logicworks

How to Maintain a Secure Containerized Environment

www.logicworks.com

Content

- **Part 1: Security**
  - What It Means to be Secure in the Public Cloud
- **Part 2: Why Containers**
  - Benefits and Challenges of Containers
  - Security Challenges with Containers
  - Risks and Remediation
- **Part 3: Securing Your Containerized Environment**
  - Managed Services for Containers
  - Example: Managing ECS and EKS Secrets
- **Summary**

Introduction

Maintaining a secure environment for your business has remained a top priority, whether it's on-premise or in the public cloud, companies and end-users expect peace of mind while interacting with business critical applications.

As the Information Technology landscape adopts containerization as an application packaging and deployment strategy, so too must the security posture and skillset of developers and decision-makers.

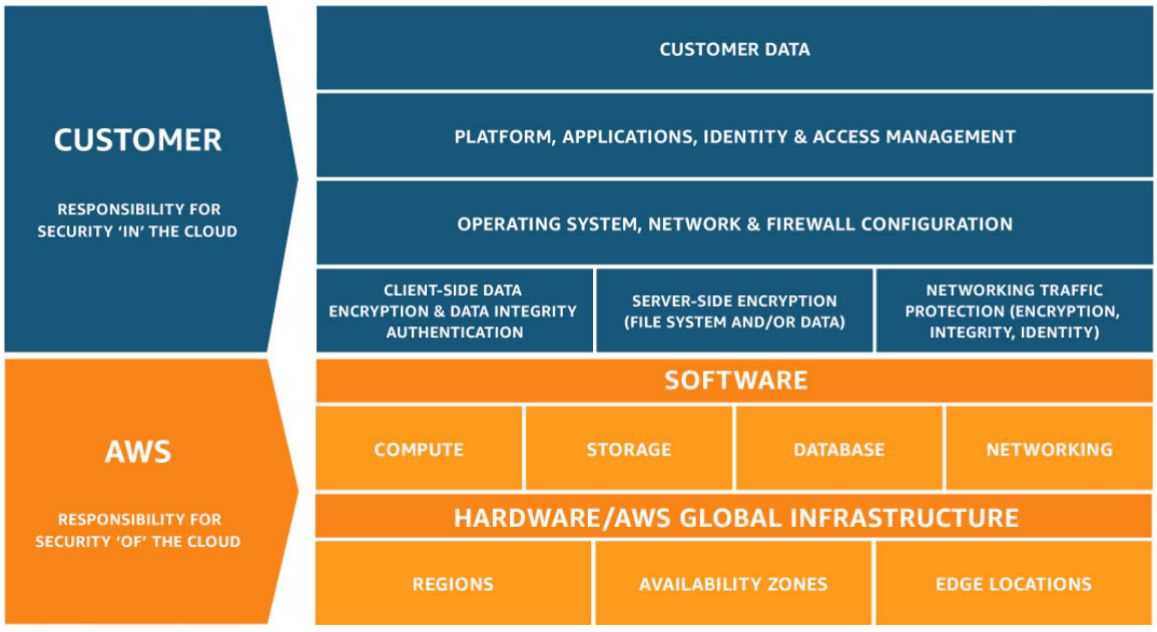
In this ebook, we will look at what it means to be secure on the public cloud, the benefits of using containers, the security challenges surrounding containerization, and some useful best practices to implement in your cloud journey.

Part 1: Security

The threat landscape doesn't take days off, never stops adapting, and is one step ahead of many current security postures. The data that has traditionally been on-premise represents a revenue stream that needs to be accessible, resilient, and more importantly secure. Having a completely on-premise data footprint puts the responsibility of security solely on the shoulders of your company; this includes the physical security of a data center's infrastructure, including: servers, storage, networking, operations, and the software stack. Who is accessing the infrastructure? Who is accessing the applications? Are we well-equipped to deal with emerging security threats? At scale, this can consume the time and efforts of your IT staff, which would be better spent developing innovative applications to drive revenue.

What It Means to be Secure in the Public Cloud

There are many factors that lead a company to the public cloud, some of which include paying only for what you use, application modernization, business continuity, and global-reach, amongst many others. When a workload is migrated to the public cloud there is a shift in responsibilities as well. With AWS for example, that shift can be depicted by understanding the Shared Responsibility Model. No longer are you responsible for the security of your physical infrastructure, instead you are responsible for the data that you store and how it is accessed.



AWS Shared Responsibility Model; Source: AWS Documentation

A benefit of cloud adoption, which cannot be understated, is the time and options developers gain for evaluating new projects. This combination encourages the use of emerging technologies and abilities to become increasingly cost effective, operationally efficient, and staying ahead of both the competition and the threat landscape. One of these emerging technologies in the world of application deployment is the use of Containers instead of traditional Virtual Machines.

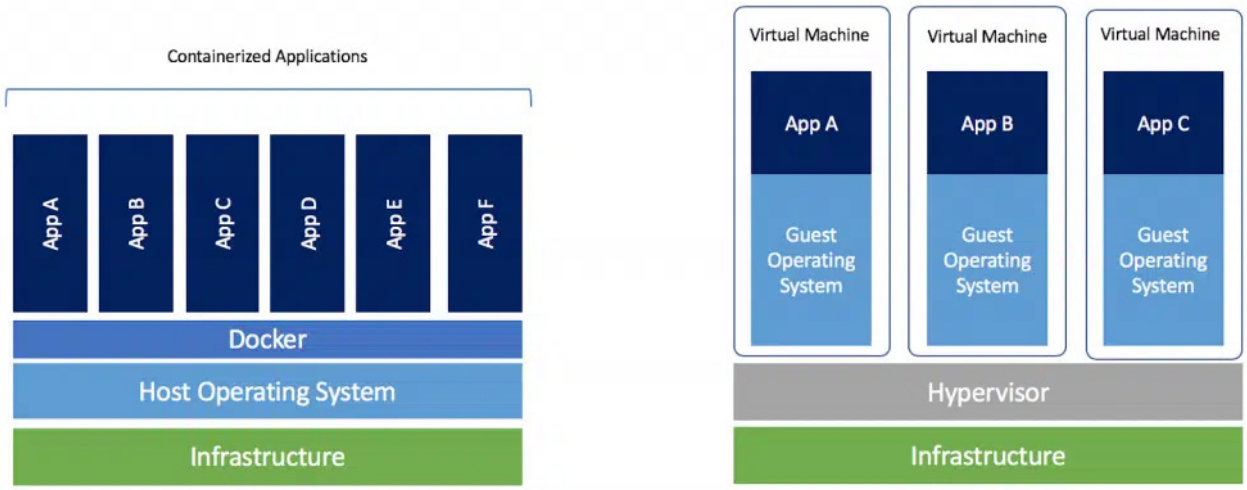
Part 2: Why Containers?

Some of the early steps in planning a new application's deployment is determining what the underlying architecture will be, how it will be developed, deployed, and managed throughout its lifecycle. One of the decisions when moving to the cloud will inevitably be whether to use monolithic instances, like their traditional Virtual Machines or to shift to Containers to produce cloud-native decoupled applications. Common use-cases for containerization include microservices, batch processing, and machine-learning. For this discussion, we will look at monolithic architecture versus microservices architecture.

Benefits and Challenges of Containers

Similar to decisions in an on-premise deployment, there are reasons for either Virtual Machines or Containers to be your flavor of choice in the public cloud. With Virtual Machines, your staff is probably well-versed in either VMware and Microsoft's hypervisor offerings. The comfortability with deploying and managing these applications using a familiar suite of tools will have its advantages in terms of training and operations amongst staff, but the number of licenses with these monolithic instances and amount of under-utilized resources per VM can lead to longer boot times for launching new instances, over-provisioning, and higher costs. With this monolithic approach, you will have a tightly coupled application that has a simpler process in securing it at the expense of needing end-to-end testing to troubleshoot, as one bug or issue can affect the entire application's performance. Plenty of applications run well in a monolithic environment, but the agility, scalability, and DevOps side of microservices are a perfect fit for considering containerization.

When using a microservices architecture, one traditionally chooses containers as a way to decouple applications into a number of smaller functions, all working to support the same application. This allows you to be agile in any new technologies you adopt. Additionally, the demand for a service can scale more precisely to match the current business need by having these microservices scale independently from one another, thus lowering the risk of over-provisioning while maintaining optimal performance. From a DevOps standpoint, you can work simultaneously to develop, update, and manage an application's different microservices independently of each other, as well as take advantage of the light-weight portability of containers to pre-package an application to deploy to any host that has the container daemon installed. This provides a much faster and collaborative DevOps environment for a team pushing to be agile and innovative. VMs are GBs in size, and require individual Guest Operating Systems. In comparison, Containers are typically very light-weight by sharing a host's OS kernel and requiring a much smaller set of files to launch an application. Container images are typically MBs in size – Think of launching an instance in seconds rather than minutes.



Containers versus Virtual Machines; Source: Docker Documentation

When it comes to resiliency, one microservice can fail without necessarily disrupting the performance of other microservices. For example, an eCommerce application may have a microservice for their web-app for creating profiles for new users while using other microservices to add goods to a cart and process orders. The ability to withstand disruption to the "new user profiles" container without impacting the "order processing" container may not

disrupt the flow of revenue nearly as much as if this were a monolithic application with an entire application failure.

With the ability to be more independent with your decoupled application architecture, the complexities of your security posture will increase. You are splitting up the applications main function into a set of smaller functions, each running independently in separate containers communicating over the network. You may lower the blast radius for certain functions of your application by decoupling a previously monolithic application, but increasing the points vulnerability in an application could be seen as a larger attack surface.

Security Challenges

With the fast-paced and efficient deployment architecture that containerization provides, one must evaluate the tradeoffs that will ensue when it comes to security. For example, you will have faster and more scalable deployments with these portable, light-weight application packages. Application resiliency can increase due to the decoupling of a traditional virtual machine into a set of microservices. A specific microservice failure can leave the application running and successfully execute separate tasks as the microservices are independent of one-another. Where the tradeoff begins is with the architecture of traditional virtual machines versus containers. With traditional virtual machines, the architecture of these monolithic applications can be secured, similarly to fortifying a castle, whereas containerization should feel like you are securing a kingdom. In any environment, we can assume having a secure network is necessary. This remains true in a microservices based environment, as microservices have their own complex set of networking patterns.

Containerzation's packaging and deployment architecture presents a number of additional areas of vulnerability that will arise if not considered during the initial shift from virtualization to containerization. The major components of these newfound vulnerabilities include image risks, registry risks, orchestrator risks, container risks, and host operating system (host OS) risks.

Risks and Remediation

Images

Container security starts with the thought of deploying and maintaining secure images. Third party images can speed up deployments, but an external image may not always be trustworthy. An image represents the required files a container will need to run an application or microservice. Each component of an image can have vulnerabilities, whether that's certain files needing a security update or the presence of malware that exposes the container as a whole. For this reason alone, it is imperative that you source your images from trusted providers. When applying security updates to the image, operations will shift from traditionally updating existing virtual machines in production to now simply redeploying a new container with the updated and more secure image. This re-deploy strategy is one reason why the lifespan of a container is shorter than a virtual machine. This is done to stay up-to-date and limit the exposure to image vulnerabilities as they arise over time. Because this is a different application architecture, this will require image vulnerability management and scanning tools designed for containerization. In a containerized environment, multiple containers or tasks can be deployed on a shared host OS, making individual container security of paramount importance. Should one container be compromised, it may leave your host OS or other containers in a vulnerable state as well. For this reason, a principle of least privilege needs to be configured amongst images, as a major vulnerability could be container-to-container or container-to-host OS access.

Registry

Now that we have addressed some of the vulnerabilities that surround the creation, configuration, and operations of container images, we will now look at what threats are present in how you store and access these images. Images are stored in registries as a way for developers to push and pull images to and from a central repository or collection of repositories. This resembles a bank vault for someone looking to gain insight into potentially sensitive or proprietary information. Common mistakes around registries include insecure connections to the registry, out-of-date images still being stored, and insufficient authentication and authorization guardrails. Connecting to the registry should always be done between two trusted endpoints with encryption in transit. This applies not only to development tools, but orchestration tools and container runtimes as well. With secure access established, the next logical step is pushing up-to-date images to the registry for security reasons and pulling those fresh images as a means of deployment. Fresh or new images that are up-to-date should follow a common identification system whether that be versioning or utilizing a tagging

system to ensure stale images are not launched. Just having an old image in a repository isn't a threat, but at scale the chances of launching an old, vulnerable image increases. Utilizing automation for lifecycle management (where applicable) is similar to retention policies on backups, it's encouraged. Furthermore, guardrails can be set to lower risk with who can upload to the registry and where images are stored. For this reason, a registry can have multiple repositories, each having their own set of read and write permissions, similar to the principle of least privilege, where only permissions required to complete a task are granted. Restrictions can also be implemented around write access so that an image cannot be successfully pushed to the registry without a prior vulnerability and compliance assessment.

Orchestrator

Orchestrator risk relies heavily on a few factors: user privileges, workload classification, and a cluster's logical network topology. An orchestration tool could have users from Test, Dev, and Production teams all requiring access. Some admins may require environment-wide access, while most users will have a principle of least privilege approach and only have orchestrator and cluster access to the resources that align with their application and stage of development.

While an organization likely has its own directory service, it is not uncommon for an orchestration tool to have an additional directory service. Managing the two can be time consuming. Additionally, limiting the number of environment-wide accounts is strongly recommended, as they can affect the entire application architecture. Multi-factor authentication should be enabled on all environment-wide accounts. When it comes to how the orchestrator will interact with changes in demand and determining where containers will begin runtime, having classification based on sensitivity will help determine parameters for containers to be deployed to production without increasing risk.

If you have a public-facing web server and a back-end financial database running in separate containers, they should deploy to different hosts. Likewise, the logical or virtual networks they are deployed in should be isolated from one another, the web-server being public and the financial app being private. Inter-container network policies can be established to allow communication amongst containers with differing classifications of sensitivity without allowing direct access from the public domain of the web-server. At scale, it is not practical for this segmentation to take place manually. Luckily, many orchestration and container security tools provide automation around grouping together containers with differing sensitivity levels, labels, and security configurations.

Container & Runtime Software

The runtime software in a container deployment has its vulnerabilities, but is a less frequent breach point. If the runtime software is compromised, it can lead to a compromised host OS or even other containers running in the cluster. Continuous monitoring and rapid remediation are required when handling the runtime software. Orchestrators can be configured to only allow the deployment of properly maintained runtimes following a Common Vulnerabilities and Exposures (CVEs) vulnerability scan.

Even after runtime software is validated, you may run into typical software vulnerabilities with containers like cross-site scripting for web-apps or SQL injection for database containers. There are even occurrences of "rogue containers", where development teams deploy a test-container and aren't diligent with the deployment's vulnerability because it's "just testing code", increasing risk. Establishing baseline policies for launching containers and logging user identities linked to container deployments will decrease the chance of having a "rogue container." Containers running as "privileged" will inherit root-user capabilities and have access to all resources running on that host. To prevent "privileged" access, you can set the ECS agent environment variable "*ECS_DISABLE_PRIVILEGED*" to "*true*". You can also create an AWS Lambda function to automatically scan your container environment variables and adjust the variable as necessary. A helpful reminder is to use additional tooling that is designed to be container-aware. As this is a recent architecture compared to virtual machines, some monitoring, filtering, and anomaly detection tools, for example, are granular enough for general virtualization but not granular enough for containerization.

Host OS

Whether you choose virtualization or containerization, your host OS is an attack surface. With virtual machines, many hypervisors provide an additional layer of security between the application and the host OS. If you run containers on a general host OS, you will be more susceptible to risk. If possible, start your application stack with a host OS that is container-specific. These are much more minimalistic in nature and present a smaller attack surface than a general OS. In either case, you do not want to mix application deployment strategies on a single host. Ideally, containers should be deployed on hosts that are running a container-specific OS, wherenon-container applications should be deployed to hosts running a general OS.

If you are looking to manage a host that is running a container-specific host OS, you are better off going through an orchestrator by logging in directly to the host OS, which could affect more

containers or applications than intended. User permissions and actions performed with the host OS, regardless of deployment architecture should be logged for anomaly detection.

Even further down the stack of risk awareness, ensuring the hardware and firmware of the host is secure. A common starting point is using a trusted platform module, or TPM. This will store information about the firmware and operating system's expected safe-state,verifying that tampering has not taken place pre-boot or during boot. Logicworks recommends leveraging a suite of security tools to ensure your host OS is secure, including Intrusion Detection and Anti-virus/Anti-malware solutions. Not only are these required for certain compliance regulations, but they are also recommended security best practices.

There is much to unpack when it comes to evaluating risk and remediation with containers. This can be attributed to the decoupling of monolithic applications and shifting to a microservices approach where different groups of people and tools are developing, deploying, and managing independent components of the application. This provides agility to DevOps teams responding to business demands and increases the need for specialization around container security. How the containerized environment is accessed, monitored, and managed is best left to a container-aware suite of tools to ensure that you have the granularity and compatibility needed to safely deploy applications at scale.

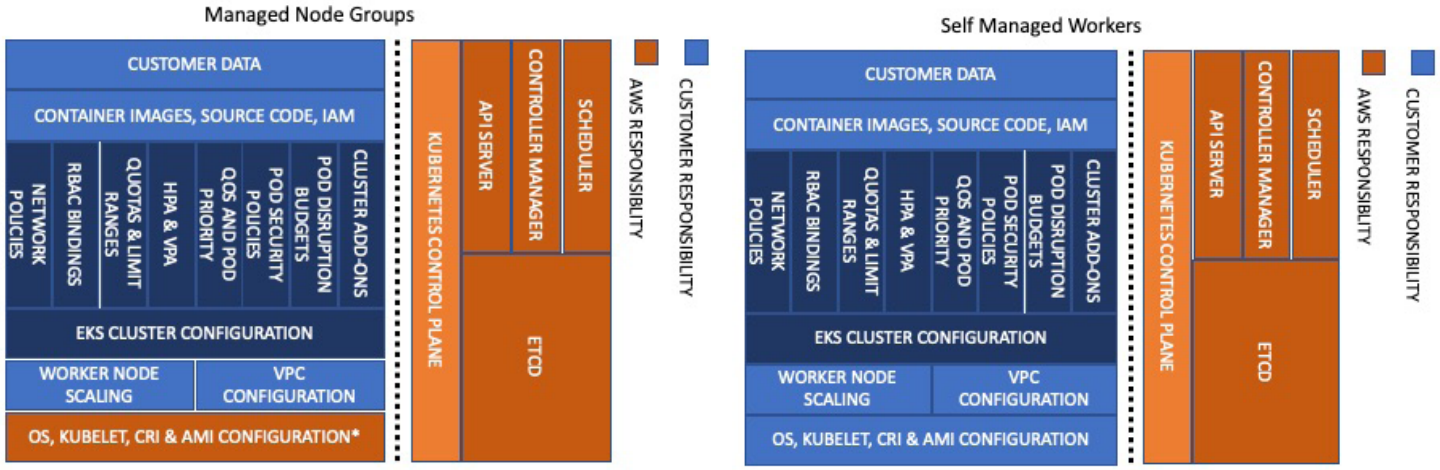
Part 3: Managed Services for Containers

We covered the general security challenges operations will face when decoupling applications, but there are additional obstacles you may encounter, like the challenge of ramping-up orchestration acumen amongst members of the datacenter that traditionally standardized on hypervisors as a means of deployment.

Docker is just one example, in recent years of a cloud-ready, cutting-edge container service at a developer's fingertips. With the abundance of services and options to deploy well-architected environments, finding a secure and operationally efficient approach that covers application development, deployment, and management pays dividends. Enter managed services!

For this discussion, we will look at fully-managed orchestration services pertaining to containers in the public cloud. Implementing containers requires a great deal of orchestration in order to allow developers the time they need to innovate and maintain a secure environment. A common approach of early-adopters was Kubernetes, an open-source orchestration software that aids in container deployment, scaling, and management.

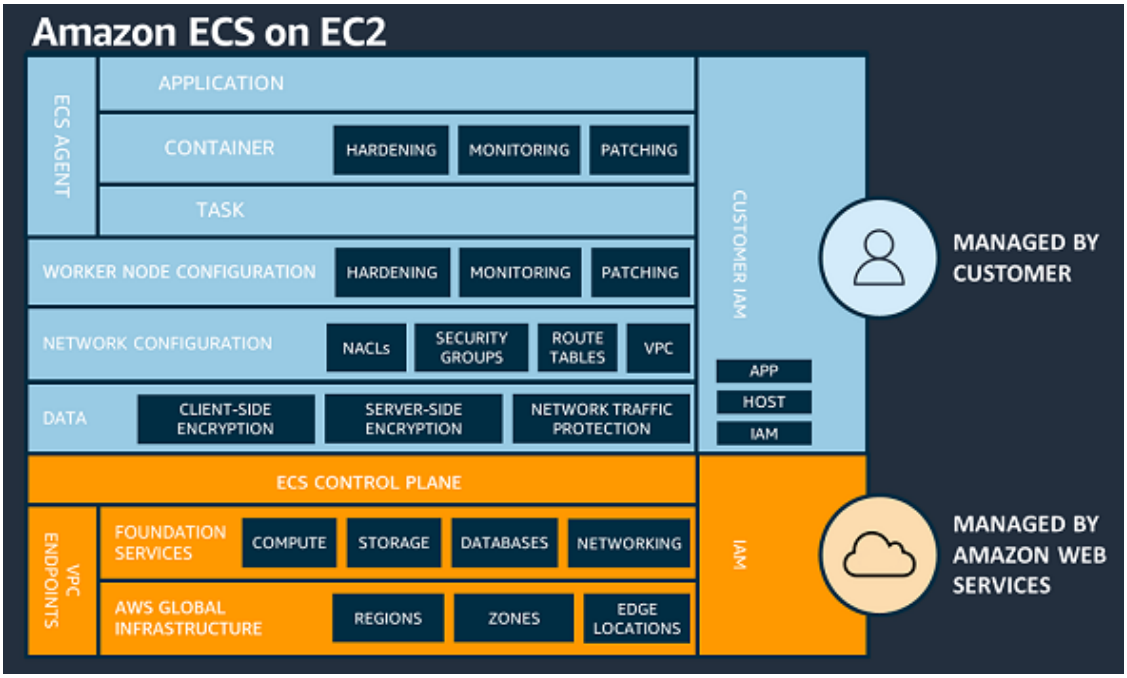
For these early adopters, open-source container tools and add-ons for Kubernetes provided the compatibility and flexibility needed to create a secure and efficient orchestration. However, managing a self-hosted Kubernetes cluster can quickly become a cumbersome task. This can lead to unnecessary operational overhead for your infrastructure teams. For this reason, a shift to AWS Elastic Kubernetes Service (EKS) as a fully-managed Kubernetes service probably makes the most sense. EKS will provide scalability and reliability for your Managed Kubernetes Control Plane and worker nodes with a multi-AZ deployment, while you are trusted with the security of authentication, authorization, pod security, runtime security, and general network security, to name a few.



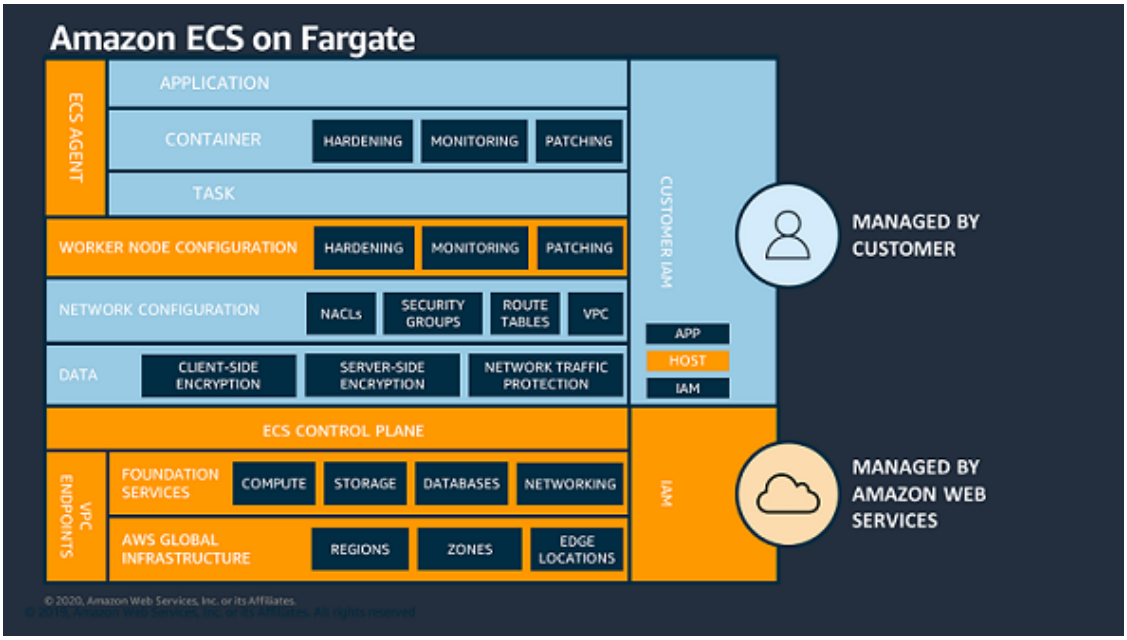
EKS Shared Responsibility Model; Source: AWS Documentation

EKS does not rely solely on AWS features, but requires additional expertise to enable certain integrations, like CoreDNS, an open-source tool used for creating DNS services in a K8S cluster. Pod Security can also be addressed with open-source tooling. Common solutions for this Policy-as-code approach include Open Policy Agent, Kubewarden, jsPolicy, and Kyverno. Authentication should be managed by AWS IAM, and Authorization through Kubernetes RBAC, in order to implement a Principle of Least Privilege. The responsibility of runtime security for your pods can be shifted to AWS with the use of Fargate's serverless deployment. Additionally, the EKS cluster endpoint is public by default, but if there is an internal security policy that forbids public API access or routing traffic outside of cluster's VPC, you can create a private endpoint and specify whitelisted CIDR blocks with access to the private endpoint.

For someone with more recent considerations of implementing a fully-managed Docker container orchestration, utilizing AWS Elastic Container Service (ECS) should be the starting point. For applications requiring persistent storage or consistently high CPU and memory usage, containers should launch with EC2 instances. If infrastructure management is not required, using AWS Fargate as the launch type would be appropriate for batch workloads or small workloads without the need for persistent storage access and minimal overhead. With the combination of ECS being a managed service and having differing launch types, the AWS Shared Responsibility Model shift can be seen below.



ECS on EC2 Shared Responsibility Model; Source: AWS Documentation



ECS on Fargate Shared Responsibility Model; Source: AWS Documentation

ECS was born in AWS and utilizes many of the familiar services a current AWS customer would be proficient in, including EC2 instances, S3 storage, ELB load-balancing, CloudWatch monitoring, CloudTrail API audit and troubleshooting, IAM authentication and authorization, and VPC for logical network isolation. With respect to security, AWS IAM should serve as the primary authorization and authentication service for cluster access.

IAM features you can use with Amazon Elastic Container Service

IAM feature	Amazon ECS support
Identity-based policies	Yes
Resource-based policies	No
Policy actions	Yes
Policy resources	Partial
Policy condition keys	Yes
ACLs	No
ABAC (tags in policies)	Yes
Temporary credentials	Yes
Principal permissions	Yes
Service roles	Yes
Service-linked roles	Yes

Supported IAM Features with ECSl; Source: AWS Documentation

If you want to get more granular than a standard identity-based policy, a custom condition can be attributed to an IAM identity-based policy, which requires either SSL or MFA in order to access a resource. Additional ECS best practices include:

- Perform static code analysis prior to image creation with tools like SonarQube
- Only allow containers to run as non-root users; initial runtime is defaulted to root but, you should create custom launch policies that only launch docker files with user directive included
- Restrict all containers from running as privileged instances. If unrestricted, these containers will have root user capabilities. Set the container environment variable to ECS_DISABLE_PRIVILEGED to true or utilize AWS Lambda to scan your container task definitions for existing “privileged” parameters
- Use immutable tags on images to only launch the latest versions from a repository
- Implement Encryption in Transit with Nitro Instances, TLS certificates, and Server Name Identification paired with an Application Load Balancer
- Run tasks in awsvpc network mode, instead of bridge mode. ECS will automatically attach an Elastic Network Interface and enroll in a Security Group to act as a firewall for those tasks

ECS and EKS can also utilize a fully-managed container registry service called Elastic Container Registry (ECR). This will allow your developers to securely store and share container images and artifacts. Container images can have binary and library vulnerabilities upon creation and develop them over time. In order to combat this, ECR utilizes Clair, an open source image scanning solution. Results from the scan can be logged to AWS Security hub and EventBridge for automated ticketing workflows. ECR also utilizes S3 for storage with encryption at rest and HTTPS for secure image transfers. By default, ECR uses AWS-256 server-side encryption with S3-managed keys. If you have an Internet Gateway in place, you will likely rely on using IAM for authentication and authorization to the ECR API public endpoint. If you do not have an Internet Gateway in place, create a private endpoint to access the API with a private IP.

ECR integration with IAM allows customers to restrict or collaborate amongst a registry’s underlying image repositories. The registry itself is private by default, but IAM identity-based policies can be implemented to invoke a Principle of Least Privilege as to only allow developers access to their required repositories.

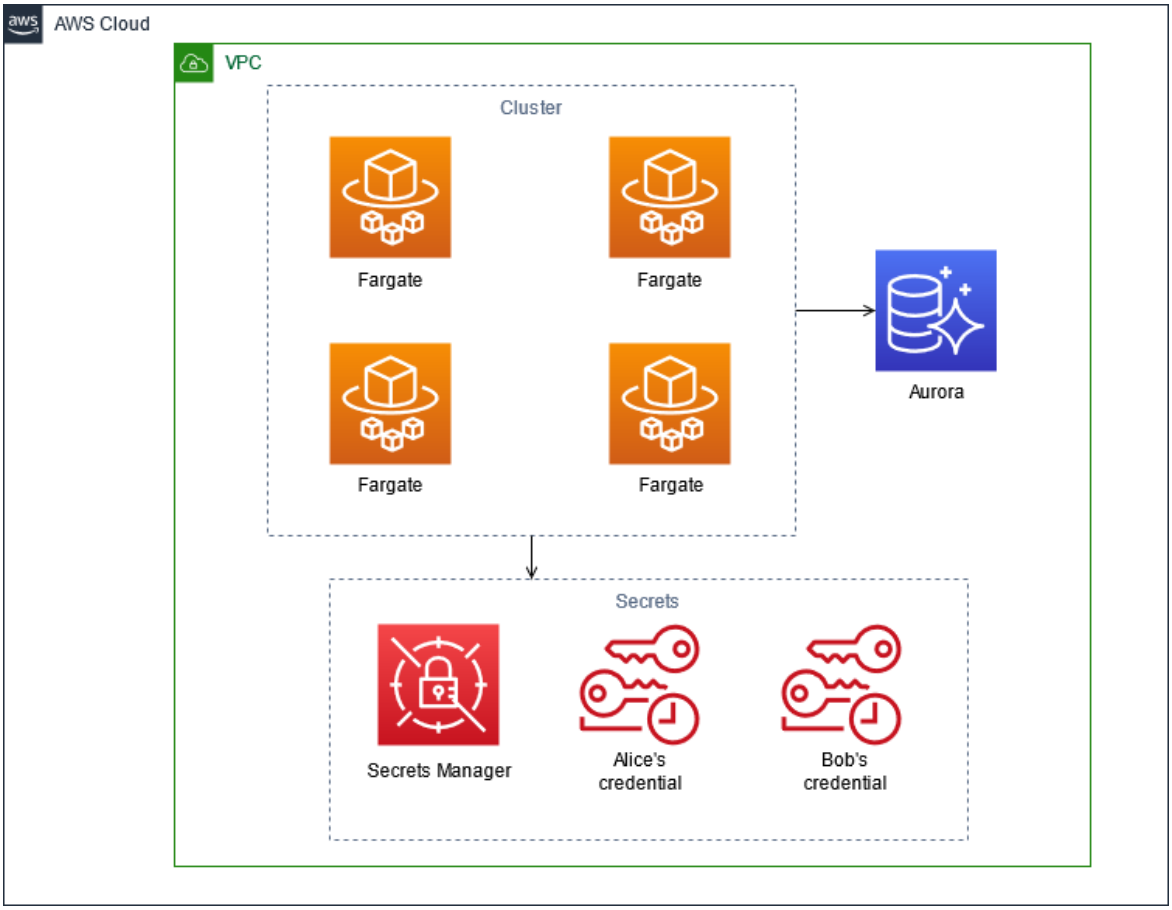
For example, if we wanted to only grant a user the ability to push, pull, and list images to one repository called “my-repo”, then IAM should be updated to reflect the following:

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"ListImagesInRepository",
         "Effect":"Allow",
         "Action":[
            "ecr:ListImages"
         ],
         "Resource":"arn:aws:ecr:us-east-1:123456789012:repository/my-repo"
      },
      {
         "Sid":"GetAuthorizationToken",
         "Effect":"Allow",
         "Action":[
            "ecr:GetAuthorizationToken"
         ],
         "Resource":"*"
      },
      {
         "Sid":"ManageRepositoryContents",
         "Effect":"Allow",
         "Action":[
                "ecr:BatchCheckLayerAvailability",
                "ecr:GetDownloadUrlForLayer",
                "ecr:GetRepositoryPolicy",
                "ecr:DescribeRepositories",
                "ecr:ListImages",
                "ecr:DescribeImages",
                "ecr:BatchGetImage",
                "ecr:InitiateLayerUpload",
                "ecr:UploadLayerPart",
                "ecr:CompleteLayerUpload",
                "ecr:PutImage"
         ],
         "Resource":"arn:aws:ecr:us-east-1:123456789012:repository/my-repo"
      }
   ]
}
```

ECR Least Privilege Exercise with IAM ; Source: AWS Documentation

Example: Managing ECS and EKS Secrets

Whether you choose ECS or EKS to orchestrate your container environment, removing hardcoded sensitive information that sits in plaintext should be implemented into your security posture. AWS Secrets Manager allows you to rotate, manage, and retrieve sensitive hardcorded information, or secrets, like API keys or database credentials.



Sample Architecture; Source: AWS Documentation

In this sample architecture, an ECS deployment is supported by AWS VPC, Fargate, Aurora, and Secrets Manager as a way to automate the rotation of sensitive credentials for multiple database users. Shifting from long-term credentials to short-term credentials with automation can decrease the risk of compromise.The compatibility of ECS and EKS with other AWS services that strengthen your security posture is a key advantage in this example versus only relying on open-source tooling to fill in orchestration and security gaps. This workflow, in addition to implementing the general security best-practices mentioned earlier, will help prevent risk in a containerized environment for the following:

- Image creation
- Image storage
- Repository access
- Container deployment
- Inter-cluster networking
- Container privilege
- Data sensitivity classification
- Orchestration
- Authentication
- Authorization

Summary

As new cloud services and features become available, a company's decision-makers and developers will face a series of choices on how to package and deploy applications to be cost-effective, high-performing, resilient, operationally efficient, and secure. Each of these represents a pillar of a well-architected cloud environment, and mastering these is not easy. As your environment grows and agility is paramount, you need to consider the tradeoffs between each of these pillars of efficiency. If one or more of these well-architected pillars are neglected, your environment may be susceptible to the ever-changing threat landscape.

Containerization certainly has its pros and cons. DevOps agility and collaboration with the light-weight, portable nature of this decoupled application package surely provides reliability and operational advantages. On the other hand, decoupling traditional monolithic virtual machines into separate containers that all share a host OS can be seen as creating a larger attack surface. Choosing to shift that container deployment to a service like AWS ECS or EKS, whether supported by EC2 instances or serverless with AWS Fargate, presents an opportunity to implement a fully-managed experience to limit human-error, control access, and decrease risk.

There are a few operational and security differences between ECS and EKS. Since ECS is proprietary to AWS, they have deep integration with their extensive portfolio and compatibility is not an issue. This may be a leading reason for a customer that is already in the cloud to choose ECS as their managed container orchestration service. EKS is more flexible in the services you wish to add-on to achieve the same business outcome. This may be the preferred route for a containerized customer that has a home-grown security posture and open-source tooling where developer familiarity is high-priority. The simplicity of ECS integration versus the flexibility of EKS open-source tooling is the primary tradeoff worth noting.

Whether you are new to the cloud or new to containers, the adoption of a fully-managed orchestration service, with the additional expertise from a certified managed service provider like Logicworks, will help you maintain a secure containerized environment.

About Logicworks

Logicworks is a cloud consulting and managed services company that helps organizations plan, architect, and manage complex cloud environments. Our team of cloud experts have helped many organizations migrate to AWS and Microsoft Azure with our unique approach to cloud strategy and design.

As an AWS premier Consulting Partner and Azure Expert MSP with HIPAA, HITRUST, PCI, ISO 27001, SOC1, and SOC2 certifications, Logicworks specializes in complex workloads for companies with high security and compliance requirements.

If you're planning new cloud projects and want expert help in avoiding common migration stumbling blocks, visit www.logicworks.com or contact us at (212) 625-5300.