Branching Strategy

Part 1: Introduction – Why Branching Strategy Matters in DevOps

"A bad branching strategy is like trying to build a skyscraper on a swamp."

🔹 What Is a Branching Strategy?

A **branching strategy** is a defined process and set of rules that your development team follows to manage changes to your codebase. It's not just a Git practice; it's a strategic approach that directly influences:

- **Release velocity**
- **Quality assurance**
- **Rollback capabilities**
- **Disaster recovery readiness**
- **Team collaboration**

In modern **DevOps culture**, where **automation, CI/CD, and speed** are crucial, an **intelligent and structured branching strategy** becomes a key pillar for high-performing software teams.

🔹 Why Is Branching Important in DevOps?

Let's look at it through a DevOps lens:

DevOps Concern	Impact of Branching Strategy
Continuous Integration	Enables parallel development without breaking builds
Continuous Delivery	Enables smooth progression through Dev, QA, PPD, to PROD
Rollbacks	Enables you to revert back to stable branches or hotfixes
Collaboration	Avoids conflicts by using isolated feature/bugfix/hotfix branches
Testing	Aligns branches with test environments like qa, ppd
Release Management	Controlled movement of tested code to production
DR (Disaster Recovery)	Using dr branch to simulate and test failovers

🔹 Real-World Scenario Without a Good Branching Strategy

Imagine a team with no branching strategy:

- All developers commit to a main branch.
- QA doesn't have their own isolated test codebase.
- Production bugs are fixed directly on live servers.
- DR is just a theory—there's no separate recovery environment.

This team will **face outages, production failures, and data loss**. Their velocity will also be crippled by firefighting rather than shipping features.

Now contrast that with a team that uses:

- Isolated feature branches → Safe dev.
- qa branch → Stable environment for testers.
- ppd → A mirror of prod with real configs.
- main → Locked, release-controlled production.
- dr → Always ready for simulated failovers.

That team ships features faster, with fewer bugs, and has peace of mind.

Part 2: Core Environments & Git Branches

In a mature DevOps setup, **environments and Git branches mirror each other**. This tight coupling ensures code integrity, traceability, and environmental parity from Dev to Production.

We'll explore both environments and their associated branches:

🌍 1. Development (Dev) Environment → dev Branch

🔸 **Purpose:**

The **Dev** environment is where all active development happens. It is **highly volatile**—bugs are expected, features are half-done, and merges happen frequently.

🔸 **Associated Branch: dev**

This is the central branch where all **feature**, **bugfix**, and **hotfix** branches are merged **after peer review and CI validation**.

🔸 **Characteristics:**

- Auto-deployed via CI/CD on every dev branch push.
- Frequent code merges.
- Often integrated with **internal tools, mocks, or stubs**.
- Logs and observability might be basic.

🔸 **Example:**

```
git checkout -b feature/user-authentication

# work work work

git commit -am "Add login logic"

git push origin feature/user-authentication


# Open PR → Review → Merge to dev

git checkout dev

git merge feature/user-authentication

git push origin dev
```

---

2. QA Environment → qa Branch

🔸 **Purpose:**

This is the **quality assurance gate**. All builds from dev are tested here by **QA engineers**—both manual and automated test suites run here.

🔸 **Associated Branch: qa**

This branch is where **tested and review-ready code from dev** is merged and deployed.

🔸 **Characteristics:**

- Stable but **not production-ready**.
- Test data may be anonymized from production.
- Usually connected to tools like Selenium, TestNG, or custom test runners.

🔸 **Movement:**

Once dev is stable:

git checkout qa

git merge dev

git push origin qa

A CI/CD pipeline then auto-deploys this to the QA environment.

🔬 3. Pre-Production (PPD) Environment → ppd Branch

🔸 **Purpose:**

Pre-Production (or **Staging**) is an **exact mirror of production**.

It's used to:

- Test configurations and infra.
- Perform performance/load tests.
- Conduct business UAT (User Acceptance Testing).
- Simulate **release freezes** and **cut-off dates**.

🔸 **Associated Branch: ppd**

Only **code that passed QA** makes it here. This is usually frozen during release windows.

🔸 **Characteristics:**

- Uses real services, real third-party integrations.
- Simulates real-world traffic.
- Data is masked but very close to PROD.
- Requires change control approval to deploy.

🔸 **Movement:**

```
# From QA to PPD
git checkout ppd
git merge qa
git push origin ppd
```

🚀 4. Production (PROD) Environment → main Branch

🔸 **Purpose:**

The **holy grail** of environments. No changes reach PROD unless:

- They've passed QA and PPD testing.
- They've been tagged and approved.
- Change management (RFCs) has cleared them.

🔸 **Associated Branch: main**

This is your **release branch**.

🔸 **Characteristics:**

- Highly protected — requires PR approvals.
- Only CI-approved and tagged commits are merged here.
- Trigger auto-deployments to production.

🔸 **Movement:**

```
# From PPD to MAIN after UAT
git checkout main
git merge ppd
git tag -a v2.3.0 -m "Release April 2025"
git push origin main --tags
```

---

5. Disaster Recovery (DR) Environment → dr Branch

🔸 **Purpose:**

Disaster Recovery simulates the worst-case scenario:

- Production is down.
- You need to restore services from backups or secondary infra.

🔸 **Associated Branch: dr**

This mirrors main (PROD) and is deployed to a secondary DR site or cloud region.

🔸 **Characteristics:**

- Used in **DR drills**.
- May be automatically or manually synced from main.
- Usually has separate cloud infra (multi-region setup).

🔸 **Movement:**

```
# Periodically or post-release
git checkout dr
git merge main
git push origin dr
```

Additional Branch Types

These are temporary or supporting branches used in the software lifecycle:

---

feature/<name>

Used for developing **new functionality** in isolation.

- Branches from: dev
- Merges to: dev
- Naming: feature/login-page, feature/api-throttle

```
git checkout -b feature/login-page dev
# Work and test
git push origin feature/login-page
```

---

bugfix/<name>

Used to fix **non-critical** bugs found in lower environments.

- Branches from: dev or qa
- Merges to: dev or qa
- Naming: bugfix/typo-in-header, bugfix/sorting-error

bash

CopyEdit

git checkout -b bugfix/sorting-error dev

---

hotfix/<name>

Used for **urgent PROD fixes** that need rapid deployment.

- Branches from: main
- Merges to: main, then back to dev, qa, ppd, dr
- Naming: hotfix/login-bug, hotfix/payment-crash

git checkout -b hotfix/login-bug main

Patch → commit → push → PR → merge to main → tag → deploy

After that:

Back merge to all active branches

GIT CHECKOUT DEV && GIT MERGE HOTFIX/LOGIN-BUG

GIT CHECKOUT QA && GIT MERGE HOTFIX/LOGIN-BUG

GIT CHECKOUT PPD && GIT MERGE HOTFIX/LOGIN-BUG

GIT CHECKOUT DR && GIT MERGE HOTFIX/LOGIN-BUG

✅ **Summary Table**

Environment	Branch	Use Case	Who Owns It	CI/CD Trigger
Dev	dev	Active development	Developers	On push
QA	qa	Test-stable code	QA Team	On merge

Environment	Branch	Use Case	Who Owns It	CI/CD Trigger
PPD	ppd	UAT/Pre-release	Leads/Product	On merge
PROD	main	Live deployments	Release Manager	On tag
DR	dr	Disaster readiness	Infra/Platform	On merge from main
Feature	feature/*	New code	Developers	On push
Bugfix	bugfix/*	Bug resolution	Developers	On push
Hotfix	hotfix/*	PROD bugs	Senior Devs	On merge & tag

Part 3: Code Movement Workflow — From Feature to DR (Step-by-Step)

🚦 *"Every line of code should follow a defined highway — from chaos to control, from feature to main, from development to disaster recovery."*

In this part, we'll break down how code flows through each environment, what checks are in place, who's responsible, and how automation is used to ensure **safety, speed, and stability**.

Let's walk through the full lifecycle using the following **example use case**:

✨ You're building a new feature: **"User Profile Page"**

🔹 **Step 1: Branch from dev → Start Work in feature/user-profile**

✅ **Developers Do:**

git checkout dev

git pull origin dev

git checkout -b feature/user-profile

- Work happens in feature/user-profile
- Unit tests are written locally or using TDD.
- Code is pushed to remote.
- A PR is opened to dev.

🔧 **Automation:**

- On push → Run **unit tests**, **lint checks**, **Trivy**, **Gitleaks**, **SonarQube**, etc.
- CI fails if any checks don't pass.

🔐 **Checks Before Merge to dev:**

- ✅ Code review approved by 1–2 devs
- ✅ CI checks passed
- ✅ No merge conflicts

🔹 **Step 2: Merge feature/user-profile → dev**

git checkout dev

git merge feature/user-profile

git push origin dev

🔧 **Automation:**

- On merge to dev, trigger Dev Environment CI/CD:
  - Build app
  - Run integration tests
  - Deploy to **Dev namespace/server**
  - Notify developers in Slack/Teams

🔹 **Step 3: Merge dev → qa (Test-Ready Code)**

When?

After all active features for a release are merged to dev, a **release candidate** is created and moved to qa.

git checkout qa

git merge dev

git push origin qa

🔧 **Automation:**

- QA pipeline triggers:
  - Build app from qa branch
  - Deploy to **QA environment**
  - Run full test suite (Smoke, Functional, Regression, API)
  - Publish test reports (e.g., Allure)
  - Notify QA team for manual validation

🔹 **Step 4: Merge qa → ppd (Staging/UAT)**

Who initiates this?

- QA Lead or Release Manager — once sign-off is complete.

```
git checkout ppd
git merge qa
git push origin ppd
```

🔧 **Automation:**

- Build & deploy to PPD (Pre-Prod)
- Validate:
  - Load Testing (e.g., JMeter)
  - UAT (User Acceptance Testing)
  - Infra config simulation (Secrets, SSO, Payment Sandboxes)

✅ **Change Freeze Begins Here:**

Only **business-approved, tested** changes can pass this stage.

🔹 **Step 5: Merge ppd → main (Production Release)**

Who does this?

- **Release Manager**, after release approvals (CAB meetings, changelogs, etc.)

```
git checkout main
git merge ppd
git tag -a v2.3.0 -m "User Profile Release - April 2025"
git push origin main --tags
```

🔧 **Automation:**

- Tag triggers the PROD CI/CD pipeline:
  - Build artifact (Docker, Jar, etc.)
  - Security scan again (Shift Left + Final Gate)
  - Deploy to PROD using:
- Blue-Green deployment

- Canary release
- Rolling update

🔹 **Step 6: Merge main → dr (Disaster Recovery)**

Why?

To **synchronize DR** with the latest production code.

git checkout dr

git merge main

git push origin dr

🔧 **Automation:**

- DR environment updated.
- DR runbooks tested:
  - Can this version be restored in another region?
  - Is failover working?

📈 **Summary: Code Flow Diagram (Text-Based)**

```
[feature/*] → dev → qa → ppd → main → dr
                ↑     ↑     ↑     ↑
            bugfix/*  |     |     |
                      hotfix/* →→→→→→
```

📌 **Notes on Merge Policies**

Merge From	Merge To	When	Who	Notes
feature/*	dev	Daily	Developers	Must pass unit tests
dev	qa	Sprint end	QA Lead	After internal review
qa	ppd	Pre-release	Release Owner	Post QA sign-off
ppd	main	Release Day	Release Manager	Needs tag
main	dr	Post-release	Infra Team	Ensures DR readiness

Hotfix Scenario

A critical bug is found in PROD: “Login page crashes.”

🔥 **Step-by-step Hotfix:**

1. Branch from main:

```
git checkout main
git pull origin main
git checkout -b hotfix/login-crash
```

2. Fix, test locally, commit:

```
git commit -am "Fix null error in login controller"
```

3. Push and PR:

```
git push origin hotfix/login-crash
```

4. Merge to main, tag release:

```
git checkout main
git merge hotfix/login-crash
git tag -a v2.3.1 -m "Login hotfix"
git push origin main --tags
```

5. **Back-merge** hotfix to all:

```
git checkout dev && git merge hotfix/login-crash && git push
git checkout qa && git merge hotfix/login-crash && git push
git checkout ppd && git merge hotfix/login-crash && git push
git checkout dr && git merge hotfix/login-crash && git push
```

6. Let automation take care of deployments.

⚙️ **Automation Matrix (CI/CD Triggers)**

Branch	Trigger	Actions
dev	On push	Unit tests, Build, Deploy to Dev
qa	On merge	Build, QA Deploy, Run test suite

Branch	Trigger	Actions
ppd	On merge	Build, Staging Deploy, Load/UAT
main	On tag	Secure Build, Production Deploy
dr	On merge	DR deploy, run DR simulation
feature/*	On push	Lint, Unit test, PR check
hotfix/*	On push & merge	CI+Deploy (if urgent), tagging required

Part 4: QA Strategies, Feature Flags, and Testing Workflows in Git Branching

"Testing isn't just about finding bugs; it's about proving that every change is safe to deliver."

This section will cover:

- QA workflow integrated into branches
- Different types of testing at each branch/environment
- Role of feature flags in progressive delivery
- How automation and manual QA are blended
- Git practices for testable code

🔸 1. QA Philosophy in DevOps Branching Strategy

Modern DevOps is built on **Shift-Left Testing**: the idea that quality is **everyone's job**, and testing should begin **as early as development**.

In our branching strategy:

- Developers test in dev via CI
- QA tests on qa branch
- Business & infra validate on ppd
- Automated release checks before main
- DR is tested in dr

This **multi-layer testing strategy** prevents:

- Unvalidated code from leaking into production
- Flaky features from impacting users
- Bugs from snowballing into production issues

🔹 2. Testing Types Mapped to Branches

Branch	Environment	Test Types	Automation?	Manual?
feature/*	Local/Dev	Unit tests, Static Analysis	✅	❌
dev	Dev	Linting, Trivy, Gitleaks, Unit + API tests	✅	❌
qa	QA	Regression, Integration, UI, API tests	✅	✅
ppd	Staging	UAT, Load Testing, Security Scan	✅	✅
main	PROD	Pre-release validation, Smoke	✅	✅ (Post)
dr	DR	Disaster Simulation Tests	✅	✅

---

3. Unit Tests: Feature Branch to Dev

Unit testing begins **in the feature branches**. Every PR to dev must pass:

- Unit test suite
- Code coverage threshold (e.g., 80%)
- Static code analysis (e.g., SonarQube)

Example pipeline step

npm test -- --coverage

or

mvn clean test jacoco:report

Developers are trained to write testable code **alongside logic**, using TDD/BDD frameworks.

🔍 4. Security Scanning and Secrets Detection

Every feature, dev, and hotfix branch goes through:

- **Trivy**: Checks Dockerfile and dependencies for CVEs
- **Gitleaks**: Detects secrets (tokens, passwords) in code
- **SonarQube**: Static code analysis for vulnerabilities, bugs, code smells

CI Example:

```
jobs:
  trivy_scan:
    runs-on: ubuntu-latest
    steps:
      - uses: aquasecurity/trivy-action@v0.0.14
        with:
          image-ref: my-app:latest
```

---

5. Integration Testing: qa Branch

When dev is merged into qa, the CI/CD pipeline triggers:

- Test environment deployment
- Integration test execution
- UI testing via Selenium/Cypress
- API test via Postman or RestAssured
- Validation of test cases stored in Zephyr/TestRail

Slack alerts notify QA teams with test outcomes.

---

6. Regression Testing in QA

Regression is key in qa:

- Verifies that new features didn't break existing functionality
- Often runs nightly (cron or GitHub scheduled workflows)
- May use Jenkins pipelines, GitHub Actions matrix builds

Tools:

- Selenium Grid
- Cucumber
- PyTest
- Allure for reporting

7. Feature Flags in dev and qa

“Release code to production without exposing it to users.”

Feature flags allow:

- Merging incomplete features safely
- Enabling/disabling features per environment or user
- Gradual rollout (progressive delivery)

Tools: **LaunchDarkly, Unleash, ConfigCat, Flagsmith**

```
if (featureFlag.isEnabled('newCheckout')) {
  renderNewCheckout();
} else {
  renderOldCheckout();
}
```

Flags are controlled via dashboards. Teams can:

- Turn on in qa, off in main
- Do A/B testing in ppd
- Rollback by disabling, not code reverting

---

8. Manual QA Testing

Even in automation-heavy environments, manual testing is essential for:

- Exploratory testing
- User experience checks
- Business logic validation

Manual QA occurs mostly in:

- qa (for active testing)
- ppd (for UAT by product team)
- dr (to simulate failover procedures)

Test cases and results are documented in QA systems like:

- Zephyr
- TestRail
- Xray

---

9. Release Candidate Workflow

When QA signs off:

git checkout ppd

git merge qa

git push origin ppd

This generates a **Release Candidate (RC)** build:

- RC build is tagged (e.g., rc-v2.3.0)
- Business users validate in PPD
- Release manager signs off

Example CI Tagging:

```
jobs:
  tag_rc:
    runs-on: ubuntu-latest
    steps:
      - name: Tag RC
        run: git tag -a rc-v2.3.0 -m "RC for user profile"
```

---

10. Load Testing in ppd

Here, load testing tools like **JMeter**, **Locust**, or **k6** are used:

- Simulate thousands of users
- Validate auto-scaling behavior
- Identify bottlenecks

CI/CD tools like Jenkins or GitHub Actions are configured to run load test jobs post-deployment to ppd.

---

11. User Acceptance Testing (UAT)

In Pre-Production:

- Real business workflows are tested
- Datasets mirror production

- Product team executes checklist-based UAT

If accepted:

- RC is promoted to **release tag** (e.g., v2.3.0)
- Merged to main

---

12. Final Pre-Prod Checks Before PROD Merge

Before merging to main, CI runs:

- Final security scans
- GitLeaks for secrets
- Change Freeze Checks
- Notification to Change Advisory Board (CAB)

Merge looks like:

```
git checkout main
git merge ppd
git tag -a v2.3.0 -m "User Profile Release"
git push origin main --tags
```

---

13. Testing in hotfix/* Flow

When an urgent issue hits PROD:

- Patch is made in hotfix/login-issue
- Immediate unit + smoke test
- Tag is created (e.g., v2.3.1)
- Rolled out to main

After production deploy, it is back-merged to all:

```
git checkout dev && git merge hotfix/login-issue && git push
git checkout qa && git merge hotfix/login-issue && git push
git checkout ppd && git merge hotfix/login-issue && git push
git checkout dr && git merge hotfix/login-issue && git push
```

---

14. Disaster Recovery Testing

The dr branch is merged with main periodically:

git checkout dr

git merge main

git push origin dr

This triggers:

- DR environment deployment
- Smoke tests on DR infra
- Failover simulation
- DR readiness reports to stakeholders

✅ 15. Summary: QA + Testing Layer Over Branching Strategy

Branch	Tests	Tooling	Triggers
feature/*	Unit, Lint, Coverage	Jest, PyTest, Sonar	Push
dev	Unit + Integration	Trivy, Gitleaks	Push/Merge
qa	Regression, UI, API	Selenium, Cucumber	Merge
ppd	UAT, Load, Performance	JMeter, Unleash	Merge
main	Smoke, Security	Trivy, GitLeaks	Tag
dr	DR Simulations	Infra Scripts	Merge

That's the complete deep-dive into **how QA, testing, and feature flags integrate into your Git branching strategy**.

Part 5: Production Deployment, Git Tagging, Rollbacks & Release Management

🚀 “Production is not a branch. It’s a sacred contract between you, your users, and your business.”

This section dives into:

- 🔒 Best practices for deploying main to PROD
- 🏷️ Git tagging strategies for versioning and traceability
- 🔁 How rollbacks work (automated + manual)
- ✅ Release approvals, documentation, and audit trails
- 🧠 Real-world deployment strategies: blue-green, canary, rolling
- 📋 Team roles, responsibilities, and checkpoints

🔸 1. The main Branch: Your Production Lifeline

The main branch:

- Represents **exactly what is in production**
- Is always deployable
- Is **write-protected**
- Only updated via **approved PRs from ppd or hotfix/***
- Requires **multi-person code reviews**
- Protected by **branch rules and CI/CD checks**

Example branch protection setup:

- ❌ No direct commits
- ✅ Minimum 2 reviewers
- ✅ All tests pass
- ✅ Signed commits

🔹 2. Git Tagging: Semantic Versioning Strategy

Git tags are immutable **snapshots** of your code used for deployments.

Use **Semantic Versioning** (SemVer):

MAJOR.MINOR.PATCH

e.g., v3.1.5

- MAJOR: Breaking changes
- MINOR: New feature, no breaking
- PATCH: Bugfixes or hotfixes

Example Workflow

```
git checkout main
git pull origin main
git tag -a v2.3.0 -m "Release - User Profile Feature"
git push origin v2.3.0
```

This tag:

- Triggers a **release pipeline**
- Deploys that exact state to PROD
- Links with release notes and audit logs

---

3. How Production Deployments Work

Deployments should be:

- ✅ Predictable
- ✅ Safe
- ✅ Reversible
- ✅ Auditable

You can achieve this through **automated CI/CD pipelines** that watch for new tags on main.

Example GitHub Actions trigger:

```
on:
  push:
    tags:
```

```
    - 'v*.*.*'

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout tagged release
        uses: actions/checkout@v3
      - name: Build & Deploy to Prod
        run: ./scripts/deploy-prod.sh
```

---

4. Release Approvals & Governance

Before tagging a release:

- ✅ QA sign-off
- ✅ Business sign-off (UAT)
- ✅ Performance benchmarks met
- ✅ Release notes created
- ✅ CAB (Change Advisory Board) approval, if applicable

Release notes include:

- Jira stories/issues
- Linked PRs
- Git commits
- Known issues

---

5. Blue-Green Deployment with Git Branches

In blue-green strategy:

- Two identical PROD environments (Blue + Green)
- Traffic is routed to one (e.g., Blue)
- New release is deployed to the other (Green)
- Smoke tests run
- If passed, switch traffic to Green

Branches used:

- main triggers deploy to Green

- main tagged → switch load balancer to Green

Rollback? Just point traffic back to Blue.

🐤 6. Canary Releases

In Canary deployment:

- Only a **small subset of users** get the new version initially
- Gradual rollout (5% → 25% → 100%)
- Monitors key metrics (errors, latency, usage)

Canary is supported using:

- Kubernetes + Istio/Argo Rollouts
- Feature flags (LaunchDarkly)
- Environment variables linked to Git tags

Rollback is automated if metrics degrade.

---

7. Rollback Strategy: Git-Based & CI-Based

A Git rollback means deploying the last working tag:

git checkout v2.2.1

git push origin v2.2.1

CI/CD triggers redeployment of the older release.

Backout strategy may include:

- Re-deploying previous Docker image
- Reverting database migrations (via Flyway/Liquibase)
- Rolling back feature flags

---

8. Rollback Scenarios

Scenario	Action	Rollback Plan
Release fails deployment	Abort deployment	Rollback to previous tag
Critical bug post-release	Patch via hotfix/*	Release patch (v2.3.1)

Scenario	Action	Rollback Plan
Traffic spikes/errors	Canary rollback	Shift traffic back
Data inconsistency	Infra rollback	Restore DB snapshot

🚒 9. Emergency Fixes: Hotfix Flow

Steps:

1. Branch from main:

git checkout -b hotfix/cache-null-fix main

2. Patch → test → commit
3. PR → Merge to main → Tag

git tag -a v2.3.1 -m "Fix caching issue"

git push origin v2.3.1

4. Back-merge to:

dev, qa, ppd, dr

Hotfixes are monitored **post-deploy for 24–48 hrs**.

📁 10. Release Documentation & Traceability

Each release is logged:

- Tag (e.g., v2.3.0)
- Date/Time
- PRs included
- Jira ticket links
- Release notes
- Build artifact hash/SHA
- Test coverage reports
- Approvers

Stored in:

- GitHub Releases

- Notion/Confluence
- Shared release calendars

---

11. Audit Trail: Git + CI Logs

- Git history (git log)
- Merge commits
- Tag history
- CI pipelines (Jenkins, GitHub Actions)
- Deployment timestamps
- Jira automation (transition to "Released")

This gives full traceability:

"Who changed what, when, and why?"

👥 12. Team Roles in PROD Releases

Role	Responsibilities
Developer	Final code/test merge
QA Lead	Test sign-off
Release Manager	Tag creation, deploy trigger
SRE/Infra	Monitor rollout, fix deploy issues
Product Owner	Approves Go/No-Go decision
CAB	Change request approval (optional)

---

13. Post-Deployment Observability

Once deployed:

- Logs and metrics are streamed to dashboards (Grafana, DataDog, NewRelic)
- Monitors include:
    - 5xx error rate
    - Latency

- DB connections
- CPU/memory usage

🔔 Alerts are routed via:

- Slack
- PagerDuty
- MS Teams

---

14. Release Checklist (Production)

Before deployment: ✅ All stories closed
✅ Tests passed (unit, regression, integration)
✅ Code reviewed
✅ Release notes ready
✅ Tag created
✅ Artifact stored in registry
✅ Infra ready
✅ Rollback plan documented
✅ Stakeholders notified

---

15. Versioning Strategy for Releases

Best practice is:

- Stick to SemVer
- Release tags match build artifacts
- Avoid reusing tag names

Examples:

- v2.3.0 → Full feature release
- v2.3.1 → Hotfix
- v2.3.2-rc → Release candidate (PPD test)
- v2.4.0-beta → Beta test with feature flags

---

16. Real-World PROD Deployment Example

Let's say your team is ready to release **User Analytics** feature.

Steps:

1. qa passed testing
2. Merged to ppd, passed UAT
3. Approval received
4. Merged to main
5. Tag created:

git tag -a v2.4.0 -m "User Analytics Release"

git push origin v2.4.0

6. CI pipeline deploys to production.
7. Monitors active for 48 hours.
8. Release note published.

Part 6: Synchronization, Rebase vs Merge, and Avoiding Merge Hell

"Code without synchronization is like a traffic system without signals — chaos is guaranteed."

🔹 1. The Problem: Merge Hell

Merge hell occurs when:

- Multiple long-lived branches diverge heavily
- Teams work in silos on features for weeks
- There's no sync from base branches
- PRs become bloated and unmergeable

Symptoms:

- Constant conflicts
- Hours lost resolving rebases
- PR reviews taking days
- Broken dev branch post-merge

Prevention is the key — and that's where smart rebasing and sync strategies come in.

🔸 2. Merge vs Rebase — What's the Difference?

Merge:

- Combines changes from one branch into another.
- Maintains commit history as-is.
- Creates a **merge commit**.

```
git checkout dev
git merge feature/new-dashboard
```

You'll see:

Merge branch 'feature/new-dashboard' into dev

Rebase:

- Rewrites commit history.
- Re-applies your changes on top of the latest dev code.
- Results in **clean, linear history**.

git checkout feature/new-dashboard

git pull --rebase origin dev

Now it looks like you branched from the latest dev HEAD — no merge commit needed.

---

3. When to Use Merge vs Rebase

Use Case	Merge	Rebase
Team work, multiple contributors	✅	❌
Clean up before PR	❌	✅
Shared branches (dev, qa, main)	✅	❌
Personal branches (feature/*)	❌	✅
You need full commit history	✅	❌
Want linear history for CI/CD	❌	✅

---

4. Best Practice: Rebase Before Merge

Before merging a feature/* branch to dev:

git checkout feature/user-profile

git fetch origin

git rebase origin/dev

Resolve conflicts (if any), then:

git push --force-with-lease

Then create a PR. The PR will:

- Be up to date with dev
- Have clean, testable commits
- Avoid unnecessary conflicts

---

5. Syncing Long-Lived Feature Branches

"Don’t wait for dev to come to you. Rebase early, rebase often."

For features that take more than a few days:

1. Set a rule: **Rebase daily from dev**
2. Test changes after every rebase
3. Communicate with teammates — “Hey, I rebased feature/reporting-ui from dev, please pull latest.”

```
git checkout feature/reporting-ui
```

```
git fetch origin
```

```
git rebase origin/dev
```

Then run your tests and push.

---

6. Avoiding Rebase on Shared Branches

Never rebase shared branches like dev, qa, ppd, or main.

Rebasing rewrites history. If other people are working on it, it will break everyone’s local repos.

Instead, use **merge** to keep shared history safe.

---

7. Back-Merge Strategy for Hotfixes

Every time a hotfix/* is merged to main, it **must** be back-merged into other active branches:

```
git checkout dev
git merge main
git push
```

```
git checkout qa
git merge main
git push
```

```
git checkout ppd
git merge main
git push
```

```
git checkout dr
```

git merge main
git push
This prevents divergence between critical branches and ensures all environments benefit from the patch.

---

8. Clean Feature Branching Rules

1. Prefix branches properly: feature/, bugfix/, hotfix/
2. One story = one branch = one PR
3. Delete feature branch post-merge
4. Never work on dev, qa, main directly
5. Rebase onto latest dev before PR

---

9. Avoiding Conflict Storms in PRs

- Keep your feature branches **small**.
- Merge or rebase often (at least daily).
- Don't let 10 devs pile into one PR.
- Don't wait till the last day to PR everything.
- Use **draft PRs** early to track changes.

---

10. CI/CD Pipelines to Prevent Drift

Pipelines can enforce sync:

- **Lint** to prevent dirty commits.
- **Check base** of PRs → is it up to date?
- **Test rebase PRs automatically**.

Example GitHub Action:

```
on:
  pull_request:
    types: [opened, synchronize]
jobs:
  rebase_check:
    runs-on: ubuntu-latest
    steps:
      - name: Check if PR is up to date
        run: git fetch origin dev && git rebase --check origin/dev
```

🧩 11. Fixing Merge Conflicts Cleanly

When rebasing and conflicts occur:

```
git rebase origin/dev

# conflict happens

# edit files, resolve manually

git add .

git rebase --continue
```

If you mess up:

```
git rebase --abort
```

If you're unsure about a force-push:

```
git push --force-with-lease
```

This ensures you're not overwriting someone else's changes.

🧬 12. Using Git Log & Graph for Clean History

Use git log with formatting:

```
git log --oneline --graph --decorate --all
```

This gives you a full branch graph to detect divergence, dangling commits, or stale branches.

Example output:

```
*   commit 19a8c5f (origin/dev)
|\
| * commit b1f9e72 (origin/feature/login)
|/
* commit 2d9e4d1
```

🧰 13. Recommended Tools for Git Hygiene

- **GitLens** (VS Code) → Visualizes rebase, authorship
- **GitKraken** → Visual branching UI
- **Sourcetree** → GUI for merge/rebase
- **CLI aliases** → For fast inspection

Alias example:

alias glog="git log --oneline --graph --decorate --all"

👥 14. Team Discipline & Sync Calendar

For large teams:

- Sync dev with main weekly
- Rebase active features daily
- Merge stale branches biweekly
- Delete dead branches monthly

You can automate stale branch reports via GitHub Actions or GitLab Scheduled Jobs.

🧱 15. Common Rebase Mistakes (And Fixes)

Mistake	Symptom	Fix
Rebased shared branch	Others' Git breaks	git reflog, recover
Force push without lease	Overwrites teammates' work	Use --force-with-lease
Rebase large PR late	Conflict storm	Rebase early, squash
Rebasing instead of merging to main	Audit trail broken	Always merge & tag in main
Pull with rebase on shared branches	Rewrite shared history	Don't do this on dev, qa, etc.

📜 16. Commit Hygiene & Squashing

Squash before merge to dev:

git rebase -i origin/dev

You'll see:

pick 9a1d3e5 Add login UI

pick f7123b2 Fix typo

pick 7bca1c9 Final refactor

Change "pick" to "squash" on 2nd and 3rd

Then push:

git push --force-with-lease

This results in 1 clean commit → better code reviews and history.

Part 7: Disaster Recovery Branch (dr) – Design, Sync, Simulation & Recovery Strategy

🛡️ *"Failing to prepare for disaster is preparing for disaster."*

In this section, we'll explore:

- What is the dr branch?
- Why it's critical in enterprise environments
- How it stays in sync with main
- CI/CD pipeline for DR deployment
- Simulating failovers with the dr branch
- DR testing frequency and ownership
- Real-world DR drills and learnings
- Full backup + restore strategies
- Tools and Git commands for DR readiness
- Integrating DR into audits and compliance

🔹 1. What is the dr Branch?

The dr (Disaster Recovery) branch is a **production-equivalent backup branch**, designed to:

- **Mirror the main branch**
- Act as the **source of truth** for a **disaster recovery (DR) environment**
- Enable failover testing without impacting live production
- Serve as the rollback path in case main is compromised
- Provide a **second deployment pipeline** to alternate infrastructure

It is NOT a development branch. It is **deployed but rarely modified** directly.

🔸 2. Why Is the dr Branch Critical?

🔥 Real-world failures:

- AWS region outage
- Production deployment corrupts DB

- DNS failure or DDoS attack

In such cases, having an **active DR environment deployed from dr** ensures:

- Business continuity
- Regulatory compliance (especially in fintech, healthcare)
- Zero-downtime architecture
- Trust from stakeholders

The dr branch:

- Enables **active-active** or **active-passive** failover
- Supports **multi-region infrastructure**
- Keeps your company running when PROD breaks

---

3. How the dr Branch Stays in Sync with main

The dr branch must always mirror main.

Sync Strategy 1: Manual Sync Post-Release

After every production release (main tag), merge main → dr:

```
git checkout dr
git pull origin dr
git merge main
git push origin dr
```

Sync Strategy 2: Automated via CI/CD

CI job that runs on every new tag in main:

```
on:
  push:
    tags:
      - 'v*.*.*'
jobs:
  sync-dr:
    runs-on: ubuntu-latest
    steps:
```

```
    - name: Checkout main
      uses: actions/checkout@v3
    - name: Merge into dr
      run: |
        git config user.name "DR Sync Bot"
        git config user.email "dr-sync@company.com"
        git checkout dr
        git merge main
        git push origin dr
```

🛠️ 4. CI/CD Pipeline for dr Deployment

The dr branch should have **its own CI/CD pipeline**, identical to main, but pointing to:

- A **secondary environment**
- **Isolated DNS**, infra, secrets, and endpoints

Use a dr-values.yaml or dr.env config to separate environments.

```
# Example Helm DR deploy
helm upgrade --install myapp ./chart -f dr-values.yaml
```

🧪 5. Disaster Recovery Testing & Simulation

🔁 *"Don't assume DR works. Simulate it."*

Types of DR Tests:

Test	Goal
Failover test	Switch DNS or load balancer to DR
Smoke test	Ensure app runs on DR env
Data restore test	Validate backup → restore
Latency test	Can DR handle same traffic as PROD?
Security test	Secrets, encryption, compliance on DR

These are triggered **weekly, biweekly, or monthly**, depending on SLA.

👥 6. Roles and Responsibilities in DR

Role	Responsibilities
DevOps	Keep dr in sync, infra config
QA	Test app behavior on DR
SRE	Monitor failover, simulate outages
Security	DR compliance and audit readiness
Product Owner	Approves DR drills and business impact test cases

---

📦 7. DR Failover Drill — Real-World Flow

Let's simulate a drill where AWS Mumbai (prod) is down and you failover to AWS Singapore (dr):

1. CI merges main → dr
2. dr triggers deploy to Singapore infra
3. Route 53 DNS cutover: prod.company.com → DR load balancer
4. Smoke tests validate:
    - App starts
    - DB reads/write working
    - Third-party services reachable
5. Once validated, DR is marked as **Active**
6. Once PROD is restored, failback begins (optional)

---

8. Verifying DR Environment Integrity

Your pipeline must validate:

- ✅ Build succeeded
- ✅ Environment variables set for DR
- ✅ Secrets decrypted correctly
- ✅ DNS routing in place

- ✅ Third-party integrations (payment, SMS) have test tokens or are mocked

Use automation tools like:

- ArgoCD with dr config
- Terraform workspaces for dual envs
- CI runners scoped to DR infra

---

9. Git Practices to Secure dr Branch

- ❌ No direct commits
- ✅ Always merged from main
- ✅ Git tag logs match PROD tags
- ✅ CI/CD artifacts traceable by SHA/tag
- ✅ Branch protection (admin-only PR merge)

---

10. DR Artifacts and Snapshots

When deploying from dr, the following should be versioned:

Artifact	Version Control?
App binaries	✅ (Artifact registry)
DB schema/migration files	✅
Secrets config	✅ (encrypted)
Helm charts / Terraform	✅
Backup restore scripts	✅
Smoke test playbooks	✅

Store in:

- GitHub repositories
- S3 (versioned)
- HashiCorp Vault or SealedSecrets for DR env

---

11. Security & Compliance in DR

Many audits (SOC 2, ISO 27001, PCI-DSS) require:

- DR plan
- Proof of drill execution
- DR environment isolation
- Secure key management
- Automated DR testing logs

Integrate audit reporting:

- In Confluence or Notion
- With Git logs + pipeline artifacts
- Slack summaries from DR runs

---

12. DR Health Monitoring

Monitor:

- DR pipeline success rate
- Artifact deploy time to DR
- Last successful DR drill
- Readiness score (0–100%)

Use:

- Grafana dashboards
- ELK stack (logs)
- Prometheus (metrics)

🧠 13. Real-World DR Mistakes and Fixes

Mistake	Impact	Fix
DR not in sync with main	Inconsistent releases	Automate main → dr
DR has different infra	Unknown bugs	Infra as Code parity

Mistake	Impact	Fix
Secrets missing	Deployment failure	DR-specific secrets rotation
No smoke tests	DR is untested	Write automated DR tests
No DR tags	Untraceable builds	Tag every DR deploy (dr-v2.3.0)

🔁 14. DR Branch Recovery Playbook (Git + CI)

1. main is down
2. Ops pulls latest dr
3. Run:

git checkout dr

git log --oneline

git tag

4. Find latest stable tag
5. Run DR pipeline (GitHub Actions, Jenkins, ArgoCD)
6. Point DNS to DR infra
7. Notify stakeholders
8. Monitor system
9. After fix, sync main ← dr if needed

🔐 15. Summary: dr Branch Readiness Checklist

- ✅ dr mirrors latest PROD tag
- ✅ CI/CD can deploy dr artifact
- ✅ DR infra exists and monitored
- ✅ Secrets/configs injected securely
- ✅ DR environment accessible via unique DNS
- ✅ DR tested monthly
- ✅ DR drill success reported to compliance

Part 8 : The Ultimate Conclusion of the DevOps Git Branching Strategy

“Security is not a stage — it's a constant across all branches.”

🔹 8.1 Secure Branching Strategy

Feature	Dev Branches	Main/DR Branches
Branch Protection	Optional	**Mandatory**
Signed Commits	Recommended	**Required**
CI Status Required	✅	✅
PR Review Count	1	2+
Direct Commits	❌	❌
Merge Strategy	Merge/Squash	Merge Only

🔹 8.2 GPG Signed Commits

git commit -S -m "Add secure logic"

Use GPG/SSH keys. Configure required signed commits in GitHub/GitLab settings.

🔹 8.3 Secrets Scanning at PR Level

✅ Enable **Gitleaks**, **GitGuardian**, or **TruffleHog** on all branches.

Example GitHub action:

```
- name: Scan for secrets
  uses: zricethezav/gitleaks-action@v2
```

🔹 8.4 DevSecOps in Branching Workflow

Branch	Security Action
feature/*	Lint + Dependency Scan
dev	Trivy, SonarQube, Gitleaks

Branch	Security Action
qa	API security tests
ppd	SAST + Infra Scan
main	Final security gate
dr	DR secret verification

🔹 **8.5 Commit Message Format (for audits)**

Enforce:

[ABC-123] Add JWT validation to login flow

- [ABC-123] → Jira/Task ID
- Standard format = traceable releases

Use tools like:

- commitlint
- semantic-release

🔹 **8.6 Git Audit Commands**

git log --author="John"

git log --grep="fix"

git log --since="2 weeks ago"

Pull exact history across environments. Useful for RCA (root cause analysis).

Part 9: CI/CD Pipeline Mapping Per Branch

⚙️ *"A branch without CI/CD is just a ticking time bomb."*

🔹 **9.1 Pipeline Structure Per Branch**

Branch	Pipeline Goals
feature/*	Fast feedback (lint, unit test)
dev	Full build + integration test

Branch	Pipeline Goals
qa	Regression + environment deploy
ppd	UAT + load test
main	Tag-based secure release
dr	Isolated DR deployment

🔹 9.2 Pipeline Example (GitHub Actions)

```
on:
  push:
    branches:
      - dev
jobs:
  build:
    steps:
      - run: npm install
      - run: npm test
      - run: sonar-scanner
```

🔹 9.3 Docker + Kubernetes Integration

- Build Docker image in dev
- Scan with **Trivy**
- Push to container registry
- Use Helm/Kustomize to deploy across qa, ppd, main, dr

🔹 9.4 Promotion Strategy (Tag-Based)

- Tag on main (e.g. v2.3.1) → deploy to prod
- Tag on dr (e.g. dr-v2.3.1) → deploy to DR

Use on: push: tags in pipeline triggers

Part 10: Metrics, Monitoring & KPIs

📊 *"You can't improve what you don't measure."*

🔹 10.1 DevOps Metrics Per Branch

Metric	Target	Branch Source
Lead Time	< 1 day	feature → main
Deployment Frequency	Daily	main
Change Failure Rate	< 15%	hotfix tracking
MTTR	< 1 hour	hotfix & rollback

🔹 10.2 Branch Age Tracking

Old branches = risk

git branch --sort=committerdate

Archive/delete inactive branches every 30 days.

🔹 10.3 Test Coverage Reports

Integrate code coverage in dev, qa:

- Use Jacoco, Coverage.py, Istanbul
- Enforce 80%+ minimum threshold

🔹 10.4 Monitoring Linked to Releases

Release v2.3.0 triggers:

- NewGrafana dashboards
- Prometheus alerts
- PagerDuty runbook refresh

Part 11: Real-World Scenarios: Release Freeze, Long Features, Cross-Team Work

🔹 11.1 Release Freeze

Before big launch:

- Freeze ppd → no new merges
- Hotfixes go from main to hotfix/* only
- Announce freeze window (e.g. 2 days)

🔹 11.2 Long-Lived Feature Branches

🔥 High risk unless synced daily

- Rebase from dev every day
- Auto-trigger tests on rebase
- Use feature flags for progressive rollout
- If delay > 2 weeks → split into sub-features

🔹 11.3 Cross-Team Collaboration

Team A	Team B
Backend	Frontend

- Shared integration/* branch
- Merged from feature/backend-x and feature/frontend-y
- CI validates combined E2E tests
- Once stable → merged to dev

🔹 11.4 Coordinating Multi-Region Releases

- main deploys to Region A
- dr deploys to Region B
- Canary routing via GeoDNS (e.g., Route53 latency policy)

Part 12: Best Practices & Anti-Patterns

✅ Best Practices

- Use semantic branch names: feature/abc-123-login-form
- Rebase feature/* before PRs

- Tag every release
- Lock main with reviews + CI
- Automate sync: main → dr
- Write changelogs from commits

❌ Anti-Patterns

Pattern	Why It Fails
Direct commit to main	Breaks prod
Rebasing shared branches	Breaks team
Too many untagged hotfixes	No traceability
No testing in qa	Bugs in PROD
Skipping rebase for long-lived branches	Merge hell
No DR validation	Failures in emergencies
