TITUS

Introducing Containers to the Netflix Cloud

ANDREW LEUNG, ANDREW SPYKER, AND TIM BOZARTH

APPROACHING CONTAINER ADOPTION IN AN ALREADY CLOUD-NATIVE INFRASTRUCTURE

In 2008 Netflix went all-in on cloud migration and began moving all of its internally hosted infrastructure to AWS (Amazon Web Services). Today almost all of Netflix runs on VMs (virtual machines) in AWS. A customer's catalog browsing experience, content recommendation calculations, and payments are all served from AWS.

Over the years Netflix has helped craft many cloud-native patterns, such as loosely coupled microservices and immutable infrastructure, that have become industry best practices. The all-in migration to the cloud has been hugely successful for Netflix. Despite already having a successful cloud-native architecture, Netflix is investing in container technology.

EMERGING CONTAINER TECHNOLOGY
Container technology enables Netflix to follow many of the same patterns already employed for VMs but in

a simpler, more flexible, and efficient way. Some of the factors driving this investment include:

End-to-end application packaging. Container images used for local development are identical (or at least very similar) to those that are run in production. This packaging allows developers to build and test applications more easily in a production-like environment, which improves reliability and reduces development overhead.

Flexible packaging. Netflix has historically provided a Java virtual machine (JVM)-oriented development and deployment environment, using a common VM image that application configurations are "baked" into. For non-JVM applications, configuring this image properly can be difficult. Container images provide an easy way to build application-specific images that have only what the application needs.

A simpler cloud abstraction. Deploying Netflix applications into virtual machines requires selecting an approximately right-sized VM instance type and configuring it to run and manage the application. Many factors affect which instance type is best, including hardware (e.g., CPU, memory, disk) dimensions, pricing, regional availability, and advanced feature support (e.g., specialized networking or storage features). For many developers, this is a confusing, machine-centric step that leaves opportunity for errors. Containers make this process easier by providing a more application-centric deployment that only calls for declaring the application's requirements.

Faster and more efficient cloud resources. Containers are lightweight, which makes building and deploying them

faster than with VM infrastructure. Additionally, since containers have only what a single application needs, they are smaller and can be packed more densely onto VMs, which reduces the overall infrastructure footprint.

These factors do not change the patterns or approaches to Netflix's existing cloud-native infrastructure. Instead, containers improve developers' productivity, allowing them to develop, deploy, and innovate faster. Containers are also emerging across the industry as the de facto technology to deploy and run cloud-native applications. Investing in containers ensures Netflix's infrastructure is aligned with key industry trends.

While the value to developer productivity drove much of the company's strategic investment, an important practical reason for investment in containers is that Netflix teams were already beginning to use them. These teams not only provided tangible evidence of how to benefit from containers, but also served to highlight the lack of internal container support.

Unique Netflix Container Challenges

In many companies, container adoption happens when building new greenfield applications or as part of a larger infrastructure refactor, such as moving to the cloud or decomposing a monolithic application into microservices. Container adoption at Netflix differs because it is driven by applications that are already running on a cloud-native infrastructure. This unique environment influenced how we approached both the technology we built and how we managed internal adoption in several ways:

➡ Since applications were not already being refactored,

it was important that they could migrate to containers without any significant changes.

➡ Since Netflix culture promotes bottom-up decisions, there is no mandate that teams adopt containers. As a result, we initially focused on only a few internal users and use cases that wanted to try containers and would see major benefits from adoption.

➡ We expect some applications to continue to run in VMs while others run in containers, so it was important to ensure seamless connectivity between them.

➡ Early container adoption use cases included both traditional microservices and a wide variety of batch jobs. Thus, the aim was to support both kinds of workloads.

➡ Since applications would be moving from a stable AWS EC2 (Elastic Compute Cloud) substrate to a new container-management layer running on top of EC2, providing an appropriate level of reliability was critical.

CONTAINERS IN AN EXISTING CLOUD INFRASTRUCTURE
Netflix's unique requirements led us to develop Titus, a container-management system aimed at Netflix's cloud infrastructure. The design of Titus focuses on a few key areas:

➡ Allowing existing Netflix applications to run unmodified in containers.
➡ Enabling these applications to easily use existing Netflix and AWS cloud infrastructure and services.
➡ Scheduling batch and service jobs on the same pool of resources.
➡ Managing cloud capacity effectively and reliably.

Titus was built as a framework on top of Apache Mesos[8],

a cluster-management system that brokers available resources across a fleet of machines. Mesos enabled us to control the aspects we deemed important, such as scheduling and container execution, while handling details such as which machines exist and what resources are available. Additionally, Mesos was already being run at large scale at several other major companies[7,12,14]. Other open-source container-management systems, such as Kubernetes[10] and Docker Swarm[6], which were launched around the time Titus was developed, provided their own ways of scheduling and executing containers. Given the specific requirements noted above, we felt that we would end up diverging from their common capabilities quickly enough to limit their benefits.

Titus consists of a replicated, leader-elected scheduler called Titus Master, which handles the placement of containers onto a large pool of EC2 virtual machines called Titus Agents, which manage each container's life cycle. Zookeeper[9] manages leader election, and Cassandra[11] persists the master's data. The Titus architecture is shown in figure 1.

Work in Titus is described by a job specification that details what to run (e.g., a container image and entry point), metadata (e.g., the job's purpose and who owns it), and what resources are required to run it, such as CPU, memory, or scheduling constraints (e.g., availability zone balancing or host affinity). Job specifications are submitted to the master and consist of a number of tasks that represent an individual instance of a running application. The master schedules tasks onto Titus agents that launch containers based on the task's job specification.

FIGURE 1: **TITUS ARCHITECTURE COMPONENTS**
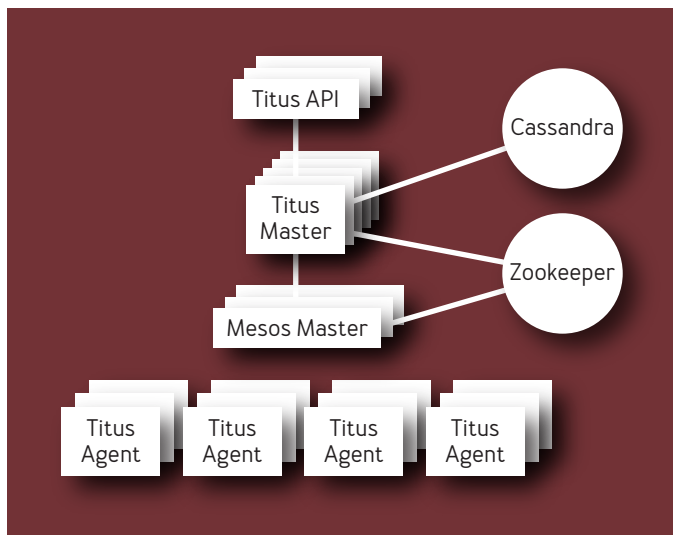
Designing for Easy Container Adoption

Most Netflix microservices and batch applications are built around parts of Netflix's cloud infrastructure, AWS services, or both. The Netflix cloud infrastructure consists of a variety of systems that provide core functionality for a Netflix application running in the cloud. For example, Eureka[18], a service-discovery system, and Ribbon[21], an IPC library, provide the mechanism that connects services. Atlas[16], a time-series telemetry system, and Edda[17], an indexing service for cloud resources, provide tooling for monitoring and analyzing services.

Many of these systems are available as open-source software[20]. Similarly, many Netflix applications use AWS

services such as S3 (Simple Storage Service) or SQS (Simple Queue Service).

To avoid requiring the applications using these services to change in order to adopt containers, Titus integrates with many of the Netflix cloud and AWS services, allowing containerized applications to access and use them easily. Using this approach, application developers can continue to depend on these existing systems, rather than needing to adopt alternative, but similar, infrastructure. This differs from other container-management systems that either provide their own or use new, container-specific infrastructure services[5].

Integrating with an Existing Cloud

In some cases, enabling access to Netflix cloud infrastructure systems through Titus was quite simple. For example, many of the Java-based platform service clients required only that Titus set specific environment variables within the container. Doing so automatically enabled usage of the distributed configuration service[15], the realtime data pipeline system[27], and others.

Other integrations required changes to the infrastructure services themselves to be able either to communicate with the Titus control plane (usually in addition to EC2) or to understand container-level data. For example, the Eureka client was updated to understand services registering from both an EC2 VM, as well as a Titus container. Similarly, the health-check polling system was changed to query Titus and provide health-check polling for containers in addition to VMs. The on-instance Atlas telemetry agent was changed to collect and emit

container-level system metrics (e.g., CPU and memory usage) from Titus agents. Previously, it collected only metrics for the entire host.

In addition to allowing Netflix applications to run in containers more easily, these integrations lowered the learning curve required to adopt containers within Netflix. Users and teams were able to use tools and processes they already knew, regardless of whether they were using VMs or containers. As an example, a team with existing Atlas telemetry dashboards and alerts could migrate their applications from VMs to containers, while keeping their telemetry and operations systems the same.

Integrating with the Netflix cloud infrastructure also allowed the Titus development team *not* to focus on rebuilding existing internal cloud components. In almost all cases, the effort to integrate with an existing Netflix service was far easier than implementing or introducing a new container-specific version of that service.

Rather than implement various deployment strategies in Titus, such as Red/Black or Rolling Upgrade, we chose to leverage Spinnaker[22], Netflix's continuous-delivery tool. Spinnaker provides the concept of a *cloud provider*, which allows it to orchestrate application deployments across Titus, as well as EC2. In addition to providing a familiar deployment tool on Titus, the use of Spinnaker allowed the Spinnaker team, which specializes in continuous delivery, to implement the logic of how to orchestrate deployments, while the Titus development team was able to focus on container scheduling and execution.

To be sure, there are aspects of the Netflix cloud that either work differently or do not work with Titus. By

integrating with existing Netflix components, however, rather than requiring that new ones be used, each of the integrations Titus provided served to lower the adoption curve incrementally for some teams and users.

Enabling AWS Integration

Another critical aspect of making container adoption easy is enabling usage of AWS services. Many Netflix applications are built around various AWS services such as S3 or SQS. Using AWS services requires the correct IAM (Identity and Access Management)[3] credentials to authorize service calls.

For applications running in EC2 VMs, identity and credential information is provided via instance metadata by a metadata service[4] that is available at a well-known IP address. This metadata service provides credentials at the granularity of an EC2 VM, which means that containerized applications on the same VM must either share the host's IAM credentials, which violates the principle of least privilege, or not use AWS services.

Titus uses a metadata service proxy that runs on each agent VM and provides containers with just their specific IAM credentials. Titus jobs declare the IAM role that is needed. When containerized applications make IAM requests to the metadata service IP, the proxy intercepts these requests via host-routing rules that redirect these requests to it.

The proxy extracts the container's IAM role from the task's configuration info that Titus provides and then uses the host's IAM *assume role* capability to acquire the specific IAM credentials and return them to the container.

The IAM *assume role* allows a principal's IAM role (in this case, the host's) to assume the identity capabilities of another principal temporarily (in this case, the container's). This approach provides containers with only *their* IAM credentials using the same IAM roles that are used in EC2 VMs. In addition to IAM roles, the proxy provides containers with Titus instance identity information instead of EC2 identity information. This information is used by client libraries such as the Eureka client.

A Common Network is Key

An important enabler for many of the integrations was a common networking infrastructure. Seamless network communication between containerized applications and existing infrastructure removed many integration and adoption hurdles.

A common solution to container networking is to provide an overlay network that creates a separate network on top of an existing one. This approach is appealing because it decouples the two networks and does not require changing the underlying one. However, an overlay segregates the containers' networking space from the existing network and requires gateways or proxies to connect them.

Another common approach is to allocate specific ports from the host's IP to containers. While this allows the container's IP to be part of the existing network IP space, it allows containers to use only specific ports that they must know upfront, limits colocating containers that use the same ports, and exposes the host's networking policies to the container. Additionally, applications and infrastructure

must handle container networking (ports) differently from how they handle VM networking (IPs). Since many Netflix systems are IP aware, but not port aware, retrofitting additional port data into the necessary systems would have been significant.

Titus provides a unique IP address to each container by connecting containers to the same AWS Virtual Private Cloud (VPC) network to which VMs are connected. Using a common VPC allows containers to share the same IP address space as VMs and use the same networking policies and features such as AWS SGs (Security Groups). This approach avoids the need to manage ports, as each container gets its own IP and full port range, and network gateways.

When launching a container that requested a routable IP address, Titus attaches an AWS ENI (Elastic Network Interface)[2] to the agent VM that is running the container. Attaching an ENI creates a new network interface on the VM from which multiple IP addresses can be assigned. These addresses are allocated from the same CIDR (classless inter-domain routing) range as VMs in the VPC, meaning containers and VMs can directly address each other's IPs. Allocating IPs via ENIs lets Titus avoid managing the available VPC IPs or directly modifying VPC routing tables.

When Titus is preparing to launch a new container, it creates a network namespace for it, assigns it a specific IP address from an ENI, and connects the container's network namespace to the host's using a veth (virtual Ethernet) interface. Routing rules on the host route all traffic for that IP to the veth interface, and routing rules within the

network namespace configure the allocated IP for the container.

Another benefit of containers sharing the same VPC network as VMs is that they use common network security policies, such as AWS Security Groups that provide virtual firewalls[1]. Each ENI can be configured to use a set of SG firewall rules that apply to any traffic coming in or out of the interface. Titus applies a container's requested SGs to the ENI with which that container is associated, allowing enforcement of SG firewall rules to its traffic.

The number of ENIs that can be attached to a VM is limited and potentially less than the number of containers that Titus could assign to it. To use ENIs more efficiently, Titus allows containers to share the same ENI. This sharing, however, is possible only when containers use the same security group configurations, as SGs can be configured only for the entire ENI. In this case, each container would have a unique IP address, with common firewall rules being applied to their traffic. An example of sharing an ENI is shown in figure 2. In this example, each of the three containers has a unique IP address, allocated from ENIs attached to the host. Containers 1 and 2, however, can route traffic through the same ENI because they both use only Security Group X.

Titus Master enables this sharing by treating SGs and ENIs as two-level resources so that it can schedule containers behind existing ENIs with the same SG configurations. Titus also provides guaranteed network bandwidth to each container via Linux traffic control. It sets a token bucket rate based on the bandwidth requested by the container. These networking features avoid changes

FIGURE 2: **TITUS CONTAINER IP CONFIGURATION**
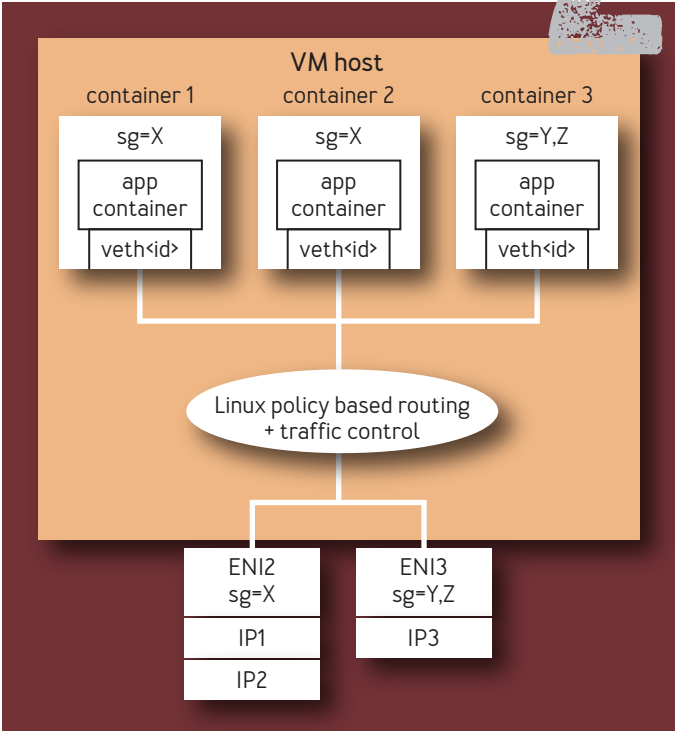
to applications migrating to containers and make an application running inside a container or VM transparent to external services.

Having a common networking infrastructure eased container adoption. External services that need to connect to an application do not have to care which technology the application is using. This transparency allows existing systems to work more easily with containerized applications and makes a hybrid environment with both VMs and containers more manageable.

Supporting Both Batch and Service Workloads

Early Netflix container use cases involved both batch-processing jobs and service applications. These workloads differ in that batch jobs are meant to run to completion and can have runtimes on the order of seconds to days, while services are meant to "run forever." Rather than managing these two different kinds of workloads with two different systems, container isolation enables these jobs to be colocated, which yields better combined cluster utilization and reduces operational burdens.

Since these two job types have different life cycles and management needs, Titus Master separates the role of *job management* from *task placement*. Job management handles the life cycle of each job type, such as a batch job's maximum runtime and retry policy or a service job's scaling policy. Task placement assigns tasks to free resources in the cluster and needs to consider only the task's required resources and scheduling constraints such as availability zone balancing.

For task placement Titus uses Fenzo[19], an extensible scheduler library for Mesos frameworks. Fenzo was built at Netflix and was already being used by an internal stream-processing system called Mantis[24]. Fenzo assigns tasks to resource offers presented by Mesos and supports a variety of scheduling objectives that can be configured and extended.

Titus uses Fenzo's bin packing in conjunction with its agent autoscaling capabilities to grow and shrink the agent pool dynamically as workload demands. Autoscaling the agent pool allows Titus to yield idle, already-purchased AWS Reserved Instances to other internal systems and

limit usage of AWS's more expensive on-demand pool. Fenzo supports the concept of a *fitness calculator*, which allows the quality of the scheduling decision to be tuned. Titus uses this feature to trade off scheduling speed and assignment quality.

While Titus Master is a monolithic scheduler, this decoupling is a useful pattern because it leverages some aspects of the two-level scheduler design[25]. Fenzo acts as a centralized resource allocator, while job managers allow decoupled management for different job types. This provides each job manager with a consistent strategy for task placement and agent management and needs them to focus only on the job life cycle. Other schedulers[26] have a similar separation of concerns, but Fenzo provides a rich API that allows job managers to support a variety of use cases and can potentially be extended to support job types with specialized needs.

Building a monolithic scheduler with separate job managers differs from other Mesos-based systems where different kinds of jobs are managed by different Mesos frameworks. In these cases, each framework acts as a full, independent scheduler for that job type. Titus is designed to be the *only* framework on the Mesos cluster, which avoids the need for resource locking and the resource visibility issues that can occur with multiple frameworks, and allows for task placement with the full cluster state available. Avoiding these issues helps Titus Master schedule more quickly with better placement decisions.

Heterogeneous Capacity Management
One of the benefits of using containers through Titus

is that it abstracts much of the machine-centric management that applications were doing in VMs. In many cases, users can tell Titus to "run this application" without worrying about where or on which instance type the container runs. Users still want some guarantees, however, around if or when their applications will run. These guarantees are particularly important when running applications with differing objectives and priorities. For example, a microservice would want to know it was capable of scaling its number of containers in response to increased traffic, even though a batch job may be consuming significant resources by launching thousands of tasks on the same cluster.

Additionally, applications running in EC2 VMs have become accustomed to AWS's concept of Reserved Instances that guarantee VM capacity if purchased in advance. To enable a more consistent concept of capacity between VMs and containers, Titus provides the concept of *tiers* and *capacity groups.*

Titus currently provides two tiers: one that ensures Titus Agent VMs are up and ready to run containers, and one that allows the agent pool to scale up and down as workload changes, as shown in figure 3. The chief difference between the two is the time offered to launch a container. The first tier, called the *critical tier,* is shown by the solid border in the figure. It enables Titus to launch containers immediately, without having to wait for EC2 to provision a VM. This tier optimizes around launch latency at the expense of running more VMs than the application may require at the moment.

The second tier, called the *flexible tier,* provisions

FIGURE 3: **THE CRITICAL AND FLEXIBLE TIERS**
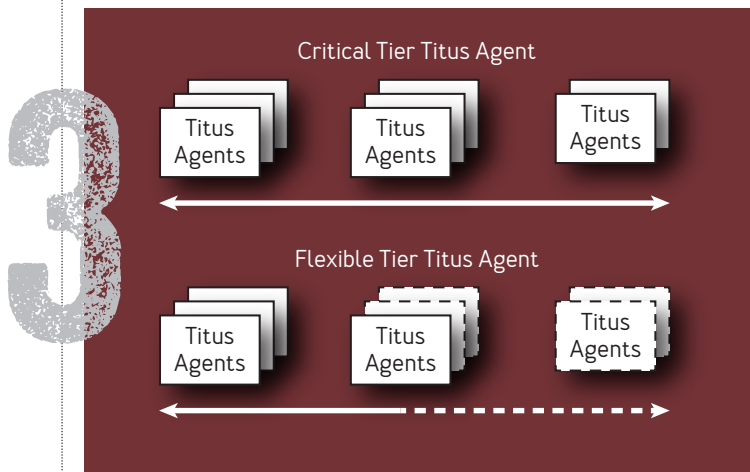
only enough agent VMs to handle the current workload
(though it does keep a small headroom of idle instances
to avoid overly aggressive scale-up and down). Scaling
the agent VMs in the flexible tier allows Titus to consume
fewer resources, but it can introduce launch latency to
tasks when an EC2 VM needs to be provisioned before the
container can be launched. Often the critical tier is used
by microservices that need their applications to scale up
quickly in response to traffic changes or batch jobs with
elements of human interaction—for example, when a user
is expecting a realtime response from the job. The number
of agents is scaled up and down as needed, shown by the
dotted border in the figure.

Capacity groups are a logical concept on top of each tier
that guarantee an application or set of applications some

amount of dedicated capacity. For example, a microservice may want a guarantee that it will have capacity to scale to match its peak traffic volume, or a batch job may want to guarantee some amount of task throughput. Prior to capacity groups, applications on Titus were subject to starvation if other applications consumed all cluster resources (often these starvations were caused by bugs in scripts submitting jobs or by users who did not consider the amount of capacity their jobs would consume). Additionally, capacity groups help users think about and communicate their expected capacity needs, which helps guide Titus's own capacity planning.

Combining capacity groups and tiers allows applications to make tradeoffs between cost (setting aside possibly unused agent resources for an application) and reliable task execution (ensuring an application cannot be starved by another). These concepts somewhat parallel the AWS concepts of Reserved Instances and On-Demand Instances. Providing similar capacity concepts eases container adoption by allowing users to think about container capacity in a similar way to how they think about VM capacity.

MANAGING CONTAINER ADOPTION
Beginning to adopt new technology is difficult for most companies, and Netflix is no different. Early on there were competing adoption concerns: either container adoption would move too quickly and lead to scale and reliability issues as Titus matured, or container adoption would be limited to only a few use cases and not warrant investment.

Despite these concerns and the lack of internal support,

a small set of teams were already adopting containers and realizing benefits. These early users provided concrete use cases where containers were solving real problems, so the initial focus was on their cases. We hypothesized that these early adopters would demonstrate the value of containers and Titus, while also allowing us to build a foundation of features that could be generalized for future use. The hope was that this approach would serve to let adoption happen organically and mitigate the concerns mentioned earlier.

These early teams ran a variety of ad hoc batch jobs and made sense as initial Titus users for several reasons. First, their use cases were more tolerant of Titus's limited availability and performance provided early on; a Titus outage would not risk the Netflix customer experience. Second, these teams were already using containers because their data-processing frameworks and languages made container images an easy packaging solution. Third, many users were data scientists, and the simplified interface appealed to their desire not to manage infrastructure. Other teams were also interested in Titus but were intentionally turned away because they were not good fits. These teams either would not see significant benefits from containers or had requirements that Titus could not easily meet at this stage.

The early users drove our focus on Netflix and AWS integrations, scheduling performance, and system availability that aided other early adopters. As we improved these aspects, we began to work on service job support. Early service adopters included polyglot applications and those where rapid development

iteration was important. These users drove the scheduler enhancements described earlier, integrations commonly used by services such as the automated canary-analysis system, and better end-to-end developer experience.

Titus currently launches around 150,000 containers daily, and its agent pool consists of thousands of EC2 VMs across multiple AWS regions. As usage has grown, so has the investment in operations. This focus has improved Titus's reliability and scalability, and increased the confidence that internal teams have in it. As a result, Titus supports a continually growing variety of internal use cases. It powers services that are part of the customer's interactive streaming experience, batch jobs that drive content recommendations and purchasing decisions, and applications that aid studio and content production.

FUTURE FOCUS AREAS
So far, Titus has focused on the basic features and functionality that enable Netflix applications to use containers. As more use cases adopt containers and as the scale increases, the areas of development focus are expected to shift. Examples of key areas where Netflix plans on investing are:

➡ While current container technologies provide important process-isolation mechanisms, they do not completely eliminate noisy neighbor interference. Sharing CPU resources can lead to context-switch and cache-contention overheads[28,13], and shared kernel components (e.g., the Network File System kernel module) are not all container-aware. We plan on improving the isolation Titus agents provide at both the userspace and kernel levels.

➡ For both batch and service applications, there are a number of advanced scheduler features that can improve their reliability and efficiency. For example, Titus currently does not reschedule a task once it is placed. As the agent pool changes or other tasks complete, it would be better for the master to reconsider a task's optimal placement, such as improving its balance across availability zones.

➡ In addition to more densely packing EC2 VMs, Titus can improve cloud usage by more intelligently using resources. For example, when capacity groups are allocated but not used, Titus could run preemptable, best-effort batch jobs on these idle resources and yield them to the reserved application when needed.

Similarly, Netflix brokers its already purchased but idle EC2 Reserved Instances among a few internal use cases[23]. Titus could make usage of these instances easier for more internal teams through a low-cost, ephemeral agent pool.

While only a fraction of Netflix's internal applications use Titus, we believe our approach has enabled Netflix to quickly adopt and benefit

Related articles

➡ Borg, Omega, and Kubernetes
Lessons learned from three container-management systems over a decade
Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes
http://queue.acm.org/detail.cfm?id=2898444

➡ Cluster-level Logging of Containers with Containers
Logging Challenges of Container-Based Cloud Deployments
Satnam Singh
http://queue.acm.org/detail.cfm?id=2965647

➡ Containers Push Toward the Mayfly Server
The container revolution represents a large-scale shift in thinking about multitasking systems.
Chris Edwards
https://cacm.acm.org/magazines/2016/12/210377-containers-push-toward-the-mayfly-server/fulltext

from containers. Though the details may be Netflix-specific, the approach of providing low-friction container adoption by integrating with existing infrastructure and working with the right early adopters can be a successful strategy for any organization looking to adopt containers.

References
1. AWS EC2 Security Groups for Linux instances; http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html.
2. AWS Elastic Network Interfaces; http://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/VPC_ElasticNetworkInterfaces.html.
3. AWS Identity and Access Management; https://aws.amazon.com/iam/.
4. AWS Instance metadata and user data; http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html.
5. Cloud Native Compute Foundation projects; https://www.cncf.io/projects/.
6. Docker Swarm; https://github.com/docker/swarm.
7. Harris, D. 2013. Airbnb is engineering itself into a data-driven company. Gigaom; https://gigaom.com/2013/07/29/airbnb-is-engineering-itself-into-a-data-driven-company/.
8. Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A.,

Joseph, A., Katz, R., Shenker, S., Stoica, I. 2011. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th Usenix Conference on Networked Systems Design and Implementation*: 295-308.

9. Hunt, P., Konar M., Junqueira F. P., and Reed, B. 2010. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference*, June 2010.

10. Kubernetes; http://kubernetes.io.

11. Lakshman, A., and Malik, P. 2009. Cassandra – a decentralized structured storage system. In *LADIS*, Oct. 2009.

12. Lester, D. 2015. All about Apache Aurora; https://blog. twitter.com/engineering/en_us/a/2015/all-about-apache-aurora.html.

13. Leverich, J., Kozyrakis, C. 2014. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the European Conference on Computer Systems.*

14. Mesosphere. 2015. Apple details how it rebuilt Siri on Mesos; https://mesosphere.com/blog/apple-details-j-a-r-v-i-s-the-mesos-framework-that-runs-siri/.

15. Netflix Archaius; https://github.com/Netflix/archaius.

16. Netflix Atlas; https://github.com/Netflix/atlas.

17. Netflix Edda; https://github.com/Netflix/edda.

18. Netflix Eureka; https://github.com/Netflix/eureka.

19. Netflix Fenzo; https://github.com/Netflix/Fenzo.

20. Netflix Open Source Software Center; https://netflix. github.io/.

21. Netflix Ribbon; https://github.com/Netflix/ribbon.

22. Netflix Spinnaker; https://www.spinnaker.io/.

23. Park, A., Denlinger, D., Watson, C. 2015. Creating your own EC2 spot market. Netflix Technology Blog; http://techblog.netflix.com/2015/09/creating-your-own-ec2-spot-market.html.

24. Schmaus, B., Carey, C., Joshi, N., Mahilani, N., Podila, S. 2016. Stream-processing with Mantis. Netflix Technology Blog; http://techblog.netflix.com/2016/03/stream-processing-with-mantis.html.

25. Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., Wilkes, J. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th European Conference on Computer Systems:* 351-364.

26. Vavilapalli, V. K., Murthy, A. C. Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., Baldeschwieler, E. 2013. Apache Hadoop YARN: yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, Article No. 5.

27. Wu, S., Wang, A., Daxini, M., Alekar, M., Xu, Z., Patel, J., Guraja, N., Bond, J., Zimmer, M., Bakas, P. 2016. Evolution of the Netflix Data Pipeline. Netflix Technology Blog; https://techblog.netflix.com/2016/02/evolution-of-netflix-data-pipeline.html.

28. Zhang, X., Tune, E., Hagmann, R., Jnagal, R., Gokhale, V., Wilkes, J. 2013. CPI2: CPU performance isolation for shared compute clusters. In *Proceedings of the European Conference on Computer Systems.*

Andrew Leung *(@anwleung) is a senior software engineer at Netflix, where he helps design, build, and operate Titus. Prior*

to Netflix, he worked at NetApp, EMC, and several startups on distributed file and storage systems. He received his Ph.D. from the University of California, Santa Cruz in 2009.

Andrew Spyker *(@aspyker) manages the Titus development team. His career focus has spanned functional, performance, and scalability work on middleware and infrastructure. Before helping with the cloud platform at Netflix, he worked as a lead performance engineer for IBM WebSphere software and the IBM cloud.*

Tim Bozarth *(@timbozarth) is a Netflix platform director focused on enabling Netflix engineers to efficiently develop and integrate their applications at scale. His career has focused on building systems to optimize for developer productivity and scalability at both Netflix and a range of startups.*