Service Provider (CSP) Managed Kubernetes Questions

In Kubernetes, what are the different methods for creating pods, and when would you use each method?

Kubernetes offers several methods for creating pods, each suited for different use cases.

- **Direct Pod creation** - creating standalone pods using `kubectl run` or pod manifest:
  - Simplest method but rarely used in production.
  - No built-in resilience (pod dies, it's gone).
  - Useful for debugging, testing, or one-off jobs.
  - Example: `kubectl run nginx --image=nginx:latest`.
  - **Use when**: running quick tests, debugging issues, executing one-time tasks that don't need persistence.
- **ReplicaSet** - ensures specified number of pod replicas running:
  - Maintains desired replica count replacing failed pods.
  - Rarely created directly (usually via Deployment).
  - Declarative definition specifying pod template and replica count.
  - **Use when**: you need basic replication without rolling updates (uncommon—Deployments are preferred).
- **Deployment** (most common for stateless applications) - higher-level abstraction managing ReplicaSets:
  - Declarative updates enabling rolling updates and rollbacks.
  - Scaling replicas up or down.
  - Maintains revision history for rollbacks.
  - Health checks ensuring new pods ready before terminating old ones.
  - **Use when**: deploying stateless applications (web servers, APIs, microservices), you need rolling updates without downtime, scaling is required, and managing long-running applications.
- **StatefulSet** - for stateful applications requiring stable identity:
  - Ordered, graceful deployment and scaling.
  - Stable network identifiers (predictable pod names).
  - Persistent storage that follows pod lifecycle.

    - Ordered rolling updates.
    - **Use when**: deploying databases (MySQL, PostgreSQL), distributed systems (Kafka, Elasticsearch, ZooKeeper), applications requiring stable network identity, and persistent storage that must survive pod rescheduling.
- **DaemonSet** - ensures pod runs on all (or subset) nodes:
    - One pod per node automatically.
    - Useful for node-level operations.
    - New nodes automatically get pod.
    - **Use when**: deploying logging agents (Fluentd, Filebeat), monitoring agents (Prometheus node exporter), network plugins, or storage daemons.
- **Job** - creates pods that run to completion:
    - Executes batch workload then terminates.
    - Retries on failure up to specified limit.
    - Tracks completion.
    - **Use when**: batch processing, ETL jobs, database migrations, or one-time tasks.
- **CronJob** - creates Jobs on schedule:
    - Like cron in Kubernetes.
    - Scheduled execution (daily backups, periodic cleanup).
    - Manages Job history.
    - **Use when**: scheduled tasks (backups, report generation), periodic maintenance, or time-based automation.
- **Helm Charts/Operators** - package managers and controllers:
    - Helm charts bundle related resources.
    - Operators extend Kubernetes with custom logic.
    - Manage complex applications declaratively.
    - **Use when**: deploying complex applications with many components, managing application lifecycle (backups, upgrades), or packaging applications for distribution.

Describe the differences between Imperative and Declarative pod creation in Kubernetes.

Imperative approach - telling Kubernetes *how* to do something step-by-step:

- **Commands**: `kubectl run nginx --image=nginx`, `kubectl create deployment web --image=nginx --replicas=3`, `kubectl scale deployment web --replicas=5`, `kubectl set image deployment/web nginx=nginx:1.21`.

- **Characteristics**: direct commands executed immediately, no manifest files (ephemeral), difficult to track or version control, harder to reproduce exact state, suitable for quick testing or debugging, and doesn't represent infrastructure as code.

Example:

```
# Create deployment imperatively
kubectl create deployment nginx --image=nginx:1.19
kubectl expose deployment nginx --port=80 --type=LoadBalancer
kubectl scale deployment nginx --replicas=5
```

Declarative approach - describing *what* desired state should be, Kubernetes figures out how:

- **Manifests**: YAML or JSON files describing desired state, apply with `kubectl apply -f manifest.yaml`, Kubernetes reconciles current state to match desired state.
- **Characteristics**: infrastructure as code (versioned, reviewed), reproducible and auditable, easier to manage at scale, supports GitOps workflows, idempotent (apply multiple times = same result), and represents source of truth.

Example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 5
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.19
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  type: LoadBalancer
  ports:
  - port: 80
```

```
  selector:
    app: nginx
```

Apply with: `kubectl apply -f nginx-deployment.yaml`

Key differences:

- **State management** - Imperative: no state file, commands create/modify directly; Declarative: manifest represents desired state, Kubernetes reconciles.
- **Version control** - Imperative: commands not version controlled; Declarative: YAML files in Git, full history.
- **Reproducibility** - Imperative: difficult to recreate exact environment; Declarative: manifest precisely recreates state.
- **Collaboration** - Imperative: hard to share/review changes; Declarative: code review process, pull requests.
- **Complexity** - Imperative: simple for quick tasks; Declarative: better for complex, production environments.
- **Auditing** - Imperative: limited audit trail; Declarative: Git history provides complete audit trail.

In production, use declarative approach because:

- Infrastructure as code enables version control and review.
- Reproducible deployments across environments.
- GitOps workflows with automated deployments.
- Easier disaster recovery (redeploy from manifests).
- Compliance and audit requirements.
- Team collaboration through pull requests.

Imperative commands useful for: quick debugging and testing, exploring Kubernetes features, emergency fixes (though should be formalized in manifests after), and learning Kubernetes.

Best practice: Use declarative manifests for all production resources, store manifests in Git, use imperative commands only for debugging/testing, document any imperative changes and update manifests accordingly, and implement GitOps with automated manifest application.

How do you ensure that security configurations and policies are consistently applied regardless of the method used for pod creation?

Consistent security enforcement requires multiple layers of controls that apply regardless of pod creation method.

Admission Controllers - intercept requests before persistence:

- **PodSecurityPolicy (deprecated)** or **Pod Security Standards** - enforce security requirements: run as non-root user, drop dangerous capabilities, use read-only root filesystem, disallow privilege escalation, and restrict volume types.
- **OPA Gatekeeper** - policy-as-code enforcement: custom policies in Rego language, enforce naming conventions, require specific labels/annotations, mandate resource limits, and block privileged containers.

Example Gatekeeper policy:

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sRequiredLabels
metadata:
  name: require-security-labels
spec:
  match:
    kinds:
    - apiGroups: [""]
      kinds: ["Pod"]
  parameters:
    labels:
    - key: "security-owner"
    - key: "security-tier"
```

- **Kyverno** - Kubernetes-native policy engine: policies written in YAML (easier than OPA), validate, mutate, or generate resources, enforce security best practices automatically.

Example Kyverno policy:

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: disallow-privileged
spec:
  validationFailureAction: enforce
  rules:
  - name: check-privileged
    match:
      resources:
        kinds:
        - Pod
    validate:
      message: "Privileged containers are not allowed"
      pattern:
        spec:
          containers:
          - =(securityContext):
```

```
              =(privileged): false
```

Network Policies - control pod-to-pod traffic regardless of how pods created: default deny ingress/egress, explicitly allow required communications, and enforce micro-segmentation.

Service Mesh (Istio, Linkerd) - mTLS between services automatically, policy enforcement at service layer, uniform security regardless of pod creation method.

Image Security:

- **Admission webhook validating images** - only allow images from approved registries: webhook checks image registry on pod creation, blocks unauthorized registries.
- **Image scanning integrated with admission** - scan images before pod creation: integrate Trivy, Clair, Anchore, block pods with critical vulnerabilities, and enforce image signature verification.

RBAC - restrict who can create pods: principle of least privilege, separate permissions for developers vs. operators, require security review for production pod creation.

Resource Quotas and Limit Ranges:

- **Quotas** prevent resource exhaustion attacks.
- **LimitRanges** enforce minimum/maximum resource requests preventing extremely privileged pods.

Security Context enforcement: Mutating webhooks automatically inject security contexts if missing, ensuring baseline security even if developer forgot.

Example mutating webhook:

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    securitycontext.webhook/inject: "true"
# Webhook automatically adds:
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 10001
    fsGroup: 10001
  containers:
  - name: app
    securityContext:
      allowPrivilegeEscalation: false
      readOnlyRootFilesystem: true
      capabilities:
        drop:
```

```
        - ALL
```

CI/CD integration: Scan manifests in pipeline before deployment, policy validation as pipeline gate, automated security testing, and deployment approval workflows.

Monitoring and enforcement: Continuous compliance scanning detecting drift, alerting on policy violations, automated remediation of non-compliant pods, and audit logging of all pod creations.

Example comprehensive enforcement:

```
# Enable Pod Security Standards
kubectl label namespace production pod-security.kubernetes.io/enforce=restricted

# Deploy Gatekeeper
helm install gatekeeper gatekeeper/gatekeeper

# Apply security policies
kubectl apply -f security-policies/

# Configure network policies
kubectl apply -f network-policies/

# Integrate image scanning
# In admission controller configuration
```

Testing security:

- Attempt creating non-compliant pods: `kubectl run test --image=nginx --privileged=true` (should be blocked).
- Verify security context applied automatically.
- Confirm network policies block unauthorized traffic.
- Validate image scanning blocks vulnerable images.

This multi-layered approach ensures security regardless of whether pods created imperatively, declaratively, via Helm, or Operators—admission controllers and policies enforce consistent security at the API server level.

What role does container image scanning play in securing pods created in a Kubernetes cluster?

Container image scanning is critical for identifying vulnerabilities and misconfigurations before pods run.

Role and importance:

- **Vulnerability detection** - identifies known CVEs in base images and application dependencies:
  - OS packages (outdated openssl, vulnerable kernel).
  - Application libraries (log4j, older npm packages).
  - Programming language runtimes.
  - Provides severity scores (critical, high, medium, low) and remediation guidance (upgrade package to version X).
- **Configuration issues** - detects insecure image configurations: running as root user, exposed secrets or credentials, insecure file permissions, and exposed ports.
- **Compliance** - ensures images meet organizational standards: approved base images only, required security labels, patch currency requirements, and licensing compliance.
- **Supply chain security** - validates image provenance: images from trusted registries, signed images verifying publisher, SBOM (Software Bill of Materials) tracking components, and detecting malicious images or tampering.

Integration points:

- **CI/CD pipeline scanning** - scan during image build: integrate scanner (Trivy, Grype, Clair, Anchore) in Dockerfile build stage, fail pipeline if critical vulnerabilities found, generate reports for tracking, and scan both base images and final application images.

Example GitLab CI:

```
stages:
  - build
  - scan
  - deploy

build:
  stage: build
  script:
    - docker build -t myapp:${CI_COMMIT_SHA} .
    - docker push myapp:${CI_COMMIT_SHA}

scan:
  stage: scan
  image: aquasec/trivy:latest
  script:
    - trivy image --severity HIGH,CRITICAL --exit-code 1 myapp:${CI_COMMIT_SHA}
  allow_failure: false

deploy:
  stage: deploy
  script:
    - kubectl set image deployment/myapp app=myapp:${CI_COMMIT_SHA}
  only:
    - master
```

- **Registry scanning** - continuous scanning in container registry: AWS ECR scan on push, Azure Container Registry integrated scanning, Google Artifact Registry vulnerability scanning, and Harbor with Trivy/Clair integration. Rescans periodically detecting newly disclosed CVEs affecting existing images.
- **Admission control scanning** - scan at pod creation time: admission webhook calls scanner before pod creation, blocks deployment if vulnerabilities exceed threshold, provides immediate feedback to developers.
  - Example admission webhook flow: Developer applies pod manifest → API server calls admission webhook → webhook queries image scanner → if critical vulnerabilities exist, webhook rejects pod → developer notified to fix vulnerabilities.
- **Runtime scanning** - ongoing monitoring of running containers: detects new vulnerabilities in running images, identifies runtime configuration issues, monitors for unexpected changes, and alerts on suspicious activity.

Scanning tools:

- **Trivy** (open source, comprehensive) - fast scanning, multiple formats (container images, filesystems, Git repos), high accuracy, integrates easily with CI/CD.
- **Grype** (open source, Anchore) - accurate vulnerability matching, SBOM support, good CI/CD integration.
- **Clair** (open source, by Quay) - static vulnerability analysis, API-driven, used by many registries.
- **Anchore Enterprise** - commercial, policy-based enforcement, detailed reporting, compliance frameworks.
- **Snyk** - developer-friendly, IDE integration, license scanning, fix recommendations.
- **Aqua Security, Prisma Cloud** - comprehensive platform including scanning, runtime protection, compliance.

Best practices:

- **Scan early and often** - scan in CI/CD before images reach production, rescan periodically (daily) for new CVEs, scan base images before building on them.
- **Establish severity thresholds** - block critical vulnerabilities, warn on high, track medium/low. Adjust based on risk tolerance.
- **Prioritize remediation** - fix exploitable vulnerabilities first, address vulnerabilities in internet-facing applications, consider CVSS score, exploitability, and asset criticality.
- **Use minimal base images** - Alpine Linux, distroless images have fewer packages = smaller attack surface, reduces vulnerability count.
- **Implement image signing** - sign images after successful scan, verify signatures at deployment, prevents tampering post-scan.
- **Track and report** - vulnerability dashboards showing trends, compliance metrics (% images passing scan), remediation tracking (time to fix).
- **Automate remediation** - automated base image updates, dependency updates via Dependabot/Renovate, rebuild images when patches available.

- **Integrate with governance** - images must pass scan before production, exceptions require security review and documentation, regular reviews ensuring compliance.

Example comprehensive scanning workflow: Build image → Scan in CI/CD (Trivy) → Push to registry if passed → Registry continuous scan (ECR) → Deploy to staging → Admission webhook verifies scan results → Deploy to production → Runtime monitoring (Falco) detecting anomalies → Periodic rescans identifying new CVEs → Automated alerts on new critical vulnerabilities → Remediation workflow triggered.

Image scanning transforms unknown security posture into managed risk, enabling informed decisions about deployment safety and providing audit trail of security validation.

Walk me through the process of creating a pod using Kubernetes YAML manifests and explain how you would apply security best practices.

I'll demonstrate creating a secure pod using declarative YAML with comprehensive security controls.

Step 1: Basic pod structure with security context:

```
apiVersion: v1
kind: Pod
metadata:
  name: secure-app
  namespace: production
  labels:
    app: secure-app
    tier: backend
    security-tier: high
  annotations:
    seccomp.security.alpha.kubernetes.io/pod: runtime/default
spec:
  # Security: Run as non-root user
  securityContext:
    runAsNonRoot: true
    runAsUser: 10001
    runAsGroup: 10001
    fsGroup: 10001
    seccompProfile:
      type: RuntimeDefault

  containers:
  - name: app
    image: myregistry.io/secure-app:v1.2.3
```

```
    imagePullPolicy: Always

    # Security: Container-level security context
    securityContext:
      allowPrivilegeEscalation: false
      readOnlyRootFilesystem: true
      capabilities:
        drop:
        - ALL
        add:
        - NET_BIND_SERVICE  # Only if needed for ports <1024
      runAsNonRoot: true
      runAsUser: 10001

    # Resource limits prevent DoS
    resources:
      requests:
        memory: "128Mi"
        cpu: "100m"
      limits:
        memory: "256Mi"
        cpu: "200m"

    # Application ports
    ports:
    - containerPort: 8080
      name: http
      protocol: TCP

    # Health checks
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
      initialDelaySeconds: 30
      periodSeconds: 10
    readinessProbe:
      httpGet:
        path: /ready
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 5

    # Environment variables from secrets/configmaps
    env:
    - name: DATABASE_URL
      valueFrom:
        secretKeyRef:
          name: app-secrets
          key: database-url
    - name: LOG_LEVEL
```

```
      valueFrom:
        configMapKeyRef:
          name: app-config
          key: log-level

    # Volumes for writable paths (since root filesystem readonly)
    volumeMounts:
    - name: tmp
      mountPath: /tmp
    - name: cache
      mountPath: /app/cache
    - name: secrets
      mountPath: /app/secrets
      readOnly: true

  # Security: Use specific service account, not default
  serviceAccountName: secure-app-sa
  automountServiceAccountToken: false  # Don't auto-mount if not needed

  # Volumes
  volumes:
  - name: tmp
    emptyDir: {}
  - name: cache
    emptyDir: {}
  - name: secrets
    secret:
      secretName: app-tls-cert
      defaultMode: 0400  # Read-only for owner only

  # Security: Image pull secret
  imagePullSecrets:
  - name: registry-credentials

  # Security: Host namespaces disabled (defaults, shown explicitly)
  hostNetwork: false
  hostPID: false
  hostIPC: false

  # Node affinity/tolerations if needed
  nodeSelector:
    workload-type: application
```

Step 2: Supporting resources (secrets, service account):

```
---
# Service account with minimal permissions
apiVersion: v1
kind: ServiceAccount
metadata:
```

```
  name: secure-app-sa
  namespace: production
automountServiceAccountToken: false

---
# Secret for database credentials
apiVersion: v1
kind: Secret
metadata:
  name: app-secrets
  namespace: production
type: Opaque
data:
  database-url: <base64-encoded-value>

---
# ConfigMap for non-sensitive configuration
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
  namespace: production
data:
  log-level: "info"
  max-connections: "100"

---
# Network Policy restricting traffic
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: secure-app-netpol
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: secure-app
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend
    ports:
    - protocol: TCP
      port: 8080
  egress:
  - to:
    - podSelector:
```

```
        matchLabels:
          app: database
    ports:
    - protocol: TCP
      port: 5432
  - to:  # Allow DNS
    - namespaceSelector:
        matchLabels:
          name: kube-system
    ports:
    - protocol: UDP
      port: 53
```

Security best practices explained:

1. **Non-root execution**: `runAsNonRoot: true` and `runAsUser: 10001` ensure container doesn't run as root, preventing privilege escalation.
2. **Read-only root filesystem**: `readOnlyRootFilesystem: true` prevents malware/attacker from modifying container filesystem, use emptyDir volumes for writable paths.
3. **Drop all capabilities**: `drop: ALL` removes all Linux capabilities, add back only specific ones needed (e.g., `NET_BIND_SERVICE` for ports <1024).
4. **No privilege escalation**: `allowPrivilegeEscalation: false` prevents processes from gaining more privileges.
5. **Seccomp profile**: `RuntimeDefault` applies default seccomp profile limiting syscalls.
6. **Resource limits**: Prevents resource exhaustion DoS attacks.
7. **Specific service account**: Don't use default service account, create dedicated one with minimal RBAC.
8. **Secrets management**: Use Kubernetes Secrets (or external secret managers like Vault) for sensitive data, never hardcode.
9. **Network policies**: Implement zero-trust micro-segmentation allowing only necessary traffic.
10. **Image best practices**: Use specific image tags (not `latest`), scan images for vulnerabilities, use private registry with authentication.
11. **Health checks**: Liveness/readiness probes ensure application health, Kubernetes restarts unhealthy pods.
12. **No host namespaces**: `hostNetwork/PID/IPC: false` isolates pod from host.

Step 3: Apply with validation:

```
# Validate syntax
kubectl apply --dry-run=client -f secure-pod.yaml

# Validate against cluster (without creating)
kubectl apply --dry-run=server -f secure-pod.yaml
```

```
# Apply to cluster
kubectl apply -f secure-pod.yaml

# Verify security context applied
kubectl get pod secure-app -o jsonpath='{.spec.securityContext}' | jq

# Check running user
kubectl exec secure-app -- id
# Output should show: uid=10001 gid=10001

# Verify network policy
kubectl describe networkpolicy secure-app-netpol

# Test network restrictions
kubectl exec secure-app -- curl other-service  # Should fail if not allowed
```

Production deployment recommendations:

- Store manifests in Git with version control.
- Use Kustomize or Helm for environment variations.
- Implement GitOps (ArgoCD, Flux) for automated deployments.
- Scan manifests with policy tools (OPA, Kyverno) in CI/CD.
- Require security review for manifest changes.
- Implement Pod Security Standards at namespace level.
- Use admission controllers enforcing security policies.
- Monitor deployed pods for compliance drift.
- Conduct regular security audits.
- Maintain documentation of security decisions.

This comprehensive approach creates a hardened pod following defense-in-depth principles, significantly reducing attack surface and blast radius if compromise occurs.