K-Native Serverless Platform

Suchitra Reddy Chinnamail
Department of Computer Science
California State University
Long Beach
suchitrareddy.chinnamail@student.csulb.edu
CSULB ID: 026627287

Varun Lingabathini
Department of Computer Science
California State University
Long Beach
varun.lingabathini@student.csulb.edu
CSULB ID: 026626468

***Abstract*—Serverless platforms have the ability to transform cloud based structure to "function-as-a-service (FaaS)" from virtual machines and "container-as-a-service (CaaS)". Besides managing and handling complex infrastructure, they are also capable of running simple codes with on scale insight of service and higher accessibility. However, there are always some drawbacks that most of the users face while implementing cloud based serverless platforms. It includes vendors lock-in, throttling, cloud security, cold start card, customers roll outs, CI/CD set up and many others. These issues probably occur with "Platform-as-a-Service (PaaS)" offers also. In order to avoid issues such as vendor lock in and computational limitations of public cloud platforms, open source serverless platform is introduced to take along the control of serverless computing to on-premises arrangements. In this paper, contributions of Knative serverless platform have been discussed to recognize and detect these challenges along with the possible solutions for the evolution in Knative serverless platform.**

Keywords—Knative, Serverless framework/platform, Kubernetes, PaaS, CaaS.

I. INTRODUCTION

Serverless framework has revolutionized the method of software deployment and implementation. The contributions of serverless platforms are considered as organized "event driven systems" just like in a same manner as PaaS based platforms are considered to manage and host runtime applications as well as IaaS offers are organized that are hosted by "virtual machines (VMs)". It has been observed that implementation of serverless platform provides more simplicity and flexibility, therefore, engineers are not restricted to manage the arrangements of the infrastructure now. But on the other hand, it has also been stated that engineers can develop 10x more functions as compared to the micro services [1]. But, the adoption of serverless framework [2] becomes economically justifiable when it comes to the productive cost model to pay only for "functional resource usage".

The increasing advantages of serverless platforms have given rise to increasing widespread agreement of serverless framework in several technologies based sectors, such as "event driven conversational agents" [3], "Internet of Things (IoT)" [4], along with "big data processing" [5]. The achievement and demand of deploying serverless cloud platforms have swayed cloud system suppliers that the demand of serverless based cloud framework exists for real. The existing serverless offerings are AWS Lambda [6], that led to offers including "IBM cloud functions" and "Google Cloud Run" [7].

Nonetheless, with the deployment of serverless software, there are greater chances to get locked in through specific cloud service supplier. In this regards, CEO of CoreOS, Alex Polvi showed a concern that refers serverless framework as "one of the worst forms of proprietary lock-in ever seen in humans' history" [8]. The core purpose behind this is that up till now, deployed applications were required by serverless frameworks that make use of registered company specified technology along with tuning to function for them. Considering an example for the case of Lambda, deploying functions needs "AWS API Gateway integration or use of "AWS Application Load Balancer (ALB)" [9], with the essential framework required to bind in "AWS Eventing system". The whole procedure is complex and burdensome for shifting to any other technology which fails in offering the similar services.

The problem of getting locked in cloud platform has not happened for the first time. Before serverless platform, same issue was raised in case of PaaS offerings, where shifting the provider demanded significantly high cost [10]. However, the solution didn't come up to be additional middleware layer along with supplementary thought on the top of the current PaaS [11]–[13] rather having a layer of abstraction on the top of everything, and industry agreed upon keeping it the perfect set of primitives for everything. At that time, K-native [14] was

introduced, an open source PaaS system that controls, builds and organizes source-centric, up-to-date, and container-based applications by means of several hosts. In addition to that it also offers fundamental tools and devices for implementation, repayments, management and service and application scaling [14]. Although there exist a lot of additional PaaS offerings having same competencies as of K-native, but the distinguishing factors that contribute massively in adoption of K-native by industries are as follows:

1) *From provider's point of view:* K-native is an open source platform which points out that it is possible to implement it in order to execute anywhere meanwhile offering adequate flexibility and perceptibility to its internals.

2) *From operator's point of view:* K-native offers affirmative and reliable model for deploying and managing applications. Moreover, it also helps in quick ramping-up, issue handling and spreading it out.

3) *From developer's point of view:* K-native sustains a stable API structure for its suppliers that provides developers possible chance for construction against that specific API meanwhile retaining the tractability to transfer from one supplier to the other at the desired time.

In this paper, for serverless framework to enhance the extensive adoption of the K-native platform and resolving the problems that has been mainly faced by Knative serverless platforms are discussed along with the solutions that K-native serverless framework provides to overcome the issues. Other than that, benefits of Knative platform have also been elaborated by making a comparison of Knative with other serverless frameworks, that has also been deliberated in this paper. Learning from K-native practice, serverless framework must provide mutual API interface at higher level of abstraction. In addition to that, a "compatible runtime contract" enabling "extensibility and interchangeability" that is needed to avert crucial challenges must also be offered by serverless platform. ALL of these factors have been discussed and elaborated in next sections of paper.

II. KNATIVE SERVERLESS PLATFORM

Knative is an extensible, flexible and open source serverless framework that adds up the basic modules to deploy, run and arrange serverless cloud based applications to Kubernetes. These cloud based serverless modules lead to enhance the developer's throughput and decrease the overall operational cost. This has led to prevalent concerns and acceptance of Knative platform by many contributors like Google cloud, IBM, Red Hat & SAP amongst other providers. It installs on "Open Shift" with the help Operators. In addition to that, by being on the top of other platforms, this serverless Knative offers itself to conformation design which becomes stress free to operate and stand by for operators. The similarities of Knative with other serverless platforms imply that it may prove to be a good competitor to proceed through a same success path as other productive serverless frameworks do.

Knative carry out the following tasks when implemented on a system [15]:

- It deploys the container
- It composes (orchestrates) the source to URL application loads on Kubernetes platform.
- It routes and manages the traffic by means of blue or green placements.
- It automatically scales up or down (even to '0') and sizes the workloads on the basis of consumer demand.
- It combines the executing services with the Eventing Environment.

A. How Knative Works?

It uncovers the "kn" API by making use of Kubernetes functions and CRDs. Through this, application is deployed via command lines. At the back end, Knative builds all the possible Kubernetes Resources including "deployment", "services" and "ingress" etc., that are needed to run applications without any hassle.

It must be kept in mind that Knative doesn't builds the pods at once. It offers simulated endpoints for application requests and takes responsibility for them. If there comes a hit to those points, it rolls the pods needed to assist the applications. This allows Knative to scale up the applications from zero to the already set threshold point or to the desired instances amount. Knative delivers application points in "[app-name]. [namespace].[custom-domain]" format assisting in recognizing application separately. Knative platform is the combination of many open source products like "Kubernetes, Istio, Prometheus, Grafana and event stream engines" [16].

III. KNATIVE COMPONENTS

Knative is comprised of three significant components, i.e. build, event and serve. Each of the components with its function has been described in detail in next section.

A. *Knative Build*

The main goal of Knative build is to generate the artifacts that can run or execute from the application code. It turns the source code into the cloud containers or functions. As Knative is constructed on atop of Kubernetes, therefore, "Knative build process" is involved in collecting, assembling and packaging of executables like "Oracle cloud infrastructure (OCI) compatible image format" [17], and "labeled and pushed to a container registry". Additionally, Build process involves some steps that includes:

- Turning source code from cloud code depository.
- Installing basic dependencies that are required by the code to run, for instance, "environment variables and software libraries".
- Generating container images.
- Placing images in container registry for helping other developers to use.

Knative makes use of fundamental resources of Kubernetes [18], [19] for "Build process". It must be known to this framework that from where to get the assets in order to fulfil the task. It is needed to state the primary elements that supports the steps. Once the build process is completed, Knative automates the container builds.

At this end, the Knative build "application programming interface (API)" brings about two key "custom resource definition" (CRD), i.e., "Build template" and "Build CRD".

- *Build Template:* It offers a set of essential procedures for specific build to happen.
- *Build CRD:* It makes use of build Template along with the requirements of "application specified build" in order to generate the code and create runtime artifact ("Docker image").

As "Knative Build" was launched at first as a part of "Knative ecosystem", its featuring sets developed rapidly afar least requirements of creating a Docker image, as was made up at the start. As an outcome, the latest styles of Knative denounced usage of "Knative Build", and that developed as distinct project with a code that was named as, "Tekton" [20]. In this paper, the requirements to package application has also been elaborated for cloud serverless frameworks to evaluate its advantages as a measure of "serverless API interface". Fig.1 shows the Open FaaS function of Knative Build component.

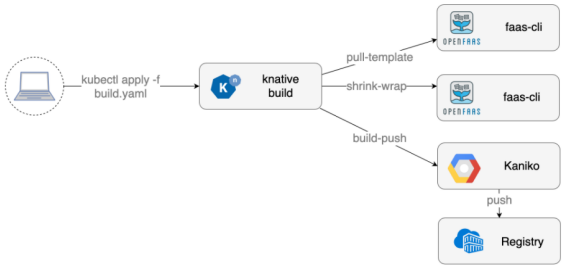

Fig. 1. Knative Build for Open FaaS Function *[21]*

B. *Knative Eventing*

This Eventing module in Knative framework offers "loose coupling of micro services in serverless". The significant three key CRDs which apprehend the "Knative Eventing API interface" are "Event source", "The Broker", & "The Trigger" [22].

- *Event Source:* Also called as "event producer" makes records of the interest in specific kinds of events to Knative that can be of two kinds i.e., internal (such as task completion) and external (such as GitHub Events).
- *Broker:* It delivers a network of "event sinks" ("also event consumers") to contribute to the Eventing actions from some particular resources of event.
- *Trigger:* It combines the event customer facility with network intended for specific kinds of events.

Some of additional CRDs that Eventing provides are:

- Enabling events filtering
- Maintaining Event registry
- Event sequencing

Nevertheless, these are considered as non-functional and are not considered necessary for "API interface Eventing". "Knative Eventing component" is considered to be reliable with "Cloud Event features" for ensuring interoperability. Fig. 2 below shows the Knative Eventing architecture for consumers.

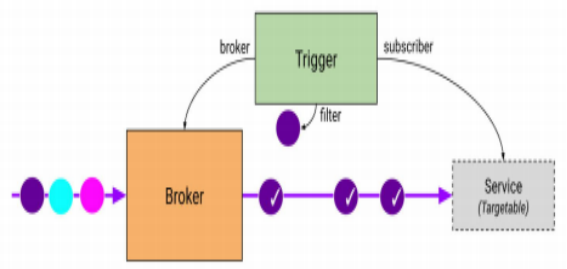

Fig. 2. Knative Eventing architecture *[23]*

C. *Knative Serving*

Knative Serving is a component that executes containers as scalable and accessible services. This component not only scales up containers to run in thousands but also scales down to point where there exist no cases of containers running at all [24].

The Knative serving is the most vital components of "API interface". Intrinsic to Kubernetes, "Knative API interface" is specified by a group of CRDs, and each of them taking "subset of functional requirements" for application of Knative. Three major assets of Knative serving API are: "Configuration", "Revision" and "Route CRD".

- *Configuration:* This resource takes the "configuration information" that is relevant to running of request case. It takes account of reference to container image for "resource constraints regarding container", "packaged application code", "configuration parameters", and "environmental variables" etc.
- *Revision:* It takes the "tagged immutable snapshot" of application configuration that is destined to a particular type of application.
- *Route:* It takes the "routing information" to several amendments of application and also the information regarding "traffic splitting and green or blue deployment".

Fig. 3 below depicts the overall structure of "Knative Serving Component".

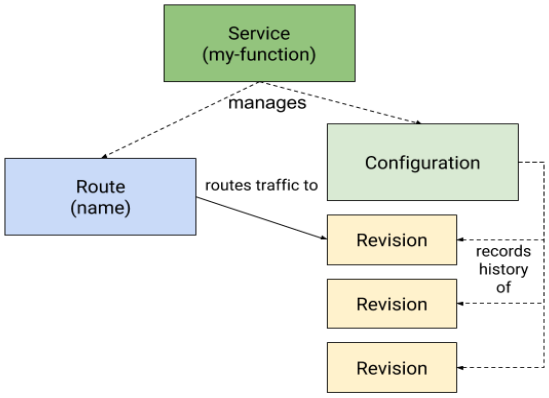

Fig. 3. Knative Serving Structure *[16]*

Fig. 4 below illustrates an advanced architecture of Knative Serving. Here, the existence of "pluggable auto scaler" elements and "service mesh" elements which are responsible for "auto scalability and server configuration" for installed applications has been highlighted. In runtime configuration of Knative API, it has generated two types of critical points [25].

- First, there is "the zero-to-many" scaling in Knative serving element for enabling application loads with inbound contemporaneous HTTP traffic. Other kinds of application/ workloads might not get advantage from the same serverless agreements.
- Second, the framework/ platform creates strong and valid assumptions for taking into consideration the application version that is prepared before making it accessible as installed service.

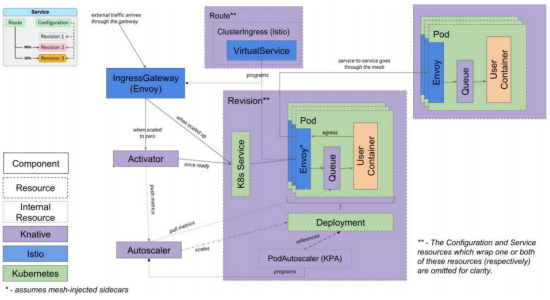

Fig. 4. Knative serving architecture *[26]*

IV. ADVANTAGES OF KNATIVE PLATFORM

Knative on Kubernetes framework provides various advantages for its users that desire to upgrade and enhance their container usage to a high level. Some of these advantages are mentioned below [27].

1) *Quicker Iterative Expansion:* Knative platform reduces the time that is consumed in the process of container building. This enables the user to develop and take out latest container versions in a faster pace. It helps in making the complex steps easy to create containers in trivial and iterative steps that is the key principle of the progressive development procedure.

2) *Emphasis on Code:* Development and operations (DevOps) may authorize the developers to manage and control their own settings (environments), but eventually it runs because of the code and coders require code. There is always an emphasis on creating virus free software and resolving development issues, but not on message configuration of bus queues for triggering of events or administrating scalability of containers. With Knative platform, all of this is possible at the same time without any hassle. It not only emphasizes on building

bug free software but also helps in creating codes.

3) *Faster Access to Serverless Computing:* To manually setting up and managing a serverless environment is a daunting task to do. Here, Knative framework allows user to rapidly make arrangements for setting up this serverless environment. As far as developers concern, they only create the containers not build the set up. It's always Knative framework that executes this as service or serverless function at the back end.

4) It is not placed as PaaS but as a general serverless platform which is used to create personal serverless PaaS [14].

5) It is a sign of establishing a distinctive framework of the serverless architecture design. Prior to Knative, every FaaS solution used to have its own events that were incompatible with other FaaS solution. But with Knative, Cloud Event standard was proposed for design [14].

6) *Complete Community Source:* Platform (communities) such as Kubernetes and Service Mesh fully provide sustenance to the framework of Knative system.

7) *Cross-platform Backing:* Knative platform is constructed on the top of Kubernetes, that gives all universal standards to several cloud providers. Hence, Knative also run the "cross platform support" and alleviates the risk or possibility of linking to the providers or vendors.

8) Knative is a complete serverless model as it has more productive designs than other serverless solutions. For instance, event model is considered to be flawless. It implements full fledge set of design that contains registration, subscription and link to peripheral Event systems.

9) Knative creates application from source code to images.

10) Knative Serving maintains the comparative phased release.

V. KNATIVE CHALLENGES AND THEIR SOLUTIONS

The Knative platform helps in solving a lot of real world issues that are faced by developers. Some of the problems includes the following [27], [28]:

1) *CI/CD set up:* "Continuous integration and continuous Deployment" work flows are the hub of DevOps procedures. Autonomous gating for application builds and installation can decrease the installation times meanwhile enhancing the application's quality. The issue is that the autonomous workflows not only takes a lot of time but also needs expertise for setting up that usually composed of variety of products. Knative is a beneficial tool that is used as a portion of pipeline "autonomous projects" in order to get DevOps teams up and executing.

2) *Customer Rollouts:* The issue with the quick release cadence is that developers do not test the softwares on their users. Rolling out latest software versions to the end users exposes its issues that eventually affects the business processes. "Knative configuration and routing system" makes it permissible for the user to uncover/expose the innovative container versions to a subgroup of the customer base and then progressively increasing the consumers with the passage of time and rapidly rolling back to the previous version if there arise any problem in the latest revision of the software.

3) *Cold Start Cost:* This is the most discussed problem of serverless system. To take the full advantage, these serverless providers opt to completely close the inactive operations/ functions. When the workload starts again, the cost cause of resuming function effects the response time. This function multiplies, when one set-up is linked with many serverless operations and are chained together. To mitigate this issue, numerous consumers embed their functions artificially just to assure that the system keeps on working. To efficiently and successfully linking this technique with a system of interconnected services, it is vital and important to know the end-to-end association among them, such that the services connected in chains may kept activated by creating necessary end-to-end tracing [29].

4) *Vendor Lock-in:* Serverless is basically specified to cloud server providers. In this scenario, transformation from one cloud platform to another cloud platform becomes difficult and extremely challenging. Although almost all the cloud providers are becoming more similar to each other and there exists some tools to operate "multi-

cloud workloads", still it is a gigantic challenge to move from one cloud provider to another. This is known as "vendor Lock-in". This issue is more common in serverless platforms. Knative provides solutions to this issue by developing an API layer common to all serverless platforms [30].

5) *Throttling:* Serverless frameworks restricts the number of synchronized applications that its functions have to run, usually both at account and discrete operation level. Once the threshold is achieved, upcoming requests are lined up in a queue that eventually increase the response time. Despite the fact that it may look counter intuitive to throttle efficiently infinite resources to compute, it avoids contact to possibly limitless bills where capacity is allocated on consumption basis. In order to avoid this issue, Knative serverless elevates the threshold. It set the threshold in such a way that it fulfils the non-functional needs in regards to response time and parallel usage. It also needs the end-to-end prominence so that it can be clearly observed what was throttled and the impact of this throttling on end-user experience [31, p. 3].

6) *Security:* It is considered to be one of the crucial challenges of serverless platforms with poorly configured applications that can lead to several issues and security related is one of the biggest. It is very necessary to configure the permission accurately for accessing any services of serverless platform. If the permissions are not specified for any function, there may exist a room for security breach. Another issue regarding security is that inside mechanism of cloud server is not relocated or transmitted to the outside cloud server. Therefore, if the serverless platform is connected to the third party service by means of HTTP, it must be assured that the data is encrypted and connections are secure. In order to avoid such security problems, it is very important to provide the accurate and exact permissions to the Knative resources for performing the tasks. Also it must be made sure to encrypt all the data. In addition to that third-party connections must be secured in Knative serverless platform [32, p. 5].

VI. KNATIVE AND OTHER SERVERLESS PLATFORMS (A COMPARISON)

In this section, a comparison among Knative and other currently deployed serverless platforms have been done based on their API and runtime contracts. On the basis of Knative analysis, seven points are taken into consideration for comparison whereas it has also been summarized in Table 1 below. Knative has been compared with the following serverless platforms: AWS Lambda [6], OpenFaaS [33], ApacheOpenWhisk [34], and Kubeless [35]. In this comparison, cost models and other cost related factors have been alleviated because of the independent charging scheme from original API installation and "runtime contract" meant for the specified platform. Details of each point is elaborated below with respect to all the existing serverless platforms/ frameworks.

A. Packaging Contract

Platforms are comprised of several ways to consume application artifacts. Nevertheless, in recent times, the huge adoption of micro service based architecture and platforms including Kubernetes with OCI functionality have gained a noteworthy attraction as standard. Frameworks like Knative, Kubeless, OpenFaaS with their Kubernetes roots provide backing to the "container image" as the sole package layout for its submissions.

- In Knative, extensive usage of "container image format" makes the consumer to make an arrangement with packaging, by hiding all the details from platform.
- "Kubeless & OpenFaaS" choose a mid-way where platforms take the user functions and integrate with the "OCI image" to be executed by serverless platform.
- On the other hand, "AWS Lambda" have opted for not disclosing the large space to user and constructed a custom way consume compiled code and execute with "predefined set of runtimes".

B. Runtime Invocation Contract (RIC)

RIC is defined as the API boundary among runtime and serverless frameworks irrespective of being user provided or platform provided. Runtime is considered as device for running the application.

- "Knative serving component" doesn't require this runtime in order to cope with and operate the applications. Instead it directly forwards the HTTP stream to application, considering it able to deal with HTTP protocols.
- AWS Lambda assumes runtime to request "Lambda Runtime API" rather than the platform. It is comprised of "/runtime/init/error" "/runtime/invocations/next" and "/runtime/invocation/$AwsRequestId/

response" endpoints that are casted off by means of run-time for pulling requests, then it forwards these endpoints to the requests, waits for the request responses, and lastly send it to the established responses [6].

- "Open Whisk" tracks the "push based technique" assuming runtimes taking the "requests over HTTP" through "/init" and "/run" endpoints.
- OpenFaaS covers every application by the help of "Watchdog service". It [33] is an "HTTP service" which is accountable for taking HTTP applications, running the request codes in process and fleeting application load through "stdin" and eventually reporting back.
- Like Open Whisk, Kubeless also needs runtime to take the application over HTTPS.

C. Application Invocation Contract (AIC)

AIC is defined as the "API boundary" amidst "runtime and application" input and output.

- "Knative Serving" at present provide backings to both HTTP (1 & 2) protocols and doesn't dwell any extra constraints on requested application or responses. In this case, Applications make use of standard HTTP characteristics and execute on Knative unaltered. Knative Eventing that forwards the event payloads to "Knative serving services" makes use of Cloud Events present atop of HTTP [26].
- Contrary to Knative, "AWS Lambda applications" completely agree to take and return payload by means of "Lambda's custom envelope". Lambda environment offers runtimes to find out an easy way to work with custom API [6].
- "Open Whisk" also makes use of "custom JSON envelope" for response or request loads. Nevertheless, it passes in "raw HTTP stream", evading conventional concept [34].
- "OpenFaaS" makes use of "stdin" and "stdout" for sending and receiving the request and responses after each application [33].
- Kubeless takes the "language specific bindings" which passes with the simple data items to applications and regain "HTTP response" items [35].

D. Execution Model

There are two main "request execution styles" in serverless platforms that are "synchronous style" and "asynchronous style".

- "Knative Serving component and Kubeless" provide backing to synchronous implementation style.
- "AWS Lambda, OpenFaaS and Open Whisk" supports both of these implementation styles [6].

Between them, elusive dissimilarities are present regarding the procedure to require platform to run application request like asynchronous through "HTTP header" in "AWS Lambda", through path prefix in OpenFaaS and Knative, through query parameter in Open Whisk. Execution style is always invisible in application of platforms/frameworks.

E. Retry Model

For synchronous style, no autonomous retries are provided by any of the platforms. All of the depend on clients for this purpose.

For asynchronous style execution, three of the platforms i.e., Knative Eventing, Open Whisk and Kubeless do not provide any built in retry feature. However, AWS Lambda logically retries on the basis of received error type. Platform errors and faults are not provided with the chance to retry such as lack of permission but application errors are retried. Asynchronous invocation that fails is retried twice [26].

F. Concurrency Model and Auto scaling

- Knative provides pluggable API in order to achieve "application auto scaling" that is built on Kubernetes API's lower level and by default transfers along with auto scaler that is based on auto scaler. User might become able to manually regulate and control the concurrency of application instance or depend on autonomous detection of system. In addition to that, Knative serverless platform also adjusts the applications to zero when there occur no requests for some time. Accuracy of this concurrency detection model suffers when Knative model sends the request onward to application [26].
- AWS Lambda presumes application runtime to put away requests in appropriate time providing every runtime to decide independently for each application instance. It also automatically scales up the on demand instances and also scales down to

zero when there is no request present in the pipeline.

- Open Whisk, Kubeless, OpenFaaS also have exactly same model as that of Knative.
- OpenFaaS carry out the scaling process on the basis of requests rate, CPU and memory consumption.
- Kubeless makes use of "Kubernetes' Horizontal Pod Auto scaler (HPA) resource" in order to regulate resource based auto scaling of CPU, memory and custom metrics. Kubeless because of HPA does not provide scale-to-zero feature.
- Unlike Kubeless, AWS Lambda and Open Whisk do not provide backing for CPU and memory scaling.

G. Traffic Splitting

- "Knative and AWS Lambda" both the platforms have in built backings for splitting the percentage of upcoming traffic to the already installed previous versions of applications.
- Unlike the above two, Open Whisk possess no such built-in support for traffics splitting. Therefore, it needs to handle the upcoming traffic by installing several applications and also making use of outsider load balancer.
- Similar to Open Whisk, OpenFaaS and Kubeless also don't have built in traffic splitting support. Yet, using the Kubernetes roots, both the serverless platforms easily inter operate with numerous service mesh techniques in order to facilitate the platforms with traffic splitting.

In Table 1 below, comparison of each of the seven points have been summarized.

Table 1. Summary of the Comparison among existing Serverless Platforms [26]

Platform	Packaging Constraints	RIC	AIC	Execution Model	Retry Model	Concurrency Model	Traffic Splitting
Knative	OCI Image	None	HTTP (1 &2) Cloud Events	Only Synchronous	None	Request & Resource based	Built in
AWS Lambda	Custom Package	HTTP Service	JSON Envelope	Both synchronous and asynchronous	Functional Failure	Only Request based	Built in
Open Whisk	Both	HTTP Service	JSON Envelope	Both	None	Only Request Based	External load balancing
OpenFaaS	Both	HTTP Service	stdin and stdout	Both	On Timeout	Request & Resource based	External service Mesh
Kubeless	Both	HTTP Service	Custom Object	Both	None	Request & Resource based	External Service Mesh

VII. CONCLUSION

In this paper, Knative serverless framework has been deliberately elaborated in detail. Important areas and challenges that Knative platform faces along with its solutions have also been discussed. Key design decisions and external contracts have been explored in detail here which are then compared with the currently existing serverless platforms. Some of the significant advantages of Knative platform are also discussed that helps not only in avoiding the challenges but also provide the opportunity to this platform to develop further by taking into considerations the key beneficial points.

For future work, it is recommended to explore and implement the use cases of Knative platform and compare them with the use cases of other existing platforms in order to extract the easiness, and extensibility of each platform. This would help a lot in knowing and predicting the degree of extent that serverless space would progress.

REFERENCES

[1]S. Taherizadeh and M. Grobelnik, "Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications," *Adv. Eng. Softw.*, vol. 140, p. 102734, Feb. 2020, doi: 10.1016/j.advengsoft.2019.102734.

[2]E. Berkner, M. Thömmes, P. Arnold, A. Shafir, A. Sholem, and E. Marchant, "Wide adoption of serverless will be led by tomorrow's giants, not today's," p. 34.

[3]G. Chandra, "KEDA — Kubernetes Based Event Driven Autoscaling," *Medium*, May 08, 2019. https://itnext.io/keda-kubernetes-based-event-driven-autoscaling-48491c79ec74 (accessed Oct. 07, 2020).

[4]J. Cao *et al.*, "An Edge-Centric Scalable Intelligent Framework To Collaboratively Execute DNN," p. 2.

[5]KLau, "Kubernetes and Big Data: A Gentle Introduction," *Medium*, Feb. 24, 2020. https://medium.com/sfu-cspmp/kubernetes-and-big-data-a-gentle-introduction-6f32b5570770 (accessed Oct. 07, 2020).

[6]"Serverless Web Apps With Knative Compared to AWS Lambda - DZone Cloud," *dzone.com*. https://dzone.com/articles/serverless-web-apps-with-knative-compared-to-aws-l (accessed Oct. 07, 2020).

[7]"Choosing a Serverless Option | Serverless Guide," *Google Cloud*. https://cloud.google.com/serverless-options (accessed Oct. 07, 2020).

[8]"'Lambda and serverless is one of the worst forms of proprietary lock-in we've ever seen in the history of humanity.'" https://www.theregister.com/2017/11/06/coreos_kubernetes_v_world/ (accessed Oct. 07, 2020).

[9]"Elastic Load Balancing - Application Load Balancers," p. 126.

[10]Kewei Sun and Ying Li, "Effort Estimation in Cloud Migration Process," in *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*, Redwood City, Mar. 2013, pp. 84–91, doi: 10.1109/SOSE.2013.29.

[11]P. R. da Cunha, P. Melo, and C. F. da Silva, "Avoiding Lock-In: Timely Reconfiguration of a Virtual Cloud Platform on Top of

Multiple PaaS and IaaS Providers," in *2013 IEEE Sixth International Conference on Cloud Computing*, Santa Clara, CA, Jun. 2013, pp. 970–971, doi: 10.1109/CLOUD.2013.36.

[12]S. Kolb and G. Wirtz, "Towards Application Portability in Platform as a Service," in *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, Oxford, United Kingdom, Apr. 2014, pp. 218–229, doi: 10.1109/SOSE.2014.26.

[13]R. Yasrab and N. Gu, "Multi-cloud PaaS Architecture (MCPA): A Solution to Cloud Lock-In," in *2016 3rd International Conference on Information Science and Control Engineering (ICISCE)*, Beijing, China, Jul. 2016, pp. 473–477, doi: 10.1109/ICISCE.2016.108.

[14]A. Cloud, "What Exactly Is Knative?," *Medium*, Sep. 10, 2019. https://medium.com/datadriveninvestor/what-exactly-is-knative-252ec94e4de7 (accessed Oct. 07, 2020).

[15] Grace, "Knative what's that now?," *Medium*, Apr. 20, 2019. https://medium.com/@grapesfrog/knative-whats-that-now-65041e585d3d (accessed Oct. 07, 2020).

[16] U. Dharmawardana, "Serverless Applications with Knative," *Medium*, Nov. 26, 2019. https://blog.usejournal.com/serverless-applications-with-knative-910caf505b79 (accessed Oct. 08, 2020).

[17]"opencontainers/image-spec," *GitHub*. https://github.com/opencontainers/image-spec (accessed Oct. 08, 2020).

[18]J. Li, S. G. Kulkarni, K. K. Ramakrishnan, and D. Li, "Understanding Open Source Serverless Platforms: Design Considerations and Performance," *Proc. 5th Int. Workshop Serverless Comput. - WOSC 19*, pp. 37–42, 2019, doi: 10.1145/3366623.3368139.

[19]"Production-Grade Container Orchestration," *Kubernetes*. https://kubernetes.io/ (accessed Oct. 08, 2020).

[20]"Tekton," *Tekton*. / (accessed Oct. 08, 2020).

[21]"First look at knative build for OpenFaaS functions," *Alex Ellis' Blog*, Apr. 17, 2019. https://blog.alexellis.io/first-look-at-knative-build-for-openfaas-functions/ (accessed Oct. 07, 2020).

[22]P. García-López, A. Arjona, J. Sampe, A. Slominski, and L. Villard, "Triggerflow: Trigger-based Orchestration of Serverless Workflows," *Proc. 14th ACM Int. Conf. Distrib. Event-Based Syst.*, pp. 3–14, Jul. 2020, doi: 10.1145/3401025.3401731.

[23]J. Klaise, A. Van Looveren, C. Cox, G. Vacanti, and A. Coca, "Monitoring and explainability of models in production," *ArXiv200706299 Cs Stat*, Jul. 2020, Accessed: Oct. 07, 2020. [Online]. Available: http://arxiv.org/abs/2007.06299.

[24]H. Rempel, "LibGuides: Zotero: Add Websites." https://guides.library.oregonstate.edu/zotero/addwebsites (accessed Oct. 07, 2020).

[25]"knative/serving," *GitHub*. https://github.com/knative/serving (accessed Oct. 08, 2020).

[26]N. Kaviani, D. Kalinin, and M. Maximilien, "Towards Serverless as Commodity: a case of Knative," in *Proceedings of the 5th International Workshop on Serverless Computing - WOSC '19*, Davis, CA, USA, 2019, pp. 13–18, doi: 10.1145/3366623.3368135.

[27]"us-en_cloud_learnhub_knative_a_complete_guide," Jun. 01, 2020. https://www.ibm.com/cloud/learn/knative (accessed Oct. 07, 2020).

[28]H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless Computing: A Survey of Opportunities, Challenges and Applications," *ArXiv191101296 Cs*, Dec. 2019, Accessed: Oct. 08, 2020. [Online]. Available: http://arxiv.org/abs/1911.01296.

[29]P.-M. Lin and A. Glikson, "Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach," *ArXiv190312221 Cs*, Mar. 2019, Accessed: Oct. 08, 2020. [Online]. Available: http://arxiv.org/abs/1903.12221.

[30]"Serverless Challenges and Solutions," *Dashbird*, Nov. 05, 2019. https://dashbird.io/knowledge-base/basic-concepts/serverless-challenges-and-solutions/ (accessed Oct. 08, 2020).

[31]P. Holditch, "The top 3 serverless computing problems and how to solve them," *InfoWorld*, Feb. 06, 2019. https://www.infoworld.com/article/3338111/the-top-3-serverless-problems-and-how-to-solve-them.html (accessed Oct. 08, 2020).

[32]"The Top 5 Pitfalls of Serverless Computing and How to Overcome Them," *Logz.io*, Jun. 14, 2019. https://logz.io/blog/pitfalls-of-serverless/ (accessed Oct. 08, 2020).

[33]"openfaas/faas-netes,"*GitHub*. https://github.com/openfaas/faasnetes (accessed Oct. 08, 2020).

[34]"An Architectural View of Apache OpenWhisk," *The New Stack*, Feb. 03, 2017. https://thenewstack.io/behind-scenes-apache-openwhisk-serverless-platform/ (accessed Oct. 08, 2020).

[35]"kubeless/kubeless,"*GitHub*.https://github.com/kubeless/kubeless (accessed Oct. 08, 2020).