Docker on AWS

Running Containers in the Cloud

Brandon Chavis – Solutions Architect

Thomas Jones – Solutions Architect

April 2015

Abstract

AWS is a natural complement to Linux containers because of the wide range of scalable infrastructure services upon which containers can be deployed. AWS Elastic Beanstalk includes integrated support for Docker containers and Amazon EC2 Container Service (Amazon ECS) was designed from the ground up to manage Docker containers at scale. Amazon ECS is built upon several key features, such as cluster management and support for multiple container schedulers, to facilitate and orchestrate large-scale deployments of containers across managed clusters of Amazon EC2 instances.

Introduction

Prior to the introduction of containers, developers and administrators were often faced with a challenging web of complex compatibility restrictions, in which an application or workload was built specifically for its predetermined environment. If this workload needed to be migrated, for example from bare metal to a virtual machine, or from a VM to the cloud, or between service providers, then this generally meant rebuilding the application or workload entirely to ensure compatibility in the new environment. Containers were conceived as an answer to this matrix of restriction and incompatibilities by providing a common interface for migrating applications between environments.

With the release of Docker, interest in Linux containers has rapidly increased. Docker facilitates isolation of an application by packaging the application itself with its dependencies, configuration files, and interfaces into a single image that can then be moved seamlessly between different hosts, ensuring a consistent runtime environment between test and production versions of the application. Docker applies several user-level features to containers that make containers more accessible and useable to the community. For example, the Docker stack provides a CLI to manage containers, ways to provision containers declaratively or interactively, and a repository to host container images.

At its core, Docker is an open source project that uses several resource isolation features of the Linux kernel to sandbox an application, its dependencies, and interfaces inside of an atomic unit called a container. Because containers roll an application together with its dependencies and interfaces into a single re-deployable unit, a container can be run on any host system with the appropriate kernel components, while shielding the application from behavioral inconsistencies due to variances in software installed on the host. Multiple containers can be run on a single host OS without the need of a hypervisor while being isolated from neighboring containers. This layer of isolation introduces consistency, flexibility, and portability that enable rapid software deployment and testing.

Container Benefits

The rapid growth of Linux containers, and Docker specifically, is being fueled by the many benefits that containers provide. These benefits are apparent across an organization: from developers and operations, to QA. The primary benefits of Docker are speed, consistency, density, and flexibility.

Speed

Perhaps the most highly touted benefit of containers is speed. When highlighting the advantages of using Docker, it's impossible to avoid referencing speed at some point in the discussion. However, simply stating that Docker is fast sells the technology short, as speed is apparent in both performance characteristics and in application lifecycle and deployment benefits. These are the two major ways that the speed of containers can help developers, system administrators, and entire organizations to act quickly.

Firstly, because the architecture of containers allows for full process isolation by using the Linux kernel namespaces and cgroups, containers do not require a full virtualization stack or hypervisor to interface with host hardware. Containers simply share the kernel of the underlying host OS, and processes running in a container are unaware of their silo. This results in no discernable performance impact for applications running inside a container.

Secondly, containers can be created very quickly—they generally start in less than a second. The speed implications become quite apparent and beneficial in the development and deployment lifecycle; due to their lightweight and modular nature, containers can enable rapid iteration of your applications. Development speed is improved by the ability to deconstruct applications into smaller units. These smaller units are advantageous because of a reduction in shared resources between application components, ideally leading to fewer compatibility issues between required libraries or packages.

Launching a container with a new release of code can be done without significant deployment overhead. Operational speed is improved, because code built in a container on a developer's local machine can be easily moved to a test server by simply moving the container. At build time, this container can be linked to other containers required to run the application stack.

Consistency

The ability to relocate entire development environments by moving a container between systems highlights yet another major advantage of Docker: the consistency and fidelity of a modular development environment provides predictable results when moving code between development, test, and production systems. By ensuring that the container

encapsulates exact versions of necessary libraries and packages, it is possible to minimize the risk of bugs due to slightly different dependency revisions.

This concept easily lends itself to a “disposable system” approach, in which patching individual containers is less preferable than building new containers in parallel, testing, and then replacing the old. Again, this practice helps avoid “drift” of packages across a fleet of containers, versions of your application, or dev/test/prod environments; the end result is more consistent, predictable, and (ideally) stable applications.

Density and Resource Efficiency

The release of Docker and the enormous community support for the project over the last eighteen months signify another step in the direction of ever-increasing density and modularity of computing resources. Much like the invention of blade servers, the pervasiveness of virtual machines, or the industry shift to cloud computing, container technology promises increases in efficiency and agility through the ability to abstract applications further from underlying operating systems and hardware.

A common approach that highlights this change is allocation of a single service or application to a container. A one-service-to-one-container mapping allows for minimalist environments that are easy for ops teams to manage and for dev teams to upgrade. This approach also allows for modular deployments of a single service or application without affecting the other services running in different containers. For example, there is limited risk of changing a common dependency shared between two applications or restarting a shared application server.

Containers facilitate enhanced resource efficiency by allowing multiple containers to run on a single system. Resource efficiency is a natural result of the isolation and allocation techniques that containers use. Containers can easily be restricted to a certain number of CPUs and allocated specific amounts of memory. By understanding what resources a container needs and what resources are available to your VM or underlying host server, it’s possible to maximize the containers running on a single host, resulting in higher density, increased efficiency of compute resources, and less money wasted on excess capacity.

The ability to “right-size” an instance via container scheduling is now far more granular and can amplify the benefits of running on a flexible and scalable platform such as Amazon EC2. Perhaps instead of scaling up and down an instance, changing the number of containers assigned to a host could be the first step of achieving optimal utilization. One of the techniques that Docker uses to drive density is utilizing a copy-on-write file system, which is both efficient and fast because it simply creates a pointer to files that are shared by multiple containers. Copy-on-write creates a copy of a file only if it is changed.

Additionally, because containers are host-platform agnostic (so long as the host is a version of Linux running the necessary Docker software) and resource efficient, they allow for an increase in the density of applications when deployed on both VMs and physical servers without incurring the performance loss of additional overhead.

Flexibility

The flexibility of Docker containers is based on their portability, ease of deployment, and small size. Similar to Git, Docker provides a simple mechanism for developers to download and install Docker containers and their subsequent applications using the command `docker pull`. This is in contrast to the sometimes extensive installation instructions that other applications may require.

Because Docker provides a standard interface, it makes containers easy to deploy wherever you like, providing portability between different versions of Linux, your laptop, or the cloud. You can run the same Docker container on any supported version of Linux, assuming that you have the Docker stack installed on the host.

Containers also provide flexibility by making microservice architecture possible. In contrast to common infrastructure models in which a virtual machine runs multiple services, packaging services inside their own container on top of a host OS allows a service to be moved between hosts, isolated from failure of other adjacent services, and protected from errant patches or software upgrades on the host system.

Because Docker provides clean, reproducible, and modular environments, it streamlines both code deployment and infrastructure management. Docker offers numerous benefits for a variety of use cases, whether in development, testing, deployment, or production.

Containers in AWS

Amazon Web Services is an elastic, secure, flexible, and developer-centric ecosystem that serves as an ideal platform for Docker deployments. AWS offers the scalable infrastructure, APIs, and SDKs that integrate tightly into a development lifecycle and accentuate the benefits of the lightweight and portable containers that Docker offers to its users. In this section, we discuss two possibilities for container deployment in AWS: AWS Elastic Beanstalk, and Amazon ECS.

AWS Elastic Beanstalk

AWS Elastic Beanstalk is a management layer for AWS services like Amazon Elastic Compute Cloud (Amazon EC2), Amazon Relational Database Service (Amazon RDS), and Elastic Load Balancing. AWS Elastic Beanstalk removes the requirement of manually launching the AWS resources required to run your application. Instead, you upload your application while AWS Elastic Beanstalk handles the details of capacity provisioning, load balancing, scaling, and application health monitoring. You can define

and configure resources programmatically using AWS APIs, the `eb` command line tool, AWS CloudFormation, or the AWS Management Console.

Additionally, AWS Elastic Beanstalk offers the ability to deploy and manage containerized applications, and the command line utility `eb` can be used to deploy both the AWS Elastic Beanstalk environment and Docker containers. AWS Elastic Beanstalk can deploy containerized applications to Amazon ECS by provisioning the cluster and any other infrastructure components that your application may require. This means that you can simply specify which container images are to be deployed, CPU and memory requirements, port mappings, container links; and AWS Elastic Beanstalk places your containers across your cluster and monitors the health of each container. With AWS Elastic Beanstalk, you can easily deploy and scale containerized web applications and avoid the complexities of provisioning the underlying infrastructure.

In situations where organizations want to leverage the benefits of containers and want the simplicity of deploying applications to AWS by uploading a container image, AWS Elastic Beanstalk may be the right choice. If more granular control over containers or custom application architectures are needed, then consider working with Amazon ECS directly.

Amazon ECS

While AWS Elastic Beanstalk is useful for deploying a limited number of containers, a web application, or for the development lifecycle, the way to run and operate container-enabled applications with more flexibility and scale is by using Amazon EC2 Container Service (Amazon ECS). Amazon ECS is designed specifically to allow you to run and manage containers across a number of hosts that are grouped into clusters.

As the number of active containers being run increases, so does the management overhead involved. Creating, maintaining, and deploying containers, implementing and administrating cluster management software, monitoring your containers, and other undifferentiated heavy lifting can detract from your organization's ability to focus on your core business. This problem is solved with the release of Amazon ECS, a highly scalable, high performance container management service that supports Docker containers and allows you to easily run distributed applications on a managed cluster of EC2 instances.

Amazon ECS provides three unique capabilities for users that want to run Docker containers. First, there is no need to install, operate, and scale your own cluster management infrastructure. With simple API calls, you can launch and stop container-enabled applications and query the complete state of your cluster. Second, Amazon ECS is designed for use with other AWS services and includes access to many familiar features like Elastic Load Balancing, EBS volumes, EC2 security groups, and IAM roles. Third, Amazon ECS offers multiple ways to manage container scheduling, supporting a wide variety of applications.

Amazon ECS makes it easy to launch containers across multiple hosts, isolate applications and users, and scale rapidly to meet changing demands of your applications and users. Using Amazon ECS, you can now launch, manage, and orchestrate thousands of Linux containers on a cluster of EC2 instances, without having to build your own cluster management backend.

Clusters are made up of container instances, which are EC2 instances running the Amazon ECS container agent that communicates instance and container state information to the cluster manager and `dockerd` (the Docker daemon). Instances running the Amazon ECS container agent automatically register with the default or specified cluster. The Amazon ECS container agent is open source and freely available (https://github.com/aws/amazon-ecs-agent), and as such, can be built into any AMI intended for use with Amazon ECS. When a cluster is created using the Amazon ECS console, an Auto Scaling group is also associated with the cluster. This ensures that the cluster grows in response to the needs of the container workload.

After instances have been deployed to a cluster, a task definition is used to define the application or service to run by defining the containers and volumes that are deployed to a host. Task definitions consist of one or more container definitions, which specify the name and location of Docker images, how to allocate resources to each container, and any links and other placement constraints. You can also specify any volume requirements you might have. Running instances of a task definition are called tasks and are the minimum unit of work in Amazon ECS.

You can run tasks by using a scheduler. Amazon ECS includes two built-in schedulers or you can write your own or integrate with third-party schedulers.

Container-Enabled AMIs

AWS has developed a streamlined, purpose-built operating system for use with Amazon ECS. The Amazon ECS-Optimized Amazon Linux AMI includes the Amazon ECS container agent (running inside a Docker container), dockerd (the Docker daemon), and removes a number of packages that are not required for running containers. Because the Amazon ECS container agent has been released under the open source Apache 2.0 license anyone can build the agent into an AMI and deploy it to an ECS cluster. As the ecosystem around Amazon ECS develops, we anticipate container-optimized partner AMIs from RedHat, Ubuntu, and others. CoreOS has already announced support for Amazon ECS.

Container Management

Amazon ECS provides cluster management capabilities to offset the administration overhead of managing a large fleet of containers. For example, imagine running ten, twenty, or hundreds of containers on a single host, and having ten, twenty, or hundreds of hosts in a cluster. Docker deployments of this scale are not uncommon, and as container technology continues to mature and becomes more widely implemented, scale

and density of deployments will continue to increase.

Instead of requiring organizations to learn about a container management suite and writing drivers or other components to interface with their applications, Amazon ECS provides optimal control and visibility over your containers, your cluster, and ultimately your applications with a simple, detailed API.

The following API operations are available for the core components of Amazon ECS (clusters, container instances, task definitions, and tasks):

```
CreateCluster
DescribeCluster
ListClusters
DeleteCluster
RegisterContainerInstance
DescribeContainerInstance
DeregisterContainerInstance
RegisterTaskDefinition
DescribeTaskDefinition
ListTaskDefinitions
DeregisterTaskDefinition
CreateService
DeleteService
DescribeService
ListService
UpdateService
RunTask
StartTask
DescribeTask
ListTasks
StopTask
```

These actions eliminate the need to build and scale your own cluster and container management solutions. You no longer have to worry about the availability of your master node or your state management solution; call the relevant actions to carry out management tasks that scale with the size of your cluster.

Scheduling

Amazon ECS is a shared state, optimistic concurrency system and provides very flexible scheduling capabilities. Amazon ECS schedulers leverage cluster state information provided by the Amazon ECS API actions to make appropriate placement decision. Currently, Amazon ECS provides two scheduler options. The RunTask action randomly distributes tasks across your cluster and tries to minimize that possibility that a single instance on your cluster will get a disproportionate number of tasks. The service scheduler, CreateService, is ideally suited to long-running stateless services. The service scheduler ensures that an appropriate number of tasks are constantly running and reschedules tasks when a task fails (e.g., if the underlying container instance fails for some reason). The service scheduler can also make sure that tasks are registered against one or more load balancers.

Containers can be mapped to a load balancer using a service. Services provide Elastic Load Balancing integration to distribute incoming traffic across containers and reduce the overhead of tracking which instance hosts which container. Services optionally provide fault-tolerance by checking the status of running tasks, restarting them if they have failed, and re-associating them to their load balancer. Services also let you deploy updates, such as changing the number of running tasks or the task definition version that should be running.

In addition to providing a set of default schedulers, Amazon ECS allows for integration with both custom schedulers and existing third party schedulers. As a proof of concept, the ECSSchedulerDriver has been written to demonstrate integration of Amazon ECS with a third-party scheduler; in this case, the Apache Mesos Framework. The Amazon ECS API actions function as primitives for development of additional custom schedulers. The List and Describe actions can obtain information about the state of a cluster, while the StartTask action can place tasks on appropriate resources based on business and application requirements.

Container Repositories

Amazon ECS is repository-agnostic, so customers can use a repository of their choice. Some repositories, such as Docker Hub, also support integration with Github or Bitbucket for automatic builds of containers when changes are pushed to your codebase. Amazon ECS can integrate with private Docker repositories, including Docker Hub Enterprise, running in AWS or an on-premises data center.

Logging and Monitoring

Amazon ECS supports monitoring of the underlying EC2 instances in a cluster using Amazon CloudWatch.

Storage

Amazon ECS allows you to store and share information using data volumes. Data volumes are useful as a persistent data store that can be shared between multiple containers on a host, as empty, non-persistent scratch space for containers, or as an exported volume from one container to be mounted by other containers. ECS task definitions allow you to reference the location of an appropriate location on the host (either on instance storage or using EBS volumes). You can then reference the volume from specific container definitions. You can also let Docker manage the volume and share volumes between containers.

Volumes have an optional `sourcePath` parameter that references a directory on the underlying host, and a `containerPath` parameter that references the mountpoint inside the container. From within the container, writes (for example) toward the `containerPath` value are persisted to the underlying volume defined in `sourcePath`.

If a `sourcePath` value is not provided, Docker treats the defined data volumes as scratch space, and the data is not persisted past the life of the container. This is useful if multiple containers in a task need to reference a common temporary cache for the duration of their task.

Data volumes can also define the relationship of storage between two containers by using the `volumesFrom` parameter. In this configuration, container “A” can have a data volume defined on the underlying host, and other containers can reference the data volume presented through container “A”. The `volumesFrom` parameter requires a `sourceContainer` argument to specify which container's data volume should be mounted. The mountpoint for the exported volume is defined by the container that is exporting the shared volume.

Networking

Amazon ECS allows you to take advantage of native Docker features like port mapping and container linking, while building on host-level Amazon EC2 networking features such as security groups and IP addresses.

Amazon ECS *describe** API actions facilitate service discovery so that applications in a cluster can easily find one another as they are started, stopped, or moved.

Docker links are also supported by Amazon ECS, including support for injected environment variables and /etc/hosts.conf entries to allow for simple discovery of other linked containers. For more advanced users, it is possible to specify the exact security group, network interface, and IP address resources that should be used by a container.

Container Security

Running Docker containers within the execution environment of customer-controlled EC2 instances builds on the familiar Amazon EC2 isolation frameworks such as AWS Identity and Access Management (IAM), security groups, and Amazon Virtual Private Cloud (Amazon VPC).

AWS customers maintain control over the Docker daemon itself, the operating system and the underlying EC2 instance when using Amazon ECS. Customers also have control via the AWS deployment & management service family as well as through native configuration via the tooling of their choice. This operational model enables customers to leverage the isolation capabilities of Docker containers and other software isolation frameworks (such as iptables, SELinux, and AppArmor) in conjunction with the underlying AWS security controls in order to meet their particular security, risk and compliance requirements. For example, customers can provision EC2 instances with different configurations in order to satisfy security, segregation or functional requirements.

Amazon ECS helps scale this approach through the concept of clusters, which act as a placement boundary for the execution of any given task definition. This relationship enables customers to provision different sets of security configurations and isolation frameworks (such as assignment to different VPCs or EC2 instances) seamlessly and assign them to different sets of task definitions, even as the number of running tasks and container instances scales elastically.

Container Use Cases

There are many ways in which using containers on AWS can benefit your organization.

Batch Jobs

Containers can be used for Batch and ETL (Extract, Transform, Load) jobs by packaging the job into a container and deploying the container into a cluster. It is possible to improve the resource utilization of the cluster by running different versions of the same job or multiple jobs on the cluster. By running job containers on an existing cluster, the jobs start quickly; it is possible to grow capacity dynamically in response to demand. Batch and ETL job capacity can be shared with other processes such as application containers, taking advantage of fluctuations in demand on the existing cluster or using Amazon EC2 Spot pricing to service the load.

Distributed Applications

Containers are an excellent building block for distributed applications and microservice architecture. Distributed applications use design patterns of loose coupling, elasticity,

message passing, and queuing to scale when required. The strengths of containers (speed, density, consistency, and flexibility) help enable distributed application deployments by allowing for quick provisioning and updates across a cluster of heterogeneous servers. Microservice architecture goes one step further by breaking each service down into its own encapsulated unit; containers are a perfect match for this encapsulation.

Batch processing is another area where containers make a lot of sense. It is quick and easy to start up a large quantity of containers to process a batch of jobs and then stop them after the processing is complete.

Amazon ECS provides an easy way to manage a large cluster of instances and containers programmatically. With Amazon ECS, it is possible to launch thousands of containers in seconds, allowing rapid scaling and iteration. The cluster can support a mix of applications—for example, both short and long running tasks on the same EC2 instance. Amazon ECS is designed to enable provisioning of new containers into the cluster in the most efficient way. If more control is required, this behavior can be overridden by using the Amazon ECS API, which also allows for integration with other tools—for example, connecting Amazon ECS to your current software delivery process.

Continuous Integration and Deployment

Containers are a keystone component of continuous integration (CI) and continuous deployment (CD) workflows. Because containers can be built programmatically using Dockerfiles, containers can be automatically rebuilt any time a new code revision is committed.

Immutable deployments are natural with Docker. Each deployment is a new set of containers, and it's easy to roll back by deploying containers that reference previous images. Automation simplifies frequent deployments. Amazon ECS provides API actions that make deployments easy by providing the complete state of the cluster and the ability to deploy containers using one of the built-in schedulers or a custom scheduler.

Amazon ECS can leverage existing CI features in tools like GitHub, Jenkins, and Docker Hub while adding the ability to deploy images to your Amazon ECS clusters automatically and continuously.

Docker Hub supports CI through automated builds. Automated builds leverage the combination of a Dockerfile that specifies how the container should be built, and a GitHub repository that specifies which code should be deployed in the container. Using a GitHub service hook, Docker Hub can monitor the repository for new commits of code and take a specified action—usually deploying the code into a freshly built container.

Amazon ECS includes the ability to create services and manage long-running applications. A task definition defines the containers to run, while a service consists of a

task definition, a count of how many tasks to run, and optionally the Elastic Load Balancing load balancers associated with the running containers. Amazon ECS then ensures that the correct number of tasks are running, auto-recovers stopped tasks, and registers the EC2 instances with the load balancers. In a CI model, the container definition references a specific image that is tagged by the build pipeline. Each task definition is idempotent; when there is a new image, you can create a new task definition. You can then deploy updates by updating the service's task definition and Amazon ECS stops tasks running the old task definition and start tasks with the new task definition.

Because of how easy Amazon ECS makes deploying new containers, it is a perfect fit for CI/CD tools that exist in the Docker ecosystem today.

Platform as a Service

Platform as a service (PaaS) is generally defined as a set of software, tools, and the underlying infrastructure packaged as a complete unit. In this type of service model, customers have a common base stack upon which they can deploy their application, which may remove much of the overhead of purchasing, licensing, and deploying individual components of the stack. Typical offerings in a PaaS stack might include libraries, middleware, databases, and the underlying OS, which are provided by the PaaS organization; the consumer usually performs resource configuration.

For PaaS providers, there may be significant challenges in quickly deploying large numbers of identical resources in a modular fashion, isolating end users from each other, and integrating new iterations of platform offerings into production. Because containers inherently facilitate isolation of resources and users, allow for extremely efficient utilization of resources, and make creating and deploying template resources a simple process, containers are a logical choice for a PaaS offering.

Each individual product offered by the PaaS provider could be built into its own container, and then deployed on demand with very little boot time in response to a customer need. Amazon ECS can take much of the undifferentiated heavy lifting off the shoulders of the PaaS provider by handling the scheduling and cluster management. Because Amazon ECS leverages the same physical datacenters that power Amazon EC2, the PaaS provider only has to worry about deploying the products that define their platform offerings, while leaving the infrastructure and container management to AWS.

Architectural Considerations

When a task is built, it is important to keep in mind that all containers defined in the task are placed onto a single instance in the cluster. Because of this, it's logical that a task represents a single contiguous application that has multiple tiers requiring inter-container communication. A LAMP stack that runs a particular application should have all

components—an Apache container, a PHP container, and a MYSQL container—defined in the same task. Tasks give users the ability to allocate resources to containers, so containers can be evaluated on resource requirements and collocated to use available compute and memory resources fully. CPU resources are allocated in units of CPU cores, where 1024 units represent a single core. Memory can be allocated on a per megabyte basis. This granularity allows balancing of host resources and container requirements.

Amazon ECS provides three API actions for placing containers onto hosts: StartTask, RunTask, and CreateService. The RunTask action uses Amazon ECS scheduler logic to place a task on an open host. If more control is needed, use the StartTask action, as this allows a specific cluster instance to be passed as a value in the API call.

The CreateService action allows for the creation of a Service object, which is the combination of a TaskDefinition object and an existing Elastic Load Balancing load balancer. By creating a Service object, multiple instances of application stacks can be deployed and associated with load balancers. A container's host port mapping can be mapped directly to an Elastic Load Balancing instance listener port. Multiple containers can be mapped to the same load balancer via different listeners. For example, to expose both an Apache container listening on host port 80 and a Tomcat container listening on host port 8080 through the same load balancer, two listeners need to be created and mapped to the container's host ports. There is a fixed relationship between container ports and listeners.

Because containers cannot use links to communicate across host (instance) boundaries, there can be challenges with advertising internal container state, such as current IP address and application status, to other containers running on separate hosts within the cluster. In a dynamic container cluster, containers may frequently change state or obtain a new IP address and port mappings as a result of termination and re-instantiation. The concept of “service discovery” exists with the intention of addressing these challenges. At a high-level, service discovery patterns generally involve containers updating a common key-value store with information about their state (e.g., available or running), the port they are currently listening to, and their current IP address. The Amazon ECS *describe* API actions can serve as primitives for service discovery functionality. For example, the *describe-service* action returns the load balancer used for a service, which can provide a simple service discovery method.

Walkthough

This walkthrough helps you create your first cluster, register a task definition, and deploy a container. The following steps guide you through launching a simple NGINX webserver inside an ECS container using an ECS task and a community dockerfile.

NOTE: For all preliminary setup requirements, such as registering an AWS account, configuring IAM, setting up a VPC, and installing the CLI, see the Setting Up topic in the *Amazon EC2 Container Service Developer Guide*.

For more information about any of the specific commands, see the Amazon EC2 Container Service API Reference.

1) Create a new cluster:

Every account is automatically given a default cluster (named "default"). However, you can create your own with the `create-cluster` command:

```
$ aws ecs create-cluster --cluster-name Walkthrough
{
    "cluster": {
        "clusterName": "Walkthrough",
        "status": "ACTIVE",
        "clusterArn":
"arn:aws:ecs:region:aws_account_id:cluster/Walkthrough"
```

Note: By default, each AWS account is limited to two clusters.

2) Add instances:

Next, launch instances into your cluster. If you simply launch a container-optimized AMI and do nothing else, the instance automatically registers to the default cluster.

If you would like to control which cluster the instances register to, you need to input UserData to populate the cluster name into the /etc/ecs/ecs.config file. For more information, see the Running Commands on Your Linux Instance at Launch topic in the *Amazon EC2 User Guide for Linux Instances*.

Paste the following into the userdata field when you launch your instances:

```
#!/bin/bash
echo ECS_CLUSTER=your_cluster_name >> /etc/ecs/ecs.config
```

For this walkthrough, you will launch a webserver so it is important to configure the correct security group permissions and allow inbound access from anywhere on port 80.

3) Run a quick check:

After a few moments, your instances should have registered to the cluster. To check, use the `list-container-instances` command:

```
$ aws ecs list-container-instances --cluster Walkthrough
{
    "containerInstanceArns": [
        "arn:aws:ecs:region:accountid:container-
instance/54353373-c32c-463e-96d9-0ac7db5f6a9d",
        "arn:aws:ecs:region:accountid:container-
instance/9235f233-d23e-4c36-9130-81d072b2d630"
    ]
}
```

To dig into the instances more, you can also describe the instances using the `describe-container-instances` command:

```
$ aws ecs describe-container-instances --cluster default --
container-instances container_instance_UUID

$ ecs describe-container-instances --cluster Walkthrough --
container-instances 9235f233-d23e-4c36-9130-81d072b2d630
{
    "failures": [],
    "containerInstances": [
        {
            "status": "ACTIVE",
            "registeredResources": [
                {
                    "integerValue": 1024,
                    "longValue": 0,
                    "type": "INTEGER",
                    "name": "CPU",
                    "doubleValue": 0.0
                },
                {
                    "integerValue": 996,
                    "longValue": 0,
                    "type": "INTEGER",
                    "name": "MEMORY",
                    "doubleValue": 0.0
                },
                {
```

```
                    "name": "PORTS",
                    "longValue": 0,
```

```
                    "doubleValue": 0.0,
                    "stringSetValue": [
                        "2376",
                        "22",
                        "51678",
                        "2375"
                    ],
                    "type": "STRINGSET",
                    "integerValue": 0
                }
            ],
            "ec2InstanceId": "i-c5690f29",
            "agentConnected": true,
            "containerInstanceArn": "arn:aws:ecs:us-east-
1:accountid:container-instance/9235f233-d23e-4c36-9130-
81d072b2d630",
            "remainingResources": [
                {
                    "integerValue": 1024,
                    "longValue": 0,
                    "type": "INTEGER",
                    "name": "CPU",
                    "doubleValue": 0.0
                },
                {
                    "integerValue": 996,
                    "longValue": 0,
                    "type": "INTEGER",
                    "name": "MEMORY",
                    "doubleValue": 0.0
                },
                {
                    "name": "PORTS",
                    "longValue": 0,
                    "doubleValue": 0.0,
                    "stringSetValue": [
                        "2376",
                        "22",
                        "51678",
                        "2375"
                    ],
                    "type": "STRINGSET",
                    "integerValue": 0
                }
            ]
        }
```

```
    ]
}
```

4) Register the task:

Before you can run a task on your ECS cluster, you must register a task definition. Task definitions are lists of containers grouped together. For more information about the available task definition parameters, see Amazon ECS Task Definitions in the *Amazon EC2 Container Service Developer Guide*.

The following example creates a JSON file with the task definition and names it nginx_task.json. This specific task launches a pre-configured NGINX container from the Docker Hub repository:

```
{
“containerDefinitions”:[
  {
    "image": "dockerfile/nginx",
    "name": "nginx",
    "cpu": 1000,
    "memory": 500,
    "essential": true,
    "links": [
    ],
    "entryPoint": [
      "/bin/sh"
    ],
    "environment": [
    ],
    "portMappings": [
      {
        "containerPort": 80,
        "hostPort": 80
      }
    ]
  }
],
“family”: “whitepaper”
}
```

After creating the task definition file, register it with Amazon ECS:

```
$ aws ecs register-task-definition --family Walkthrough --
container-definitions file://$HOME/nginx_task.json
{
    "taskDefinition": {
        "taskDefinitionArn": "arn:aws:ecs:us-east-
1:accountid:task-definition/Walkthrough:1",
        "containerDefinitions": [
            {
                "environment": [],
                "name": "nginx",
                "links": [],
                "image": "dockerfile/nginx",
                "essential": true,
                "portMappings": [
                    {
                        "containerPort": 80,
                        "hostPort": 80
                    }
                ],
                "entryPoint": [
                    "/bin/sh"
                ],
                "memory": 500,
                "cpu": 1000
            }
        ],
        "family": "Walkthrough",
        "revision": 1
    }
}
```

5) Run the task:

To run your task on your cluster, take note of the taskDefinition instance value (Walkthrough:1) returned after task registration in the previous step. The instance of the task definition is called by `run-task`. You can also obtain the ARN by using the `aws ecs list-task-definitions` command.

```
$ aws ecs run-task --cluster Walkthrough --task
Walkthrough:1 --count 1

{
```

```
    "tasks": [
        {
```

```
            "taskArn": "arn:aws:ecs:us-east-
1:accountid:task/c79b8d02-1969-40c0-be7b-a9c0261104d2",
            "overrides": {
                "containerOverrides": [
                    {
                        "name": "nginx"
                    }
                ]
            },
            "lastStatus": "PENDING",
            "containerInstanceArn": "arn:aws:ecs:us-east-
1:accountid:container-instance/54353373-c32c-463e-96d9-
0ac7db5f6a9d",
            "desiredStatus": "RUNNING",
            "taskDefinitionArn": "arn:aws:ecs:us-east-
1:accountid:task-definition/Walkthrough:1",
            "containers": [
                {
                    "containerArn": "arn:aws:ecs:us-east-
1:accountid:container/d31b241d-8166-4f12-b900-
c1187bed3c1e",
                    "taskArn": "arn:aws:ecs:us-east-
1:accountid:task/c79b8d02-1969-40c0-be7b-a9c0261104d2",
                    "lastStatus": "PENDING",
                    "name": "nginx"
        }
      ]
    }
  ]
}
```

6) Test your container:

Lastly, test that your container is responding. Because you mapped the container port 80 to the instance port 80, you can use the curl utility to test the public IP address of the instance and get a response.

```
$ curl -v 54.173.144.196 >> /dev/null
* Rebuilt URL to: 54.173.144.196/
* Hostname was NOT found in DNS cache
```

```
*   Trying 54.173.144.196...
  % Total    % Received % Xferd  Average Speed   Time
Time     Time  Current
```

```
                                 Dload  Upload   Total
Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:-
-:-- --:--:--     0* Connected to 54.173.144.196
(54.173.144.196) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.38.0
> Host: 54.173.144.196
> Accept: */*
>
< HTTP/1.1 200 OK
* Server nginx/1.6.2 is not blacklisted
< Server: nginx/1.6.2
< Date: Wed, 17 Dec 2014 19:33:33 GMT
< Content-Type: text/html
< Content-Length: 867
< Last-Modified: Wed, 17 Dec 2014 17:06:20 GMT
< Connection: keep-alive
< ETag: "5491b80c-363"
< Accept-Ranges: bytes
<
{ [data not shown]
100   867  100   867    0     0   6546      0 --:--:-- --:-
-:-- --:--:--  6568
* Connection #0 to host 54.173.144.196 left intact
```

Conclusion

Using containers in conjunction with AWS can accelerate your software development by creating synergy between your development and operations teams. The efficient and rapid provisioning, the promise of “build once, run anywhere”, the separation of duties via a common standard, and the flexibility of portability that containers provide offer advantages to organizations of all sizes. By providing a range of services that support containers along with an ecosystem of complimentary services, AWS makes it easy to get started with containers while providing the necessary tools to run containers at scale.

Further Reading

For more information, see the following resources:

- Amazon EC2 Container Service documentation (http://aws.amazon.com/documentation/ecs/)
- Docker (www.docker.com)
- Deploying AWS Elastic Beanstalk Applications from Docker Containers (*AWS Elastic Beanstalk Developer Guide*)
- CoreOS Linux AMI (AWS Marketplace)
- ECS Mesos Scheduler Driver

Notices

Document Revisions

Version 1.0 – initial release

Appendix A: Supported Linux Versions

Amazon is a contributor to the official Docker Repos Program. At the time of publication, Amazon Linux 2014.03 includes the docker-0.9.0 package, and the most up-to-date versions are available in our repositories. Other Linux distributions in AWS support Docker as well; Docker is included with RHEL 7 and is available for RHEL 6 in the Extra Packages for Enterprise Linux (EPEL) repositories. CoreOS, a minimal Linux OS designed specifically for Docker, is available in the AWS Marketplace.