🎯 CATEGORY 1: APPLIED DEVOPS SCENARIOS

Scenario 1: Direct Production Edits (nano/SSH)

Problem: Developers edit production directly using `nano`/SSH

Why Bad:

- ❌ No version control → can't track/rollback changes

- ❌ No testing → bugs go to production

- ❌ No code review → security risks

- ❌ No audit trail → can't track who changed what

- ❌ Direct server access → security vulnerability

First Steps:

1. **Immediate**: Set up Git repository, move all code to version control

2. **CI/CD Pipeline**: Create CI/CD pipeline with stages: lint → test → build → deploy

3. **Infrastructure as Code**: Use Kubernetes manifests or Helm charts instead of manual edits

4. **Access Control**: Remove SSH access, require all changes via Git

5. **Monitoring**: Add health checks (`/healthz`, `/readyz`) and metrics (Prometheus) to detect issues early

Key Points:

- CI/CD pipeline should have: lint → test → security → build → container-build → deploy

- Use Helm for templating (separates logic from configuration)

- Health probes in Kubernetes deployments

- Container registry for images

Scenario 2: Application Crashes on Traffic Spikes

Problem: App can't handle traffic spikes, needs autoscaling

Solution - HPA (Horizontal Pod Autoscaler):

- **Configuration**: Min 2, Max 5 replicas, Target CPU 30%

- **How It Works**: Kubernetes monitors CPU, scales up when >30%, down when <30%

- **Metrics**: Uses Prometheus metrics (`/metrics` endpoint, port 9090)

Additional Steps:

1. **Health Probes**: Liveness (`/healthz`) and Readiness (`/readyz`) probes

2. **Stateless Design**: App must be stateless (use external database/storage, not local filesystem)

3. **Load Balancing**: Kubernetes Service distributes traffic across replicas

4. **Monitoring**: Grafana dashboard to visualize scaling

Key Points:

- HPA enabled in production with pod anti-affinity for fault tolerance

- Metrics exposed on port 9090 with Prometheus annotations

- Application must be stateless to scale horizontally

Scenario 3: Secrets in Git Repository

Problem: Secrets hardcoded in Git, security risk

Solution - Secrets Management:

1. **Remove from Git**: Delete all secrets from codebase

2. **CI/CD Variables**: Store secrets in CI/CD variables (encrypted, masked)

3. **Injection at Deploy**: Use templating (Helm `--set-string`) to inject secrets at deploy time

4. **Kubernetes Secrets**: Secrets created in K8s, never in Git

5. **Rotation**: Easy to rotate (update CI/CD variable, redeploy)

Implementation Flow:

- Secrets in CI/CD variables: `DATABASE_URL`, `JWT_SECRET`, `MINIO_ACCESS_KEY`, etc.

- Templating command: `--set-string serverSecret.extra.DATABASE_URL="${DATABASE_URL}"`

- Kubernetes Secret created, pods mount as env vars

- **Never in Git**: Values files have empty placeholders, secrets only in CI/CD

Scenario 4: No Monitoring/Visibility

Problem: Can't see what's happening in production

Solution - Observability:

1. **Metrics**: Prometheus metrics (4 golden signals)

2. **Health Checks**: Liveness and Readiness probes

3. **Logging**: Application logs via `kubectl logs`

4. **Visualization**: Grafana dashboards

Implementation:

- **Metrics Service**: Expose Prometheus metrics endpoint

- **Metrics**: `http_request_duration_seconds`, `http_requests_total`, `http_errors_total`

- **Health**: `/healthz` (liveness), `/readyz` (readiness, checks DB)

- **Prometheus**: Port 9090, Service annotations `prometheus.io/scrape: "true"`

- **Grafana**: Dashboards with panels (latency, errors, CPU, pod health, HPA)

Scenario 5: Manual Deployments

Problem: Deployments done manually, error-prone

Solution - CI/CD Pipeline:

1. **Automate**: CI/CD pipeline (e.g., GitLab CI/CD)

2. **Stages**: Lint → Test → Security → Build → Container → Scan → Deploy

3. **Atomic Deployments**: `--atomic` flag for automatic rollback

4. **Zero-Downtime**: Rolling updates (new pods ready before old removed)

Pipeline Stages:

- **Lint**: Code quality checks

- **Test**: Unit tests with coverage requirements

- **Security**: Secret detection, vulnerability scanning

- **Build**: Application builds

- **Container Build**: Docker images to registry

- **Image Scan**: Security scanning (Trivy)

- **Deploy**: Helm to Kubernetes, atomic with rollback

🧠 CATEGORY 2: DEVOPS CONCEPTS

Concept 1: Cattle vs. Pets

Definition:

- **Pets**: Named servers, manually maintained, irreplaceable (if dies = crisis)

- **Cattle**: Numbered servers, identical, easily replaced (if dies = replace it)

How It Influences Autoscaling:

- **Stateless Design**: Apps must be stateless (external database/storage, not local filesystem)

- **Immutable Infrastructure**: Containers built once, never modified (Docker images in CI/CD)

- **Horizontal Scaling**: Add/remove identical pods (HPA: 2-5 replicas)

- **No Manual Intervention**: Pods created/destroyed automatically (Kubernetes)

- **Fault Tolerance**: Pod crashes → Kubernetes creates new one (liveness probe)

Key Points:

- Stateless applications enable horizontal scaling

- Identical pods can be replaced without data loss

- HPA automatically creates/destroys pods based on metrics

- Pod anti-affinity ensures pods spread across nodes

Concept 2: Idempotency

Definition: Operation produces same result whether executed once or multiple times.

Why Important:

- **Safe Retries**: Deployment fails → retry without side effects

- **Consistency**: Same config applied multiple times = same result

- **CI/CD Reliability**: Pipeline re-run without breaking things

Examples:

- ✅ `helm upgrade --install` (idempotent - can run multiple times)

- ✅ `kubectl apply` (idempotent - creates if missing, updates if exists)

- ✅ Database migrations (idempotent - safe to run multiple times)

- ✅ Docker builds (same image = same result)

Non-Idempotent (Bad):

- ❌ Deleting file (can't delete twice)

- ❌ Incrementing counter (different result each time)

Concept 3: GitOps

Definition: Infrastructure and code in Git, Git = single source of truth. Changes via Git commits, not manual commands.

Principles:

1. **Declarative**: Infrastructure in YAML (Kubernetes manifests, Helm charts)

2. **Version Controlled**: All config in Git (e.g., `k8s/`, `helm/` directories)

3. **Automated**: CI/CD applies changes (e.g., `.gitlab-ci.yml` deploys on `main`)

4. **Observable**: Monitor what's deployed (Grafana dashboard)

Implementation:

- **Infrastructure as Code**: All K8s resources in version control

- **CI/CD Pipeline**: Auto-deploys on push to main branch
- **No Manual Changes**: All deployments via Git commits, not manual commands
- **Rollback**: Git history allows reverting to previous versions
- **Secrets**: Injected from CI/CD variables, not in Git

Benefits:
- Audit trail (who changed what, when)
- Easy rollback (revert Git commit)
- Consistency (same config in staging/production)

Concept 4: Shift Left

Definition: Move testing, security, quality checks earlier in development lifecycle (left = earlier in timeline).

Traditional (Bad): Test and fix issues in production

Shift Left (Good): Test and fix issues during development

CI/CD Pipeline Stages (Shift Left):

1. **Lint Stage** (earliest): Code quality checks, catches syntax errors early
2. **Test Stage**: Unit tests run before deployment, coverage requirements, failures block deployment
3. **Security Stage**: Scans for hardcoded secrets, catches security issues early
4. **Build Stage**: Builds fail fast if code doesn't compile, catches dependency issues early
5. **Image Scan Stage**: Scans Docker images for vulnerabilities
6. **Deploy Stage** (last): Only deploys if all previous stages pass

Benefits:

- Faster feedback (find issues in minutes, not days)

- Lower cost (fixing bugs in dev cheaper than production)

- Better quality (catch issues before users see them)

Concept 5: Observability (Metrics/Logs/Traces)

Definition: Three pillars of observability

Three Pillars:

1. **Metrics**: Numerical measurements over time (CPU, memory, request rate)

2. **Logs**: Text records of events (application logs, error messages)

3. **Traces**: Request flow through distributed systems

Implementation:

Metrics (Prometheus):

- **Four Golden Signals**:

 - **Latency**: `http_request_duration_seconds` (histogram, buckets 0.1s-10s)

 - **Traffic**: `http_requests_total` (counter, labels: method, route, status)

 - **Errors**: `http_errors_total` (counter, status >= 400)

 - **Saturation**: Custom metrics (e.g., `posts_total`, `users_total`)

- **Exposure**: `/metrics` endpoint on port 9090

- **Scraping**: Prometheus scrapes via Service annotations (`prometheus.io/scrape: "true"`)

Logs:

- Application logs via `kubectl logs`

- Structured logging

Health Checks:

- **Liveness Probe**: `/healthz` (is app running?)

- **Readiness Probe**: `/readyz` (is app ready for traffic? checks database)

Visualization: Grafana dashboard with panels (latency, errors, CPU, pod health, etc.)

Concept 6: Immutable Infrastructure

Definition: Servers/containers never modified after creation. To update, create new containers and destroy old ones.

Traditional (Mutable): SSH into server, edit files, install packages → server drifts from original state

Immutable: Build new container image, deploy it, destroy old containers → always consistent

How Containerization Enables It:

- **Docker Images**: Built once in CI/CD (e.g., `build-server-image`, `build-client-image` jobs)

- **No Manual Changes**: Containers are read-only (except volumes)

- **Version Control**: Images tagged with Git commit (e.g., `IMAGE_TAG: $CI_COMMIT_REF_SLUG`)

- **Rolling Updates**: Kubernetes replaces old pods with new ones (zero-downtime)

Benefits:

- Consistency (dev = staging = production)

- Reproducibility (same image = same behavior)

- Easy rollback (deploy previous image)

- Security (no manual changes = fewer vulnerabilities)

Concept 7: CI/CD Pipelines

Definition: Automated pipeline that builds, tests, and deploys code.

Pipeline Stages (example):

1. **Lint Stage**: ESLint checks code, catches syntax/style errors early

2. **Test Stage**: Unit tests with coverage requirements, ensures code works before deployment

3. **Security Stage**: Scans for hardcoded secrets, prevents secrets in Git

4. **Build Stage**: Creates build artifacts, artifacts cached

5. **Container Build Stage**: Builds Docker images, pushes to registry, images tagged with commit SHA

6. **Image Scan Stage**: Scans images for HIGH/CRITICAL vulnerabilities

7. **Deploy Stage**: Deploys to Kubernetes using Helm, creates ConfigMaps, Secrets, Deployments, Services, Ingress

Key Features:

- **Atomic Deployments**: `--atomic` flag rolls back on failure

- **Secrets Injection**: Secrets from CI/CD variables, not in Git

- **Zero-Downtime**: Rolling updates (new pods ready before old ones removed)

Concept 8: Version Control

Definition: System for tracking changes to files over time.

Why Important:

- **History**: See who changed what, when, why

- **Rollback**: Revert to previous versions

- **Collaboration**: Multiple developers work on same code

- **Branching**: Test changes without affecting main code

Best Practices:

- Commit frequently with clear messages

- Use branches for features

- Never commit secrets

- Use pull requests for code review

Concept 9: Infrastructure as Code (IaC)

Definition: Managing infrastructure using code/configuration files instead of manual processes.

Benefits:

- **Version Control**: Infrastructure changes tracked in Git

- **Reproducibility**: Same config = same infrastructure

- **Automation**: CI/CD can deploy infrastructure

- **Consistency**: Dev = staging = production

Tools:

- **Kubernetes**: Declarative YAML manifests

- **Helm**: Templating engine for K8s (separates logic from config)

- **Docker Compose**: Local development orchestration

Concept 10: Security in DevOps

Definition: Integrating security practices throughout the development lifecycle.

Security Measures:

1. **Secret Management**:
  - Secrets in CI/CD variables (encrypted, masked)
  - Never in Git
  - Injected at deploy time via templating

2. **Security Scanning**:
  - **Secret Detection**: Scans for hardcoded secrets in Git
  - **Image Scanning**: Scans Docker images for vulnerabilities (HIGH/CRITICAL)

3. **Access Control**:
  - RBAC in Kubernetes
  - ServiceAccount for CI/CD
  - No direct SSH access to containers

4. **Network Security**:
  - NetworkPolicy for traffic control

  - Ingress with TLS (cert-manager, Let's Encrypt)

5. **Application Security**:
  - JWT authentication
  - Password hashing (bcrypt)
  - Input validation
  - XSS prevention

Shift Left Security: Security checks early in pipeline (secret detection, image scanning)

🔍 CATEGORY 3: RETROSPECTIVE & ANALYSIS

> **Note**: This section contains project-specific details. Replace with your own project information.

Your Project Summary

Technical Stack:
- **Frontend**: Next.js (React), TypeScript, Tailwind CSS
- **Backend**: NestJS (Node.js), TypeScript, Prisma ORM
- **Database**: PostgreSQL (Kubernetes deployment)
- **Storage**: MinIO (S3-compatible object storage)
- **Containerization**: Docker (multi-stage builds)
- **Orchestration**: Kubernetes (Deployments, Services, Ingress, PVCs)
- **Templating**: Helm (separates logic from configuration)

- **CI/CD**: GitLab CI/CD (7 stages: lint → test → security → build → container-build → image-scan → deploy)

- **Monitoring**: Prometheus (metrics), Grafana (dashboards)

- **Security**: Trivy (image scanning), gitleaks (secret detection)

Architecture Decisions:

- **Stateless Design**: No local filesystem storage (PostgreSQL + MinIO external)

- **Health Probes**: Liveness (`/healthz`) and Readiness (`/readyz`)

- **Autoscaling**: HPA (2-5 replicas, CPU-based, production only)

- **Fault Tolerance**: Pod anti-affinity (pods on different nodes)

- **Secrets Management**: GitLab CI variables → Helm → Kubernetes Secrets

- **Templating**: Helm (values.yaml for staging, values-prod.yaml for production)

Question: "What technical stack did you use? What would you change?"

Current Stack:

- **Frontend**: Next.js (good for SSR, but adds complexity)

- **Backend**: NestJS (excellent for large apps, but might be overkill for simple CRUD)

- **Database**: PostgreSQL (perfect choice, relational data)

- **Storage**: MinIO (good S3-compatible, but AWS S3 might be better for production)

- **Orchestration**: Kubernetes + Helm (industry standard, but complex for small teams)

- **CI/CD**: GitLab CI (good, but could add more stages like integration tests)

What I Would Change:

1. **Simplify Frontend**: Consider React without Next.js if SSR not needed (faster builds)

2. **Add Integration Tests**: E2E tests in CI/CD pipeline (currently only unit tests)

3. **Use Managed Services**: Consider managed PostgreSQL (AWS RDS) instead of self-hosted

4. **Add Monitoring Alerts**: Set up Prometheus alerts (currently only dashboards)

5. **Improve Security**: Add OWASP dependency scanning (currently only Trivy for images)

6. **Optimize Images**: Use multi-stage builds more aggressively (reduce image size)

7. **Add Blue-Green Deployment**: For zero-downtime (currently only rolling updates)

Why These Changes:

- **Managed Services**: Less operational overhead, better reliability

- **More Testing**: Catch integration issues before production

- **Alerts**: Proactive issue detection (not just reactive dashboards)

- **Smaller Images**: Faster deployments, lower costs

Question: "How did you ensure your application is stateless? Why is this important?"

How We Ensured Statelessness:

1. **Database Externalization**:

  - All data in PostgreSQL (Kubernetes service, not local)

  - ORM connects to service name (e.g., `postgresql:5432`, not localhost)

  - Data persists in PersistentVolumeClaim (survives pod restarts)

2. **File Storage Externalization**:

  - Images uploaded to MinIO (S3-compatible)

  - Storage service uses `PutObject`/`GetObject` (never writes to `/tmp`)

  - Files accessible from any pod (shared storage)

3. **Session Management**:

  - JWT tokens stored client-side (browser localStorage)

  - Sessions in database (not in-memory)

  - No pod-local session storage

4. **No Local State**:

  - No files written to container filesystem

  - No in-memory caches (all data from database/storage)

Why Important:

- **Horizontal Scaling**: Multiple pods can serve requests (no data inconsistency)

- **Pod Failures**: If pod crashes, another pod takes over (no data loss)

- **Rolling Updates**: Zero-downtime deployments (old pods removed after new ones ready)

- **Chaos Engineering**: Application survives random pod kills

Verification:

- Pods can be deleted and recreated without data loss

- HPA scales to multiple replicas, all serve same data

- Rolling updates work without downtime

Question: "Explain your secrets management strategy. Why is it secure?"

Strategy: **Never commit secrets to Git**. All secrets stored in CI/CD variables, injected only during deployment.

Implementation:

1. **Storage**:
   - Secrets in CI/CD variables (encrypted at rest, masked in logs)
   - Variables: `DATABASE_URL`, `JWT_SECRET`, `MINIO_ACCESS_KEY`, `MINIO_SECRET_KEY`, `NEXTAUTH_SECRET`

2. **Injection**:
   - Templating uses empty placeholders in values files (not in Git)
   - CI/CD job passes secrets via `--set-string` flags
   - Secrets only exist in: CI/CD variables → CI job → Kubernetes Secrets → Pods

3. **Security Measures**:
   - **Masked**: Secrets masked in CI/CD logs (can't see values)
   - **Protected**: Variables marked as "protected" (only available in protected branches)
   - **Base64 Encoding**: Kubernetes Secrets base64-encoded (not plaintext in etcd)
   - **No Git History**: Secrets never in Git, so no risk of exposure in history

Why Secure:
- **No Git Exposure**: Even if repo is public, secrets aren't exposed
- **Access Control**: Only CI/CD job can access secrets (not developers)
- **Audit Trail**: CI/CD logs who accessed secrets (if enabled)
- **Rotation**: Easy to rotate (update CI/CD variable, redeploy)

Example Flow:

1. Developer pushes code (no secrets in code)

2. CI/CD job reads `JWT_SECRET` from CI/CD variables

3. Templating command: `--set-string serverSecret.extra.JWT_SECRET="${JWT_SECRET}"`

4. Kubernetes Secret created (base64-encoded)

5. Pod mounts secret as environment variable

Question: "How did you implement monitoring? What metrics do you track?"

Implementation:

1. **Metrics Service** (e.g., `server/src/metrics/metrics.service.ts`):

  - **Four Golden Signals**:

    - **Latency**: `http_request_duration_seconds` (histogram, buckets: 0.1s-10s)

    - **Traffic**: `http_requests_total` (counter, labels: method, route, status)

    - **Errors**: `http_errors_total` (counter, status >= 400)

    - **Saturation**: Custom metrics (e.g., `posts_total`, `users_total`)

  - **Exposure**: Separate server on port 9090, `/metrics` endpoint

  - **Format**: Prometheus text format

2. **Metrics Middleware**:

  - Intercepts all HTTP requests

  - Records duration, method, route, status code

  - Calls metrics service

3. **Kubernetes Integration**:

  - **Service Annotations**: `prometheus.io/scrape: "true"`, `prometheus.io/port: "9090"`

  - **NetworkPolicy**: Allows Prometheus from monitoring namespace

  - **Scraping**: Prometheus scrapes every 30 seconds

4. **Health Checks**:

  - **Liveness**: `/healthz` (app running?)

  - **Readiness**: `/readyz` (app ready for traffic? checks database)

5. **Visualization**:

  - **Grafana Dashboard**: Multiple panels (latency, errors, CPU, pod health, HPA, etc.)

  - **Thresholds**: Color-coded (green/yellow/orange/red)

  - **Time Range**: Configurable (e.g., 6 hours default)

What We Track:

- Request latency (95th percentile)

- Request rate (requests/second)

- Error rate (errors/second)

- Pod health (ready/unready)

- CPU usage (for HPA)

- Database status

- Storage usage

Use Cases:

- **Performance**: Identify slow endpoints (latency > 2s)

- **Reliability**: Detect errors (error rate > 0.1/s)

- **Scaling**: HPA uses CPU metrics (scale when > 30%)

- **Debugging**: Pod restarts indicate crashes

Question: "Why did you choose Helm over Kustomize? (or vice versa)"

Helm vs. Kustomize:

Helm:
- **Pros**:
  - Templating engine (Go templates)
  - Values files (separates logic from config)
  - Package manager (can share charts)
  - Atomic deployments (`--atomic` flag)
  - Industry standard, large community
- **Cons**:
  - More complex (templates can be hard to read)
  - Requires Helm binary
  - Steeper learning curve

Kustomize:
- **Pros**:
  - Native to Kubernetes (built into `kubectl`)
  - Simpler (YAML overlays)
  - No external tools needed
  - Easy to understand
- **Cons**:

  - Less powerful (no templating)

  - Harder to share/reuse

  - No atomic deployments

Why I Chose Helm:

- **Templating**: Need to separate logic from config (values.yaml vs templates)

- **Multiple Environments**: Staging and production (values.yaml vs values-prod.yaml)

- **Secrets Injection**: Easy to inject secrets via `--set-string`

- **Atomic Deployments**: `--atomic` flag for automatic rollback

- **Industry Standard**: More widely used, better documentation