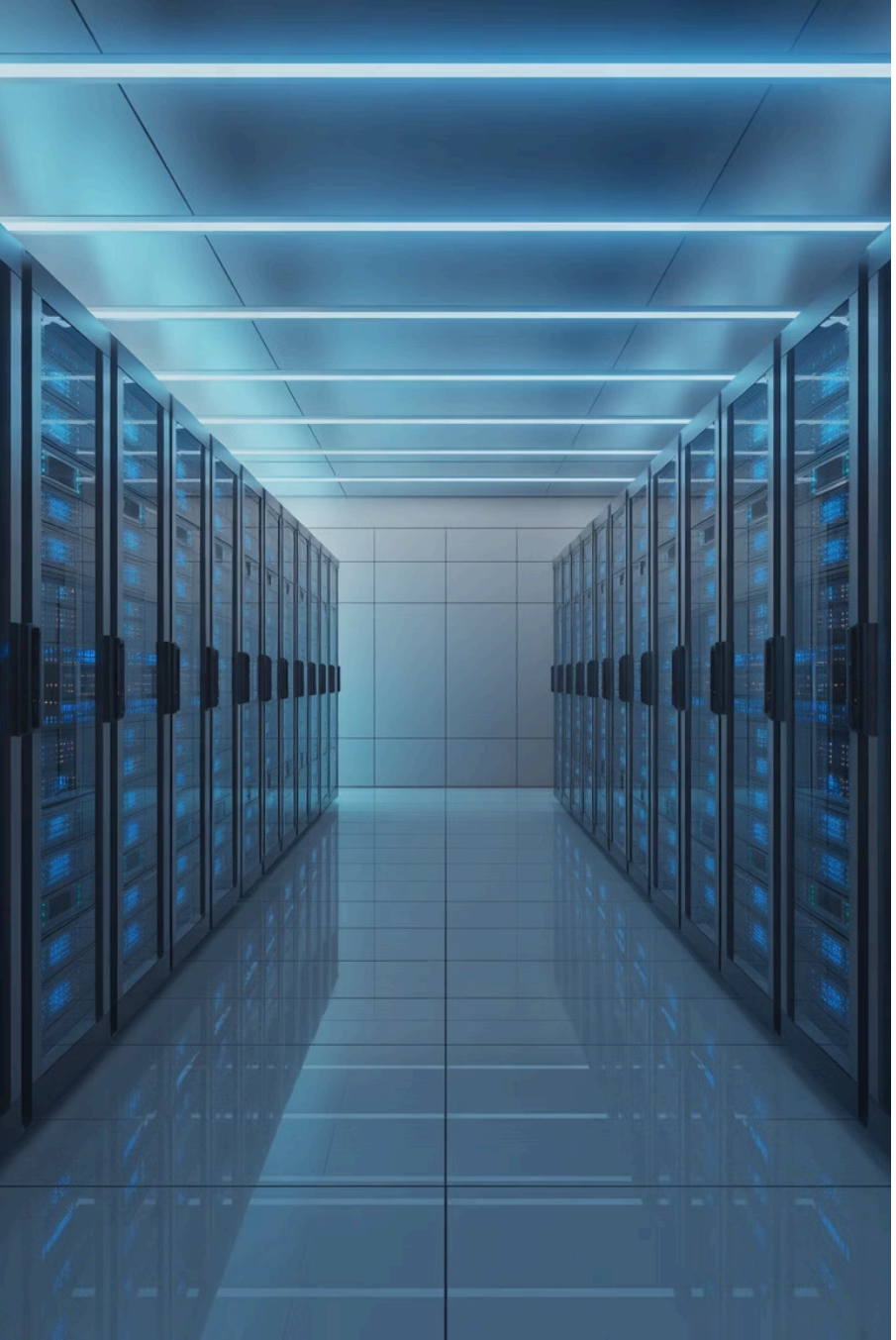


Kubernetes Core Concepts: A Practical Refresher

A hands-on guide for DevOps engineers at TechFlow Solutions

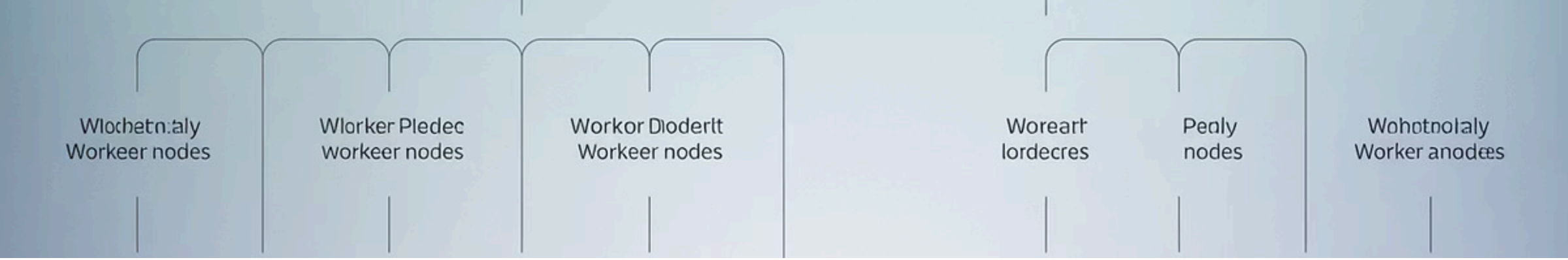
Meet TechFlow Solutions

Our Scenario

TechFlow Solutions is a mid-sized e-commerce company running a microservices platform on Kubernetes. Throughout this presentation, we'll follow their DevOps team as they manage their production clusters.

Their platform includes:

- Customer-facing web applications
- Payment processing services
- Inventory management systems
- Analytics and reporting tools



Understanding Cluster Architecture

At TechFlow, the Kubernetes cluster is divided into two main components: the Control Plane (the brain) and Worker Nodes (the muscle). Think of it like a restaurant kitchen—the Control Plane is the head chef coordinating everything, while Worker Nodes are the line cooks actually preparing dishes.

The Control Plane makes all scheduling decisions, maintains cluster state, and responds to events. Worker Nodes run the actual application containers and report back their status.

Control Plane: The Command Center

kube-apiserver

The front door for all cluster operations. When TechFlow's engineers run kubectl commands, they're talking to this component.

etcd

The cluster's memory—stores everything about desired state, current state, and configuration data in a distributed key-value store.

controller-manager

Continuously watches the cluster and fixes drift. If TechFlow's payment service should have 3 replicas but only has 2, this fixes it.

scheduler

Decides which node gets which pod based on resource requirements, constraints, and availability.

A Day in the Life: Control Plane in Action

1

Engineer deploys new app version

Sarah at TechFlow runs: `kubectl apply -f payment-service-v2.yaml`

2

API server validates and stores

The request is authenticated, validated against RBAC policies, and written to etcd.

3

Controller manager detects change

The Deployment controller notices the new desired state and creates a ReplicaSet.

4

Scheduler assigns pods

New pods are assigned to nodes with sufficient CPU and memory for the payment service.

Worker Nodes: Where Applications Live

Node Components

kubelet: The node agent that ensures containers are running as specified. At TechFlow, if a payment container crashes, kubelet restarts it automatically based on the restart policy.

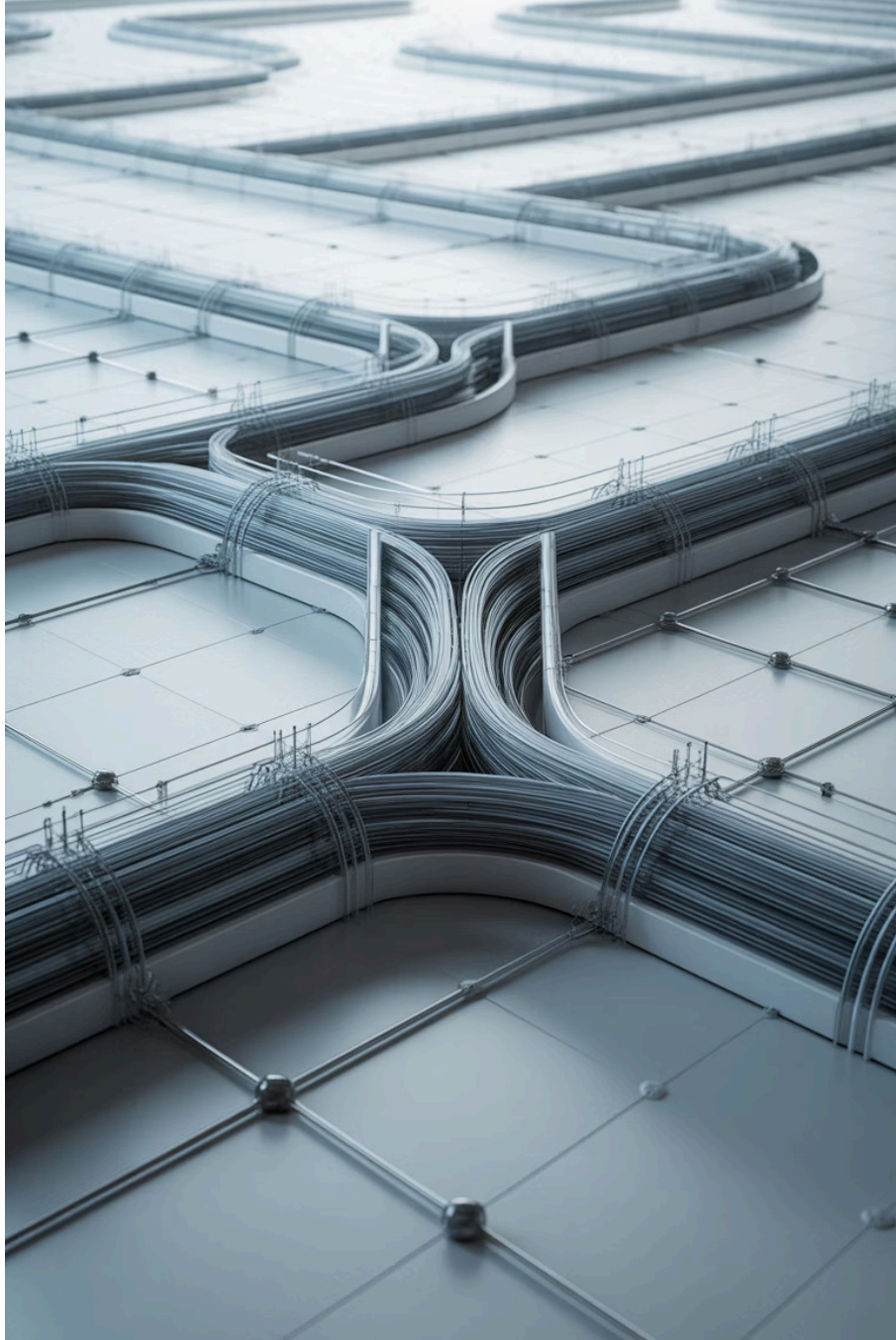
kube-proxy: Manages networking rules so that requests to `payment-service.prod.svc.cluster.local` reach the right pod, even as pods are created and destroyed.

Each worker node runs these components plus the container runtime (containerd or Docker) that actually executes containers.

Container Network Interface (CNI)

TechFlow uses Calico as their CNI plugin. The CNI is responsible for assigning IP addresses to pods and enabling pod-to-pod communication across nodes. Without it, a frontend pod on Node 1 couldn't talk to a database pod on Node 2.

Popular CNI options include Calico (network policies + routing), Cilium (eBPF-based networking with advanced security), Flannel (simple overlay network), and Weave (automatic mesh networking). The choice depends on your security requirements and network complexity. Cilium leverages eBPF for high-performance networking, advanced security policies, and deep observability into network traffic.



Anatomy of a Kubernetes Manifest

YAML Structure Basics

Every Kubernetes resource follows the same structure:

- **apiVersion:** Which API group (apps/v1, v1, batch/v1)
- **kind:** Resource type (Pod, Deployment, Service)
- **metadata:** Name, namespace, labels, annotations
- **spec:** Desired state specification

TechFlow stores all manifests in Git for version control and audit trails.

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: payment-service
 namespace: production
spec:
 replicas: 3
 selector:
 matchLabels:
 app: payment
 template:
 metadata:
 labels:
 app: payment
 spec:
 containers:
 - name: payment
 image: techflow/payment:v2
 ports:
 - containerPort: 8080
```

Namespaces: Organizing the Chaos

production

Customer-facing services with strict resource quotas and network policies. Payment and checkout services live here.

staging

Pre-production testing environment. Sarah's team tests new releases here before promoting to production.

development

Experimental workspace for developers. Lower resource limits and relaxed policies for rapid iteration.

monitoring

Observability stack isolated from application workloads. Prometheus, Grafana, and log aggregation tools.

Namespaces provide logical isolation, enabling RBAC policies, resource quotas, and network segmentation without needing separate clusters.

Pods: The Smallest Deployable Unit

A pod is one or more containers that share networking and storage. At TechFlow, most pods contain a single container, but their logging sidecar pattern uses two containers per pod—one for the application and one for log shipping.

Restart Policies: Always (default for long-running services), OnFailure (for Jobs), or Never. TechFlow's payment service uses "Always" so crashed containers restart automatically.

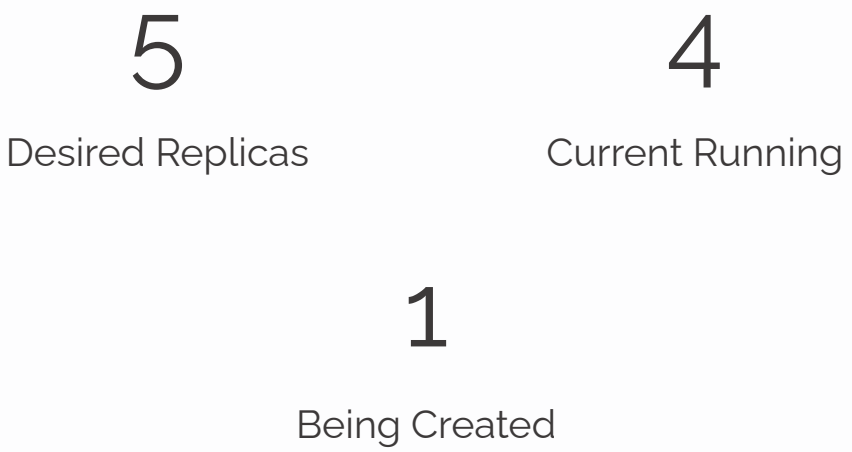
Pods are ephemeral—they're not resurrected after failure. Instead, controllers like Deployments create new pods with new IPs.

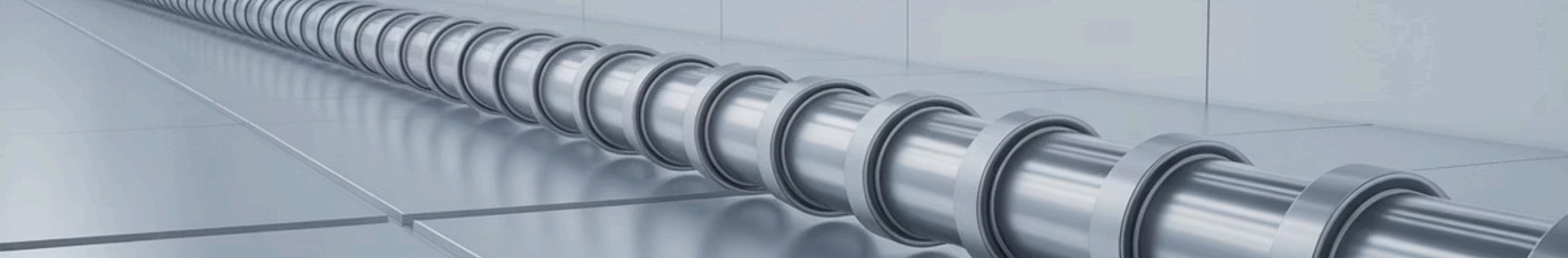
ReplicaSets: Maintaining Desired Count

The Pod Babysitter

ReplicaSets ensure a specified number of pod replicas are running at all times. If TechFlow's payment service should have 5 replicas and one pod crashes, the ReplicaSet immediately creates a replacement.

You rarely create ReplicaSets directly—Deployments manage them for you, creating new ReplicaSets during rollouts while scaling down old ones.





Deployments: Rolling Updates Made Easy

Deployments are TechFlow's primary way to run stateless applications. They provide declarative updates, automated rollouts, and easy rollbacks. When Sarah updates the payment service to v2.1, the Deployment gradually replaces old pods with new ones, ensuring zero downtime.

Key features include versioning (revision history), rollout strategies (RollingUpdate or Recreate), and rollback capabilities. If v2.1 has bugs, Sarah can roll back to v2.0 with a single command: kubectl rollout undo deployment/payment-service

DaemonSets: One Pod Per Node

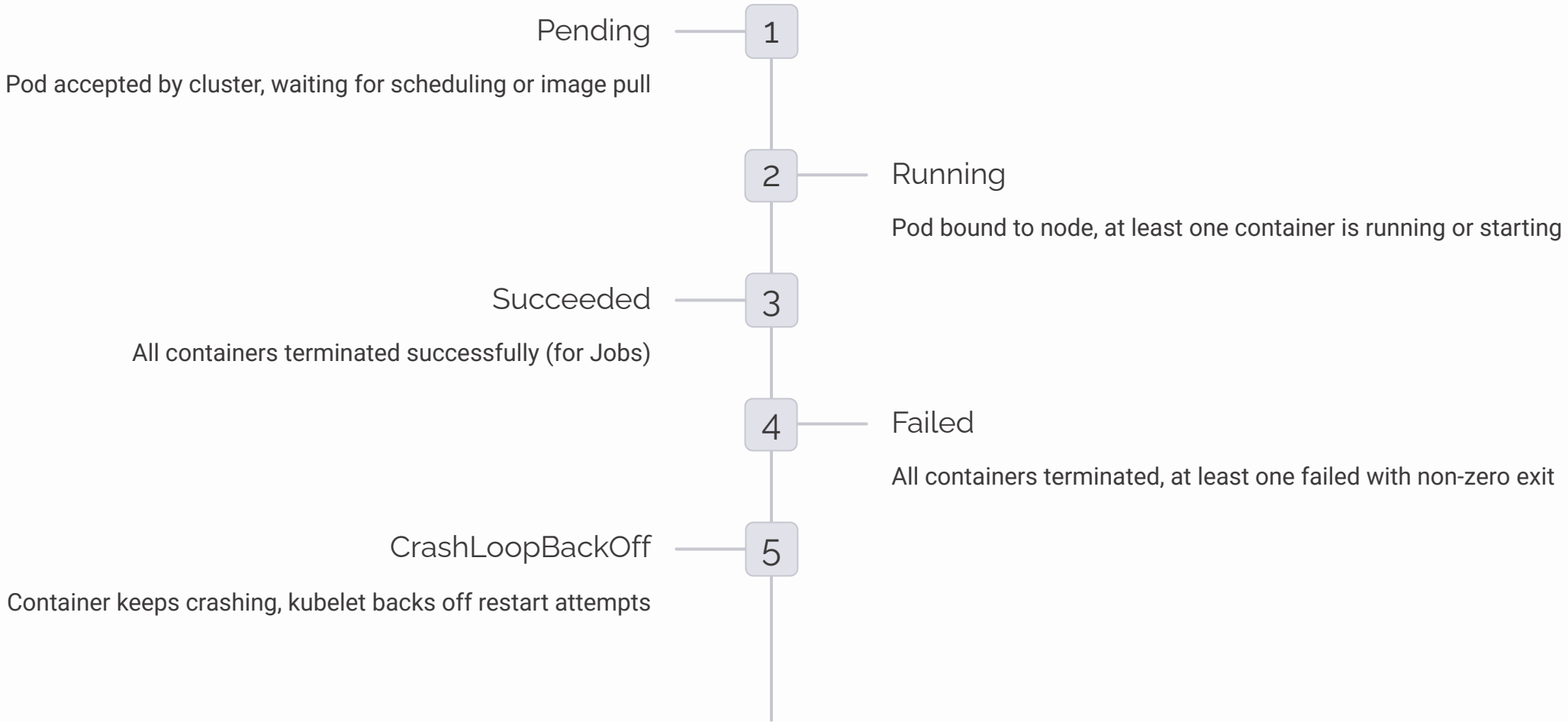
System-Level Services

DaemonSets ensure exactly one pod runs on every node (or a subset of nodes). TechFlow uses DaemonSets for:

- **Log collectors:** Fluentd on every node gathering application logs
- **Monitoring agents:** Node exporters for Prometheus metrics
- **Security agents:** Host-based intrusion detection

When TechFlow adds a new node, the DaemonSet automatically schedules its pods there. No manual intervention needed.

Pod Lifecycle States



Manual Scheduling with nodeName

Bypassing the Scheduler

Occasionally, TechFlow needs to pin a pod to a specific node for troubleshooting or specialized hardware access. The `nodeName` field forces a pod onto a particular node, bypassing the scheduler entirely.

```
spec:
  nodeName: node-gpu-01
  containers:
  - name: ml-training
    image: techflow/ml:v1
```

Use sparingly—this removes scheduler intelligence and can cause resource contention.

Taints and Tolerations: Workload Isolation

Taints repel pods from nodes unless pods have matching tolerations. TechFlow taints their GPU nodes with `gpu=true:NoSchedule` so only ML workloads (which have the toleration) can run there. Regular application pods stay away.

Common use cases: dedicated nodes for specific teams, isolating problematic workloads, or reserving nodes for high-priority services. Think of taints as "No Trespassing" signs that only certain pods can ignore.

NodeSelector and NodeAffinity

NodeSelector (Simple)

Hard requirement using label matching. TechFlow uses `nodeSelector: {disktype: ssd}` to ensure database pods only run on SSD-backed nodes.

```
spec:
  nodeSelector:
    disktype: ssd
```

NodeAffinity (Advanced)

Flexible rules with required and preferred conditions. Can express "prefer nodes in us-west-2a, but us-west-2b is acceptable" for high-availability deployments.

```
affinity:
 nodeAffinity:
 preferredDuringScheduling...
```

Rollouts and Rollbacks

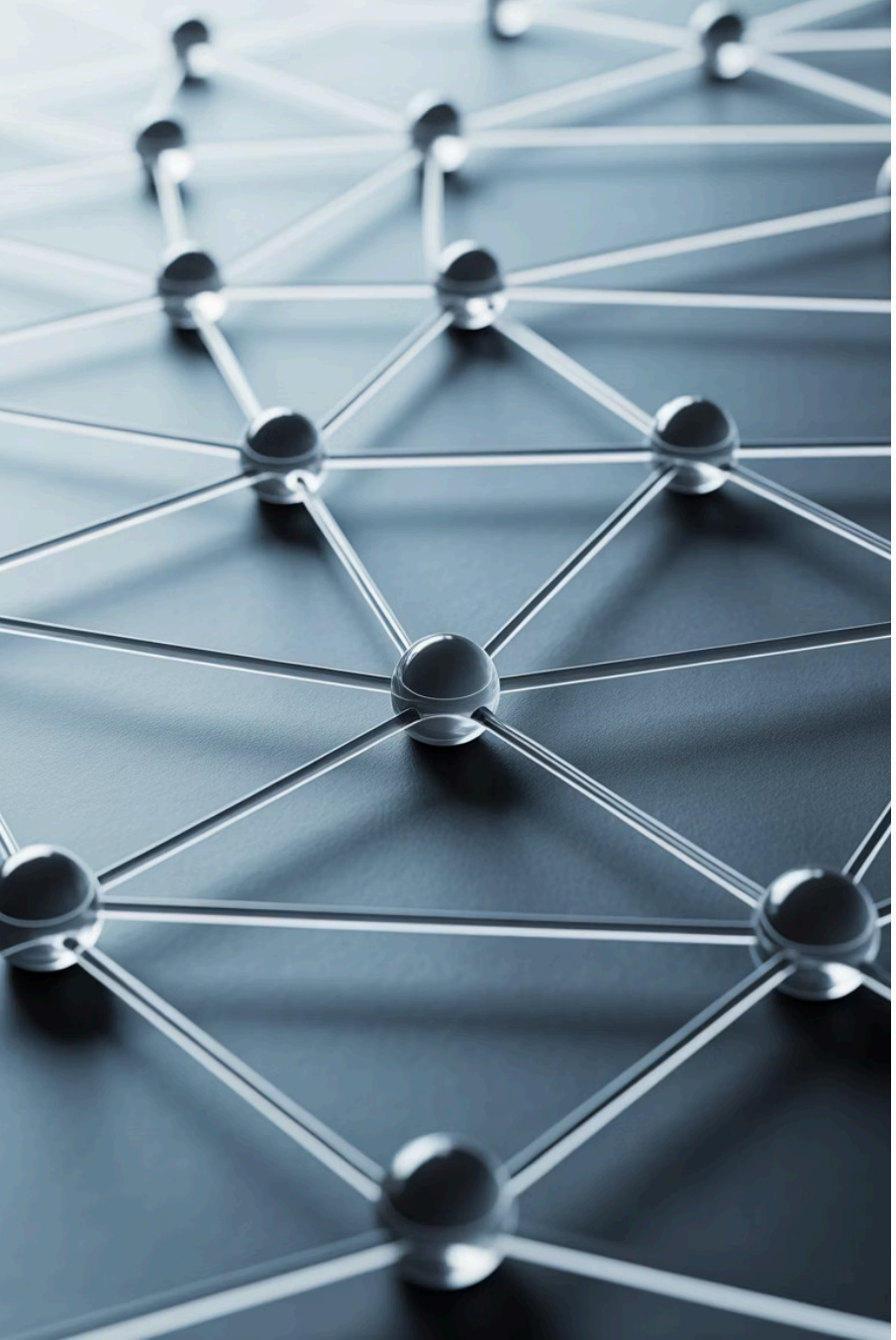
Version Management

When TechFlow deploys payment-service v2.2, Kubernetes performs a rolling update by default. It creates new pods with v2.2 while gradually terminating v2.1 pods, ensuring service continuity.

Key commands:

- `kubectl rollout status deployment/payment-service` - Watch progress
- `kubectl rollout history deployment/payment-service` - View revision history
- `kubectl rollout undo deployment/payment-service` - Instant rollback

Rollback is nearly instantaneous because previous ReplicaSets are retained with their pod templates ready.



The Kubernetes Networking Model

Kubernetes networking is built on three fundamental principles that TechFlow relies on daily:

Every pod gets its own IP address

No NAT between pods. The frontend pod at 10.244.1.5 talks directly to the database pod at 10.244.2.8 across nodes.

Pods can communicate without NAT

Flat network space enables simple service discovery and eliminates port mapping complexity.

Nodes can communicate with all pods

Kubelet on any node can reach any pod for health checks and management operations.

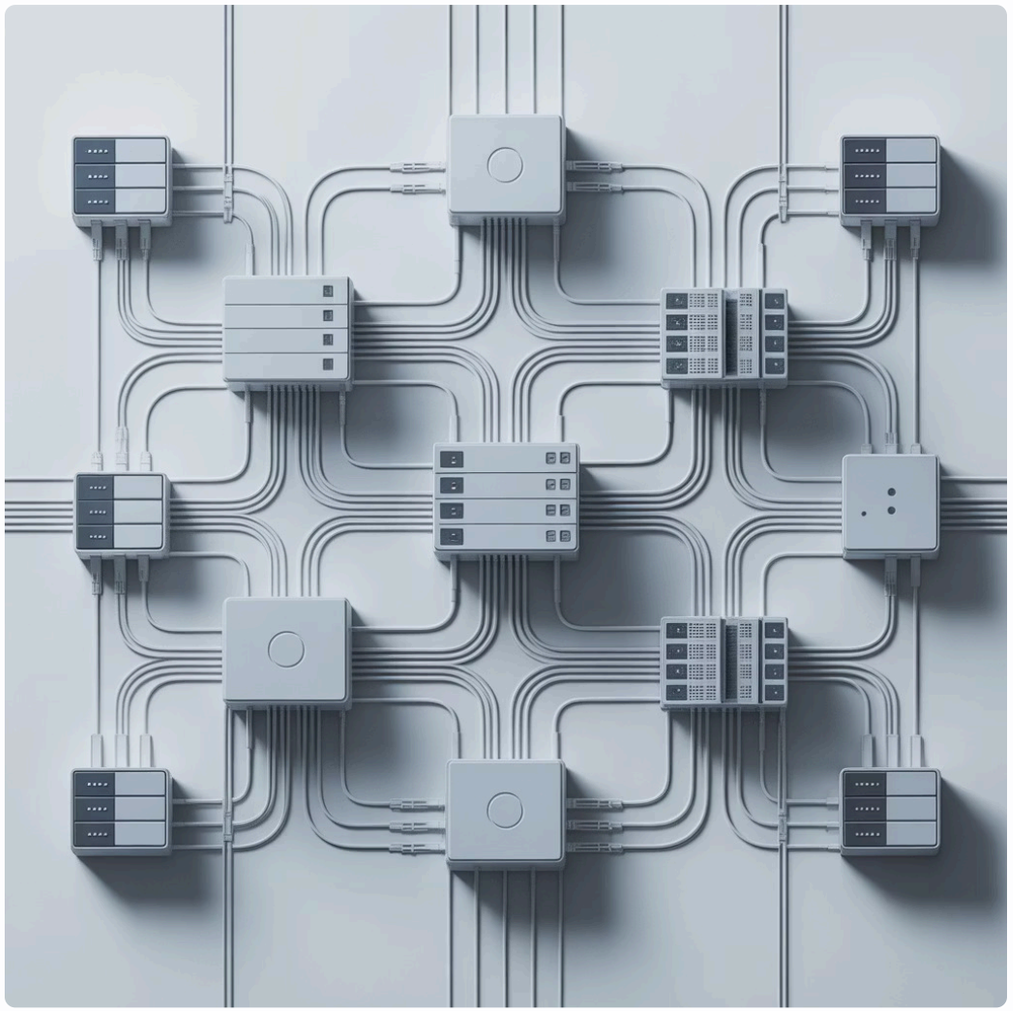
Service Types: ClusterIP

Internal-Only Access

ClusterIP (the default) exposes a service on an internal cluster IP. TechFlow's backend database service uses ClusterIP because it should never be accessible from outside the cluster.

```
apiVersion: v1
kind: Service
metadata:
  name: postgres-db
spec:
  type: ClusterIP
  ports:
  - port: 5432
  selector:
    app: postgres
```

Applications access it via DNS: `postgres-db.production.svc.cluster.local`



The service IP remains stable even as backend pods are created and destroyed, providing a reliable endpoint for other services.

Service Types: NodePort and LoadBalancer

NodePort

Exposes the service on each node's IP at a static port (30000-32767). TechFlow uses this for internal admin tools accessible via any node's IP address.

```
type: NodePort
ports:
- port: 8080
  nodePort: 30080
```

Access via: `http://<node-ip>:30080`

LoadBalancer

Provisions an external load balancer from the cloud provider. TechFlow's customer-facing web application uses this type, which creates an AWS ELB automatically.

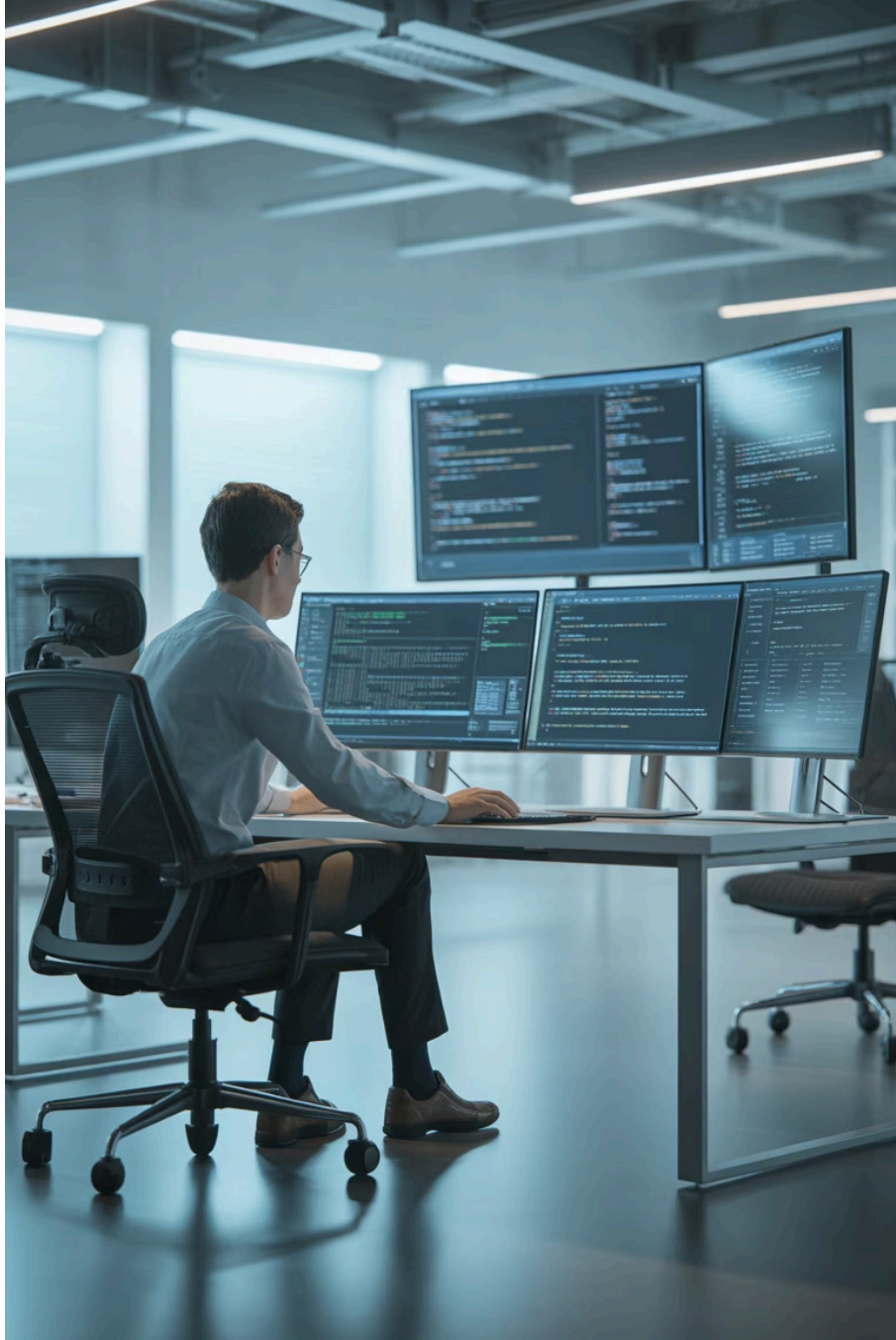
```
type: LoadBalancer
ports:
- port: 443
  targetPort: 8443
```

AWS assigns a public IP or hostname for external traffic.

Troubleshooting: The Systematic Approach

When TechFlow's payment service starts failing at 3 AM, Sarah follows a methodical troubleshooting workflow. Panic doesn't fix clusters—systematic investigation does.

The key is working from high-level cluster state down to container-level details, gathering evidence at each layer before jumping to conclusions. Most issues reveal themselves within the first three steps.



Step-by-Step Troubleshooting Process

Describe the object

`kubectl describe pod payment-service-abc123` - Shows events, conditions, and current state

View logs

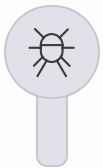
`kubectl logs payment-service-abc123` - Check application output. Add `--previous` for crashed containers.

Exec into container

`kubectl exec -it payment-service-abc123 -- sh` - Inspect filesystem, test connectivity, check configs

Check cluster events

`kubectl get events --sort-by='.lastTimestamp'` - Recent cluster-wide activity and errors



Debug with ephemeral containers

`kubectl debug payment-service-abc123 -it --image=busybox` - Add debugging tools without modifying pod

Common Failure Scenarios

CrashLoopBackOff

Application crashes immediately on startup. Usually configuration errors, missing dependencies, or failed health checks. Check logs first.

ImagePullBackOff

Can't pull container image. Wrong image name, missing registry credentials, or network issues. Verify image exists and imagePullSecrets are configured.

Node NotReady

Node lost communication with control plane. Check kubelet logs on the node, verify network connectivity, and inspect resource pressure.

Service DNS Failures

Pods can't resolve service names. CoreDNS might be unhealthy, or network policies blocking DNS traffic on port 53.

Real-World Debug Example

The Mystery of the Failing Payment Service

TechFlow's payment service started returning 500 errors. Sarah's investigation:

1. `kubectl get pods` - All pods Running, not obvious
2. `kubectl describe pod payment-xyz` - Noticed high memory usage warning
3. `kubectl logs payment-xyz` - Found "OutOfMemory" errors in Java heap
4. `kubectl top pod payment-xyz` - Confirmed memory at 95% of limit

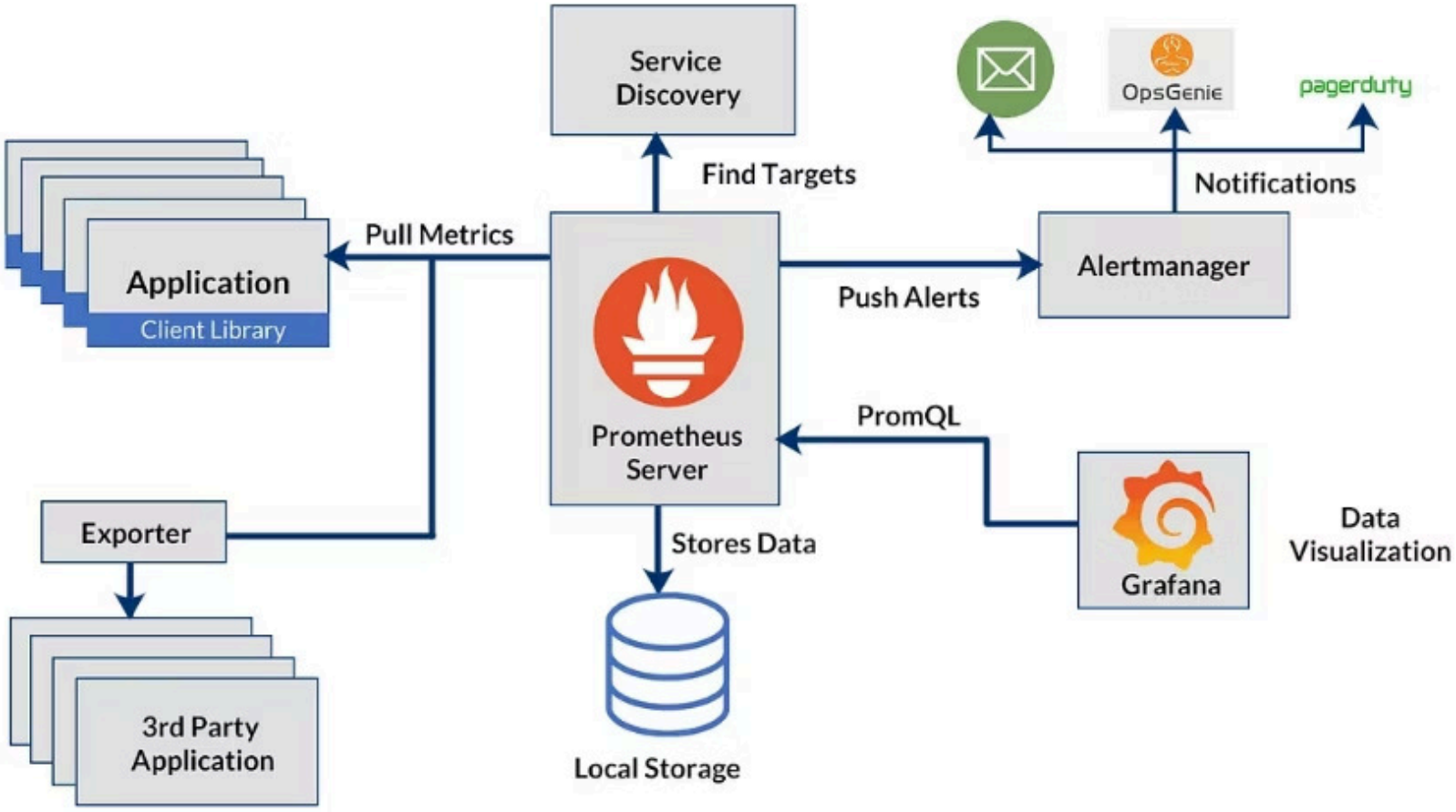
Solution: Increased memory limit from 512Mi to 1Gi in deployment spec. Problem solved.

Observability: Prometheus Architecture

TechFlow uses Prometheus for metrics collection across their Kubernetes clusters. Prometheus is a pull-based system—it scrapes metrics from targets at regular intervals rather than receiving pushed data.

Key components in TechFlow's setup:

- **node-exporter:** DaemonSet collecting hardware and OS metrics from every node (CPU, memory, disk I/O)
- **kube-state-metrics:** Exposes cluster-level metrics about Kubernetes objects (deployment replica counts, pod status)
- **Application metrics:** Payment service exposes custom business metrics at `/metrics` endpoint


Service Discovery
Find Targets
OpsGenie
pagerduty
Notifications
Application
Client Library
Pull Metrics
Prometheus Server
Push Alerts
Alertmanager
PromQL
Exporter
Stores Data
Grafana
Data Visualization
3rd Party Application
Local Storage

Grafana: Visualizing Cluster Health

Dashboard-Driven Insights

Grafana connects to Prometheus and transforms raw metrics into actionable dashboards. TechFlow's operations team monitors several key dashboards:

- Cluster overview (node status, pod count, resource usage)
- Namespace breakdowns (CPU/memory per namespace)
- Application performance (request latency, error rates)
- Cost allocation (resource consumption by team)

Alerts are configured in Grafana to notify the team when payment service latency exceeds 500ms or when any node's disk usage exceeds 85%.

Monitoring in Action: A TechFlow Story

1

10:00 AM - Normal operations

Grafana dashboard shows healthy metrics across all services

2

10:15 AM - Alert triggered

Prometheus detects payment service pod CPU usage at 95% for 5 minutes

3

10:17 AM - Team investigates

Sarah checks Grafana, sees request rate doubled due to marketing campaign

4

10:20 AM - Horizontal scaling

Team scales payment deployment from 3 to 6 replicas: `kubectl scale deployment payment-service --replicas=6`

5

10:25 AM - Resolution

CPU usage drops to 60% per pod, request latency returns to normal. Alert clears automatically.

Key Takeaways for Production Success

1

Master the fundamentals

Understanding control plane components, pod lifecycle, and networking model enables effective troubleshooting when things go wrong at 3 AM.

2

Embrace declarative configuration

Store all manifests in Git. Use Deployments over manual pod creation. Let Kubernetes maintain desired state while you focus on defining what you want.

3

Build observability from day one

Prometheus and Grafana aren't optional—they're essential for understanding cluster behavior, capacity planning, and rapid incident response.

4

Practice systematic troubleshooting

Follow the five-step diagnostic process. Gather evidence methodically rather than making assumptions. Most issues are configuration problems, not Kubernetes bugs.

TechFlow's DevOps team succeeds because they treat Kubernetes as a platform to master, not magic to fear. With these core concepts refreshed, you're equipped to do the same.