A Zero Trust Reference Architecture for Production-Ready Amazon EKS Environments

Praveen Chaitanya Jakku

Independent Researcher
USA

Abstract:
Amazon Elastic Kubernetes Service has become a practical foundation for running production microservices, but production readiness requires more than cluster availability, autoscaling, and faster deployments. As Kubernetes platforms become more connected to cloud services, CI/CD systems, container registries, secrets stores, and internal business applications, the security model must move away from implicit trust. A Zero Trust approach is well suited for Amazon EKS because it treats every user, workload, network path, and deployment action as something that must be verified, authorized, and continuously observed. This article presents a practical Zero Trust reference architecture for production-ready Amazon EKS environments. The architecture focuses on identity-based access, least privilege, workload isolation, network segmentation, secrets management, secure delivery pipelines, admission control, and continuous visibility. The goal is not to describe Zero Trust as a single product, but to show how AWS-native services, Kubernetes controls, and disciplined DevOps practices can work together to create a secure and reliable production platform.

Keywords: Zero Trust Architecture; Amazon EKS; Kubernetes Security; Cloud-Native Security; Workload Identity; IAM Roles for Service Accounts; Least Privilege Access; Network Segmentation; Kubernetes RBAC; Secrets Management; Secure CI/CD; Admission Control; Container Security; Runtime Monitoring; DevSecOps.

1. Introduction

Kubernetes has moved from an experimental platform to a normal part of enterprise application delivery. Many organizations now use Amazon Elastic Kubernetes Service to run APIs, microservices, batch jobs, integration layers, and internal business platforms. EKS reduces the burden of managing the Kubernetes control plane, but the responsibility for securing workloads, identities, network paths, secrets, and deployment processes still remains with the platform and application teams.

The challenge is that a Kubernetes environment is highly dynamic. Pods are created and destroyed frequently. Services communicate across namespaces. CI/CD pipelines deploy new container images. Applications call AWS services such as Amazon S3, Amazon RDS, AWS Secrets Manager, Amazon SQS, AWS KMS, and Amazon DynamoDB. In this kind of environment, security cannot depend only on a private VPC, a firewall rule, or the idea that "internal" traffic is automatically safe.

Zero Trust provides a stronger design model. NIST describes Zero Trust as an approach that shifts security from static network perimeters toward users, assets, and resources, where access decisions are made based on identity, policy, and context rather than network location alone (Rose et al., 2020). For EKS, this means that no pod, user, service account, pipeline, or network path should receive unnecessary trust simply because it exists inside the cluster or inside the AWS account.

A production-ready EKS environment should therefore be designed around clear trust boundaries. Human access should be role-based and auditable. Workloads should use dedicated identities. AWS permissions should be scoped to specific applications. Network communication should be explicitly allowed. Secrets should be protected outside source code and container images. Deployments should pass security checks before reaching production. Runtime activity should be visible enough for both operational troubleshooting and security investigation.

This article presents a practical reference architecture for applying Zero Trust principles to Amazon EKS. The discussion is written for cloud engineers, DevOps teams, platform teams, and security architects who are responsible for building or improving production Kubernetes environments.

The main contribution of this article is a layered reference model that connects Zero Trust principles with practical EKS controls across identity, networking, secrets, delivery, policy enforcement, and observability. This model can help platform teams evaluate production readiness without treating Kubernetes security as a collection of disconnected tools.

2. Why Zero Trust Matters for EKS

A traditional infrastructure model often places heavy trust in the internal network. Once a server is inside the private network, it may be allowed to communicate freely with many other internal systems. That model becomes risky in Kubernetes because the environment is more distributed and fluid. A single node may run pods from different applications. A namespace may contain services with different security requirements. A compromised container may attempt to reach internal APIs, cloud metadata endpoints, secrets, or databases.

EKS also creates multiple layers of identity. There are IAM users and roles, Kubernetes users and groups, service accounts, node roles, CI/CD identities, and application-level identities. If these are not designed carefully, permissions can become too broad. For example, giving a powerful IAM role to a worker node may allow many unrelated pods to reach AWS services they do not need. Giving broad Kubernetes permissions to a deployment pipeline may allow the pipeline to change cluster-wide security settings instead of only deploying approved workloads.

Zero Trust helps by reducing these broad assumptions. Each workload should have a defined identity. Each identity should have limited permissions. Each network path should exist for a reason. Each deployment should meet policy requirements before it is accepted. Each access decision should leave evidence that can be reviewed later.

This does not mean every system must be blocked from every other system. It means trust should be intentional and narrow. A frontend service may call a backend API, but it does not need direct database access. A reporting job may read selected S3 objects, but it should not update IAM policies. A CI/CD pipeline may deploy an application, but it should not read unrelated production secrets. This is the practical value of Zero Trust in EKS.

Research on Kubernetes security also supports this direction. Shamim, Bhuiyan, and Rahman identified practices such as RBAC, patching, network policies, and pod-level restrictions as important parts of Kubernetes security hardening (Shamim et al., 2020). Container security research has also shown that containerized environments require careful attention to image security, host isolation, runtime behavior, and orchestration-layer controls (Sultan et al., 2019; Viktorsson et al., 2020).

3. Architecture Principles

A Zero Trust EKS architecture should begin with a few clear principles.

The first principle is identity-first access. Every user, workload, and automation system should have a distinct identity. For human users, this usually means IAM roles mapped to Kubernetes RBAC permissions. For workloads, it means Kubernetes service accounts connected to scoped AWS IAM roles. For CI/CD systems, it means deployment identities that are separate from administrator identities.

The second principle is least privilege. Permissions should be limited to the minimum actions and resources required. This applies to AWS IAM policies, Kubernetes RBAC, security groups, network policies, and secrets access. Least privilege also improves operational clarity because teams can understand what each service is allowed to do.

The third principle is explicit network access. Services should not communicate freely only because they are inside the same cluster. Communication should follow the application architecture. If a backend service does not need to receive traffic from another namespace, that path should not be open.

The fourth principle is secure-by-default deployment. Production workloads should meet baseline requirements before they are allowed to run. These requirements may include non-root containers, restricted host access, approved image registries, resource limits, and dedicated service accounts.

The fifth principle is continuous visibility. Logs, metrics, audit events, deployment history, and network activity should be collected centrally. A team cannot enforce Zero Trust effectively if it cannot see what users, workloads, and pipelines are doing.

Figure 1: Layered Zero Trust Model for Amazon EKS

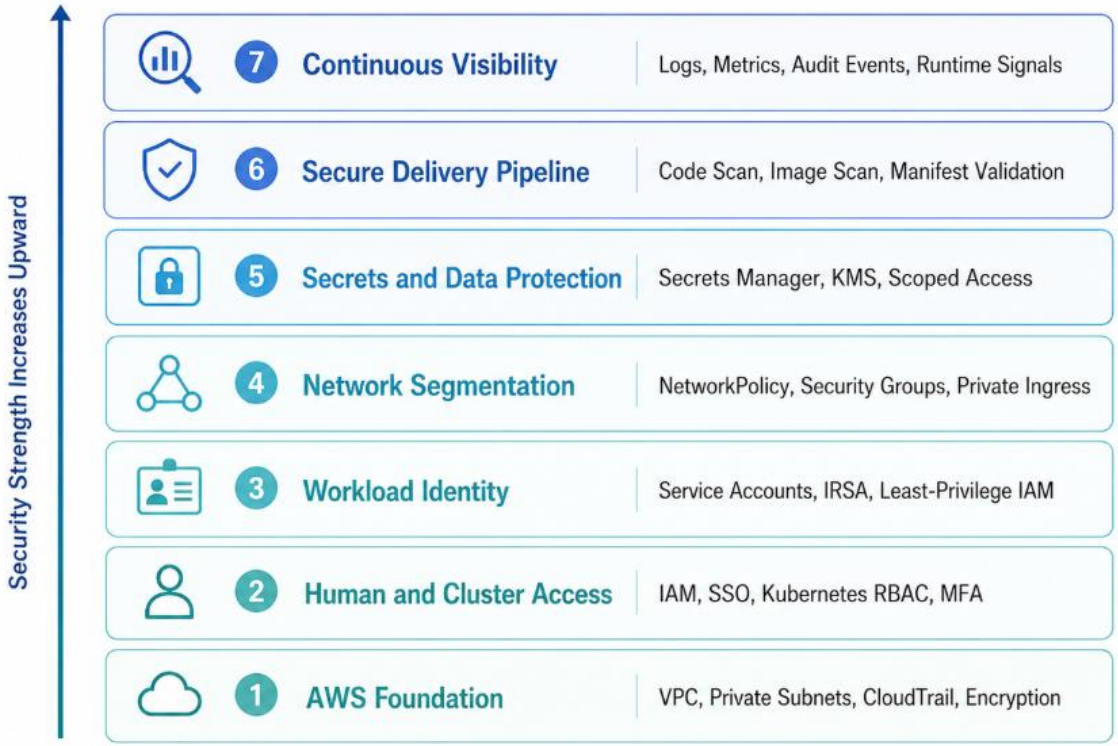

Figure 1. Layered Zero Trust model for production-ready Amazon EKS environments.
Each layer reduces implicit trust by enforcing identity, least privilege, segmentation, secure delivery, and continuous visibility.

4. AWS Account and Network Foundation

A secure EKS architecture starts before the cluster is created. AWS account structure, VPC design, IAM boundaries, logging, and encryption all influence the security posture of the cluster. Production workloads should be separated from development and testing environments, preferably through separate AWS accounts or strongly controlled boundaries. This separation limits blast radius and allows stricter controls for production systems.

Worker nodes should run in private subnets. Public subnets should be reserved for controlled entry points such as load balancers or NAT gateways. The EKS API endpoint should be restricted based on operational needs. For sensitive environments, administrative access should flow through approved private connectivity such as VPN, Direct Connect, or controlled bastion patterns.

Security groups should be designed with care. A common mistake is allowing the worker node security group to become too broad. If the node security group can reach a database, then every pod on that node may effectively share that network reachability. More granular network controls, including workload-specific access paths and pod-level segmentation patterns, help reduce unnecessary trust between applications and backend services.

A strong foundation should also include CloudTrail, VPC Flow Logs, KMS encryption, centralized logging, private connectivity to AWS services where appropriate, and clear separation between public and private ingress. These controls do not replace workload-level security, but they provide important layers of defense. This layered foundation follows the broader Zero Trust principle that trust should not depend only on network location, but should be supported by identity, policy, and continuous evaluation (Rose et al., 2020).

5. Cluster Access and Human Identity

Human access to production EKS clusters should be limited, role-based, and reviewed regularly. Cluster-admin access should not become a default permission for senior engineers or support users. Even trusted engineers should receive access based on their actual responsibility.

A practical access model separates users into groups. Platform administrators may manage cluster add-ons, node groups, ingress controllers, and policies. Application teams may manage deployments only within their namespaces. Support teams may receive read-only access for troubleshooting. Security teams may inspect audit logs, RBAC, policies, and workload configurations. CI/CD systems should use dedicated deployment identities rather than human credentials.

Kubernetes RBAC is the main control for this layer. RBAC should be mapped to IAM roles and enterprise identity processes where possible. Production access should be temporary when practical, especially for elevated permissions. Break-glass access should exist for emergencies, but it should be logged and reviewed after use.

This access model supports Zero Trust because it removes the assumption that a user who can authenticate to the cluster should automatically have broad access. Authentication proves who the user is. Authorization decides what the user is allowed to do. This approach is consistent with both Zero Trust principles and Kubernetes security practices that emphasize least privilege and careful access control (Rose et al., 2020; Shamim et al., 2020).

6. Workload Identity with IAM Roles for Service Accounts

Workload identity is one of the most important parts of EKS security. Applications often need access to AWS services such as S3, DynamoDB, SQS, SNS, KMS, Secrets Manager, and Parameter Store. A weak design gives these permissions to the worker node IAM role. That approach is convenient, but it creates a large trust boundary because many pods may share the same node-level permissions.

IAM Roles for Service Accounts provides a better design because it allows Kubernetes service accounts to be mapped to scoped AWS permissions. In a production EKS environment, this pattern helps separate workload permissions from node permissions. Instead of allowing every workload on a node to inherit broad access, each application can use its own service account and receive only the AWS permissions it needs.

Each application should use a dedicated Kubernetes service account. The service account should be linked to an IAM role that allows only the actions and resources required by that workload. This workload-specific identity model follows the least-privilege access principle recommended in Zero Trust architecture and Kubernetes security research (Rose et al., 2020; Shamim et al., 2020).

For example, a frontend service may not need any AWS data access. A backend API may need permission to read one database credential from Secrets Manager and access a specific S3 prefix. A reporting job may need read-only access to selected objects. A message processor may need permission to read from one queue and write to another. These should be separate IAM policies, not one shared role.

This approach reduces damage if a workload is compromised. The attacker receives only the permissions assigned to that workload identity. It also improves auditability because the platform team can see which application identity accessed which AWS resource.

7. Namespace, Pod, and Runtime Isolation

Namespaces are useful for organizing ownership, permissions, quotas, secrets, and network policies. They should not be treated as complete security boundaries by themselves, but they are still an important part of a production EKS design.

Each production namespace should have a clear owner. It should use dedicated service accounts, RBAC bindings, resource quotas, limit ranges, network policies, and scoped secrets. The default namespace should not be used for application workloads. Separating workloads by namespace makes it easier to apply least privilege and understand responsibility.

Pod security is equally important. Containers should avoid privileged mode, host networking, host PID access, hostPath volumes, unnecessary Linux capabilities, and root execution unless there is a documented reason. Images should be minimal, patched, and scanned before deployment.

Container security research treats host-container, inter-container, and container-host protection as separate concerns that must be addressed together rather than assumed safe by default (Sultan et al., 2019). Kubernetes

container runtime research also shows that security and performance choices in container runtimes can affect production design decisions, especially when teams balance isolation strength with operational efficiency (Viktorsson et al., 2020).

In practice, teams may use policy engines such as OPA Gatekeeper or Kyverno to enforce additional controls that are specific to their environment. The right approach is gradual enforcement. Start with audit mode, identify violations, work with application teams, and then enforce baseline requirements. This avoids sudden production disruption while still moving the platform toward stronger security.

8. Network Segmentation and Service Communication

Network segmentation is a core Zero Trust control. In many Kubernetes clusters, pods can communicate broadly across namespaces unless policies are configured to restrict them. This default openness can increase the impact of a compromised workload.

Kubernetes NetworkPolicy should be used to define allowed traffic between pods and namespaces. A practical model is to begin with sensitive namespaces and move toward deny-by-default communication. After that, explicit allow rules can be created for required service flows.

A typical production flow may allow traffic from the ingress controller to frontend services, from frontend services to backend APIs, and from backend APIs to selected databases or queues. Monitoring components may be allowed to scrape metrics endpoints. Log agents may be allowed to forward logs. Other traffic should be restricted unless there is a clear requirement.

For access to AWS resources such as RDS or ElastiCache, workload-level network control provides an additional layer of protection. This is useful when different workloads run on shared compute but require different network permissions. Network segmentation also aligns with Kubernetes security practices that recommend limiting unnecessary communication paths and applying least-privilege access within the cluster (Shamim et al., 2020).

Ingress should also be separated by exposure level. Public services should use controlled public ingress. Internal services should use private load balancers, internal DNS, and restricted security groups. TLS should be enforced for external traffic. For highly sensitive service-to-service communication, mutual TLS through a service mesh may be considered, but it should be adopted only when the team is prepared to operate the additional complexity.

9. Secrets Management and Sensitive Configuration

Secrets are one of the easiest places for a Kubernetes platform to become weak. Database passwords, API tokens, private keys, certificates, and connection strings should not be stored in source code, container images, or plain deployment files. Kubernetes Secrets provide a native object type for sensitive values, but they still require careful control through RBAC, namespace isolation, encryption, and access review.

AWS Secrets Manager and AWS Systems Manager Parameter Store are practical options for storing sensitive application configuration. For EKS workloads, secret access should be tied to workload identity so that applications retrieve only the values they are allowed to use. This is stronger than copying credentials into deployment files or sharing the same secret broadly across multiple applications.

The strongest design combines external secret storage with workload identity. A workload should retrieve only the secrets it needs, and that permission should be tied to its service account through scoped IAM access. A frontend service should not be able to read backend database credentials. A reporting job should not be able to retrieve API tokens unless that is part of its business function. A lower-environment workload should not have access to production secrets.

Secrets management should follow the same Zero Trust principle used throughout the platform: access should be granted based on workload identity, least privilege, and the actual need of the application rather than broad environment-level trust (Rose et al., 2020; Shamim et al., 2020).

Secret rotation should also be part of normal operations. Rotation is difficult when secrets are copied into many places or manually embedded in manifests. Keeping secrets in a managed store and retrieving them through controlled workload identity makes rotation and auditing easier.

10. Secure CI/CD and Software Supply Chain

A Zero Trust EKS architecture must include the software delivery pipeline. If a CI/CD system can deploy to production, it is part of the production trust boundary. A weak pipeline can bypass many controls that exist inside the cluster.

The pipeline should scan source code, dependencies, container images, and Kubernetes manifests before deployment. Secret scanning should prevent credentials from entering repositories. Image scanning should identify known vulnerabilities in base images and application layers. Manifest scanning should detect risky workload settings such as privileged containers, missing resource limits, hostPath mounts, or public service exposure.

The software delivery pipeline should also be treated as part of the trust boundary. Prior research on package managers and software update systems has shown that distribution channels and update mechanisms can become security risks if integrity, authorization, and compromise recovery are not designed carefully (Cappos et al., 2008; Samuel et al., 2010).

Image promotion should be controlled. The image tested in staging should be the same image promoted to production, preferably referenced by immutable tags or image digests. Production clusters should pull images only from approved registries. Deployment identities should be scoped to the namespaces and actions they need. A pipeline should not be able to disable admission policies, modify unrelated secrets, or change cluster-wide RBAC.

Microservices also increase the need for disciplined release and version control because applications are composed of independently deployable services that must still behave as one reliable system. Prior work on microservices and DevOps highlights how cloud-native architectures depend on automation, deployment consistency, and operational feedback loops (Balalaie et al., 2016). API gateway and versioning patterns further show that controlled service evolution is important when multiple services and clients depend on one another (Akbulut & Perros, 2019).

11. Admission Control and Policy Enforcement

Admission control turns platform standards into enforceable rules. Without admission control, security depends heavily on manual review. Manual review is valuable, but it does not scale well in a fast-moving Kubernetes environment.

Admission policies can block privileged containers, hostPath volumes, missing resource limits, unapproved registries, use of the default service account, unsafe service types, and workloads that lack required ownership labels. They can also enforce baseline pod security requirements and prevent risky changes from reaching production.

Policy-as-code tools such as OPA Gatekeeper and Kyverno are commonly used for this type of control. The policy tool should be treated like production code. Policies should be version-controlled, reviewed, tested, and rolled out gradually. A good adoption path is audit first, then warning, then enforcement in lower environments, and finally enforcement in production.

Admission control supports the same Kubernetes security practices identified in prior research, including least privilege, restricted workload configuration, and reduction of unsafe defaults (Shamim et al., 2020). Admission control should also be practical. Policies that block legitimate delivery without clear guidance will create friction. The better model is to provide secure templates, documented exceptions, and reusable deployment patterns so teams can meet the policy without slowing down every release.

Figure 2: Secure Deployment Workflow for EKS Workloads

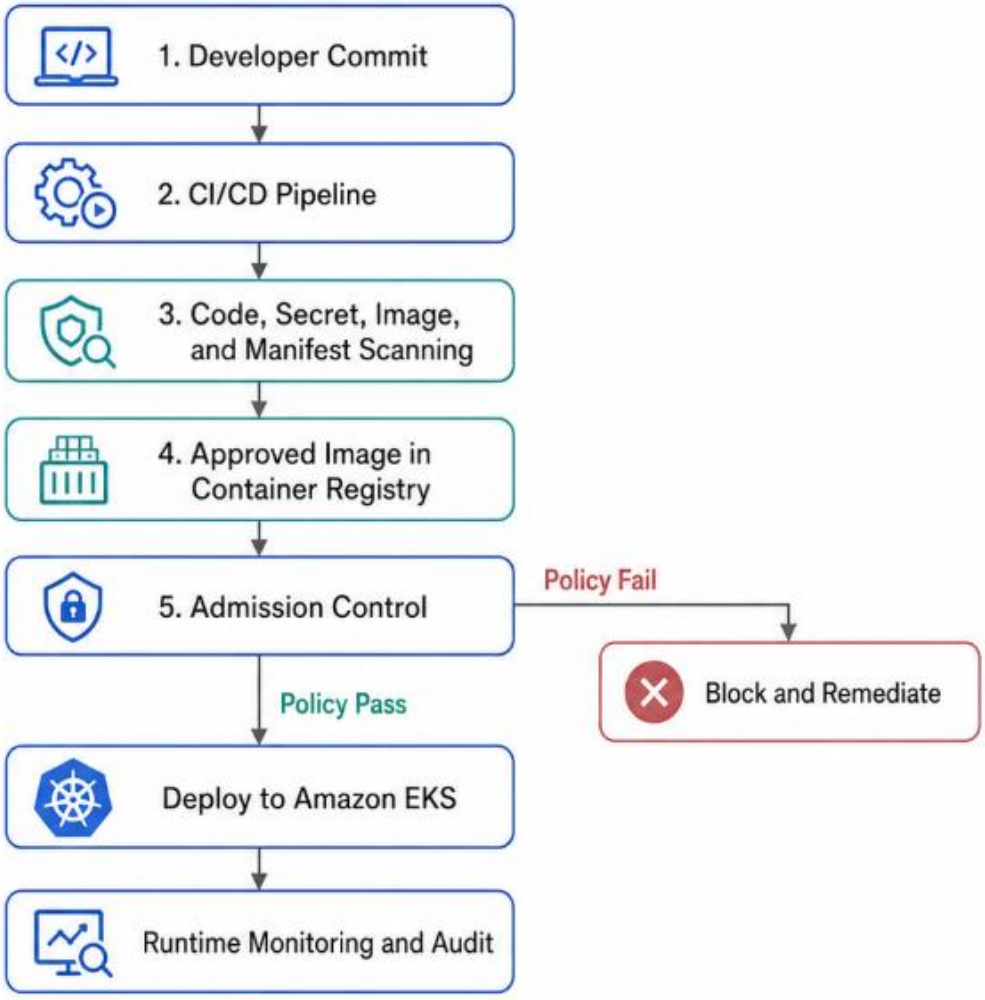

Figure 2. Secure deployment workflow for EKS workloads.
Security checks are applied before deployment, at admission time, and during runtime monitoring.

12. Observability, Audit, and Continuous Compliance

Zero Trust requires visibility. A platform team cannot verify access, investigate incidents, or improve controls without logs and telemetry. A production EKS environment should collect Kubernetes audit logs, EKS control plane logs, application logs, metrics, traces, CloudTrail events, VPC Flow Logs, container runtime events, and CI/CD deployment history.

These signals support both reliability and security. If a service begins calling an unexpected endpoint, if a user changes RBAC permissions, if a pod attempts unusual AWS API calls, or if a new image is deployed outside the normal process, the team should have enough evidence to investigate.

Continuous visibility supports Zero Trust by allowing teams to evaluate access decisions, detect abnormal behavior, and review whether policies are working as intended (Rose et al., 2020). Logs and metrics should be centralized outside the cluster where possible. If the cluster is unavailable or compromised, teams should still have access to historical records.

Alerting should focus on meaningful signals such as unusual privilege changes, failed authentication attempts, unexpected secret access, suspicious network flows, and abnormal workload restarts.

Continuous compliance should also become part of the operating model. IAM roles, RBAC bindings, network policies, security groups, secrets, admission policies, and image vulnerabilities should be reviewed regularly. Cluster and node upgrades should be planned. Findings should be prioritized based on risk, exploitability, and business impact.

13. Proposed Reference Architecture

A practical Zero Trust reference architecture for Amazon EKS can be organized across six layers.

At the AWS foundation layer, production workloads should run in isolated accounts or strongly separated environments. Worker nodes should be private. The EKS API endpoint should be restricted. CloudTrail, VPC Flow Logs, KMS encryption, and centralized logging should be enabled. Security groups should be specific rather than broad.

At the cluster identity layer, human users should access the cluster through IAM roles mapped to Kubernetes RBAC. Cluster-admin access should be limited. Namespace access should follow ownership. Break-glass permissions should be temporary and auditable.

At the workload identity layer, every application should use a dedicated Kubernetes service account. IRSA should provide scoped AWS permissions. The node IAM role should not become the permission source for every workload.

At the network layer, public ingress should be minimized. Internal services should use private access paths. NetworkPolicy should restrict pod-to-pod communication. Security groups for pods should be used where workloads need more granular AWS resource access.

At the secrets and configuration layer, sensitive values should be stored in AWS Secrets Manager, Systems Manager Parameter Store, or carefully managed Kubernetes Secrets. Access should be tied to workload identity. Secrets should not be stored in source code, images, or plain manifests.

At the delivery and runtime layer, CI/CD should scan code, dependencies, images, and manifests. Admission control should enforce workload standards. Logs, metrics, audit events, and security findings should be centralized for review and investigation.

This architecture does not remove all risk, but it reduces unnecessary trust. Each layer limits what can happen if another layer fails.

14. Challenges and Practical Adoption

The main challenge of applying Zero Trust in EKS is not the installation of tools, but the operational discipline required to sustain the model. Teams must understand ownership, workload identities, network dependencies, secret usage, and deployment flows. If controls are applied too aggressively, delivery teams may experience friction. If controls are too loose, the architecture does not reduce risk.

A practical adoption path should begin with visibility. Teams should first understand current IAM usage, namespace ownership, secrets, network flows, and deployment patterns. After that, controls can be added in stages. Start with dedicated service accounts and IRSA for sensitive workloads. Clean up RBAC. Add baseline admission policies. Restrict risky pod settings. Apply network policies to critical namespaces. Improve secret handling. Strengthen CI/CD scanning. Mature observability and access review.

Reusable templates are important. Helm charts, deployment standards, CI/CD stages, and policy examples should make the secure path easy for application teams. Zero Trust should not feel like an external checklist added at the end of delivery. It should become part of how the platform is built and operated.

15. Conclusion

Amazon EKS provides a strong foundation for production Kubernetes workloads, but the cluster alone does not create a secure platform. Production readiness depends on identity, least privilege, network segmentation, secrets protection, controlled delivery, policy enforcement, and continuous visibility.

A Zero Trust reference architecture helps teams reduce implicit trust across the platform. Human users should have role-based access. Workloads should use dedicated identities. AWS permissions should be scoped through IRSA. Network paths should be explicit. Secrets should be managed carefully. CI/CD systems should be treated as part of the production trust boundary. Admission control should prevent unsafe workloads from running. Observability should provide evidence for both operations and security.

The practical path is to use the controls already available across AWS and Kubernetes and combine them into a consistent operating model. When implemented carefully, Zero Trust makes EKS environments more secure, auditable, and resilient without requiring teams to abandon the speed and flexibility that made Kubernetes valuable in the first place.

REFERENCES:

1. Rose, S., Borchert, O., Mitchell, S., & Connelly, S. (2020). *Zero Trust Architecture*. National Institute of Standards and Technology, Special Publication 800-207. DOI: 10.6028/NIST.SP.800-207.
2. Shamim, M. S. I., Bhuiyan, F. A., & Rahman, A. (2020). XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices. *2020 IEEE Secure Development Conference (SecDev)*, 58–64. DOI: 10.1109/SecDev45635.2020.00025.
3. Sultan, S., Ahmad, I., & Dimitriou, T. (2019). Container Security: Issues, Challenges, and the Road Ahead. *IEEE Access, 7*, 52976–52996. DOI: 10.1109/ACCESS.2019.2911732.
4. Viktorsson, W., Klein, C., & Tordsson, J. (2020). Security-Performance Trade-offs of Kubernetes Container Runtimes. *2020 IEEE 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. DOI: 10.1109/MASCOTS50786.2020.9285946.
5. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes. *Communications of the ACM, 59*(5), 50–57. DOI: 10.1145/2890784.
6. Cappos, J., Samuel, J., Baker, S., & Hartman, J. H. (2008). A Look in the Mirror: Attacks on Package Managers. *Proceedings of the 15th ACM Conference on Computer and Communications Security*, 565–574. DOI: 10.1145/1455770.1455841.
7. Samuel, J., Mathewson, N., Cappos, J., & Dingledine, R. (2010). Survivable Key Compromise in Software Update Systems. *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 61–72. DOI: 10.1145/1866307.1866315.
8. Akbulut, A., & Perros, H. G. (2019). Software Versioning with Microservices through the API Gateway Design Pattern. *2019 9th International Conference on Advanced Computer Information Technologies (ACIT)*, 289–292. DOI: 10.1109/ACITT.2019.8779872.
9. Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016). Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software, 33*(3), 42–52. DOI: 10.1109/MS.2016.64.