The Chronicles of Clusterville: A Kubernetes Tale

A five-chapter epic where containers come to life and orchestration becomes adventure

---

Chapter 1: The Birth of Clusterville and The Great Pod Housing Crisis

In the beginning, there was chaos. Applications ran wild on bare metal servers, fighting for resources, crashing into each other, and generally making a mess of things. Then came **Master Node**, the wise overseer who would bring order to this digital wilderness.

Master Node wasn't alone. He had three loyal advisors who formed the **Control Plane**:

API Server - The Grand Communicator, a fast-talking diplomat who spoke every language. Every request, every command, every whisper in Clusterville had to pass through him. He was the only one who could talk to **etcd**, the ancient librarian.

Spec Deep Dive: API Server runs on port 6443 (typically) and is the central management entity. It validates and configures data for pods, services, and controllers. When you run `kubectl`, you're talking to API Server. He's stateless and horizontally scalable.

etcd - The Memory Keeper, an obsessive record-keeper who wrote everything down in a distributed key-value diary. "Pod-27 lives at 10.244.1.5," she'd mutter while scribbling. "Deployment-Frontend wants 3 replicas." She never forgot anything and kept multiple backup copies because trust issues.

Spec Deep Dive: etcd is the source of truth. It stores cluster state, configuration data, and metadata. It uses the Raft consensus algorithm - needs a quorum (N/2 + 1) to function. If you have 3 etcd instances, you can lose 1. With 5, you can lose 2. Odd numbers are preferred. Default port: 2379 for client requests, 2380 for peer communication.

Scheduler - The Tetris Master, who decided which **Worker Node** (the actual neighborhoods) should house which **Pod** (the apartment buildings). He was obsessed with efficiency and fairness, always calculating, always optimizing.

Spec Deep Dive: The Scheduler watches for unscheduled pods through API Server. It filters nodes (predicates) - checking if nodes have enough CPU/memory, if ports are available, if labels match. Then it scores remaining nodes (priorities) based on resource balance, affinity rules, taints/tolerations. The highest-scoring node wins. This happens in milliseconds.

Controller Manager - The Anxious Parent, who constantly worried about everyone's wellbeing. He ran multiple sub-personalities: ReplicaSet Controller (counts pods obsessively), Deployment Controller (manages rollouts), Node Controller (checks if worker nodes are alive), and more. If something wasn't matching the "desired state," he'd panic and fix it immediately.

Spec Deep Dive: Controller Manager runs control loops that watch the cluster state through API Server and make changes to move current state toward desired state. Each controller is technically a separate process, but they're compiled into a single binary for efficiency. Reconciliation loop typically runs every 10 seconds.

The Worker Nodes: Where Life Happens

While Master Node and the Control Plane lived in their ivory tower, the real action happened in the **Worker Nodes** - the bustling neighborhoods of Clusterville. Each Worker Node had three essential residents:

Kubelet - The Neighborhood Mayor, a middle manager who took orders from the Control Plane and made sure they happened locally. He was on every Worker Node, constantly reporting back to API Server: "Yes, Pod-27 is running. Yes, Container-A inside Pod-27 is healthy. Yes, yes, I'm checking!"

Spec Deep Dive: Kubelet is the primary node agent. It registers the node with API Server, receives PodSpecs (Pod definitions), and ensures containers are running and healthy by working with the container runtime. It reports node and pod status back to API Server every 10 seconds (default). It also manages the mounting of volumes into pods. Kubelet doesn't manage containers not created by Kubernetes.

Kube-proxy - The Traffic Cop, who managed all the networking magic. He maintained network rules and made sure traffic going to a "Service" address got distributed properly to the actual Pods behind it. He knew every shortcut, every route, and never caused a traffic jam.

Spec Deep Dive: Kube-proxy maintains network rules on nodes using iptables, IPVS, or eBPF (depending on mode). When a Service is created, kube-proxy creates rules to forward traffic from the Service's ClusterIP to backend Pods. In iptables mode (most common), it translates Service IPs to Pod IPs. In IPVS mode, it provides better performance for large services with load balancing algorithms like round-robin, least connection, etc.

Container Runtime - The Construction Crew (Docker, containerd, CRI-O), who actually built and maintained the containers. They were the muscle, the ones doing the physical work while everyone else managed and coordinated.

Spec Deep Dive: The Container Runtime Interface (CRI) allows Kubernetes to use different runtimes. containerd is now the most common (Docker is deprecated in K8s but containerd was always doing the real work under Docker). The runtime pulls images, creates container namespaces, manages cgroups for resource limits, and reports container status back to Kubelet.

Pods: The Apartment Buildings

In Clusterville, applications didn't live in houses - they lived in **Pods**, which were like apartment buildings. But here's the weird thing: most Pods only had ONE apartment (container). Sometimes they'd have 2-3 apartments (containers) that were so closely related they needed to share the same building - same address (IP), same utilities (volumes), same mailbox (network namespace).

Pod-Frontend-X7B3 was a typical resident. She was a cozy one-bedroom (single nginx container) sitting on Worker-Node-1.

Spec Deep Dive:

yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-frontend-x7b3
  labels:
    app: frontend
    tier: web
spec:
  containers:
  - name: nginx
    image: nginx:1.19
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
    ports:
    - containerPort: 80
```

Resources explained:

- **requests**: "I need at least this much to function" - used by Scheduler to place the Pod
- **limits**: "Don't let me use more than this" - enforced by the node
- CPU measured in millicores (m): 1000m = 1 CPU core
- Memory in bytes: Mi (mebibytes), Gi (gibibytes)

If a container exceeds memory limits: **OOMKilled** (Out of Memory Killed) - one of the most common errors you'll see. The container gets terminated and might restart based on restartPolicy.

If a container exceeds CPU limits: It gets throttled (CPU cycles restricted), but not killed. You'll see high CPU throttling metrics.

The Great Crash of Pod-Frontend

One day, Pod-Frontend-X7B3 crashed. Just stopped responding. Citizens panicked. But Controller Manager, ever vigilant, noticed immediately.

"The desired state says we should have 3 Frontend pods," he muttered, checking his notes. "But we only have 2 running. UNACCEPTABLE!"

He spun up **Pod-Frontend-K9M2** on Worker-Node-2 within seconds. Crisis averted.

But this crash revealed something important: How did anyone know the pod had crashed? Enter our first set of special characters...

Chapter 2: The Three Wise Probes and the Health Inspector

After the Great Crash, the Council of Clusterville decided they needed a better early warning system. They couldn't just wait for pods to completely die before taking action. They needed... **Probes**.

Three types of probes were appointed, each with a different philosophy:

Liveness Probe - "The Grim Reaper"

Liveness was morbid. His only job was to check if containers were alive or dead. Not "functioning well" - just "are you literally alive?" If you failed his check, you got restarted. No second chances. No negotiations.

"Are you alive?" he'd ask every 10 seconds, knocking on Pod-Frontend's door.

Spec Deep Dive:

yaml

```yaml
livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
    httpHeaders:
    - name: Custom-Header
      value: LivenessCheck
  initialDelaySeconds: 15  # Wait this long before first check
  periodSeconds: 10        # Check every 10 seconds
  timeoutSeconds: 1        # Wait 1 second for response
  failureThreshold: 3      # Fail 3 times before restarting
  successThreshold: 1      # Only need 1 success to be "alive"
```

Probe Types:

1. **httpGet**: Makes HTTP GET request, expects 200-399 status code
2. **tcpSocket**: Tries to open TCP connection
3. **exec**: Runs a command in the container, expects exit code 0

Common Liveness Errors:

- **CrashLoopBackOff**: Container keeps failing liveness checks, gets restarted, fails again, repeat. The backoff increases exponentially (0s, 10s, 20s, 40s... up to 5 minutes).
- **initialDelaySeconds too short**: Container hasn't finished starting, liveness check fails, container gets killed. Set this LONGER than your startup time.
- **Checking dependencies**: Never check external dependencies (database, APIs) in liveness. You don't want to restart your app because your database is slow. Only check internal health.

One day, Pod-Backend-Database was under heavy load. His liveness probe checked if the database could respond in 1 second. Under load, it took 3 seconds. Liveness Probe declared him DEAD and restarted him. This made things WORSE. All connections were lost, causing more load on other pods, triggering more restarts. This is called a **restart cascade** or **thundering herd problem**.

The fix: Use Readiness Probe for traffic management, save Liveness for actual deadlock/hung process detection.

Readiness Probe - "The Bouncer"

Readiness was more diplomatic. Her job was to determine if a pod was ready to receive traffic. She didn't kill anyone - she just told the Service, "Don't send traffic here yet."

This was crucial during startups, during heavy load, or during temporary issues.

Spec Deep Dive:

yaml

```yaml
readinessProbe:
  httpGet:
    path: /ready
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 5
  failureThreshold: 3
```

When Readiness Probe failed, the pod was removed from the Service's **Endpoints** list. Kube-proxy stopped sending traffic. But the pod kept running - it got a chance to recover.

Real-world scenario: Pod-Frontend needs to load a 500MB cache file at startup. Takes 30 seconds.

- Liveness: `initialDelaySeconds: 40` (give it time to start)
- Readiness: `initialDelaySeconds: 5`, checking every 5 seconds

For the first 30 seconds, Readiness fails, so no traffic is routed. Once the cache loads and `/ready` returns 200, Readiness passes, and traffic flows in. Liveness doesn't interfere because it's not checking yet.

Startup Probe - "The Patient Mentor"

Startup was the newest probe, added because some applications were slow starters. Legacy Java apps that took 90 seconds to boot. Machine learning models that loaded huge weights at startup.

Before Startup Probe existed, you had to set huge `initialDelaySeconds` on Liveness, which meant you wouldn't detect a hung process for 90+ seconds even after the pod was running.

Startup Probe solved this: "I'll handle the slow startup. Once I succeed, Liveness and Readiness take over."

Spec Deep Dive:

yaml

```yaml
startupProbe:
  httpGet:
    path: /startup
    port: 8080
  periodSeconds: 10
  failureThreshold: 30    # 30 failures * 10 seconds = 5 minutes max startup
  # No initialDelaySeconds needed

livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
  periodSeconds: 10
  failureThreshold: 3
```

The genius: Liveness and Readiness are DISABLED until Startup Probe succeeds once. This gives slow-starting apps up to `periodSeconds * failureThreshold` time to start (5 minutes in this example), but once started, Liveness catches issues within 30 seconds (10s * 3 failures).

The Tale of the Misconfigured Probe

Pod-Payment-Gateway had a developer who thought they were being clever:

yaml

```yaml
livenessProbe:
  httpGet:
    path: /health
    port: 8080
  periodSeconds: 1        # Check every second! So responsive!
  timeoutSeconds: 5        # Give it 5 seconds to respond
  failureThreshold: 1     # One strike and you're out!
```

Disaster struck. A minor network blip caused one health check to fail. Pod-Payment-Gateway was immediately restarted. In the middle of processing payments. Transactions lost. Customers angry. Developers sad.

Probe Best Practices:

1. **Liveness**: Set `failureThreshold` to 3+. Give benefit of doubt.
2. **Readiness**: Can be more sensitive, but not too aggressive
3. **periodSeconds**: 10 seconds is usually fine. Don't check every second (wasteful).
4. **timeoutSeconds**: 1-2 seconds unless you have good reason
5. **Separate endpoints**: `/healthz` (liveness) vs `/ready` (readiness) with different logic
6. **Never check dependencies in Liveness**: Only check internal health

---

Chapter 3: The Love Triangle - Ingress, Services, and the DNS Matchmaker

Pod-Frontend and Pod-Backend had a problem. They were in love (they needed to communicate), but they kept changing addresses. Every time a pod restarted, it got a new IP. How could they find each other?

Enter **Service** - the matchmaker who provided stable identities.

Service: The Stable Address

Service-Frontend was like a phone number that always worked, even if you changed phones. She had a fixed ClusterIP (internal IP address) and a name (`frontend-service`). When anyone wanted to talk to Frontend pods, they called the Service, and she'd route them to healthy pods.

Spec Deep Dive:

yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  selector:
    app: frontend        # Matches pods with this label
  ports:
  - protocol: TCP
    port: 80             # Service port
    targetPort: 8080     # Container port
  type: ClusterIP        # Default, internal only
```

How it works:

1. Service is created, API Server assigns it a ClusterIP (e.g., 10.96.0.15)
2. CoreDNS (the DNS service) creates a record: `frontend-service.default.svc.cluster.local` → 10.96.0.15
3. Kube-proxy on each node creates iptables rules: Traffic to 10.96.0.15:80 → distribute to Pod IPs on port 8080
4. When Pod-Backend wants to reach Frontend, it just calls `http://frontend-service` (DNS resolves it)

Service Types:

ClusterIP (default): Only accessible within cluster

yaml

```yaml
type: ClusterIP
```

NodePort: Accessible from outside cluster on a specific port (30000-32767) on ANY node

yaml

```yaml
type: NodePort
  ports:
  - port: 80
    targetPort: 8080
    nodePort: 30080     # Optional, auto-assigned if omitted
```

Access via `http://<any-node-ip>:30080`. Traffic goes to Service, then to Pods (might be on different nodes).

LoadBalancer: Creates an external load balancer (cloud provider specific)

yaml

```yaml
type: LoadBalancer
```

On AWS, this creates an ELB/ALB. On GCP, a Load Balancer. Gets an external IP.

ExternalName: Maps to an external DNS name (no proxying)

yaml

```yaml
type: ExternalName
externalName: my-database.example.com
```

Ingress: The Grand Gateway

But LoadBalancer Services had a problem: Each one created a separate (expensive) cloud load balancer. For 10 services, you paid for 10 load balancers.

Ingress was the solution - a sophisticated host/path-based router. One load balancer, many services.

Spec Deep Dive:

yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: main-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    cert-manager.io/cluster-issuer: "letsencrypt-prod"
spec:
  ingressClassName: nginx    # Which controller to use
  tls:
  - hosts:
    - myapp.example.com
    secretName: myapp-tls     # Where TLS cert is stored
  rules:
  - host: myapp.example.com
    http:
      paths:
      - path: /api
        pathType: Prefix
        backend:
          service:
            name: backend-service
            port:
              number: 80
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend-service
            port:
              number: 80
```

How Ingress Works:

1. You deploy an **Ingress Controller** (nginx-ingress, traefik, HAProxy, etc.) - this is a pod that watches for Ingress resources
2. The Ingress Controller creates a LoadBalancer Service (one for all Ingresses)
3. You create Ingress resources defining rules
4. Ingress Controller configures itself based on these rules
5. Traffic flow: Internet → LoadBalancer → Ingress Controller Pod → Service → Backend Pods

pathType Options:

- **Prefix**: Matches /api, /api/, /api/users, /api/users/123
- **Exact**: Only matches exact path
- **ImplementationSpecific**: Depends on controller

Common Ingress Issues:

Issue 1: 404 Not Found despite correct Ingress config

- Check: Is the Ingress Controller installed? `kubectl get pods -n ingress-nginx`
- Check: Does the Service exist and have Endpoints? `kubectl get endpoints backend-service`
- Check: Is the path rewrite correct? If backend expects `/` but gets `/api/users`, it fails.

Issue 2: Default backend - 404

- The Ingress needs a default backend for unmatched requests
- Usually you deploy a default-backend service that shows a nice 404 page

Issue 3: TLS/HTTPS not working

- Check: Does the Secret exist? `kubectl get secret myapp-tls`
- Check: Is cert-manager running and issuing certificates?
- Check: Are DNS records pointing to the LoadBalancer IP?

The Great Ingress Saga

One day, Traffic Controller wanted to update the Ingress rules. He wanted `myapp.example.com/v2` to route to the new Backend-V2 service.

yaml

```yaml
- path: /v2
  pathType: Prefix
  backend:
    service:
      name: backend-v2-service
      port:
        number: 80
```

He applied the config. Immediately, half the traffic started failing!

What happened?: Path priority matters. His config had:

yaml

```yaml
- path: /          # This matches EVERYTHING
  ...
- path: /v2        # This never gets hit!
  ...
```

Fix: Most specific paths first:

yaml

```yaml
  - path: /v2
    ...
  - path: /api
    ...
  - path: /
    ...
```

Another common gotcha - rewrite-target:

Without rewrite:

- Request: `GET /api/users`
- Ingress forwards to backend: `GET /api/users`
- Backend expects `/users` (doesn't know about `/api` prefix)
- Result: 404

With rewrite:

yaml

```yaml
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$2
  ...
  path: /api(/|$)(.*)
```

- Request: `GET /api/users`
- Ingress forwards: `GET /users`
- Backend happy!

Chapter 4: The Vault Heist - Secrets, ConfigMaps, and the PV/PVC Storage Wars

Every application in Clusterville had secrets. Database passwords, API keys, TLS certificates. Where should they store them?

ConfigMaps and Secrets: The Dynamic Duo

ConfigMap was the town crier - information everyone could know. Configuration settings, non-sensitive environment variables.

Secret was the spy - information that needed to be protected (though not as well as everyone thought - secrets in Kubernetes are only base64 encoded, not encrypted at rest by default).

ConfigMap Spec:

yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  database_url: "postgres://db.example.com:5432"
  log_level: "info"
  feature_flags.json: |
    {
      "new_feature": true,
      "beta_mode": false
    }
```

Secret Spec:

yaml

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: app-secrets
type: Opaque
data:
  db_password: cGFzc3dvcmQxMjM=  # base64 encoded
  api_key: c2VjcmV0a2V5NDU2      # base64 encoded
```

Using them in Pods:

As environment variables:

yaml

```yaml
spec:
  containers:
  - name: app
    env:
    - name: DATABASE_URL
      valueFrom:
        configMapKeyRef:
          name: app-config
          key: database_url
    - name: DB_PASSWORD
      valueFrom:
        secretKeyRef:
          name: app-secrets
          key: db_password
```

As mounted files:

yaml

```yaml
spec:
  containers:
  - name: app
    volumeMounts:
    - name: config-volume
      mountPath: /etc/config
    - name: secret-volume
      mountPath: /etc/secrets
      readOnly: true
  volumes:
  - name: config-volume
    configMap:
      name: app-config
  - name: secret-volume
    secret:
      secretName: app-secrets
```

Now `/etc/config/database_url` contains the database URL, and `/etc/secrets/db_password` contains the password.

Secret Types:

- **Opaque**: Generic secret (default)
- **kubernetes.io/service-account-token**: Service account token
- **kubernetes.io/dockerconfigjson**: Docker registry credentials

yaml

```yaml
type: kubernetes.io/dockerconfigjson
data:
  .dockerconfigjson: <base64-encoded-docker-config>
```

- **kubernetes.io/tls**: TLS certificate and key

yaml

```yaml
type: kubernetes.io/tls
data:
  tls.crt: <base64-cert>
  tls.key: <base64-key>
```

Enter HashiCorp Vault: The Real Spy

But Secret wasn't really secure. Anyone with access to etcd could read secrets. Anyone with kubectl access could decode them (`kubectl get secret app-secrets -o jsonpath='{.data.db_password}' | base64 -d`).

The town hired **Vault** - a professional secret keeper who lived outside Clusterville (external service) but worked closely with the residents.

Vault's Approach:

1. Secrets are encrypted at rest in Vault (not Kubernetes)
2. Applications authenticate to Vault to retrieve secrets
3. Secrets can be dynamically generated (like database credentials with TTL)
4. Full audit logging of who accessed what secret when

Integration Methods:

Method 1: Vault Agent Sidecar

yaml

```yaml
spec:
  containers:
  - name: app
    ...
  - name: vault-agent
    image: vault:1.12.0
    # Vault agent authenticates, fetches secrets, writes to shared volume
```

Method 2: Vault CSI Provider Uses the Secrets Store CSI driver to mount Vault secrets as volumes:

yaml

```yaml
spec:
  volumes:
  - name: secrets-store
    csi:
      driver: secrets-store.csi.k8s.io
      readOnly: true
      volumeAttributes:
        secretProviderClass: "vault-database"
```

Method 3: External Secrets Operator Syncs secrets from Vault to Kubernetes Secrets:

yaml

```yaml
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: vault-example
spec:
  refreshInterval: 1h
  secretStoreRef:
    name: vault-backend
    kind: SecretStore
  target:
    name: app-secrets
  data:
  - secretKey: db_password
    remoteRef:
      key: database/config
      property: password
```

The Heist Gone Wrong:

One day, Pod-API-Gateway needed a database password. The developer hardcoded it:

yaml

```yaml
env:
- name: DB_PASSWORD
  value: "SuperSecret123"  # DON'T DO THIS!
```

Anyone running `kubectl describe pod` could see it. Anyone with access to the Git repo where this YAML was stored could see it. It got checked into version control history forever.

The right way:

1. Store in Vault or external secret manager
2. Use ExternalSecrets or CSI driver to bring into K8s
3. Reference in Pod spec as a Secret
4. Never commit secrets to Git (use tools like git-secrets, detect-secrets)

PersistentVolumes and PersistentVolumeClaims: The Storage Real Estate Market

Pod-Database had a problem: Every time he restarted, his data vanished! Pods are ephemeral - their storage is temporary.

He needed **PersistentVolume** - real estate that existed independently of pods.

The Storage Hierarchy:

1. **PersistentVolume (PV)**: The actual storage (like a warehouse)
2. **PersistentVolumeClaim (PVC)**: A request for storage (like renting warehouse space)

3. **StorageClass**: The real estate agency that can provision PVs on-demand

PersistentVolume Spec:

yaml

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-database
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
  - ReadWriteOnce          # RWO: one node can mount read-write
  persistentVolumeReclaimPolicy: Retain
  storageClassName: fast-ssd
  hostPath:                # DON'T USE IN PRODUCTION
    path: /data/pv-database
```

AccessModes:

- **ReadWriteOnce (RWO)**: One node mounts read-write (most common)
- **ReadOnlyMany (ROX)**: Many nodes mount read-only
- **ReadWriteMany (RWX)**: Many nodes mount read-write (needs NFS, or cloud provider support)

ReclaimPolicy:

- **Retain**: PV remains after PVC deletion (manual cleanup required)
- **Delete**: PV automatically deleted when PVC deleted (data lost!)
- **Recycle**: Deprecated (basic scrub `rm -rf /volume/*`)

PersistentVolumeClaim Spec:

yaml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-database
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: fast-ssd
```

Kubernetes matches PVCs to PVs based on:

1. StorageClass name
2. Access modes compatibility
3. Capacity (PV must be >= PVC request)

Using in Pod:

yaml

```yaml
spec:
  containers:
  - name: postgres
    volumeMounts:
    - mountPath: /var/lib/postgresql/data
      name: database-storage
  volumes:
  - name: database-storage
    persistentVolumeClaim:
      claimName: pvc-database
```

StorageClass: Dynamic Provisioning

Creating PVs manually is tedious. **StorageClass** enables dynamic provisioning.

yaml

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-ssd
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp3
  iopsPerGB: "50"
  fsType: ext4
allowVolumeExpansion: true
volumeBindingMode: WaitForFirstConsumer
```

volumeBindingMode:

- **Immediate**: PV created immediately when PVC is created
- **WaitForFirstConsumer**: PV created when a pod using the PVC is scheduled (better for topology constraints)

Now when you create a PVC with `storageClassName: fast-ssd`, Kubernetes automatically:

1. Calls the AWS EBS provisioner
2. Creates an EBS volume
3. Creates a PV pointing to that EBS volume
4. Binds the PVC to the PV

Common PV/PVC Issues:

Issue 1: PVC stuck in Pending

- Check: `kubectl describe pvc pvc-database`
- Common causes:
  - No PV available matching the request
  - StorageClass provisioner not installed
  - Insufficient capacity in the StorageClass
  - Access mode not supported

Issue 2: Pod stuck in ContainerCreating

- Check: `kubectl describe pod database-pod`
- Often: "Unable to attach or mount volumes"
- Causes:
  - PVC doesn't exist
  - PV is bound to a different node (RWO) and pod scheduled on wrong node
  - Permissions issue (fsGroup, runAsUser mismatch)

Issue 3: Data persistence not working

- Check: Is volumeMount path correct?
- Check: Is the app actually writing to that path?
- For databases, check subPath usage:

yaml

```
volumeMounts:
- mountPath: /var/lib/postgresql/data
  name: database-storage
  subPath: postgres     # Creates /postgres subdirectory
```

The Great Data Loss of Node-3:

Pod-Database was running happily on Node-3 with a hostPath PV (local storage on the node). Then Node-3 crashed. Pod-Database was rescheduled to Node-4. But the data was on Node-3!

Lesson: **Never use hostPath for production data**. Use:

- **Cloud provider volumes**: AWS EBS, GCP Persistent Disk, Azure Disk
- **Network storage**: NFS, CephFS, GlusterFS
- **Distributed storage**: Rook/Ceph, Longhorn, OpenEBS

These allow pods to move between nodes while keeping data accessible.

Chapter 5: The Politics of Placement - Affinity, Taints, and Tolerations

Not all pods could live anywhere. Some were picky. Some nodes were exclusive. This created Clusterville's most complex social dynamics.

Node Affinity: "I Want to Live Near..."

Pod-GPU-Trainer needed GPUs for machine learning. He couldn't just live anywhere. He used **nodeSelector** (the simple way) or **nodeAffinity** (the sophisticated way).

Simple nodeSelector:

yaml

```yaml
spec:
  nodeSelector:
    gpu: "true"
    instance-type: "g4dn.xlarge"
```

This is inflexible - either a node matches ALL labels or the pod won't schedule.

Advanced nodeAffinity:

yaml

```yaml
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: gpu
            operator: In
            values:
            - "true"
          - key: gpu-type
            operator: In
            values:
            - nvidia-tesla-v100
            - nvidia-tesla-a100
      preferredDuringSchedulingIgnore
```