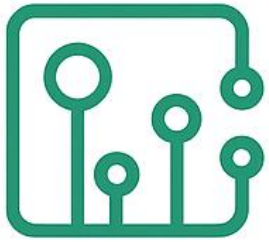
KUBERNETES SERVICES

Complete Guide to Load Balancing & Networking

- ✅ ClusterIP, NodePort & LoadBalancer
- ✅ Best Practices & Real Examples
- ✅ Troubleshooting & Hands-on Labs

Salwan Mohamed
DevOps & Platform Engineer



What Are Kubernetes Services?

Services provide STABLE access to pods:

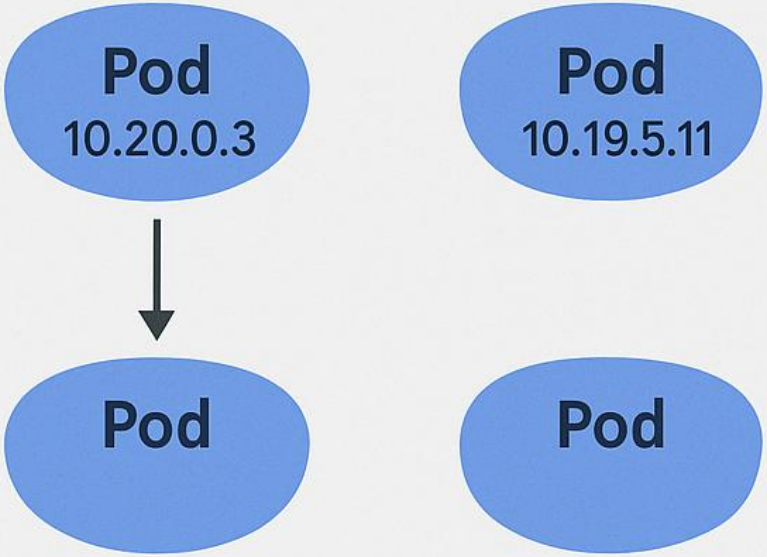
- Abstract away changing pod IPs
- Load balance traffic across pods
- Enable service discovery
- Act as internal load balancers

Think of them as traffic directors! 🎯

WHY DO WE NEED SERVICES?

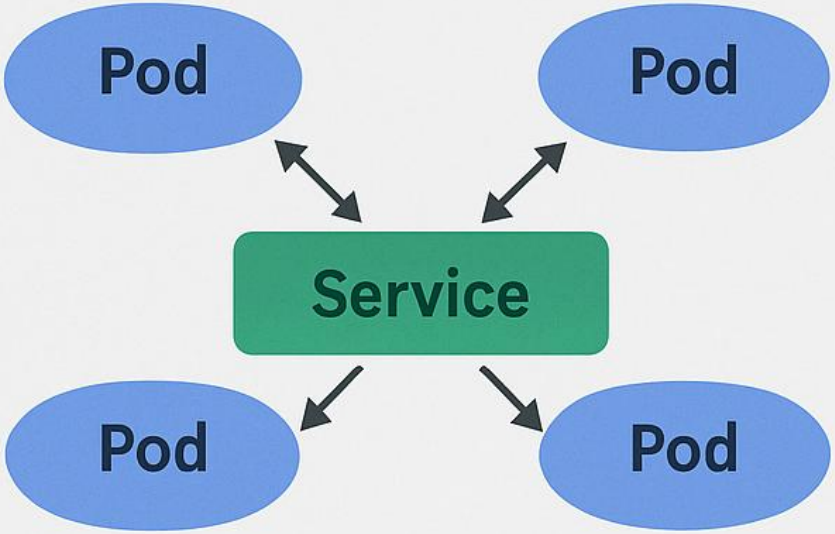
Without Services:

- Pods get random IP addresses
- IPs change when pods restart
- Direct pod access is unreliable
- No load balancing



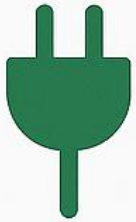
With Services

- Stable endpoint for pods
- Automatic load balancing
- Service discovery built-in
- High availability guaranteed



4 KUBERNETES SERVICE TYPES

ClusterIP
Internal only



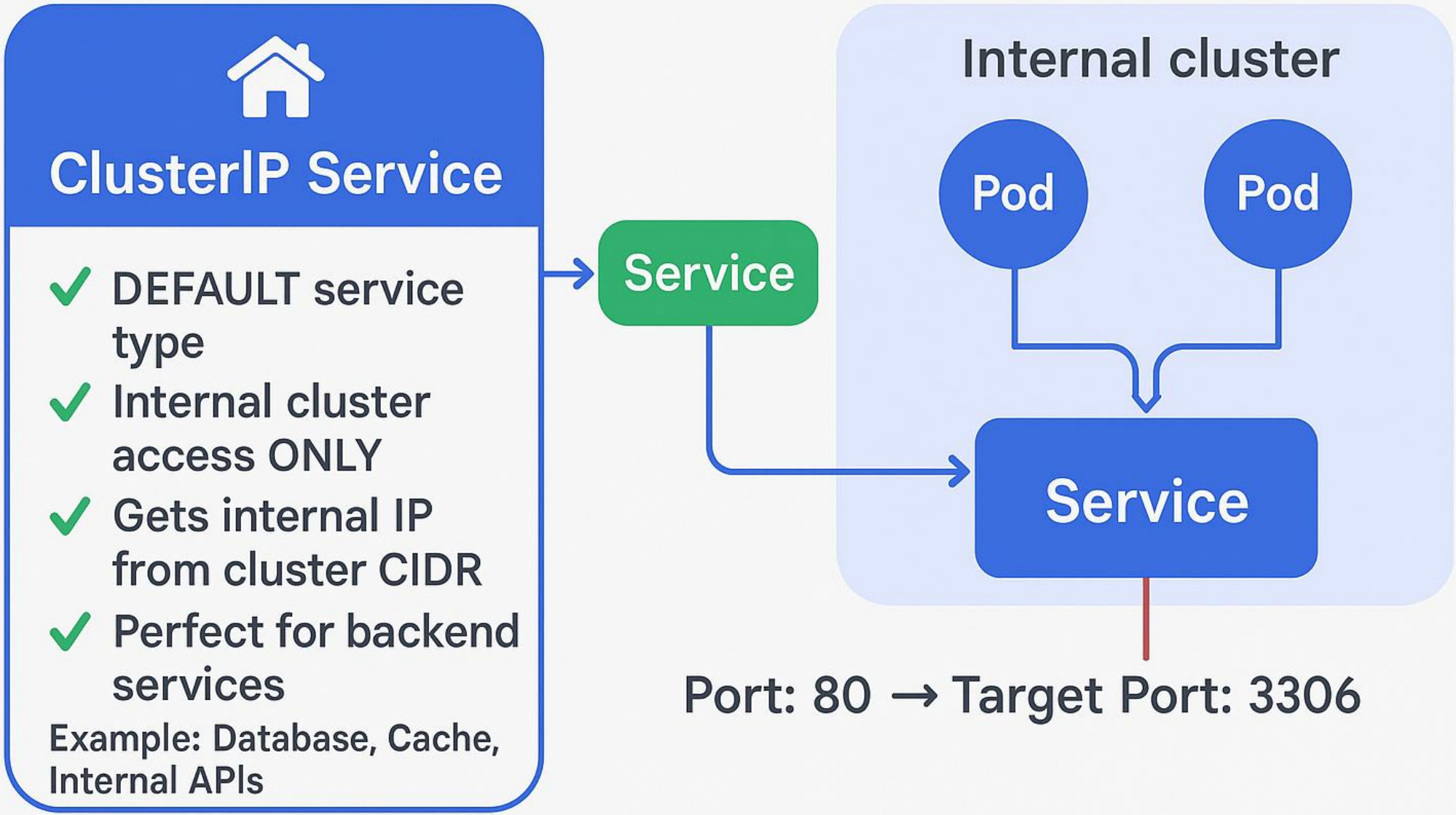
NodePort
External via node ports

LoadBalancer
Cloud load balancer

ExternalName
DNS CNAME mapping

Each serves different use cases!

ClusterIP Deep Dive



ClusterIP Best Practices

- Use for sensitive services (databases)
- Name services clearly (user-db-svc)
- Match selectors with pod labels
- Keep target ports consistent
- Monitor with service endpoints
- Never expose databases externally

NodePort Deep Dive

NodePort Service

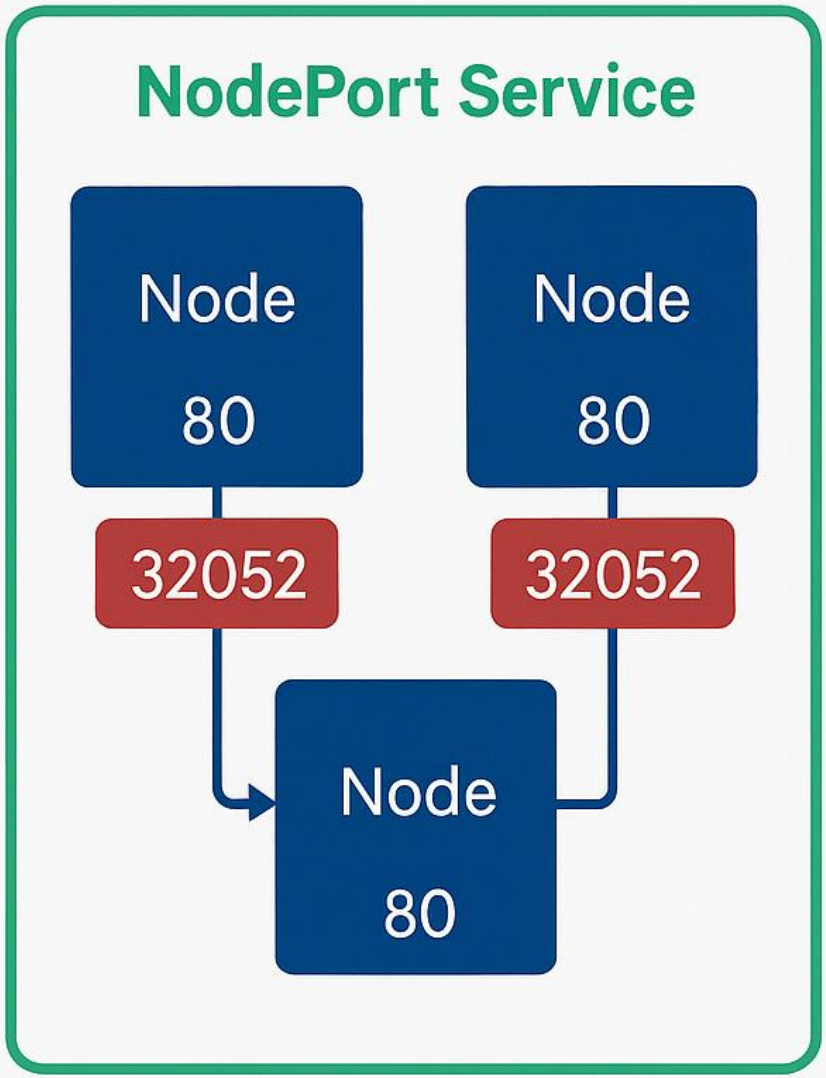
- ✔ Exposes service on ALL nodes
- ✔ Port range: 30000–32767
- ✔ External access without load balancer
- ✔ Builds on ClusterIP

Use Case:

Development, Testing, Legacy apps

External Port:

32052 → Internal Port: 80



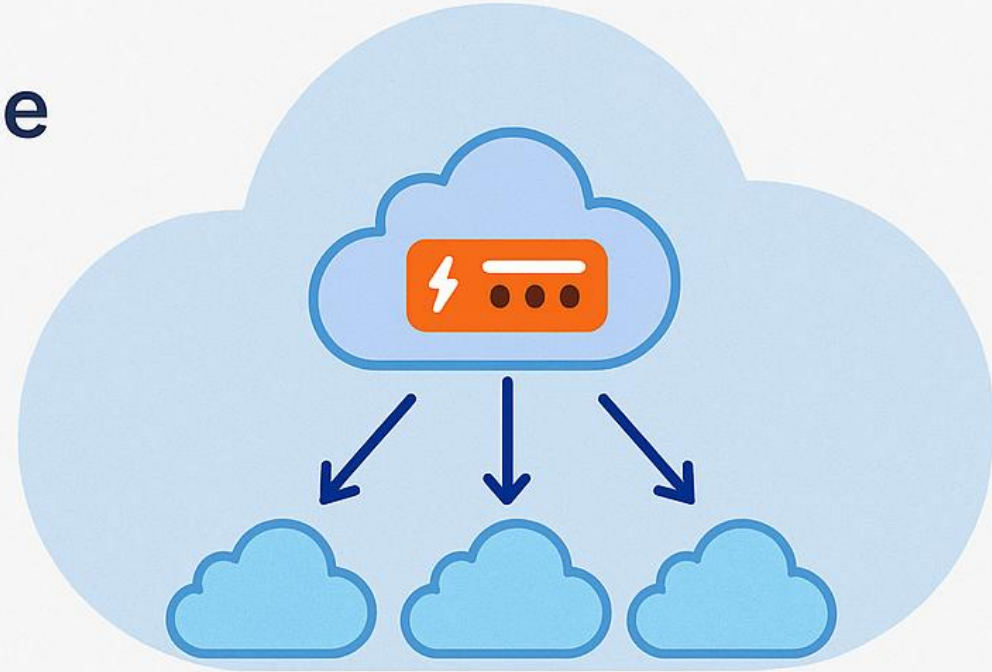
NodePort Best Practices

- Specify nodePort to avoid random ports
- Use only for dev/test environments
- NOT recommended for production
- Document port assignments
- Consider using Ingress instead
- Keep port ranges organized

LoadBalancer Deep Dive

LoadBalancer Service

- ✔ Creates external cloud load balancer
- ✔ Gets public IP address
- ✔ Automatic cloud integration
- ✔ Production-ready solution



aws

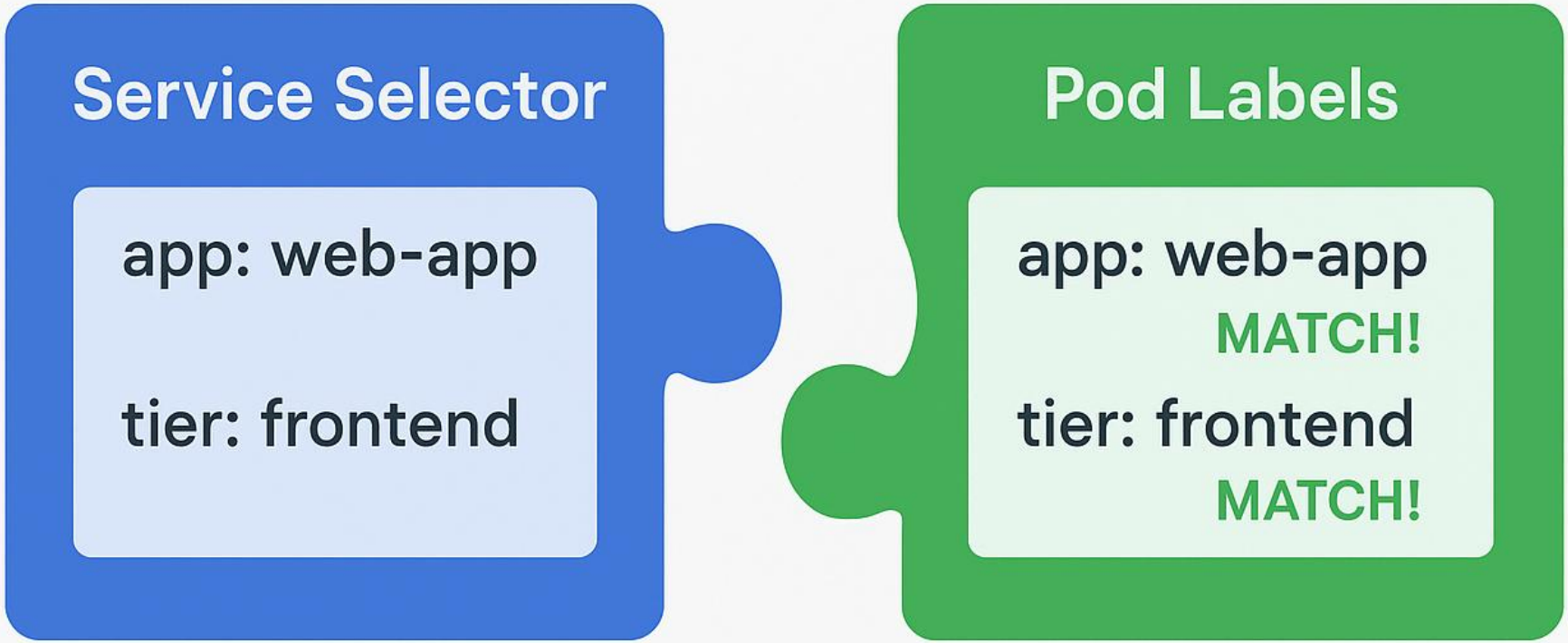
Cost: Each service = New load balancer! 💰

LoadBalancer Best Practices

- EXPENSIVE - Each service = New LB
- Use Ingress for multiple services
- Reserve for critical external services
- Monitor costs regularly
- Configure proper health checks
- Use for high-availability needs

Service Selectors & Labels

Selectors & Labels: The Magic Connection



No match = No traffic routing!

3 Ways to Create Services

Imperative Commands

```
kubectl
create
service
clusterip
```

2

Expose Existing Resources

```
kubectl epose
deployment
```

3

Declarative Manifests

```
kubectl apply
-f service.yaml
```

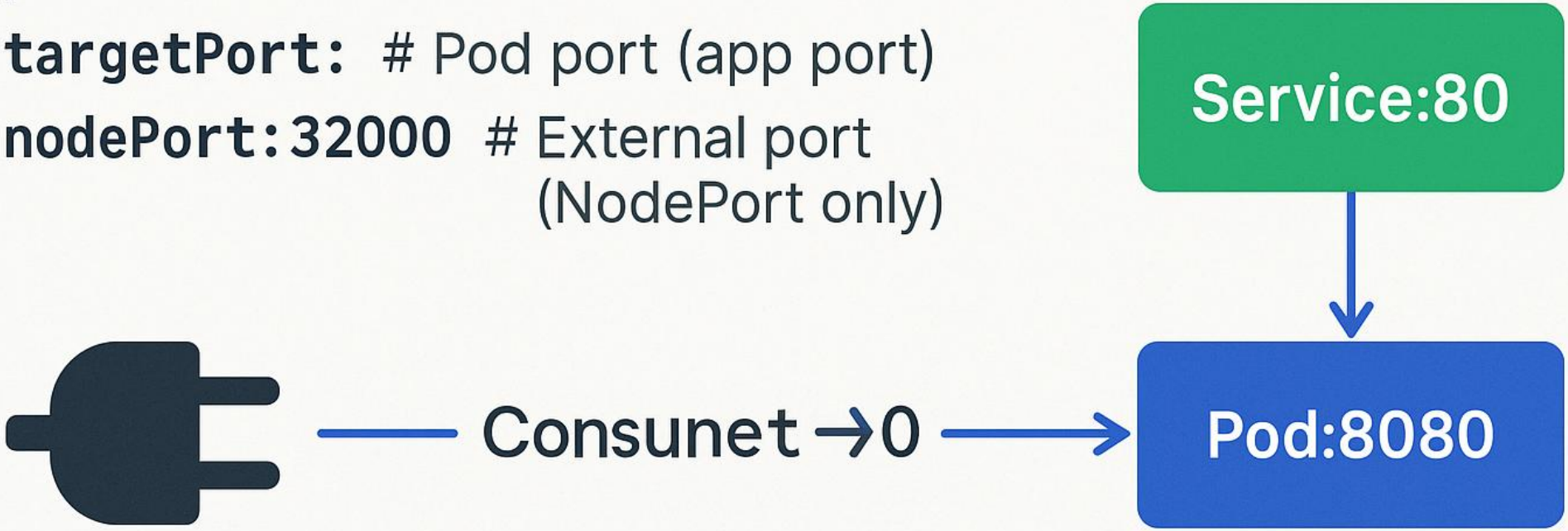
Basic Service Manifest

```
apiVersion: v1
kind: Service
metadata: name web-service
spec:
   selector: app: web-app
   ports
      port: 80
      targetPort: 8080
   type: ClusterIP
```

Service Port Configuration

Understanding Service Ports

```
port: 80          # Service listens here
targetPort:       # Pod port (app port)
nodePort:32000    # External port
                    (NodePort only)
```



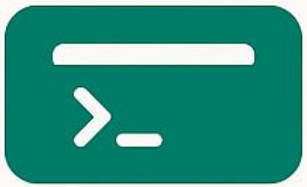
Think: Frontend ⟶ Backend mapping

Service Discovery in Kubernetes

DNS (DNS (Recommended)

```
http://web-service.default.svc.cluster.local
```

2



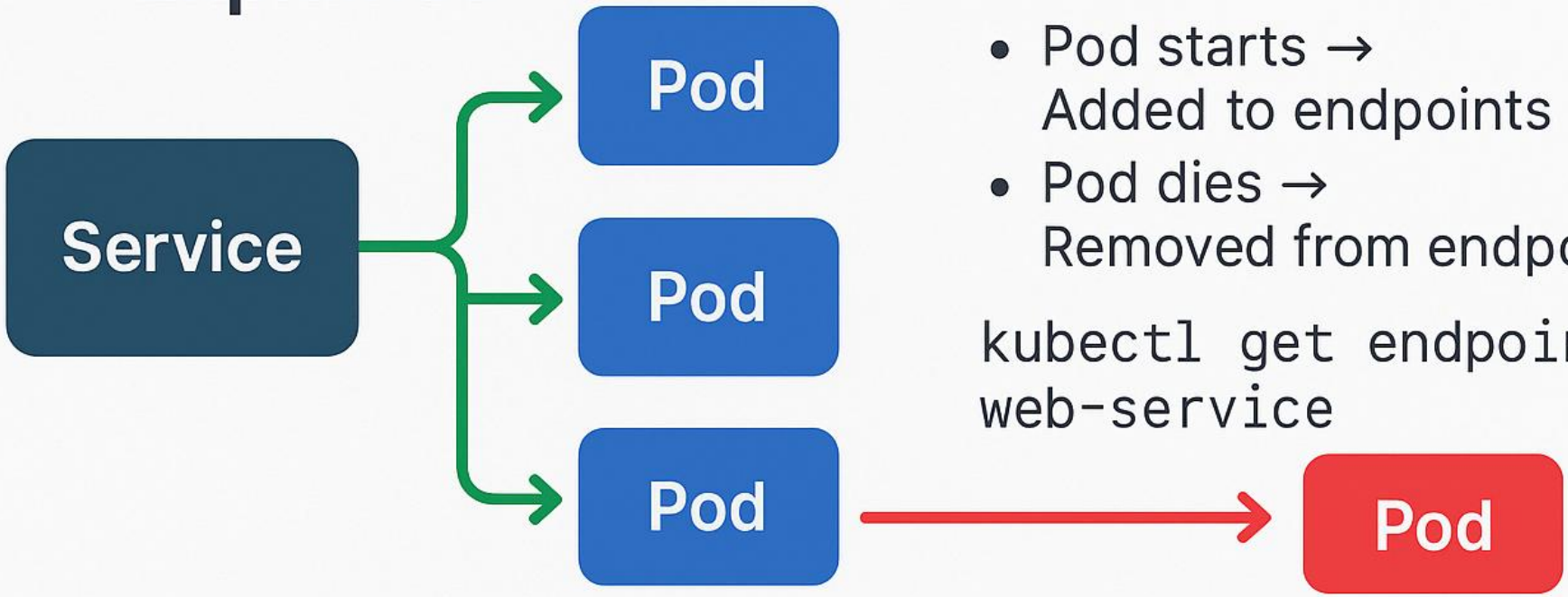
Environment Variables

```
WEB_SERVICE_HOST=10.0.0.1
WEB_SERVICE_PORT=80
```

DNS is automatic and dynamic! 🎯

Endpoints in Action

Service Endpoints Explained



Service automatically tracks pod IPs:

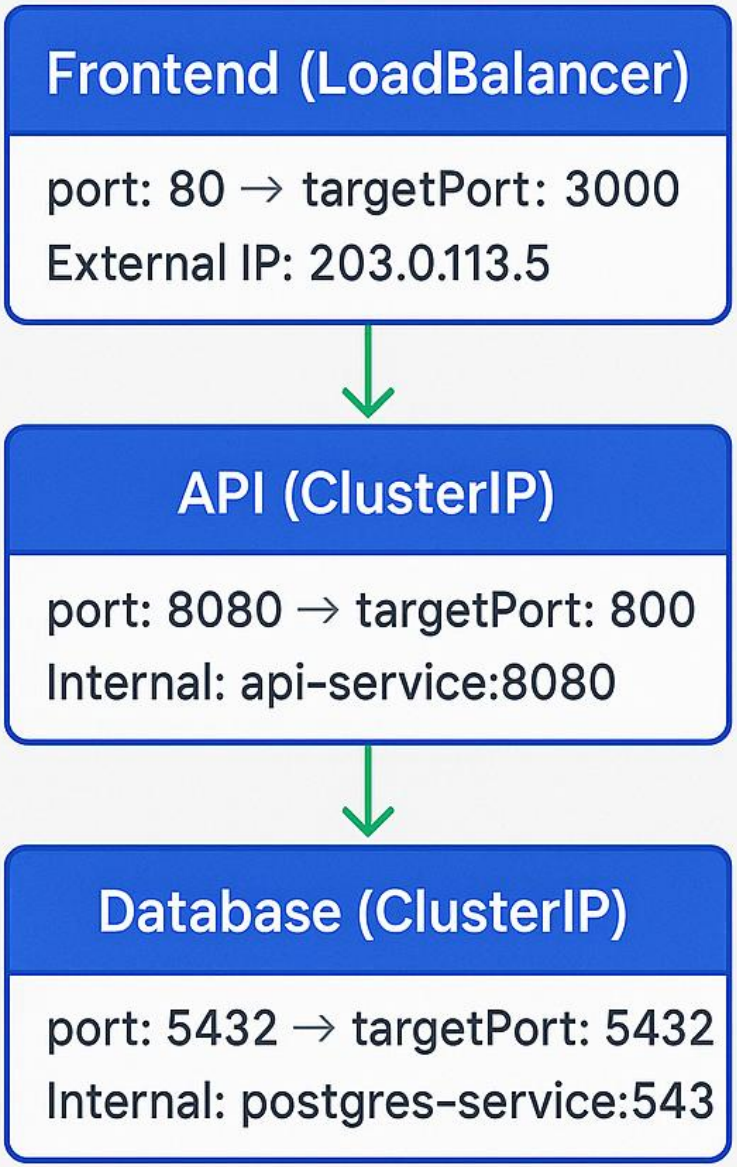
- Pod starts → Added to endpoints
- Pod dies → Removed from endpoints

```
kubectl get endpoints web-service
```

Endpoints = Pod IPs that receive traffic

No endpoints = No traffic routing!

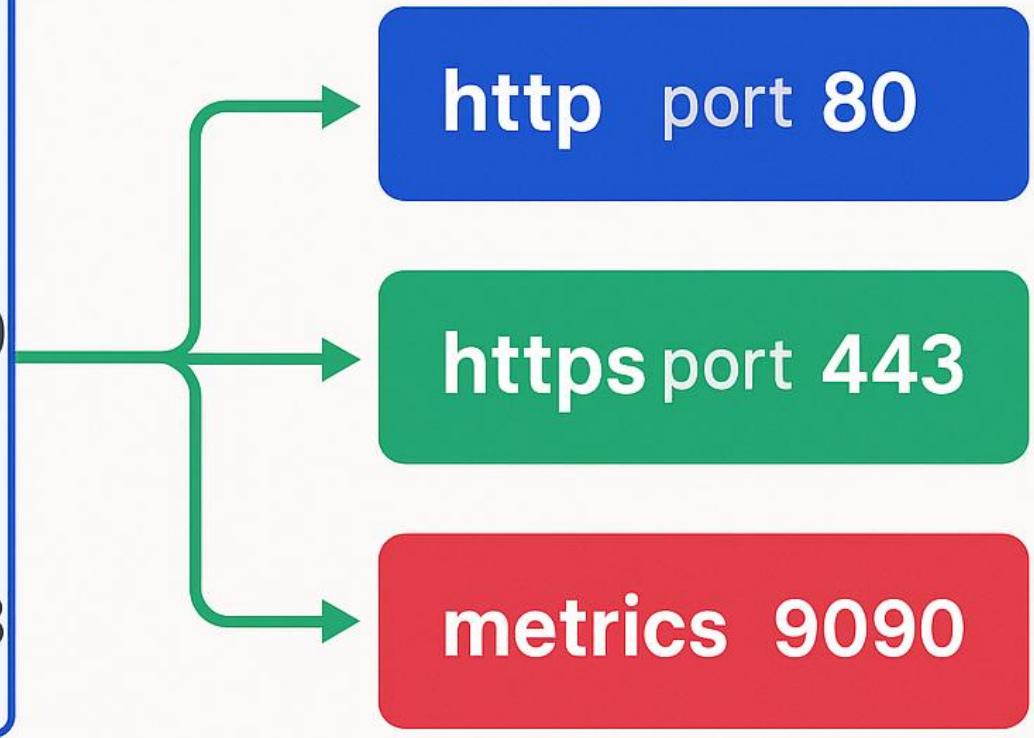
Real Example: E-commerce Platform



MULTI-PORT SERVICES

🔌 Multi-Port Service Configuration

```
spec:
  name: http
    port 80
  targetPort 8080
  name: https
    port 443
  targetPort 8443
```



Name ports for clarity! 📝

Service Types: When to Use

ClusterIP

- ✓ Internal microservices
- ✓ Databases & caches
- ✗ External user access

NodePort

- ✓ Development/ testing
- ✓ Legacy applications
- ✗ Production workloads

LoadBalancer

- ✓ Production external apps
- ✓ High availability needs
- ✗ Cost-sensitive projects

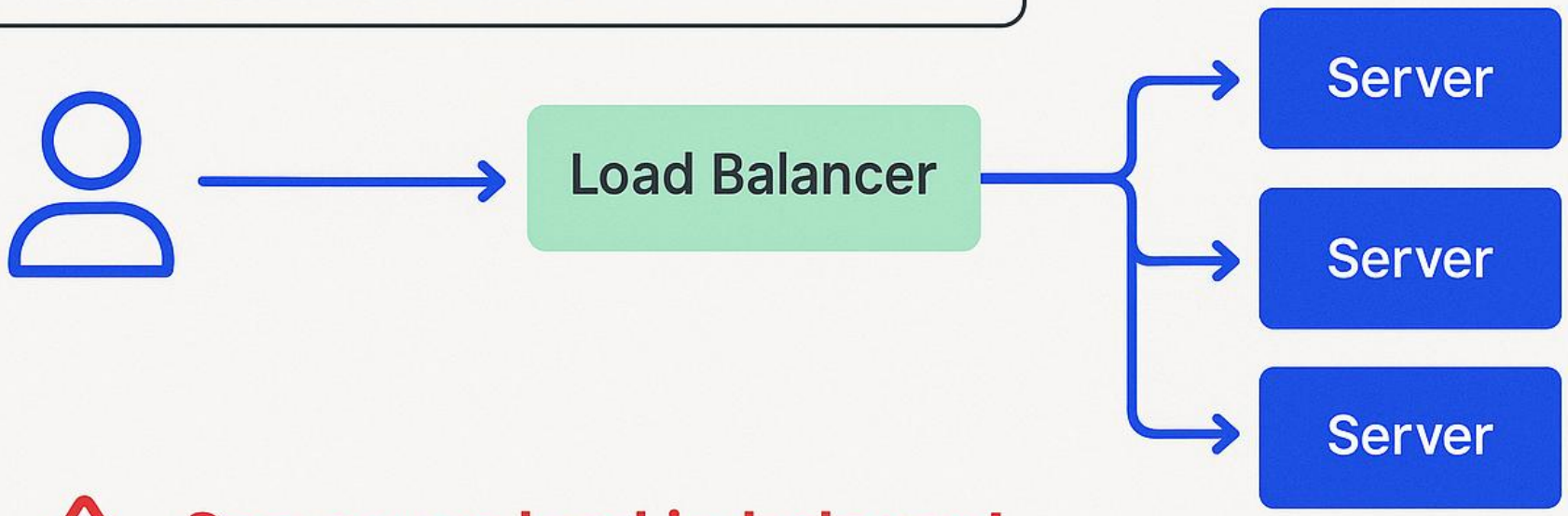
Session Affinity

Session Affinity (Sticky Sessions)

```
spec:
sesssionAffinity: ClientIP
sessionAffinityConfig:
  clientIP:
  timeoutSeconds: 3600
```

Use Cases:

- Stateful applications
- User session data
- Shopping carts



Can cause load imbalance!

Services WITHOUT Selectors

Use for:

- external databases
- legacy systems
- third-party services

Manual Endpoints

```
kind: Endpoints
metadata:
 name: external-db
 subsets:
  - addresses:
    ip: 192.168.1.100
  - ports
    port 5432
```

Headless Services

```
spec:
clusterIP: None
  # No cluster IP!
```

Returns pod IPs directly via DNS:

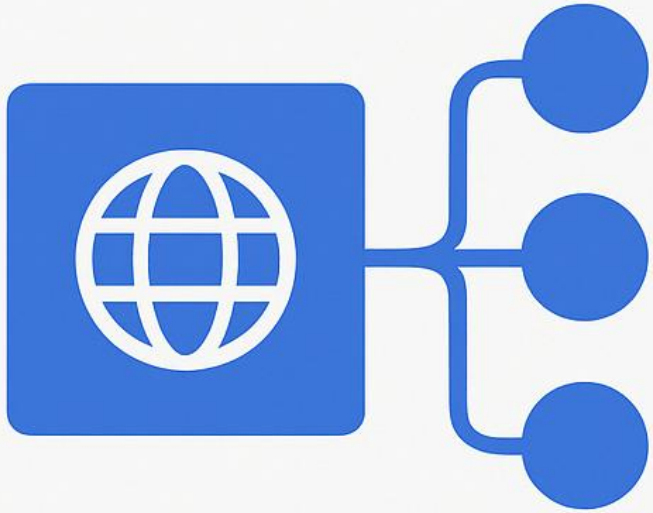
```
web-0.web-service.default.svc.cluster.local
web-1.web-service.default.svc.cluster.local
```

Perfect for StatefulSets! 🎯

Integrating External Services

Three approaches:

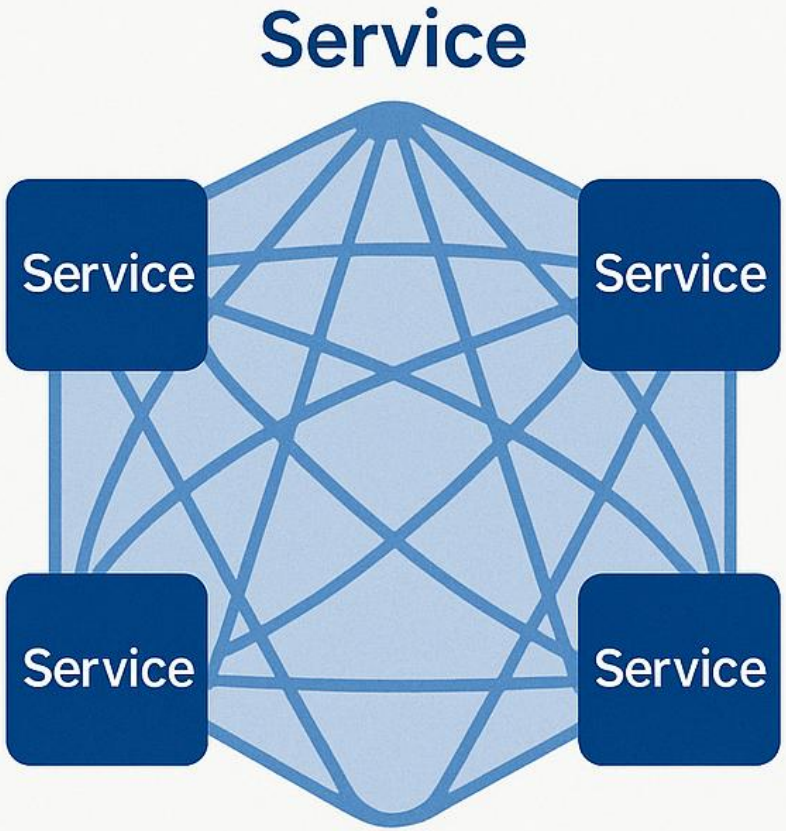
1. Service without selector + Endpoints
2. ExternalName service (DNS CNAME)
3. Service with external IPs



Best for: Databases, APIs, Legacy systems

Enables gradual migration!

Service Mesh Integration



Istio/Linkerd enhance services with:

- ✓ Advanced traffic management
- ✓ Security policies
- ✓ Observability
- ✓ Circuit breaking
- ✓ Retry mechanisms

Services provide foundation!

Cost Optimization Strategies

Service Cost Optimization

LoadBalancer Costs:

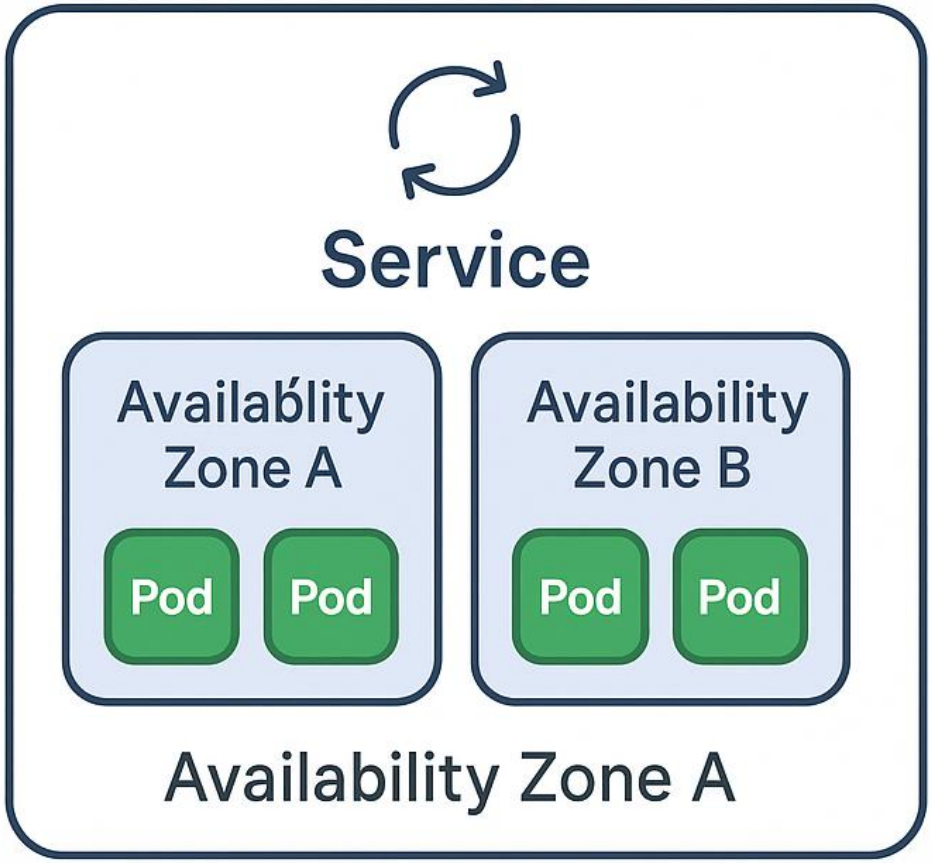
- Each service = $10-20/month
- 10 services = $100-200/month!

Solutions:

- ✓ Use Ingress for HTTP/HTTPS
- ✓ Combine services when possible
- ✓ Use NodePort for internal tools
- ✓ Implement service consolidation

HIGH AVAILABILITY PATTERNS

High Availability with Services



Multi-Zone Deployment

- ✓ Pods across availability zones
- ✓ Service load balances automatically
- ✓ Zone failures handled gracefuly

Health Checks

- ✓ Readiness probes
- ✓ Liveness probes
- ✓ Automatic pod replacement

COMMON SERVICE MISTAKES

- ❌ Mismatched selectors/labels
- ❌ Wrong target ports
- ❌ Missing readiness probes
- ❌ Too many LoadBalancers
- ❌ No resource limits
- ❌ Ignoring endpoints

Always verify endpoints first!

DEBUGGING SERVICES: CONNECTION ISSUES

STEP 1: CHECK SERVICE CONFIGURATION

```
kubectl get service my-service
kubectl describe service my-service
```

- Correct type
- Proper ports
- Valid selectors
- External IP (if LoadBalancer)

DEBUGGING SERVICES

Debugging Step 2: Endpoints

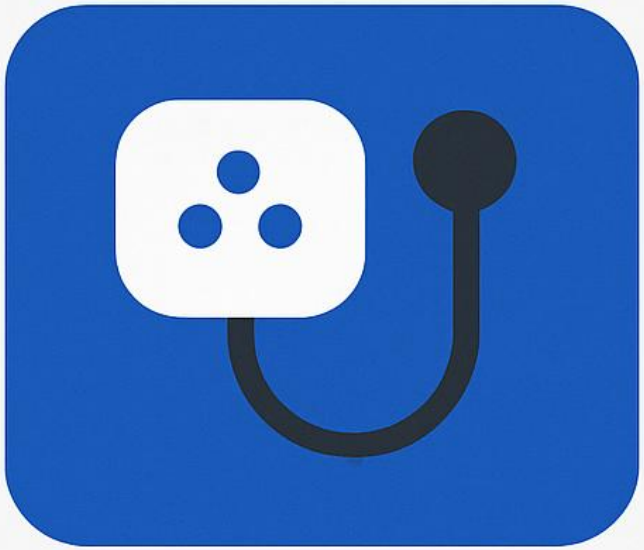
```
kubectl get endpoints my-service
```

- No endpoints = No matching pods
- Empty = Selector mismatch
- Multiple IPs = Working correctly

Check pod labels match service selector!

🚑 Debugging Step 3: Pod Health

```
kubectl get pods -l app=my-app
kubectl describe pod my-pod
```



Check:

- ✔ Pod running status
- ✔ Readiness probe passing
- ✔ Container port exposed
- ✔ Application listening on correct port

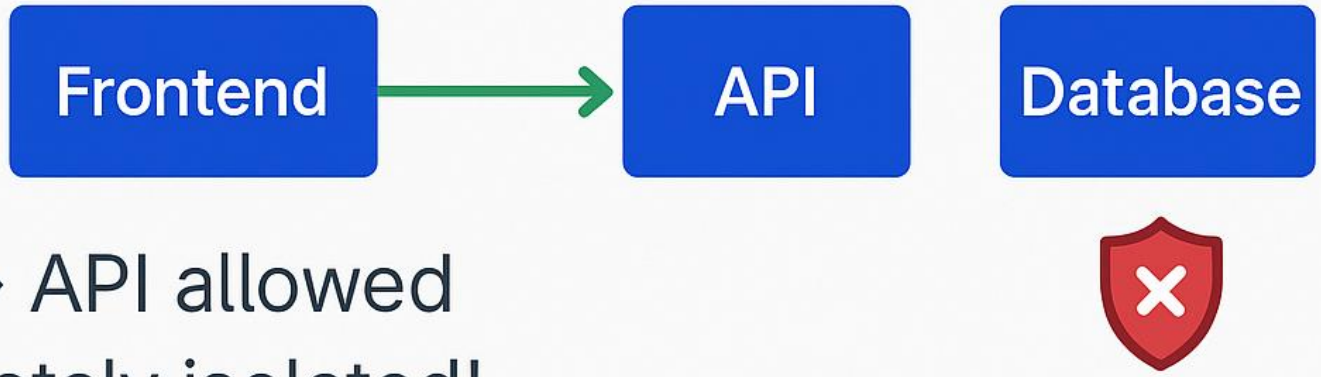
Readiness = Service traffic eligibility!

🔒 Network Policies & Services

Services respect NetworkPolicies:

- ✓ Ingress rules control inbound
- ✓ Egress rules control outbound
- ✓ Default deny recommended

Example:



Only frontend → API allowed
Database completely isolated!

Service Monitoring Essentials

Key Metrics:

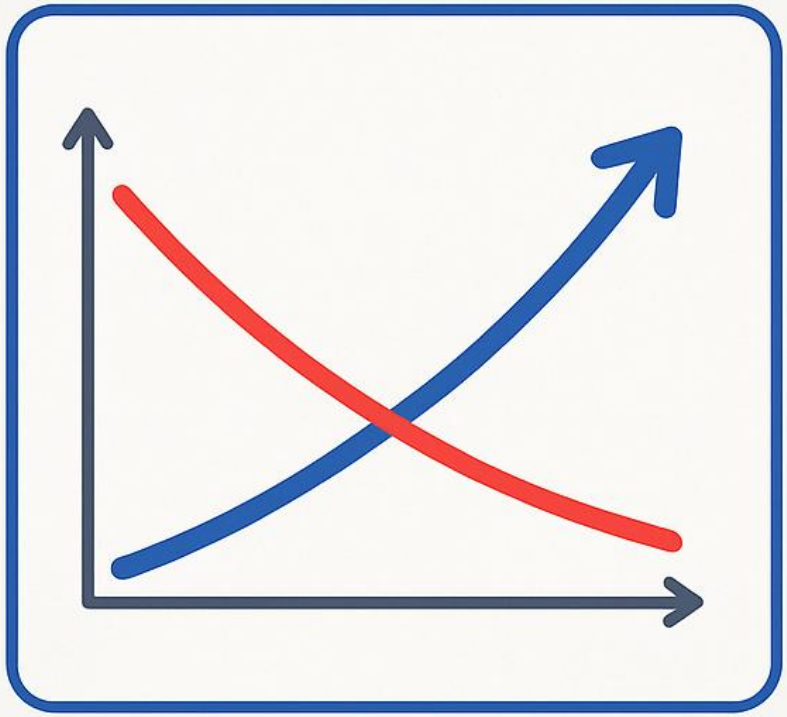
- ✔ Request rate (RPS)
- ✔ Error rate (4xx/5xx)
- ✔ Response latency
- ✔ Endpoint availability
- ✔ Connection count

Tools:

Prometheus,
Grafana
Istio

Service Performance Optimization

1. Optimize pod resource requests/limits
2. Use readiness probes correctly
3. Configure appropriate replica count
4. Enable horizontal pod autoscaling
5. Monitor and adjust based on metrics



Right-size for your workload!

Service Security Checklist

- Use ClusterIP for internal services
- Implement network policies
- Enable TLS/mTLS
- Use service accounts
- Regular security scanning
- Limit LoadBalancer exposure
- Monitor access logs

Testing Service Connectivity

Internal Testing:

```
kubectl run test-pod
--image=busybox -it -rm

wget -O-
http://my-service:80
```

External Testing:

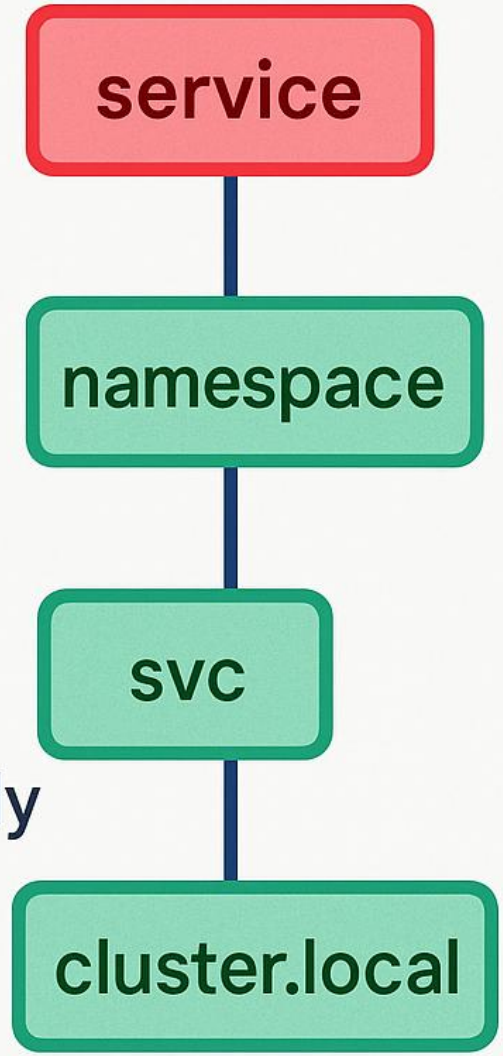
```
curl
httprnal-ip:80

telnet
external-ip 80
```

Always test from pod perspective!

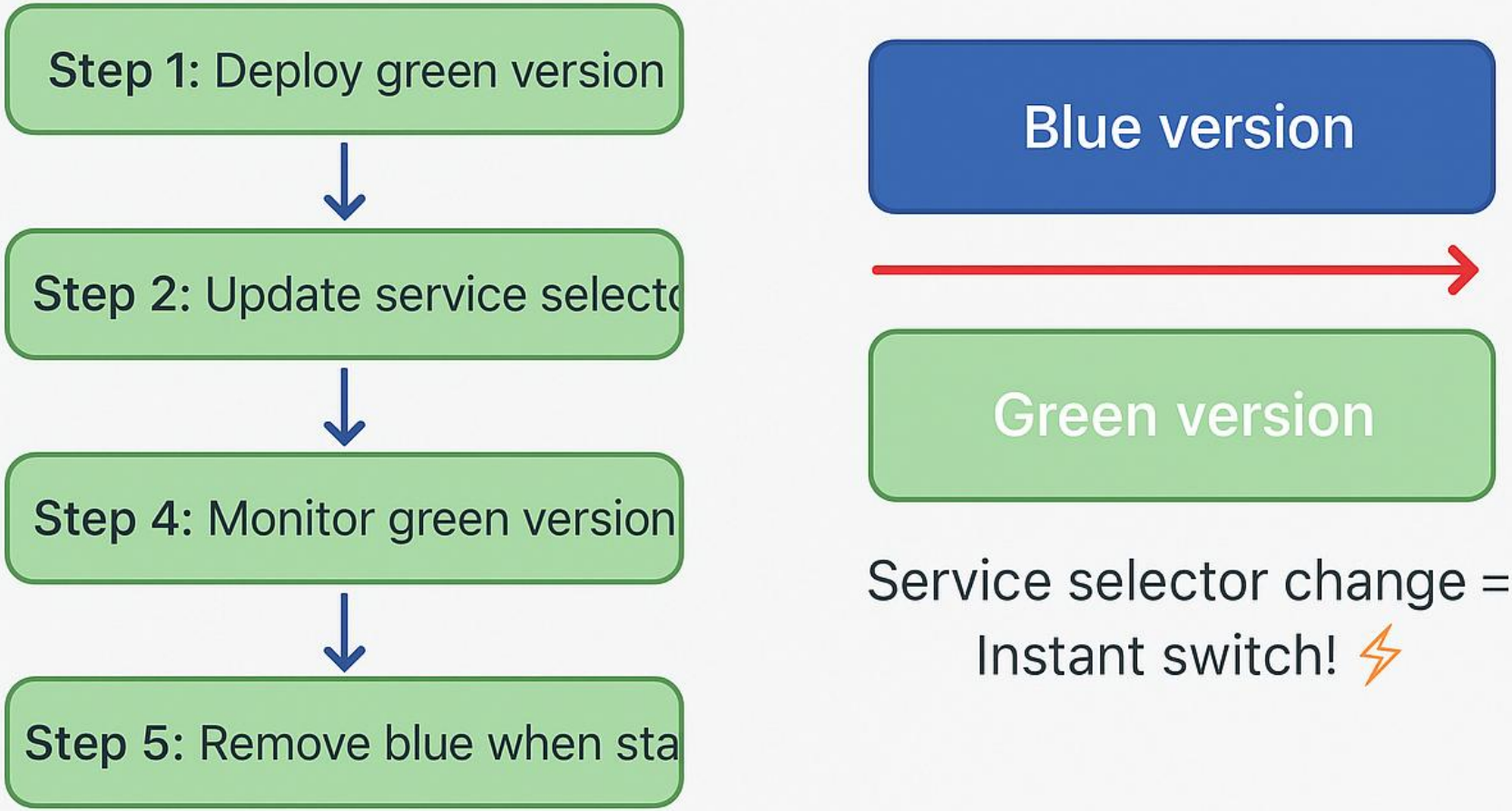
Service Discovery Best Practices

- ✓ Use DNS names, not IPs
- ✓ Include namespace in cross-ns calls
- ✓ Implement circuit breakers
- ✓ Cache DNS lookups appropriately
- ✓ Handle service unavailability gracefuly



BLUE-GREEN DEPLOYMENTS

Blue-Green Deployments

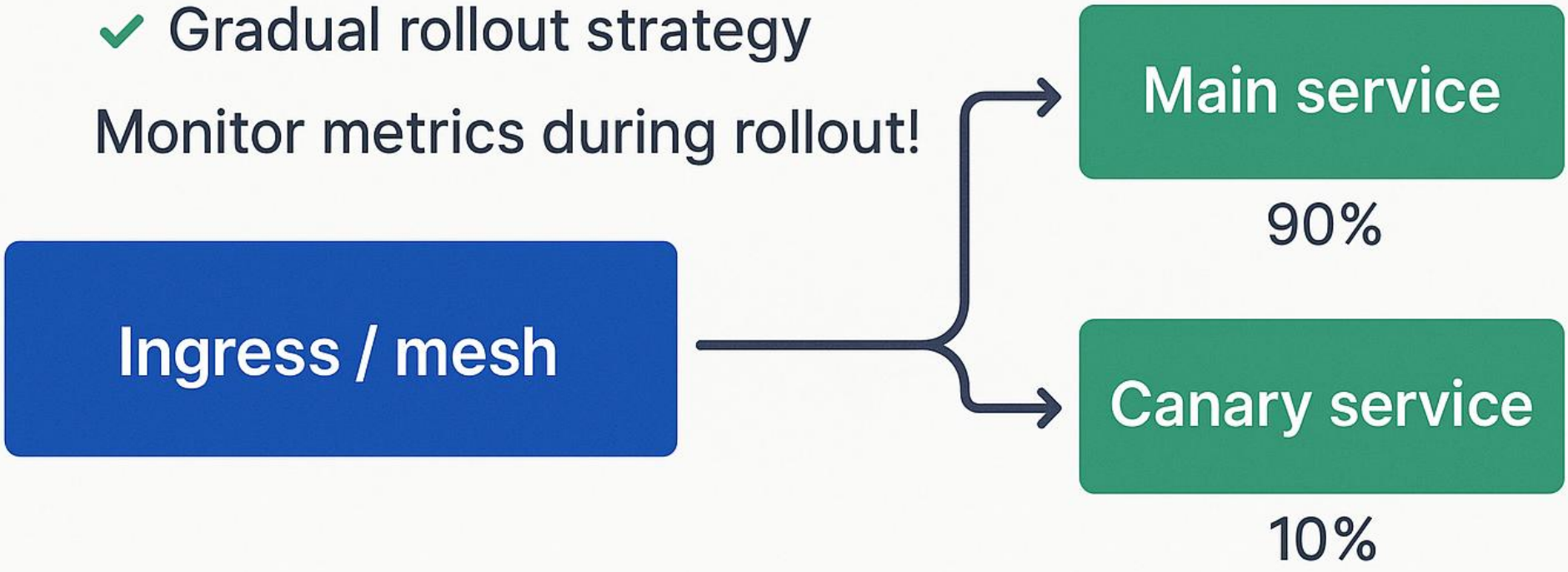


🐤 Canary Deployments

Use multiple services:

- ✔ Main service (90% traffic)
- ✔ Canary service (10% traffic)
- ✔ Ingress/mesh for traffic split
- ✔ Gradual rollout strategy

Monitor metrics during rollout!



Essential Troubleshooting Commands

```
kubectl get svc                          # List services
kubectl describe svc my-service          # Service details
kubectl get endpoints my-service         # Check endpoints
kubectl logs -l app=my-app               # Pod logs
kubectl port-forward svc/my-svc 8080:80  # Test locally
```

Production-Ready Service Template

```
apiVersion: v1
kind: Service
metadata: {{.serviceName }}
namespace: {{ .namespace }}
labels: {app:{{ .appName})
vercion: {{ .version}
spec:
type:{{ .servceiceType}
solector: app:{{.appnme
ports: http
  targetPort: {{ .containerPort}
```

⚖ Load Balancing Algorithms

Kubernetes Uses:

- ✔ Round-robin (default)
- ✔ Random with equal weights
- ✔ Session affinity (when enabled)

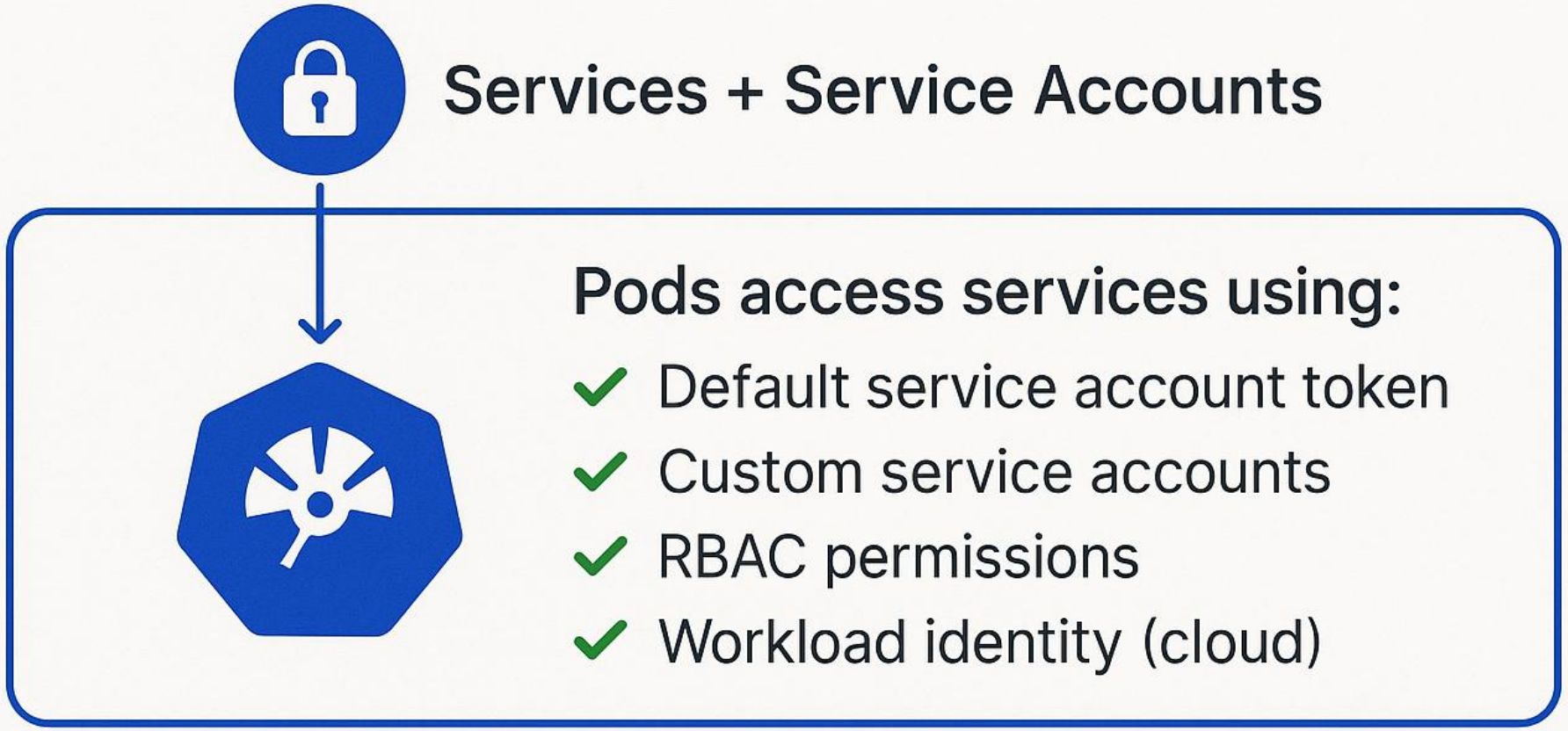
Not Available:

- ✘ Least connections
- ✘ Weighted round-robin
- ✘ Health-based routing

Use service mesh for advanced LB!

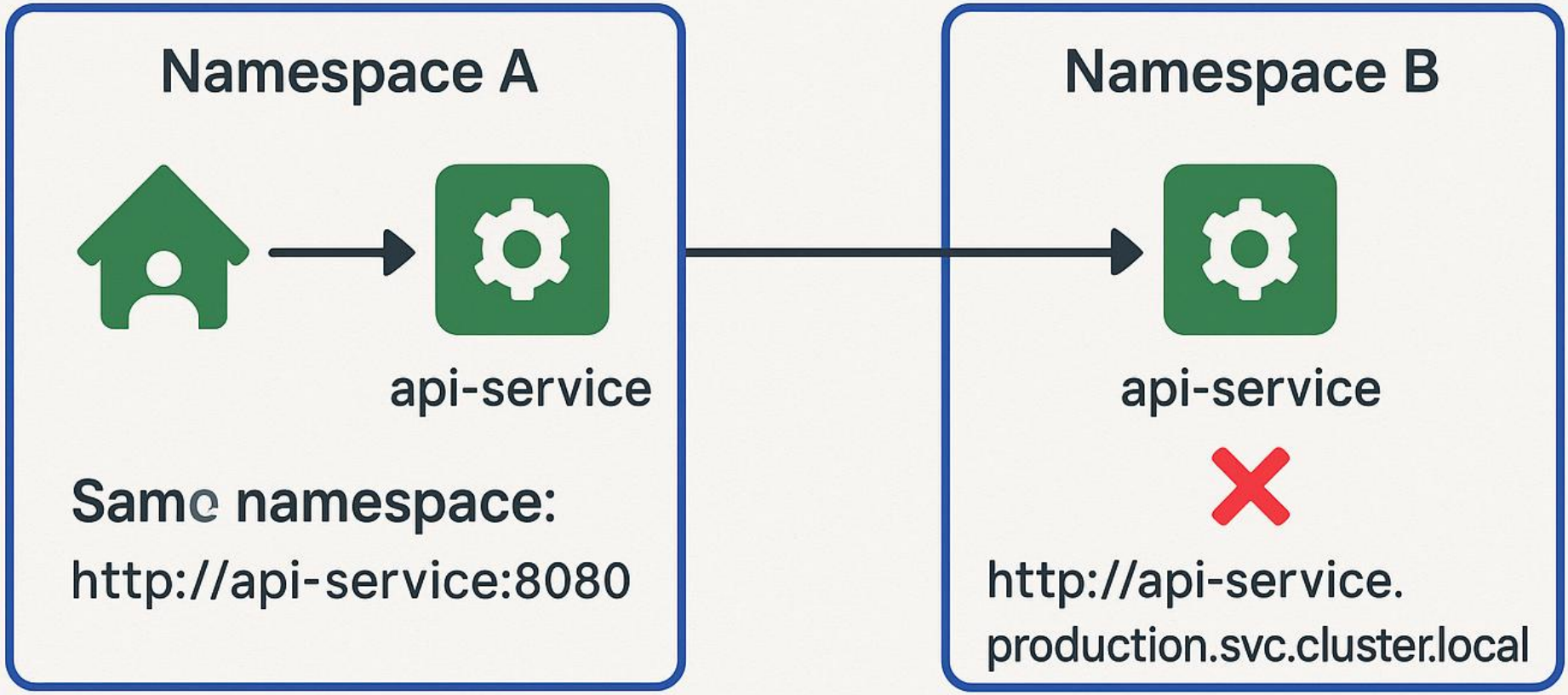
Service Account Integration

RBAC permissions diagram



Secure service-to-service communication!

Service in Different Namespaces

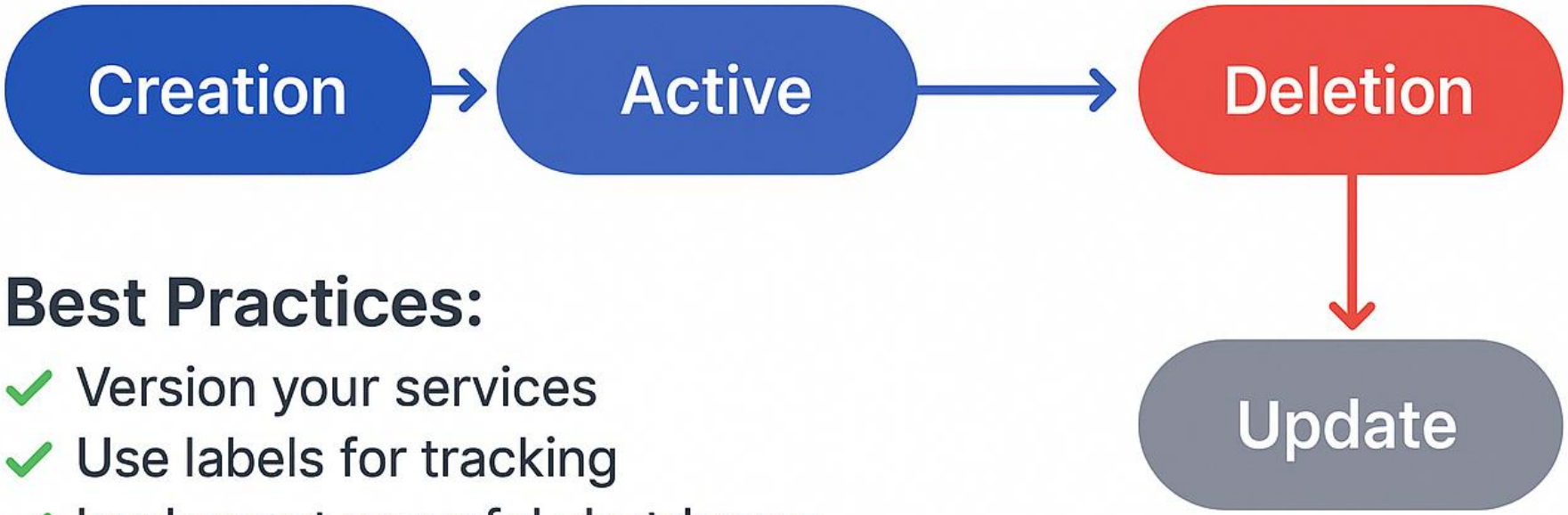


Best Practice:

- ✓ Use NetworkPolicies for isolation
- ✓ Document cross-namespace dependencies

Slide 44: Service Lifecycle Management

🔄 Service Lifecycle



Best Practices:

- ✅ Version your services
- ✅ Use labels for tracking
- ✅ Implement graceful shutdowns
- ✅ Monitor throughout lifecycle
- ✅ Clean up unused services

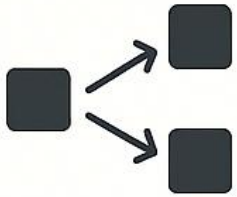
Lifecycle = Cost + Security impact!

💰 🔒

Advanced Service Patterns

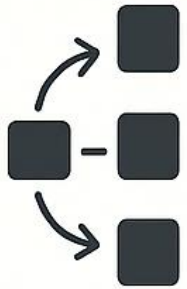
1 Fan-out

One service →
Multiple backends



2 Fan-in

Multiple services →
One backendd



Circuit breaker

Failure isolation

Bulkhead

Resource isolation

Retry

Transient failure handling

Combine patterns for resilience!

Cloud Provider Differences

LoadBalancer Differences

AWS

- Application/ Network LB
- Annotations for configuration

GCP

- HTTP(S) + TCP/UDP LB
- Global/Regional options

Azure

- Standard/Basic LB
- Internal/External options

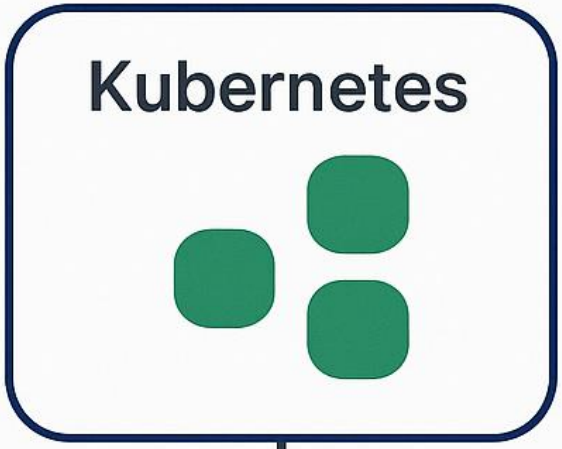
Check provider documentation! 📖

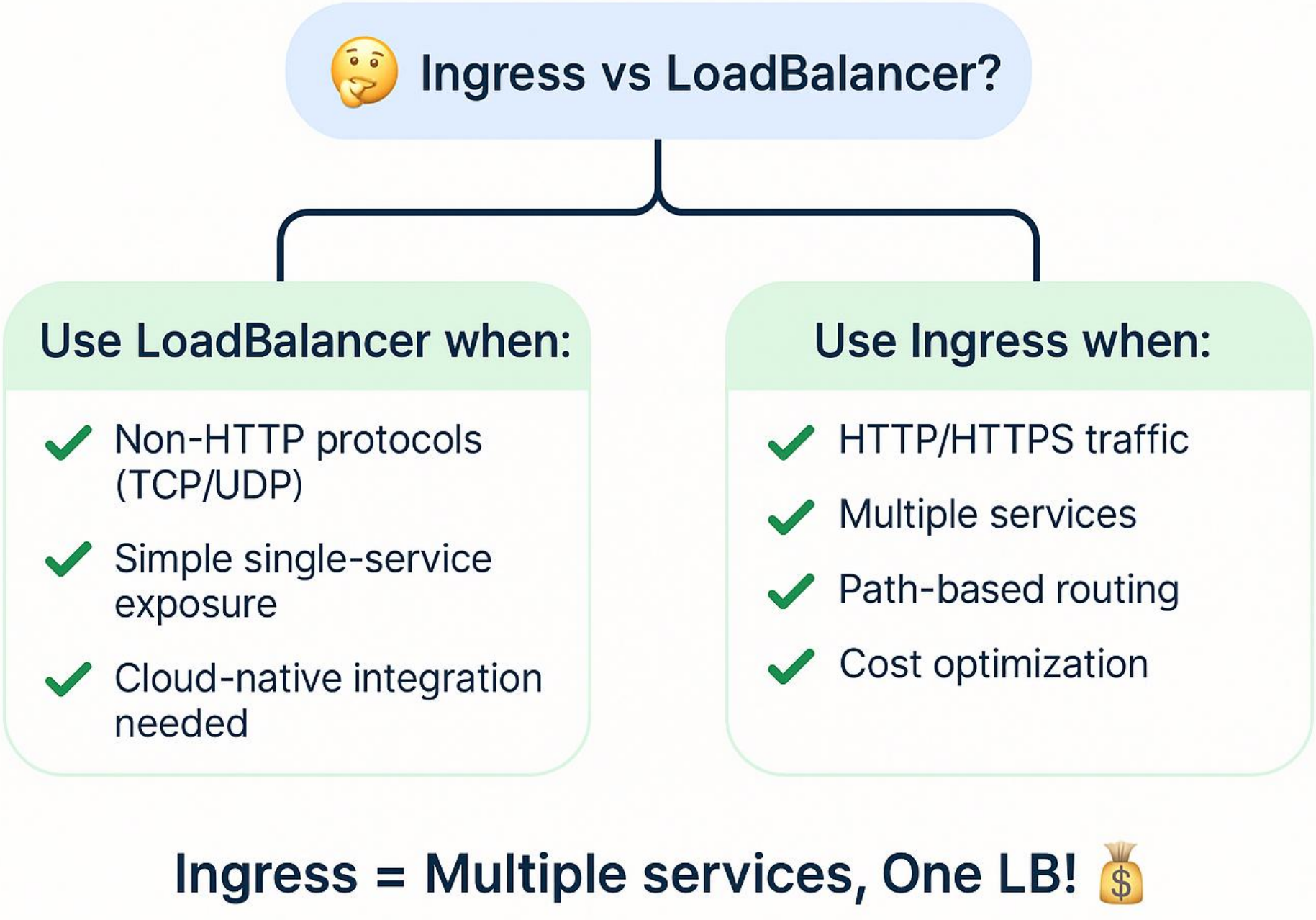
Service Migration Strategies

Legacy — Legacy → Kubernetes

1. Create service without selector
2. Point to external endpoints
3. Migrate pods gradually
4. Update service selector
5. Remove external endpoints

→ Zero-downtime migration!




🤔 Ingress vs LoadBalancer?
Use LoadBalancer when:
Non-HTTP protocols (TCP/UDP)
Simple single-service exposure
Cloud-native integration needed
Use Ingress when:
HTTP/HTTPS traffic
Multiple services
Path-based routing
Cost optimization
Ingress = Multiple services, One LB! 💰

Production Checklist

✅ Production Service Checklist

🔍 Configuration:

- ✓ Resource limits set
- ✓ Health checks configured
- ✓ Labels and selectors match
- ✓ Appropriate service type

🔒 Security:

- ✓ NetworkPolicies enabled
- ✓ Service accounts configured
- ✓ TLS certificates valid

📶 Monitoring:

- ✓ Metrics collection enabled
- ✓ Alerting cenfigured

Key Takeaways

- Services provide stable pod access
- Choose type based on use case
- ClusterIP for internal, LoadBalancer for external
- Always check endpoints first when debugging
- Use Ingress for cost optimization
- Monitor service performance continuously
- Implement security from day one

Master services = Master Kubernetes!

🚀 Ready to Master Kubernetes Services?

👍 **Like** if this helped you

🔁 **Repost** to help others

💬 **Share** your service experiences

🔔 **Follow** for more K8s content

Questions? Drop them in comments!

Next: Kubernetes Ingress Controllers Deep Dive

Salwan Mohamed
DevOps and Platform Engineer