17

Git and Version Control

Beyond Push and Pull

THE MODERN QA ENGINEER SKILLSET

Git and Version Control: Beyond Push and Pull

Branching Strategies, Collaboration, and QA Workflows

Yuri Syuganov

Modern QA Course

https://modernQAcourse.com

Git and Version Control: Beyond Push and Pull

Branching Strategies, Collaboration, and QA Workflows

Book 17 of 26 in the series *The Modern QA Engineer Skillset*

First Edition · Version 1.0.0 · 2026-07-11

Set in Charter, IBM Plex Sans, and IBM Plex Mono. Typeset with Pandoc and XeLaTeX.

Modern QA Course · https://modernQAcourse.com · support@modernQAcourse.com

Contents

Introduction

About This Book

Every engineer who writes code uses `git add`, `git commit`, and `git push` daily. That is the floor, not the ceiling. This book takes you beyond the basics and into the territory where Git becomes a genuine engineering tool – one that helps you isolate regressions in minutes, manage complex release pipelines, collaborate effectively through pull requests, and keep repositories clean and maintainable.

Git is not just a code storage utility. It is a debugging tool (`bisect`), a collaboration protocol (pull requests and code review), a release management system (tags and semantic versioning), and a historical record that tells the story of every decision your team ever made. This book teaches you how to use all of those dimensions.

The material is organized into seven chapters, each building on the previous. You will start with branching strategies that shape how teams work, move through the rebase-versus-merge debate with concrete guidance, learn to write pull requests that reviewers appreciate, practice code review through a QA lens, hunt regressions with `git bisect`, manage releases with tags, and master the practical day-to-day Git commands that save hours of frustration.

Every chapter includes interactive exercises at three difficulty levels, pro tips from real-world experience, and common mistakes to avoid. The book concludes with a capstone project that ties every concept together and a comprehensive glossary.

Who This Book Is For

- **QA engineers** who want to move beyond basic Git usage and become power users who can investigate regressions, write excellent pull requests, and influence their team's branching strategy.
- **Software developers** who are comfortable with basic Git but want to deepen their understanding of branching models, interactive rebase, cherry-picking, and release workflows.
- **Team leads and engineering managers** who need to choose a branching strategy, set up PR workflows, and establish code review practices.
- **DevOps engineers** who configure CI/CD pipelines and need to understand how branching strategies, tagging, and release workflows interact with automation.
- **Students and bootcamp graduates** who learned the basics of Git and want professional-grade skills that will serve them throughout their career.

Prerequisites

Before reading this book, you should be comfortable with:

- Basic Git commands: `git init`, `git add`, `git commit`, `git push`, `git pull`, `git clone`
- The concept of branches (creating, switching, deleting)
- Using a terminal or command line
- A GitHub, GitLab, or Bitbucket account for the collaboration exercises
- A text editor or IDE with Git integration (VS Code, IntelliJ, etc.)

If you can create a repository, make commits, and push to a remote, you are ready for this book. Everything else, we will build from here.

Table of Contents

- **Chapter 2**: Rebase vs. Merge – Two Philosophies, One Codebase
- **Chapter 3**: Pull Request Workflows – Writing PRs That Get Reviewed
- **Chapter 4**: Code Review – The QA Lens
- **Chapter 5**: Git Bisect and Cherry-Pick – Hunting Regressions and Applying Targeted Fixes
- **Chapter 6**: Tagging and Releases – Marking History, Managing Versions
- **Chapter 7**: Gitignore, Hooks, and Practical Git Mastery
- **Capstone Project**: The Full Release Cycle
- **Glossary**

CHAPTER 1

Branching Strategies

Why Your Branching Strategy Matters

The branching strategy your team uses is not a minor implementation detail. It determines how you manage feature development, when you run which tests, how releases map to test results, and how quickly you can deliver hotfixes to production. Understanding the strategy is not optional – it directly shapes your entire development and test execution plan.

There are three dominant branching strategies in modern software development, and each one makes different trade-offs between safety, speed, and complexity.

The Three Main Strategies at a Glance

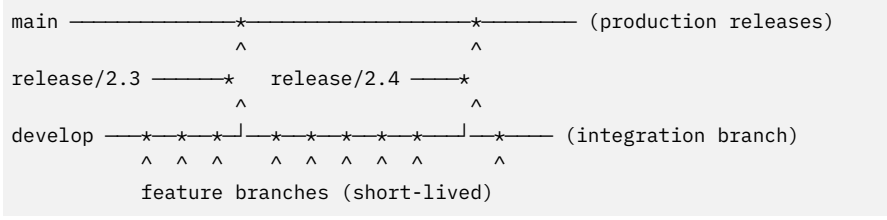
Aspect	GitFlow	GitHub Flow	Trunk-Based Development
Main branches	`main` + `develop`	`main` only	`main` only
Feature branches	Branch from `develop`, merge back to `develop`	Branch from `main`, merge via PR	Short-lived branches (< 1 day), merge to `main`

continued on next page

Aspect	GitFlow	GitHub Flow	Trunk-Based Development
Release process	`release/*` branch for stabilization	Deploy from `main` after merge	Deploy from `main` continuously
Hotfixes	`hotfix/*` branch from `main`	Branch from `main`, merge via PR	Fix on `main` directly
Complexity	High – multiple long-lived branches	Low – one main branch	Low – but requires mature CI/CD
Best for	Scheduled releases, multiple versions in production	SaaS with continuous deployment	Teams with strong test automation and feature flags

1.1 GitFlow in Detail

GitFlow, introduced by Vincent Driessen in 2010, uses two long-lived branches (`main` and `develop`) plus short-lived branches for features, releases, and hotfixes. It is the most structured of the three strategies.



How GitFlow Works

1. **`main`** always reflects the current production state. Every commit on `main` is a release.
2. **`develop`** is the integration branch. All feature branches merge here.
3. **Feature branches** branch from `develop` and merge back to `develop` when complete.

4. **Release branches** (`release/2.4`) branch from `develop` when the team is ready to prepare a release. Only bug fixes go on the release branch – no new features.
5. **Hotfix branches** (`hotfix/2.3.1`) branch from `main` for urgent production fixes. They merge to both `main` and `develop`.

Implications for Testing

- **Test on `develop`**: Every feature merged to `develop` should trigger regression tests.
- **Test again on `release/*`**: The release branch is for stabilization. Each bug fix on the release branch needs testing.
- **Double testing burden**: You may test a feature on `develop` and then test the same feature again on the release branch.
- **Hotfix testing**: Hotfixes branch from `main`, get tested in isolation, then merge to both `main` and `develop`.
- **Environment mapping**: `develop` maps to a dev/integration environment, `release/*` maps to staging, `main` maps to production.

When to Use GitFlow

- Your product has scheduled releases (quarterly, monthly).
- Multiple versions are supported simultaneously (v2.3 and v2.4 both in production).
- Regulatory requirements demand a stabilization phase before release.
- Your team is large and needs the structure that multiple branches provide.

> **Pro Tip**: GitFlow adds overhead. If you are deploying to production multiple times per day, GitFlow's release branches will slow you down. Use it when you genuinely need the stabilization phase, not because "that's how we've always done it."

1.2 GitHub Flow in Detail

GitHub Flow, popularized by GitHub itself, uses a single `main` branch. All work happens on feature branches that merge via pull requests. There are no `develop` or `release` branches.

```
main ──────●──●──●──●──●──●──●──●──●──●──●──●──── (always deployable)
           ^  ^  ^  ^  ^  ^  ^  ^  ^
           PRs from feature branches
```

How GitHub Flow Works

1. **`main`** is always deployable. If something is on `main`, it is production-ready.
2. **Feature branches** branch from `main`, contain all work, and merge back via pull request.
3. **Pull requests** are the primary quality gate. CI runs all tests on the PR before merge.
4. **Deploy from `main`** after merge. Many teams deploy automatically on every merge.

Implications for Testing

- **Test on PR**: Every pull request triggers the full test suite. This is your primary quality gate.
- **Main is always deployable**: If tests pass on the PR and the PR is merged, `main` should be safe to deploy at any time.
- **No stabilization phase**: There is no release branch. If something is broken on `main`, fix it with another PR.
- **Simpler environment mapping**: Feature branches deploy to preview environments, `main` deploys to production.

When to Use GitHub Flow

- Your team does continuous deployment (deploy multiple times per day).
- You have a single version in production (typical for SaaS).

- Your CI pipeline is fast and reliable enough to gate PRs effectively.
- Your team is small to medium and does not need the structure of GitFlow.

> **Common Mistake**: Treating `main` as a "development branch" where broken code is acceptable. In GitHub Flow, `main` must always be deployable. If your tests do not run on every PR, GitHub Flow will burn you.

1.3 Trunk-Based Development in Detail

Trunk-based development takes GitHub Flow to its logical extreme. Feature branches live less than a day – often less than a few hours. Developers commit to `main` frequently, often multiple times per day.

```
main ──****************************──── (continuous stream of small commits)
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
       Very short-lived branches (hours, not days)
```

How Trunk-Based Development Works

1. **`main`** (the "trunk") receives commits continuously.
2. **Short-lived branches** exist only for the duration of a single code review, typically hours.
3. **Feature flags** replace feature branches. Incomplete features are deployed behind flags, not on separate branches.
4. **Every commit must pass all tests**. There is no stabilization period.

Implications for Testing

- **Every commit must pass all tests**: The pipeline must be fast and reliable. There is no "fix it later" window.
- **Feature flags replace feature branches**: Incomplete features are deployed behind flags, which means your tests must be flag-aware.
- **Testing shifts left dramatically**: QA reviews requirements and writes tests before development starts, because there is no time to catch up later.

- **Flaky tests are existential threats**: A flaky test that blocks `main` blocks the entire team. Fix flaky tests immediately, with the same urgency as a production incident.

When Trunk-Based Development Works

- The team has high test automation maturity (>90% of tests automated).
- CI pipeline completes in under 10 minutes.
- Feature flags are available and well-managed.
- The team has strong code review discipline.
- Engineers are experienced and disciplined about small, incremental commits.

> **Pro Tip**: Trunk-based development is not "no branches." It is "very short-lived branches." The goal is to minimize divergence from `main`, not to eliminate branches entirely.

1.4 Adapting Your Test Plan to the Strategy

Different branching strategies require different test execution timing:

Strategy	Unit Tests	Integration Tests	E2E Tests	Full Regression
GitFlow	On every push	On PR to develop	On PR to develop	On release branch, before merge to main
GitHub Flow	On every push	On every PR	On every PR	On merge to main (pre-deploy)
Trunk-Based	On every push	On every push or PR	On merge to main	Continuously (every deploy)

1.5 Transitioning Between Strategies

Teams often start with GitFlow and migrate toward GitHub Flow or trunk-based development as their automation matures. If your team is considering a transition, here is what you need to prepare.

Moving from GitFlow to GitHub Flow

1. **Speed up the pipeline**: GitHub Flow relies on PR checks as the primary gate. If your pipeline takes 40 minutes, PRs will be painful.
2. **Eliminate the release branch test phase**: All testing must happen on the PR. Any tests that only ran on the release branch must move to the PR pipeline.
3. **Improve test reliability**: In GitFlow, a flaky test on the release branch might be tolerated. In GitHub Flow, a flaky test blocks every PR.

Moving from GitHub Flow to Trunk-Based

1. **Implement feature flags**: You need a way to deploy incomplete features safely.
2. **Reduce pipeline time to under 10 minutes**: Developers cannot wait 20 minutes for every small commit.
3. **Achieve near-zero flaky test rate**: Every test failure on main blocks the team.
4. **Shift QA earlier**: QA must participate in design and requirements, not wait for code.

1.6 Interactive Exercise: Map Your Branching Strategy

Step-by-Step

```
# Step 1: Examine your current project's branching structure
git branch -a

# Step 2: Look at the recent merge history to understand the pattern
git log --oneline --graph --decorate --all -20

# Step 3: Count how many long-lived branches exist
git branch -r | wc -l

# Step 4: Check the age of feature branches
git for-each-ref --sort=-committerdate refs/remotes/ \
  --format='%(committerdate:short) %(refname:short)'

# Step 5: Measure your average PR pipeline time
# (Check your CI dashboard for this metric)
```

Exercises

Beginner

1. Identify which branching strategy your current team uses (or the team at your last job). Write down the branch names and what each one is used for.
2. Draw the branch diagram for your project on paper or a whiteboard. Where do features, releases, and hotfixes live?
3. List three advantages and three disadvantages of your current branching strategy.

Intermediate

4. Map out when each type of test runs in your current workflow. Are there gaps where code can reach production without certain tests running?
5. If your team uses GitFlow, identify tests that run twice (on `develop` and on `release`). Could you eliminate the duplication without reducing coverage?

6. If your team uses GitHub Flow, measure your PR pipeline time. Is it under 15 minutes? If not, identify the bottleneck.

Advanced

7. Write a proposal for migrating your team from GitFlow to GitHub Flow. Include the prerequisites (pipeline speed, test reliability, etc.) and a phased migration plan.
8. Design a feature flag strategy that would enable trunk-based development for your project. What flags would you need? How would tests interact with them?
9. Create a test execution matrix for all three branching strategies, customized to your project's specific test types and environments.

Career Translation

Resume phrasing

- Evaluated and recommended branching strategies (GitFlow, GitHub Flow, trunk-based) for a team of 12 engineers, aligning test execution plans with branch lifecycle to reduce regression escape rate by 30%.
- Designed branch-to-environment mapping for CI/CD pipelines, ensuring automated test suites ran at the correct integration points across development, staging, and production.
- Led migration from GitFlow to GitHub Flow, reducing release cycle time from two weeks to continuous deployment while maintaining full regression coverage on every pull request.

Cover letter framing

- Understanding branching strategies is not an academic exercise – it directly determines when and where tests run, how quickly regressions are caught, and how confidently a team can ship. I evaluate branching models against a team's automation maturity and release cadence, then design test execution plans that match the strategy's strengths and cover its gaps.

Interview framing

- "I approach branching strategy as a testing architecture decision, not just a development workflow choice. The strategy dictates where I place quality gates: in GitFlow I plan for testing at develop, release, and main; in GitHub Flow I invest heavily in PR-level checks; in trunk-based I ensure every commit is deployable with feature-flag-aware tests. The trade-off is always between safety and velocity, and the right answer depends on the team's CI maturity, flaky test rate, and release cadence."

What not to say

- "We just use GitFlow because that is what everyone uses." – Shows no critical evaluation of trade-offs; interviewers want to hear why a strategy fits a context.
- "Branching does not affect QA." – Reveals a fundamental misunderstanding of how code integration timing shapes test planning.
- "I only test on the main branch." – Signals that you do not understand shift-left testing or pre-merge quality gates.
- "We do not need a branching strategy, we just commit to main." – Conflates undisciplined commits with trunk-based development, which requires rigorous automation and feature flags.

Interview Depth Check

Question 1

Prompt: Your company currently uses GitFlow with a two-week release branch stabilization period. The VP of Engineering wants to move to weekly releases. The test suite takes 45 minutes to run, and the flaky test rate is around 8%. How would you advise the team on changing the branching strategy, and what prerequisites would you insist on before making the switch?

What a strong answer should cover:

- Recognition that the current pipeline speed and flaky test rate are blockers for faster release cadence
- A phased migration plan rather than an overnight switch

- Specific, measurable prerequisites (pipeline under 15 minutes, flaky rate under 2%)
- The relationship between branching strategy and test execution timing

Example answer:

- I would first present the current blockers: a 45-minute pipeline means PR feedback loops are painful in GitHub Flow, and an 8% flaky rate would block PRs constantly. Before migrating, we need to parallelize or shard the test suite to get under 15 minutes, and quarantine or fix flaky tests to get under 2%. I would propose a phased approach: first, move to GitHub Flow on one service as a pilot while keeping GitFlow on others. Measure the pipeline time, flaky rate, and developer satisfaction on the pilot. Once the prerequisites are met, roll out across the organization. I would not recommend trunk-based development until feature flags are in place and the pipeline is under 10 minutes.

Question 2

Prompt: A team member argues that trunk-based development is inherently riskier than GitFlow because there is no stabilization period. How do you respond?

What a strong answer should cover:

- The distinction between risk mitigation via process (stabilization branch) versus risk mitigation via automation (continuous testing, feature flags)
- That trunk-based development shifts risk management left, not eliminates it
- The hidden costs of long-lived branches (merge conflicts, delayed integration, duplicated testing)

Example answer:

- I would acknowledge the concern and reframe it: trunk-based development does not eliminate the stabilization phase – it replaces a manual stabilization branch with continuous automated verification. Every commit runs the full test suite, which means regressions are caught within minutes rather than discovered during a two-week stabilization window. The trade-off is that this requires investment in test automation maturity, pipeline speed, and feature flags. GitFlow gives you a safety net at the cost of slower feedback loops and duplicated testing effort. Trunk-based development gives you faster feedback at the cost of higher upfront investment in automation. Neither is inherently riskier – the risk profile depends on the team's infrastructure.

Question 3

Prompt: You join a new team and discover they are using GitHub Flow, but the main branch has been broken three times in the last month. Deployments were rolled back twice. What is your diagnosis and remediation plan?

What a strong answer should cover:

- Root cause analysis: likely insufficient PR-level testing, inadequate branch protection rules, or test gaps
- Specific remediation steps: branch protection, required checks, test coverage analysis
- Monitoring and metrics to prevent recurrence

Example answer:

- My first step is a post-mortem on the three breakages: what type of failures were they (test gaps, flaky tests that were ignored, missing test types like integration or E2E)? Then I would audit the branch protection rules – are PR reviews required? Are CI checks mandatory before merge? Is the branch required to be up to date with main before merging? I would add mandatory status checks for unit, integration, and E2E tests on every PR. I would review the test suite for gaps in the areas that broke. I would also introduce a post-merge

smoke test that runs immediately after merge to main, with an automatic rollback trigger if critical tests fail. Finally, I would set up a dashboard tracking main branch stability over time so the team has visibility into the trend.

About This Sample

You have just read **Chapter 1 of Git and Version Control: Beyond Push and Pull** — the free sample. The complete book continues with every remaining chapter, the interview Q&A, and all of the code.

Get the full book

Sample readers get a discount at
https://leanpub.com/modernqa-17-git-version-control/c/SAMPLE10

Read online first

Chapter one of **every** book in the series is free to read at
https://modernqacourse.com/library

About the Author

Yuri Syuganov is a software quality engineer with more than twenty years of hands-on testing experience, spanning the arc from manual test design through Selenium-era automation to today's AI-augmented, agent-driven frameworks. He is the author of The Modern QA Engineer Skillset, a 26-book series built on a blunt analysis of what companies actually expect from QA engineers, and the creator of modernQAcourse.com, where the course behind the series is free and readers can bring real testing problems to experienced practitioners. He revises the series as the industry changes — because it does.

Also in This Series

The Modern QA Engineer Skillset — a 26-book course in modern software quality assurance.

AI-Augmented QA (Books 1–10)

- **Book 1:** Agent Skills for Browser Automation: The Complete Guide
- **Book 2:** AI-Augmented Test Design: From Prompts to Test Suites
- **Book 3:** Agentic Testing Architectures: Building Autonomous QA Systems
- **Book 4:** API and Contract Testing with AI: Schema to Shield
- **Book 5:** Performance and Chaos Engineering: Breaking Things on Purpose
- **Book 6:** Observability-Driven Testing: Testing in Production
- **Book 7:** Security Testing for AI Applications: The New Attack Surface
- **Book 8:** Infrastructure as Code Testing: Terraform to Kubernetes
- **Book 9:** Mobile and Cross-Platform Testing: 24,000 Devices, One Strategy
- **Book 10:** Visual and Accessibility Testing: Pixels and People

Technical Foundations (Books 11–19)

- **Book 11:** Manual Testing Fundamentals: The Bedrock
- **Book 12:** Programming for QA: Python, JavaScript, and TypeScript from Zero to Fluent
- **Book 13:** Browser Test Automation: Playwright, Patterns, and the Death of Fragile Tests

- **Book 14:** API Testing Fundamentals: Postman to Pytest
- **Book 15:** SQL and Database Testing: Query Your Way to Quality
- **Book 16:** CI/CD Pipelines: From Commit to Confidence
- **Book 17:** Git and Version Control: Beyond Push and Pull
- **Book 18:** Test Management Tools: Making Quality Visible
- **Book 19:** Linux and Command Line: Your Production Survival Kit

Professional Skills (Books 20–26)

- **Book 20:** Agile and Scrum for QA: Quality in Every Sprint
- **Book 21:** Communication and Stakeholder Management for QA: Influence Without Authority
- **Book 22:** Test Strategy and Quality Metrics: Measuring What Matters
- **Book 23:** QA Leadership and Mentoring: Multiplying Your Impact
- **Book 24:** Technical Writing for QA: Documents That Drive Action
- **Book 25:** Interview Preparation and Career Strategy: Land the Job
- **Book 26:** Testing Like a Senior: Patterns & Anti-Patterns Across Software QA Career Levels

Visit https://modernQAcourse.com for the full course catalog, updates, and companion resources.

Stuck on a Real Problem?

The full course behind this series is free to read at https://modernQAcourse.com — and it stays free.

When a chapter meets the real world and something breaks, post it on the **Help Board** at https://modernQAcourse.com. Asking is free, and experienced practitioners answer right in the thread. Need hands-on help? Book a 1:1 screen-share session with a QA expert.