Deploying Containerized Applications in Amazon Elastic Container Service
Table of Contents

Summary

Development Environment

Tool/Service Selection
  - Docker
  - Jenkins
  - Nexus
  - Terraform/CloudFormation Templates
  - ECR
  - IAM
  - CloudWatch
  - EC2
  - ECS

Reference Architecture
  - Base Images
  - Dockerfiles
  - Vagrant Environment
  - ECR Creation and Initial Seeding
  - VPC Creation
  - ECS Cluster Creation
  - Cloud Formation Templates
  - CI/CD Automation
  - Updating Applications
  - Redeployment of the POC

Moving to Production
  - Cloud Migration Strategy
  - Application Migration
  - VPC Architecture
  - Scaling Considerations
  - Capacity Planning
  - Managing Images
  - Change Management

Recommendations
Summary

This case study is intended to provide a summary of a POC where Shadow-Soft helped a client deploy containerized applications in Amazon Elastic Container Service. It will cover the background of the project, tool selection, architecture, and implementation of the POC. It will also look at suggestions for migrating applications to Amazon Web Services (AWS).

Our client, a mid-market bank, has an initiative to migrate some on-premise applications to the AWS cloud. There is a significant push to improve the current Continuous Deployment / Delivery process on-premise today.

The case study will review the process for provisioning bare metal systems and explore tools that will help automate this process. Learn how to define infrastructure as code and identify tools used to deploy the applications on-premise and in the cloud.
Development Environment

For all projects, Shadow-Soft recommends a tool set that enables both the client and consultant(s) to collaborate in an effective fashion. Collaborative development requires a means of communication, version control, automation, and a way to virtualize production. In the absence of client provided tools, Shadow-Soft uses Open Source products to provide the required functionality.

Communication

• Rocket Chat - Rocket Chat is a web chat server

Version Control

• Gogs - Gogs is Go based version control system

Production/Platform Virtualization

• Vagrant - Vagrant provides a command-line client for managing virtual environments, and an interpreter for the text-based definition of what each environment looks like, called Vagrantfiles.

• Docker - Docker is an open source software development platform. Its main benefit is to package applications in "containers," allowing them to be portable among any system running the Linux operating system (OS).

Automation/Deployment

• Jenkins - Jenkins is a continuous integration and continuous delivery environment.

• Terraform - Terraform is an infrastructure as code software by HashiCorp.

Supporting Infrastructure

• Docker Registry - A registry is a storage and content delivery system, holding named Docker images, available in different tagged versions.

• Nginx - Nginx is an open source software for web serving, reverse proxying, caching, load balancing, media streaming, and more.

• MongoDB - MongoDB stores data in flexible, JSON-like documents, and serves as the back end for Rocket Chat.

• Nexus - Nexus is a binary repository manage
Tool/Service Selection

To support the required flexibility and speed to market the following tool/services are jointly selected for this client.

Docker

Docker container is an open source software development platform. It’s main benefit is to package applications in “containers,” allowing them to be portable among any system running the Linux operating system (OS).

Containers virtualize the OS, splitting it up into virtualized compartments to run container applications. This approach allows pieces of code to be put into smaller, easily transportable pieces that can run anywhere. It’s a way to make applications even more distributed, and strip them down into specific functions.

Docker provides advantages when hosting applications in containers like:

- Application Isolation - this allows for running multiple applications on the same machine in a cost-effective manner
- Making the application platform cloud agnostic - using docker to containerize the application makes it easy to host the docker container on any infrastructure platform, like on-premise, or any cloud service provider like AWS, Microsoft Azure
- Docker containers are immutable in nature - they will work in the way they are supposed to.

Jenkins

Jenkins offers a simple way to set up a continuous integration or continuous delivery environment for almost any combination of languages and source code repositories using pipelines, as well as automating other routine development tasks. While Jenkins doesn’t eliminate the need to create scripts for individual steps, it does gives a faster and more robust way to integrate the entire chain of build, test, and deployment tools than can be easily built.

The client’s reason for using Jenkins lies in the CI/CD environment that it provides. With just a click of a button, the whole infrastructure can be spun up. Also the whole process can be audited without much effort.
Nexus

Nexus is a repository manager. It allows for proxy, collection, and management of dependencies so that there are not constantly juggling of a collection of JARs. It makes it easy to distribute your software. Internally, you configure your build to publish artifacts to Nexus and they then become available to other developers.

Using Nexus as a repository manager provides the client with the ability to host an internal repository, which allows for control and auditing of the binaries.

Terraform/CloudFormation Templates

AWS CloudFormation enables for the creation and provisioning of AWS infrastructure deployments predictably and repeatedly.

Terraform allows users to define an infrastructure in a high-level configuration language, from which it can create an execution plan.

With using either CloudFormation Templates or Terraform, the client can realize the benefits like:

- Ease of Use - its easy to classify and station a suite of secure platform’s resources
- Its flexible and declarative - write the resources required, and rest of the steps required for the environment to be stood up will be taken of accordingly
- Customization through parameters - the same template can be reused for standing up a completely different environment on AWS by giving the template a different set of parameters
- Version control your infrastructure - the templates can be backup and saved for reuse by uploading it to a version control system like Gogs

ECR

Amazon Elastic Container Registry (ECR) is a fully-managed Docker container registry that makes it easy for developers to store, manage, and deploy Docker container images. Amazon ECR is integrated with ECS.

Reasons for using ECR:

- Amazon ECR eliminates the need to operate private container repositories or worry about scaling the underlying infrastructure.
- Amazon ECR hosts images in a highly available and scalable architecture, allowing reliable deployment of containers for the applications
- Integration with AWS Identity and Access Management (IAM) provides resource-level control of each repository.
IAM

AWS Identity and Access Management (IAM) is a web service that helps securely control access to AWS resources. IAM is used to control who is authenticated (signed in) and authorized (has permissions) to use resources.

Advantages of IAM:

- Granular Permissions - Different levels of permissions can be granted to different people for different resources
- Secure access to AWS resources for applications that run on Amazon EC2 - IAM features can be used to securely provide credentials for applications that run on EC2 instances. These credentials provide permissions for the application to access other AWS resources.
- Multi-Factor Authentication - Two-factor authentication can be added to an account and to individual users for extra security.

CloudWatch

Amazon CloudWatch is a monitoring service for AWS cloud resources and the applications that run on AWS. Amazon CloudWatch can be used to collect and track metrics, collect and monitor log files, set alarms, and automatically react to changes in AWS resources. Amazon CloudWatch can monitor AWS resources such as Amazon EC2 instances, Amazon DynamoDB tables, and Amazon RDS DB instances, as well as custom metrics generated by applications and services, and any log files that the applications generate.

Why use Amazon CloudWatch:

- Monitor Amazon EC2 - the metrics can be used for scaling in and out when the Amazon EC2 instance being monitored is behind an AutoScaling Group
- Custom Metrics Monitoring - Using Amazon CloudWatch, the application logs being generated can be used as a metric for the EC2 instance.
- Use/Set Alarm - based on the metrics being monitored, alarms can be set, which can take additional actions based on the type of alarm, like telling the AutoScaling Group to perform a scale-in/scale-out event, or send notifications to the concerned entity using Amazon Simple Notification Service (SNS)
**EC2**

Amazon Elastic Compute Cloud (Amazon EC2) provides scalable computing capacity in the AWS cloud. Using Amazon EC2 eliminates the need to invest in hardware, so that the applications can be developed and deployed faster. Amazon EC2 can be used to launch as many or as few virtual servers as needed, configure security and networking, and manage storage. Amazon EC2 enables scaling up or down to handle changes in requirements or spikes in popularity, reducing the need to forecast traffic.

**ECS**

Amazon Elastic Container Service (Amazon ECS) is a highly scalable, fast, container management service that makes it easy to run, stop, and manage Docker containers on a cluster. You can host your cluster on a serverless infrastructure that is managed by Amazon ECS by launching your services or tasks using the Fargate launch type. For more control you can host your tasks on a cluster of Amazon Elastic Compute Cloud (Amazon EC2) instances that you manage by using the EC2 launch type. ECS is comprised of:

**Clusters:** An Amazon ECS cluster is a logical grouping of tasks or services. If you are running tasks or services that use the EC2 launch type, a cluster is also a grouping of container instances. When you first use Amazon ECS, a default cluster is created for you, but you can create multiple clusters in an account to keep your resources separate.

**Task Definitions:** A task definition is required to run Docker containers in Amazon ECS. Some of the parameters you can specify in a task definition include:

- The Docker images to use with the containers in your task
- How much CPU and memory to use with each container
- The launch type to use, which determines the infrastructure on which your tasks are hosted
- Whether containers are linked together in a task

You can define multiple containers in a task definition. The parameters that you use depend on the launch type you choose for the task.

**Services:** Amazon ECS allows you to run and maintain a specified number (the “desired count”) of instances of a task definition simultaneously in an Amazon ECS cluster. This is called a service. If any of your tasks should fail or stop for any reason, the Amazon ECS service scheduler launches another instance of your task definition to replace it and maintain the desired count of tasks in the service.
The POC illustrates the build, deployment, and maintenance of a containerized application (a nodejs app) on the Amazon ECS. The entire application stack, outside of VPC creation, is defined as a CloudFormation Template. Additionally Terraform code was developed to show that option. All code was stored in source control.
**Base Images**

The base image used for creating all other images is the official CentOS 7 image that was pulled from the Docker Hub. This image is well maintained and is used to create the base image. The base image is a representative of how images should be maintained in the future.

CentOS is a Linux distribution that provides a free, enterprise-class, community supported computing platform functionally compatible with its upstream source, Red Hat Enterprise Linux (RHEL), which is why this platform has been used in the creation of the base image.

From this base image, 3 additional images are created for hosting Gogs, Rocket Chat, and Jenkins. Gogs is used in the POC to serve as the source control where all the future images and cloudformation templates would be stored and version controlled. Rocket Chat will be used for communication between Shadow-Soft and the client. Finally Jenkins will be used for CI/CD.

Additionally a runtime image, Java 1.8, and a representative application image (Hello World) built on top of the Java image were created. The hello-world image is representative of the production application that will be hosted later on.

The Dockerfiles necessary for creating the docker images for hosting the above mentioned applications are stored in source control (Gogs).

**Dockerfiles**

The official Jenkins Dockerfile, available from the Docker Hub, has been modified such that it is now using the base image as its base image. Extra steps have been taken to add docker installation as a part of the modified Jenkins Dockerfile, to make it able to automate the building of docker images as a part of the CI/CD pipeline and push the image to the Amazon ECR. The Jenkins images also install necessary plugins like the Docker-plugin, blue-ocean plugin, and others that allows it to communicate with AWS.
Vagrant Environment

Vagrant is used in the POC to create a Linux VM (CentOS 7), to provide a common environment for every individual working on the project, to work out of. A Vagrantfile (file, which the Vagrant software uses to build the VM) is created for the same, thus furthering the approach of infrastructure-as-a-code methodology.

The Linux environment has aws cli pre-installed on it (steps described in the Vagrantfile) with the correct aws credentials and the configuration file. These files are required for communicating with the client’s AWS account via the command line. The Linux environment also has docker pre-installed on it, to build and test the docker images being created before they are pushed to the ECR repo for usage in AWS.

ECR Creation and Initial Seeding

The prerequisite to start working with the AWS ECS service is that the images, which the ECS cluster will be using as a part of its service definition, have to be present in the AWS ECR repository. By default, the repositories necessary won’t be available, and will have to be created manually. This can be done either by going to the AWS console, or via the command line.

NOTE: The name of the docker image built must match the name of the repository in Amazon ECR for the image to be successfully uploaded. This can be done by tagging the built image with the correct name.

The docker images can be pushed to the ECR repository either via the command line, or via the AWS console. If using the command line, the first step required would be to login to the ecr repo, followed by pushing the image to the ecr repo. When pushing the image, the image name will be matched with the repo name, and the image will be pushed to the correct repo.

For the POC, AWS ECR repositories where created for base image, Rocket Chat, Gogs, Jenkins, Java, and the hello-world app, and images were pushed to the corresponding repository.
VPC Creation

For the purpose of the POC, the AWS ECS environment, along with all the dependencies necessary to successfully run the environment, like ELB, ASG, Target Groups, were built under the default VPC provided by AWS, in the us-east1 region.

NOTE: For the production environment, a separate VPC will have to be created by the Security team.

The VPC can be created in one of the ways described below:

1. By using the AWS Console, under the VPC section
2. By using the AWS CLI via the command line
3. By using cloudformation template/terraform

ECS Cluster Creation

For the AWS ECS environment, an ECS cluster has to be stood up. Steps to be followed:

1. Prerequisite: docker images have to be pushed to the ECR repository
2. Create a cluster
3. Create a task definition for the cluster
4. Create a service for the cluster to run the tasks using the task definition

The above steps can be completed to stand up an ECS cluster, using either AWS Console, or CloudFormation Templates/Terraform.

For the POC, the initial ECS Cluster was created to host Amazon EC2 instances that used services to run the task definitions for Gogs, Rocket Chat, and Jenkins. The task definitions created for the three pushed images exposed container port 8080 for Jenkins, 3000 for Gogs and 3030 for Rocket Chat, for the three to be accessible from outside the AWS environment. The task definitions were created to run on a m2.large EC2 instance.
Cloud Formation Templates

CloudFormation templates (CFT) provide the power for version controlling the infrastructure, and build it from scratch by the click of a button. The same CFT can be used to work in different ways by providing it with different sets of parameters.

Separate CFTs were built for automating/replicating the creation of the complete AWS ECS Cluster environments for hosting the Gogs, Rocket Chat and Jenkins services. That were previously build manually. Another CFT was created for hosting the hello-world application environment. The templates were tested on the AWS environment, and on successful creation of the environment were uploaded to the Gogs server.

CI/CD Automation

The Jenkins service hosted on the AWS ECS cluster has the plugins necessary for the POC installed on it as a part of the image creation.

Jenkins pipeline to add CI/CD into the POC environment. In Jenkins, pipelines were created for automating the process of building the docker image using Dockerfiles by pulling them from a source control manager (Gogs). The docker images are built locally, on the container image, using the docket plugin for Jenkins. Next, the images are uploaded to the AWS ECR, as a part of the pipeline, using the AWS ECR plugin for Jenkins. The Jenkins pipeline reads its instructions from a Jenkinsfile (a text file that contains the definition of a Jenkins Pipeline and is checked into source control). The Jenkinsfile and the corresponding Dockerfiles, along with any other files necessary for the successful image creation are uploaded to the Gogs server for version control.

A separate pipeline and a Jenkinsfile were created for the hello-world application. The difference between this pipeline and the rest was that additional stages were added in the pipeline to run the CFT as a part of the pipeline.
Diagram 1.2 represents the flow chart for the hello-world pipeline in Jenkins. It also represents the flow for the successful creation of the stack when running CFT.

Another pipeline, hello-world-terraform, was created in Jenkins to host the same hello-world application using Terraform. The following diagram represents the flow chart for the hello-world pipeline in Jenkins. The diagram also represents the flow for the successful creation of the stack when running Terraform.
Diagram 1.3
The final part added to the POC was the creation of a deployment methodology for application version update. Two deployment methodologies were created and tested for the project.

1. Rolling Update:

This was tackled in the CloudFormation Template, under the AutoScaling Resource section. This deployment method, keeps a minimum number of instances specified still running, and replaces the other instances, and once they are up and running, it replaces the next batch of instances, all the while keeping the minimum number of instances always running. Diagram 1.4 \ represents how rolling update is performed

Diagram 1.4
2. Blue/Green Deployment:

This methodology was tackled in Terraform, by using modules in the application.tf file. Blue-green deployment is a technique that reduces downtime and risk by running two identical production environments called Blue and Green.

At any time, only one of the environments is live, with the live environment serving all production traffic. For this example, Blue is currently live and Green is idle.

When a new version of the software is prepared, deployment and the final stage of testing takes place in the environment that is not live: in this example, Green. Once the software in Green environment has been deployed and fully tested, switch the router so all incoming requests now go to Green instead of Blue. Green is now live, and Blue is idle.
This technique can eliminate downtime due to application deployment. In addition, blue-green deployment reduces risk: if something unexpected happens with the new version on Green, you can immediately roll back to the last version by switching back to Blue. Diagram 1.5 represents Blue/Green deployment.

NOTE: All the steps taken above have been taken under several assumptions about the live environment. Although the steps taken follow the DevOps/AWS best-practices, it can be changed to make the whole process more in-line with how the client wants it to be.
Redeployment of the POC

Redeployment of the POC can be done in two different ways depending on whether the stack that is already running needs to be Updated or Deleted and Re-deployed:

NOTE: Every AWS ECS Cluster that has been created for the POC has a corresponding CFT attached with it; no manual steps are required to perform any steps.

1. Updating the Stack: A running stack needs to be updated if the current configuration for the stack is not what is required, or if the stack requires changes. For updating a running stack, just re-deploy the CFT with the correct set of parameters. The CFT is smart enough to go through the running configuration, and only update the resources that requires an update. Once complete, it will show the output as UPDATE_COMPLETE. If something goes wrong, it will rollback to the previous running configuration, while displaying where the error occurred.

2. Re-Deploying the Stack: in case the stack needs to be re-deployed; delete the current stack first, which will delete all the resources running under the current configuration, and then deploy the CFT again with the changed configuration.

NOTE:

1. For updating and re-deploying the stack, the stack-name that was used is necessary, as that is how the CFT knows which stack to work on.

2. All the steps like updating, re-deploying can be carried out either from the command line using the AWS CLI, or the AWS Console, with Console being the better of the two options as it provides visual updates for the running CFT.

The terraform script has a Jenkins pipeline created for it, and hence all terraform related steps can be done via Jenkins.
Moving to Production

The reference architecture above has been designed and implemented making assumptions about the future state of the client’s AWS environment. In order to move from a POC into a development or production ready systems there are many key decisions that must be made around VPC implementation, process ownership, image lifecycle, application and infrastructure scaling, security, and change management. Changes to the POC code will depend heavily on the end result of the client’s VPC implementation and processes.

Cloud Migration Strategy

There are four primary migration paths that have emerged as common strategies employed by many organizations.

Lift and Shift

The quickest way to move applications to the cloud is to migrate the actual machine images. This method works the best when an organization is already leveraging virtual machines as their base architecture. True ROI is not realized in this model but it is the least impactful.

Re-Platform

For organizations that are primarily utilizing legacy middleware, or are looking to move from something proprietary to something open source (i.e. WebSphere to JBoss), a popular strategy is to Re-Platform. While less involved that refactoring your application special care is required to ensure feature parity between the old platform and the new one.

Application Refactoring

Clients who want to move into containers and/or microservice architectures will usually require heavy refactoring of their application stacks. Completely rethinking the application most likely will require code rewrites and re architecture to achieve the maximum efficiency. Cloud providers support various platforms for deploying and managing microservice based architectures.

Shift to SaaS

In a Shift to SaaS model organizations abandon their own tool sets and use services provided by the platform. Generally if that services is not a core competency for your organization, and vendor lock in is not an issue, you should consider using a service provided by the platform. eg Leveraging Amazon RDS instances instead of maintaining your own relational databases.
Application Migration

Any or all of the above methods can be used to migrate your organization to the cloud. When considering which applications to migrate and what strategy to leverage consider the following exercise to aid you.

The Figure above each circle represents an application. Draw the table on a whiteboard and rank your applications according to the following:

- **Section 1** - Low Impact, Low Difficulty: Web applications that immediate candidates for migration. These applications generally don’t have external dependencies and are easy to migrate. They are also Low Impact meaning the service availability isn’t necessarily critical.

- **Section 2** - High Difficulty, Low Impact: Identifies applications whose service availability isn’t critical but may be fairly difficult to determine a migration strategy. These applications may have many interdependencies making it cumbersome for migration. Considering these applications are low in impact service availability isn’t necessarily critical.

- **Section 3** - High Impact, Low Difficulty: Applications that are important to the business but not necessarily difficult to migrate. These are legacy applications that may have yet to have been virtualized or running many applications on a single VM.

- **Section 4** - High Impact, High Difficulty: Applications that are tough to migrate and extremely critical to the business. They are also critical to the business and require extensive planning to ensure a smooth migration.

Use the results of this exercise to help you determine the appropriate migration strategy.
VPC Architecture

The diagram below represents an enterprise VPC architecture. Each VPC, namely DEV, QA, and PROD, will have the same infrastructure inside, which will be what was shown in the previous diagram. They will have a peering with the Management VPC, which will have the appropriate roles and users created in it to provide the management with visibility into the other three VPCs.
Scaling Considerations

The process to create services that scale automatically is natively supported by the ECS console, CLI, and SDK. A service owner can describe the desired, minimum and maximum number of tasks, create one or more scaling policies, and Service Auto Scaling handles the rest. The service scheduler is Availability Zone–aware and automatically distributes your ECS tasks across multiple zones. The Auto Scaling group for the ECS cluster manages the availability of the cluster across multiple zones to provide resiliency and dependability.

These features allow organizations to match deployed capacity to the incoming application load, using scaling policies for both the ECS service and the Auto Scaling group in which the ECS cluster runs. Multi-AZ clusters make your ECS infrastructure highly available, keeping it safeguarded from potential zone failure. The Availability Zone–aware ECS scheduler manages, scales, and distributes the tasks across the cluster, thus making your architecture highly available.

To accurately scale applications, and the underlying clusters, it is vital to understand the Service Level Objective (SLOs) and key Service Level Indicators (SLIs) for the service(s) that are being provided.

SLOs are typically a target value or range of values for a service level that is measured by an SLI, and should directly correlate to user satisfaction. A natural structure for SLOs is $\text{SLI} \leq \text{target}$, or lower bound $\leq \text{SLI} \leq$ upper bound. Defining SLOs can be complicated as they are often driven by the users of your service instead of the services owners.

An SLI is a carefully defined quantitative measure of some aspect of the level of service that is provided. Most services consider request latency—how long it takes to return a response to a request—as a key SLI. Other common SLIs include the error rate, often expressed as a fraction of all requests received, and system throughput, typically measured in requests per second. The measurements are often aggregated: i.e., raw data is collected over a measurement window and then turned into a rate, average, or percentile. Another important SLI is availability, or measure of time that your service is able to be used.

Although SLIs are ultimately dependent on the service being provided there are four key indicators that serve as a starting point.
1. Traffic

A measure of how much demand is being placed on your system, measured in a high-level system-specific metric. For a web service, this measurement is usually HTTP requests per second, perhaps broken out by the nature of the requests (e.g., static versus dynamic content). For an audio streaming system, this measurement might focus on network I/O rate or concurrent sessions. For a key-value storage system, this measurement might be transactions and retrievals per second.

2. Errors

The rate of requests that fail, either explicitly (e.g., HTTP 500s), implicitly (for example, an HTTP 200 success response, but coupled with the wrong content), or by policy (for example, “If you committed to one-second response times, any request over one second is an error”). Where protocol response codes are insufficient to express all failure conditions, secondary (internal) protocols may be necessary to track partial failure modes. Monitoring these cases can be drastically different: catching HTTP 500s at your load balancer can do a decent job of catching all completely failed requests, while only end-to-end system tests can detect that you’re serving the wrong content.

3. Latency

The time it takes to service a request. It’s important to distinguish between the latency of successful requests and the latency of failed requests. For example, an HTTP 500 error triggered due to loss of connection to a database or other critical backend might be served very quickly; however, as an HTTP 500 error indicates a failed request, factoring 500s into your overall latency might result in misleading calculations. Therefore, it’s important to track error latency, as opposed to just filtering out errors.

4. Saturation

How “full” your service is. A measure of your system fraction, emphasizing the resources that are most constrained (e.g., in a memory-constrained system, show memory; in an I/O-constrained system, show I/O). Note that many systems degrade in performance before they achieve 100% utilization, so having a utilization target is essential.

AWS provides a built-in means of tracking these metrics via CloudWatch metrics. Amazon CloudWatch collects and processes raw data from Amazon ECS into readable, near real-time metrics. These statistics are recorded for a period of two weeks, so that you can access historical information and gain a better perspective on how your clusters or services are performing. Amazon ECS metric data is automatically sent to CloudWatch in 1-minute periods. CloudWatch Alarms can be used in conjunction with AutoScaling policies to automatically scale out and in both ECS services and the underlying cluster. The diagram below shows a reference architecture for leveraging CloudWatch and AutoScaling Policies.
For more information about leveraging CloudWatch within ECS, see the Amazon Elastic Container Service User Guide.
Capacity Planning

Each ECS cluster is made up of multiple EC2 instances. While highly dependent on the type of workload, the sizing of nodes in the cluster, in terms of CPU, RAM, and storage resources, should consider the following:

All nodes should:

- Fulfill the minimal requirements for the task definitions running in the cluster. If nodes do not meet this requirement services will fail to start.
- Be of the type that reflect the target workload, eg nodes with more or less RAM, GPU access, etc

Ideal node size will vary based on your workloads, so it is impossible to define a universal standard size. Node size should therefore be determined by experimentation and testing actual workloads, and refined iteratively. A good starting point is to select a standard or default machine type in your environment and leverage an 80/40 rule (Add capacity at 80%, and remove capacity at 40%) for defining your auto scaling policies. Ensure that metrics such as CPU, memory, disk space, number of running container, disk I/O, and cost are combined with application metrics to inform your cluster sizing.

Since workload type has such a heavy influence on cluster sizing, proper planning is required to determine what workloads will run on a cluster when the cluster will be used by multiple groups. For instance services that have drastically different requirements for CPU, RAM, and disk space can interfere the availability of each other and should not be run on the same cluster. In general you should scale your cluster down if capacity drops below 40%, however this decrease in demand can be offset by running services that have binary usage requirements, eg a cluster that is in high demand during the day but not at night can be used to run batch processes when demand is low. This is primarily a function of cost as running multiple clusters can increase overall spend.
Managing Images

Docker works with and consists of the following components:

- **Docker Container Format**: A format for describing and interacting with a containerized application or a component of an application. Each container is based upon an image that holds the necessary configuration data for that container.

- **Image**: This is a static snapshot of the configuration of the container. It is a read-only layer that is never modified. The image is updated by writing a new image on top of the old one – while the old one is preserved.

- **Registry**: Repository of images

- **Dockerfile**: a configuration build file with instructions for Docker image. This is how build procedures are automated, shared, and reused. Advantages of Docker

- **Docker Client/Server**: Docker based architectures use a Client/Server model for both managing and interacting with containers.

**Image Lifecycle**

Base images represent the starting point for all other images used inside an organization. Base images can be pulled from the Docker Hub or created internally using a special image called Scratch. Organizations should tightly control what base images are allowed for use in their environments. Base images are usually controlled by the Infrastructure team with input from Security. Treat Base images the same way you would treat base AMI’s or vmWare images from the standpoint of security and ownership. When selecting or creating Base images pay attention to the following:

- **Distribution**: Docker’s flexibility means that almost any base distribution can host your application, however, you should consider whether your containerized application requires specific libraries or tools from a specific system, and whether or not there is a preferred corporate distribution.

- **Image size**: Base images usually contain a minimal operating system with a set of tools needed for basic operations. To preserve your environment small and efficient, size should also be taken into account when picking the right base image. The size varies; you can take advantage of super small base images, such as 2MB busy-box, or use a standard minimal operating system, such as Fedora or CentOS that are up to 200MB in size.

- **Updates**: Not all community images are necessarily rebuilt on a regular basis or when security vulnerabilities are addressed. You should therefore consider using base images from “official repositories” on Docker Hub, and confirm their update policy in advance. Alternatively your organization may choose to update their own base images.

Runtime images are one iteration from a Base image. These are built based on the corporately approved bases and contain the runtime environment, but not the application code, for applications that are developed. Examples of Runtime images are Java, Ruby, Jboss, and Apache. Runtime images can contain the necessary development tools such as compilers and build libraries. If your Runtime images container development tools then you will require a different Runtime image to build your Application images.
Application images are the image type that is most often run in a production capacity. Application images are normally built from Runtime images and add the application code.

There are many other best practices for creating images, such as chaining commands in Dockerfiles, using multi stage builds, and ensuring that unnecessary packages are not installed. Best practices for image creation are outside the scope of this document. Please refer to docs.docker.com/develop/develop-images/dockerfile_best-practices/ for further information.

**Storing Images**

Images should be stored in a central repository that is accessible by developers and production clusters alike. The Registry is generally installed into the Management VPC. To provide stricter separation multiple registries can be used, eg Dev/Prod. The pushing and pulling of registry images should be controlled via a service account and routed through an automation platform such as Jenkins. This ensures that developers do not overwrite images inadvertently and can be used to enforce naming conventions, versioning, namespacing, and access control. Amazon’s Elastic Container Registry is a secure and highly available option for storing Docker images. Other options include JFrog’s Artifactory, Nexus, and the Docker Trusted Registry. Other registry options may offer more flexibility and control than ECR but at the cost of simplicity.

**Image Security**

Security of Docker images is largely based on the process controls put in to place around them. Images should be be regularly scanned for security compliance and patched in lock step with your machine images. Leveraging a platform such as Twistlock or Aqua can provide an additional layer of security for Docker images. Base image use should be enforced throughout the organization, ensuring that all running containers have been instantiated from trusted sources.

Restricting access and capabilities reduces the amount of surface area potentially vulnerable to attack. Docker’s default settings are designed to limit Linux capabilities. While the traditional view of Linux considers OS security in terms of root privileges versus user privileges, modern Linux has evolved to support a more nuanced privilege model: capabilities. Allowing or removing kernel level capabilities gives organizations fine tuned control of the permissions of Docker containers.

Docker further reduces the attack surface by restricting access by containerized applications to the physical devices on a host, through the use of the device resource control groups (cgroups) mechanism. Containers have no default device access and have to be explicitly granted device access. These restrictions protect a container host kernel and its hardware, whether physical or virtual, from the running applications.

Docker containers use copy-on-write file systems, which allow use of the same file system image as a base layer for multiple containers. Even when writing to the same file system image, containers do not notice the changes made by another container, thus effectively isolating the processes running in independent containers.
Modern Linux kernels have many additional security constructs in addition to the referred concept of capabilities, namespaces and cgroups. Docker can leverage existing systems like TOMOYO, AppArmor, SELinux and GRSEC, some of which come with security model templates available out of the box for Docker containers. You can further define custom policies using any of these access control mechanisms.

**Change Management**

**Architecture Review Board**

The Architecture Review Board will approve documentation that supports both the infrastructure and application architecture for a given AppStack. The AppStack stake owner should provide diagrams that describe how the application will be orchestrated in the cloud.

Each AppStack will be created using Terraform. This terraform will be composed of a security section, an infrastructure section, and an application section, with each section defining that particular part on the cloud. Most of the attributes for creating the AppStack will be defined as variables and will have to be entered with appropriate, pre-defined, and validated inputs while bringing up the AppStack.

Another important item that will have to be looked over will be the IAM Roles, such that the AWS resources will have the appropriate permissions to interact with each other.

**Design Approval Process**

Each AppStack entering the cloud needs to undergo an architecture review. This section will detail what this team should look for to ensure an application meet approval standards prior to entering the cloud.

1. The Base Image which will be used to host the application
2. The Security Groups – Exposed Ports
3. IAM Roles, Policies
The POC has been created with several assumptions considering the future of the application in the production environment, but with one application in one container. In the production, there will be five different environments with eight applications to be deployed in each.

Our recommendations includes best practices based on the AWS WAF, Terraform, and DevOps:

- Have one VPC for every environment. Make sure that the CIDR block provided for each VPC is big enough to handle 8 applications in each.

- Have 8 sets of subnets under every VPC, with each subnet set supporting one Application. Note: The Application Load Balancer can span multiple subnets inside the VPC, so placing different applications under different set of subnets will not cause an issue

- Have separate Terraform scripts for every application (Application specific Terraform); this way, most of the parameters like the Cluster name, service name, task definition name, EC2 type, container name, can be hard coded into the Terraform. The VPC id, subnet, security group (basically the stuff around Infrastructure and Security) will be the parameters. And based on how the bifurcation of the scripts, either the parameters can be filled by the application team, and then it can be approved or disapproved by the Security and Infrastructure team, or the outputs from the two team’s scripts can be used as the input for the Application teams script.

- Based on the infrastructure and the application teams inputs, the correct EC2 instance type will be selected for every application. For example, a more compute intensive application will use C4 or C5, while if it is memory intensive, it will end being either an X1 or an R4 instance type.

- The scale in/scale out will happen on the EC2 instances, which will be behind an AutoScaling Group, hosting the services based on the CloudWatch Metrics. The CloudWatch Metrics, based on the application, will either be the ones available, or will be combination of custom metrics and the ones available by AWS.
Call our consultants and discover the right solution for your business:

770-546-0077

shadow-soft.com or email contact@shadow-soft.com

Since 2008, Shadow-Soft has been evangelizing and deploying open source software and open standards to help customers “take the power back” from their technology vendors. Shadow-Soft provides consulting and managed services across three specialties: DevOps, Application Infrastructure, and Cloud.