1️⃣ Editing Production Directly

Editing production servers directly (for example using nano) is a serious DevOps anti-pattern. It removes version control, meaning there is no history of changes, no accountability, and no easy rollback if something breaks. Changes go live without testing, which increases the risk of bugs reaching users and causing outages. It also creates security risks because direct server access is hard to audit and control.

The correct approach is to use Git as a single source of truth, where all changes are tracked and reviewed. A CI/CD pipeline should automatically test and deploy code in a controlled manner. Infrastructure should be managed using Infrastructure as Code, ensuring consistency across environments. Direct access to production servers should be restricted, and basic monitoring should be added to detect issues early. Automation and traceability together improve reliability.

2️⃣ Traffic Spikes / Application Crashes

Applications that cannot handle traffic spikes usually rely on manual scaling, which is slow and unreliable. During high traffic, this leads to downtime, poor user experience, and potential revenue loss. Manual fixes do not scale and often happen too late.

From a DevOps perspective, systems should be designed to scale automatically. This requires horizontal scaling instead of vertical scaling, meaning adding more instances rather than upgrading a single server. The application should be stateless so that any instance can handle any request. Load balancing is needed to distribute traffic evenly. Autoscaling based on metrics such as CPU or request count ensures the system responds to demand in real time. Health checks and monitoring help remove unhealthy instances automatically. Modern systems must scale by design, not by human intervention.

3️⃣ Secrets in Code or Git

Storing secrets such as API keys, passwords, or tokens in code or Git repositories is a major security risk. Once a secret is committed, it becomes part of the Git history and can be exposed permanently. Anyone with repository access can misuse these secrets, leading to data breaches or system compromise.

The correct DevOps practice is to never store secrets in version control. Secrets should be managed separately using secret managers or CI/CD environment variables. They should be injected at deployment or runtime rather than hard-coded. This allows secrets to be rotated without changing application code and reduces the blast radius of

leaks. Access should follow the principle of least privilege. Treating secrets as configuration instead of code improves security, compliance, and operational safety.

4️⃣ No Monitoring or Visibility

Without monitoring, failures are often detected by users instead of the engineering team. This leads to slow incident response, poor reliability, and difficult debugging because there is no clear insight into what went wrong. Operating systems without visibility turns troubleshooting into guesswork.

A proper DevOps setup includes observability through metrics, logs, and health checks. Metrics provide information about system health and performance, logs explain what happened inside the system, and health checks help detect failing services early. Dashboards give teams real-time visibility into system behavior, while alerts notify them before users are affected. Monitoring enables faster debugging, better scaling decisions, and higher system reliability. In DevOps, visibility is essential because problems cannot be fixed if they cannot be seen.

5️⃣ Cattle vs. Pets

The “Cattle vs. Pets” analogy describes how servers are managed. Pets are individually maintained, irreplaceable, and require manual care—if a pet server dies, it’s a crisis. Cattle, by contrast, are disposable, identical, and replaceable; if one fails, it is destroyed and replaced automatically. This mindset is critical for modern DevOps because it enables autoscaling and fault tolerance. Stateless design is required, so that any server can handle requests without dependency on local data. Kubernetes exemplifies cattle management: pods can be created or destroyed automatically, with load balancing and health checks ensuring reliability. Treating infrastructure as cattle reduces operational overhead, supports rolling updates, and allows horizontal scaling without downtime. Modern cloud-native applications rely on this approach for resilience, automation, and predictable behavior.

6️⃣ Idempotency

Idempotency ensures that performing an action multiple times results in the same outcome as doing it once. This property is crucial for automation and CI/CD pipelines because it allows retries without side effects, ensuring consistency and reliability. For example, kubectl apply or helm upgrade --install are idempotent: running them multiple times does not break the system. Non-idempotent operations, like incrementing counters or deleting files without checks, can cause inconsistent states. Idempotency

makes deployments predictable and safe, reduces the risk of errors, and simplifies automated processes. In DevOps, automation must be reliable, repeatable, and safe to re-execute—idempotency guarantees that pipelines and scripts behave consistently even when rerun, enabling confident continuous deployment.

7️⃣ GitOps

GitOps is a DevOps approach where Git serves as the single source of truth for both application code and infrastructure. All changes are made via Git commits, which trigger automated deployments through CI/CD pipelines. This approach ensures that deployments are versioned, auditable, and reproducible. Rollbacks are easy because Git history records all changes. Infrastructure as code and declarative configurations, like Kubernetes manifests or Helm charts, enable GitOps workflows. By integrating GitOps, operations follow the same workflow as development: changes are reviewed, tested, and deployed automatically. GitOps increases reliability, reduces human error, improves collaboration, and provides traceability, making it a core best practice for modern, cloud-native DevOps environments.

8️⃣ Shift Left

Shift Left is the practice of moving testing, security, and quality checks earlier in the development lifecycle. By detecting issues during development instead of production, feedback is faster, bugs are cheaper to fix, and overall software quality improves. For instance, running unit tests, linting, and security scans in early CI/CD stages prevents problems from reaching live environments. Security shift-left measures, such as scanning for secrets or vulnerabilities during the build, ensure that risks are mitigated proactively. By shifting left, teams catch errors before deployment, accelerate delivery, and reduce downtime. In DevOps, Shift Left reduces risk, increases efficiency, and aligns with the principle of continuous improvement and early validation.

9️⃣ Observability

Observability is the ability to understand the internal state of a system based on its outputs. It relies on three pillars: metrics, logs, and traces. Metrics provide numeric measurements of system health, logs record events and errors, and traces show the flow of requests across distributed components. Observability enables teams to detect, diagnose, and fix issues faster. For example, Prometheus metrics can monitor latency, traffic, and error rates, while Grafana dashboards visualize system performance. Without observability, debugging is guesswork. With it, teams can proactively scale

applications, improve reliability, and respond to failures effectively. Observability transforms operational data into actionable insights, supporting automation and high availability in DevOps workflows.

🔟 Immutable Infrastructure

Immutable infrastructure means that servers or containers are never modified after deployment. Updates are performed by replacing instances with new ones rather than editing in place. This ensures consistency across environments, simplifies rollback, and reduces configuration drift. For example, Docker images built in CI/CD pipelines are immutable: any change requires building a new image, which is then deployed via Kubernetes. Rolling updates replace old pods with new ones without downtime, and previous versions can be restored easily. Immutable infrastructure reduces human errors, ensures predictable deployments, and aligns with DevOps principles like automation, reproducibility, and stateless design. “Rebuild, don’t repair” is the key mindset.

1️⃣1️⃣ CI/CD Pipelines

CI/CD pipelines automate the process of building, testing, and deploying software. Continuous Integration ensures that code changes are automatically tested and integrated into the main branch, while Continuous Deployment automates release to production. Automation reduces human error, accelerates delivery, and allows safe rollbacks. Pipelines can include stages for linting, testing, security scans, building Docker images, and deployment. Tools like GitLab CI/CD or GitHub Actions orchestrate these stages. By standardizing the deployment process, CI/CD pipelines increase reliability, consistency, and feedback speed. Automation replaces manual work, ensuring that software is delivered faster, safer, and with predictable outcomes.

1️⃣2️⃣ Infrastructure as Code (IaC)

Infrastructure as Code (IaC) manages infrastructure using configuration files and code rather than manual processes. IaC ensures that environments are version-controlled, reproducible, and automated. Kubernetes manifests, Helm charts, and Terraform scripts are examples of IaC tools. By defining infrastructure declaratively, teams can create consistent staging, testing, and production environments, reducing drift and human error. CI/CD pipelines can deploy infrastructure automatically, enabling repeatable and reliable provisioning. IaC allows rollback to previous configurations and

simplifies scaling. In DevOps, infrastructure should be reproducible, testable, and automated, ensuring predictable deployments and operational efficiency.

1️⃣3️⃣ Security in DevOps (DevSecOps)

Security in DevOps is continuous and integrated throughout the software lifecycle. Best practices include least privilege access, secret management, vulnerability scanning, and shift-left security checks. Secrets should never be committed to Git and should be injected securely at deployment. CI/CD pipelines can scan code and container images for vulnerabilities early. Role-Based Access Control (RBAC) and network policies limit access and reduce risk. By embedding security into all stages of development, DevSecOps ensures compliance, reduces incidents, and aligns with automation principles. Continuous monitoring, proactive scanning, and early detection make security a default, not an afterthought.

🟢 CATEGORY 3: RETROSPECTIVE & ANALYSIS

(Easy marks if answered maturely)

1️⃣ Technical Stack (Keep It Short)

- Frontend framework
- Backend framework
- Database
- Containers
- CI/CD
- Orchestration
- Monitoring

2️⃣ What Would You Change?

Good answers include:

- Simplify architecture
- Add more testing
- Use managed services

- Improve alerting
- Reduce operational overhead

👉 *Show learning, not perfection*

3️⃣ **Statelessness**

- No local storage
- External database
- External file storage
- Enables scaling and resilience

👉 *Stateless apps scale and recover easily*