1. What is CNI (Container Network Interface) in EKS?

What is CNI?
Think of CNI as the "networking rules" that tell containers how to talk to each other and the outside world.

If your containers are like houses in a neighborhood, CNI is like the postal system that decides:

- What address each house gets
- How mail gets delivered between houses
- Which houses can talk to each other
- How houses connect to the outside world

How CNI Works in EKS
When you create an EKS cluster, containers need to:

- Get their own IP addresses
- Communicate with other containers
- Access the internet
- Be accessible from outside the cluster

CNI plugins handle all of this networking automatically.

Default CNI in EKS: Amazon VPC CNI
EKS comes with Amazon's own CNI plugin called **VPC CNI**.

What it does:

- Gives each pod (container group) a real IP address from your VPC
- No complex networking translation needed
- Pods can directly communicate with other AWS services
- Works seamlessly with AWS security groups and network ACLs

Key benefits:

- **Simple**: Pods get regular AWS IP addresses
- **Fast**: Direct networking without extra layers
- **Secure**: Can use standard AWS networking security
- **Compatible**: Works with existing AWS networking tools

Alternative CNI Options
You can also use other CNI plugins like:

- **Calico**: Good for advanced network policies
- **Weave**: Simple overlay networking
- **Flannel**: Basic networking solution

2. How does AWS VPC CNI work in EKS?

The **AWS VPC CNI plugin** does the following:

- Each EKS worker node gets an **ENI** with **secondary IP addresses**.
- When pods are created, these secondary IPs are **assigned directly to pods**.
- Pods get **VPC-native IPs** (same IP range as EC2), enabling **native routing and security group support**.
- It uses the amazon-k8s-cni daemonset to manage ENIs/IPs dynamically.

✅ **Pros:** High performance, VPC integration

⚠️ **Cons:** Limited number of pods per node (based on ENI/IP limits)

1. The Hierarchy & Components

```
VPC (10.0.0.0/16)
├── Subnet A (10.0.1.0/24) - AZ-1a
├── Subnet B (10.0.2.0/24) - AZ-1b
└── EKS Cluster
    └── Worker Node (EC2 Instance)
        └── Pods (Containers)
```

2. Real Example with IP Addresses

VPC Level

- **VPC CIDR**: 10.0.0.0/16 (65,536 IP addresses available)
- **Subnet A**: 10.0.1.0/24 (256 IP addresses available)
- **Subnet B**: 10.0.2.0/24 (256 IP addresses available)

EC2 Worker Node Level

- **EC2 Instance**: Gets primary IP **10.0.1.100** from Subnet A
- **Instance Type**: m5.large (can support up to 10 ENIs, 30 IPs total)

Pod Level

Pod 1: Web Application

- **Pod IP**: 10.0.1.101 (from same subnet as EC2)
- **Container 1**: Nginx → shares Pod IP 10.0.1.101
- **Container 2**: App Server → shares Pod IP 10.0.1.101

Pod 2: Database

- **Pod IP**: 10.0.1.102
- **Container 1**: MySQL → uses Pod IP 10.0.1.102

Pod 3: Cache

- **Pod IP**: 10.0.1.103
- **Container 1**: Redis → uses Pod IP 10.0.1.103

3. How CNI Assigns IPs - The Process

Step 1: EC2 Instance Preparation

```
EC2 Instance (10.0.1.100)
├── Primary ENI: 10.0.1.100 (for the node itself)
├── Secondary IPs pre-allocated by CNI:
│   ├── 10.0.1.101 (ready for Pod 1)
│   ├── 10.0.1.102 (ready for Pod 2)
│   ├── 10.0.1.103 (ready for Pod 3)
│   └── ... (more IPs pre-allocated)
```

Step 2: When Pod is Created

1. **Kubernetes**: "I need to create a web pod"

2. **CNI Plugin**: "I'll assign IP 10.0.1.101 to this pod"
3. **CNI**: Creates virtual network interface in pod's namespace
4. **CNI**: Routes traffic from 10.0.1.101 to the pod

Step 3: Container Communication Within Pod

```
Pod 1 (IP: 10.0.1.101)
├── Container 1 (Nginx): localhost:80
├── Container 2 (App): localhost:3000
└── Shared Network Namespace
    └── All containers share IP: 10.0.1.101
```

4. Traffic Flow Examples

Example 1: Container to Container (Same Pod)

```
Nginx Container → App Container
localhost:80 → localhost:3000
(Uses loopback interface - super fast!)
```

Example 2: Pod to Pod (Same Node)

```
Web Pod (10.0.1.101) → Database Pod (10.0.1.102)
├── Source: 10.0.1.101:random_port
├── Destination: 10.0.1.102:3306
├── Route: Through EC2's internal networking
└── No internet involved - stays within EC2
```

Example 3: Pod to Pod (Different Nodes)

```
Node 1 Pod (10.0.1.101) → Node 2 Pod (10.0.2.101)
├── Source: 10.0.1.101:random_port
├── Destination: 10.0.2.101:3306
├── Route: Through VPC networking
├── Path: Subnet A → VPC Router → Subnet B
└── AWS handles the routing automatically
```

Example 4: External Traffic to Pod

```
Internet → Load Balancer → Pod
├── Internet: Any public IP
├── ALB: Public IP (managed by AWS)
├── ALB routes to: 10.0.1.101:80
└── Pod receives traffic on its VPC IP
```

5. IP Address Assignment Process

Before Pod Creation:

EC2 Instance: 10.0.1.100
Available IPs: 10.0.1.101, 10.0.1.102, 10.0.1.103...
Pods: None yet

After Creating 3 Pods:

EC2 Instance: 10.0.1.100
Pod 1: 10.0.1.101 (Web app)
Pod 2: 10.0.1.102 (Database)
Pod 3: 10.0.1.103 (Cache)
Available: 10.0.1.104, 10.0.1.105...

6. Key Points

Why Pods Share IPs but Containers Don't Get Individual IPs?

- **Pod**: Smallest deployable unit in Kubernetes
- **Containers in Pod**: Share network namespace (like roommates sharing apartment)
- **Different Ports**: Containers use different ports on same IP
  - Container 1: 10.0.1.101:80
  - Container 2: 10.0.1.101:3000

CNI's Job Summary:

1. **Pre-allocate** IP addresses from VPC subnet to EC2 instance
2. **Assign** one IP per pod (not per container)
3. **Route** traffic between pods using VPC networking
4. **Manage** IP address pool automatically

Traffic Never Leaves AWS:

- Pod-to-Pod communication uses VPC internal networking
- No NAT or translation needed
- Direct IP connectivity
- Very fast and secure

7. What Happens When Pod Scales?

Scaling Up:

Before: 3 pods using 10.0.1.101-103
After: 6 pods using 10.0.1.101-106
CNI automatically assigns 10.0.1.104-106 to new pods

If EC2 Runs Out of IPs:

Option 1: CNI requests more IPs from AWS (if available in subnet)
Option 2: New pods scheduled on different EC2 instances
Option 3: Cluster autoscaler creates new EC2 instances

---

3. What are the networking requirements for EKS?

Think of EKS Like Building a Neighborhood

Imagine you're building a secure neighborhood (EKS cluster) where different houses (containers) need to communicate safely.

1. VPC - Your Neighborhood Boundary

What it is: A Virtual Private Cloud - your own private section of AWS cloud.

Real example:

Your VPC: 10.0.0.0/16
(This gives you 65,536 IP addresses to work with)

Why you need it: Just like a neighborhood needs boundaries, your EKS cluster needs its own private network space.

2. Public and Private Subnets - Different Streets

You need **both types** across **multiple availability zones** (like different districts):

Public Subnets (Front Streets)

- **Purpose:** For load balancers and resources that need internet access
- **Example:**
  - Public Subnet A: 10.0.1.0/24 (in us-east-1a)

    - Public Subnet B: 10.0.2.0/24 (in us-east-1b)

Private Subnets (Back Streets)

- **Purpose:** For your worker nodes (EC2 instances) - more secure
- **Example:**
    - Private Subnet A: 10.0.10.0/24 (in us-east-1a)
    - Private Subnet B: 10.0.20.0/24 (in us-east-1b)

Why multiple AZs? If one availability zone goes down, your cluster keeps running in the other.

3. Internet Access - The Postal System

Internet Gateway (for public subnets)

- **What it does:** Allows direct internet access
- **Like:** Main post office for sending/receiving mail

NAT Gateway (for private subnets)

- **What it does:** Allows private resources to access internet for updates, but blocks incoming internet traffic
- **Like:** A security guard who lets residents go out but doesn't let strangers come in
- **Cost:** About $45/month per NAT Gateway

Real scenario: Your worker nodes need to download container images from DockerHub, but you don't want them directly accessible from internet.

4. Security Groups - The Security Guards

Think of these as **firewall rules** that control who can talk to whom.

EKS creates default security groups, but here's what they do:

Control Plane Security Group

Allows: Worker nodes to communicate with EKS API
Blocks: Random internet traffic

Worker Node Security Group

Allows:
- Communication between worker nodes
- Communication with control plane
- Your applications' specific ports

5. CIDR Ranges - Address Planning

You need to plan IP addresses for different components:

Example Planning:

```
VPC CIDR: 10.0.0.0/16 (65,536 addresses total)

├── Subnets: 10.0.1.0/24 to 10.0.20.0/24 (5,120 addresses)
├── Worker Nodes: Use subnet IPs (e.g., 10.0.10.5, 10.0.10.6)
├── Pods: Use additional IPs from same subnets (10.0.10.100, 10.0.10.101)
└── Services: Separate range (10.100.0.0/16) - internal only
```

Why separate ranges? So different components don't fight over the same IP addresses.

6. DNS Resolution - The Phone Book

What you need: Enable DNS resolution in your VPC settings.

What it does:

- Pods can find each other by name instead of remembering IP addresses
- Like having a phone book where you can call "database-service" instead of remembering "10.0.10.100"

Real-World Example Setup

Let's say you're building a simple web application:
VPC: my-app-vpc (10.0.0.0/16)

Public Subnets (for load balancers):
├── us-east-1a: 10.0.1.0/24
└── us-east-1b: 10.0.2.0/24

Private Subnets (for worker nodes):
├── us-east-1a: 10.0.10.0/24
└── us-east-1b: 10.0.20.0/24

Internet Access:
├── Internet Gateway → Public subnets
└── NAT Gateway → Private subnets (for worker nodes)

Worker Nodes:
├── Node 1: 10.0.10.5 (in private subnet A)
└── Node 2: 10.0.20.5 (in private subnet B)

Your Pods:
├── Web Pod: 10.0.10.100
├── API Pod: 10.0.10.101
└── DB Pod: 10.0.20.100

What AWS Does vs What You Need to Plan

AWS Handles Automatically:

- Basic security group rules for EKS
- Pod IP assignment (via CNI)
- Internal DNS for services

You Need to Plan:

- VPC and subnet sizing
- Whether you need NAT Gateways (costs money!)
- Custom security group rules for your apps
- How many availability zones to use

Common Beginner Mistakes

1. **Making subnets too small** - Plan for growth!
2. **Forgetting NAT Gateway costs** - $45/month each
3. **Not using multiple AZs** - Single point of failure
4. **Overlapping CIDR ranges** - Causes IP conflicts

Quick Checklist for Beginners

✅ VPC with enough IP addresses (at least /16)
✅ Public subnets in 2+ availability zones
✅ Private subnets in 2+ availability zones
✅ Internet Gateway attached to VPC
✅ NAT Gateway in each public subnet (for private subnet internet access)
✅ DNS resolution enabled in VPC
✅ Let EKS create default security groups (modify later if needed)

---

4. How do you implement network policies in EKS?

To implement **Network Policies** in EKS:

1. Install a **CNI that supports NetworkPolicy**, such as:
   - **Calico**
   - **Cilium**
2. Create Kubernetes NetworkPolicy resources that:
   - Restrict or allow pod-to-pod communication
   - Define ingress/egress rules based on labels, namespaces, IP blocks

🔒 AWS VPC CNI **does not support Kubernetes NetworkPolicy natively**, so you need an additional CNI like Calico.

---

5. What is the difference between ClusterIP, NodePort, and LoadBalancer services?

Type	Description	Use Case
ClusterIP	Default type. Exposes service **internally** within the cluster.	Internal microservices communication
NodePort	Exposes service on a **static port on each node IP** (e.g., nodeIP:30001).	Quick access for testing or dev
LoadBalancer	Provisions an **external AWS ELB** and routes traffic to pods via NodePort.	Exposing apps publicly on the internet

Think of Services as Different Ways to Reach Your Restaurant

Imagine your application is a restaurant, and you need different ways for customers to reach it.

1. ClusterIP - Internal Restaurant (Default)

What it is: A private dining room that only people inside the building can access.

How it works:

- Gets an internal IP address that only exists inside your EKS cluster
- Other pods can reach it, but nothing from outside the cluster can

Real Example:

Your Database Service:
- Service Name: my-database
- ClusterIP: 10.100.50.25 (internal only)
- Port: 3306

Other pods can connect using:
- my-database:3306 (by name)
- 10.100.50.25:3306 (by IP)

When to use:

- Database connections
- Internal APIs between microservices
- Cache services (Redis)
- Any service that should NOT be accessible from internet

Cost: FREE ✅

---

2. NodePort - Restaurant with Side Entrance
What it is: Opens a specific door (port) on every worker node that leads to your service.
How it works:

- Kubernetes picks a port between 30000-32767
- This port opens on EVERY worker node
- Traffic to any node's IP + that port goes to your service

Real Example:
Your Web App Service:
- Service Type: NodePort
- NodePort: 30080 (chosen by Kubernetes)
- Your worker nodes: 10.0.10.5, 10.0.20.5

You can access your app at:
- 10.0.10.5:30080 (node 1)
- 10.0.20.5:30080 (node 2)
- Both go to the same app!

When to use:

- Quick testing during development
- Internal company applications
- When you don't want to pay for a load balancer
- Temporary access to debug applications

Limitations:

- You need to know node IP addresses
- Port numbers are weird (30000+)
- No automatic failover if a node dies
- Security groups need to allow the NodePort

Cost: FREE ✅

---

3. LoadBalancer - Restaurant with Valet Service
What it is: AWS creates a fancy load balancer that automatically directs customers to your restaurant.
How it works:

- AWS creates an Application Load Balancer (ALB) or Network Load Balancer (NLB)
- Load balancer gets a public DNS name
- Automatically distributes traffic across healthy pods
- Handles SSL certificates, health checks, etc.

Real Example:
Your Public Web App:
- Service Type: LoadBalancer
- AWS creates: my-app-12345.us-east-1.elb.amazonaws.com
- Automatically routes to healthy pods across multiple nodes

When to use:

- Production applications that need internet access
- When you need high availability
- When you want SSL/HTTPS termination
- When you need automatic health checks

Cost: ~$16-25/month per load balancer 💰

Visual Comparison with Real Traffic Flow

ClusterIP Example:

Frontend Pod → my-database:3306 → Database Pod

(All internal, no external access possible)

NodePort Example:

Developer → 10.0.10.5:30080 → Worker Node → Your App Pod

(Direct access to node, then to pod)

LoadBalancer Example:

Internet User → my-app.elb.amazonaws.com → AWS Load Balancer → Healthy Worker Nodes → Your App Pods

(Professional setup with automatic failover)

Complete Real-World Example

Let's say you're building an e-commerce app with 3 components:

1. Database (ClusterIP)

yaml

```
Database Service:
- Type: ClusterIP
- Name: postgres-service
- Internal IP: 10.100.0.10
- Port: 5432
- Accessible by: Only other pods in cluster
```

2. Admin Panel (NodePort)

yaml

```
Admin Service:
- Type: NodePort
- NodePort: 30090
- Access via: 10.0.10.5:30090
- Use case: Internal admin access for your team
```

3. Customer Website (LoadBalancer)

yaml

```
Web Service:
- Type: LoadBalancer
- AWS ELB: mystore-123.us-east-1.elb.amazonaws.com
- Access via: Public internet
- Features: SSL, auto-scaling, health checks
```

Decision Tree: Which Service Type to Choose?

Ask yourself:

1. **Is this for internal communication only?**
   - YES → Use **ClusterIP**
   - Examples: Database, cache, internal APIs
2. **Do you need external access but want to save money?**
   - YES → Use **NodePort**
   - Examples: Development testing, internal tools
3. **Do you need production-ready internet access?**
   - YES → Use **LoadBalancer**

    - Examples: Customer-facing websites, public APIs

Common Beginner Mistakes

❌ **Using LoadBalancer for everything** → Expensive!
❌ **Using NodePort in production** → Not professional, hard to manage
❌ **Exposing databases with LoadBalancer** → Security risk!
❌ **Forgetting security groups for NodePort** → Can't access the service

Pro Tips for Beginners

✅ **Start with ClusterIP** for internal services
✅ **Use LoadBalancer** only for services that need internet access
✅ **NodePort is great for quick testing** but not for production
✅ **One LoadBalancer can handle multiple services** using path-based routing
✅ **AWS EKS can automatically create load balancers** when you create LoadBalancer services

Quick Summary

Service Type	Internet Access	Cost	Best For
ClusterIP	❌ Internal only	FREE	Databases, internal APIs
NodePort	⚠️ If nodes are public	FREE	Development, testing
LoadBalancer	✅ Full internet	~$20/month	Production websites

Remember: You can change service types anytime - start simple and upgrade as needed!

1. What is the role of the CNI plugin in EKS?
A. Assign IAM roles to pods
B. Manage storage for containers
C. Allocate IPs and handle pod networking
D. Schedule pods on worker nodes

✅ **Correct Answer:** C. Allocate IPs and handle pod networking

---

2. Which CNI plugin is the default for Amazon EKS?
A. Flannel
B. Weave
C. Amazon VPC CNI
D. Calico

✅ **Correct Answer:** C. Amazon VPC CNI

---

3. In the Amazon VPC CNI, how do pods get IP addresses?
A. Shared with EC2 nodes
B. Through NAT translation
C. Assigned directly from the VPC subnet
D. Assigned randomly from Kubernetes internal pool
✅ **Correct Answer:** C. Assigned directly from the VPC subnet

---

4. What is a limitation of the AWS VPC CNI plugin?
A. Cannot integrate with IAM
B. Requires NAT Gateway for all traffic
C. Limited number of pods per EC2 node
D. Only works with Windows containers
✅ **Correct Answer:** C. Limited number of pods per EC2 node

---

5. Why do containers within a single pod share the same IP address?
A. Each container uses a different ENI
B. Containers are hosted on different subnets
C. They share the same network namespace
D. They are exposed via LoadBalancer services
✅ **Correct Answer:** C. They share the same network namespace

---

6. Which CNI plugin should you use if you want to enforce Kubernetes NetworkPolicies?
A. Amazon VPC CNI
B. Calico
C. Weave
D. kube-proxy
✅ **Correct Answer:** B. Calico

---

7. What happens when a pod is created on a worker node using the VPC CNI plugin?
A. An ENI is attached directly to the pod
B. A secondary IP from the node's ENI is assigned to the pod
C. The pod shares the node's primary IP
D. A NAT route is configured for pod access
✅ **Correct Answer:** B. A secondary IP from the node's ENI is assigned to the pod

---

8. In a VPC-based EKS cluster, what allows internet access for private subnets?
A. Elastic IP
B. Security Group
C. Internet Gateway
D. NAT Gateway
✅ **Correct Answer:** D. NAT Gateway

---

9. What is the function of the amazon-k8s-cni DaemonSet in EKS?
A. Logs pod traffic
B. Handles kubelet upgrades
C. Manages ENIs and secondary IPs on nodes
D. Provisions Load Balancers
✅ **Correct Answer:** C. Manages ENIs and secondary IPs on nodes

---

10. What is the main advantage of using VPC-native IPs for pods in EKS?
A. Pods can communicate via private DNS
B. Pods avoid port conflicts

C. Pods can directly use AWS security groups and networking tools
D. Pods automatically get external load balancers
✅ **Correct Answer:** C. Pods can directly use AWS security groups and networking tools