Toward Lighter Containers for the Edge

Misun Park, Ketan Bhardwaj, Ada Gavrilovska
Georgia Institute of Technology

Abstract

Edge computing environments, being resource-limited, cannot tolerate the bloat in size of edge applications due to use of thick, complex software runtimes and hardware acceleration support. But such capabilities are critical to support rich, diverse and high performance applications. Performance includes deployment time, responsiveness and scalability, and is impacted by the bloat if using cloud-native container-based systems at the edge. If not addressed, this will limit the ability of the edge to scale to increasing number of workloads.

This paper makes a case for a new featherweight system – Pocket– to support edge computing. Pocket addresses the limitations in current container-based systems while retaining their benefits. Pocket achieves this by splitting containerized applications into two parts: application container and a bloat-causing execution environment container. Experimental evaluations of an early prototype show that by sharing the execution environment containers across multiple application containers, Pocket is able to achieve significant reductions of the resource pressure at the edge, thus presenting a path toward greater efficiency and scalability for edge computing.

1 Introduction

As edge computing is gaining momentum, the software/hardware infrastructure that will enable it is getting defined [2, 5, 9, 12, 20, 21]. Across these initiatives, cloud-native container-based solutions, in particular Docker containers and Kubernetes, are emerging as leading contenders that will shape the edge computing software stack. Containers offer a number of benefits by offering a well-established robust infrastructure with years of investments of a large developers community, unconstrained application development with support for packaging, distribution and deployment, and desirable properties such as near-native performance and isolation. However, edge computing presents unique execution and deployment constraints, distinct from datacenter-based cloud environments. This warrants a deeper investigation of the performance, efficiency, and scalability implications of using a cloud-native technology at the edge.

Edge applications are as rich and diverse as cloud applications, and developing them raises the need for using complex runtimes, such as for analytics and ML/AI [18, 19]. Their data and performance requirements are such that they require hardware acceleration support [4, 10, 11, 13]. The software/hardware requirements of these application stacks lead to a bloat in container sizes, which impacts container deployment times, resource footprint, responsiveness/launch times, execution time, and other metrics critical when operating at the edge. For instance, Table 1 shows that the size of a container supporting a CPU-based TensorFlow library increases by nearly 17× compared to the latest "vanilla" Linux container. This factor will only increase once support for GPU accelerators, such as the CUDA runtime, is added [18].

This is a major problem for the edge, where, due to the fixed resource capacity at an edge location, the only way to scale to larger numbers of services or service instances is by reducing the per instance resource requirements, and thus improving the resource efficiency. The edge is envisioned as instrumental in addressing the latency and data movement requirements for many applications that rely on support for such "thick" runtimes, for AR/VR (e.g., Unity [26]), machine learning [1, 6, 7], event streaming [16, 28], etc. These applications will need to be deployed and configured dynamically, and can potentially be long-running. Furthermore, each edge location will need to support many such applications, each deployed independently with their own container images, operating potentially on behalf of different users or stakeholders. *To ensure the scalability of edge computing for meeting the demand of the many new services and use cases it is positioned to support, it is critical to revisit current trends of defining container-based software stacks for the edge.*

In response, we explore an approach which combines use of specialized higher-level runtime services as part of a container-based infrastructure, combined with lightweight, *"pocket-sized"* application-specific containers. The outcome of this is **Pocket**, an *edge-native* solution that creates an edge that can scale to a larger number of application instances, each of which can be independently configured, deployed, and man-

Images from Docker Hub	ubuntu:latest	tensorflow/tensorflow: 2.0.1-py3)
Size	25.9 MB	428.21 MB

Table 1: Image sizes of a container with heavy runtime. Heavy runtimes such as accelerator support or machine learning frameworks lead to bloat in image size.

aged. More specifically, by partitioning a container into two parts, Pocket proposes a new application architecture that is more suitable for the edge environment, providing improved application performance and resource efficiency. At the same time, Pocket retains the advantages of container-provided packaging, delivery infrastructure, and resource management.

2 Limitations of Containers at the Edge

Monolithic containers. Monolithic containers provide convenience in terms of managing legacy applications, while retaining a familiar environment and access to file systems and resources, as in a native deployment. However, in terms of image size, traditional monolithic containers are not suitable for all edge applications. Container images with heavy runtimes, such as TensorFlow or NVidia CUDA, range in size from hundreds of megabytes to a few gigabytes. For edge environments, this presents challenges in terms of dynamic image distribution and management over (wide-area) networks.

Monolithic containers also present challenges in terms of the application responsiveness. Creating and launching a container with the `docker run` command, generally takes a few seconds, and the launch time typically increases with image sizes, and the number of concurrently created containers. Warm launch or restart techniques reduce this overhead [22, 23], but at the expense of maintaining in-memory snapshots of pre-initialized container state. For a resource-constrained edge, the footprint size of the snapshots will have an impact on scaling to larger numbers of instances. More importantly, warm restart alone is not sufficient for containers encapsulating heavy runtimes – in our measurements restarting from a warm snapshot of a pre-initialized TensorFlow image still requires 1.6s, which is orders of magnitude larger than the latency ranges relevant for many emerging edge computing applications [24].

Modernized container-based applications. A new trend in datacenters is to modernize legacy applications by rewriting them for microservice-based and serverless deployments. While these approaches address some of the issues with monolithic containers raised above, they require modification to applications. This will limit how existing applications and application-development skills and toolchains can be leveraged to create the edge-based application ecosystem. Furthermore, they pose additional limitations in terms of runtime overheads related to RPC, data serialization and data movement costs, or expectations of executing only short-lived stateless codes, that will further impact their applicability.

3 Pocket Overview

Approach. The above discussion points to two desirable features of an edge-native solution for applications: (i) use of shared, long-running runtime services as part of the edge platform stack to reduce bloat and improve resource efficiency at the edge, and (ii) use of lightweight data sharing methods to eliminate the need for serialization and copying of complex data structures. To address these needs, Pocket uses the concept of *pocketizing* which allows edge applications to be treated as (i) containerized *pockets* that only include application-specific functionality, and (ii) containerized services that comprise the execution environment including the heavy runtimes, hardware accelerator-related software support, etc. While the containerized services are assumed to be deployed offline, perhaps as part of the edge platform stack, and can be bulky, the pockets are featherweight container with small image size, and can be quickly deployed and executed while still providing the same functionality as full containers. *Pockets become the new application deployment unit for edge computing.*

Unique about the approach advocated by Pocket is the way containerized pockets interact with the containerized services. The pockets latch onto service domains, but have access to their own namespace and carry out their own functions. Pockets get access to the execution environment (including software frameworks and access to specialized hardware) of the service domain but use it for their function. In this manner, Pocket allows multiple applications to share runtime resources, while also offering a path to leverage desirable containerization properties such as packaging, deployment of legacy applications, and per application-pocket resource controls (via Docker's *cgroups*).

There are several advantages that Pocket achieves in doing this. First, it allows for compact pocket (application)- containers, enabling faster deployment with lower network bandwidth requirement. Second, pocket containers contain only application functionality, increasing cohesion and maintainability.

By designing Pocket to be based on containers, we leverage the existing momentum of a large developer community, providing delivery infrastructure for containers (e.g., container registries), support for orchestration of containers and their resources, including important system-level mechanisms (cgroups, namespaces, runc, etc.) that are reaching maturity. Any edge system should be built on these, as opposed to starting from scratch. One concrete example of this is that Pocket leverages the built-in resource management support for containers in the form of new cgroups being created; the Pocket platform reuses cgroups for application containers without creating new ones and avoids associated overheads. Finally, the advantage of being able to operate as and in the container ecosystem is that Pocket can immediately provide benefits to existing service architectures and cloud applications being planned for deployment at edge.

Design Considerations. Core design decisions toward realizing Pocket concern support for (i) concurrency, to permit the concurrent execution of pocket applications, (ii) lightweight cross-container interactions, to limit the impact of the pocket-service communications, and (iii) dynamic resource management, to control how service container resources are adjusted and allocated to pockets.

Spawning separate monolithic Docker instances for each request trivially achieves concurrency and provides per-instance resource controls. For Pocket, this requires support for concurrency and resource allocation controls in the shared service runtime. For instance, TensorFlow v2.0 core supports the notion of sessions [8] which encapsulate the environment in which operations in a graph are run. Each session owns its own resources which allows for resource management across pockets. Important to note is that use of sessions replicates some portions of the runtime functionality and resources, however, there are benefits from reusing the rest of the runtime stack.

A conventional way for cross-container interactions in microservice based architecture is use of RPC. Popular frameworks such as gRPC introduce data movement runtime overheads due to data serialization, which can be significant for data structures used in DNN such as tensors [18]. They also increase resource demand, since now each pocket container would need to have access to a gRPC runtime, which translates to both launch time and execution time overheads.

One way to remove data movement costs is to provide the service container with access to the pocket namespace, and then to simply pass identifiers as part of the cross-domain RPCs. At a fundamental level, this approach trades data movement (serialization, copying) cost for session-based permission management. This can be achieved by exposing a session interface in the service container. For Pocket-rpc we achieve this by leveraging the existing session support in TensorFlow. This retains the runtime overheads of per-pocket resource footprint increases due to the RPC runtime, which for gRPC is significant. Another appoach to completely eliminate the gRPC-dependency in Pocket is to offer a generic SSH interface to service containers, which we use in Pocket-ssh. The trade-offs here depend on the kind of support for concurrency in the runtime and the overhead associated with passing control, data, resource allocations and namespace isolation. In that sense, the above decisions impact the binding among pocket and service containers, and the mechanisms and granularity at which pocket resources can be managed.

Exposing the pocket namespace to the service container requires changes in the conventional view of logical isolation in containers. Specifically, Pocket does in a way circumvent the isolation provided by Linux-provided mount namespaces, as they are implemented today. However, one can envision support for hierarchical mount namespaces with an API-level permission management module running in the service, much like the permission module in Android.

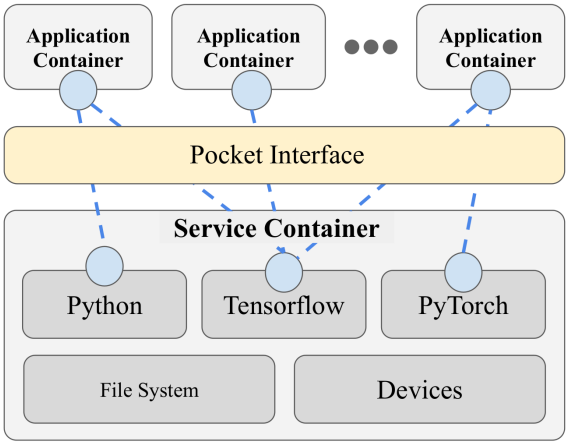


Figure 1: Pocket Architecture: Pocket plays a role as an interface/medium between a server container and a client containers and manages their health and communication. Client containers can benefit from shared runtimes offered by Pocket.

Early Implementation Details. Pocket divides a service container into two layers of containers: a pocket application container and a service container. Pocket exposes commandline APIs, such as `pocket [service_container_name] on` and `pocket [application_container_name]`, to create and launch first the service container, and then any other application container. Internally, Pocket runs Docker containers, puts them into its containers pool, and facilitates their interactions.

The service container exposes its resources to applications by presenting a pocket login interface akin to a secure remote login server. Since each pocket login is a different isolated instance, service containers can support several application containers simultaneously, and as a result, the heavy runtimes can be shared. Service containers are created as a privileged container in order to dynamically bind and reuse the cgroup of the container in a pocket login session, and also because some namespaces have to be visible to the service container to facilitate application execution.

The application containers contain the application itself and functionality needed to interact with the service container – a gRPC runtime for Pocket-rpc or an SSH login session in Pocket-ssh. Application containers are created by Pocket and stored internally in the container pool, and launched in response to client requests. As such any container can be an application container, the only requirement is that it contains files relevant to its workload in its own home directory.

At launch, the application container opens a pocket login session to the server container. For Pocket-ssh, this also involves mounting (parts of) the pocket file system as a subdirectory in the service container. In this session, an application workload runs in the context of the service container. Each session is a process in the service container, but the cgroup of the application container is reused to manage session resources.

The communication among the host, service and application containers, as well as the support for initializing cross-

Processors	Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz; 2 Processors; 24 cores; 48 threads
Motherboard	Dell Inc. 0CNCJW
OS Drive	ATA ST9250610NS
Memory	128GiB
Operating System	18.04.3 LTS (GNU/Linux 4.15.0-76-generic x86_64)
Software Settings	tensorflow/tensorflow:2.1.0-py3 gcc 7.4.0 python 3.6.9 tensorflow 2.1.0 libprotoc 3.11.2
Base Images	tensorflow: 2.0.0-py3 (476.25 MB, for Monolithic apps and Pocket service container) python:3.6-slim-buster (55.96 MB, for Pocket app container)

Table 2: Testbed Setup

container interactions, are facilitated via a Pocket daemon. The daemon also provides a monitoring and debugging interface.

One intricate implementation detail of the current implementation of Pocket is how to mount paths of application containers to the service container *dynamically*, since mount namespaces of the service containers are isolated from the host on creation. At present, to facilitate dynamic binding, Pocket errs on the side of simplicity, and mounts the entire file system of the application when its container is created. This can be hardened by separating application container specific files, e.g., pocket login keys from workload specific files in different directories and selectively mounting only the workload specific directory into the service container namespace. Another idea is to use separate volumes which then can be mounted through an orchestrator like Kubernetes.

4 Initial Results and Discussion

The goal of the experimental evaluation is to provide early insights into the feasibility of the approach. We aim to answer the following questions:

- What is the impact of Pocket on application performance (execution time) and responsiveness (launch time)?
- What is the impact of Pocket on the scalability of the edge infrastructure in terms of supporting larger number of application instances?

Testbed. The experimental testbed is summarized in Table 2. All experiments are performed using an object detection application operating on a single image input using Yolo version 3, ported to TensorFlow 2 [30].

Application performance. Table 3 compares the application performance with Pocket vs. with monolithic Docker, with respect to application execution time. Each experiment consists of one or more concurrent instances of Pocket clients generating object detection requests. The results show that Pocket improves execution time, reducing in down to 0.26× for the 20 client case.

# Instances	Pocket (second)	Monolithic (second)
n=1	10.75	10.64
n=5	9.944	11.288
n=10	4.442	12.335
n=20	3.3245	12.663

Table 3: End-to-end average execution time

# Instances	Pocket-ssh	Pocket-rpc	Monolithic
1	58.69	2793.50	2575.63
10	63.55	6622.37	5800.86

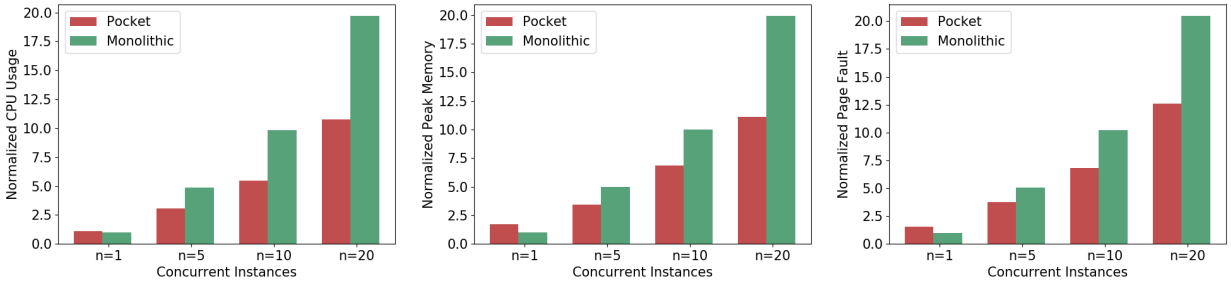
Table 4: Mean time to launch 1 & 10 concurrent instances

We also measure the application launch time with Pocket and Docker. The gRPC-based version of Pocket does not lead to performance improvements since it substitutes the initialization overhead of the TensorFlow for the gRPC runtime. Pocket-ssh eliminates the initialization-time overheads of gRPC, resulting in application responsiveness that is 44× faster than that of conventional container launch times. When running multiple concurrent workload instances, i.e., 10 in Table 4, the benefits of Pocket with respect to application responsiveness further increase.

Scalability of edge resources. To demonstrate the impact of Pocket on the efficiency of the edge platform, and its ability to scale to larger number of instances, we compare Pocket and monolithic Docker with respect to several system-level metrics, while varying the number of concurrent instances. As Figure 2 shows, Pocket is overall superior with respect to all of the measured attributes – CPU utilization, peak memory usages and page faults. For the case of a single concurrent Pocket container, the results expose the resource overheads of Pocket. This is not surprising, given that Pocket uses a service container with a resource footprint comparable to the monolithic instance, and an additional Pocket application container. However, as the number of concurrent instances starts increasing, these overheads are outweighed by the benefits of using a shared application runtime backend, and lead to 2× reduction in resource utilization across different metrics. This is important, as the design goal of Pocket is to provide improvements in scenarios where multiple function instances use the runtime service. Furthermore, the compound effect of reduced execution time *and* reduced resource consumption (e.g., CPU), points to opportunities for nearly 3× reduced energy demand, based on our measurements.

5 Related Work

Recently, there have been a number of developments on improving the resource footprint, performance and responsiveness of container-based workloads. SOCK [23] is aimed at reducing the launch time of Docker containers by relying on warm restart from snapshots of pre-initialized images. While this greatly reduces start-up time, upon launch the runtimes are not shared. Runtime memory overheads are reduced



(a) CPU Usage measured in CPU Seconds **(b)** Peak Memory Usage **(c)** Page Faults

Figure 2: Normalized performance measurements with varying the number of concurrent instances

through techniques such as those adopted in AWS Layers or Catalyzer [15] targeting serverless computing. Both of these projects rely on memory or image file overlays to promote sharing of read-only state, which, combined with warm restart, contributes to improved launch times compared to monolithic approaches. These solutions require additional program restructuring to separate the custom from shared state. Furthermore, regardless of the degree of memory sharing across application instances, there remain separate runtime instances for each Lambda function.

Catalyzer [15] and other solutions [14] exploit language-level techniques for encapsulation and isolation, that can be used in conjunction with managed runtimes. Such techniques are limited to the target programming languages, for instance Rust in [14], and cannot be generally used for existing cloud applications without a significant porting effort. This is because cloud-native applications are multi-glot, which is one of the reasons they are deployed using OS containers. For example, most microservices support lambdas written in scripting languages such as node.js and Python, so moving all such implementations to Rust might not be practical.

Pocket shares similar goals with Cntr [27], which also proposes a solution to reduce container image sizes without restricting their programming stack or deployment. Cntr also splits a container into two, storing the application in a "slim" image, and storing the tools for debugging the application in a "fat" image. However, the notion of fat images in Cntr is different from the service containers in Pocket. Cntr's fat images provide an interface to tools and runtimes to avoid application container having to install additional packages that are not relevant to the core application functionality. Thus, unlike in Pocket, the services in the a fat containers in Cntr are off the critical path and their resource management and performance are not a key focus of the design. Moreover, a slim container is granted access to a fat container which is opposite of what Pocket achieves, i.e., Pocket service containers gains access to the application container resources and namespace to carry out the application's core functions.

Finally, concerning the shared use of accelerators such as GPUs, which are important for the edge use cases that Pocket considers, prior work has identified a number of optimization opportunities through the use of higher-level (library) interface virtualization and use of shared runtimes such as CUDA [17,25], in general, and more recently, in the context of TensorFlow applications [29]. Pocket allows such techniques to be integrated in the service container as well. Concerning the image sizes of TensorFlow containers with GPU support, according to the image history from Docker Hub, most of the image bloat (around 800 MB) comes from integrating CUDA support, including for toolkits such as cuDNN. Nvidia-docker [3] allows for Docker instances to leverage the layered image support to facilitate access to the physical device, but do not reduce the bloat from the upper levels of the full runtime stack. Future work will further investigate the tradeoffs that Pocket can provide over such low-level techniques.

6 Summary and Future Directions

With Pocket, we started exploring the design choices for edge-native support for application deployment, execution and scalability. Through our experimental evaluation, we investigated the obstacles in using cloud-native container-based solution, a prevailing trend in ongoing software stack proposals for the edge. We designed Pocket, which argues for edge infrastructure based on much more lightweight containers, to maintain performance and scalability, and retain application portability and the convenience of the container-based toolchains. We plan to continue to evolve the Pocket prototype, to investigate its suitability for different classes of edge use cases, and to seek appropriate isolation and security mechanisms.

We believe that given the ultimate goals of delivering a software solution for the edge that provides performance and scalability, and minimizes the disruption to existing applications and toolchains, a prototype such as Pocket presents a unique design point, and opens up fertile ground for further research opportunities for the community.

Acknowledgement. We thank the anonymous reviewers and our shepherd, Jacob Gorm Hansen, for their feedback toward improving the paper. This work is supported by VMware and the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

Discussion

Edge computing (coupled with 5G advancements) is tasked to address the challenges to support latency-sensitive real-time applications such as autonomous driving, industrial automation, AR/VR, etc. The crux of the solution is to migrate those applications from central cloud servers to the edge of the network. To make this possible and practical, requires an appropriate software solution that is well-suited for the edge infrastructure and the semantics of the edge applications.

Pocket presents an approach to accelerate the migration of a rich, existing applications base to an *edge-ready* status, in a way that creates opportunities for a scalable, multi-tenant edge with ability to concurrently host many workloads. Pocket is a design point in a spectrum of options – from creating entirely new types of application implementations optimized to operate in the edge resource footprint, to building out an entirely new ecosystem of edge-specific (micro)services. We believe that Pocket is an important design choice, as it creates an opportunity to maximize the utilization of the edge infrastructure while leveraging existing applications, toolchains, and developers expertise, particularly given the nascent state of edge computing. We welcome the community discussion on how future edge developments should navigate this space, and the role they see for a solution such as Pocket in jump-starting the applications' process of edge migration.

We recognize the limitations of the current implementation of Pocket, and acknowledge that some of the "fixes", related to dynamic coupling of pocket containers with the platform service(s), security and isolation, privilege management, etc., will introduce some overheads. We believe that there are technical solutions that address these limitations, while still retaining the benefits of Pocket, in terms of effort required for application edge-readiness and scalability. Ongoing work delves deeper into quantitatively and qualitatively understanding the tradeoffs between Pocket and other designs based on lighter-weight containers, ranging from popular microservice-based and serverless computing, to other approach developed in the research community.

Open questions such as distribution and deployment of service-side functionality, discovery and resolution of application-runtime dependencies, etc., are largely orthogonal to the main ideas presented in this paper, but we acknowledge that they must be explored for a robust operation of a Pocket-based edge.

References

[1] Caffe | Deep Learning Framework. https://caffe.berkeleyvision.org/.

[2] ETSI Mobile Edge Computing. http://goo.gl/Qef61X.

[3] NVIDIA Container Toolkit. github.com/NVIDIA/nvidia-docker.

[4] The NVIDIA EGX Platform for Edge Computing. https://www.nvidia.com/en-us/data-center/products/egx-edge-computing/.

[5] OpenEdge Computing. http://openedgecomputing.org/.

[6] PyTorch. https://www.pytorch.org.

[7] TensorFlow. https://www.tensorflow.org/.

[8] TensorFlow Core v2.0: Session. https://www.tensorflow.org/api_docs/java/reference/org/tensorflow/Session.

[9] Akraino Edge Stack. https://www.lfedge.org/projects/akraino/, 2018.

[10] ALBANESE, A., CROSTA, P., MEANI, C., AND PAGLIERANI, P. GPU-accelerated Video Transcoding Unit for Multi-access Edge Computing Scenarios. In *The Sixteenth International Conference on Networks (ICN'17)* (04 2017).

[11] ANANTHANARAYANAN, G., BAHL, P., BODÍK, P., CHINTALAPUDI, K., PHILIPOSE, M., RAVINDRANATH, L., AND SINHA, S. Real-Time Video Analytics: The Killer App for Edge Computing. *Computer 50*, 10 (2017), 58–67.

[12] BHARDWAJ, K., SHIH, M.-W., AGARWAL, P., GAVRILOVSKA, A., KIM, T., AND SCHWAN, K. Fast, Scalable and Secure Onloading of Edge Functions Using AirBox. In *Proceedings of the 1st IEEE/ACM Symposium on Edge Computing (SEC'16)* (Washington, DC, 2016).

[13] BIOOKAGHAZADEH, S., ZHAO, M., AND REN, F. Are FPGAs Suitable for Edge Computing? In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)* (Boston, MA, jul 2018), USENIX Association.

[14] BOUCHER, S., KALIA, A., ANDERSEN, D. G., AND KAMINSKY, M. Putting the "Micro" Back in Microservice. In *USENIX Annual Technical Conference (ATC'18)* (2018), pp. 645–650.

[15] DU, D., YU, T., XIA, Y., ZANG, B., YAN, G., QIN, C., WU, Q., AND CHEN, H. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (2020), ASPLOS '20, p. 467–481.

[16] FU, X., GHAFFAR, T., DAVIS, J. C., AND LEE, D. EdgeWise: A Better Stream Processing Engine for the Edge. In *USENIX Annual Technical Conference (ATC'19)* (2019), pp. 929–946.

[17] GUPTA, V., GAVRILOVSKA, A., SCHWAN, K., KHARCHE, H., TOLIA, N., TALWAR, V., AND RANGANATHAN, P. GViM: GPU-Accelerated Virtual Machines. In *Proceedings of the 3rd ACM Workshop on System-Level Virtualization for High Performance Computing* (New York, NY, USA, 2009), HPCVirt '09, Association for Computing Machinery, p. 17–24.

[18] HSU, K.-J., BHARDWAJ, K., AND GAVRILOVSKA, A. Couper: DNN Model Slicing for Visual Analytics Containers at the Edge. In *ACM/IEEE Symposium on Edge Computing (SEC'19)* (Washington, DC, 2019).

[19] JIANG, A. H., WONG, D. L., CANEL, C., TANG, L., MISRA, I., KAMINSKY, M., KOZUCH, M. A., PILLAI, P., ANDERSEN, D. G., AND GANGER, G. R. Mainstream: Dynamic Stem-Sharing for Multi-Tenant Video Processing. In *USENIX Annual Technical Conference, USENIX ATC'18* (Boston, MA, USA, July 2018).

[20] LF EDGE: Building an Open Source Framework for the Edge. https://www.lfedge.org.

[21] World's First Public Mobile Edge. https://mobiledgex.com, 2019.

[22] NADGOWDA, S., SUNEJA, S., AND KANSO, A. Comparing Scaling Methods for Linux Containers. In *Cloud Engineering (IC2E), 2017 IEEE International Conference on* (2017), IEEE, pp. 266–272.

[23] OAKES, E., YANG, L., ZHOU, D., HOUCK, K., HARTER, T., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. {SOCK}: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)* (2018), pp. 57–70.

[24] RODRIGUEZ, P. The Edge: Evolution or Revolution. In *ACM/IEEE Symposium on Edge Computing (SEC'17)* (2017).

[25] SENGUPTA, D., GOSWAMI, A., SCHWAN, K., AND PALLAVI, K. Scheduling Multi-tenant Cloud Workloads on Accelerator-based Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)* (2014).

[26] TECHNOLOGIES, U. Unity Real-Time Development Platform | 3D, 2D VR & AR Visualizations. https://unity.com/.

[27] THALHEIM, J., BHATOTIA, P., FONSECA, P., AND KASIKCI, B. Cntr: Lightweight OS Containers. In *USENIX Annual Technical Conference (ATC'18)* (2018), pp. 199–212.

[28] VISCLOUD. Streaming Analytics Framework (SAF), 2018.

[29] YU, P., AND CHOUWDHURY, M. Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Application. In *Proc of MLSYS'20* (2020).

[30] ZHANG, Z. zzh8829/yolov3-tf2. https://github.com/zzh8829/yolov3-tf2, Feb. 2020. original-date: 2019-04-03T17:57:49Z.