Multi-Tenancy: A Concept Whose Time Has Come and (Almost) Gone

Christoph Bussler
Oracle Corporation, Redwood City, CA 94065, U.S.A.

Keywords: Cloud Software Engineering, Multi-tenancy, Container-based Computing, Server-less Computing.

Abstract: With the emergence of server-less computing the need for multi-tenancy in application services diminishes and eventually disappears as server-less computing supports the isolation between tenants by cloud account automatically. A server-less application installed into a customer's cloud account is isolated from other customer's cloud accounts by means of the underlying cloud provider infrastructure automatically. Aside from perfect partitioning in all aspects, this server-less computing simplifies the implementation of an application service since multi-tenancy does not have to be implemented or managed at all by the application service logic itself. The position brought forward in this paper is that the concept of multi-tenancy for application design and implementation is obsolete in context of application services implemented based on server-less computing.

1 INTRODUCTION

The primary driver behind multi-tenancy is efficient resource utilization in cloud infrastructures. A multi-tenant resource serves many tenants concurrently in order to avoid being underutilized or even idling.

"The term 'software multitenancy' refers to a software architecture in which a single instance of software runs on a server and serves multiple tenants. A tenant is a group of users who share a common access with specific privileges to the software instance. With a multitenant architecture, a software application is designed to provide every tenant a dedicated share of the instance – including its data, configuration, user management, tenant individual functionality and non-functional properties. Multitenancy contrasts with multi-instance architectures, where separate software instances operate on behalf of different tenants." (Wikipedia 2018).

A tenant in this context refers to a customer (a commercial or government organization, or an individual) that has an account in the cloud infrastructure and uses the cloud's services like SaaS (Software as a Service), PaaS (Platform as a Service) and/or IaaS (Infrastructure as a Service).

The cloud offered by a cloud provider like, for example, Amazon, Google, Microsoft or Oracle is implemented as a set of data centers placed in different geographical locations around the globe. A data center at its core consists of a set of physical hardware machines (including storage devices) organized into racks that host the various resources or cloud services.

Resources are cloud services like data analytics, machine learning, enterprise data integration, process integration, but also databases, document stores, block storage, or middleware like message queues. Basically, each cloud service provided by a cloud is a resource.

In general, resources are classified as SaaS, PaaS and IaaS cloud services. A tenant can access one or more of the resources in these classes. For the following discussion, however, the classification is not relevant.

How are (multi-tenant) application services designed and implemented utilizing cloud resources like IaaS or PaaS resources?

1.1 Multi-Tenant Service Implementation Strategy: State-of-the-Art and the New Kid on the Block

In general there are many architectural approaches to implementing multi-tenant application services (like

Bussler, C.
Multi-Tenancy: A Concept Whose Time Has Come and (Almost) Gone.
DOI: 10.5220/0006963303160323
In *Proceedings of the 14th International Conference on Web Information Systems and Technologies (WEBIST 2018)*, pages 316-323
ISBN: 978-989-758-324-7

for example a supply chain management system). This position paper calls out two approaches:

- Container-based computing (state of the art)
- Server-less computing (new kid on the block)

The currently predominantly used container-based computing infrastructure is Kubernetes (Kubernetes, 2018). Kubernetes is a container management system mainly – but not exclusively – used in conjunction with Docker (Docker, 2018). Kubernetes can create clusters consisting of sets of Docker containers (realized as Kubernetes pods). Possibly cooperating Kubernetes pods implement the application service functionality.

Cloud providers support in general two approaches for tenants to use Kubernetes. One is for a tenant to install the Kubernetes runtime software on virtual machines into a cloud. The other is for a tenant to use Kubernetes as a service. The former requires a tenant to install Kubernetes itself on IaaS resources, whereas the latter supports launching a Kubernetes cluster as a PaaS service without having to install the Kubernetes runtime software itself first.

Docker containers are launched from Docker images via the Kubernetes concept of Kubernetes services and Kubernetes pods and are constantly running (except when being recovered in case of failures or deleted when scaling down).

To efficiently utilize these constantly running containers within the Kubernetes pods they in general execute functionality for several tenants concurrently and thereby implement a multi-tenant software architecture.

This container-based approach requires tenant management in order to know which tenants exist, requires the ability to create log statements or log files separated by tenant identifiers, requires an engineering approach that ensures that the execution threads in a container are isolated to avoid cross-tenant contamination, requires the ability to move tenants between clusters for capacity and load adjustments, requires container scaling strategies that take tenant-specific load into consideration – just to name specific multi-tenant functionalities.

The new kid on the block is server-less computing. Server-less computing is an approach that abstracts away the computing and middleware infrastructure (aka, IaaS and PaaS). Server-less computing provides the ability to register/upload code and dependencies without any reference to infrastructure. Example systems are AWS Lambda (AWS Lambda, 2018), Azure Functions (Azure Functions, 2018), Google (Google Cloud Functions, 2018), or Oracle Functions (Oracle Fn, 2018).

In such an approach the application service logic is implemented and registered as functions that possibly trigger other functions or events, and they themselves are triggered by invocations or events. For example, a function can enqueue a message into a queue and that message when dequeued can trigger another function (implementing an event-based pattern).

Implementing an application service is therefore (on a high level) a set of functions and triggers interacting with various cloud components that do not require installation or management of Kubernetes services or Docker containers. Installing an application service means only to upload the functions and triggers of functions.

The server-less approach does not create and run containers (that are potentially underutilized or idling) that a software engineer has to try to optimize and a customer has to pay for if used or not. To the contrary, the tenant only pays for function executions and not for the infrastructure use as the underlying cloud execution environment is independent of the tenant's execution needs.

Every tenant in a cloud can install the required application services by uploading the functions and triggers that make up the service. Every tenant executes these in the context of their cloud account and that is by default isolated from other cloud accounts of other tenants (complete partitioning).

The implementation of a function or trigger does not have to be aware of multi-tenancy: it can safely assume that is being executed for exactly one tenant only based on the separation of cloud accounts. The server-less cloud environment ensures partitioning.

1.2 Context: 3rd Party Software Provider

The discussion in this paper is from the viewpoint of a 3rd party software provider (for example, a start-up, an software vendor, a consulting company or a customer building application services itself) that builds software, like a supply chain management software, for its customers. This software is going to be deployed into the cloud of a cloud provider (and thereby becoming an application service). Once deployed, customers can licence the service from the 3rd party software provider and use it.

Note, a customer of a 3rd party software provider might or might not be a cloud tenant. If the 3rd party software provider implements a service in the cloud, it is the tenant. If the 3rd party provider asks its customer to install software into a cloud, then the customers become tenants as they interact with the

cloud directly. The 3rd party provider has not role during runtime execution.

For emphasis, the discussion in this position paper is not (!) from the viewpoint of a cloud provider itself (like Amazon, Google, Microsoft or Oracle) and their internal implementation. This position paper does not discuss how the cloud infrastructures of the various cloud providers are implemented internally. This viewpoint is irrelevant to the discussion since a 3rd party software provider or a tenant must and will use the public cloud interface as provided by the cloud provider in order to install its service into a cloud for its customers.

1.3 Position: Multi-tenancy Is (Almost) a Concept of the past

"AWS Lambda has stamped a big DEPRECATED on containers" (Brazeal, 2018).

Aside from this flashy prediction about the future of container-based technology, the fact that server-less computing does neither require the implementation of multi-tenant logic, nor the configuration and management of low level constructs like Docker images and Kubernetes configurations lets me to take the position that multi-tenancy is a concept of the past.

The uptake of server-less computing is ongoing and expectations are that due to its higher level of abstraction the uptake will continue on a broad basis. A good place to see the uptake activity in industry can be found here: (Medium, 2018).

Server-less computing does not require multi-tenancy logic as server-less computing is by default automatically providing isolation between tenants as well as automatic efficient resource utilization.

Therefore, my position is: multi-tenancy is a concept whose time has come and (almost) gone.

1.4 Outline

The paper discusses relevant cloud service topologies in order to set the context in Section 2. Afterwards multi-tenancy is discussed in more detail in Section 3 and provides a glimpse into its complexity. Section 4 characterizes and summarizes server-less computing in context of multi-tenancy. Section 5 states the position explicitly after the detailed discussion. Section 6 discusses related work and Section 7 concludes.

2 CLOUD SERVICE TOPOLOGIES

Given a cloud, there are several possible topologies that a 3rd party software provider can implement to make application services (application logic) available to its customers, for example a supply chain management service.

2.1 Installed Kubernetes

The following Figure 1 shows the case of installed Kubernetes. Figure 1 (a): A 3rd party software provider can install a Kubernetes cluster (oval) itself in the cloud (rectangle) onto VMs it creates and then place its customers on it (triangles). The 3rd party software provider is the cloud tenant in this case. Figure 1 (b): Alternatively, the 3rd party provider can ask each of its customers to become a tenant in a cloud and each install their own Kubernetes cluster on VMs containing the supply chain management service. In this case there would be one tenant at most in a given Kubernetes cluster. Figure 1 (b) shows two customers.

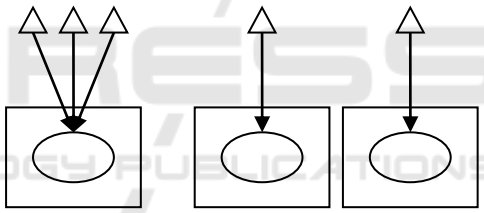

Figure 1: Cloud Service Topologies (Kubernetes).

2.2 Kubernetes-as-a-Service

In the case of Kubernetes-as-a-Service (CaaS, 2018) the Kubernetes functionality is provided by the cloud directly itself as a service (and therefore does not require the installation by tenants). It is very similar to the case of installed Kubernetes. The only difference is that the 3rd party software provider (or the tenants) do not have to create VMs in order to install Kubernetes, but instead ask the cloud to create a Kubernetes cluster without any installation taking place. This cluster is then created by the cloud and made available. The topologies would look like the same as those in Figure 1.

2.3 Server-Less Computing

The 3rd party software provider can also choose to

use a server-less computing approach instead of a container-based approach. In this case the 3rd party software provider has to implement functions and function triggers. Like in case of Kubernetes there are two alternative approaches, outlined in Figure 2.

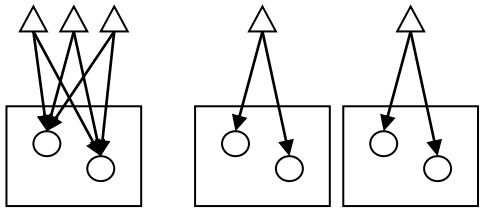

Figure 2: Cloud Service Topologies (Functions).

In case of Figure 2 (a) the 3rd party software provider is a tenant and it creates functions in its cloud account. It then makes those functions accessible to its customers. An example of this case is discussed in (Becker, 2018). In case of Figure 2 (b) the 3rd party software provider asks each of its customers to become a tenant and install the functions for itself. Figure 2 (b) shows two customer accounts.

2.4 Cloud Service Topology Analysis

All 6 previously discussed topologies variations (Installed Kubernetes, Kubernetes-as-a-Service, Server-less Functions, each either 3rd party software provider or customer provisioned) are possible options for a 3rd party software provider to make a service available to its customers. In some cases the 3rd party software provider is a cloud tenant (and the customers are not; the cases labelled (a)), in some cases the customers are tenants themselves (the cases labelled (b)).

The topologies have a major difference that is relevant to this position paper: the cases labelled (a) are those where the software itself has to be able to serve many requests from different customers concurrently (shared access). As a consequence, it has to be a multi-tenant implementation. Furthermore, the 3rd party software provider has to implement management functionality like customer account creation, on-boarding, off-boarding as customers are added or removed, or billing. This is necessary as the cloud infrastructure is not aware of the fact that the 3rd party software provider supports several clients concurrently itself.

The cases labelled (b) are very different. The software itself does not have to be multi-tenant as only one tenant uses it (non-shared access). This makes its implementation simpler. Furthermore, the customers are cloud tenants themselves and therefore the cloud account creation, on-boarding, off-boarding as well as billing and other management functions are those of the cloud itself. The 3rd party provider does not have to implement those. Especially from a customer viewpoint, the cost and the billing are transparent, meaning, the customer sees their resource consumption as it actually takes place in the cloud.

The following two sections look at the cases in (a) and (b) in more detail separated by the container-based approach and the server-less computing approach.

3 MULTI-TENANCY

3.1 Conceptual Overview

Multi-tenancy has been introduced as a concept to accomplish efficient resource utilization. Instead of installing an application system for each client separately and having the application system's resources underutilized or idling when the client is not busy or not using the application system at all, the application system is made available to several clients concurrently. If more than one client can concurrently access the application system at the same time, then the application system has to be implemented accordingly. This functionality is called multi-tenancy.

In principle the executing code has to be aware of the client it is executing the logic for so that it only accesses data for that client (for example). Data access and management has to be partitioned in the sense the clients do not see each other's data at all whatsoever.

Code has to be designed and engineered so that the execution is partitioned as well (re-entrant code). If the execution is not partitioned, then cross-tenant contamination (state of different tenants visible to each other) might happen.

Partitioning might have to be achieved in all aspects depending on a client's requirements. This might include network partitioning, file system partitioning, log file content partitioning, and so on, aside from data partitioning.

3.2 Container-based Computing

Container-based computing is the current state of the art and Kubernetes is the predominant container

management software. Containers are often Docker containers and their management (start, stop, load balance, scale, etc.) is done using Kubernetes.

The discussion in Section 3.1 applies in this context as Kubernetes does not provide any multi-tenancy support out of the box by itself. All multi-tenancy functionality has to be provided by Docker images (and therefore containers at runtime).

The following characterizes the scope of multi-tenant functionality that has to be implemented in context of container-based computing. The approach taken is to follow the life cycle of on-boarding a customer until its off-boarding and highlight some of the interesting points.

- On-boarding a tenant means to register it with a Kubernetes cluster (for example as an entry in a cluster-local database) and setting up tenant-specific resources, like for example a log file directory or a database schema in an existing database.
- Not all resources can be shared, especially those that do not provide inherent partitioning (aka, not supporting multi-tenancy). For example, a file system does not provide the notion of multi-tenancy and so a directory needs to be created for each tenant (one way of providing partitioning externally). The basic principle is that single-tenant resources have to be created as dedicated resources for each tenant separately.
- Code that is running as containers and serving tenants concurrently needs to implement partitioning (isolation between tenants), or at least being re-entrant with assurance that the invocation stack is 100% free of overlap. Not data from different tenants must ever be shared amongst different tenants.
- Logging has its own challenges as each log statement is written in context of a tenant. This means that either logs are separated by tenants in e.g. their own directories; or each log statement contains the tenant id for which the log was written so that the log query system can guarantee partitioning.
- In a multi-tenant system where not all resources are dedicated, the noisy neighbour problem exists. For example, a tenant using a lot of processing or storage space might cause resource shortage for other tenants. This must be monitored, and resources being added to mitigate noisy neighbour situations (a tenant using resources disproportionally high). In addition, it might be necessary to limit resource utilization on a per tenant basis to limit the noisy neighbour problem.
- Tenants might change some of their logic and want a backup of only their data so that a failure in their new logic can be undone by a state restoration. This requires the knowledge of where the tenant's specific state is, how to back it up, and how to restore it.
- Off-boarding of a tenant means to e.g. backup their last state onto long-term storage (optional) and removing all data of that tenant from the container system. Each component needs to understand where it stores the tenant data and how to remove it.

The initial size of a Kubernetes cluster has to be chosen depending on the expectation of the tenant on-boarding rate and the individual resource requirements of the tenants. This requires resource capacity planning and resource capacity management at run-time.

As an intermediate summary, the amount of functionality that needs to be implemented is staggering in order to provide multi-tenancy in addition to implementing the business functionality like a supply chain management system.

This effort has been recognized by many and there are efforts underway to formalize and to implement multi-tenancy in context of Kubernetes (Franzelle, 2018).

3.3 Cost

Multi-tenancy is about infrastructure cost in the sense of reducing cost by means of efficient resource utilization. In an initial system, resources are most likely under-utilized as capacity is available to on-board tenants. As tenants are added, resource utilization improves. As soon a resource is fully (or largely) utilized, new resources are added and excess capacity might exist for a while until additional tenants are on-boarded, or existing tenants are increasing their load on the system. Effectively, there is a resource cost step function and resource utilization varies with every step.

The trade-off is between increased efficiency of resource utilization and design, engineering and implementation complexity on the one hand, and run-time management on the other (like monitoring, ensuring proper scaling, etc.).

Each customer or 3rd party software provider has to determine if this trade-off actually works in the

sense that multi-tenancy has a significantly large gain compared to the engineering and management costs of implementing the multi-tenant functionality.

3.4 Explicit Multi-Tenancy

The situation described in the previous sections can be characterized as "explicit" multi-tenancy as the application service software has to be implemented specifically to provide multi-tenant semantics.

There are operations and procedures required that would not be needed in a single tenancy system. For example, creating a schema per tenant, or a directory per tenant for holding log files.

4 SERVER-LESS COMPUTING

4.1 Conceptual Overview

Server-less computing has been available for a while, and recently receives increased interest, e.g., (Medium, 2018). Server-less computing has several major aspects:

- **No system resources.** It removes the need to create and to manage system resources. It is not necessary to create containers, VMs, file systems, etc., in order to run server-less application service code. Basically, the server-less computing interface supports the upload of code that then can be executed.
- **No management of resources.** During execution it is unnecessary (and impossible) to directly monitor and manage cloud infrastructure resources. Instead, cloud service levels are specified declaratively, like the maximum memory usage.
- **No explicit multi-tenancy.** The code implementing the business logic (like a supply chain management software) has to focus only on the business logic, and does not have to implement multi-tenancy functionality. Of course, it would be possible to go that route as outlined in Figure 2 (a). However, this is not needed since in context of server-less computing the tenants are partitioned in their own cloud account.
- **Implicit partitioning.** Tenants in a cloud are partitioned by the very fact that they are tenants. If a tenant uploads functions and executes those, partitioning is ensured. The implicit partitioning prevents by means of the cloud infrastructure tenant interference.
- **Resource use on-demand**. In case of server-less computing the tenant is guaranteed to only pay for the resources it is using. If a tenant does not use any resource, it will not get billed for it. In that sense there are no idling resources, and the code that is being uploaded does not have to worry about efficient resource utilization. The allocation of execution to resources is done by the cloud infrastructure, not by the application service code.
- **Direct billing.** Since tenants are directly interacting as tenants with the cloud environment, the cloud bills them directly and transparently.

Above the concept of functions is referenced. Functions are one type of resource that server-less computing provides. However, there are additional resources like database tables (or whole databases), queuing systems, notification systems, load balancers, etc. The common thread across all of those is that abstract declarative configuration is uploaded without the requirement of embedding this into the functional code. There is no need or requirement to upload executable images (like container images).

For example, it is possible to configure that an arriving message in a queue triggers a function. This combination and causal execution is configured and not coded in a programming language.

From an architecture viewpoint, an application service is a set of functions and correlated triggers in order to combine functionality. Of course, it can be the case that an application service only consists of functions accessing database tables, or even only functions without any other resource utilization.

4.2 Cost

In case of server-less computing there is a direct relationship between the resources used and the associated cost. Cost only arises for resources actively use, not for resources that are idling. As a consequence efficient resource utilization does not have to be implemented by the 3rd party software provider.

Of course, the cloud implementation itself might have to do resource utilization optimization, however, this is not part of the application service code and invisible to it.

4.3 Implicit Multi-tenancy

In context of server-less computing multi-tenancy is provided by the cloud infrastructure environment that compartmentalizes the tenants by means of the cloud infrastructure and cloud accounts.

From an application software development and management perspective, it comes for free as long as tenants deploy the server-less code into their tenant cloud account.

Since multi-tenancy does not have to be implemented by the application service code, it can be called "implicit" multi-tenancy.

5 POSITION: THE CONCEPT OF MULTI-TENANCY IS OBSOLETE

Based on the discussion comparing the container-based resource implementation with the server-less implementation, there is no question why server-less computing is receiving the attention that it does currently: the effort to build and to manage business logic is significantly reduced.

The position of this becomes clear on that basis. If code is implemented on a server-less computing infrastructure multi-tenancy becomes a non-requirement: the multi-tenancy concept is obsolete.

6 RELATED WORK

There is a tremendous amount of work accomplished in context of multi-tenancy, not only from an application service perspective, but also in context of security, networks, and databases, just to name a few of the affected areas (Multi-Tenancy, 2018).

However, in context of this paper, the absence of multi-tenancy is relevant while at the same time accomplishing tenant partitioning. That has not be discussed at all in the past and only recently first online publications appear that start addressing the topic. Academic literature has not addressed this topic at this point in time.

(Spillner, 2017) hints at an implementation strategy of how an infrastructure providing functions might address automatic tenant separation. However, it does not address how to accomplish the separation of functions in context of all other resources that an application service requires, like queues, databases, storage, and so on. It also does not discuss the distinction between the cases (a) and (b) above, aka, if the 3rd party software provider is the tenant or if customers are themselves tenants.

(Kanouse, 2017) makes the interesting observation that in AWS each function is executed in isolation and that this provides the multi-tenancy. However, the article does not discuss the complete set of resources that a function might require. While AWS Lambda executes each function in isolation when invoked for a single tenant, this does not automatically separate queues, notifications, storage, databases, etc. In order to have full stack separation tenants need to deploy the whole application system in their cloud account. It also does not discuss the distinction between the cases (a) and (b) above, aka, if the 3rd party software provider is the tenant or if customers are themselves tenants.

(Roberts, 2018) also indicates that AWS Lambda addresses the separation. However, again, there is no in-depth discussion and realization that AWS Lambda is only one piece of the puzzle and that in addition to functions all the other resources like queues, databases, storage, etc. need to be separated as well in order for the application code to avoid implementing multi-tenancy. It also does not discuss the distinction between the cases (a) and (b) above, aka, if the 3rd party software provider is the tenant or if customers are themselves tenants.

(Golding, 2017), as others, emphasizes that functions can run dedicated to a single tenant when invoked, but fails to discuss the whole invocation chain possibly traversing queueing systems or database systems. It also does not discuss the distinction between the cases (a) and (b) above, aka, if the 3rd party software provider is the tenant or if customers are themselves tenants.

As can be seen from the discussion of related work, little has been discussed in context of server-less computing and not having to implement the multi-tenant functionality by application services when those are properly setup in a tenant account, as outlined in this position paper.

Related work only focuses only on the aspect of functions in context of server-less application service implementation, and not the full set of cloud resources that might be used when functions execute or the tenant management functionality itself required to manage tenants.

7 CONCLUSIONS

In conclusion, the server-less computing design and implementation approach supports a significantly simpler application service design compared to the

container-based implementation approach as server-less computing does not require the creation and the management of resources.

In addition, due to the implicit multi-tenancy in context of server-less computing provided by individual cloud accounts, application service code does not have to implement multi-tenancy concepts.

Given the fact that the simpler approach based on server-less computing requires less engineering effort as well as charges tenants only for resources used, it is safe to assume that server-less computing will be the dominant application service engineering and deployment approach of the future.

As a consequence, multi-tenancy as a concept is not necessary anymore in context of application service development and as a concept its time has come and gone.

ACKNOWLEDGEMENT

I want to thank the reviewers whose review comments and suggestions improved the position paper.

REFERENCES

AWS Lambda, 2018. https://aws.amazon.com/lambda/ (last accessed 6/20/2018)

Azure Functions, 2018. https://azure.microsoft.com/en-us/services/functions/ (last accessed 6/20/2018)

Becker, 2018. https://medium.com/@tarekbecker/serverless-enterprise-grade-multi-tenancy-using-aws-76ff5f4d0a23 (last accessed 6/20/2018)

Brazeal, 2018. https://read.acloud.guru/serverless-is-eating-the-stack-and-people-are-freaking-out-and-they-should-be-431a9e0db482 (last accessed 6/13/2018)

CaaS, 2018. https://kubernetes.io/docs/setup/pick-right-solution/#hosted-solutions (last accessed 7/3/2018)

Docker, 2018. https://www.docker.com/ (last accessed 6/13/2018)

Franzelle, 2018. https://blog.jessfraz.com/post/hard-multi-tenancy-in-kubernetes/ (last accessed 6/19/2018)

Golding, 2017. Tod Golding. Building Serverless SaaS Applications on AWS https://aws.amazon.com/blogs/apn/building-serverless-saas-applications-on-aws/ (last accessed 6/20/2018)

Google Cloud Functions, 2018. https://cloud.google.com/functions/ (last accessed 6/20/2018)

Kanouse, 2017. Adam Kanouse and Craig Booth. How AWS Lambda Changed the Game of Multi-tenancy. https://narrativescience.com/Resources/Resource-Library/Article-Detail-Page/how-aws-lambda-changed-the-game-of-multi-tenancy (last accessed 6/20/2018)

Kubernetes, 2018. https://kubernetes.io/ (last accessed 6/13/2018)

Medium, 2018. https://medium.com/tag/serverless (last accessed 6/15/2018)

Multi-Tenancy, 2018. https://scholar.google.com/scholar?q=multi+tenancy (last accessed 2018)

Oracle Fn, 2018. https://developer.oracle.com/opensource/serverless-with-fn-project (last accessed 6/20/2018)

Roberts, 2018. Mike Roberts. Serverless Architectures. https://www.martinfowler.com/articles/serverless.html (last accessed 6/20/2018)

Spillner, 2017. Joseph Spillner. Snafu: Function-as-a-Service (FaaS) Runtime Design and Implementation. CoRR abs/1703.07562 (2017)

Wikipedia, 2018. https://en.wikipedia.org/wiki/Multitenancy (last accessed 6/13/2018)

DISCLAIMER

The views expressed here are my own and do not necessarily reflect the views of Oracle.