option in Figure 2 is the *Serverful Hosting Pattern* (see Section 3.3) as it enables consumers to have full control over the deployment stack and scaling configuration including provisioning the desired infrastructure, installing software components, and defining component's scaling rules. A more simplified option is the *Consumer-managed Container Hosting Pattern* (see Section 3.4) in which the deployment stack required for running containers needs to be managed to a lesser extent, e.g., provider-managed container orchestrators which require scaling configuration to define infrastructure needed for running containers. Since container images serve as deployment artifacts, consumers can customize software components in such deployment stacks.

The next three patterns are more related to serverless-style component hosting as they are provider-managed in at least one of the aforementioned aspects. In the *Provider-defined Stack Hosting Pattern* (see Section 3.5), a deployment stack is provider-managed, but consumers need to manage the scaling configuration, e.g., Database-as-a-Service or PaaS offerings that require defining infrastructure resources for respective software components.

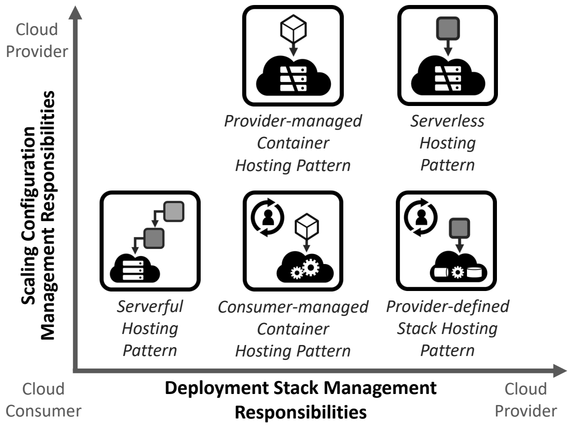
Cloud Provider
Scaling Configuration Management Responsibilities
Provider-managed Container Hosting Pattern
Serverless Hosting Pattern
Serverful Hosting Pattern
Consumer-managed Container Hosting Pattern
Provider-defined Stack Hosting Pattern
Cloud Consumer
Deployment Stack Management Responsibilities
Cloud Provider

Figure 2: A spectrum of component hosting patterns covering different combinations of deployment stack and scaling configuration management responsibilities.

In the *Provider-managed Container Hosting Pattern* (see Section 3.6), consumers no longer manage scaling configuration, i.e., providers allocate infrastructure resources required for running containers. Deployment stack management is also reduced: consumers can only modify the runtime environment via custom container images. Finally, the *Serverless Hosting Pattern* (see Section 3.7) apart from the provider-managed deployment stack also reduces the management overhead for scaling configuration, since providers are responsible for it, e.g., code snippets hosted using Function-as-a-Service (FaaS).

3.2 Patterns Structure

The patterns presented in the next subsections are structured following the best practices employed by researchers and practitioners (Alexander, 1977; Buschmann et al., 2007; Coplien, 1996; Gamma et al., 1995; Fehling et al., 2014a; Wellhausen and Fiesser, 2011; Richardson, 2018; Endres et al., 2017). Each component hosting pattern has a name and a catchy icon to simplify memorability. The *problem* and *context* are described in the eponymous paragraphs. Influencing factors that characterize the problem are described in the *forces* paragraph. The *solution* paragraph describes a possible solution with a solution sketch, and the *example* paragraph discusses a simple example of this pattern implementation using a ToDo list application in which a Java application stores and modifies to-do items in a NoSQL database. The *result* paragraph discusses the resulting context after the pattern is applied, and the *known uses* paragraph presents at least three real-world occurrences of the pattern implementation to demonstrate that the Rule of Three (Coplien, 1996) holds, hence, making the introduced pattern valid.

3.3 Serverful Hosting Pattern

Problem: *How to host a software component while retaining control over all deployment stack components and the scaling configuration?*

Context. A software component needs to be hosted on a custom deployment stack with possibly nested software layers such that the cloud consumer is able to define the infrastructure resources and scaling rules.
Forces. To choose the right deployment stack for a software component, cloud consumers need to understand the advantages and drawbacks of cloud service models and the corresponding management efforts. Other factors such as the chosen architectural style also affect the suitability of deployment stacks, e.g., hosting a monolith with many custom requirements vs. hosting a fine-grained microservice.
Solution. Host software components on a deployment stack primarily managed by cloud consumers which enables hosting all required components, e.g., an operating system and a web server on top of it. Hence, for a given deployment artifact type, the cloud consumer is responsible for installing an appropriate operating system and runtime environment. The chosen operating system either runs on a virtual machine, e.g., using IaaS offerings, or on a physical server, e.g., on premises or using bare metal cloud offerings. A solution sketch in Figure 3 shows a software component hosted on a deployment stack in which the cloud consumer is responsible for hosting the infrastructure components and managing scaling configuration. Hence, in the presented solution sketch the user is responsible for keeping track of all components in the stack to successfully complete the deployment, e.g., installing and running required dependencies.

Example. Figure 3 shows a simplified Java-based ToDo list application that stores to-do items in a NoSQL database hosted on the IaaS offering from AWS. Most components in the chosen deployment stack are managed by the cloud consumer. Another deployment stack variant could use a bare metal cloud offering instead of IaaS, allowing cloud consumers to also control the underlying hypervisor.
Result. When applied, this hosting pattern provides the highest degree of control over the deployment stack and scaling configuration. The provisioned infrastructure and all deployed components are managed and configured by cloud consumers. This allows fine-tuning each deployed component in the stack and enables manual integration with other components possibly deployed using more provider-managed offerings. However, since providers are not aware of all installed components in such deployment stacks, ensuring that the component is running and available becomes an additional responsibility for cloud consumers. An example of such additional effort would be providing a healthcheck configuration, e.g., AWS Elastic Load Balancing allows defining health check rules that allow it to control instances' health status.
Known Uses. Multiple IaaS offerings enable implementing this pattern, e.g., AWS EC2 (Amazon Web Services, 2021c), Azure IaaS (Microsoft, 2021d), and Google Compute Engine (Google, 2021c) enable provisioning virtual machines for hosting components and their dependencies, while cloud consumers are responsible for scaling configuration. Moreover, cloud providers offer bare metal servers, e.g., IBM Cloud Bare Metal Servers (IBM, 2021a) allows provisioning dedicated physical machines to set up any number of infrastructure layers for hosting desired application components that can be managed by consumers afterwards.

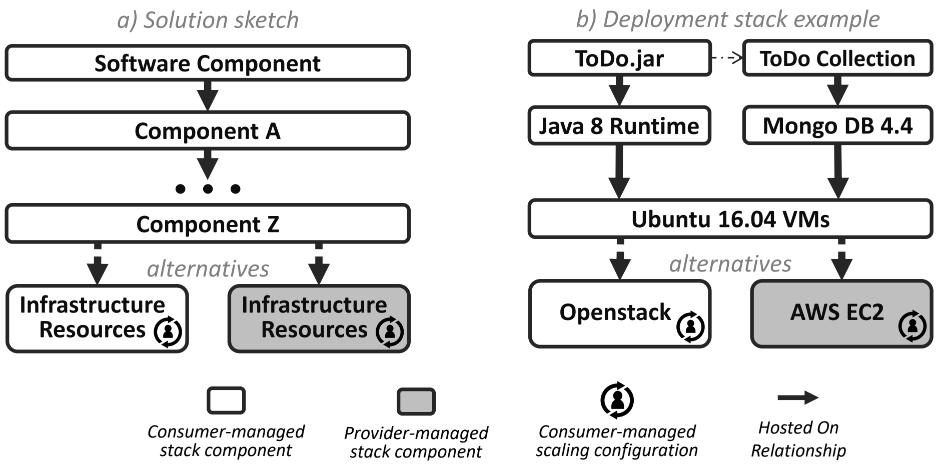

Figure 3: A solution sketch (a) and an example deployment (b) for the *Serverful Hosting Pattern*.

3.4 Consumer-managed Container Hosting Pattern

Problem: *How to host a software component while managing only the runtime environment it runs on and the scaling configuration?*

Context. A software component needs to be hosted such that its runtime environment is customizable, and the cloud consumer is responsible for managing the component's scaling configuration.

Forces. Containers help reducing the infrastructure costs and simplify software deployment. Moreover, provisioning tasks require additional technical expertise, e.g., network configuration, shifting the focus from business logic to infrastructure. At the same time, managing the component's runtime environment and scaling configuration might still be desired, e.g., add pre-installed proprietary libraries and fine-tune scaling rules for container instances.

Solution. Host software components on a deployment stack with provider-managed container engines or container orchestrators that allow managing scaling configuration. Figure 4 shows the solution sketch in which a containerized software component is hosted on a provider-managed container engine. Here, providers are responsible for deploying and managing the infrastructure for running containers, e.g., container orchestrator such as Kubernetes offered as a service. The cloud consumer must provide a container image encompassing the application component and its dependencies. Moreover, the cloud consumer is responsible for the scaling configuration including allocation of infrastructure resources used by container engine and the definition of scaling rules.

Example. Figure 4 shows a ToDo list application from Section 3.3 hosted on the managed Kubernetes service from AWS. Only the runtime environment in this deployment stack is managed by the cloud consumer as a part of the provided container image. Moreover, the consumer is responsible for the scaling configuration, including the required infrastructure resources and scaling rules.

Result. After applying this hosting pattern, the lower-level infrastructure layers in the deployment stack are managed by providers, although cloud consumers can manage the scaling configuration and runtime environment by defining custom container images with required dependencies. While the degree of control is reduced, users are still able to introduce modifications to the deployment stack and manage the hosted component, which makes this pattern a less demanding variant for hosting containerized components.

Known Uses. Managed container orchestrators such as Kubernetes-as-a-Service offerings are available in multiple public clouds, including such examples as IBM Cloud Kubernetes Service (IBM, 2021b), Azure Kubernetes Service (Microsoft, 2021e), and AWS Elastic Kubernetes Service (Amazon Web Services, 2021f). While managed Kubernetes offerings might vary significantly feature-wise, the overall goal to have a managed Kubernetes cluster allows deploying and running user-provided container images, hence shifting most management efforts to container-specific tasks. Moreover, some CaaS offerings such as AWS Elastic Container Service (Amazon Web Services, 2021e) with the EC2-based pricing mode allow hosting containers such that consumers manage the infrastructure resources for running containers and define the scaling rules for resulting container instances.

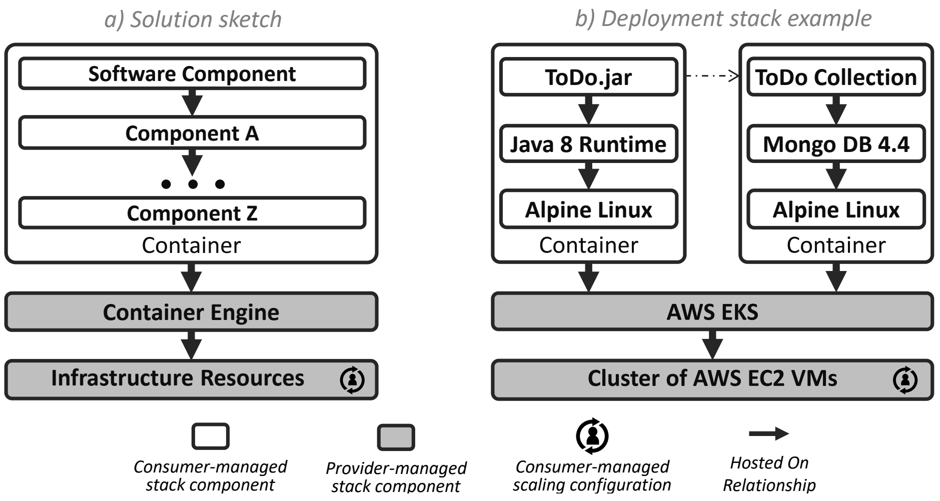

Figure 4: A solution sketch (a) and an example deployment (b) for the *Consumer-managed Container Hosting Pattern*.

3.5 Provider-defined Stack Hosting Pattern

Problem: *How to host a software component without managing the deployment stack while retaining control over the scaling configuration?*

Context. A software component needs to be hosted on any compatible deployment stack without other stack management requirements, but with cloud consumers managing the scaling configuration.

Forces. Often, software components are ready to be hosted on a standard platform without additional modifications, e.g., a Java WAR file or a database schema. Provisioning a virtual machine and installing Java individually would then be an overhead. However, for availability requirements it can be beneficial to control the scaling configuration, e.g., to fine-tune scaling rules. This also applies to products such as databases or message queues for hosting database schemas and topics, where the underlying infrastructure resources can be configured manually.

Solution. Host software components on a provider-defined deployment stack which is compatible with the given deployment artifact type and enables cloud consumers to manage the scaling configuration as shown in Figure 5. For example, PaaS offerings allow hosting compatible deployment artifacts on provider-defined deployment stacks, e.g., a deployment stack required to run a Java8 application. In such cases, the deployment stack management is reduced to the point of selecting a desired option, whereas the scaling configuration is still consumer-managed, e.g., selecting the amount of virtual machines and managing component's scaling rules. Another example is related to certain Database-as-a-Service offerings such as AWS DocumentDB, which allow allocating infrastructure resources for database-specific deployment artifacts.

Example. Figure 5 shows a ToDo list application from Section 3.3 hosted using AWS Beanstalk and AWS DocumentDB. The desired runtime is selected from provider-defined options, i.e., the compatible Java runtime and a MongoDB-compatible document database. The cloud consumer, however, still is responsible for managing the scaling configuration.

Result. After applying this pattern, the deployment stack management is significantly reduced, since consumers can directly select a pre-defined runtime for a given type of deployment artifact. Hence, this deployment variant allows focusing more on the actual component being hosted instead of managing the underlying deployment stack. However, due to retained control over scaling configuration, consumers are still able to influence the extent to which resources are used, and the scaling rules can typically be defined without relying on the provider.

Known Uses. Multiple PaaS offerings enable implementing this pattern, e.g., AWS Elastic Beanstalk Service (Amazon Web Services, 2021d) and Azure App Service (Microsoft, 2021a). AWS Beanstalk enables hosting software components developed for various platforms using pre-defined deployment stacks. Infrastructure resources required for running to-be-deployed components must be defined by developers in a form of AWS EC2 virtual machines representing required deployment stacks. Similarly, Azure's App Service allows hosting software components developed for multiple different platforms and control the allocation of underlying infrastructure. Certain product-specific offerings also fit into this category, e.g., Amazon DocumentDB (Amazon Web Services, 2021a) or Oracle's Database Classic Cloud Service (Oracle, 2021) allow managing the resources allocated for using database-specific components.

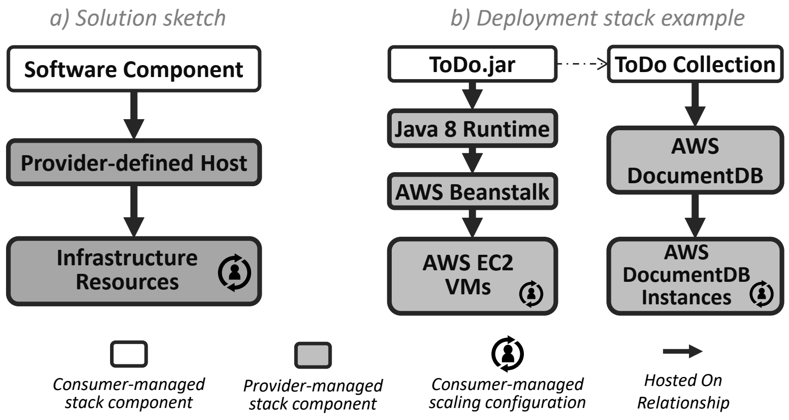

Figure 5: A solution sketch (a) and an example deployment (b) for the *Provider-defined Stack Hosting Pattern*.

3.6 Provider-managed Container Hosting Pattern

Problem: *How to host a software component while managing only the runtime environment it runs on?*

Context. A software component needs to be hosted such that its runtime environment is customizable.
Forces. Managing the entire deployment stack might not be necessary for cases without specific customization requirements. Instead, cloud consumers might want to focus on managing only the runtime environment, e.g., when proprietary libraries are required to run the software component. In such cases, cloud consumers can also benefit from provider-managed scaling configuration, e.g., infrastructure resources management and scaling container instances.
Solution. Host software components on a deployment stack with provider-managed container engines that do not require managing scaling configuration. Figure 6 shows the solution sketch in which a containerized software component is hosted on a provider-managed container engine. Management of runtime environment remains possible via container images, but the scaling configuration is now provider's responsibility. With provider-managed scaling configuration, cloud consumers do not need to manage infrastructure resources based on component's operation mode, e.g., long-running containers versus stateless tasks execution. For example, in some CaaS offerings cloud consumers do not need to allocate infrastructure resources, and providers are responsible for scaling containers in and out.

Example. Figure 6 shows a ToDo list application from Section 3.3 hosted using AWS Fargate, a serverless container-centric offering from Amazon. The desired runtime is defined as a part of container images, i.e., the compatible Java runtime and a MongoDB, with provider-managed scaling configuration. Since defining the runtime in container images for standard products such as databases involves unnecessary overhead, it might be simpler for such components to directly implement the Serverless Hosting Pattern discussed in the next subsection.
Result. When applied, this pattern allows configuring the runtime environment by creating custom container images, while deployment stack management efforts are reduced. In addition, with provider-managed scaling configuration, the infrastructure resources required to run containers are allocated by providers, including the simplified scaling rule definition managed primarily by providers. Components hosted using this pattern can be combined with components hosted using the Serverless Hosting Pattern to implement serverless applications.
Known Uses. Multiple CaaS offerings enable implementing this pattern, e.g., AWS Fargate (Amazon Web Services, 2021g), Azure Container Instances (Microsoft, 2021b), and Google CloudRun (Google, 2021b) allow hosting container images while the deployment stack and scaling configuration is managed by providers, including scaling containers to zero instances for some offerings. Another example is related to more specialized offerings, e.g., Iron Worker (Iron.io, 2021) enables execution of containerized background tasks.

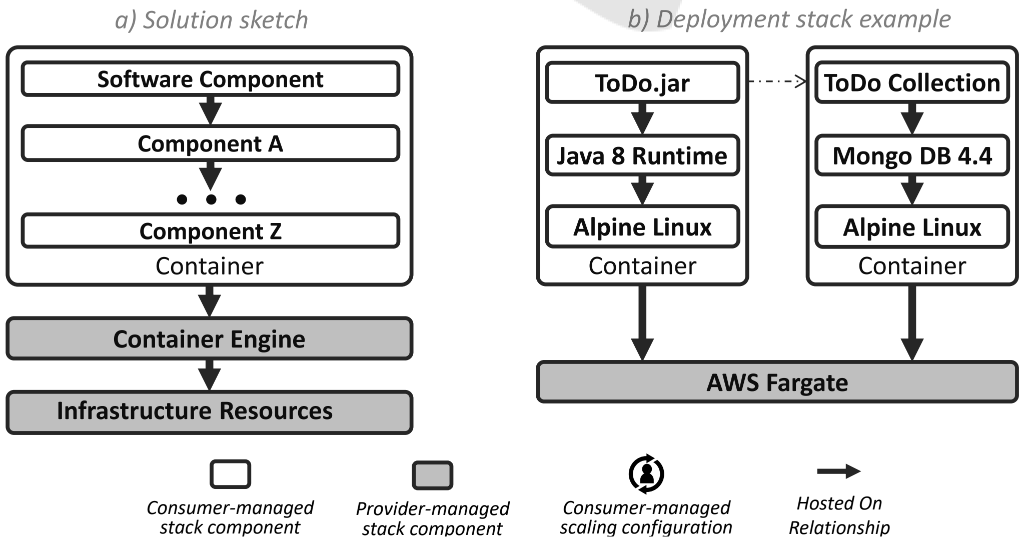

Figure 6: A solution sketch (a) and an example deployment (b) for the *Provider-managed Container Hosting Pattern*.

3.7 Serverless Hosting Pattern

Problem: *How to host a software component without managing its deployment stack or scaling configuration?*

Context. A software component needs to be hosted on a compatible deployment stack without further deployment stack or scaling configuration management.
Forces. In certain cases, it is preferable to simply select a pre-defined runtime environment without managing the deployment stack components and scaling configuration. For instance, defining a runtime and managing the scaling configuration might be an overhead when only a small code snippet needs to be hosted. Tighter coupling with providers can also simplify integration with other provider-specific services.
Solution. Host software components on a provider-defined deployment stack which is compatible with the given deployment artifact type and does not require cloud consumers to manage the scaling configuration. Figure 7 shows the solution sketch in which a software component is hosted on a provider-defined stack which does not require cloud consumers to manage scaling configuration. For example, using public Function-as-a-Service platforms one can host source code snippets and configure them to be invoked when specific cloud events occur. Many products such as databases and message queues are also offered as services that do not require managing underlying infrastructure resources, e.g., serverless DBaaS offerings. Another example is Software-as-a-Service (SaaS) offerings, in which components typically require minor configuration efforts to allow using them, whereas users are not aware which deployment stacks are used in the background. For example, consumers can host static web pages via GitHub Pages by configuring a repository in a specific way.

Example. Figure 7 shows a ToDo list application from Section 3.3 hosted using AWS Lambda and AWS DynamoDB, serverless FaaS and DBaaS offerings from Amazon. The desired runtime is selected from the list of available runtime environments, i.e., the compatible Java runtime and a NoSQL database compatible similar to MongoDB.
Result. When applied, this pattern results in a simplified deployment of a component, where typically only a deployment artifact compatible with the underlying platform and specific configurations need to be provided to start using the component, whereas the underlying platform is responsible for scaling configuration. Even more control is given up in SaaS offerings, where a particular product can be used by providing product-specific configurations. This hosting pattern is the least managed option in terms of deployment stack and scaling configuration management, which allows focusing more on business logic and desired component interactions. Furthermore, this pattern eases the integration with other provider services, e.g., trigger FaaS functions using events from DBaaS.
Known Uses. Multiple FaaS offerings including AWS Lambda (Amazon Web Services, 2021h), Azure Functions (Microsoft, 2021c), and Google Cloud Functions (Google, 2021a) enable implementing this pattern. The hosted code is automatically scaled based on demand, and also can be integrated with events originating from other provider services due to a tighter-coupling with provider's infrastructure. Other examples include object storage services such as AWS S3 (Amazon Web Services, 2021i) or databases such as Amazon Aurora Serverless (Amazon Web Services, 2021b), which do not require consumers to manage the scaling configuration. Instead, providers employ pricing models in which consumers are charged based on how products are used in terms of computational power, storage, data transfer, etc.

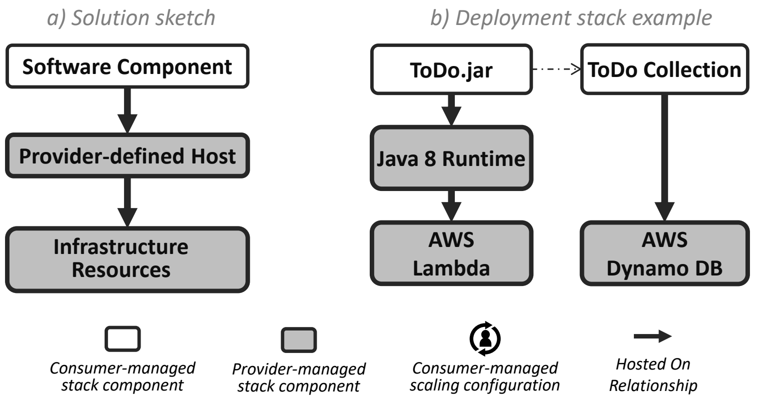

Figure 7: A solution sketch (a) and an example deployment (b) for the *Serverless Hosting Pattern*.

4 ON PATTERNS' VALIDITY

Following the approach in (Endres et al., 2017), we identified pattern candidates based on the analysis of documentation of existing hosting solutions. In particular, we considered existing application hosting approaches and technologies, their documentations, implementations of the most downloaded artifacts in their official repositories, and scientific publications as a basis for our knowledge collection. Based on the found commonalities, we elaborated pattern candidates, which we iteratively refined to obtain the proposed deployment patterns, as prescribed by the guidelines in (Fehling et al., 2014b).

We also validated the proposed patterns based on the *Rule of Three* (Coplien, 1996), which states that a pattern is valid if we can find three independent occurrences of such pattern in the real world. Table 1 shows that this is the case for the proposed hosting patterns, by recapping the known uses of such patterns already discussed in Section 3.

Table 1: Real-world occurrences of the proposed patterns.

Hosting Pattern	**Real-world Occurrences**
Serverful Hosting	IBM Cloud Bare Metal Servers, AWS EC2, Azure IaaS, Google Compute Engine
Consumer-managed Container Hosting	IBM Cloud Kubernetes Service, Azure kubernetes Service, AWS Elastic Kubernetes Service, AWS Elastic Container Service (EC2-based pricing)
Provider-defined Stack Hosting	AWS Beanstalk, Azure App Service, Amazon DocumentDB, Oracle's Database Classic Cloud Service
Provider-managed Container Hosting	AWS Fargate, Azure Container Instances, Google CloudRun, Iron Worker
Serverless Hosting	AWS Lambda, Azure Functions, Google Cloud Functions, IBM Cloud Functions, AWS S3, Amazon Aurora Serverless

5 DISCUSSION

Following the discussion in Section 3.1, one could argue that the hosting patterns in which providers manage at least one of the discussed dimensions (Sections 3.5 and 3.6) can also be seen as serverless-style hosting options. Firstly, services allowing to implement the *Provider-managed Container Hosting Pattern* are indeed often called serverless by cloud providers. For example, AWS Fargate or Google CloudRun differ from more consumer-

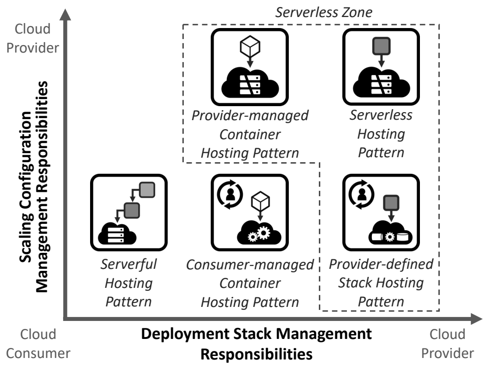
Serverless Zone
Cloud Provider
Scaling Configuration Management Responsibilities
Provider-managed Container Hosting Pattern
Serverless Hosting Pattern
Serverful Hosting Pattern
Consumer-managed Container Hosting Pattern
Provider-defined Stack Hosting Pattern
Cloud Consumer
Deployment Stack Management Responsibilities
Cloud Provider

Figure 8: A spectrum of component hosting patterns and the notion of "serverless zone" encompassing several patterns.

managed container-centric services as they do not require allocating the underlying infrastructure resources, hence resembling the *Serverless Hosting Pattern* in this aspect. However, from the perspective of the deployment stack management, this pattern is different from the Serverless Hosting Pattern: it offers more freedom to cloud consumers, e.g., FaaS or certain DBaaS offerings do not really allow modifying the deployment stack[1]. Secondly, the *Provider-defined Stack Hosting Pattern* enables choosing a deployment stack pre-defined by provider (a specific Java runtime in a PaaS offering, a database of specific version using DBaaS, etc.), thus, also resembling the *Serverless Hosting Pattern*. However, this pattern differs in freedom of the underlying scaling configuration management. Moreover, cloud service models enabling to implement it, e.g., PaaS, are less often associated with the term serverless in the grey literature and by practitioners (Leitner et al., 2019).

These overlaps among patterns make it not entirely fair to say that a serverless-style hosting can only be implemented using the Serverless Hosting Pattern, essentially raising a question of what the term serverless actually encompasses. Considering how fast cloud services evolve and how often cloud providers tend to overlap features in different offerings, introducing a strict definition of "serverless-ness" seems non-trivial and rather restrictive. Instead, we think that the idea of "serverless-ness" can be perceived in a fuzzier sense, i.e., the hosting pattern can be considered serverless when at least *certain management requirements* are met. Figure 8 shows the patterns' spectrum discussed in Section 3.1 with the so-called "serverless zone" in it, which encompasses the patterns with the highest degree of management in at least one of the dimensions. The idea

[1]Certain FaaS platforms support container images as deployment artifacts, which, essentially, also allows implementing the *Provider-managed Container Hosting Pattern*.

of such serverless zone could enable having several distinct patterns that vary in degree of management related to specific aspects still perceived as serverless-style hosting, which in its turn allows flexibly selecting component hosting options as well as verifying whether a given application's topology conforms to the serverless architectural style.

Another interesting point worth emphasizing is how the introduced hosting patterns can be composed with patterns from other pattern languages, e.g., microservice patterns or enterprise integration patterns. For example, a well-known API Gateway pattern (Richardson, 2018) can be implemented differently based on which hosting pattern is chosen. As a result, when deciding on application's components one can think about a composition of patterns, e.g., an API Gateway combined with the Serverful Hosting pattern can be implemented by installing Kong Gateway on a Linux-based VM, whereas the API Gateway combined with the Serverless Hosting pattern can be implemented by simply using AWS API Gateway service. Similar composition reasoning can be applied when deciding on other pattern compositions too, e.g., implementation of a Message Filter pattern (Hohpe and Woolf, 2004) can be different depending on which hosting pattern is chosen. However, when speaking about concrete solutions, the more provider-managed a hosting pattern is, the more it is coupled with available provider offerings, essentially, raising a need for having a knowledge base defining concrete solutions for hosting patterns (and pattern compositions) to facilitate the overall decision-making process. For example, the Provider-managed Container Hosting Pattern is mainly related to provider-managed container services such as AWS Fargate, making it less pervasive than other patterns, at least in the context of currently-available cloud offerings.

6 RELATED WORK

Cloud-related patterns have been widely studied. Fehling et al. (Fehling et al., 2014a) introduce cloud computing patterns supporting developers in building cloud-native applications, viz., applications built for running in cloud and structured to fully exploit its potentials. Other examples in this direction (Erl et al., 2015; Pahl et al., 2018; Davis, 2019) also provide patterns for structuring cloud-native applications. Hohpe and Woolf (Hohpe and Woolf, 2004) introduce enterprise integration patterns (EIPs), which are also used to structure cloud-native applications (Yussupov et al., 2020). The above works differ from ours as they propose patterns for *designing* cloud applications, whereas our patterns focus on their *hosting*.

Similar considerations apply to patterns introduced in the domain of serverless computing. Taibi et al. (Taibi et al., 2020) and Zambrano (Zambrano, 2018) elicit patterns for architecting serverless applications. Hong et al. (Hong et al., 2018) present patterns for improve the security of cloud-hosted services, based on serverless computing. Hence, these works also focus on *designing* cloud applications, whereas we focus on *how to host* them.

Jamshidi et al. (Jamshidi et al., 2015) instead propose a catalog of patterns for migrating on-premise applications to the cloud. The catalog of migration patterns is refined in the subsequent work (Jamshidi et al., 2017), which also proposes a concrete method for enacting pattern-based migration of on-premise application to the cloud. These patterns (Jamshidi et al., 2015; Jamshidi et al., 2017) differ from our hosting patterns, as they focus on adapting existing service-based applications to allow *migrating* them to the cloud, rather than on how to actually *deploy* such applications. They can, however, be used in conjunction with the hosting patterns we propose: developers may first exploit migration patterns to enable deploying applications in the cloud, and then implement our patterns for hosting the application components.

To the best of our knowledge, the only existing work organizing knowledge on application deployment into patterns focuses on deployment model types. Endres et al. (Endres et al., 2017) distinguish two possible approaches for specifying the deployment of an application, viz., declarative vs. imperative application deployment. Intuitively, the declarative deployment consists in specifying the desired state for an application, by relying on a deployment engine to automatically determining and enacting the sequence of operations for enforcing such state. The imperative deployment instead consists in explicitly specifying the sequence of operations to enact for deploying an applications. These patterns are, hence, complementary to our hosting patterns: developers can use our patterns to *derive* the required deployment steps for a given component, whose actual *specification* can then be either declarative or imperative, depending on the developers' needs.

Another important aspect related to patterns is how to find suitable patterns and traverse to related solutions in different interconnected pattern languages. Falkenthal et al. (Falkenthal and Leymann, 2017) introduce the concept of solution languages which facilitate the navigation from the level of patterns to concrete solutions, e.g., implementation of a pattern using specific technologies. Leymann et al. (Leymann and Barzen, 2020) propose an approach and tool for navigating through different pattern languages inspired by the analogy with cartography. Such approaches can help linking the hosting patterns introduced in this work with other pattern languages such as cloud computing patterns (Fehling et al., 2014a), and support the search for concrete solutions.

Finally, it is worth relating our hosting patterns to the existing works on pattern-based application deployment. Harzenetter et al. (Harzenetter et al., 2020) provide a model-based solution for deploying applications, whose deployment specification exploit cloud patterns to specify application/infrastructure components in a vendor-agnostic manner. Yussupov et al. (Yussupov et al., 2020) use EIPs to generically specify the integration of application service. In both cases, abstract patterns are then automatically replaced by concrete components when the deployment of an application is actually enacted. Solutions like (Harzenetter et al., 2020) and (Yussupov et al., 2020) can hence benefit from the patterns we propose in this paper. Indeed, such solutions could extend the set of supported patterns by also including our hosting patterns, hence making application deployment specifications even more generic.

7 CONCLUSIONS

In this work, we introduced five hosting patterns and the corresponding pattern primitives defining a common vocabulary for formulating the patterns. The introduced hosting patterns aim to facilitate the decision making process for choosing the most suitable hosting variant for a given software component. We also positioned the presented patterns in a so-called application components hosting spectrum, showing how they represent different combinations of management responsibilities related to deployment stack and scaling configuration management. Finally, we validated the proposed patterns by showing how they fulfill the *Rule of Three* (Coplien, 1996), viz., by presenting at least three real-world occurrences of each pattern.

For future work, we plan to devise a decision support system helping application administrators in choosing the hosting solution most suited to their needs. We plan to investigate which are the well-known advantages and disadvantages of each hosting pattern, or when it is better to adopt one pattern, e.g., by means of a multivocal review. We could then exploit the obtained knowledge to start devising a decision support system, which could also be enhanced by exploiting other existing systems to further refining decisions, e.g., PaaSfinder (Kolb, 2019) or FaaStener (Yussupov et al., 2021), if a cloud consumer needs to decide which PaaS or FaaS platform to use to deploy a given application. Another interesting research direction is to analyze how our hosting patterns can be extended based on other dimensions such as application health management. Furthermore, we intend to define more precise links between the hosting patterns and existing pattern languages such as cloud computing patterns (Fehling et al., 2014a) to simplify finding composite solutions using existing approaches such as Pattern Atlas (Leymann and Barzen, 2020) and Pattern Views (Weigold et al., 2020).

ACKNOWLEDGMENTS

This work is partially funded by the European Union's Horizon 2020 research and innovation project *RADON* (825040). We would also like to thank the anonymous referees, whose feedback helped improving this paper.

REFERENCES

Alexander, C. (1977). *A pattern language: towns, buildings, construction*. Oxford university press.

Amazon Web Services (2021a). Amazon DocumentDB. https://aws.amazon.com/documentdb. [Online].

Amazon Web Services (2021b). AWS Aurora. https://aws.amazon.com/rds/aurora/serverless. [Online].

Amazon Web Services (2021c). AWS EC2. https://aws.amazon.com/ec2. [Online].

Amazon Web Services (2021d). AWS Elastic Beanstalk. https://aws.amazon.com/elasticbeanstalk. [Online].

Amazon Web Services (2021e). AWS Elastic Container Service. https://aws.amazon.com/ecs. [Online].

Amazon Web Services (2021f). AWS Elastic Kubernetes Service. https://aws.amazon.com/eks. [Online].

Amazon Web Services (2021g). AWS Fargate. https://aws.amazon.com/fargate. [Online].

Amazon Web Services (2021h). AWS Lambda. https://aws.amazon.com/lambda. [Online].

Amazon Web Services (2021i). AWS S3. https://aws.amazon.com/s3. [Online].

Baldini, I. et al. (2017). Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer.

Buschmann, F. et al. (2007). *Pattern-Oriented Software Architecture: On Patterns And Pattern Language*, volume 5. John wiley & sons.

Coplien, J. O. (1996). Software patterns.

Councill, B. and Heineman, G. T. (2001). Definition of a software component and its elements. *Component-based software engineering: putting the pieces together*, pages 5–19.

Davis, C. (2019). *Cloud Native Patterns: Designing change-tolerant software*. Manning Publications.

Endres, C. et al. (2017). Declarative vs. imperative: Two modeling patterns for the automated deployment of applications. In *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*, pages 22–27. Xpert Publishing Services (XPS).

Erl, T. et al. (2015). *Cloud Computing Design Patterns*. Prentice Hall Press, 1st edition.

Falkenthal, M. and Leymann, F. (2017). Easing pattern application by means of solution languages. In *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*, pages 58–64.

Fehling, C. et al. (2014a). *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer.

Fehling, C. et al. (2014b). A process for pattern identification, authoring, and application. In *Proceedings of the 19th European Conference on Pattern Languages of Programs*, pages 1–9.

Gamma, E. et al. (1995). Elements of reusable object-oriented software. *Reading: Addison-Wesley*.

Google (2021a). Google Cloud Functions. https://cloud.google.com/functions. [Online].

Google (2021b). Google CloudRun. https://cloud.google.com/run. [Online].

Google (2021c). Google Compute Engine. https://cloud.google.com/compute. [Online].

Harzenetter, L. et al. (2020). Pattern-based Deployment Models Revisited: Automated Pattern-driven Deployment Configuration. In *Proceedings of PATTERNS 2020*, pages 40–49. Xpert Publishing Services.

Hohpe, G. and Woolf, B. (2004). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional.

Hong, S. et al. (2018). Go serverless: Securing cloud via serverless design patterns. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*.

IBM (2021a). IBM Cloud Bare Metal Servers. https://www.ibm.com/cloud/bare-metal-servers. [Online].

IBM (2021b). IBM Cloud Kubernetes Service. https://www.ibm.com/cloud/container-service/resources. [Online].

Iron.io (2021). Iron Worker. www.iron.io/worker. [Online].

Jamshidi, P. et al. (2015). Cloud migration patterns: A multi-cloud service architecture perspective. In *Service-Oriented Computing - ICSOC 2014 Workshops*, pages 6–19, Cham. Springer International Publishing.

Jamshidi, P. et al. (2017). Pattern-based multi-cloud architecture migration. *Software: Practice and Experience*, 47(9):1159–1184.

Jonas, E. et al. (2019). Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*.

Kolb, S. (2019). *On the Portability of Applications in Platform as a Service*. PhD thesis, University of Bamberg, Germany.

Lau, K.-K. and Wang, Z. (2007). Software component models. *IEEE Transactions on software engineering*, 33(10):709–724.

Leitner, P. et al. (2019). A mixed-method empirical study of function-as-a-service software development in industrial practice. *Journal of Systems and Software*, 149:340–359.

Leymann, F. and Barzen, J. (2020). Pattern Atlas. *arXiv preprint arXiv:2006.05120*.

Messerschmitt, D. G. (2007). Rethinking components: From hardware and software to systems. *Proceedings of the IEEE*, 95(7):1473–1496.

Microsoft (2021a). Azure App Service. https://azure.microsoft.com/en-us/services/app-service. [Online].

Microsoft (2021b). Azure Container Instances. https://azure.microsoft.com/en-us/services/container-instances. [Online].

Microsoft (2021c). Azure Functions. https://azure.microsoft.com/en-us/services/functions. [Online].

Microsoft (2021d). Azure IaaS. https://azure.microsoft.com/en-us/overview/what-is-azure/iaas. [Online].

Microsoft (2021e). Azure Kubernetes Service. https://azure.microsoft.com/en-us/services/kubernetes-service. [Online].

OASIS (2015). *TOSCA Simple Profile in YAML Version 1.0*. Organization for the Advancement of Structured Information Standards (OASIS).

Oracle (2021). Oracle Database Classic Cloud Service. docs.oracle.com/en/cloud/paas/database-dbaas-cloud/index.html. [Online].

Pahl, C. et al. (2018). Architectural principles for cloud software. 18(2).

Richardson, C. (2018). *Microservices patterns*. Manning Publications Company,.

Taibi, D. et al. (2020). Patterns for serverless functions (function-as-a-service): A multivocal literature review. In *CLOSER*, pages 181–192.

Weigold, M. et al. (2020). Pattern Views: Concept and Tooling of Interconnected Pattern Languages. In *Proceedings of the 14th Symposium and Summer School on Service-Oriented Computing (SummerSOC 2020)*, pages 86–103.

Wellhausen, T. and Fiesser, A. (2011). How to write a pattern? a rough guide for first-time pattern authors. In *Proceedings of the 16th European Conference on Pattern Languages of Programs*, pages 1–9.

Yussupov, V. et al. (2020). Pattern-based modelling, integration, and deployment of microservice architectures. In *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*, pages 40–50.

Yussupov, V. et al. (2021). FaaSten your decisions: A classification framework and technology review of Function-as-a-Service platforms. *Journal of Systems and Software*, 175.

Zambrano, B. (2018). *Serverless Design Patterns and Best Practices: Build, Secure, and Deploy Enterprise Ready Serverless Applications with AWS to Improve Developer Productivity*. Packt Publishing.