RECap: Run-Escape Capsule for On-demand Managed Service Delivery in the Cloud

Shripad Nadgowda, Sahil Suneja, Canturk Isci
IBM TJ Watson Research Center, NY USA

Abstract

Application runtimes are undergoing a fundamental transformation in the cloud, from general-purpose operating systems (OSes) in virtual machines (VMs) to lightweight, minimal OSes in microcontainers. On one hand, such transformation is helping reduce application footprint in the cloud to increase agility, density and to minimize attack surface. On the other hand it makes it challenging to implement system and application management tasks. Inspired from the on-demand Function as a Service (FaaS) model in *serverless* computing, in RECap we are designing a cloud-native solution to deliver systems and application management tasks through specially-managed Capsule containers. Capsule containers are dynamically attached to the running containers for the duration of their implemented function and are safely removed from application context afterwards. More generally, RECap framework allows us to design disaggregated on-demand managed service delivery for containers in the cloud. In this paper, we describe the motivation and the emerging opportunity for RECap in the cloud. We discuss its core design principles, performance, security and manageability trade-offs. We present current design of RECap for the Kubernetes platform.

1 Introduction

The application deployment environment has been shifting from traditional physical machines to virtual machines (VMs), and more recently to containers. Technology maturity is fueling the desire for a leaner, special purpose, and more secure applications and execution environments. We see the ideals of immutability[1] and minimization[2, 3] beginning to gain momentum.

This push towards application footprint reduction is at odds with a necessary dependency on application monitoring and management utilities. General-purpose operating environments offer various such utilities, for example *ps*, *top*, *gdb*, *strace*, *netcat*, *iostat*, *iperf*, etc. And, if not present, these utilities are easily accessible from standard package repositories to install and add to the application's runtime. Different IaaS cloud providers even offer VM image templates[4][5] baked with all common tools for better application management. With minimization approaches removing these auxiliary components from the container runtime, and the immutability ideal prohibiting their *installation*, application management has become highly challenging in the cloud.

In this paper, we propose a novel approach to address this growing challenge in the cloud. We present our disaggregated managed service delivery platform for container clouds, RECap. RECap is inspired from the fundamentals behind serverless computing [6, 7, 8] to deliver functionality on demand. We provide application monitoring and management tasks for lean, immutable container runtimes, without baking them into the container image. In RECap, we orchestrate the delivery of management functionality, on-demand to an application container, through a special-purpose, *dynamically-linked* sidecar container called Capsule. The **Cap**sule is attached to the container only for the duration of **R**unning the add-on functionality, after which it is detached or **E**scapes the application container.

We envision RECap as a value-added offering of a cloud platform, similar to the likes of load balancing, auto scaling, and migration [9]. In this context, and designed to operate on one of the most popular container cloud platforms, Kubernetes, RECap enables an on-demand managed service delivery for container clouds, where the micro-services stay lean and immutable, while the cloud platform takes on the responsibility of providing auxiliary services.

2 Motivation

With technology maturity, containers are being adopted as a lightweight virtualization alternative to VMs, with applications being on-boarded on *PaaS* container clouds, such as Amazon Container service[10], Google Container Engine[11] and IBM Container Service[12]. With increasing popularity, as is common with most new technologies, the container runtime is also undergoing transformation for betterment, as shown in Figure 1.

Following the principles of the microservice architecture, the first wave saw large monolithic applications being separated into multiple independent microservice containers. But, container images were still based on general-purpose operating runtimes like Ubuntu, Debain,

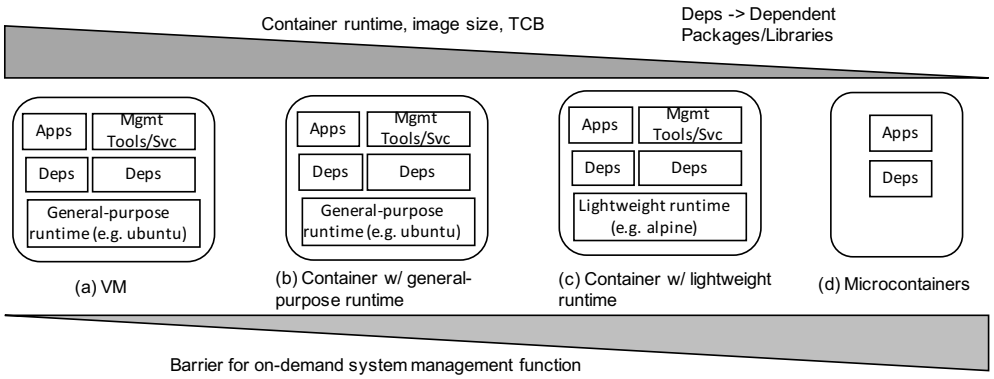

Figure 1: Application runtime evolution

CentOS, RHEL, etc. Thus, the runtime had most system management tools (and other unused software) available, or a package repository to pull those from. Basing container images off of general purpose OSes, however, created more challenges, not just for managing and securing containers, but also for the host platform[13]. The traditional concerns of package conflicts and vulnerability exposure with a large software footprint stayed the same, due to existence of several auxiliary packages borrowed from a general purpose OS. In addition, the then-fragile container boundary and its proximity to the host OS, meant that the large container footprint made the shared-host itself vulnerable to a breach from any of its guest containers.

The next wave saw application footprint reduction, with container images being based upon lightweight runtimes, like Alpine Linux[14]. This certainly helped reduce the size of images, e.g. a *nodejs* container image size reduction from 431*MB* to 66*MB* on the lightweight Alpine base runtime. However, the runtime still contained the secondary system management tools and their dependent packages, and thus the package vulnerability exposure still remained.

Recently, new techniques [3][15][16][17][18] emerged that perform static and dynamic analysis of applications to identify all the runtime resources (e.g. binaries, libraries, configuration files etc.) necessary for *only* the primary application to run. These techniques then create a *minimum* runtime image containing only those resources. For certain applications (e.g., new golang apps) only a single statically linked executable is created and stored inside an image. Such minimum runtimes are also defined as *microcontainers*. These techniques help reduce application runtime footprint by up to 95%[16]. The reduction in the package footprint, more importantly, reduces the TCB (Trusted Computing Base) to lower the vulnerability exposure.

To substantiate our claims regarding application runtime minimization trend, we survey all 136 official application images from public DockerHub[19] repository. Most of these applications had multiple runtimes identified with different *tags*, but in our study we considered only one image per application with default *latest* tag. As shown in the Figure 2, there are still about 75% images built from the base of general-purpose OS runtimes like debain, rhel etc. At the same time, there is steady movement in migrating to leaner runtimes like alpine (13%) and even microcontainers (12%) in form of scratch containers with minimal-runtime or a single statically built executable.

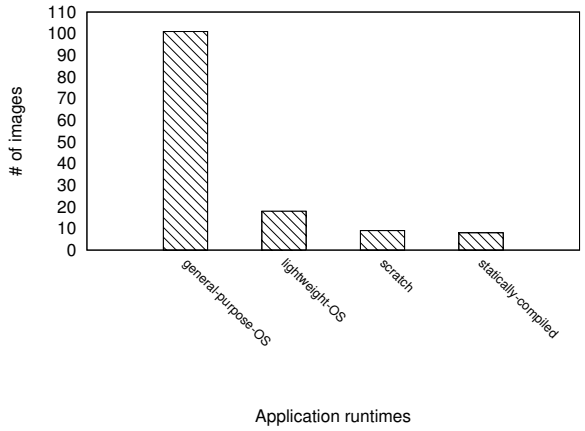

Figure 2: Docker Hub Image survey

On one hand, the path from general-purpose runtimes to microcontainers has led to leaner, and more secure application environment. On the other hand, it has made the management of the applications more challenging, by removing the corresponding utilities. Furthermore, the container immutability ideal also precludes the traditional *installation* method, and dictates a new image be built from scratch, each time a container is to be changed. This translates to creating a new container instance just to monitor the application!

We thus believe there is a need and an opportunity for the the cloud platform itself to deliver such capabilities on-demand. The analogy here is akin to the common traditional OS utilities for process monitoring, analysis, debugging and management, where they can be dynamically attached or applied to running pro-

cesses. Inspired from another emerging cloud programming model, serverless computing or Function-as-a-Service, in RECap we have designed a disaggregated managed service delivery framework for cloud, to provide common add-on functionalities for running containers, on demand.

3 Use Cases

The RECap framework is suitable for delivery of most common software functions inside containerized applications, such as:

System and Application Administration: Common system administrative tasks like malware scans, clock time corrections with *ntpupdate*, log rotations, etc., are performed periodically, and typically implemented as cron jobs on general-purpose runtimes. Similarly, applications themselves have various management tasks, for example, data storage applications performing periodic consistency checks with *MD*5, archival with *tar*, etc. Such administrative and management tasks can be easily de-coupled from the primary application runtime, and provided as on-demand functions by the cloud platform through RECap.

Debugging: At times, there is need to provide ad-hoc debugging capabilities into the application runtimes like *gdb, strace, tcpdump, iperf,* etc. On general-purpose platforms these capabilities are added by installing their packages from standard repositories. While on a container cloud platform, following standard DevOps practices, respective images of microservices need to be rebuilt with new capabilities, and re-deployed to make them available to use. On the other hand, using RECap such capabilities can be bundled into capsules, and delivered on-demand to the running application containers.

Monitoring: Tracking of system-level resource utilization metrics (CPU, memory, etc.), as well as application-level metrics (number of connections, requests, workers, etc.) for microservice containers, can also be provided as on-demand functionality over RECap.

Here, we have highlighted only the first few use cases of RECap. As our design matures we are adding new managed services for use cases including auditing, tracing, security and compliance among others.

4 System Design

We are designing RECap for Kubernetes which is one of the most most popular container cloud platforms. Figure 3 shows the overall architecture of RECap. We describe its key components below.

4.1 Capsule image creation

For every managed service RECap provides, first an image needs to be created encapsulating the operating runtime for that service. This image essentially contains the service executable, associated binaries, libraries and necessary configuration objects. Users can leverage all of the existing tools and practices (e.g., Dockerfile) for creating a Capsule image, with the only requirement being compliance with the standard OCI (Open Container Initiative) format. This is an important distinction compared to the functions provided by the existing serverless platform. In serverless platforms, the users can only provide function scripts (python, javascript, etc,) that are commonly statically bound to a trigger. In RECap , the scope of provided managed services are expanded to include dynamically-linked executables, which can bind to have actual live context of a running application. Therefore we require the Capsule image to contain the complete runtime for the provided service. We expect the Capsule ecosystem to evolve around the RECap platform, akin to the virtual appliance marketplaces that emerged around VM services. As the platform gains adoption, we expect common, ready-to-use service images to emerge over open registries or from cloud provider catalogues.

4.2 RECap deployment

Kubernetes supports different deployment resource *kinds* for microservices including Statefulset, Daemonset, Cronjobs etc. [20]. These deployment *kinds* define how microservices are deployed on the cloud substrate and managed throughout their lifecycle. We are are adding a new deployment *kind* to kubernetes, to facilitate deployment of the Capsule functions characterized by their following unique requirements:

1) Ensuring affinity: The Capsule container is to be deployed on the same node as the applicable microservice(s). Such affinity is facilitated through kubernetes *labels*. In the deployment manifest, matching labels are defined to establish association between the Capsule container and the microservice containers.

2) Shared namespace: To operate within the target guest's context, the Capsule container needs to share necessary Linux namespaces with the running microservice container. For this, we utilize kubernetes *pods*– the basic deployable units in kubernetes within which all containers share all namespaces. For RECap, we need to dynamically attach (Run) and detach (Escape) namespaces between the Capsule and the microservice container.

3) Scheduler: We currently define three scheduling or *exec* policies for a Capsule – (a) Periodic: these are similar to the kubernetes *cronjobs* except they are executed in-context of the respective microservices as opposed to

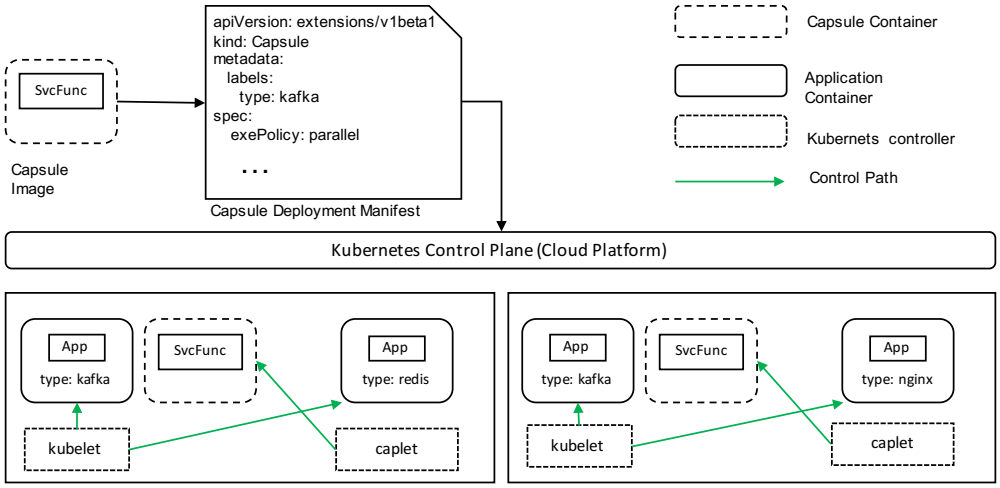

Figure 3: RunEscape Capsule Architecture

the global cluster context, (b) Parallel: these functions are executed simultaneously for all applicable microservices, e.g. running *iperf* client and server Capsule between two microservices, and (c) Iterative: these functions are applied to all applicable microservices in-order, e.g. *logrotation* done iteratively to avoid disk bottleneck.

4.3 RECap controller

A standard kubernetes cluster runs a *kubelet* as a primary "node agent" on every node, which ensures that the containers are started as per the deployment specification. However, as a design principle, a *kubelet* enforces immutability for deployment specification, and does not allow attaching a new container to an already running one. Therefore, we deploy a separate "node agent" on every node, specifically for managing the lifecycle of Capsule containers, called *Caplet*. The *Caplet* controllers are responsible for starting the Capsule containers against the running microservice containers (with the required namespace sharing), monitoring the execution of the Capsule functions, and finally safely removing the Capsule containers.

4.4 Application Security

In RECap, we adopt the following measures to ensure safety of application containers from the dynamically 'run-escape'-ing Capsules:

1) Capsule Image scan: As a secure DevOps practice for a container platform, every application image is scanned for security vulnerabilities and compliance violations[21][22][23], before it can be used to instantiate container(s). Since, a Capsule container shares the namespace(s) with an application container, any vulnerability in the Capsule will make the application equally vulnerable. Therefore, it is important to ensure that every Capsule image is subjected to the same scanning and validation practices as the applications themselves. In RECap, we enforce the standard OCI format for the Capsule images, therefore these images can be easily integrated with the existing DevOps processes.

Scenarios	Image Build	App Deploy	Docker Exec	Capsule RunEscape
Function exists in app container	0	0	0.083s	0
Function does not exist in app container	5.24s	0.29s	0.083s	0
Capsule image is present	0	0	0	0.243s
Capsule image is not present	6.2s	0	0	0.243s

Table 1: Evaluation of function delivery on RECap framework

2) Sandboxing Capsule container: It is important to secure the application against potentially malicious Capsules. In RECap, we leverage various container sandboxing techniques, to control the access and actions a Capsule can perform inside an application container. These include sharing only the required *namespaces*, and controlling system call interface through *seccomp* and system functions through Linux *capabilities*.

3) Safe sharing: One of the important features of RECap is the ability to share a single Capsule across multiple applications. Therefore, it is imperative to ensure that there is no information leak across multiple runs of the Capsule. Currently, we enforce this for persistent state only by using a Copy-on-Write (CoW) layer in the Capsule at the start of every run, and removing this layer while escaping the application container.

5 Evaluation

To validate RECap's correctness, we verified the output of various utilities, ranging from process listing to

vulnerability scanning, to match when run inside the Capsule and the application container.

Next, to evaluate the efficiency of the RECap framework, we consider a debugging use case wherein *tcpdump* function needs to be implemented inside a running *nginx* application container. Further, we define a new metric called *function latency* or $f_{latency}$ as total turn-around time for implementing any add-on system management function inside a running microservice on cloud. For evaluation we consider four possible scenarios as shown in Table 1.

First row of Table 1 represents the baseline scenario: the application container is built from a general-purpose runtime and the required service is available inside the container image. In this case, the service is simply triggered by the container from its context. This is performed via the standard `docker exec` command in our evaluation and takes only 83*ms* to complete.

As discussed in Section 2, for minimum-runtimes or microcontainers, these add-on functions are not available by default. Therefore, following DevOps principles, a new application image needs to built and redeployed with required new function, which then can be triggered by user with $f_{latency}$ of 5.6*s* (row 2). For different microservices their respective images need to be build independently, as a result each incurs the same $f_{latency}$.

On RECap platform, Capsule images are expected to be available for most common functions. During execution a Capsule container is created and attached to the running microservice with $f_{latency}$ of 243*ms* (row 3).

Although rare, but if not present, the Capsule image for *tcpdump* function has to be built with an additional build latency of 6.2*s* (row 4). Capsule images incur higher build latency because they are built from a small generic base image. So less of the dependencies exist within and need to be added as part of the build. Unlike the DevOps flow of row 2, a single Capsule image need to built only once and thereafter is shared between several microservices containers.

Overall, we observe that RECap can be used very efficiently for delivering add-on functionailty.

6 Related Work

The continuing maturity and adoption of containers in the cloud environment, provides unique opportunities for more efficient service delivery. In our previous work[9], we had argued making container applications and the cloud platform aware of each other for optimizing cloud management tasks such as migration and autoscaling. With RECap, we add to the cloud platform's arsenal of managed services.

Sidecars[24] is one such unique attribute of a container ecosystem, which enables augmenting containerized applications with auxiliary capabilities such as intelligent routing and load balancing[25], service discovery, registration and communication[26, 27], etc. In RECap, we use sidecars to deliver application management functionality, with one difference being the sidecar lifetime. While a regular sidecar is created and deleted with the primary application container, a Capsule is created on-demand, and attached to the application container only for the duration of executing its software function. Additionally, with regular sidecars, while a separate sidecar container is typically created for each application container, the Capsule for a given function can be shared between all the applicable containers (say, belonging to the same tenant). A separate Capsule is created only for a new software function.

Another unique proposition with containers in cloud is the serverless platform or Function-as-a-Service, which allows spawning functions in response to events[6, 7, 8]. This on-demand functionality aspect inspires our RECap solution to deliver management software to application (micro)containers (Section 2). WatchIT[28] also uses on-demand container instantiation, but they do so to control an IT admin's view of the host, by restricting the admin's actions inside a 'perforated' container. Motivated by the serverless delivery approach, Ali et al.[29] also propose using containers to implement new functionality on the host, instead of installing new packages directly. In RECap, we are enabling a more generic cloud-native framework for managed service delivery for microservices.

Finally, while host-level [30], container-tailored[31], or introspection-based[32][33] solutions are capable of providing some system-level tasks from outside the container context, such as resource and metrics monitoring, RECap covers a broader functionality spectrum, including even application-level management tasks. Additionally, RECap is designed to be more easily consumable in a container cloud setting, offering service delivery as a run-escape sidecar in a kubernetes environment.

7 Conclusion

We highlighted the need and opportunity for on-demand delivery of managed services, in the context of modern container clouds, driven by the impetus of immutability and runtime minimization philosophy. Towards this vision, we presented our serverless-computing-inspired RECap framework, to enable on-demand managed service delivery, via our Run-Escape Capsule sidecars as the delivery vehicle. We also discuss the design and operational model for integrating RECap into one of the most popular cloud substrate, i.e. Kubernetes and duly stated the security, manageability and performance trade-offs. In addition to monitoring, debugging, and system administration, we aim to add new managed services as our design matures, for use cases including auditing, security and compliance, amongst others.

References

[1] Major Hayden and Richard Carbone. Securing linux containers. *GIAC (GCUX) Gold Certification, Creative Commons Attribution-ShareAlike 4.0 International License*, 19, 2015.

[2] Travis Reeder. Microcontainers-tiny, portable docker containers, 2017.

[3] Docker-slim. `https://github.com/docker-slim/docker-slim`.

[4] Amazon Elastic Compute Cloud. Amazon web services. *Retrieved November*, 9:2011, 2011.

[5] Daniel Aguado, Thomas Andersen, Aram Avetisyan, Jeff Budnik, Mihai Criveti, Adrian Doroiman, Andrew Hoppe, Gerardo Menegaz, Alejandro Morales, Adrian Moti, et al. *A practical approach to cloud IaaS with IBM SoftLayer: Presentations guide*. IBM Redbooks, 2016.

[6] Amazon, Inc. `https://aws.amazon.com/lambda/`.

[7] IBM, Inc. `https://console.bluemix.net/openwhisk/`.

[8] Google, Inc. `https://cloud.google.com/functions/`.

[9] Shripad Nadgowda, Sahil Suneja, and Canturk Isci. Paracloud: bringing application insight into cloud operations. In *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)*. USENIX Association, 2017.

[10] Amazon EC2 Container Service. `https://aws.amazon.com/ecs/`.

[11] Container Engine. `https://cloud.google.com/container-engine/`.

[12] IBM Bluemix. `https://console.ng.bluemix.net/`.

[13] Byungchul Tak, Canturk Isci, Sastry Duri, Nilton Bila, Shripad Nadgowda, and James Doran. Understanding security implications of using containers in the cloud. In *USENIX Annual Technical Conference (USENIX ATC 17)*, pages 313–319. USENIX Association, 2017.

[14] Alpine Linux: Small. Simple, Secure. `https://www.alpinelinux.org/about/`.

[15] Katharina Gschwind, Constantin Adam, Sastry Duri, Shripad Nadgowda, and Maja Vukovic. Optimizing service delivery with minimal runtimes. In *Proceedings of the 15th International Conference on Service-Oriented Computing*, 2017.

[16] Vaibhav Rastogi, Chaitra Niddodi, Sibin Mohan, and Somesh Jha. New directions for container debloating. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, pages 51–56. ACM, 2017.

[17] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. Cimplifier: automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 476–486. ACM, 2017.

[18] Microcontainers Tiny, Portable Docker Containers. `https://blog.iron.io/microcontainers-tiny-portable-containers/`.

[19] Docker. Official Docker Repository. `https://hub.docker.com/explore/`.

[20] Kubernetes. Daemonset. `https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/`.

[21] IBM. Managing container and image security with vulnerability advisor. `https://console.bluemix.net/docs/containers/va/`, 2015.

[22] Amazon, Inc. Amazon Inspector: Automated security assessment service to help improve the security and compliance of applications deployed on AWS. `https://aws.amazon.com/inspector/`.

[23] Docker. Scan images for vulnerabilities. `https://docs.docker.com/datacenter/dtr/2.4/guides/user/manage-images/scan-images-for-vulnerabilities/`.

[24] Brendan Burns and David Oppenheimer. Design patterns for container-based distributed systems. In *HotCloud*, 2016.

[25] Istio. `https://istio.io`.

[26] AirBnb. SmartStack: Service Discovery in the Cloud. `https://medium.com/airbnb-engineering/smartstack-service-discovery-in-the-cloud-4b8a080de619`.

[27] Netflix. Prana. `https://github.com/Netflix/Prana/wiki`.

[28] Noam Shalev, Idit Keidar, Yaron Weinsberg, Yosef Moatti, and Elad Ben-Yehuda. Watchit: Who watches your it guy? In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 515–530. ACM, 2017.

[29] Ali Kanso and Alaa Youssef. Serverless: beyond the cloud. In *Proceedings of the 2nd International Workshop on Serverless Computing*, pages 6–10. ACM, 2017.

[30] Sysdig. Cloud FunctionsDocker Monitoring, Kubernetes Monitoring and more. `https://sysdig.com/opensource/`.

[31] Twistlock. `https://www.twistlock.com/`.

[32] Cloudviz. Agentless system crawler. `https://github.com/cloudviz/agentless-system-crawler`.

[33] Ricardo Koller, Canturk Isci, Sahil Suneja, and Eyal De Lara. Unified monitoring and analytics in the cloud. In *HotCloud*, 2015.