Blue-Green vs Canary Deployments

1. The Real Problem This Exists to Solve

Deploying new code to production is risky. A single bug can take down your entire service, affecting all users simultaneously. Traditional deployment strategies are all-or-nothing: you either run the old version or the new version. If the new version has issues, you're racing against time to rollback before too much damage occurs.

Real production scenario:

- Deploy new version of payment service
- New version has critical bug: double-charges customers
- **Traditional deployment (all-at-once):**
  - All servers updated simultaneously
  - 100% of traffic hits buggy version
  - 10,000 customers double-charged in 5 minutes
  - Emergency rollback takes 10 minutes
  - 15 minutes of total impact
  - Financial damage, customer trust lost, refunds required
  - PR crisis, potential regulatory issues
- **Blue-Green deployment:**
  - New version deployed to "green" environment
  - Old version still running in "blue" environment
  - Run smoke tests against green
  - Bug detected before routing traffic
  - Switch back to blue (instant)
  - Zero customer impact
  - No double-charges, no refunds, no crisis
- **Canary deployment:**
  - New version deployed alongside old version
  - 1% of traffic routed to new version
  - Bug detected: 100 customers affected
  - Automatic rollback triggered
  - Remaining 99% of customers unaffected
  - Minimal damage: 100 customers vs 10,000

The fundamental problem: All-or-nothing deployments create binary outcomes (total success or catastrophic failure). Blue-green deployments eliminate downtime and enable instant rollback. Canary deployments detect issues early by exposing only a small percentage of traffic to new code, containing the blast radius of bugs.

Without blue-green/canary:

- Simultaneous update of all servers
- All users affected by bugs immediately
- Slow rollback (redeploy old version)
- High-risk deployments
- Downtime during rollback

With blue-green/canary:

- Parallel environments or gradual rollout
- Limited user exposure during validation

- Instant rollback (flip traffic switch)
- Low-risk deployments
- Zero downtime

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: Rolling Update Without Health Checks (Silent Failures)

```
// Incorrect: Deploy to servers one by one without health validation
async function rollingUpdate(servers: string[], newVersion: string) {
  for (const server of servers) {
    await deployToServer(server, newVersion);
    await sleep(5000); // Wait 5 seconds, assume it's ready
    // No health check!
  }
}

// Deploy continues even if servers are failing
```

Why it seems reasonable:

- Simple linear progression
- Gradual rollout (one server at a time)
- Time buffer between deployments

How it breaks:

```
Deployment timeline:
T0: Deploy to server-1 (crashes immediately)
T5: Deploy to server-2 (crashes immediately)
T10: Deploy to server-3 (crashes immediately)
...
T50: All 10 servers deployed and crashed

Problem:
- No health validation
- Crashes not detected
- Deployment continues to completion
- Entire fleet down
- Manual intervention required

Load balancer:
- Sends traffic to crashed servers
- 100% error rate
- No working servers remain
```

Production symptoms:

```
Monitoring:
15:00 - Deployment started
15:01 - Server-1: 500 errors spike
15:02 - Server-2: 500 errors spike
```

```
15:03 - Server-3: 500 errors spike
15:10 - All servers reporting errors
15:10 - Load balancer: No healthy targets
15:10 - Service completely down

"Why didn't the deployment stop when servers started failing?"
"Because we didn't check..."
```

❌ Incorrect Approach #2: Blue-Green Without Pre-Traffic Validation (Deploy and Hope)

```
// Incorrect: Switch traffic immediately after deployment
async function blueGreenDeploy() {
  // Deploy new version to green environment
  await deployToGreen(newVersion);

  // Immediately switch traffic
  await loadBalancer.switchToGreen(); // DANGER!

  // Hope everything works...
}
```

Why it seems reasonable:

- Fast deployment
- No waiting time
- Simple process

How it breaks:

```
Blue environment: Running v1.0 (stable)
Green environment: Deploy v2.0
Load balancer: Switch 100% traffic to green

Immediate failures:
- Database migration didn't complete
- Config file missing
- New dependency not installed
- Service crashes on startup

Result:
- Instant 100% outage
- All traffic hitting broken green environment
- Blue still healthy but not receiving traffic
- "Instant rollback" benefit lost because we switched too early
```

Production symptoms:

```
15:00:00 - Green deployment complete
15:00:05 - Traffic switched to green
15:00:06 - Error rate: 100%
15:00:07 - "Oh no, green is broken!"
15:00:08 - Switch back to blue
```

```
15:00:10 - Service restored

Postmortem:
"We had blue-green deployment but didn't validate green before switching.
Total outage: 5 seconds (could have been much worse).
Blue-green should prevent outages, not cause them."
```

❌ Incorrect Approach #3: Canary Without Automated Rollback (Manual Monitoring)

```
// Incorrect: Route 10% to canary, manually watch metrics
async function canaryDeploy() {
  await deployCanary(newVersion);

  // Route 10% traffic to canary
  await loadBalancer.setWeight('canary', 10);

  console.log('Canary deployed. Please monitor metrics and rollback manually if
needed.');
  // Human must watch metrics and take action
}
```

Why it seems reasonable:

- Gradual rollout
- Safety net (only 10% affected)
- Human judgment for go/no-go

How it breaks:

```
Canary deployment:
15:00 - Deploy canary, route 10% traffic
15:05 - Canary error rate: 50% (vs 0.1% baseline)
15:10 - Engineer notices errors in dashboard
15:12 - Engineer investigates
15:15 - Engineer decides to rollback
15:16 - Rollback initiated
15:20 - Rollback complete

Problem:
- 20 minutes of 10% traffic hitting broken canary
- 10% of users × 20 minutes = significant impact
- Human in the loop = slow response time

What if engineer is in meeting?
What if engineer is asleep (night deployment)?
What if multiple metrics need to be checked?

Canary defeats the purpose if rollback is manual and slow.
```

Production symptoms:

```
Slack:
15:05 - Monitoring bot: "⚠️ Canary error rate elevated"
15:07 - Monitoring bot: "🚨 Canary error rate critical"
15:10 - Engineer: "Investigating..."
15:12 - Engineer: "Rolling back..."
15:16 - Engineer: "Rollback in progress..."
15:20 - Engineer: "Issue resolved."

Incident report:
"Canary deployment took 20 minutes to rollback.
10% of traffic affected.
Impact: 12,000 failed requests.
Root cause: No automated rollback. Manual intervention required."
```

❌ Incorrect Approach #4: Percentage-Based Canary for Stateful Services (Data Inconsistency)

```
// Incorrect: Canary with sticky sessions breaking writes
app.post('/api/update-profile', async (req, res) => {
  // User might be on old version or canary version randomly
  await updateProfile(req.user.id, req.body);
  res.json({ success: true });
});

// Problem: User on old version writes data
// Then user on canary version reads data
// Schema mismatch if canary expects different format!
```

Why it seems reasonable:

- Standard canary approach
- Works for stateless services
- Random traffic distribution

How it breaks:

```
Old version (90%): Expects profile schema v1
  { name: string, email: string }

Canary version (10%): Expects profile schema v2
  { firstName: string, lastName: string, email: string }

User request flow:
1. User loads page (hits old version)
2. User submits form
3. Form data written by old version (schema v1)
4. Page refreshes (hits canary by chance)
5. Canary tries to read data (expects schema v2)
6. Fields missing/null: firstName, lastName
7. UI breaks, validation errors
```

```
Worse scenario (writes):
1. User on canary writes data (schema v2)
2. User on old version reads data
3. Old version can't parse new schema
4. Data corruption or crash
```

Production symptoms:

```
Bug reports:
"My first name disappeared!"
"Profile page showing errors."
"Can't update my information."

Database:
- Some users have schema v1: { name: "John Doe" }
- Some users have schema v2: { firstName: "Jane", lastName: "Smith" }
- Inconsistent data across user base

Incident:
"Canary deployment caused data inconsistency.
10% of users have corrupted profiles.
Requires data migration script to fix."
```

❌ Incorrect Approach #5: No Rollback Plan (One-Way Door)

```
// Incorrect: Deploy canary without ability to rollback
async function canaryDeploy() {
  // Run database migration (destructive)
  await db.query('ALTER TABLE users DROP COLUMN old_field');

  // Deploy canary
  await deployCanary(newVersion);

  // If canary fails, we can't rollback! Old code expects old_field.
}
```

Why it seems reasonable:

- Clean migrations
- Remove deprecated fields
- Move forward, not backward

How it breaks:

```
Deployment:
1. Drop old_field from database
2. Deploy canary (uses new_field)
3. Canary has bug, needs rollback
4. Rollback to old version
5. Old version tries to read old_field
6. old_field doesn't exist!
7. Old version crashes
```

```
8. Can't rollback successfully
9. Stuck with broken canary

"We have a rollback button but it doesn't work!"
"Because the database migration is irreversible."

Solution requires:
- Restore database from backup
- Manually fix data
- Downtime required
- Defeats purpose of safe deployment
```

3. Correct Mental Model (How It Actually Works)

Blue-Green Deployment

Two identical production environments exist: **blue** (current) and **green** (new). Traffic points to blue. Deploy to green, validate, then switch traffic. If issues arise, instantly switch back to blue.

```
Initial state:
Blue (v1.0): ████████ 100% traffic
Green: [idle]

Deploy new version:
Blue (v1.0): ████████ 100% traffic
Green (v2.0): [deployed, running tests]

Validate green:
✓ Health checks pass
✓ Smoke tests pass
✓ Database migrations applied

Switch traffic:
Blue (v1.0): [idle, kept running]
Green (v2.0): ████████ 100% traffic

If issue detected:
Blue (v1.0): ████████ 100% traffic (switch back)
Green (v2.0): [investigating]
```

Canary Deployment

New version deployed alongside old version. Small percentage of traffic (1-5%) routed to canary. Monitor metrics. If metrics healthy, gradually increase percentage. If metrics degraded, automatic rollback.

```
Initial state:
Production (v1.0): ████████████████ 100%

Deploy canary:
Production (v1.0): ███████████████ 95%
Canary (v2.0): █ 5%
```

```
Monitor metrics (5 minutes):
Error rate: v1.0 (0.1%) vs v2.0 (0.1%) ✓
Latency: v1.0 (50ms) vs v2.0 (52ms) ✓
CPU: v1.0 (40%) vs v2.0 (42%) ✓

Increase canary:
Production (v1.0): ██████████ 50%
Canary (v2.0): ██████████ 50%

Full rollout:
Production (v1.0): [retired]
Canary (v2.0): ████████████████ 100%

If metrics fail:
Production (v1.0): ████████████████ 100% (automatic rollback)
Canary (v2.0): [killed]
```

4. Correct Design & Algorithm

Blue-Green Algorithm

```
1. Deploy to green environment
2. Run health checks against green
3. Run smoke tests against green
4. If all pass: Switch traffic to green
5. If any fail: Keep traffic on blue, investigate
6. Monitor green with 100% traffic
7. If issue detected: Instant switch back to blue
8. After stabilization period: Retire blue
```

Canary Algorithm

```
1. Deploy canary alongside production
2. Route small percentage (1-5%) to canary
3. Monitor metrics for 5-10 minutes:
   - Error rate (should be ≤ baseline)
   - Latency p50/p99 (should be ≤ baseline + threshold)
   - Resource usage (should be reasonable)
4. If metrics healthy: Increase canary percentage (10% → 25% → 50% → 100%)
5. If metrics unhealthy: Automatic rollback to 0%
6. Repeat until 100% canary
7. Retire old version
```

Automated Rollback Decision

```
function shouldRollback(canaryMetrics, baselineMetrics): boolean {
  const errorRateThreshold = 1.5; // 50% increase
  const latencyThreshold = 1.2; // 20% increase

  if (canaryMetrics.errorRate > baselineMetrics.errorRate * errorRateThreshold) {
    return true; // Too many errors
  }

  if (canaryMetrics.p99Latency > baselineMetrics.p99Latency * latencyThreshold) {
    return true; // Too slow
  }

  return false;
}
```

5. Full Production-Grade Implementation

```
import express from 'express';
import { Pool } from 'pg';

/**
 * Deployment environment
 */
interface Environment {
  name: string;
  version: string;
  servers: string[];
  healthy: boolean;
}

/**
 * Traffic routing configuration
 */
interface TrafficConfig {
  blueWeight: number;
  greenWeight: number;
  canaryWeight: number;
}

/**
 * Metrics for deployment validation
 */
interface Metrics {
  errorRate: number;
  p50Latency: number;
  p99Latency: number;
  requestCount: number;
  cpuUsage: number;
  memoryUsage: number;
}

/**
 * Blue-Green deployment manager
 */
class BlueGreenDeployment {
```

```typescript
  private blue: Environment;
  private green: Environment;
  private active: 'blue' | 'green' = 'blue';

  constructor(
    private loadBalancer: LoadBalancer,
    private healthChecker: HealthChecker
  ) {
    this.blue = { name: 'blue', version: '1.0.0', servers: [], healthy: true };
    this.green = { name: 'green', version: '', servers: [], healthy: false };
  }

  /**
   * Deploy new version to inactive environment
   */
  async deploy(newVersion: string): Promise<void> {
    const targetEnv = this.active === 'blue' ? this.green : this.blue;

    console.log(`[Blue-Green] Deploying ${newVersion} to ${targetEnv.name}`);

    try {
      // Step 1: Deploy to inactive environment
      await this.deployToEnvironment(targetEnv, newVersion);

      // Step 2: Health checks
      console.log(`[Blue-Green] Running health checks on ${targetEnv.name}`);
      const healthChecksPassed = await
this.healthChecker.checkEnvironment(targetEnv);

      if (!healthChecksPassed) {
        throw new Error(`Health checks failed for ${targetEnv.name}`);
      }

      // Step 3: Smoke tests
      console.log(`[Blue-Green] Running smoke tests on ${targetEnv.name}`);
      const smokeTestsPassed = await this.runSmokeTests(targetEnv);

      if (!smokeTestsPassed) {
        throw new Error(`Smoke tests failed for ${targetEnv.name}`);
      }

      // Step 4: Validate with 1% traffic
      console.log(`[Blue-Green] Validating with 1% traffic`);
      await this.loadBalancer.setTraffic({
        blueWeight: this.active === 'blue' ? 99 : 1,
        greenWeight: this.active === 'green' ? 99 : 1,
        canaryWeight: 0,
      });

      await this.sleep(60000); // Monitor for 1 minute

      const metrics = await this.collectMetrics(targetEnv);
      if (this.metricsUnhealthy(metrics)) {
        throw new Error('Metrics degraded during validation');
      }

      // Step 5: Switch traffic
      console.log(`[Blue-Green] Switching traffic to ${targetEnv.name}`);
      await this.switchTraffic(targetEnv.name as 'blue' | 'green');

      // Step 6: Monitor new environment
      console.log(`[Blue-Green] Monitoring ${targetEnv.name} with 100% traffic`);
      await this.sleep(300000); // Monitor for 5 minutes

      const finalMetrics = await this.collectMetrics(targetEnv);
      if (this.metricsUnhealthy(finalMetrics)) {
        console.error(`[Blue-Green] Metrics unhealthy, rolling back!`);
        await this.rollback();
        throw new Error('Deployment failed, rolled back');
      }

      console.log(`[Blue-Green] Deployment successful!`);
      targetEnv.healthy = true;

    } catch (error) {
      console.error(`[Blue-Green] Deployment failed:`, error);
      await this.rollback();
      throw error;
    }
  }

  /**
   * Instant rollback to previous environment
   */
  async rollback(): Promise<void> {
    const previousEnv = this.active === 'blue' ? 'blue' : 'green';
    console.log(`[Blue-Green] Rolling back to ${previousEnv}`);

    await this.switchTraffic(previousEnv);
    console.log(`[Blue-Green] Rollback complete`);
  }

  /**
   * Switch 100% traffic to target environment
   */
  private async switchTraffic(target: 'blue' | 'green'): Promise<void> {
    await this.loadBalancer.setTraffic({
      blueWeight: target === 'blue' ? 100 : 0,
      greenWeight: target === 'green' ? 100 : 0,
      canaryWeight: 0,
    });

    this.active = target;
  }
```

```typescript
  private async deployToEnvironment(env: Environment, version: string):
Promise<void> {
    env.version = version;
    // Simulate deployment
    await this.sleep(5000);
  }

  private async runSmokeTests(env: Environment): Promise<boolean> {
    // Run critical path tests
    return true; // Placeholder
  }

  private async collectMetrics(env: Environment): Promise<Metrics> {
    // Collect real metrics from monitoring system
    return {
      errorRate: 0.05,
      p50Latency: 50,
      p99Latency: 200,
      requestCount: 1000,
      cpuUsage: 40,
      memoryUsage: 60,
    };
  }

  private metricsUnhealthy(metrics: Metrics): boolean {
    return metrics.errorRate > 1.0 || metrics.p99Latency > 500;
  }

  private sleep(ms: number): Promise<void> {
    return new Promise(resolve => setTimeout(resolve, ms));
  }
}

/**
 * Canary deployment manager
 */
class CanaryDeployment {
  private baselineMetrics: Metrics | null = null;
  private canaryMetrics: Metrics | null = null;

  constructor(
    private loadBalancer: LoadBalancer,
    private metricsCollector: MetricsCollector
  ) {}

  /**
   * Deploy canary with automated progressive rollout
   */
  async deploy(newVersion: string): Promise<void> {
    console.log(`[Canary] Deploying ${newVersion}`);
    try {
      // Step 1: Deploy canary servers
      await this.deployCanary(newVersion);

      // Step 2: Collect baseline metrics
      this.baselineMetrics = await this.metricsCollector.collect('production');
      console.log(`[Canary] Baseline metrics:`, this.baselineMetrics);

      // Step 3: Progressive rollout
      const stages = [1, 5, 10, 25, 50, 100];

      for (const percentage of stages) {
        console.log(`[Canary] Routing ${percentage}% traffic to canary`);

        await this.loadBalancer.setTraffic({
          blueWeight: 100 - percentage,
          greenWeight: 0,
          canaryWeight: percentage,
        });

        // Monitor for 5 minutes (or less for small percentages)
        const monitorDuration = Math.min(percentage * 10000, 300000);
        await this.sleep(monitorDuration);

        // Collect canary metrics
        this.canaryMetrics = await this.metricsCollector.collect('canary');
        console.log(`[Canary] Canary metrics at ${percentage}%:`,
this.canaryMetrics);

        // Automated rollback decision
        if (this.shouldRollback()) {
          console.error(`[Canary] Metrics unhealthy at ${percentage}%, rolling
back!`);
          await this.rollback();
          throw new Error('Canary deployment failed, rolled back');
        }

        console.log(`[Canary] Metrics healthy at ${percentage}%`);
      }

      console.log(`[Canary] Deployment successful!`);

    } catch (error) {
      console.error(`[Canary] Deployment failed:`, error);
      throw error;
    }
  }

  /**
   * Automated rollback decision based on metrics
   */
  private shouldRollback(): boolean {
    if (!this.baselineMetrics || !this.canaryMetrics) {
      return false;
    }

    const errorRateThreshold = 1.5; // 50% increase
    const latencyThreshold = 1.2; // 20% increase

    // Error rate comparison
    if (this.canaryMetrics.errorRate > this.baselineMetrics.errorRate *
errorRateThreshold) {
      console.error(
        `[Canary] Error rate too high: ${this.canaryMetrics.errorRate} vs baseline
${this.baselineMetrics.errorRate}`
      );
      return true;
    }

    // Latency comparison
    if (this.canaryMetrics.p99Latency > this.baselineMetrics.p99Latency *
latencyThreshold) {
      console.error(
        `[Canary] Latency too high: ${this.canaryMetrics.p99Latency}ms vs baseline
${this.baselineMetrics.p99Latency}ms`
      );
      return true;
    }

    // Resource usage comparison
    if (this.canaryMetrics.cpuUsage > 90 || this.canaryMetrics.memoryUsage > 90) {
      console.error(`[Canary] Resource usage too high`);
      return true;
    }

    return false;
  }

  /**
   * Instant rollback to 0% canary traffic
   */
  async rollback(): Promise<void> {
    console.log(`[Canary] Rolling back to 0% canary traffic`);

    await this.loadBalancer.setTraffic({
      blueWeight: 100,
      greenWeight: 0,
      canaryWeight: 0,
    });

    // Kill canary servers
    await this.killCanary();

    console.log(`[Canary] Rollback complete`);
  }

  private async deployCanary(version: string): Promise<void> {
    // Deploy canary servers
    await this.sleep(5000);
  }

  private async killCanary(): Promise<void> {
    // Terminate canary servers
    await this.sleep(1000);
  }

  private sleep(ms: number): Promise<void> {
    return new Promise(resolve => setTimeout(resolve, ms));
  }
}

/**
 * Load balancer abstraction
 */
class LoadBalancer {
  async setTraffic(config: TrafficConfig): Promise<void> {
    console.log(`[LoadBalancer] Setting traffic:`, config);
    // In production: Update Nginx, HAProxy, AWS ALB, etc.
    await this.sleep(1000);
  }

  private sleep(ms: number): Promise<void> {
    return new Promise(resolve => setTimeout(resolve, ms));
  }
}

/**
 * Health checker
 */
class HealthChecker {
  async checkEnvironment(env: Environment): Promise<boolean> {
    console.log(`[HealthChecker] Checking ${env.name}`);

    for (const server of env.servers) {
      const healthy = await this.checkServer(server);
      if (!healthy) {
        console.error(`[HealthChecker] Server ${server} unhealthy`);
        return false;
      }
    }

    return true;
  }

  private async checkServer(server: string): Promise<boolean> {
    // Check /health endpoint
    // Check database connectivity
    // Check dependencies
    return true; // Placeholder
  }
}

/**
 * Metrics collector
 */
class MetricsCollector {
  async collect(environment: string): Promise<Metrics> {
    // Collect from Prometheus, CloudWatch, Datadog, etc.
    return {
      errorRate: Math.random() * 0.5,
      p50Latency: 50 + Math.random() * 20,
      p99Latency: 200 + Math.random() * 100,
      requestCount: 10000,
      cpuUsage: 40 + Math.random() * 20,
      memoryUsage: 60 + Math.random() * 20,
    };
  }
}

// Usage example
const loadBalancer = new LoadBalancer();
const healthChecker = new HealthChecker();
const metricsCollector = new MetricsCollector();

const blueGreen = new BlueGreenDeployment(loadBalancer, healthChecker);
const canary = new CanaryDeployment(loadBalancer, metricsCollector);

// Blue-Green deployment
// await blueGreen.deploy('2.0.0');

// Canary deployment
// await canary.deploy('2.1.0');
```

6. Correct Usage Patterns (Where This Shines)

When to Use Blue-Green

- **Zero downtime requirement**: Banking, e-commerce
- **Instant rollback needed**: High-stakes deployments
- **Database migrations**: Can test migrations on green first

When to Use Canary

- **High-traffic services**: Millions of requests/second
- **New features**: Validate before full rollout
- **A/B testing**: Compare old vs new

7. Failure Modes & Edge Cases

Blue-Green: Database Migrations

Problem: Green uses new schema, blue uses old schema.

Solution: Backward-compatible migrations (expand-migrate-contract pattern).

Canary: Session Affinity

Problem: User switches between old/new versions mid-session.

Solution: Sticky sessions (route same user to same version).

Canary: Low Traffic

Problem: 1% of low traffic = not enough data to validate.

Solution: Use absolute traffic (100 requests) instead of percentage.

8. Performance Characteristics & Tradeoffs

Blue-Green

- **Infrastructure cost:** 2× capacity (both environments running)
- **Deployment time:** Fast (parallel environments)
- **Risk:** Medium (100% traffic switch at once)

Canary

- **Infrastructure cost:** 1.05× capacity (small canary overhead)
- **Deployment time:** Slow (gradual rollout)
- **Risk:** Low (limited exposure)

9. Foot-Guns & Common Mistakes (DO NOT SKIP)

Mistake 1: No Health Checks Before Traffic Switch

Fix: Always validate green before routing traffic.

Mistake 2: Manual Canary Monitoring

Fix: Automated rollback based on metrics.

Mistake 3: Stateful Canary Without Session Affinity

Fix: Use sticky sessions or avoid canary for stateful services.

Mistake 4: No Rollback Plan

Fix: Backward-compatible changes, test rollback procedure.

Mistake 5: Insufficient Monitoring

Fix: Track error rate, latency, resource usage per environment.

10. When NOT to Use This (Anti-Patterns)

Blue-Green: Small Services

Overkill for services with <100 requests/second.

Canary: Breaking Changes

Can't canary if new version requires incompatible schema.

Both: Localhost Development

Use these strategies for production, not local development.

11. Related Concepts (With Contrast)

Feature Flags

Difference: Flags toggle features within same deployment, blue-green/canary deploy different versions.

Rolling Update

Difference: Rolling updates replace servers sequentially, blue-green has parallel environments.

Immutable Infrastructure

Related: Blue-green naturally uses immutable infrastructure (new servers, not updated servers).

12. Production Readiness Checklist

Blue-Green

- [ ] Two identical environments (blue, green)
- [ ] Load balancer can switch traffic instantly
- [ ] Health checks on both environments
- [ ] Smoke tests before traffic switch
- [ ] Rollback procedure tested
- [ ] Monitoring per environment

Canary

- [ ] Canary servers alongside production
- [ ] Traffic routing by percentage
- [ ] Automated metrics collection
- [ ] Automated rollback based on metrics
- [ ] Progressive rollout stages (1% → 5% → 10% → 100%)
- [ ] Session affinity for stateful services

Both

- [ ] Database migrations are backward-compatible
- [ ] Rollback plan documented and tested
- [ ] Monitoring dashboards per environment
- [ ] Alerting for deployment failures

- [ ] Runbook for emergency rollback