Zero Downtime Deployment with Deployment Strategies

By DevOps Shack

[Click here for DevSecOps & Cloud DevOps Course]

DevOps Shack

Zero Downtime Deployment

with Deployment Strategies

Table of Contents

- Auto/manual rollback mechanisms

8. **Best Practices & Case Study**
   - Real-world implementation example
   - Zero downtime deployment checklist

📌 1. Introduction & Importance of Zero Downtime Deployment

What is Zero Downtime Deployment?

Zero Downtime Deployment refers to the process of updating an application or service without causing any interruption to its availability. In other words, users can continue interacting with the application seamlessly while updates are being deployed behind the scenes. This approach is crucial for modern, high-availability systems—especially those serving global users 24/7.

Downtime, even if it's for a few minutes, can lead to a poor user experience, loss of revenue, customer dissatisfaction, and in some sectors like finance or healthcare, even critical failures.

Traditional deployment methods often involve stopping the current version of an application, replacing it with the new one, and then restarting the system. During this window, services become temporarily unavailable. This method might work in development or small-scale applications, but it's unacceptable in production environments where availability is a priority.

Why Zero Downtime Matters

1. User Experience

Customers expect services to be available at all times. Whether it’s an e-commerce checkout, a banking transaction, or booking a flight—any disruption could result in frustration and lost trust. In competitive industries, users may switch to a rival's service after even a single bad experience.

2. Global User Base

Applications today are used globally. There’s no longer a perfect "off-hour" to deploy an update. For instance, 3 AM in New York is prime time in parts of Asia. Zero Downtime Deployment ensures that no matter the time zone, users don’t face service interruptions.

3. Revenue Protection

Even short downtimes can cause financial losses. Imagine a payment gateway going down during a flash sale. Lost transactions directly impact the bottom

line. In B2B applications, it can also affect service-level agreements (SLAs) and result in penalties.

4. Operational Efficiency

Zero Downtime strategies often require automation, monitoring, and well-defined processes, which improve the overall maturity of the DevOps lifecycle. This, in turn, reduces manual errors, increases deployment frequency, and boosts developer confidence in making changes.

5. Brand Trust & Reputation

For tech-first companies, uptime is a part of their brand identity. A history of smooth, uninterrupted releases builds trust among users and stakeholders. On the flip side, frequent downtimes (especially during updates) tarnish credibility.

Key Concepts Associated with Zero Downtime

- **High Availability (HA):** Ensuring that the system remains available even if some components fail or are being updated.
- **Immutable Deployments:** Rather than modifying the existing application, deploy a new version in parallel.
- **Blue-Green Deployment:** A strategy where two environments (blue & green) are used, and traffic is switched between them.
- **Canary Releases:** Rolling out the update to a small group first to catch issues early.
- **Graceful Shutdown:** Allowing active requests to complete before killing a process.
- **Rolling Updates:** Updating servers one at a time while keeping the service live.

Common Pitfalls Without Zero Downtime

- Users see 500 errors or maintenance pages during deployments.
- Incomplete database migrations corrupt data or crash services.
- Long deployments cause customer support overload.

- Developers fear releasing because it could break production.

📌 2. Deployment Strategies for Zero Downtime

Implementing zero downtime requires thoughtful planning around *how* new code is rolled out. A bad deployment strategy can take your entire system down. Below are the core deployment strategies used to achieve zero downtime in real-world systems, each with its unique advantages and trade-offs.

✅ Blue-Green Deployment

Concept:
Maintain two identical environments—**Blue** (currently live) and **Green** (new version). Users are routed to Blue. When the new version is ready in Green and passes tests, you switch traffic from Blue to Green.

Workflow:

1. Blue is live, Green is idle.
2. Deploy the new version to Green.
3. Run integration/smoke tests on Green.
4. Switch traffic from Blue to Green.
5. Blue becomes the new backup.

Benefits:

- Easy rollback: Just switch traffic back.
- Safer testing in a production-identical environment.

Trade-offs:

- Requires double the infrastructure.
- Complex traffic routing setup (DNS or load balancer).

✅ Canary Deployment

Concept:
Gradually roll out the new version to a small subset of users, monitor behavior, then increase traffic over time.

Workflow:

1. Deploy to 5% of users (the "canaries").
2. Monitor for errors, latency, logs.
3. Increase to 25%, 50%, then 100% if stable.

Benefits:

- Issues can be caught before full deployment.
- Real user feedback on the new version.

Trade-offs:

- More complex deployment scripts or orchestration tools needed.
- May require smart routing and real-time traffic splitting.

✅ Rolling Update

Concept:
Deploy the new version incrementally across instances or containers, one by one, while keeping the system online.

Workflow:

1. Out of N instances, take one offline.
2. Update and restart it.
3. Repeat for each instance.

Benefits:

- No need for duplicate environments.
- Easier to manage on containerized or clustered systems.

Trade-offs:

- Rollback is slower and less straightforward.

- If not monitored properly, partial failures can impact users.

✅ Feature Toggles (Feature Flags)

Concept:
Release the code to production but hide or control access to new features using configuration flags. You can turn features on or off without redeploying.

Use Cases:

- Gradual rollout to internal users.
- A/B testing.
- Instant rollback of specific features.

Benefits:

- Deployment and release become decoupled.
- Risk is limited to specific features, not entire systems.

Trade-offs:

- Can add technical debt if flags aren't cleaned up.
- Logic complexity increases in code.

✅ Shadow Deployment

Concept:
Deploy the new version alongside the old one and duplicate live traffic to it—*without affecting users*. The new version processes real requests but doesn't return results to end users.

Benefits:

- Safe way to test performance and logic at scale.
- Useful for testing rewrites or entirely new stacks.

Trade-offs:

- Doubles resource usage.
- Complicated to capture and route traffic without introducing latency.

Choosing the Right Strategy

The best deployment strategy depends on:

- **Team maturity and tooling**
- **Traffic volume and risk tolerance**
- **Application architecture (monolith vs microservices)**
- **Cloud or infrastructure capabilities**

For example:

- A startup may begin with Rolling Updates and add Feature Flags for flexibility.
- A mature enterprise might combine Canary + Blue-Green for highly critical services.

Tooling to Support Deployment Strategies

Most modern DevOps tools support these strategies out of the box:

- **Kubernetes:** Native support for Rolling Updates and Canary via controllers.
- **AWS CodeDeploy / ECS / EKS:** Support Blue-Green and Canary.
- **Argo Rollouts / Spinnaker:** Advanced progressive delivery and observability.
- **LaunchDarkly / Unleash:** Dedicated tools for Feature Toggles.

📌 3. Application Architecture Considerations

Zero downtime deployment isn't just about how you push code—it heavily depends on *how your application is architected*. Even the best deployment strategy can fail if the application isn't designed to handle live updates gracefully. This section outlines key architectural principles and best practices required to support smooth, interruption-free deployments.

✅ 1. Stateless Applications

Statelessness means that the application does not rely on local or in-memory data that persists between requests. Each instance should handle requests independently without shared session memory or temporary files.

Why It Matters:

- You can shut down and restart instances at any time without affecting user sessions.
- Makes scaling and rolling updates much easier.

Best Practices:

- Use external session stores (e.g., Redis, Memcached).
- Don't store files locally—use object storage like Amazon S3.
- Use consistent, idempotent APIs.

✅ 2. Graceful Startup and Shutdown

Applications should **start up and shut down gracefully** to avoid interrupting in-flight requests or leaving the system in an inconsistent state.

Startup Considerations:

- Wait until the app is fully initialized (DB connections, caches, etc.) before accepting traffic.
- Use readiness probes (especially in Kubernetes) to delay traffic routing.

Shutdown Considerations:

- Stop accepting new requests immediately.
- Allow current requests to finish processing.
- Close connections cleanly and flush logs/buffers.

Tools/Features:

- SIGTERM signal handling
- Kubernetes preStop hooks
- Application lifecycle hooks in frameworks like ASP.NET Core (IHostApplicationLifetime), Spring Boot, etc.

✅ 3. Backward-Compatible Changes

Zero downtime demands that **new versions must work with old data, and vice versa**, at least during the rollout period.

Why? During phased rollouts or canary deployments, multiple versions of the app may be live simultaneously.

How to Achieve This:

- Avoid breaking API changes.
- Support old and new schema versions (e.g., using versioned APIs).
- Database migrations should be additive and non-destructive (covered in detail later).

Example:

- Instead of renaming a column, add the new column, update both in the app, then remove the old column in a later deployment.

✅ 4. Decoupled Components & Microservices

Monolithic architectures make zero downtime much harder, as one failure can take everything down. **Microservices**, or at least decoupled modules, improve deployability.

Benefits:

- Deploy one service without affecting the entire system.
- Scope and isolate failures to specific areas.
- Easier to implement canary and feature toggle strategies.

Best Practices:

- Use queues (e.g., RabbitMQ, Kafka) for async communication.
- Design services with well-defined contracts and interfaces.
- Handle partial failures gracefully (e.g., fallbacks, retries).

✅ 5. Resilience & Fault Tolerance

Zero downtime doesn't mean zero issues—it means the system can *tolerate* issues.

Resilience Patterns:

- **Circuit Breakers:** Prevent cascading failures.
- **Retries with Exponential Backoff:** Handle transient issues.
- **Fallbacks:** Gracefully degrade functionality.

Libraries:

- Polly (.NET)
- Resilience4j (Java)
- Istio (for service mesh-level resilience)

✅ 6. Observability Built-In

You can't deploy safely if you don't know what's happening inside your app.

Architectural Additions:

- Logging at each request/response layer
- Application metrics (latency, throughput, errors)
- Tracing for distributed services (OpenTelemetry, Jaeger)

Benefits:

- Real-time detection of degraded performance or errors during deployment.
- Easy to correlate issues with specific components or versions.

✅ 7. Idempotent Endpoints

Every API endpoint should be **idempotent**—calling it multiple times shouldn't cause different results. This is critical for retry-safe deployments and graceful recovery from transient failures.

Example:

- Submitting a payment twice should not double-charge a user.
- Use unique request IDs or tokens to prevent re-processing.

✅ 8. Version-Aware Clients

In some architectures, clients (such as mobile apps or other services) directly interact with APIs. It's crucial that the backend supports **versioning** so that older clients remain functional during/after a deployment.

Techniques:

- URI versioning: /api/v1/users
- Header-based versioning
- Media type versioning (HATEOAS style)

📌 4. CI/CD Pipeline & Automation

A robust CI/CD pipeline is the engine behind zero downtime deployment. It automates the process of building, testing, and deploying code with minimal human intervention and maximum consistency. Without automation, zero downtime deployment becomes error-prone, slow, and difficult to scale.

✅ Key Stages of a CI/CD Pipeline for Zero Downtime

1. Code Commit & Version Control

Every change starts with a commit to a Git repository (GitHub, GitLab, Bitbucket, etc.). This commit triggers the CI/CD pipeline.

Best Practices:

- Use branch protection rules.
- Require code reviews and linting.
- Tag releases with semantic versioning.

2. Build Stage

This stage compiles your code, installs dependencies, and generates build artifacts like Docker images, binaries, or static files.

Example (GitHub Actions - .NET Core Docker build):

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Code
        uses: actions/checkout@v3

      - name: Set up .NET
```

```
      uses: actions/setup-dotnet@v3
      with:
        dotnet-version: 7.0.x

    - name: Build App
      run: dotnet build --configuration Release

    - name: Publish App
      run: dotnet publish -c Release -o out

    - name: Docker Build
      run: docker build -t myapp:${{ github.sha }} .
```

3. Test Stage

Run unit tests, integration tests, and smoke tests to validate the build.

```
- name: Run Tests
  run: dotnet test --no-build --verbosity normal
```

Include:

- Unit tests
- Integration tests with mock services
- Load and performance tests (in staging)

4. Security & Static Analysis (Optional but Recommended)

Use tools like:

- SonarCloud, Snyk for static code analysis
- OWASP ZAP, Trivy for vulnerability scanning

```
- name: Scan Docker Image for Vulnerabilities
  uses: aquasecurity/trivy-action@master
  with:
    image-ref: 'myapp:${{ github.sha }}'
```

5. Artifact Storage (Docker Hub / ECR)

Push versioned builds to container registries so they can be safely pulled by deployment tools.

```
- name: Login to Docker Hub
  run: echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u ${{ secrets.DOCKER_USERNAME }} --password-stdin

- name: Push Docker Image
  run: docker push myapp:${{ github.sha }}
```

6. Deploy Stage (Zero Downtime Strategies)

Automate deployments using tools that support canary, rolling, or blue-green strategies.

Example: Rolling Update via kubectl (Kubernetes):

```
- name: Set Kubernetes Context
  uses: azure/setup-kubectl@v3

- name: Rolling Update Deployment
  run: |
    kubectl set image deployment/myapp-deployment myapp=myapp:${{ github.sha }}
    kubectl rollout status deployment/myapp-deployment
```

K8s Deployment YAML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
spec:
  replicas: 4
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    spec:
      containers:
        - name: myapp
          image: myregistry/myapp:latest
          ports:
            - containerPort: 80
```

7. Health Checks & Readiness Probes

Automated pipelines should check if the new deployment is healthy before switching traffic.

Kubernetes readiness probe:

```
readinessProbe:
  httpGet:
    path: /health
```

```
  port: 80
 initialDelaySeconds: 10
 periodSeconds: 5
```

8. Rollback Automation (Optional but Powerful)

Configure pipelines to **automatically rollback** on failure using Helm/Kustomize/GitOps controllers.

Example: GitHub Action rollback step (K8s):

```
- name: Rollback on Failure
 if: failure()
 run: |
  kubectl rollout undo deployment/myapp-deployment
```

✅ **Tools to Power This Pipeline**

Stage	Tools
CI	GitHub Actions, GitLab CI
Build & Test	Docker, .NET CLI, Jest, etc.
Deploy	Helm, ArgoCD, Spinnaker
Monitoring	Prometheus, Grafana, Sentry
Rollbacks	Kubernetes, Flux, Argo Rollouts

📌 5. Infrastructure Setup for Zero Downtime Deployment

Even with the best CI/CD strategy, you won't achieve zero downtime unless your **infrastructure** is equally resilient, redundant, and dynamically scalable. This section focuses on the design, provisioning, and configuration of cloud/on-prem infrastructure that supports non-disruptive deployments.

✅ 1. Load Balancers

Purpose: Distribute incoming traffic across multiple instances of your app.

Why It's Crucial: During deployment, load balancers ensure that only *healthy instances* serve traffic. They enable rolling, blue-green, and canary deployments by routing traffic appropriately.

Popular Load Balancers:

- AWS ALB/ELB
- NGINX/HAProxy
- Azure Load Balancer
- Traefik (K8s-native)

AWS Example: ALB Target Group Setup

- Attach app instances to **target groups**
- Use **health checks** to deregister unhealthy instances
- Switch target groups during Blue-Green deployments

```
aws elbv2 register-targets \
  --target-group-arn <arn> \
  --targets Id=i-abc123,Id=i-def456
```

✅ 2. Auto-Scaling Groups

Purpose: Maintain high availability and performance by automatically scaling in/out based on demand.

Usage:

- Replace instances being updated or rolled back.
- Spin up new nodes for the new app version (used in blue-green or canary setups).

Kubernetes (Horizontal Pod Autoscaler)

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: myapp-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: myapp-deployment
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 70
```

✅ 3. Immutable Infrastructure

Concept: Never modify live infrastructure. Instead, deploy new versions from scratch and replace the old ones.

Tools That Support This:

- **Terraform / Pulumi** for provisioning
- **Packer** to build golden machine images
- **Kubernetes** via declarative YAML manifests

Terraform Sample Snippet:

```
resource "aws_instance" "app" {
  ami           = "ami-xyz"
  instance_type = "t3.micro"

  user_data = file("deploy.sh")

  lifecycle {
    create_before_destroy = true
  }
}
```

✅ 4. Service Discovery

When containers or instances are recreated during deployment, their IPs change. Service discovery lets your app dynamically find and connect to dependencies.

Popular Tools:

- Kubernetes DNS (built-in)
- AWS Cloud Map / ECS Service Discovery
- HashiCorp Consul

Example (Kubernetes):

```
curl http://myapp-service.namespace.svc.cluster.local
```

✅ 5. Database Redundancy & Resilience

While app deployments can be zero-downtime, **databases** are often a bottleneck.

Best Practices:

- Use managed DBs (RDS, Cloud SQL) with Multi-AZ setup.
- Enable automatic backups and replication.
- Avoid long schema locks—use **online migrations** (next point will cover this in depth).

Multi-AZ Setup (AWS RDS):

- Active-passive replication
- Automatic failover

✅ 6. Network Design & Isolation

For secure and reliable deployments, infrastructure should follow **layered network separation**.

Best Practices:

- Public vs. private subnets
- Isolate staging and production VPCs
- Use security groups/firewall rules to limit access

Example AWS VPC Layout:

- Public Subnet: ALB, NAT Gateway
- Private Subnet: App Servers, Databases
- Route Tables: NAT out from private → internet

✅ 7. Container Orchestration (Kubernetes / ECS)

If you're aiming for dynamic scaling, rolling updates, and service health checks, you'll likely deploy using a container orchestrator.

Benefits of Kubernetes:

- Declarative infrastructure
- Rolling updates out-of-the-box
- Native support for health checks, autoscaling, and service discovery

Sample Deployment.yaml with Rolling Strategy:

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1
    maxUnavailable: 1
```

✅ **8. Observability Infrastructure**

Your infrastructure should always be paired with:

- **Centralized Logging** (ELK, Loki, CloudWatch)
- **Metrics Monitoring** (Prometheus + Grafana)
- **Tracing** (Jaeger, Zipkin, AWS X-Ray)

Example Grafana Dashboard Panels:

- Uptime % over time
- Pod restarts
- Request latency spikes during deploys

Bonus: Infrastructure as Code (IaC)

Everything above should be **codified** using:

- Terraform (infra)
- Helm/Kustomize (K8s manifests)
- CI/CD scripts (GitHub Actions, GitLab CI)

This guarantees repeatability, traceability, and automation — all critical for zero downtime.

📌 6. Database Schema Changes Without Downtime

💡 Why It's Tricky

Unlike stateless applications, databases are **stateful**, shared components. During deployments, if your app and DB schema change at different times, **you risk broken queries, data corruption, or locked tables**.

To safely evolve your schema, your changes must be:

- **Backward compatible**
- **Non-blocking**
- **Applied incrementally**

✅ Key Principles for Safe DB Changes

1. Backward-Compatible Schema Changes

Golden Rule: New code must work with the old schema, and old code must work with the new schema.

That means:

- Never drop or rename a column that's still being accessed.
- Never change data types in-place.
- Avoid using NOT NULL without defaults on new columns.

2. Use a 3-Step Deployment Pattern

Let's say you want to **rename a column from full_name to name**.

Do it like this:

🔹 Step 1 – Add the new column

```
ALTER TABLE users ADD COLUMN name VARCHAR(255);
```

Update your app to **write to both name and full_name, but still read from full_name**.

- **Step 2 – Backfill and switch reads**

```
UPDATE users SET name = full_name WHERE name IS NULL;


-- Update app to read from `name` and stop using `full_name`
```

- **Step 3 – Drop the old column**

```
ALTER TABLE users DROP COLUMN full_name;
```

3. Online Migrations (Avoid Locks)

Use migration tools or techniques that **don't lock tables**, especially in production.

Tools:

- **PostgreSQL**: pg_repack, gh-ost
- **MySQL**: pt-online-schema-change, gh-ost
- **ORM-based**: Entity Framework, Sequelize, TypeORM with versioning

Example (Entity Framework Core - .NET):

```
dotnet ef migrations add AddNameColumn

dotnet ef database update
```

Use **Idempotent SQL scripts** if you're not using ORMs.

4. Schema Migration Automation in CI/CD

You can integrate schema changes in your pipeline.

Example GitHub Action for EF Core:

```
- name: Apply EF Core Migrations
  run: dotnet ef database update --project MyApp.DataAccess
```

Caution: Only do this in staging/UAT or **gated production steps** to prevent accidental destructive changes.

5. Safe Default Values

Avoid ALTER TABLE ... ADD COLUMN x TYPE NOT NULL without a default. It can lock your table for a long time on large datasets.

Instead:

```
ALTER TABLE orders ADD COLUMN status VARCHAR(20) DEFAULT 'pending' NULL;


-- Later...
UPDATE orders SET status = 'pending' WHERE status IS NULL;


-- Finally (optional):
ALTER TABLE orders ALTER COLUMN status SET NOT NULL;
```

6. Use Feature Flags for Schema Rollouts

Let your app toggle between old and new schema logic.

Example (TypeScript/Vue/.NET):

```
if (featureFlags.UseNewCustomerSchema) {
   query = "SELECT name FROM users";
} else {
   query = "SELECT full_name FROM users";
}
```

7. Monitor Schema Changes in Real Time

Track slow queries, lock waits, and migration duration with:

- **New Relic, DataDog APM**
- **pg_stat_activity** (Postgres)
- **Information Schema / Performance Schema** (MySQL)

8. Rollback Strategy for Schema Changes

Be prepared:

- Keep a backup of data before destructive changes.
- Apply reversible migrations (via tools like Flyway, Liquibase).
- Never drop columns immediately — deprecate first, then remove after N deployments.

Sample SQL Migration File (Safe)

```
-- V1__Add_new_name_column.sql

-- Step 1: Add column
ALTER TABLE users ADD COLUMN name VARCHAR(255);

-- Step 2: Backfill
UPDATE users SET name = full_name WHERE name IS NULL;

-- Step 3: Allow NULLs until app updates

-- Step 4: Mark for removal (handled in next release)
-- ALTER TABLE users DROP COLUMN full_name;
```

📌 7. Deployment Strategies for Zero Downtime

Zero downtime deployment isn't just about automation and good infrastructure — it's also about *how* you push your changes live. These strategies ensure that users don't experience any interruptions, even as new code replaces old.

🔹 1. Rolling Deployments

Update a few instances at a time while others continue serving traffic.

🔧 **How it works:**

- The load balancer keeps routing traffic to **healthy old instances**.
- A few pods/servers are updated to the new version.
- Once healthy, the next batch is rolled out.

✅ **Pros:**

- Minimal extra infrastructure.
- Built-in in Kubernetes.

❌ **Cons:**

- If a bug sneaks in, some users may hit the broken version before rollback.

📄 **Kubernetes Example:**

```
strategy:
 type: RollingUpdate
 rollingUpdate:
  maxUnavailable: 1
  maxSurge: 1
```

K8s will:

- Launch 1 extra pod (surge).
- Take down 1 old pod.

- Wait for the new one to become healthy.
- Repeat until complete.

🔹 2. Blue-Green Deployment

Run two identical environments side by side: **Blue (old)** and **Green (new)**. Switch traffic instantly once the new version is verified.

🔧 **How it works:**

1. Deploy v2 (Green) in parallel to v1 (Blue).
2. Test Green in isolation.
3. Flip the router/load balancer from Blue → Green.

✅ **Pros:**

- Instant rollback (just flip back to Blue).
- Perfect for major version jumps.

❌ **Cons:**

- Requires double infra (2 environments).

🛠️ **NGINX Switch Example:**

```
upstream app {
 server green.example.com; # switch from blue.example.com
}

server {
 location / {
  proxy_pass http://app;
 }
}
```

Update NGINX config → reload:

```
nginx -s reload
```

🔹 3. Canary Deployment

Release new code to **a small % of users**, then ramp up if all goes well.

🔧 **How it works:**

- Route 1% of traffic to the new version.
- Monitor metrics (errors, latency).
- Gradually increase to 5% → 25% → 100%.

✅ **Pros:**

- Safest way to deploy sensitive changes.
- Easy to rollback early if issues are detected.

❌ **Cons:**

- Needs traffic splitting logic (K8s + Istio/Linkerd, NGINX with Lua, AWS ALB rules).

📄 **Kubernetes + Istio Example:**

```
# VirtualService to split traffic 90/10
spec:
  http:
    - route:
        - destination:
            host: myapp
            subset: v1
          weight: 90
        - destination:
            host: myapp
            subset: v2
```

```
    weight: 10
```

Ramp this up over time via CI/CD pipeline steps.

🔹 4. Shadow Deployment

Run the new version in the background and mirror real traffic to it, but don't return responses.

🔧 **How it works:**

- Requests are duplicated.
- Only the original version responds.
- New version logs, runs, and is monitored silently.

✅ **Pros:**

- Catch bugs before exposing to users.
- Great for schema changes, ML models.

❌ **Cons:**

- Needs traffic mirroring capabilities (Envoy, Istio).

🔹 5. A/B Testing Deployment

Use feature flags or routing to test two or more versions of a feature **for targeted user segments**.

🔧 **How it works:**

- Users are split by region, user ID hash, browser type, etc.
- Test and collect metrics independently.
- Roll out winner globally.

📄 **Example: Feature Flag in Code**

```
if (featureFlags.newBillingPage) {
  showNewPage();
```

```
} else {
  showLegacyPage();
}
```

Use tools like:

- LaunchDarkly
- Unleash
- Flagsmith
- Azure App Configuration (.NET)

🚨 Rollback Strategy for All Deployments

Always make rollback **automatic and fast**. For example:

GitHub Actions Step:

```
- name: Rollback if failure
  if: failure()
  run: kubectl rollout undo deployment/myapp
```

Or rollback blue-green by flipping the route back.

🧠 When to Use What?

Strategy	Use Case
Rolling	Standard K8s apps, small safe updates
Blue-Green	Major version jumps, config revamp
Canary	High-traffic, critical production apps
Shadow	Testing dangerous migrations or new logic
A/B Testing	UI/UX changes, pricing experiments

📌 8. Monitoring, Alerts & Rollbacks

A zero-downtime deployment **doesn't end when you ship code** — it ends when you're **sure it's running perfectly**. This section ensures you know the moment something breaks, and more importantly, can react *before users even notice*.

🔹 1. Health Checks

Before and after deployment, health checks tell your orchestrator or load balancer whether an app instance is ready to serve.

Kubernetes Example:

```
livenessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 10
```

- livenessProbe: Is the app alive?
- readinessProbe: Is the app *ready* to serve traffic?

🟢 Only route traffic to instances passing **readiness probes**.

🔹 2. Centralized Logging

Don’t check logs on each pod or VM. Aggregate everything.

Tools:

- ELK Stack (Elasticsearch + Logstash + Kibana)
- Loki + Grafana
- AWS CloudWatch Logs
- Azure Monitor

Log format should be structured:

```
{
 "timestamp": "2025-04-10T18:42:11Z",
 "level": "ERROR",
 "service": "billing-service",
 "message": "Payment gateway timeout",
 "user_id": 321
}
```

Then you can search across services using filters like:

```
level:ERROR AND service:billing-service
```

🔹 3. Application Metrics

Real-time dashboards help answer:

- Are errors spiking?
- Is latency increasing?
- Are requests failing?

Tools:

- Prometheus + Grafana (self-hosted)
- DataDog, New Relic, Sentry (managed)
- AWS CloudWatch Metrics

Example Prometheus Query:

```
rate(http_requests_total{status=~"5.."}[1m])
```

→ Tracks 500 errors over the last minute.

Grafana Dashboard Panels:

- Request success/failure rate
- Latency (p95, p99)

- App throughput (req/sec)
- Pod/container restarts

🔹 4. User Behavior Monitoring

It’s not enough to know your servers are fine. Are your users seeing issues?

Tools:

- Sentry (Frontend+Backend error tracking)
- FullStory / Hotjar (Session replay)
- PostHog / Mixpanel (Event analytics)

These tools show:

- JS errors after deployment
- Drop in click-through or checkout rates
- Rage clicks (frustrated users)

🔹 5. Automated Alerts

Set thresholds to trigger real-time alerts when something goes wrong.

Alert Channels:

- Slack / Microsoft Teams
- PagerDuty
- Email / SMS

Alerting Example (Prometheus AlertManager):

```
groups:
- name: app-alerts
  rules:
  - alert: HighErrorRate
    expr: rate(http_requests_total{status=~"5.."}[5m]) > 0.05
```

```
for: 1m
labels:
  severity: critical
annotations:
  summary: "High error rate on production"
```

🔹 6. Deployment Monitoring in CI/CD

CI/CD tools can track deployment health post-deploy:

- GitHub Actions → Use checks or wait-for-deployment steps.
- ArgoCD / Flux → Sync status and rollback if degraded.
- Jenkins → Custom build stages for canary observability.

GitHub Action: Wait for Health Check

```
- name: Wait for Healthy Deployment
  run: kubectl rollout status deployment/my-app
```

🔹 7. Automatic Rollbacks

If error rates spike or health checks fail, rollback immediately.

Kubernetes Example:

```
kubectl rollout undo deployment myapp
```

GitHub Actions:

```
- name: Rollback on Failure
  if: failure()
  run: kubectl rollout undo deployment/myapp
```

Or integrate rollback into tools like:

- **ArgoCD** (Auto-sync with rollback)
- **Spinnaker**

- **GitLab Auto DevOps**

🔹 8. Post-Deployment Verification

After deployment:

- Run synthetic tests
- Check test transactions
- Validate key user flows

Tools:

- Cypress / Playwright (end-to-end testing)
- Postman Monitor (API testing)
- Selenium Grid for UI

You can automate these checks in your pipeline:

```
- name: Run Post-Deployment E2E Tests
  run: npx cypress run --record
```

✅ Conclusion: Mastering Zero Downtime Deployments

Achieving **zero downtime deployments** isn't magic — it's the result of combining smart architecture, reliable automation, and a strong rollback safety net. It's about planning for failure, deploying with caution, and validating that everything works *before* your users even notice a change.

Here's what you've now mastered:

1. **Immutable Infrastructure** – No more patching servers in place.
2. **Service Health Checks** – Know when your app is truly ready.
3. **Traffic Management with Load Balancers** – Seamless routing, seamless rollbacks.
4. **CI/CD Pipelines** – Automate everything with confidence.
5. **Feature Flags** – Decouple deployments from releases.
6. **Safe Database Changes** – No more locking tables or corrupting data.
7. **Smart Deployment Strategies** – Blue-Green, Rolling, Canary — use the right tool for the right job.
8. **Monitoring & Rollbacks** – Catch and fix issues in real-time.

With these tools and practices, you're not just deploying code — you're deploying *safely, confidently, and continuously*.