Towards Serverless as Commodity: a case of Knative

Conference Paper · October 2019

DOI: 10.1145/3366623.3368135

3 authors, including:

Nima Kaviani
University of British Columbia - Vancouver
38 PUBLICATIONS **387** CITATIONS

SEE PROFILE

Towards Serverless as Commodity: a case of Knative

Nima Kaviani
nkavian@us.ibm.com
IBM Cloud Labs
San Francisco, California, USA

Dmitriy Kalinin
dkalinin@pivotal.io
Pivotal Labs
San Francisco, California, USA

Michael Maximilien
maxim@us.ibm.com
IBM Cloud Labs
San Jose, California, USA

Abstract

Serverless computing promises to evolve cloud computing architecture from VMs and containers-as-a-service (CaaS) to function-as-a-service (FaaS). This takes away complexities of managing and scaling underlying infrastructure and can result in simpler code, cheaper realization of services, and higher availability. Nonetheless, one of the primary drawbacks customers face when making decision to move their software to a serverless platform is the potential for getting locked-in with a particular provider. This used to be a concern with Platform-as-a-Service (PaaS) offerings too. However with Kubernetes emerging as the industry standard PaaS layer, PaaS is closer to becoming commodity with the Kubernetes API as its common interface. The question is if a similar unification for the API interface layer and runtime contracts can be achieved for serverless. If achieved, this would free up serverless users from their fears of platform lock-in. Our goal in this paper is to extract a minimal common denominator model of execution that can move us closer to a unified serverless platform. As contributors to Knative [13] with in-depth understanding of its internal design, we use Knative as the baseline for this comparison and contrast its API interface and runtime contracts against other prominent serverless platforms to identify commonalities and differences. Influenced by the work in Knative, we also discuss challenges as well as the necessary evolution we expect to see as serverless platforms themselves reach commodity status.

Keywords serverless, cloud, scalability, performance

1 Introduction

Serverless computing has introduced a new paradigm shift in deploying software. In essence, serverless offerings can be seen as managed event-driven systems in the same way that PaaS offerings are managed hosted runtimes with applications; and IaaS offerings are managed hosted virtual machines. On the one hand, the offered flexibility and simplicity of serverless deployment frees engineers from having to manage the infrastructure. Some early testimonies indicate that engineers are able to develop 10x more functions than microservices [7]. On the other hand, the lucrative cost model of paying only for the functional resource usage, makes adoption of serverless economically justifiable.

These benefits have resulted in increasing widespread adoption of serverless in various technology sectors, from event driven conversational agents [5], to Internet of Things (IoT) [25], and even big data processing [26]. Early successes have convinced cloud providers that the demands for serverless cloud offerings is real. Concrete offerings today is AWS's Lambda [4] which has led to offerings such as IBM Cloud Functions [11], and Google Cloud Run [9], among others.

However, one of the biggest fears of deploying software on serverless platforms is the possibilities of getting locked-in with a particular serverless provider. The concern is to the point that Alex Polvi, CEO at CoreOS, refers to it "as one of the worst forms of proprietary lock-in ever seen in humans' history"[1]. The primary reason is that until recently, serverless platforms required deployed applications to use proprietary company-specific technology, as well as tuning, in order for them to operate. For the case of Lambda as an example, deploying a function requires integration with the AWS API Gateway[2], or the usage of an AWS Application Load Balancer (ALB)[3], as well as the necessary scaffolding needed to tie into the AWS eventing system. All of this makes it cumbersome to move to another technology that fails to offer identical services.

This is not the first time that concerns with lock-in have surfaced in the cloud ecosystem. Prior to serverless, similar concerns were raised for PaaS offerings where it was shown that changes in providers can lead to significant migration costs[10, 30]. The solution however, did not happen to be an extra middleware layer with additional abstraction on top of the existing PaaS [8, 17, 33], but instead to have a layer of abstraction that is widely agreed upon in the industry as the ideal set of primitives for everything else to be built on top of. This has been the story of Kubernetes[21]. Kubernetes is an open source PaaS system for managing containers across multiple hosts. It also provides basic mechanisms for deployment, maintenance, and scaling of the resulting services or applications[21]. While there have been other PaaS offerings with similar capabilities to Kubernetes, the unique differentiating factors that have contributed to its massive adoption by the industry are threefold:

1. From a provider's perspective, Kubernetes is open source and IaaS agnostic. This implies that it can be adopted to run anywhere while offering enough flexibility and visibility into its internals.
2. From an operator's perspective, it provides a declarative and consistent model for application deployment and management. This helps with faster ramp-up, troubleshooting, and extending it.

3. From a developer's perspective, it maintains a consistent API model across its providers. This gives the developers the possibility to build against that API while maintaining the flexibility to migrate from one provider to another if and when needed.

Our position is that, for a serverless platform to gain widespread adoption and solve the lock-in problem, it needs to follow a similar model. Learning from the Kubernetes experience, at a higher level of abstraction serverless should offer a common API interface and a compatible runtime contract that enables the extensibility and interchangeability required to avoid lock-in.

2 Knative

In our attempt to envision serverless as commodity, we start by going over the API interface and the runtime contracts in Knative and using it as the baseline to draw comparisons against other serverless platforms. We chose Knative as the base line, first because we have been actively contributing to Knative and have a deep understanding of its design decisions; and second, because it fits two of the three adoption criteria that we enumerated earlier as the success factors in Kubernetes's PaaS dominance.

Similar to Kubernetes, Knative is also open, extensible, and flexible. This has already resulted in wide-spread interest and adoption of Knative by the likes of Google, RedHat, and IBM among other providers. Also, built on top of Kubernetes, Knative lends itself to a declarative configuration model that is easy for the operators to adhere to. There commonalities imply that Knative might be a good candidate to follow a similar success path to that of Kubernetes. By analyzing the Knative API interface and its runtime contract and comparing it to existing serverless platforms, we aim to understand and adjust efforts in moving towards a severless platform that could commoditize serverless as Kubernetes did for container orchestration platforms.

2.1 Knative Serving

The serving component in Knative is the core to its API interface. Native to Kubernetes, the Knative API interface is defined through a collection of Custom Resource Definitions (CRDs), each capturing a subset of functional requirements for a Knative application. The three top-level resources in the Knative API are the *Configuration*, the *Revision*, and the *Route* CRDs.

1. *Configuration* as the name indicates captures configuration information pertaining to the execution of an application instance. This includes reference to a container image for the packaged application code, configuration parameters, container-related resource constraints, environment variables, etc.
2. *Revision* captures a tagged immutable snapshot of the application configuration bound to a specific version of the application.
3. *Route* captures routing information to different revisions of an application, as well as information on traffic splitting and blue/green deployments.

Figure 1 shows a high-level architecture for Knative serving. For brevity, we skip describing the internals of Knative or other intermediate CRDs used to manage applications, and refer the readers to [16] for an in-depth review. At a high level however, we highlight the existence of a pluggable autoscaler component as well as the service mesh components that are responsible for autoscalability and network configuration for deployed applications. In its runtime contract also, Knative has made two critical choices [15]. First, the zero-to-many scaling in the serving component is there to enable workloads with inbound synchronous HTTP traffic. As such other types of workload may not benefit from the same serverless guarantees. Second, the platform makes strong assumptions about considering an application version ready prior to making it available as a deployed service.

2.2 Knative Eventing

The eventing component in Knative delivers on the promise of loose coupling of microservices in serverless. The three primary CRDs that capture the eventing API interface in Knative are the *Event Source*, the *Broker*, and the *Trigger*.

- *Event Source* (or event producer) registers interest in certain types of events, which can be of two types: internal (e.g., task completion) or external (e.g., GitHub events) to Knative.
- *Broker* offers a channel for event sinks (or event consumers) to subscribe to events from specific event sources.
- *Trigger* binds an event consumer service to a channel for particular types of events.

The eventing component offers additional CRDs to enable filtering of events, event sequencing, maintaining a registry of events, etc. However, we do consider them as non-functional and hence not essential to the eventing API interface. More on these CRDs can be found in [14]. It is however worth noting that to ensure interoperability, the evneting component is consistent with CloudEvent specification[6] developed by the Serverless working group of the Cloud Native Computing Foundation.

2.3 Knative Build

The primary goal for Knative Build has been to create runnable artifacts from application code. Built on top of Kubernetes, the build process in Knative involves compiling and packaging of the executable as an OCI compatible image format[27], labeled and pushed to a container registry. To this end, the

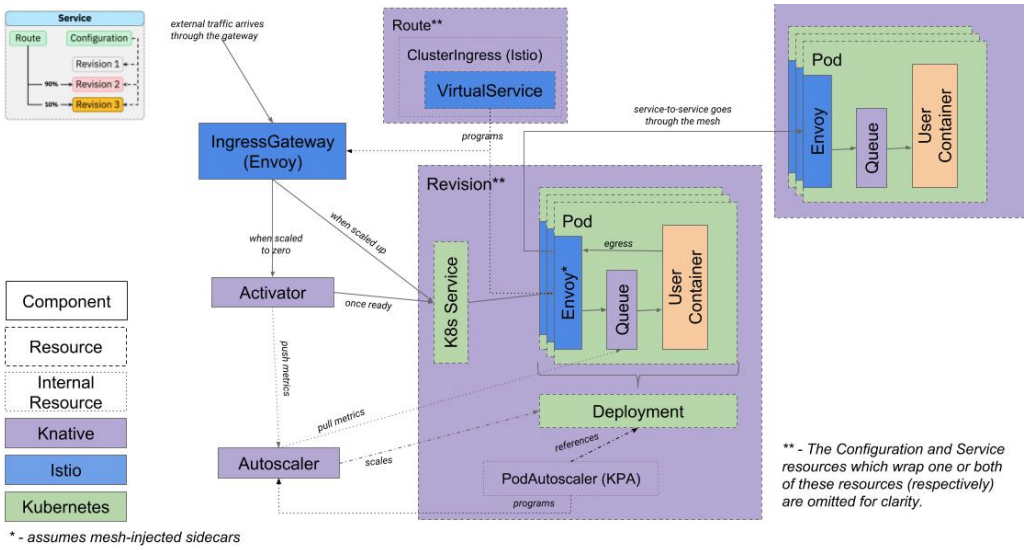

Figure 1. Knative Serving Architecture

build API in Knative entails two primary CRDs, a *BuildTemplate* and a *Build* CRD.

- *BuildTemplate* provides the set of necessary steps for a particular build to happen.
- *Build* utilizes the BuildTemplate with application specific build requirements for the code to be built and the runtime artifact (i.e., docker image) to be created.

While Knative Build initially started as part of the Knative ecosystem, its set of features very quickly grew beyond the minimal requirements of building a docker image as was initially imagined. As a result, the newer versions of Knative deprecated the use of Knative Build, and it spun off into a separate project code-named Tekton [32]. Nonetheless, in our analysis we include the requirement to package applications for serverless platforms to assess its merits as part of a serverless API interface.

3 Comparing Serverless Platforms

In order to arrive to an understanding of a common API layer for serverless, in this section we try to draw a comparison among existing serverless platforms on their API and runtime contracts. Based on the analysis of Knative, as done in Section 2, we have identified seven points of comparison as listed in Table 2. In our efforts to compare serverless platforms on these merits, we compare Knative with the following serverless platforms: AWS Lambda[4], Apache OpenWhisk [11], OpenFaaS [28], and Kubeless [19][1]. Table 2 summarizes our comparison. It is important to mention that in our comparison we have deliberately factored out cost models and cost related comparison points. This is primarily because we consider the charging scheme independent from the actual API implementation or runtime contract for the platform. We also use the words application and function interchangeably to refer to the user provided code artifact that the serverless platform is responsible to execute. Finally it should be noted that our comparison of these serverless platforms is based on data as of August 2019.

3.1 Packaging Contract

Historically platforms embraced different ways to ingest application artifacts. However, recently with wider adoption of micro-services architecture and popularity of platforms such as Kubernetes, container image formats (e.g., OCI) have gained significant traction as a standard. Knative, OpenFaaS, and Kubeless, given its Kubernetes roots, support container image as the only packaging format for their applications. In Knative, explicitly requiring the use of the container image format requires users to deal with packaging, keeping such details hidden away from the platform. OpenFaaS and Kubeless have taken an intermediary path where user functions are taken by the platform and baked into an OCI image to be run by the platform. Alternatively some platforms, such as AWS Lambda, have chosen not to expose such large

1. Packaging Contract
2. Runtime Invocation Contract
3. Application Invocation Contract
4. Execution Model
5. Retry Model
6. Concurrency Model & Autoscaling
7. Traffic Splitting

Table 1. Comparison points for Serverless API interface

[1]Google Cloud Run is not included since it implements Knative APIs.

Platform	Packaging Contract	Runtime Contract	Application Contract	Execution Model	Retry Model	Concurrency Model	Traffic Splitting
Knative	OCI Image	None	HTTP/1 HTTP/2 CloudEvents	Sync	None	Request-based Resource-based	Built-in
Lambda	Custom Packaging	HTTP Service (pull based)	JSON Envelope	Sync / Async	Functional Failures	Request-based	Built-in
OpenWhisk	OCI Image & Custom Packaging	HTTP Service (push based)	JSON Envelope Opt-in HTTP/1	Sync / Async	None	Request-based	External Load Balancing
OpenFaaS	Custom Packaging into OCI Image	HTTP Service (push based)	Stdin / Stdout	Sync / Async	On timeout	Request-based Resource-based	External Service Mesh
Kubeless	Custom Packaging into OCI Image	HTTP Service (push based)	Custom Object	Sync / Async	None	Request-based Resource-based	External Service Mesh

Table 2. Comparision of existing Serverless Platforms.

surface area to the end users and have instead created a custom way to ingest source or compiled code and run it with a predefined set of frameworks/runtimes. Recently, however, Lambda started allowing users to bring in custom runtimes for their applications, ultimately allowing users to take control of how to manage their applications' dependencies. OpenWhisk on the other hand supports both OCI images and plain source code archive as the packaging format.

3.2 Runtime Invocation Contract

We define *runtime invocation contract* as the API boundary between the the platform and the runtime, whether it is platform provided or user provided. We define *runtime* as an engine in charge of executing the application. Knative Serving does not have a runtime to manage applications. It forwards the HTTP stream directly to the application, expecting the application to be capable of dealing with the HTTP protocol. AWS Lambda expects runtimes to call Lambda Runtime API (instead of being called by the platform). This API is composed of "/runtime/init/error", "/runtime/invocations/next" and "/runtime/invocation/$AwsRequestId/response" endpoints (misc. endpoints are ommitted) which are used by the runtime to pull requests, forward them to the application, wait for the application responses, and finally submit the received responses [22]. OpenWhisk follows a push-based mechanism, expecting runtimes to accept requests over HTTP via "/init" and "/run" endpoints. OpenFaaS wraps each application with a Watchdog service. Watchdog [29] is an HTTP service that is responsible for accepting HTTP requests, executing application code in a separate process, passing request payload via stdin and ultimately responding back. Kubeless similar to OpenWhisk requires runtimes to accept requests over HTTP [20].

3.3 Application Invocation Contract

We define *application invocation contract* as the API boundary between the runtime and the application's inputs and outputs. Knative Serving currently supports both HTTP/1 and HTTP/2 protocols and does not place any additional constraints on requests or responses. Given that, applications that use typical HTTP features can run unmodified on Knative. Knative Eventing (which can forward event payloads to Knative Serving services) uses CloudEvents (JSON) envelope on top of HTTP. Unlike Knative, in AWS Lambda applications *exclusively* accept and return payload using Lambda's custom JSON envelope. Lambda ecosystem provides libraries and runtimes to make it easier to work with its custom API. To integrate with larger web ecosystem, AWS also provides an integration via AWS ALB (or API Gateway) to convert plain HTTP requests and responses to Lamba's custom JSON envelope [23]. OpenWhisk also uses custom JSON envelope for request and response payloads; however, it has an opt-in option to pass in raw HTTP stream, bypassing established abstraction. OpenFaaS uses stdin and `stdout` as means to send and receive request and responses from an application [29]. Kubeless has language specific bindings that pass along plain data objects to applications and retrieve HTTP response objects [20].

3.4 Execution Model

Two primary request execution styles in serverless are synchronous (sync) and asynchronous (async). Knative Serving and Kubeless only support the sync execution style. AWS Lambda, OpenWhisk, OpenFaaS however supports both execution styles out of the box. Among them there are subtle differences in how to require the platform to execute application request as async: via HTTP header (AWS Lambda), via query param (OpenWhisk), via path prefix (OpenFaaS). In all platforms, the execution style is invisible to the application.

3.5 Retry Model

None of the platforms offer automatic retries for synchronous executions, and rely on the clients to retry. For asynchronous executions, Knative Eventing, OpenWhisk and Kubeless do not offer builtin retry functionality. AWS Lambda intelligently retries based on a type of error received. Platform errors are not retried (e.g. lack of permission, exceeding quota), and application level errors are. Failed asynchronously invocations will be retried twice. Various calling AWS services also include their own custom retry logic.

OpenFaaS will retry asynchronous invocations that exceed configured `ack_wait` time. This setting is global.

3.6 Concurrency Model & Autoscaling

Knative offers pluggable API to perform application autoscaling (built on lower level Kubernetes APIs such as scale subresource) and by default ships with request based autoscaler. Users may adjust application instance concurrency manually or rely on automatic system detection. Knative also autoscales applications to zero when no requests have been made for a period of time. Given Knative's model for forwarding requests to applications (push-based), accuracy of automatic concurrency detection may suffer. Unlike Knative, AWS Lambda expects application runtimes to consume requests when appropriate (pull-based, as mentioned in 3.2), hence, giving each runtime an opportunity to decide - independently per application instance - how many concurrent requests an application instance can handle. Similar to Knative, AWS Lambda automatically scales up the number of instances on demand, and supports scaling down to zero when application is idle. OpenWhisk, OpenFaaS and Kubeless have a similar model to Knative. OpenFaaS can perform autoscaling based on request rate, CPU and memory consumption. Kubeless similar to Knative uses Kubernetes's HorizontalPodAutoscaler (HPA) resource to control scaling of resources based on CPU, memory, or custom metrics. Given its use of HPA, it does not offer a scale-to-zero functionality. AWS Lambda and OpenWhisk do not support CPU or memory autoscaling.

3.7 Traffic Splitting

Both Knative and AWS Lambda have built-in support splitting percentage of traffic to different previously deployed versions of an application. OpenWhisk, on the other hand has no builtin support for traffic splitting, so it must be handled by deploying multiple applications and the use of an external load balancer. Kubeless and OpenFaaS also do not provide builtin support for traffic splitting; however, given their Kubernetes roots, they easily can interoperate with various service mesh technologies to enable traffic splitting [12, 24].

4 Discussion

Given the comparison above we think ultimate serverless platform API will have following characteristics:

i) Packaging Contract: We believe packaging applications in the form of OCI images helps with unifying the serverless platforms, even though it puts a burden on the application developer. In fact, there are tools (e.g. Kpack[18]) that simplify creating OCI images from application source code. By investing into making this workflow streamlined, serverless platforms that embrace OCI images will offer both convenience for majority of use cases, and flexibility for times when custom application packaging is required.

ii) Runtime Invocation Contract: We believe presence of a runtime and allowing it to consume requests at its own pace (i.e. a pull-based model) helps with managing application instance concurrency (see the discussion on Concurrency Model). Additionally by decoupling fetching of a request from submitting its corresponding response, application is not bound by time or connection healthiness as it would be in case of a synchronous push-based model. For this reason, we advocate for the runtime to implement a pull-based mechanism.

iii) Application Invocation Contract: An ideal serverless platform should standardize on a single envelope for all request and response formats. As mentioned earlier, one of the emerging standards in this field is CloudEvents [6]. There is also a desire to support traditional HTTP based workloads on top of a serverless platform. Custom conversion of HTTP calls to platform specific events or custom data objects is a source of fragmentation. Also modifications required to happen to an application to support custom event formats introduce a barrier to entry. As a compromise we think that runtime should be able to convert incoming CloudEvent into an HTTP request and vice versa. With further integration of an ingress load balancer that can convert plain HTTP request to a CloudEvent, developers will have a standard system for having HTTP workloads on a serverless platform.

iv) Execution Model: An ideal serverless platform should support sync and async operations without putting a burden on application developer to support these execution models differently. Decision power for which execution model to use should be in hands of the application caller. We have made a reference implementation to enable support for asynchronous invocation in Knative and have already discussed it with the community[31].

v) Retry Model: The emergence of service meshes within the Kubernetes ecosystem and their offered features to do retries, monitoring, and metrics collection can be considered as an example of what an ultimate serverless platform should achieve. Particularly retries, metrics, and monitoring should be offered to application developers without having them change their application source code. Currently most serverless platforms provide retry mechanisms only for async execution style; however, we believe support for retries and metrics particularly local to the serverless platform is required as part of the runtime contract in the ultimate serverless platform.

vi) Concurrency Model & Autoscaling: In our opinion the important metric that an ideal serverless platform should be optimizing for is the duration between the time when request is received and the time when the request is started to be processed. To optimize this metric, the serverless platform needs to know when and what application instance is able to process the request. Pull-based model (i.e., runtime/application pulling next invocation request) seems to be more appropriate for such task versus push-based model

(upstream component forwarding invocation request to one of application instances). The intelligence needed to effectively distribute requests to individual application instances can be placed in various components (e.g. load balancer, autoscaling service, or application instance / runtime). We believe that having the intelligent pull-based runtime per application instance is a good architectural pattern as the runtime has access to all necessary data to make decisions for the application instance. This simply leaves the rest of the system only in charge of minimizing request start duration.

vii) Traffic Splitting: It is always possible to implement traffic splitting by having developers manage multiple application versions as separate applications. However, having a simpler, more compact representation of application versions removes that burden from the developers. By explicitly capturing application versions in the serverless platform data model, platform may further help developers in configuration and visualization of traffic splitting.

5 Conclusion

Serverless platforms seem to be the next evolution to managed applications on Clouds. While Kubernetes made possible a common layer creating container-managed cloud application orchestration, and thus full Platform-as-a-Service to be created on top, there is not quite a agreed upon serverless layer to create function-as-a-service. Though we believe one is emerging in Knative.

In this paper we posit that as for container-based platforms, serverless platforms will experience a maturation cycle which will result in a common open source layer as the basis for future offerings in the space. We use Knative as a baseline for this potential common layer and explored its key design decisions and external contracts. After extracting what we believe are the salient interfaces, we compared and contrasted them with leading serverless commercial and OSS offerings. The results show that there are clear areas where strong agreement exists, while in some places these platforms diverge drastically. The differences could of course be consolidated, however, no clear winning strategy can be identified at this point. We believe that it is actually good to have such differences since it will allow innovation with usage and use cases which hopefully result in a winning strategy to evolve.

In a future paper we would like to explore what we believe are canonical use cases for serverless and experiment with the existing platforms and try to extract comparison points on how easy, performant, and extensible each platform is for these use cases. Doing so will better inform and predict how the serverless space will evolve and mature.

References

[1] Alex's Polvi's interview with the Register. 2017-11-06. https://www.theregister.co.uk/2017/11/06/coreos_kubernetes_v_world/.

[2] Amazon API Gateway 2019. https://aws.amazon.com/api-gateway/.

[3] Amazon Application LoadBalancer 2019. https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html.

[4] AWS Lambda 2019. https://aws.amazon.com/lambda.

[5] Building a Serverless Customer Service Bot 2017. https://github.com/awslabs/serverless-chatbots-workshop.

[6] CloudEvents Specification 2019. https://github.com/cloudevents/spec/blob/master/spec.md#design-goals.

[7] Austen Collins. 2017. Building the communication fabric for an event-driven system. https://s3-us-west-2.amazonaws.com/emit-website/2017-slides/building+the+communication+fabric+for+serverless+architectures.pdf.

[8] Paulo Rupino Cunha, Paulo Melo, and Catarina Ferreira Da Silva. 2013. Avoiding Lock-In: Timely Reconfiguration of a Virtual Cloud Platform on Top of Multiple PaaS and IaaS Providers. 970–971. https://doi.org/10.1109/CLOUD.2013.36

[9] Google Cloud Run 2019. https://cloud.google.com/run/.

[10] Mohammad Hajjat, Xin Sun, Yu-Wei Eric Sung, David Maltz, Sanjay G. Rao, Kunwadee Sripanidkulchai, and Mohit Tawarmalani. 2010. Cloudward Bound: Planning for Beneficial Migration of Enterprise Applications to the Cloud. *Computer Communication Review - CCR*, 243–254. https://doi.org/10.1145/2043164.1851212

[11] IBM Cloud Functions 2019. https://console.bluemix.net/openwhisk/.

[12] Istio 2019. https://istio.io/.

[13] Knative 2019. https://knative.dev.

[14] Knative Eventing Documentation 2019. https://knative.dev/docs/eventing/.

[15] Knative Runtime Contract 2019. https://github.com/knative/serving/blob/master/docs/runtime-contract.md.

[16] Knative Serving Documentation 2019. https://knative.dev/docs/serving/.

[17] Stefan Kolb and Guido Wirtz. 2014. Towards Application Portability in Platform as a Service. *Proceedings - IEEE 8th International Symposium on Service Oriented System Engineering, SOSE 2014.* https://doi.org/10.1109/SOSE.2014.26

[18] Kpack 2019. https://content.pivotal.io/blog/introducing-kpack-a-kubernetes-native-container-build-service.

[19] Kubeless 2019. https://github.com/kubeless/kubeless.

[20] Kubeless Runtimes 2019. https://github.com/kubeless/runtimes.

[21] Kubernetes 2019. https://kubernetes.io/.

[22] Lambda Runtime Contract 2019. https://docs.aws.amazon.com/lambda/latest/dg/runtimes-api.html.

[23] Lambda Services 2019. https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html.

[24] Linkerd 2019. https://linkerd.io/.

[25] Bryan Liston. 2017. Implementing a Serverless AWS IoT Backend with AWS Lambda and Amazon DynamoDB. https://goo.gl/d3W3cu.

[26] Sunil Mallya. 2017. Ad Hoc Big Data Processing Made Simple with Serverless MapReduce. https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/.

[27] OCI Image Format 2019. https://github.com/opencontainers/image-spec.

[28] OpenFaaS 2019. https://www.openfaas.com/.

[29] OpenFaaS Watchdog 2019. https://docs.openfaas.com/architecture/watchdog/.

[30] Kewei Sun and Ying Li. 2013. Effort Estimation in Cloud Migration Process. *Proceedings - 2013 IEEE 7th International Symposium on Service-Oriented System Engineering, SOSE 2013*, 84–91. https://doi.org/10.1109/SOSE.2013.29

[31] Support for Async in Knative 2019. https://github.com/knative/serving/issues/4522.

[32] Tekton Pipelines 2019. https://tekton.dev/.

[33] Robail Yasrab and Naijie Gu. 2016. Multi-cloud PaaS Architecture (MCPA): A Solution to Cloud Lock-In. 473–477. https://doi.org/10.1109/ICISCE.2016.108