Complete Beginner's Guide to Networking for DevOps

Table of Contents

What is Networking?

The Simple Explanation

Think of networking like a postal system for computers. Just as your home has an address so mail carriers know where to deliver letters, every computer and application needs an "address" so data knows where to go.

Real-World Analogy:

- Your home address = IP Address (like 192.168.1.10)
- The postal service = Network Infrastructure
- A letter = Data packet
- Different rooms in your house = Different services (web server, database, etc.)

Basic Terms You Need to Know

IP Address: A unique number that identifies a device on a network.

- Example: `192.168.1.100`
- Think of it as a phone number or home address
- Every device that connects to a network needs one

Port: A specific "door" on a device where services listen for connections.

- Example: Port 80 is for websites (HTTP), Port 443 is for secure websites (HTTPS)
- Think of ports as different mailboxes at the same address
- Your computer can have thousands of ports

Subnet: A section of a larger network, like neighborhoods in a city.

- Example: `10.0.1.0/24` means addresses from 10.0.1.0 to 10.0.1.255
- Helps organize and secure your network
- Makes network management easier

DNS (Domain Name System): Converts website names to IP addresses.

- You type: `www.google.com`
- DNS translates it to: `142.250.80.46`
- Think of it as a phone book for the internet

Firewall: Security gates that control what traffic can pass through.

- Blocks bad traffic, allows good traffic
- Like a security guard checking IDs at a building entrance
- Essential for keeping your systems safe

Why Do We Need Networking?

Problem 1: My Application Can't Work Alone

The Situation: You've built a web application. It has:

- A website (frontend) that users see
- A server (backend) that processes requests
- A database that stores information

The Problem Without Networking: These three parts can't talk to each other! They're like three people in soundproof rooms trying to have a conversation.

How Networking Solves It: Networking creates "communication channels" between these parts:

```
User's Browser ↔ Frontend Server ↔ Backend Server ↔ Database
```

Each arrow represents a network connection that lets data flow.

Problem 2: Users Can't Reach My Application

The Situation: You've built an amazing app on your laptop. Now what?

The Problem Without Networking: Your app only works on your laptop. Nobody else can use it!

How Networking Solves It: Networking lets you:

1. Put your app on a server in the cloud (like Azure)
2. Give it a public address (like `myapp.com`)
3. Let anyone on the internet connect to it

Problem 3: My Application Needs to Stay Secure

The Situation: Your app handles sensitive data like passwords or credit cards.

The Problem Without Proper Networking: Without security controls, hackers could:

- Access your database directly
- Steal customer information
- Shut down your services

How Networking Solves It: Networking provides security layers:

- **Firewalls**: Block unauthorized access
- **Private Networks**: Hide sensitive parts from the internet
- **Encryption**: Scramble data so hackers can't read it
- **Access Controls**: Only allow specific traffic through

Problem 4: My Application is Slow or Crashes

The Situation: 100,000 users try to use your app at the same time. Disaster!

The Problem Without Smart Networking:

- One server gets overwhelmed
- Users see errors or super slow responses
- Your app crashes

How Networking Solves It:

- **Load Balancers**: Spread users across multiple servers
- **Multiple Regions**: Serve users from servers close to them
- **Traffic Management**: Route users to healthy servers automatically

---

How Does Networking Work?

The Journey of a Web Request

Let's follow what happens when you type `www.example.com` in your browser:

Step 1: DNS Lookup

```
You: "I want to go to www.example.com"
DNS: "That's at IP address 20.30.40.50"
```

Your computer asks a DNS server to translate the website name into an IP address.

Step 2: Routing

```
Your Computer → Internet → Example.com's Server
```

Your request travels through multiple routers (like passing a letter through several post offices) until it reaches the destination.

Step 3: Security Check

```
Firewall: "Is this request allowed?"
Firewall: "Yes, it's going to port 443 (HTTPS)"
Firewall: "Let it through!"
```

The server's firewall checks if your request is allowed.

Step 4: Load Balancer

```
Load Balancer: "We have 10 servers. Server #3 has the least work."
Load Balancer: "Sending this request to Server #3"
```

If the site is popular, a load balancer picks which server handles your request.

Step 5: Application Processing

```
Server #3: "Got a request for the homepage"
Server #3: "Let me get data from the database"
Database: "Here's the data you need"
Server #3: "Building the webpage..."
```

The server processes your request and talks to other services if needed.

Step 6: Response

```
Server #3 → Load Balancer → Internet → Your Computer
```

The response travels back through the same path, and you see the webpage!

Total Time: Usually 100-500 milliseconds (faster than you can blink!)

The OSI Model: The Seven Layers (Simplified)

Think of networking like sending a package:

Layer 7 - Application Layer (What you interact with)

- This is the application itself (web browser, email client)
- Example: You click "Send" in your email

Layer 4 - Transport Layer (How data is transferred)

- Breaks data into smaller chunks and ensures they all arrive
- Example: TCP makes sure all email chunks arrive in order

Layer 3 - Network Layer (Where to send data)

- Figures out the route from source to destination
- Example: Routers use IP addresses to forward your email

Layer 2 - Data Link Layer (Moving between nearby devices)

- Handles communication on the same local network
- Example: Your computer talking to your WiFi router

Layer 1 - Physical Layer (The actual wires and signals)

- The physical cables, WiFi radio waves, fiber optics
- Example: Electrical signals traveling through cables

Why This Matters:

- Different problems happen at different layers
- When troubleshooting, knowing which layer helps you fix issues faster
- DevOps tools often work at specific layers

When to Use Different Networking Solutions

Scenario 1: Small Application (Just Starting Out)

What You Have:

- One web server
- One database
- Less than 1,000 users

What Networking You Need:

- ✅ Simple Virtual Network (one subnet)
- ✅ Basic firewall rules
- ✅ One public IP address
- ❌ Don't need: Load balancer, multiple regions, complex routing

Why: Keep it simple! You don't need fancy networking when you're starting. Focus on getting your app working.

Example Setup:

```
Internet
   ↓
Firewall (allow port 443)
   ↓
Virtual Network (10.0.0.0/16)
   ↓
Subnet (10.0.1.0/24)
   ├── Web Server (10.0.1.10)
   └── Database (10.0.1.20)
```

Scenario 2: Growing Application (Getting Popular)

What You Have:

- Multiple web servers
- Database server
- 10,000+ users
- Starting to get slow during peak times

What Networking You Need:

- ✅ Virtual Network with multiple subnets
- ✅ Load Balancer (distribute traffic)
- ✅ Separate subnet for databases (security)
- ✅ Better firewall rules
- ❌ Don't need yet: Multiple regions, VPN

Why: You need to handle more traffic and separate your application into layers for better security and performance.

Example Setup:

```
Internet
   ↓
Load Balancer (distributes traffic)
   ↓
Virtual Network (10.0.0.0/16)
   ├── Web Subnet (10.0.1.0/24)
   │   ├── Web Server 1 (10.0.1.10)
   │   ├── Web Server 2 (10.0.1.11)
   │   └── Web Server 3 (10.0.1.12)
   └── Database Subnet (10.0.2.0/24)
       └── Database (10.0.2.10) - Private, not accessible from internet
```

Scenario 3: Enterprise Application (Serious Business)

What You Have:

- Dozens or hundreds of servers
- Multiple applications
- International users
- Strict security requirements
- Need 99.99% uptime

What Networking You Need:

- ✅ Hub-and-spoke network architecture
- ✅ Multiple regions (US, Europe, Asia)
- ✅ Global load balancer

- ✅ VPN or private connection to on-premises
- ✅ Advanced firewall with intrusion detection
- ✅ Private endpoints for databases
- ✅ DDoS protection

Why: You need enterprise-grade reliability, security, and performance. Users expect your app to always work, no matter where they are.

Example Setup:

```
Global Load Balancer (Azure Front Door)
    ├── US Region
    │   ├── Load Balancer
    │   ├── Web Servers (5x)
    │   └── Database (replicated)
    ├── Europe Region
    │   ├── Load Balancer
    │   ├── Web Servers (5x)
    │   └── Database (replicated)
    └── Asia Region
        ├── Load Balancer
        ├── Web Servers (5x)
        └── Database (replicated)
```

Scenario 4: Microservices on Kubernetes

What You Have:

- Application split into many small services
- Running in containers
- Using Kubernetes (AKS)
- Services need to talk to each other frequently

What Networking You Need:

- ✅ Container networking (Azure CNI or Kubenet)
- ✅ Ingress controller (manages incoming traffic)
- ✅ Network policies (control service-to-service traffic)
- ✅ Service mesh (optional, for advanced scenarios)
- ✅ Internal load balancing between services

Why: Microservices create complex communication patterns. You need networking that can handle hundreds of services talking to each other while keeping everything secure.

Example Setup:

```
Internet
   ↓
Ingress Controller
   ↓
Kubernetes Cluster
   ├── Frontend Service (3 pods)
   ├── API Gateway Service (2 pods)
   ├── User Service (5 pods)
   ├── Order Service (5 pods)
   ├── Payment Service (3 pods)
   └── Database Services
```

Where Does Networking Fit in Your Architecture?

Understanding Layers in Your Application

Think of your application like a building with different floors:

Floor 5: User Layer (Top Floor - What users see)

- Web browsers, mobile apps
- **Networking Here:** DNS, Global load balancers, CDN
- **Why:** Get users to the right server quickly

Floor 4: Presentation Layer (Public facing)

- Web servers, API gateways
- **Networking Here:** Load balancers, Firewalls, Public IPs
- **Why:** Handle incoming traffic, protect against attacks

Floor 3: Application Layer (Business logic)

- Application servers, Microservices
- **Networking Here:** Internal load balancers, Service discovery
- **Why:** Services need to communicate efficiently

Floor 2: Data Layer (Storage)

- Databases, Cache, File storage
- **Networking Here:** Private subnets, Private endpoints
- **Why:** Most sensitive layer, needs maximum security

Floor 1: Infrastructure Layer (Foundation)

- Virtual networks, Routers, Firewalls
- **Networking Here:** Virtual networks, Subnets, Route tables
- **Why:** Foundation everything else builds on

Where to Place Different Components

In the Public Subnet (Internet-facing):

- ✅ Load Balancers
- ✅ Web servers (if needed)
- ✅ Jump boxes (for admin access)
- ❌ Never: Databases, internal APIs, sensitive services

In the Private Subnet (Hidden from internet):

- ✅ Application servers
- ✅ Databases
- ✅ Internal APIs
- ✅ Cache servers
- ✅ Message queues

In the Management Subnet (Admin access):

- ✅ Monitoring tools
- ✅ CI/CD agents
- ✅ Admin tools
- Access only through VPN or Bastion

Example: E-commerce Website Architecture

Let's design a simple online store:

```
INTERNET
   ↓
[FIREWALL + DDoS Protection]
   ↓
[Global Load Balancer]
   ↓
PUBLIC SUBNET (10.0.1.0/24)
   ├── Application Gateway (Web Application Firewall)
   └── Jump Box (for admin access)
   ↓
APPLICATION SUBNET (10.0.2.0/24)
   ├── Web Server 1 (Product pages)
   ├── Web Server 2 (Shopping cart)
   ├── API Server 1 (Inventory API)
   └── API Server 2 (Payment API)
   ↓
DATA SUBNET (10.0.3.0/24)
   ├── Database Server (Customer data, Orders)
   ├── Redis Cache (Session data)
   └── File Storage (Product images)
   ↓
```

```
[Backup to another region]
```

Traffic Flow for "Buy a Product":

1. Customer visits website → Global Load Balancer → Application Gateway
2. Application Gateway routes to Web Server based on URL
3. Web Server calls API Server to check inventory
4. API Server queries Database
5. Database responds to API Server
6. API Server responds to Web Server
7. Web Server sends page to Customer

Security at Each Layer:

- Internet → Firewall blocks bad traffic
- Public Subnet → Only ports 80/443 open
- Application Subnet → Only accessible from public subnet
- Data Subnet → Only accessible from application subnet

Step-by-Step: Your First Network Setup

Scenario: Deploy a Simple Web Application

What we're building: A simple website with a web server and database in Azure.

What you'll need:

- Azure subscription (free tier works!)
- Basic understanding of the Azure portal
- 30 minutes of time

Step 1: Plan Your Network

Before touching Azure, answer these questions:

1. **What IP range should I use?**
   - Use `10.0.0.0/16` for your virtual network
   - This gives you 65,536 IP addresses (more than enough!)
2. **How many subnets do I need?**
   - Web Subnet: `10.0.1.0/24` (256 addresses)
   - Database Subnet: `10.0.2.0/24` (256 addresses)
3. **What needs to access what?**
   - Internet → Web Server (port 443 HTTPS)
   - Web Server → Database (port 5432 PostgreSQL)

    - Your computer → Web Server (port 22 SSH for management)
4. **What should be blocked?**
    - Internet → Database (BLOCK EVERYTHING)
    - Random ports on Web Server (BLOCK)

Write this down! This is your network design.

Step 2: Create the Virtual Network

What is this: A virtual network is like your own private network in Azure. Nothing can get in or out unless you allow it.

In Azure Portal:

1. Search for "Virtual Networks"
2. Click "Create"
3. Fill in:
    - **Name:** `myapp-vnet`
    - **Region:** Choose one close to you
    - **IP Address Space:** `10.0.0.0/16`
4. Add Subnets:
    - **Subnet 1 Name:** `web-subnet`
    - **Subnet 1 Range:** `10.0.1.0/24`
    - **Subnet 2 Name:** `database-subnet`
    - **Subnet 2 Range:** `10.0.2.0/24`
5. Click "Review + Create"

What just happened? You created a private network in Azure. Right now, it's empty, but it's ready to host your servers.

Step 3: Create Security Rules (Network Security Groups)

What is this: Think of NSGs as security guards that check every packet of data trying to enter or leave.

Create NSG for Web Subnet:

1. Search for "Network Security Groups"
2. Click "Create"
3. Name it: `web-nsg`
4. Add Inbound Rules:

    Rule 1: Allow HTTPS from Internet

    - Source: `Any`
    - Destination: `Any`
    - Port: `443`
    - Action: `Allow`

  - Priority: `100`
  - Name: `Allow-HTTPS`

  Rule 2: Allow SSH from your IP only

  - Source: `Your IP address` (Azure can detect this)
  - Destination: `Any`
  - Port: `22`
  - Action: `Allow`
  - Priority: `110`
  - Name: `Allow-SSH-MyIP`

  Rule 3: Deny everything else (implicit - already exists)
5. Associate this NSG with `web-subnet`

Create NSG for Database Subnet:

1. Create another NSG: `database-nsg`
2. Add Inbound Rules:

   Rule 1: Allow PostgreSQL from Web Subnet only

   - Source: `10.0.1.0/24` (web subnet)
   - Destination: `Any`
   - Port: `5432`
   - Action: `Allow`
   - Priority: `100`
   - Name: `Allow-PostgreSQL-WebSubnet`

   Rule 2: Deny everything else (implicit)
3. Associate this NSG with `database-subnet`

What just happened? You created security rules! Now:

- Only HTTPS traffic from internet can reach web servers
- Only you can SSH to web servers
- Only web servers can talk to the database
- Everything else is blocked

Step 4: Deploy Your Servers

Deploy Web Server:

1. Create a Virtual Machine
2. Put it in `web-subnet`
3. Give it a public IP address (so users can reach it)
4. Install your web application

Deploy Database Server:

1. Create Azure Database for PostgreSQL
2. Put it in `database-subnet`
3. **No public IP!** (it's private)
4. Configure connection string for web server

Step 5: Test Your Setup

Test 1: Can you reach the website?

```
Open browser → Go to your web server's IP
✅ Should work (port 443 is open)
```

Test 2: Can the web server reach the database?

```
SSH to web server → Try connecting to database
✅ Should work (web subnet can access database subnet)
```

Test 3: Can the internet reach the database?

```
Try connecting to database from your computer
❌ Should FAIL (this is good! Database is protected)
```

Step 6: Make It Better

Add a Domain Name:

1. Register a domain (like `myawesomeapp.com`)
2. Point it to your web server's IP address
3. Now users can visit `myawesomeapp.com`

Add SSL Certificate:

1. Get a free certificate from Let's Encrypt
2. Install it on your web server
3. Now your site is secure (HTTPS)

Add Monitoring:

1. Enable Azure Monitor
2. Set up alerts for high CPU, network errors
3. You'll know when something goes wrong

Congratulations! You've set up your first network in Azure!

Networking for DevOps Engineers

Your Role in Networking

What DevOps Engineers Do:

- Design and implement network infrastructure
- Automate network configuration
- Ensure applications can communicate
- Troubleshoot connectivity issues
- Balance security with accessibility

What You Don't Do:

- Manage physical network hardware (that's for on-premises network engineers)
- Configure office WiFi (that's for IT support)
- Deal with ISP issues (usually handled by vendors)

Phase 1: Planning (Before You Create Anything)

Goal: Understand what you're building before you build it.

Questions to Ask:

1. **About the Application:**
    - What services need to communicate?
    - How much traffic do we expect?
    - What's the acceptable downtime?
    - Where are our users located?
2. **About Security:**
    - What data is sensitive?
    - Who needs access to what?
    - What compliance rules apply (HIPAA, GDPR, etc.)?
    - What are the biggest security risks?
3. **About Scale:**
    - How many users now?
    - How many users in 1 year?
    - Do we need multiple regions?
    - Will we need to scale quickly?

Document Your Answers: Create a simple document with:

- Network diagram (even a hand-drawn one!)
- IP address allocation plan
- Security requirements

- List of services and their communication needs

Example Planning Document:

```
Application: Online Store
Expected Users: 10,000 now, 100,000 in 1 year
Regions: Start with US, expand to Europe later

Components:
- Web Frontend (needs public access)
- API Backend (needs access from Frontend)
- Database (needs access from Backend only)
- Redis Cache (needs access from Backend only)

Security Requirements:
- PCI compliance for payment data
- Database must be private
- All external traffic must use HTTPS
- Audit logs for all access

IP Plan:
- VNet: 10.0.0.0/16
- Web Subnet: 10.0.1.0/24
- App Subnet: 10.0.2.0/24
- Data Subnet: 10.0.3.0/24
```

Phase 2: Implementation (Building Your Network)

Goal: Turn your plan into actual infrastructure.

Step 1: Create Virtual Network

Why Infrastructure as Code?

- You can recreate your network if something breaks
- You can version control it (track changes)
- You can review changes before applying them
- You can use the same config in dev, staging, prod

Example with Terraform:

```
# This code creates a virtual network
resource "azurerm_virtual_network" "main" {
  name                = "myapp-vnet"
  location            = "eastus"
  resource_group_name = "myapp-rg"
  address_space       = ["10.0.0.0/16"]
}

# This code creates a subnet for web servers
resource "azurerm_subnet" "web" {
  name                 = "web-subnet"
  resource_group_name  = "myapp-rg"
  virtual_network_name = azurerm_virtual_network.main.name
  address_prefixes     = ["10.0.1.0/24"]
}

# This code creates a subnet for databases
resource "azurerm_subnet" "database" {
  name                 = "database-subnet"
  resource_group_name  = "myapp-rg"
  virtual_network_name = azurerm_virtual_network.main.name
  address_prefixes     = ["10.0.2.0/24"]
}
```

What this does: Creates your network in Azure. Run `terraform apply` and it builds everything!

Step 2: Set Up Security

Create Firewall Rules:

```
# Network Security Group for web servers
resource "azurerm_network_security_group" "web" {
  name                = "web-nsg"
  location            = "eastus"
  resource_group_name = "myapp-rg"

  # Allow HTTPS from anywhere
  security_rule {
    name                       = "AllowHTTPS"
    priority                   = 100
    direction                  = "Inbound"
    access                     = "Allow"
    protocol                   = "Tcp"
    source_port_range          = "*"
    destination_port_range     = "443"
    source_address_prefix      = "*"
    destination_address_prefix = "*"
  }
}
```

What this does: Creates security rules. Only HTTPS traffic is allowed to web servers.

Step 3: Deploy Load Balancer

Why You Need This:

- Distributes traffic across multiple servers
- If one server dies, traffic goes to healthy servers
- Handles SSL certificates for you

Example:

```
# Create a load balancer
az network lb create \
  --resource-group myapp-rg \
  --name myapp-lb \
  --sku Standard \
  --frontend-ip-name frontend \
  --backend-pool-name backend-pool

# Configure health checks
az network lb probe create \
  --resource-group myapp-rg \
  --lb-name myapp-lb \
  --name health-check \
  --protocol http \
  --port 80 \
  --path /health
```

What this does: Creates a load balancer that checks if your servers are healthy and distributes traffic.

Phase 3: Testing (Make Sure It Works)

Goal: Verify everything works before users see it.

Test 1: Basic Connectivity

```
# Can web servers reach the database?
# SSH to web server, then:
ping 10.0.2.10  # Database IP
# Should work!

# Can the internet reach the database directly?
ping <database-public-ip>
# Should NOT work! (database should be private)
```

Test 2: Security Rules

```
# Try accessing web server on port 22 (SSH) from random IP
# Should be blocked!

# Try accessing web server on port 443 (HTTPS) from anywhere
# Should work!
```

Test 3: Load Balancer

```
# Send 100 requests to the load balancer
for i in {1..100}; do
  curl https://myapp-lb.azure.com
done

# Check if requests are distributed across servers
# All servers should have received some requests
```

Test 4: Failure Scenarios

```
# Stop one web server
# Load balancer should detect it's unhealthy
# Traffic should go to remaining servers
# Users should see no errors!
```

Phase 4: Monitoring (Know What's Happening)

Goal: See problems before users complain.

What to Monitor:

1. **Connectivity:**
    - Are all services reachable?
    - Is the load balancer healthy?
    - Are any firewall rules blocking legitimate traffic?
2. **Performance:**
    - How fast are network requests?
    - Is bandwidth saturated?
    - Are there any bottlenecks?
3. **Security:**
    - Are there unusual traffic patterns?
    - Are there repeated login failures?
    - Is someone scanning for vulnerabilities?

Setting Up Monitoring:

```
# Enable Network Watcher (Azure's network monitoring tool)
az network watcher configure \
  --resource-group myapp-rg \
  --locations eastus \
  --enabled true

# Enable flow logs (see all network traffic)
az network watcher flow-log create \
  --resource-group myapp-rg \
  --name web-flow-logs \
  --nsg web-nsg \
  --storage-account mylogsaccount \
  --enabled true
```

Set Up Alerts:

- Alert when web server is unreachable
- Alert when database connections fail
- Alert when unusual traffic spikes occur
- Alert when firewall blocks spike

Phase 5: Maintenance (Keep It Running)

Goal: Keep your network healthy and secure.

Weekly Tasks:

- Review security logs for suspicious activity
- Check monitoring dashboards for anomalies
- Verify backup systems are working

Monthly Tasks:

- Review and clean up unused resources (saves money!)
- Update firewall rules if needs changed
- Check for Azure networking feature updates
- Review costs and optimize

Quarterly Tasks:

- Full security audit
- Disaster recovery test (can you recover from failure?)
- Capacity planning (will you run out of IP addresses?)
- Update documentation

When Something Breaks:

1. Don't panic!
2. Check monitoring dashboards (what's the error?)
3. Check recent changes (did someone change firewall rules?)
4. Test connectivity systematically (where exactly is it failing?)
5. Fix the issue
6. Document what happened and how you fixed it

Networking for Azure DevOps Engineers

Your Special Role

What Makes Azure DevOps Different: You're not just setting up networks manually—you're automating everything through CI/CD pipelines!

Your Job:

- Automate network infrastructure deployment
- Integrate network configuration into CI/CD
- Ensure pipelines can deploy to network resources
- Make network changes through code, not clicking

Understanding Your Tools

What is CI/CD?

- **CI (Continuous Integration):** Automatically test code when developers commit
- **CD (Continuous Deployment):** Automatically deploy code to production

How Networking Fits: Your pipeline needs to:

1. Deploy network infrastructure (VNets, subnets, NSGs)
2. Deploy applications to those networks
3. Configure DNS, load balancers, etc.
4. Test that everything can communicate

Setting Up Pipeline Networking

Problem: Your pipeline needs to deploy to Azure, but how does it connect?

Solution: Service Connections

Step 1: Create Service Principal

```
# This creates an identity for your pipeline
az ad sp create-for-rbac \
  --name "azure-devops-pipeline" \
  --role contributor \
  --scopes /subscriptions/{subscription-id}

# Output gives you:
# - Application ID (Client ID)
# - Password (Client Secret)
# - Tenant ID
# Write these down securely!
```

Step 2: Configure Azure DevOps

1. Go to Azure DevOps → Project Settings
2. Click "Service Connections"
3. Click "New Service Connection"
4. Choose "Azure Resource Manager"
5. Enter the IDs from Step 1

What just happened? Your pipeline can now deploy to Azure! It has permissions to create resources.

Pipeline Example: Deploy Network Infrastructure

Scenario: Every time you push code, automatically deploy network changes.

azure-pipelines.yml:

```
# This pipeline deploys your network infrastructure
```

```
trigger:
  - main  # Run when code is pushed to main branch

pool:
  vmImage: 'ubuntu-latest'

steps:
# Step 1: Install Terraform
- task: TerraformInstaller@0
  inputs:
    terraformVersion: 'latest'

# Step 2: Initialize Terraform
- task: TerraformTaskV2@2
  inputs:
    command: 'init'
    workingDirectory: '$(System.DefaultWorkingDirectory)/terraform'
  displayName: 'Terraform Init'

# Step 3: Plan changes (preview what will change)
- task: TerraformTaskV2@2
  inputs:
    command: 'plan'
    workingDirectory: '$(System.DefaultWorkingDirectory)/terraform'
  displayName: 'Terraform Plan'

#
```

Step 4: Apply changes (only if approved)

- task: TerraformTaskV2@2 inputs: command: 'apply' workingDirectory: '$(System.DefaultWorkingDirectory)/terraform' commandOptions: '-auto-approve' displayName: 'Terraform Apply' condition: and(succeeded(), eq(variables['Build.SourceBranch'], 'refs/heads/main'))

Step 5: Test the network

- script: | echo "Testing network connectivity..."

  ## Test if resources are reachable

  az network vnet show --name myapp-vnet --resource-group myapp-rg displayName: 'Verify Network Deployment'

```
**What this pipeline does:**
1. Installs Terraform
2. Shows what will change (preview)
3. Applies the changes to Azure
4. Verifies everything deployed correctly

**Best Practice:** Add a manual approval step before deploying to production!
```

~~~
### Handling Secrets Safely

**Problem:** Your database connection string contains a password. You can't put that in
code!

**Solution: Azure Key Vault**

**Step 1: Store Secret in Key Vault**
```bash
# Create Key Vault
az keyvault create \
  --name myapp-keyvault \
  --resource-group myapp-rg

# Store database password
az keyvault secret set \
  --vault-name myapp-keyvault \
  --name "DatabasePassword" \
  --value "super-secret-password"
~~~

Step 2: Use in Pipeline

```yaml
# In your pipeline, retrieve secrets from Key Vault
steps:
- task: AzureKeyVault@2
  inputs:
    azureSubscription: 'MyServiceConnection'
    KeyVaultName: 'myapp-keyvault'
    SecretsFilter: 'DatabasePassword'
  displayName: 'Get Secrets from Key Vault'

# Now you can use $(DatabasePassword) in your pipeline
- script: |
    echo "Configuring app with database password..."
    # The password is available as $(DatabasePassword)
  displayName: 'Configure Application'
```

What this does: Keeps secrets out of your code. Pipeline fetches them securely when needed.

Multi-Environment Deployments

Problem: You have Dev, Staging, and Production. Each needs different network config.

Solution: Environment Variables

Step 1: Define Environments

```
Dev:
  - VNet: 10.1.0.0/16
  - Small VMs
  - Minimal security
```

```
Staging:
  - VNet: 10.2.0.0/16
  - Medium VMs
  - Production-like security

Production:
  - VNet: 10.3.0.0/16
  - Large VMs
  - Maximum security
  - Multiple regions
```

Step 2: Pipeline with Stages

```yaml
stages:
# Deploy to Dev automatically
- stage: Dev
  variables:
    environment: 'dev'
    vnetPrefix: '10.1.0.0/16'
  jobs:
  - job: DeployDev
    steps:
    - script: echo "Deploying to Dev with VNet $(vnetPrefix)"

# Deploy to Staging with approval
- stage: Staging
  dependsOn: Dev
  variables:
    environment: 'staging'
    vnetPrefix: '10.2.0.0/16'
  jobs:
  - deployment: DeployStaging
    environment: 'staging'  # Requires approval in Azure DevOps
    strategy:
      runOnce:
        deploy:
          steps:
          - script: echo "Deploying to Staging with VNet $(vnetPrefix)"

# Deploy to Production with multiple approvals
- stage: Production
  dependsOn: Staging
  variables:
    environment: 'production'
    vnetPrefix: '10.3.0.0/16'
  jobs:
  - deployment: DeployProduction
    environment: 'production'  # Requires multiple approvals
    strategy:
      runOnce:
        deploy:
          steps:
```

```
          - script: echo "Deploying to Production with VNet $(vnetPrefix)"
```

What this does:

- Dev deploys automatically (for quick testing)
- Staging requires one approval
- Production requires multiple approvals (safety!)

Testing Network Changes in Pipelines

Problem: How do you know your network changes work before deploying?

Solution: Automated Tests

Step 1: Write Tests

```
#!/bin/bash
# test-network.sh

echo "Testing network configuration..."

# Test 1: Check if VNet exists
echo "Test 1: Checking VNet..."
az network vnet show --name myapp-vnet --resource-group myapp-rg
if [ $? -eq 0 ]; then
  echo "✅ VNet exists"
else
  echo "❌ VNet does not exist"
  exit 1
fi

# Test 2: Check if subnets exist
echo "Test 2: Checking subnets..."
SUBNETS=$(az network vnet subnet list --vnet-name myapp-vnet --resource-group myapp-rg --
query "[].name" -o tsv)
if [[ $SUBNETS == *"web-subnet"* ]]; then
  echo "✅ Web subnet exists"
else
  echo "❌ Web subnet missing"
  exit 1
fi

# Test 3: Check NSG rules
echo "Test 3: Checking NSG rules..."
HTTPS_RULE=$(az network nsg rule show --nsg-name web-nsg --resource-group myapp-rg --name
AllowHTTPS)
if [ $? -eq 0 ]; then
  echo "✅ HTTPS rule exists"
else
  echo "❌ HTTPS rule missing"
  exit 1
fi
```

```
echo "All tests passed! ✅"
```

Step 2: Add to Pipeline

```
- script: |
    chmod +x test-network.sh
    ./test-network.sh
  displayName: 'Run Network Tests'
```

What this does: Automatically verifies your network is configured correctly. If tests fail, deployment stops!

Networking for AKS Administrators

What is AKS?

Simple Explanation: AKS (Azure Kubernetes Service) is like a smart manager for your containers. Instead of running one app on one server, you run hundreds of small containers that AKS manages automatically.

Why Networking is Different:

- You have hundreds of containers, not just a few servers
- Containers come and go constantly (they're temporary)
- Containers need to talk to each other all the time
- You need to manage traffic to thousands of "mini-apps"

Choosing Your Network Model

The Big Decision: Azure CNI vs Kubenet

Think of it like choosing between two road systems for your city:

Option 1: Azure CNI (Integrated Network)

- Every container (pod) gets a real IP address from your Azure network
- Like giving every house in your city a real street address

Pros:

- ✅ Containers can talk directly to Azure services
- ✅ You can use Azure networking features
- ✅ Simpler to understand
- ✅ Better for production

Cons:

- ❌ Uses LOTS of IP addresses
- ❌ Need to plan IP space carefully
- ❌ More expensive

When to use: Production applications, when you need Azure integration, when you have plenty of IP addresses

Option 2: Kubenet (NAT Network)

- Containers get fake internal IP addresses
- Like using apartment numbers inside a building

Pros:

- ✅ Saves IP addresses
- ✅ Cheaper
- ✅ Good for small clusters

Cons:

- ❌ Containers hidden behind NAT (harder to access directly)
- ❌ Limited Azure integration
- ❌ More complex routing

When to use: Development environments, small clusters, IP address conservation is critical

Step-by-Step: Create Your First AKS Cluster

Scenario: Deploy a microservices application on Kubernetes

Step 1: Plan Your IP Addresses

Important Math: Calculate how many IPs you need!

```
Formula: (Number of Nodes × Max Pods per Node) + Node IPs + Buffer

Example:
- 5 nodes
- 30 pods per node maximum
- Calculation: (5 × 30) + 5 + 50 = 205 IPs needed
- Use /24 subnet (256 IPs) to be safe
```

Your Plan:

```
VNet: 10.0.0.0/16 (65,536 IPs total)
├── AKS Node Subnet: 10.0.1.0/24 (256 IPs)
├── AKS Pod Subnet: 10.0.2.0/23 (512 IPs)
├── App Gateway Subnet: 10.0.4.0/24 (256 IPs)
└── Database Subnet: 10.0.5.0/24 (256 IPs)
```

Step 2: Create the AKS Cluster

```
# Create resource group
az group create --name myaks-rg --location eastus

# Create virtual network
```

```
az network vnet create \
  --resource-group myaks-rg \
  --name aks-vnet \
  --address-prefix 10.0.0.0/16 \
  --subnet-name aks-subnet \
  --subnet-prefix 10.0.1.0/24

# Get subnet ID (you'll need this)
SUBNET_ID=$(az network vnet subnet show \
  --resource-group myaks-rg \
  --vnet-name aks-vnet \
  --name aks-subnet \
  --query id -o tsv)

# Create AKS cluster with Azure CNI
az aks create \
  --resource-group myaks-rg \
  --name myaks-cluster \
  --network-plugin azure \
  --vnet-subnet-id $SUBNET_ID \
  --docker-bridge-address 172.17.0.1/16 \
  --dns-service-ip 10.2.0.10 \
  --service-cidr 10.2.0.0/24 \
  --enable-managed-identity \
  --node-count 3 \
  --generate-ssh-keys
```

What each setting means:

- `--network-plugin azure`: Use Azure CNI (integrated networking)
- `--vnet-subnet-id`: Which subnet to put nodes in
- `--dns-service-ip`: IP for Kubernetes DNS service (must be in service-cidr)
- `--service-cidr`: IP range for Kubernetes services (cluster-internal IPs)
- `--node-count 3`: Start with 3 servers

Step 3: Connect to Your Cluster

```
# Get credentials to access cluster
az aks get-credentials \
  --resource-group myaks-rg \
  --name myaks-cluster

# Verify you can connect
kubectl get nodes

# You should see:
# NAME                                STATUS   ROLE    AGE   VERSION
# aks-nodepool1-12345678-vmss000000   Ready    agent   5m    v1.28.0
# aks-nodepool1-12345678-vmss000001   Ready    agent   5m    v1.28.0
# aks-nodepool1-12345678-vmss000002   Ready    agent   5m    v1.28.0
```

Congratulations! You have a Kubernetes cluster!

Deploying Your First Application

Scenario: Deploy a simple web app with a database

Step 1: Deploy a Database (PostgreSQL)

```yaml
# database.yaml
apiVersion: v1
kind: Service
metadata:
  name: postgres
spec:
  ports:
  - port: 5432
  selector:
    app: postgres
  clusterIP: None  # Headless service
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
spec:
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
      - name: postgres
        image: postgres:13
        ports:
        - containerPort: 5432
        env:
        - name: POSTGRES_PASSWORD
          value: "mypassword"  # In real life, use secrets!
```

Deploy it:

```
kubectl apply -f database.yaml

# Check if it's running
kubectl get pods
# NAME                          READY   STATUS    RESTARTS   AGE
# postgres-5f7b8c9d6f-xyz12     1/1     Running   0          30s
```

What just happened?

- Created a PostgreSQL database inside your cluster
- It got an internal IP address automatically
- Other apps in the cluster can reach it using name "postgres"

Step 2: Deploy Web Application

```yaml
# webapp.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
spec:
  replicas: 3  # Run 3 copies for reliability
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
      - name: webapp
        image: myregistry/webapp:latest
        ports:
        - containerPort: 8080
        env:
        - name: DATABASE_URL
          value: "postgres://postgres:5432"  # Using service name!
---
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
spec:
  type: LoadBalancer  # This gets a public IP!
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: webapp
```

Deploy it:

```
kubectl apply -f webapp.yaml

# Wait for public IP to be assigned
kubectl get service webapp-service --watch

# When EXTERNAL-IP shows an IP (not <pending>), you're ready!
# NAME              TYPE           EXTERNAL-IP      PORT(S)        AGE
# webapp-service    LoadBalancer   20.30.40.50      80:30123/TCP   2m
```

Test it:

```
# Visit the external IP in your browser
curl http://20.30.40.50

# You should see your web app! 🎉
```

What just happened?

- Deployed 3 copies of your web app
- Azure created a load balancer automatically
- Load balancer got a public IP
- Traffic is distributed across 3 app copies
- App can connect to database using "postgres" name

Setting Up Ingress (Better Than LoadBalancer)

Problem: LoadBalancer gives you one IP per service. If you have 10 services, that's 10 IPs and 10 load balancers. Expensive!

Solution: Ingress Controller - One load balancer for all your services!

Think of it like:

- LoadBalancer = Each apartment has its own mailbox on the street
- Ingress = One central mailbox for the building that sorts mail by apartment number

Step 1: Install NGINX Ingress Controller

```
# Add Helm repository (package manager for Kubernetes)
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
helm repo update

# Install NGINX Ingress
helm install ingress-nginx ingress-nginx/ingress-nginx \
  --namespace ingress-nginx \
  --create-namespace \
  --set controller.service.annotations."service\.beta\.kubernetes\.io/azure-load-balancer-
health-probe-request-path"=/healthz
```

Wait for it to get an IP:

```
kubectl get service -n ingress-nginx --watch

# When you see an EXTERNAL-IP, note it down!
# This is your ONE IP for ALL services
```

Step 2: Create Ingress Rules

```yaml
# ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: myapp-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - host: myapp.com
    http:
      paths:
      # Route /api to API service
      - path: /api
        pathType: Prefix
        backend:
          service:
            name: api-service
            port:
              number: 8080
      # Route / to frontend service
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend-service
            port:
              number: 80
```

Deploy it:

```
kubectl apply -f ingress.yaml
```

What this does:

- `myapp.com/api` → goes to api-service
- `myapp.com/` → goes to frontend-service
- One IP, multiple services!

Network Policies (Security Between Pods)

Problem: By default, any pod can talk to any other pod. That's not secure!

Solution: Network Policies - Firewall rules for pods

Example: Only frontend can talk to API

```yaml
# network-policy.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: api-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: api  # Apply to pods with label app=api
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend  # Only allow from frontend pods
    ports:
    - protocol: TCP
      port: 8080
```

Deploy it:

```bash
# First, enable network policies on your cluster
az aks update \
  --resource-group myaks-rg \
  --name myaks-cluster \
  --network-policy azure

# Then apply the policy
kubectl apply -f network-policy.yaml
```

What this does:

- ✅ Frontend pods CAN talk to API pods
- ❌ Database pods CANNOT talk to API pods
- ❌ Random pods CANNOT talk to API pods

Test it:

```
# From frontend pod - should work
kubectl exec -it frontend-pod -- curl http://api-service:8080

# From database pod - should fail
kubectl exec -it database-pod -- curl http://api-service:8080
# Error: connection refused (this is good!)
```

Connecting to Azure Services (Private Endpoints)

Problem: Your app needs Azure SQL Database, but you don't want it exposed to the internet.

Solution: Private Endpoint - Put Azure services INSIDE your network!

Step 1: Create Azure SQL with Private Endpoint

```
# Create SQL Server
az sql server create \
  --name myapp-sqlserver \
  --resource-group myaks-rg \
  --location eastus \
  --admin-user sqladmin \
  --admin-password "SuperSecret123!"

# Disable public access
az sql server update \
  --name myapp-sqlserver \
  --resource-group myaks-rg \
  --public-network-access Disabled

# Create private endpoint
az network private-endpoint create \
  --name sql-private-endpoint \
  --resource-group myaks-rg \
  --vnet-name aks-vnet \
  --subnet database-subnet \
  --private-connection-resource-id $(az sql server show --name myapp-sqlserver --resource-
group myaks-rg --query id -o tsv) \
  --group-id sqlServer \
  --connection-name sql-connection

# Create private DNS zone
az network private-dns zone create \
  --resource-group myaks-rg \
  --name privatelink.database.windows.net

# Link DNS zone to VNet
az network private-dns link vnet create \
  --resource-group myaks-rg \
  --zone-name privatelink.database.windows.net \
  --name sql-dns-link \
  --virtual-network aks-vnet \
  --registration-enabled false
```

Step 2: Connect from AKS

```yaml
# Your app in AKS
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  template:
    spec:
      containers:
      - name: myapp
        image: myapp:latest
        env:
        - name: DB_HOST
          value: "myapp-sqlserver.privatelink.database.windows.net"
        - name: DB_PORT
          value: "1433"
```

What this does:

- SQL Server has NO public IP
- SQL Server is INSIDE your virtual network
- Only pods in your cluster can reach it
- Traffic never touches the internet!

Troubleshooting Common AKS Network Issues

Problem 1: "Pod can't reach the internet"

Symptoms:

```bash
kubectl exec -it my-pod -- curl https://google.com
# Hangs or times out
```

How to fix:

```bash
# Check 1: Is DNS working?
kubectl exec -it my-pod -- nslookup google.com
# If this fails, DNS is the problem

# Check 2: Can pod reach IP directly?
kubectl exec -it my-pod -- curl http://8.8.8.8
# If this works, it's definitely DNS

# Fix: Restart CoreDNS
kubectl rollout restart deployment/coredns -n kube-system

# Check 3: Are NSG rules blocking?
az network nsg rule list \
  --resource-group myaks-rg \
  --nsg-name aks-nsg \
  --query "[].{Name:name, Access:access, Direction:direction, Priority:priority}"

# If you see blocking rules, update them
```

Problem 2: "Can't access my service from outside"

Symptoms:

```
curl http://my-loadbalancer-ip
# Connection refused or timeout
```

How to fix:

```
# Check 1: Does service have external IP?
kubectl get service my-service
# EXTERNAL-IP should show an IP, not <pending>

# Check 2: Are pods running?
kubectl get pods -l app=my-service
# All should be Running, not Error or CrashLoopBackOff

# Check 3: Are pods healthy?
kubectl get endpoints my-service
# Should show pod IPs, not <none>

# Check 4: Test from inside cluster
kubectl run test-pod --image=busybox -it --rm -- wget -O- http://my-service
# If this works, problem is external access

# Check 5: Check NSG rules
az network nsg rule list --resource-group myaks-rg --nsg-name aks-nsg
# Look for rules blocking your service port
```

Problem 3: "Pods can't talk to each other"

Symptoms:

```
# From frontend pod
kubectl exec -it frontend-pod -- curl http://backend-service
# Connection refused
```

How to fix:

```
# Check 1: Is backend service running?
kubectl get service backend-service
kubectl get pods -l app=backend

# Check 2: Is there a network policy blocking it?
kubectl get networkpolicy
kubectl describe networkpolicy <policy-name>
```

```
# If policy is too restrictive, update it:
kubectl edit networkpolicy <policy-name>

# Check 3: Test DNS resolution
kubectl exec -it frontend-pod -- nslookup backend-service
# Should return cluster IP

# Check 4: Check endpoints
kubectl get endpoints backend-service
# Should show backend pod IPs
```

Common Problems and How to Fix Them

Problem 1: "I can't SSH to my server"

Symptoms:

```
ssh user@myserver.com
# Connection timed out
```

Possible Causes & Fixes:

Cause 1: NSG blocking SSH

```
# Check NSG rules
az network nsg rule list \
  --resource-group myapp-rg \
  --nsg-name my-nsg \
  --query "[?destinationPortRange=='22']"

# If no rule allows SSH, add one:
az network nsg rule create \
  --resource-group myapp-rg \
  --nsg-name my-nsg \
  --name AllowSSH \
  --priority 100 \
  --source-address-prefixes "YOUR_IP/32" \
  --destination-port-ranges 22 \
  --access Allow \
  --protocol Tcp
```

Cause 2: Server not in public subnet

```
# Check if server has public IP
az vm show \
  --resource-group myapp-rg \
  --name myserver \
  --query "publicIps"

# If no public IP and server is in private subnet, use jump box
ssh -J jumpbox@public-ip user@private-server-ip
```

Cause 3: SSH service not running

```
# Use Azure Serial Console or Run Command
az vm run-command invoke \
  --resource-group myapp-rg \
  --name myserver \
  --command-id RunShellScript \
  --scripts "systemctl status sshd"
```

Problem 2: "Website is slow"

Symptoms:

- Pages take 5+ seconds to load
- Database queries timeout
- Users complain

Diagnosis Steps:

Step 1: Check where the slowness is

```
# Test from outside
curl -w "Time: %{time_total}s\n" -o /dev/null -s https://myapp.com

# Test from inside network
az vm run-command invoke \
  --resource-group myapp-rg \
  --name web-server \
  --command-id RunShellScript \
  --scripts "curl -w 'Time: %{time_total}s\n' -o /dev/null -s http://database-server"
```

Step 2: Check network metrics

```
# Check bandwidth usage
az monitor metrics list \
  --resource
/subscriptions/{sub}/resourceGroups/{rg}/providers/Microsoft.Network/loadBalancers/{lb} \
  --metric ByteCount \
  --start-time 2024-01-01T00:00:00Z \
  --end-time 2024-01-01T23:59:59Z
```

Common Fixes:

Fix 1: Add load balancer (if you don't have one)

- Distributes traffic across multiple servers
- Prevents one server from being overwhelmed

Fix 2: Enable Accelerated Networking

```
az network nic update \
  --resource-group myapp-rg \
  --name my-nic \
  --accelerated-networking true
```

Fix 3: Move resources closer together

- Put web servers and database in same region
- Put web servers and database in same availability zone

Fix 4: Add caching

- Use Azure Redis Cache for frequently accessed data
- Use CDN for static files (images, CSS, JS)

Problem 3: "I'm getting 'Name or service not known' errors"

Symptoms:

```
ping myservice.com
# ping: myservice.com: Name or service not known
```

Possible Causes & Fixes:

Cause 1: DNS not configured

```
# Check DNS settings
cat /etc/resolv.conf

# Should show DNS servers like:
# nameserver 168.63.129.16 (Azure default DNS)

# If missing, configure DNS in VNet
az network vnet update \
  --resource-group myapp-rg \
  --name my-vnet \
  --dns-servers 168.63.129.16
```

Cause 2: DNS record doesn't exist

```
# Check if DNS record exists
nslookup myservice.com

# If not found, add DNS record
az network dns record-set a add-record \
  --resource-group myapp-rg \
  --zone-name myservice.com \
  --record-set-name www \
  --ipv4-address 20.30.40.50
```

Cause 3: Private DNS zone not linked to VNet

```
# Check private DNS zone links
az network private-dns link vnet list \
  --resource-group myapp-rg \
  --zone-name privatelink.database.windows.net

# If missing, create link
az network private-dns link vnet create \
  --resource-group myapp-rg \
  --zone-name privatelink.database.windows.net \
  --name my-link \
  --virtual-network my-vnet \
  --registration-enabled false
```

Problem 4: "SSL certificate errors"

Symptoms:

```
Your connection is not private
NET::ERR_CERT_AUTHORITY_INVALID
```

Possible Causes & Fixes:

Cause 1: Certificate expired

```
# Check certificate expiration
echo | openssl s_client -servername myapp.com -connect myapp.com:443 2>/dev/null | openssl
x509 -noout -dates

# Renew certificate (using Let's Encrypt)
certbot renew
```

Cause 2: Certificate doesn't match domain

```
# Check certificate domains
echo | openssl s_client -servername myapp.com -connect myapp.com:443 2>/dev/null | openssl
x509 -noout -text | grep DNS

# Make sure certificate includes your domain
# If not, generate new certificate with correct domain
```

Cause 3: Certificate not installed correctly

```
# Check certificate chain
openssl s_client -connect myapp.com:443 -showcerts

# Should show:
# - Server certificate
# - Intermediate certificate(s)
# - Root certificate

# If missing intermediate, install them
```

Problem 5: "Out of IP addresses!"

Symptoms:

```
# Can't create new resources
Error: No more IP addresses available in subnet
```

How to fix:

Option 1: Add more IP addresses to subnet (if possible)

```
# Check current subnet size
az network vnet subnet show \
  --resource-group myapp-rg \
  --vnet-name my-vnet \
  --name my-subnet \
  --query addressPrefix

# Expand subnet (only if adjacent IPs are free)
az network vnet subnet update \
  --resource-group myapp-rg \
  --vnet-name my-vnet \
  --name my-subnet \
  --address-prefixes "10.0.1.0/23"  # Doubles available IPs
```

Option 2: Create new subnet

```
# Create additional subnet
az network vnet subnet create \
  --resource-group myapp-rg \
  --vnet-name my-vnet \
  --name my-subnet-2 \
  --address-prefixes "10.0.2.0/24"

# Deploy new resources to new subnet
```

Option 3: Clean up unused IPs

```
# List all IP addresses
az network nic list \
  --resource-group myapp-rg \
  --query "[].{Name:name, IP:ipConfigurations[0].privateIPAddress, InUse:virtualMachine}"

# Delete IPs attached to deleted VMs
az network nic delete --resource-group myapp-rg --name unused-nic
```

Prevention: Always plan for growth! Use subnets larger than you think you need.

Quick Reference Guide

Essential Commands Cheat Sheet

Azure Networking:

```
# List all virtual networks
az network vnet list --output table

# Show VNet details
az network vnet show --resource-group myapp-rg --name my-vnet

# List subnets in a VNet
az network vnet subnet list --resource-group myapp-rg --vnet-name my-vnet --output table

# Check effective security rules
az network nic list-effective-nsg --resource-group myapp-rg --name my-nic

# Test connectivity between resources
az network watcher test-connectivity \
  --resource-group myapp-rg \
  --source-resource web-vm \
  --dest-resource database-vm \
  --dest-port 5432
```

Kubernetes Networking:

```
# List all services
kubectl get services --all-namespaces
```

```
# Show service details
kubectl describe service my-service

# List network policies
kubectl get networkpolicy --all-namespaces

# Test DNS from pod
kubectl run test --image=busybox -it --rm -- nslookup my-service

# Check pod connectivity
kubectl run test --image=nicolaka/netshoot -it --rm -- curl http Complete Beginner's Guide
to Networking for DevOps

## Table of Contents
1. [What is Networking?](#what-is-networking)
2. [Why Do We Need Networking?](#why-do-we-need-networking)
3. [How Does Networking Work?](#how-does-networking-work)
4. [When to Use Different Networking Solutions](#when-to-use-different-networking-solutions)
5. [Where Does Networking Fit in Your Architecture?](#where-does-networking-fit-in-your-
architecture)
6. [Step-by-Step: Your First Network Setup](#step-by-step-your-first-network-setup)
7. [Networking for DevOps Engineers](#networking-for-devops-engineers)
8. [Networking for Azure DevOps Engineers](#networking-for-azure-devops-engineers)
9. [Networking for AKS Administrators](#networking-for-aks-administrators)
10. [Common Problems and How to Fix Them](#common-problems-and-how-to-fix-them)

---

## What is Networking?

### The Simple Explanation

Think of networking like a postal system for computers. Just as your home has an address so
mail carriers know where to deliver letters, every computer and application needs an
"address" so data knows where to go.

**Real-World Analogy:**
- Your home address = IP Address (like 192.168.1.10)
- The postal service = Network Infrastructure
- A letter = Data packet
- Different rooms in your house = Different services (web server, database, etc.)

### Basic Terms You Need to Know

**IP Address**: A unique number that identifies a device on a network.
- Example: `192.168.1.100`
- Think of it as a phone number or home address
- Every device that connects to a network needs one

**Port**: A specific "door" on a device where services listen for connections.
- Example: Port 80 is for websites (HTTP), Port 443 is for secure websites (HTTPS)
- Think of ports as different mailboxes at the same address
- Your computer can have thousands of ports

**Subnet**: A section of a larger network, like neighborhoods in a city.
- Example: `10.0.1.0/24` means addresses from 10.0.1.0 to 10.0.1.255
- Helps organize and secure your network
- Makes network management easier

**DNS (Domain Name System)**: Converts website names to IP addresses.
- You type: `www.google.com`
- DNS translates it to: `142.250.80.46`
- Think of it as a phone book for the internet

**Firewall**: Security gates that control what traffic can pass through.
- Blocks bad traffic, allows good traffic
- Like a security guard checking IDs at a building entrance
- Essential for keeping your systems safe

---

## Why Do We Need Networking?

### Problem 1: My Application Can't Work Alone

**The Situation:**
You've built a web application. It has:
- A website (frontend) that users see
- A server (backend) that processes requests
- A database that stores information

**The Problem Without Networking:**
These three parts can't talk to each other! They're like three people in soundproof rooms
trying to have a conversation.

**How Networking Solves It:**
Networking creates "communication channels" between these parts:
```

User's Browser ←—→ Frontend Server ←—→ Backend Server ←—→ Database

```
Each arrow represents a network connection that lets data flow.

### Problem 2: Users Can't Reach My Application

**The Situation:**
You've built an amazing app on your laptop. Now what?

**The Problem Without Networking:**
Your app only works on your laptop. Nobody else can use it!

**How Networking Solves It:**
Networking lets you:
1. Put your app on a server in the cloud (like Azure)
2. Give it a public address (like `myapp.com`)
3. Let anyone on the internet connect to it
```

```
### Problem 3: My Application Needs to Stay Secure

**The Situation:**
Your app handles sensitive data like passwords or credit cards.

**The Problem Without Proper Networking:**
Without security controls, hackers could:
- Access your database directly
- Steal customer information
- Shut down your services

**How Networking Solves It:**
Networking provides security layers:
- **Firewalls**: Block unauthorized access
- **Private Networks**: Hide sensitive parts from the internet
- **Encryption**: Scramble data so hackers can't read it
- **Access Controls**: Only allow specific traffic through

### Problem 4: My Application is Slow or Crashes

**The Situation:**
100,000 users try to use your app at the same time. Disaster!

**The Problem Without Smart Networking:**
- One server gets overwhelmed
- Users see errors or super slow responses
- Your app crashes

**How Networking Solves It:**
- **Load Balancers**: Spread users across multiple servers
- **Multiple Regions**: Serve users from servers close to them
- **Traffic Management**: Route users to healthy servers automatically

---

## How Does Networking Work?

### The Journey of a Web Request

Let's follow what happens when you type `www.example.com` in your browser:

**Step 1: DNS Lookup**
```

You: "I want to go to www.example.com" DNS: "That's at IP address 20.30.40.50"

```
Your computer asks a DNS server to translate the website name into an IP address.

**Step 2: Routing**
```

Your Computer → Internet → Example.com's Server

```
Your request travels through multiple routers (like passing a letter through several post
offices) until it reaches the destination.

**Step 3: Security Check**
```

Firewall: "Is this request allowed?" Firewall: "Yes, it's going to port 443 (HTTPS)" Firewall: "Let it through!"

```
The server's firewall checks if your request is allowed.

**Step 4: Load Balancer**
```

Load Balancer: "We have 10 servers. Server #3 has the least work." Load Balancer: "Sending this request to Server #3"

```
If the site is popular, a load balancer picks which server handles your request.

**Step 5: Application Processing**
```

Server #3: "Got a request for the homepage" Server #3: "Let me get data from the database" Database: "Here's the data you need" Server #3: "Building the webpage..."

```
The server processes your request and talks to other services if needed.

**Step 6: Response**
```

Server #3 → Load Balancer → Internet → Your Computer

```
The response travels back through the same path, and you see the webpage!

**Total Time:** Usually 100-500 milliseconds (faster than you can blink!)

### The OSI Model: The Seven Layers (Simplified)

Think of networking like sending a package:

**Layer 7 - Application Layer** (What you interact with)
- This is the application itself (web browser, email client)
- Example: You click "Send" in your email

**Layer 4 - Transport Layer** (How data is transferred)
- Breaks data into smaller chunks and ensures they all arrive
- Example: TCP makes sure all email chunks arrive in order

**Layer 3 - Network Layer** (Where to send data)
- Figures out the route from source to destination
- Example: Routers use IP addresses to forward your email

**Layer 2 - Data Link Layer** (Moving between nearby devices)
- Handles communication on the same local network
- Example: Your computer talking to your WiFi router
```

```
**Layer 1 - Physical Layer** (The actual wires and signals)
- The physical cables, WiFi radio waves, fiber optics
- Example: Electrical signals traveling through cables

**Why This Matters:**
- Different problems happen at different layers
- When troubleshooting, knowing which layer helps you fix issues faster
- DevOps tools often work at specific layers

---

## When to Use Different Networking Solutions

### Scenario 1: Small Application (Just Starting Out)

**What You Have:**
- One web server
- One database
- Less than 1,000 users

**What Networking You Need:**
- ✅ Simple Virtual Network (one subnet)
- ✅ Basic firewall rules
- ✅ One public IP address
- ❌ Don't need: Load balancer, multiple regions, complex routing

**Why:**
Keep it simple! You don't need fancy networking when you're starting. Focus on getting your app working.

**Example Setup:**
```

Internet ↓ Firewall (allow port 443) ↓ Virtual Network (10.0.0.0/16) ↓ Subnet (10.0.1.0/24) ├── Web Server (10.0.1.10) └── Database (10.0.1.20)

```
### Scenario 2: Growing Application (Getting Popular)

**What You Have:**
- Multiple web servers
- Database server
- 10,000+ users
- Starting to get slow during peak times

**What Networking You Need:**
- ✅ Virtual Network with multiple subnets
- ✅ Load Balancer (distribute traffic)
- ✅ Separate subnet for databases (security)
- ✅ Better firewall rules
- ❌ Don't need yet: Multiple regions, VPN

**Why:**
```

```
You need to handle more traffic and separate your application into layers for better
security and performance.

**Example Setup:**
```

Internet ↓ Load Balancer (distributes traffic) ↓ Virtual Network (10.0.0.0/16) ├── Web Subnet (10.0.1.0/24) │ ├── Web Server 1 (10.0.1.10) │ ├── Web Server 2 (10.0.1.11) │ └── Web Server 3 (10.0.1.12) └── Database Subnet (10.0.2.0/24) └── Database (10.0.2.10) - Private, not accessible from internet

```
### Scenario 3: Enterprise Application (Serious Business)

**What You Have:**
- Dozens or hundreds of servers
- Multiple applications
- International users
- Strict security requirements
- Need 99.99% uptime

**What Networking You Need:**
- ✅ Hub-and-spoke network architecture
- ✅ Multiple regions (US, Europe, Asia)
- ✅ Global load balancer
- ✅ VPN or private connection to on-premises
- ✅ Advanced firewall with intrusion detection
- ✅ Private endpoints for databases
- ✅ DDoS protection

**Why:**
You need enterprise-grade reliability, security, and performance. Users expect your app to
always work, no matter where they are.

**Example Setup:**
```

Global Load Balancer (Azure Front Door) ├── US Region │ ├── Load Balancer │ ├── Web Servers (5x) │ └── Database (replicated) ├── Europe Region │ ├── Load Balancer │ ├── Web Servers (5x) │ └── Database (replicated) └── Asia Region ├── Load Balancer ├── Web Servers (5x) └── Database (replicated)

```
### Scenario 4: Microservices on Kubernetes

**What You Have:**
- Application split into many small services
- Running in containers
- Using Kubernetes (AKS)
- Services need to talk to each other frequently

**What Networking You Need:**
- ✅ Container networking (Azure CNI or Kubenet)
- ✅ Ingress controller (manages incoming traffic)
- ✅ Network policies (control service-to-service traffic)
- ✅ Service mesh (optional, for advanced scenarios)
- ✅ Internal load balancing between services
```

```
**Why:**
Microservices create complex communication patterns. You need networking that can handle
hundreds of services talking to each other while keeping everything secure.

**Example Setup:**
```

Internet ↓ Ingress Controller ↓ Kubernetes Cluster ├── Frontend Service (3 pods) ├── API Gateway Service (2 pods) ├── User Service (5 pods) ├── Order Service (5 pods) ├── Payment Service (3 pods) └── Database Services

```
---

## Where Does Networking Fit in Your Architecture?

### Understanding Layers in Your Application

Think of your application like a building with different floors:

**Floor 5: User Layer** (Top Floor - What users see)
- Web browsers, mobile apps
- **Networking Here:** DNS, Global load balancers, CDN
- **Why:** Get users to the right server quickly

**Floor 4: Presentation Layer** (Public facing)
- Web servers, API gateways
- **Networking Here:** Load balancers, Firewalls, Public IPs
- **Why:** Handle incoming traffic, protect against attacks

**Floor 3: Application Layer** (Business logic)
- Application servers, Microservices
- **Networking Here:** Internal load balancers, Service discovery
- **Why:** Services need to communicate efficiently

**Floor 2: Data Layer** (Storage)
- Databases, Cache, File storage
- **Networking Here:** Private subnets, Private endpoints
- **Why:** Most sensitive layer, needs maximum security

**Floor 1: Infrastructure Layer** (Foundation)
- Virtual networks, Routers, Firewalls
- **Networking Here:** Virtual networks, Subnets, Route tables
- **Why:** Foundation everything else builds on

### Where to Place Different Components

**In the Public Subnet** (Internet-facing):
- ✅ Load Balancers
- ✅ Web servers (if needed)
- ✅ Jump boxes (for admin access)
- ❌ Never: Databases, internal APIs, sensitive services

**In the Private Subnet** (Hidden from internet):
```

```
- ✅ Application servers
- ✅ Databases
- ✅ Internal APIs
- ✅ Cache servers
- ✅ Message queues

**In the Management Subnet** (Admin access):
- ✅ Monitoring tools
- ✅ CI/CD agents
- ✅ Admin tools
- Access only through VPN or Bastion

### Example: E-commerce Website Architecture

Let's design a simple online store:
```

INTERNET ↓ [FIREWALL + DDoS Protection] ↓ [Global Load Balancer] ↓ PUBLIC SUBNET (10.0.1.0/24) ├── Application Gateway (Web Application Firewall) └── Jump Box (for admin access) ↓ APPLICATION SUBNET (10.0.2.0/24) ├── Web Server 1 (Product pages) ├── Web Server 2 (Shopping cart) ├── API Server 1 (Inventory API) └── API Server 2 (Payment API) ↓ DATA SUBNET (10.0.3.0/24) ├── Database Server (Customer data, Orders) ├── Redis Cache (Session data) └── File Storage (Product images) ↓ [Backup to another region]

```
**Traffic Flow for "Buy a Product":**
1. Customer visits website → Global Load Balancer → Application Gateway
2. Application Gateway routes to Web Server based on URL
3. Web Server calls API Server to check inventory
4. API Server queries Database
5. Database responds to API Server
6. API Server responds to Web Server
7. Web Server sends page to Customer

**Security at Each Layer:**
- Internet → Firewall blocks bad traffic
- Public Subnet → Only ports 80/443 open
- Application Subnet → Only accessible from public subnet
- Data Subnet → Only accessible from application subnet

---

## Step-by-Step: Your First Network Setup

### Scenario: Deploy a Simple Web Application

**What we're building:**
A simple website with a web server and database in Azure.

**What you'll need:**
- Azure subscription (free tier works!)
- Basic understanding of the Azure portal
- 30 minutes of time

### Step 1: Plan Your Network
```

```
**Before touching Azure, answer these questions:**

1. **What IP range should I use?**
   - Use `10.0.0.0/16` for your virtual network
   - This gives you 65,536 IP addresses (more than enough!)

2. **How many subnets do I need?**
   - Web Subnet: `10.0.1.0/24` (256 addresses)
   - Database Subnet: `10.0.2.0/24` (256 addresses)

3. **What needs to access what?**
   - Internet → Web Server (port 443 HTTPS)
   - Web Server → Database (port 5432 PostgreSQL)
   - Your computer → Web Server (port 22 SSH for management)

4. **What should be blocked?**
   - Internet → Database (BLOCK EVERYTHING)
   - Random ports on Web Server (BLOCK)

**Write this down! This is your network design.**

### Step 2: Create the Virtual Network

**What is this:** A virtual network is like your own private network in Azure. Nothing can
get in or out unless you allow it.

**In Azure Portal:**

1. Search for "Virtual Networks"
2. Click "Create"
3. Fill in:
   - **Name:** `myapp-vnet`
   - **Region:** Choose one close to you
   - **IP Address Space:** `10.0.0.0/16`

4. Add Subnets:
   - **Subnet 1 Name:** `web-subnet`
   - **Subnet 1 Range:** `10.0.1.0/24`
   - **Subnet 2 Name:** `database-subnet`
   - **Subnet 2 Range:** `10.0.2.0/24`

5. Click "Review + Create"

**What just happened?**
You created a private network in Azure. Right now, it's empty, but it's ready to host your
servers.

### Step 3: Create Security Rules (Network Security Groups)

**What is this:** Think of NSGs as security guards that check every packet of data trying to
enter or leave.
```

```
**Create NSG for Web Subnet:**

1. Search for "Network Security Groups"
2. Click "Create"
3. Name it: `web-nsg`
4. Add Inbound Rules:

   **Rule 1: Allow HTTPS from Internet**
   - Source: `Any`
   - Destination: `Any`
   - Port: `443`
   - Action: `Allow`
   - Priority: `100`
   - Name: `Allow-HTTPS`

   **Rule 2: Allow SSH from your IP only**
   - Source: `Your IP address` (Azure can detect this)
   - Destination: `Any`
   - Port: `22`
   - Action: `Allow`
   - Priority: `110`
   - Name: `Allow-SSH-MyIP`

   **Rule 3: Deny everything else** (implicit - already exists)

5. Associate this NSG with `web-subnet`

**Create NSG for Database Subnet:**

1. Create another NSG: `database-nsg`
2. Add Inbound Rules:

   **Rule 1: Allow PostgreSQL from Web Subnet only**
   - Source: `10.0.1.0/24` (web subnet)
   - Destination: `Any`
   - Port: `5432`
   - Action: `Allow`
   - Priority: `100`
   - Name: `Allow-PostgreSQL-WebSubnet`

   **Rule 2: Deny everything else** (implicit)

3. Associate this NSG with `database-subnet`

**What just happened?**
You created security rules! Now:
- Only HTTPS traffic from internet can reach web servers
- Only you can SSH to web servers
- Only web servers can talk to the database
- Everything else is blocked

### Step 4: Deploy Your Servers
```

```
**Deploy Web Server:**

1. Create a Virtual Machine
2. Put it in `web-subnet`
3. Give it a public IP address (so users can reach it)
4. Install your web application

**Deploy Database Server:**

1. Create Azure Database for PostgreSQL
2. Put it in `database-subnet`
3. **No public IP!** (it's private)
4. Configure connection string for web server

### Step 5: Test Your Setup

**Test 1: Can you reach the website?**
```

Open browser → Go to your web server's IP ✅ Should work (port 443 is open)

```
**Test 2: Can the web server reach the database?**
```

SSH to web server → Try connecting to database ✅ Should work (web subnet can access database subnet)

```
**Test 3: Can the internet reach the database?**
```

Try connecting to database from your computer ❌ Should FAIL (this is good! Database is protected)

```
### Step 6: Make It Better

**Add a Domain Name:**
1. Register a domain (like `myawesomeapp.com`)
2. Point it to your web server's IP address
3. Now users can visit `myawesomeapp.com`

**Add SSL Certificate:**
1. Get a free certificate from Let's Encrypt
2. Install it on your web server
3. Now your site is secure (HTTPS)

**Add Monitoring:**
1. Enable Azure Monitor
2. Set up alerts for high CPU, network errors
3. You'll know when something goes wrong

**Congratulations!** You've set up your first network in Azure!

---

## Networking for DevOps Engineers
```

```
### Your Role in Networking

**What DevOps Engineers Do:**
- Design and implement network infrastructure
- Automate network configuration
- Ensure applications can communicate
- Troubleshoot connectivity issues
- Balance security with accessibility

**What You Don't Do:**
- Manage physical network hardware (that's for on-premises network engineers)
- Configure office WiFi (that's for IT support)
- Deal with ISP issues (usually handled by vendors)

### Phase 1: Planning (Before You Create Anything)

**Goal:** Understand what you're building before you build it.

**Questions to Ask:**

1. **About the Application:**
   - What services need to communicate?
   - How much traffic do we expect?
   - What's the acceptable downtime?
   - Where are our users located?

2. **About Security:**
   - What data is sensitive?
   - Who needs access to what?
   - What compliance rules apply (HIPAA, GDPR, etc.)?
   - What are the biggest security risks?

3. **About Scale:**
   - How many users now?
   - How many users in 1 year?
   - Do we need multiple regions?
   - Will we need to scale quickly?

**Document Your Answers:**
Create a simple document with:
- Network diagram (even a hand-drawn one!)
- IP address allocation plan
- Security requirements
- List of services and their communication needs

**Example Planning Document:**
```

Application: Online Store Expected Users: 10,000 now, 100,000 in 1 year Regions: Start with US, expand to Europe later

Components:

- Web Frontend (needs public access)

- API Backend (needs access from Frontend)
- Database (needs access from Backend only)
- Redis Cache (needs access from Backend only)

Security Requirements:

- PCI compliance for payment data
- Database must be private
- All external traffic must use HTTPS
- Audit logs for all access

IP Plan:

- VNet: 10.0.0.0/16
- Web Subnet: 10.0.1.0/24
- App Subnet: 10.0.2.0/24
- Data Subnet: 10.0.3.0/24

~~~
### Phase 2: Implementation (Building Your Network)

**Goal:** Turn your plan into actual infrastructure.

**Step 1: Create Virtual Network**

**Why Infrastructure as Code?**
- You can recreate your network if something breaks
- You can version control it (track changes)
- You can review changes before applying them
- You can use the same config in dev, staging, prod

**Example with Terraform:**
```hcl
# This code creates a virtual network
resource "azurerm_virtual_network" "main" {
  name                = "myapp-vnet"
  location            = "eastus"
  resource_group_name = "myapp-rg"
  address_space       = ["10.0.0.0/16"]
}

# This code creates a subnet for web servers
resource "azurerm_subnet" "web" {
  name                 = "web-subnet"
  resource_group_name  = "myapp-rg"
  virtual_network_name = azurerm_virtual_network.main.name
  address_prefixes     = ["10.0.1.0/24"]
}

# This code creates a subnet for databases
resource "azurerm_subnet" "database" {
~~~

```
  name                 = "database-subnet"
  resource_group_name  = "myapp-rg"
  virtual_network_name = azurerm_virtual_network.main.name
  address_prefixes     = ["10.0.2.0/24"]
}
```

What this does: Creates your network in Azure. Run `terraform apply` and it builds everything!

Step 2: Set Up Security

Create Firewall Rules:

```
# Network Security Group for web servers
resource "azurerm_network_security_group" "web" {
  name                = "web-nsg"
  location            = "eastus"
  resource_group_name = "myapp-rg"

  # Allow HTTPS from anywhere
  security_rule {
    name                       = "AllowHTTPS"
    priority                   = 100
    direction                  = "Inbound"
    access                     = "Allow"
    protocol                   = "Tcp"
    source_port_range          = "*"
    destination_port_range     = "443"
    source_address_prefix      = "*"
    destination_address_prefix = "*"
  }
}
```

What this does: Creates security rules. Only HTTPS traffic is allowed to web servers.

Step 3: Deploy Load Balancer

Why You Need This:

- Distributes traffic across multiple servers
- If one server dies, traffic goes to healthy servers
- Handles SSL certificates for you

Example:

```
# Create a load balancer
az network lb create \
  --resource-group myapp-rg \
  --name myapp-lb \
  --sku Standard \
  --frontend-ip-name frontend \
  --backend-pool-name backend-pool

# Configure health checks
```

```
az network lb probe create \
  --resource-group myapp-rg \
  --lb-name myapp-lb \
  --name health-check \
  --protocol http \
  --port 80 \
  --path /health
```

What this does: Creates a load balancer that checks if your servers are healthy and distributes traffic.

Phase 3: Testing (Make Sure It Works)

Goal: Verify everything works before users see it.

Test 1: Basic Connectivity

```
# Can web servers reach the database?
# SSH to web server, then:
ping 10.0.2.10  # Database IP
# Should work!

# Can the internet reach the database directly?
ping <database-public-ip>
# Should NOT work! (database should be private)
```

Test 2: Security Rules

```
# Try accessing web server on port 22 (SSH) from random IP
# Should be blocked!

# Try accessing web server on port 443 (HTTPS) from anywhere
# Should work!
```

Test 3: Load Balancer

```
# Send 100 requests to the load balancer
for i in {1..100}; do
  curl https://myapp-lb.azure.com
done

# Check if requests are distributed across servers
# All servers should have received some requests
```

Test 4: Failure Scenarios

```
# Stop one web server
# Load balancer should detect it's unhealthy
# Traffic should go to remaining servers
# Users should see no errors!
```

Phase 4: Monitoring (Know What's Happening)

Goal: See problems before users complain.

What to Monitor:

1. **Connectivity:**
   - Are all services reachable?
   - Is the load balancer healthy?
   - Are any firewall rules blocking legitimate traffic?
2. **Performance:**
   - How fast are network requests?
   - Is bandwidth saturated?
   - Are there any bottlenecks?
3. **Security:**
   - Are there unusual traffic patterns?
   - Are there repeated login failures?
   - Is someone scanning for vulnerabilities?

Setting Up Monitoring:

```
# Enable Network Watcher (Azure's network monitoring tool)
az network watcher configure \
  --resource-group myapp-rg \
  --locations eastus \
  --enabled true

# Enable flow logs (see all network traffic)
az network watcher flow-log create \
  --resource-group myapp-rg \
  --name web-flow-logs \
  --nsg web-nsg \
  --storage-account mylogsaccount \
  --enabled true
```

Set Up Alerts:

- Alert when web server is unreachable
- Alert when database connections fail
- Alert when unusual traffic spikes occur
- Alert when firewall blocks spike

Phase 5: Maintenance (Keep It Running)

Goal: Keep your network healthy and secure.

Weekly Tasks:

- Review security logs for suspicious activity
- Check monitoring dashboards for anomalies
- Verify backup systems are working

Monthly Tasks:

- Review and clean up unused resources (saves money!)
- Update firewall rules if needs changed
- Check for Azure networking feature updates
- Review costs and optimize

Quarterly Tasks:

- Full security audit
- Disaster recovery test (can you recover from failure?)
- Capacity planning (will you run out of IP addresses?)
- Update documentation

When Something Breaks:

1. Don't panic!
2. Check monitoring dashboards (what's the error?)
3. Check recent changes (did someone change firewall rules?)
4. Test connectivity systematically (where exactly is it failing?)
5. Fix the issue
6. Document what happened and how you fixed it

Networking for Azure DevOps Engineers

Your Special Role

What Makes Azure DevOps Different: You're not just setting up networks manually—you're automating everything through CI/CD pipelines!

Your Job:

- Automate network infrastructure deployment
- Integrate network configuration into CI/CD
- Ensure pipelines can deploy to network resources
- Make network changes through code, not clicking

Understanding Your Tools

What is CI/CD?

- **CI (Continuous Integration):** Automatically test code when developers commit
- **CD (Continuous Deployment):** Automatically deploy code to production

How Networking Fits: Your pipeline needs to:

1. Deploy network infrastructure (VNets, subnets, NSGs)
2. Deploy applications to those networks
3. Configure DNS, load balancers, etc.
4. Test that everything can communicate

Setting Up Pipeline Networking

Problem: Your pipeline needs to deploy to Azure, but how does it connect?

Solution: Service Connections

Step 1: Create Service Principal

```
# This creates an identity for your pipeline
az ad sp create-for-rbac \
  --name "azure-devops-pipeline" \
  --role contributor \
  --scopes /subscriptions/{subscription-id}

# Output gives you:
# - Application ID (Client ID)
# - Password (Client Secret)
# - Tenant ID
# Write these down securely!
```

Step 2: Configure Azure DevOps

1. Go to Azure DevOps → Project Settings
2. Click "Service Connections"
3. Click "New Service Connection"
4. Choose "Azure Resource Manager"
5. Enter the IDs from Step 1

What just happened? Your pipeline can now deploy to Azure! It has permissions to create resources.

Pipeline Example: Deploy Network Infrastructure

Scenario: Every time you push code, automatically deploy network changes.

azure-pipelines.yml:

```
# This pipeline deploys your network infrastructure
```

```
trigger:
  - main  # Run when code is pushed to main branch

pool:
  vmImage: 'ubuntu-latest'

steps:
# Step 1: Install Terraform
- task: TerraformInstaller@0
  inputs:
    terraformVersion: 'latest'

# Step 2: Initialize Terraform
- task: TerraformTaskV2@2
  inputs:
    command: 'init'
    workingDirectory: '$(System.DefaultWorkingDirectory)/terraform'
  displayName: 'Terraform Init'

# Step 3: Plan changes (preview what will change)
- task: TerraformTaskV2@2
  inputs:
    command: 'plan'
    workingDirectory: '$(System.DefaultWorkingDirectory)/terraform'
  displayName: 'Terraform Plan'

#
```

Step 4: Apply changes (only if approved)

- task: TerraformTaskV2@2 inputs: command: 'apply' workingDirectory: '$(System.DefaultWorkingDirectory)/terraform' commandOptions: '-auto-approve' displayName: 'Terraform Apply' condition: and(succeeded(), eq(variables['Build.SourceBranch'], 'refs/heads/main'))

Step 5: Test the network

- script: | echo "Testing network connectivity..."

  # Test if resources are reachable

  az network vnet show --name myapp-vnet --resource-group myapp-rg displayName: 'Verify Network Deployment'

```
**What this pipeline does:**
1. Installs Terraform
2. Shows what will change (preview)
3. Applies the changes to Azure
4. Verifies everything deployed correctly
```

~~~
**Best Practice:** Add a manual approval step before deploying to production!

### Handling Secrets Safely

**Problem:** Your database connection string contains a password. You can't put that in
code!

**Solution: Azure Key Vault**

**Step 1: Store Secret in Key Vault**
```bash
# Create Key Vault
az keyvault create \
  --name myapp-keyvault \
  --resource-group myapp-rg

# Store database password
az keyvault secret set \
  --vault-name myapp-keyvault \
  --name "DatabasePassword" \
  --value "super-secret-password"
~~~

Step 2: Use in Pipeline

```yaml
# In your pipeline, retrieve secrets from Key Vault
steps:
- task: AzureKeyVault@2
  inputs:
    azureSubscription: 'MyServiceConnection'
    KeyVaultName: 'myapp-keyvault'
    SecretsFilter: 'DatabasePassword'
  displayName: 'Get Secrets from Key Vault'

# Now you can use $(DatabasePassword) in your pipeline
- script: |
    echo "Configuring app with database password..."
    # The password is available as $(DatabasePassword)
  displayName: 'Configure Application'
```

What this does: Keeps secrets out of your code. Pipeline fetches them securely when needed.

Multi-Environment Deployments

Problem: You have Dev, Staging, and Production. Each needs different network config.

Solution: Environment Variables

Step 1: Define Environments

```
Dev:
  - VNet: 10.1.0.0/16
  - Small VMs
```

```
  - Minimal security

Staging:
  - VNet: 10.2.0.0/16
  - Medium VMs
  - Production-like security

Production:
  - VNet: 10.3.0.0/16
  - Large VMs
  - Maximum security
  - Multiple regions
```

Step 2: Pipeline with Stages

```yaml
stages:
# Deploy to Dev automatically
- stage: Dev
  variables:
    environment: 'dev'
    vnetPrefix: '10.1.0.0/16'
  jobs:
  - job: DeployDev
    steps:
    - script: echo "Deploying to Dev with VNet $(vnetPrefix)"

# Deploy to Staging with approval
- stage: Staging
  dependsOn: Dev
  variables:
    environment: 'staging'
    vnetPrefix: '10.2.0.0/16'
  jobs:
  - deployment: DeployStaging
    environment: 'staging'  # Requires approval in Azure DevOps
    strategy:
      runOnce:
        deploy:
          steps:
          - script: echo "Deploying to Staging with VNet $(vnetPrefix)"

# Deploy to Production with multiple approvals
- stage: Production
  dependsOn: Staging
  variables:
    environment: 'production'
    vnetPrefix: '10.3.0.0/16'
  jobs:
  - deployment: DeployProduction
    environment: 'production'  # Requires multiple approvals
    strategy:
      runOnce:
        deploy:
          steps:
          - script: echo "Deploying to Production with VNet $(vnetPrefix)"
```

What this does:

- Dev deploys automatically (for quick testing)
- Staging requires one approval
- Production requires multiple approvals (safety!)

Testing Network Changes in Pipelines

Problem: How do you know your network changes work before deploying?

Solution: Automated Tests

Step 1: Write Tests

```
#!/bin/bash
# test-network.sh

echo "Testing network configuration..."

# Test 1: Check if VNet exists
echo "Test 1: Checking VNet..."
az network vnet show --name myapp-vnet --resource-group myapp-rg
if [ $? -eq 0 ]; then
  echo "✅ VNet exists"
else
  echo "❌ VNet does not exist"
  exit 1
fi

# Test 2: Check if subnets exist
echo "Test 2: Checking subnets..."
SUBNETS=$(az network vnet subnet list --vnet-name myapp-vnet --resource-group myapp-rg --
query "[].name" -o tsv)
if [[ $SUBNETS == *"web-subnet"* ]]; then
  echo "✅ Web subnet exists"
else
  echo "❌ Web subnet missing"
  exit 1
fi

# Test 3: Check NSG rules
echo "Test 3: Checking NSG rules..."
HTTPS_RULE=$(az network nsg rule show --nsg-name web-nsg --resource-group myapp-rg --name
AllowHTTPS)
if [ $? -eq 0 ]; then
  echo "✅ HTTPS rule exists"
else
  echo "❌ HTTPS rule missing"
  exit 1
fi
```

```
echo "All tests passed! ✅"
```

Step 2: Add to Pipeline

```
- script: |
    chmod +x test-network.sh
    ./test-network.sh
  displayName: 'Run Network Tests'
```

What this does: Automatically verifies your network is configured correctly. If tests fail, deployment stops!

Networking for AKS Administrators

What is AKS?

Simple Explanation: AKS (Azure Kubernetes Service) is like a smart manager for your containers. Instead of running one app on one server, you run hundreds of small containers that AKS manages automatically.

Why Networking is Different:

- You have hundreds of containers, not just a few servers
- Containers come and go constantly (they're temporary)
- Containers need to talk to each other all the time
- You need to manage traffic to thousands of "mini-apps"

Choosing Your Network Model

The Big Decision: Azure CNI vs Kubenet

Think of it like choosing between two road systems for your city:

Option 1: Azure CNI (Integrated Network)

- Every container (pod) gets a real IP address from your Azure network
- Like giving every house in your city a real street address

Pros:

- ✅ Containers can talk directly to Azure services
- ✅ You can use Azure networking features
- ✅ Simpler to understand
- ✅ Better for production

Cons:

- ❌ Uses LOTS of IP addresses
- ❌ Need to plan IP space carefully
- ❌ More expensive

When to use: Production applications, when you need Azure integration, when you have plenty of IP addresses

Option 2: Kubenet (NAT Network)

- Containers get fake internal IP addresses
- Like using apartment numbers inside a building

Pros:

- ✅ Saves IP addresses
- ✅ Cheaper
- ✅ Good for small clusters

Cons:

- ❌ Containers hidden behind NAT (harder to access directly)
- ❌ Limited Azure integration
- ❌ More complex routing

When to use: Development environments, small clusters, IP address conservation is critical

Step-by-Step: Create Your First AKS Cluster

Scenario: Deploy a microservices application on Kubernetes

Step 1: Plan Your IP Addresses

Important Math: Calculate how many IPs you need!

```
Formula: (Number of Nodes × Max Pods per Node) + Node IPs + Buffer

Example:
- 5 nodes
- 30 pods per node maximum
- Calculation: (5 × 30) + 5 + 50 = 205 IPs needed
- Use /24 subnet (256 IPs) to be safe
```

Your Plan:

```
VNet: 10.0.0.0/16 (65,536 IPs total)
├── AKS Node Subnet: 10.0.1.0/24 (256 IPs)
├── AKS Pod Subnet: 10.0.2.0/23 (512 IPs)
├── App Gateway Subnet: 10.0.4.0/24 (256 IPs)
└── Database Subnet: 10.0.5.0/24 (256 IPs)
```

Step 2: Create the AKS Cluster

```
# Create resource group
az group create --name myaks-rg --location eastus

# Create virtual network
```

```
az network vnet create \
  --resource-group myaks-rg \
  --name aks-vnet \
  --address-prefix 10.0.0.0/16 \
  --subnet-name aks-subnet \
  --subnet-prefix 10.0.1.0/24

# Get subnet ID (you'll need this)
SUBNET_ID=$(az network vnet subnet show \
  --resource-group myaks-rg \
  --vnet-name aks-vnet \
  --name aks-subnet \
  --query id -o tsv)

# Create AKS cluster with Azure CNI
az aks create \
  --resource-group myaks-rg \
  --name myaks-cluster \
  --network-plugin azure \
  --vnet-subnet-id $SUBNET_ID \
  --docker-bridge-address 172.17.0.1/16 \
  --dns-service-ip 10.2.0.10 \
  --service-cidr 10.2.0.0/24 \
  --enable-managed-identity \
  --node-count 3 \
  --generate-ssh-keys
```

What each setting means:

- `--network-plugin azure`: Use Azure CNI (integrated networking)
- `--vnet-subnet-id`: Which subnet to put nodes in
- `--dns-service-ip`: IP for Kubernetes DNS service (must be in service-cidr)
- `--service-cidr`: IP range for Kubernetes services (cluster-internal IPs)
- `--node-count 3`: Start with 3 servers

Step 3: Connect to Your Cluster

```
# Get credentials to access cluster
az aks get-credentials \
  --resource-group myaks-rg \
  --name myaks-cluster

# Verify you can connect
kubectl get nodes

# You should see:
# NAME                                STATUS   ROLE    AGE   VERSION
# aks-nodepool1-12345678-vmss000000   Ready    agent   5m    v1.28.0
# aks-nodepool1-12345678-vmss000001   Ready    agent   5m    v1.28.0
# aks-nodepool1-12345678-vmss000002   Ready    agent   5m    v1.28.0
```

Congratulations! You have a Kubernetes cluster!

Deploying Your First Application

Scenario: Deploy a simple web app with a database

Step 1: Deploy a Database (PostgreSQL)

```yaml
# database.yaml
apiVersion: v1
kind: Service
metadata:
  name: postgres
spec:
  ports:
  - port: 5432
  selector:
    app: postgres
  clusterIP: None  # Headless service
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
spec:
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
      - name: postgres
        image: postgres:13
        ports:
        - containerPort: 5432
        env:
        - name: POSTGRES_PASSWORD
          value: "mypassword"  # In real life, use secrets!
```

Deploy it:

```bash
kubectl apply -f database.yaml

# Check if it's running
kubectl get pods
# NAME                          READY   STATUS    RESTARTS   AGE
# postgres-5f7b8c9d6f-xyz12     1/1     Running   0          30s
```

What just happened?

- Created a PostgreSQL database inside your cluster
- It got an internal IP address automatically
- Other apps in the cluster can reach it using name "postgres"

Step 2: Deploy Web Application

```yaml
# webapp.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
spec:
  replicas: 3  # Run 3 copies for reliability
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
      - name: webapp
        image: myregistry/webapp:latest
        ports:
        - containerPort: 8080
        env:
        - name: DATABASE_URL
          value: "postgres://postgres:5432"  # Using service name!
---
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
spec:
  type: LoadBalancer  # This gets a public IP!
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: webapp
```

Deploy it:

```
kubectl apply -f webapp.yaml

# Wait for public IP to be assigned
kubectl get service webapp-service --watch

# When EXTERNAL-IP shows an IP (not <pending>), you're ready!
# NAME              TYPE           EXTERNAL-IP      PORT(S)        AGE
# webapp-service    LoadBalancer   20.30.40.50      80:30123/TCP   2m
```

Test it:

```
# Visit the external IP in your browser
curl http://20.30.40.50

# You should see your web app! 🎉
```

What just happened?

- Deployed 3 copies of your web app
- Azure created a load balancer automatically
- Load balancer got a public IP
- Traffic is distributed across 3 app copies
- App can connect to database using "postgres" name

Setting Up Ingress (Better Than LoadBalancer)

Problem: LoadBalancer gives you one IP per service. If you have 10 services, that's 10 IPs and 10 load balancers. Expensive!

Solution: Ingress Controller - One load balancer for all your services!

Think of it like:

- LoadBalancer = Each apartment has its own mailbox on the street
- Ingress = One central mailbox for the building that sorts mail by apartment number

Step 1: Install NGINX Ingress Controller

```
# Add Helm repository (package manager for Kubernetes)
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
helm repo update

# Install NGINX Ingress
helm install ingress-nginx ingress-nginx/ingress-nginx \
  --namespace ingress-nginx \
  --create-namespace \
  --set controller.service.annotations."service\.beta\.kubernetes\.io/azure-load-balancer-
health-probe-request-path"=/healthz
```

Wait for it to get an IP:

```
kubectl get service -n ingress-nginx --watch

# When you see an EXTERNAL-IP, note it down!
# This is your ONE IP for ALL services
```

Step 2: Create Ingress Rules

```yaml
# ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: myapp-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - host: myapp.com
    http:
      paths:
      # Route /api to API service
      - path: /api
        pathType: Prefix
        backend:
          service:
            name: api-service
            port:
              number: 8080
      # Route / to frontend service
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend-service
            port:
              number: 80
```

Deploy it:

```
kubectl apply -f ingress.yaml
```

What this does:

- `myapp.com/api` → goes to api-service
- `myapp.com/` → goes to frontend-service
- One IP, multiple services!

Network Policies (Security Between Pods)

Problem: By default, any pod can talk to any other pod. That's not secure!

Solution: Network Policies - Firewall rules for pods

Example: Only frontend can talk to API

```yaml
# network-policy.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: api-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: api  # Apply to pods with label app=api
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend  # Only allow from frontend pods
    ports:
    - protocol: TCP
      port: 8080
```

Deploy it:

```bash
# First, enable network policies on your cluster
az aks update \
  --resource-group myaks-rg \
  --name myaks-cluster \
  --network-policy azure

# Then apply the policy
kubectl apply -f network-policy.yaml
```

What this does:

- ✅ Frontend pods CAN talk to API pods
- ❌ Database pods CANNOT talk to API pods
- ❌ Random pods CANNOT talk to API pods

Test it:

```
# From frontend pod - should work
kubectl exec -it frontend-pod -- curl http://api-service:8080

# From database pod - should fail
kubectl exec -it database-pod -- curl http://api-service:8080
# Error: connection refused (this is good!)
```

Connecting to Azure Services (Private Endpoints)

Problem: Your app needs Azure SQL Database, but you don't want it exposed to the internet.

Solution: Private Endpoint - Put Azure services INSIDE your network!

Step 1: Create Azure SQL with Private Endpoint

```
# Create SQL Server
az sql server create \
  --name myapp-sqlserver \
  --resource-group myaks-rg \
  --location eastus \
  --admin-user sqladmin \
  --admin-password "SuperSecret123!"

# Disable public access
az sql server update \
  --name myapp-sqlserver \
  --resource-group myaks-rg \
  --public-network-access Disabled

# Create private endpoint
az network private-endpoint create \
  --name sql-private-endpoint \
  --resource-group myaks-rg \
  --vnet-name aks-vnet \
  --subnet database-subnet \
  --private-connection-resource-id $(az sql server show --name myapp-sqlserver --resource-
group myaks-rg --query id -o tsv) \
  --group-id sqlServer \
  --connection-name sql-connection

# Create private DNS zone
az network private-dns zone create \
  --resource-group myaks-rg \
  --name privatelink.database.windows.net

# Link DNS zone to VNet
az network private-dns link vnet create \
  --resource-group myaks-rg \
  --zone-name privatelink.database.windows.net \
  --name sql-dns-link \
  --virtual-network aks-vnet \
  --registration-enabled false
```

Step 2: Connect from AKS

```
# Your app in AKS
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  template:
    spec:
      containers:
      - name: myapp
        image: myapp:latest
        env:
        - name: DB_HOST
          value: "myapp-sqlserver.privatelink.database.windows.net"
        - name: DB_PORT
          value: "1433"
```

What this does:

- SQL Server has NO public IP
- SQL Server is INSIDE your virtual network
- Only pods in your cluster can reach it
- Traffic never touches the internet!

Troubleshooting Common AKS Network Issues

Problem 1: "Pod can't reach the internet"

Symptoms:

```
kubectl exec -it my-pod -- curl https://google.com
# Hangs or times out
```

How to fix:

```
# Check 1: Is DNS working?
kubectl exec -it my-pod -- nslookup google.com
# If this fails, DNS is the problem

# Check 2: Can pod reach IP directly?
kubectl exec -it my-pod -- curl http://8.8.8.8
# If this works, it's definitely DNS

# Fix: Restart CoreDNS
kubectl rollout restart deployment/coredns -n kube-system

# Check 3: Are NSG rules blocking?
az network nsg rule list \
  --resource-group myaks-rg \
  --nsg-name aks-nsg \
  --query "[].{Name:name, Access:access, Direction:direction, Priority:priority}"

# If you see blocking rules, update them
```

Problem 2: "Can't access my service from outside"

Symptoms:

```
curl http://my-loadbalancer-ip
# Connection refused or timeout
```

How to fix:

```
# Check 1: Does service have external IP?
kubectl get service my-service
# EXTERNAL-IP should show an IP, not <pending>

# Check 2: Are pods running?
kubectl get pods -l app=my-service
# All should be Running, not Error or CrashLoopBackOff

# Check 3: Are pods healthy?
kubectl get endpoints my-service
# Should show pod IPs, not <none>

# Check 4: Test from inside cluster
kubectl run test-pod --image=busybox -it --rm -- wget -O- http://my-service
# If this works, problem is external access

# Check 5: Check NSG rules
az network nsg rule list --resource-group myaks-rg --nsg-name aks-nsg
# Look for rules blocking your service port
```

Problem 3: "Pods can't talk to each other"

Symptoms:

```
# From frontend pod
kubectl exec -it frontend-pod -- curl http://backend-service
# Connection refused
```

How to fix:

```
# Check 1: Is backend service running?
kubectl get service backend-service
kubectl get pods -l app=backend

# Check 2: Is there a network policy blocking it?
kubectl get networkpolicy
kubectl describe networkpolicy <policy-name>
```

```
# If policy is too restrictive, update it:
kubectl edit networkpolicy <policy-name>

# Check 3: Test DNS resolution
kubectl exec -it frontend-pod -- nslookup backend-service
# Should return cluster IP

# Check 4: Check endpoints
kubectl get endpoints backend-service
# Should show backend pod IPs
```

Common Problems and How to Fix Them

Problem 1: "I can't SSH to my server"

Symptoms:

```
ssh user@myserver.com
# Connection timed out
```

Possible Causes & Fixes:

Cause 1: NSG blocking SSH

```
# Check NSG rules
az network nsg rule list \
  --resource-group myapp-rg \
  --nsg-name my-nsg \
  --query "[?destinationPortRange=='22']"

# If no rule allows SSH, add one:
az network nsg rule create \
  --resource-group myapp-rg \
  --nsg-name my-nsg \
  --name AllowSSH \
  --priority 100 \
  --source-address-prefixes "YOUR_IP/32" \
  --destination-port-ranges 22 \
  --access Allow \
  --protocol Tcp
```

Cause 2: Server not in public subnet

```
# Check if server has public IP
az vm show \
  --resource-group myapp-rg \
  --name myserver \
  --query "publicIps"

# If no public IP and server is in private subnet, use jump box
ssh -J jumpbox@public-ip user@private-server-ip
```

Cause 3: SSH service not running

```
# Use Azure Serial Console or Run Command
az vm run-command invoke \
  --resource-group myapp-rg \
  --name myserver \
  --command-id RunShellScript \
  --scripts "systemctl status sshd"
```

Problem 2: "Website is slow"

Symptoms:

- Pages take 5+ seconds to load
- Database queries timeout
- Users complain

Diagnosis Steps:

Step 1: Check where the slowness is

```
# Test from outside
curl -w "Time: %{time_total}s\n" -o /dev/null -s https://myapp.com

# Test from inside network
az vm run-command invoke \
  --resource-group myapp-rg \
  --name web-server \
  --command-id RunShellScript \
  --scripts "curl -w 'Time: %{time_total}s\n' -o /dev/null -s http://database-server"
```

Step 2: Check network metrics

```
# Check bandwidth usage
az monitor metrics list \
  --resource
/subscriptions/{sub}/resourceGroups/{rg}/providers/Microsoft.Network/loadBalancers/{lb} \
  --metric ByteCount \
  --start-time 2024-01-01T00:00:00Z \
  --end-time 2024-01-01T23:59:59Z
```

Common Fixes:

Fix 1: Add load balancer (if you don't have one)

- Distributes traffic across multiple servers
- Prevents one server from being overwhelmed

Fix 2: Enable Accelerated Networking

```
az network nic update \
  --resource-group myapp-rg \
  --name my-nic \
  --accelerated-networking true
```

Fix 3: Move resources closer together

- Put web servers and database in same region
- Put web servers and database in same availability zone

Fix 4: Add caching

- Use Azure Redis Cache for frequently accessed data
- Use CDN for static files (images, CSS, JS)

Problem 3: "I'm getting 'Name or service not known' errors"

Symptoms:

```
ping myservice.com
# ping: myservice.com: Name or service not known
```

Possible Causes & Fixes:

Cause 1: DNS not configured

```
# Check DNS settings
cat /etc/resolv.conf

# Should show DNS servers like:
# nameserver 168.63.129.16 (Azure default DNS)

# If missing, configure DNS in VNet
az network vnet update \
  --resource-group myapp-rg \
  --name my-vnet \
  --dns-servers 168.63.129.16
```

Cause 2: DNS record doesn't exist

```
# Check if DNS record exists
nslookup myservice.com

# If not found, add DNS record
az network dns record-set a add-record \
  --resource-group myapp-rg \
  --zone-name myservice.com \
  --record-set-name www \
  --ipv4-address 20.30.40.50
```

Cause 3: Private DNS zone not linked to VNet

```
# Check private DNS zone links
az network private-dns link vnet list \
  --resource-group myapp-rg \
  --zone-name privatelink.database.windows.net

# If missing, create link
az network private-dns link vnet create \
  --resource-group myapp-rg \
  --zone-name privatelink.database.windows.net \
  --name my-link \
  --virtual-network my-vnet \
  --registration-enabled false
```

Problem 4: "SSL certificate errors"

Symptoms:

```
Your connection is not private
NET::ERR_CERT_AUTHORITY_INVALID
```

Possible Causes & Fixes:

Cause 1: Certificate expired

```
# Check certificate expiration
echo | openssl s_client -servername myapp.com -connect myapp.com:443 2>/dev/null | openssl
x509 -noout -dates

# Renew certificate (using Let's Encrypt)
certbot renew
```

Cause 2: Certificate doesn't match domain

```
# Check certificate domains
echo | openssl s_client -servername myapp.com -connect myapp.com:443 2>/dev/null | openssl
x509 -noout -text | grep DNS

# Make sure certificate includes your domain
# If not, generate new certificate with correct domain
```

Cause 3: Certificate not installed correctly

```
# Check certificate chain
openssl s_client -connect myapp.com:443 -showcerts

# Should show:
# - Server certificate
# - Intermediate certificate(s)
# - Root certificate

# If missing intermediate, install them
```

Problem 5: "Out of IP addresses!"

Symptoms:

```
# Can't create new resources
Error: No more IP addresses available in subnet
```

How to fix:

Option 1: Add more IP addresses to subnet (if possible)

```
# Check current subnet size
az network vnet subnet show \
  --resource-group myapp-rg \
  --vnet-name my-vnet \
  --name my-subnet \
  --query addressPrefix

# Expand subnet (only if adjacent IPs are free)
az network vnet subnet update \
  --resource-group myapp-rg \
  --vnet-name my-vnet \
  --name my-subnet \
  --address-prefixes "10.0.1.0/23"  # Doubles available IPs
```

Option 2: Create new subnet

```
# Create additional subnet
az network vnet subnet create \
  --resource-group myapp-rg \
  --vnet-name my-vnet \
  --name my-subnet-2 \
  --address-prefixes "10.0.2.0/24"

# Deploy new resources to new subnet
```

Option 3: Clean up unused IPs

```
# List all IP addresses
az network nic list \
  --resource-group myapp-rg \
  --query "[].{Name:name, IP:ipConfigurations[0].privateIPAddress, InUse:virtualMachine}"

# Delete IPs attached to deleted VMs
az network nic delete --resource-group myapp-rg --name unused-nic
```

Prevention: Always plan for growth! Use subnets larger than you think you need.

Quick Reference Guide

Essential Commands Cheat Sheet

Azure Networking:

```
# List all virtual networks
az network vnet list --output table

# Show VNet details
az network vnet show --resource-group myapp-rg --name my-vnet

# List subnets in a VNet
az network vnet subnet list --resource-group myapp-rg --vnet-name my-vnet --output table

# Check effective security rules
az network nic list-effective-nsg --resource-group myapp-rg --name my-nic

# Test connectivity between resources
az network watcher test-connectivity \
  --resource-group myapp-rg \
  --source-resource web-vm \
  --dest-resource database-vm \
  --dest-port 5432
```

Kubernetes Networking:

```
# List all services
kubectl get services --all-namespaces
```

```
# Show service details
kubectl describe service my-service

# List network policies
kubectl get networkpolicy --all-namespaces

# Test DNS from pod
kubectl run test --image=busybox -it --rm -- nslookup my-service

# Check pod connectivity
kubectl run test --image=nicolaka/netshoot -it --rm -- curl http://my-service

# View pod IPs
kubectl get pods -o wide

# Check service endpoints
kubectl get endpoints my-service

# View ingress details
kubectl describe ingress my-ingress
```

Troubleshooting Commands:

```
# Ping test
ping -c 4 myserver.com

# DNS lookup
nslookup myserver.com
dig myserver.com

# Test specific port
telnet myserver.com 443
nc -zv myserver.com 443

# Trace route to destination
traceroute myserver.com

# Check listening ports
netstat -tuln
ss -tuln

# Test HTTP endpoint
curl -v https://myserver.com
wget --spider https://myserver.com
```

Common Port Numbers

Service	Port	Protocol	Notes
HTTP	80	TCP	Unencrypted web traffic

Service	Port	Protocol	Notes
HTTPS	443	TCP	Encrypted web traffic
SSH	22	TCP	Secure shell access
RDP	3389	TCP	Windows remote desktop
FTP	21	TCP	File transfer
DNS	53	TCP/UDP	Domain name resolution
SMTP	25, 587	TCP	Email sending
PostgreSQL	5432	TCP	Database
MySQL	3306	TCP	Database
MongoDB	27017	TCP	Database
Redis	6379	TCP	Cache
Kubernetes API	6443	TCP	Cluster management
Kubernetes NodePort	30000-32767	TCP	Service exposure

IP Address Ranges (Private)

Range	CIDR	Number of IPs	Best for
10.0.0.0 - 10.255.255.255	10.0.0.0/8	16,777,216	Large enterprises
172.16.0.0 - 172.31.255.255	172.16.0.0/12	1,048,576	Medium organizations
192.168.0.0 - 192.168.255.255	192.168.0.0/16	65,536	Small networks, home

Subnet Size Calculator

CIDR	Subnet Mask	Available IPs	Use Case
/24	255.255.255.0	254	Small subnet (single app)
/23	255.255.254.0	510	Medium subnet
/22	255.255.252.0	1,022	Large subnet
/21	255.255.248.0	2,046	Very large subnet
/20	255.255.240.0	4,094	AKS clusters
/16	255.255.0.0	65,534	Entire VNet

Note: Azure reserves 5 IPs in each subnet, so actual usable IPs = shown - 5

Azure Service Tags (for NSG Rules)

Service Tag	What it represents
Internet	All public internet IPs
VirtualNetwork	All IPs in your VNet and peered VNets
AzureLoadBalancer	Azure Load Balancer health probes
AzureCloud	All Azure datacenter IPs
Storage	Azure Storage service IPs
Sql	Azure SQL service IPs
AzureKeyVault	Azure Key Vault service IPs
AzureActiveDirectory	Azure AD service IPs

Example Usage:

```
# Allow traffic from Azure Load Balancer health probes
az network nsg rule create \
  --resource-group myapp-rg \
  --nsg-name my-nsg \
  --name AllowAzureLB \
  --priority 100 \
  --source-address-prefixes AzureLoadBalancer \
  --destination-port-ranges "*" \
  --access Allow
```

Private Endpoint DNS Zones

When using private endpoints, you need these DNS zones:

Azure Service	Private DNS Zone
Azure SQL Database	privatelink.database.windows.net
Azure Storage (Blob)	privatelink.blob.core.windows.net
Azure Storage (File)	privatelink.file.core.windows.net
Azure Key Vault	privatelink.vaultcore.azure.net
Azure Container Registry	privatelink.azurecr.io
Azure Cosmos DB	privatelink.documents.azure.com
Azure App Service	privatelink.azurewebsites.net

Best Practices Summary

Security Best Practices

1. Always Use Private Subnets for Sensitive Data

```
✅ DO: Put databases in private subnets
❌ DON'T: Give databases public IPs
```

2. Follow the Principle of Least Privilege

```
✅ DO: Only open ports you need
✅ DO: Only allow specific source IPs
❌ DON'T: Open all ports to 0.0.0.0/0
❌ DON'T: Use "Allow All" rules
```

3. Use Network Segmentation

```
✅ DO: Separate tiers (web, app, data) into different subnets
✅ DO: Use network policies in Kubernetes
❌ DON'T: Put everything in one flat network
```

4. Always Use HTTPS

```
✅ DO: Use SSL/TLS certificates
✅ DO: Redirect HTTP to HTTPS
✅ DO: Use Let's Encrypt for free certificates
❌ DON'T: Transmit sensitive data over HTTP
```

5. Implement Defense in Depth

```
Multiple layers of security:
- Perimeter: DDoS protection, WAF
- Network: NSGs, firewalls, network policies
- Application: Authentication, authorization
- Data: Encryption at rest and in transit
```

Performance Best Practices

1. Put Resources Close Together

```
✅ DO: Use same region for resources that talk frequently
✅ DO: Use same availability zone when possible
✅ DO: Use proximity placement groups for ultra-low latency
```

2. Use Load Balancing

```
✅ DO: Distribute traffic across multiple instances
✅ DO: Configure proper health checks
✅ DO: Use autoscaling to handle traffic spikes
```

3. Enable Accelerated Networking

```
# For VMs that need high performance
az network nic update \
  --resource-group myapp-rg \
  --name my-nic \
  --accelerated-networking true
```

4. Use CDN for Static Content

```
✅ DO: Cache images, CSS, JavaScript on CDN
✅ DO: Use CDN for users far from your servers
❌ DON'T: Serve all content from origin server
```

5. Optimize DNS

```
✅ DO: Use Azure DNS for low latency
✅ DO: Cache DNS records appropriately
✅ DO: Use short TTLs only when necessary
```

Cost Optimization Best Practices

1. Clean Up Unused Resources

```
# Find unused public IPs
az network public-ip list \
  --query "[?ipConfiguration==null].{Name:name, ResourceGroup:resourceGroup}"

# Delete unused IPs (they cost money!)
az network public-ip delete --name unused-ip --resource-group myapp-rg
```

2. Right-Size Your Resources

```
✅ DO: Use Standard Load Balancer only when needed
✅ DO: Start with Basic tier and upgrade if needed
❌ DON'T: Over-provision "just in case"
```

3. Use Reserved Capacity

```
✅ DO: Reserve VPN Gateway, ExpressRoute for 1-3 years
✅ DO: Save up to 70% with reservations
```

4. Monitor Bandwidth Usage

```
✅ DO: Keep traffic within same region (free)
✅ DO: Use VNet peering instead of VPN (faster, cheaper)
❌ DON'T: Send unnecessary data between regions
```

Operational Best Practices

1. Use Infrastructure as Code

```
✅ DO: Define everything in Terraform/Bicep
✅ DO: Version control your configurations
✅ DO: Review changes before applying
❌ DON'T: Make manual changes in production
```

2. Document Everything

```
Must-have documentation:
- Network diagram
- IP allocation plan
- NSG rules with justification
- DNS records
- Troubleshooting runbooks
```

3. Implement Monitoring and Alerting

```
Monitor:
- Connectivity to critical services
- Bandwidth utilization
- Failed connection attempts
- NSG rule hits

Alert on:
- Service unreachable
- Unusual traffic patterns
- Security rule violations
- Approaching IP exhaustion
```

4. Test Disaster Recovery

```
✅ DO: Test failover procedures quarterly
✅ DO: Document recovery steps
✅ DO: Measure actual recovery time
❌ DON'T: Assume your DR plan works without testing
```

5. Keep Learning

```
✅ DO: Stay updated on Azure networking features
✅ DO: Read postmortems of networking incidents
✅ DO: Practice in lab environments
✅ DO: Learn from others' mistakes
```

Learning Path and Next Steps

For Complete Beginners

Week 1-2: Learn the Basics

- ✅ Understand IP addresses, subnets, ports
- ✅ Learn what DNS does
- ✅ Understand basic firewall concepts
- ✅ Create your first virtual network in Azure
- ✅ Deploy a simple VM and access it

Week 3-4: Build Something Simple

- ✅ Deploy a web server and database
- ✅ Configure NSG rules
- ✅ Set up a load balancer
- ✅ Add a custom domain
- ✅ Install SSL certificate

Month 2: Learn More Advanced Concepts

- ✅ Hub-and-spoke architecture
- ✅ VNet peering
- ✅ Private endpoints
- ✅ Application Gateway
- ✅ Network monitoring

Month 3: Start with Kubernetes

- ✅ Create your first AKS cluster
- ✅ Deploy applications
- ✅ Set up ingress
- ✅ Implement network policies
- ✅ Connect to Azure services

For DevOps Engineers

Focus Areas:

1. **Infrastructure as Code** (Master Terraform or Bicep)
2. **CI/CD Integration** (Automate network deployments)
3. **Security** (Zero trust, least privilege)
4. **Monitoring** (Azure Monitor, Network Watcher)
5. **Troubleshooting** (Systematic problem-solving)

Practice Projects:

- Deploy a multi-tier application with proper network segmentation
- Implement hub-and-spoke architecture
- Set up hybrid connectivity (VPN or ExpressRoute)
- Build a CI/CD pipeline that deploys network infrastructure
- Implement disaster recovery with multi-region setup

For AKS Administrators

Focus Areas:

1. **Container Networking** (CNI, network plugins)
2. **Service Mesh** (Istio or Linkerd)
3. **Ingress Controllers** (NGINX, Application Gateway)
4. **Network Policies** (Pod-to-pod security)
5. **Observability** (Distributed tracing, metrics)

Practice Projects:

- Deploy a microservices application
- Implement network policies for zero trust
- Set up Application Gateway Ingress Controller
- Configure service mesh with mTLS
- Implement blue-green deployments

Recommended Resources

Official Documentation:

- Microsoft Learn: Azure Networking paths
- Azure Documentation: Networking section
- Kubernetes Documentation: Networking section

Hands-On Practice:

- Azure Free Tier: Get $200 credit to practice
- Kubernetes Playground: Free K8s clusters for learning
- Terraform Documentation: Infrastructure as Code tutorials

Communities:

- Stack Overflow: Azure and Kubernetes tags
- Reddit: r/AZURE, r/kubernetes, r/devops
- Azure Tech Community: Forums and blogs
- CNCF Slack: Cloud Native community

Certifications:

- AZ-104: Azure Administrator Associate
- AZ-305: Azure Solutions Architect Expert
- CKA: Certified Kubernetes Administrator
- CKAD: Certified Kubernetes Application Developer

Final Thoughts

Remember These Key Points

1. Networking is a Journey, Not a Destination

- You don't need to know everything at once
- Start simple, add complexity as needed
- Learn by doing, not just reading
- Make mistakes in dev/test, not production

2. Security First, Always

- Default to deny, explicitly allow what's needed
- Private by default, public only when necessary
- Multiple layers of security are better than one
- Regular audits catch issues before they become problems

3. Document As You Go

- Your future self will thank you
- Your teammates will thank you
- Your successor will thank you
- Good documentation saves hours of troubleshooting

4. Automate Everything

- Manual changes lead to errors
- Infrastructure as Code ensures consistency
- Automation enables quick recovery
- Version control gives you a safety net

5. Monitor and Alert Proactively

- Know about problems before users do
- Trends help you plan capacity
- Logs are invaluable for troubleshooting
- What gets measured gets improved

Common Mistakes to Avoid

❌ **Don't:**

- Give databases public IPs
- Use the same network design for dev and prod
- Make changes without testing first
- Ignore security warnings
- Over-complicate simple scenarios
- Forget to document your changes
- Assume everything will work perfectly
- Skip monitoring and alerting

✅ **Do:**

- Start with secure defaults
- Test in dev/staging first
- Use Infrastructure as Code
- Implement proper monitoring
- Keep it as simple as possible
- Document your architecture
- Plan for failure
- Learn from incidents

Your Action Plan

This Week:

1. Create a simple virtual network in Azure
2. Deploy a VM and configure NSG rules
3. Practice SSH access and basic troubleshooting

This Month:

1. Deploy a multi-tier application
2. Implement load balancing
3. Set up basic monitoring
4. Write your infrastructure as code

This Quarter:

1. Deploy an AKS cluster (if relevant to your role)
2. Implement a complete CI/CD pipeline
3. Practice disaster recovery procedures
4. Optimize costs and performance

This Year:

1. Master Infrastructure as Code
2. Implement enterprise-grade security
3. Build multi-region architectures
4. Consider getting certified

Glossary

AKS (Azure Kubernetes Service): Microsoft's managed Kubernetes service that simplifies deploying and managing containerized applications.

Azure CNI: Container Network Interface that gives pods IP addresses directly from the virtual network.

CIDR (Classless Inter-Domain Routing): A method for allocating IP addresses and routing. Example: 10.0.0.0/24.

DNS (Domain Name System): Translates domain names (like google.com) to IP addresses.

Egress: Outbound network traffic leaving your network.

Ingress: Inbound network traffic coming into your network.

IP Address: A unique numerical label assigned to each device on a network.

Kubernetes: An open-source platform for automating deployment, scaling, and management of containerized applications.

Load Balancer: Distributes incoming network traffic across multiple servers.

NAT (Network Address Translation): Translates private IP addresses to public IP addresses.

Network Policy: Kubernetes rules that control traffic between pods.

NSG (Network Security Group): Azure firewall that contains security rules to allow or deny network traffic.

Pod: The smallest deployable unit in Kubernetes, containing one or more containers.

Port: A logical endpoint for network communication. Each service listens on a specific port.

Private Endpoint: Brings Azure services into your virtual network with a private IP address.

Route Table: Controls how network traffic is directed between subnets and networks.

Service: In Kubernetes, an abstraction that defines a logical set of pods and how to access them.

Subnet: A logical subdivision of a larger network.

VNet (Virtual Network): An isolated network in Azure where you deploy your resources.

VPN (Virtual Private Network): Creates a secure connection over the internet between networks.

Congratulations!

You've completed this comprehensive networking guide! You now understand:

✅ What networking is and why it's important ✅ How networking works from basics to advanced ✅ When to use different networking solutions ✅ Where networking fits in your architecture ✅ How to plan, implement, and maintain networks ✅ How to troubleshoot common problems ✅ Best practices for security, performance, and operations

Remember: Networking might seem complex, but by breaking it down into manageable pieces and practicing regularly, you'll become confident in designing and managing cloud networks.

Keep learning, keep practicing, and don't be afraid to make mistakes in your test environments!

Good luck on your networking journey! 🚀 Complete Beginner's Guide to Networking for DevOps