Enterprise Playwright + TypeScript Automation Handbook

> **Comprehensive Guide: From Basics to Enterprise-Level Practices**
>
> *A complete onboarding and training resource for automation teams*

Document Overview

Attribute	Details
Framework	Playwright + TypeScript
Purpose	Enterprise onboarding, training, and reference
Audience	Automation Engineers (Junior to Senior), Leads, DevOps
Coverage	Git, GitHub, CI/CD, Best Practices, Team Workflows
Created	December 31, 2024

Table of Contents

Part 1: Foundation

- Section 1 — Enterprise Framework Overview
- Section 2 — Repository & Branching Strategy
- Section 3 — Daily Tester Workflow

Part 2: Git & Version Control

- Section 4 — Git Concepts Explained Simply

Part 3: CI/CD & Automation

- Section 5 — CI/CD Fundamentals
- Section 6 — GitHub Actions Deep Dive
- Section 7 — CI/CD Pipeline for Playwright Framework

Part 4: Enterprise Operations

- Section 8 — How Employees Use CI/CD Pipeline
- Section 9 — Managing CI/CD for Multiple Teams

Part 5: Standards & Practices

- Section 10 — Enterprise Best Practices
- Section 11 — Roles & Responsibilities
- Section 12 — Common Enterprise Mistakes & Solutions
- Section 13 — Real-World Enterprise Workflow Summary

Bonus

- Interview Perspective
- Troubleshooting Tables
- Do's and Don'ts
- Real Enterprise Examples
- Quick Reference Card

How to Use This Document

For New Team Members:

1. Start with Sections 1-4 for foundational concepts
2. Read Sections 5-7 to understand CI/CD
3. Study Section 3 for daily workflow
4. Reference Sections 10-12 for best practices

For Experienced Engineers:

1. Jump to Sections 8-9 for enterprise patterns

2. Review Section 10 for standards compliance
3. Use Bonus sections for interview prep

For Leads/Managers:

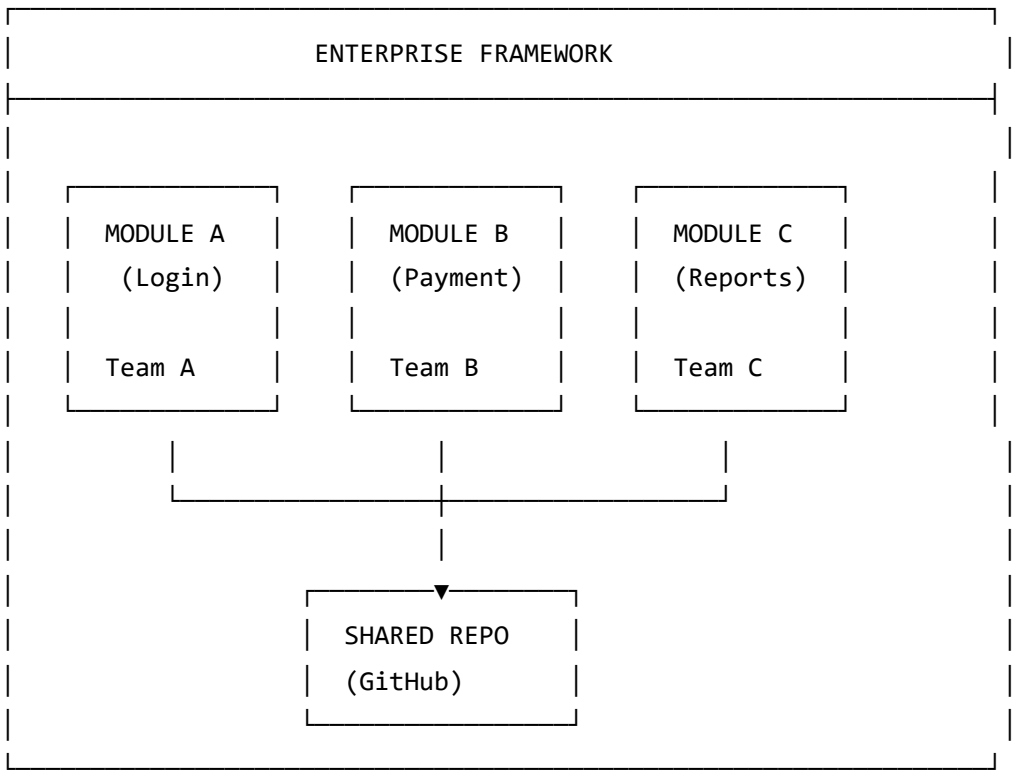
1. Focus on Section 11 for role definitions
2. Review Section 9 for multi-team management
3. Use Section 13 for workflow overview

SECTIONS 1-3: Enterprise Framework, Branching & Daily Workflow

SECTION 1 — Enterprise Framework Overview

1.1 What is an Enterprise Automation Framework?

An **enterprise automation framework** is a structured system where multiple employees work together to automate software testing across different application modules.



1.2 How Multiple Employees Work on Different Modules

In an enterprise, the application is divided into **modules**:

Module	Example	Team Assigned
Authentication	Login, Logout, Password Reset	Team A
User Management	Registration, Profile, Settings	Team B
Payments	Checkout, Billing, Refunds	Team C
Reports	Dashboard, Analytics, Export	Team D

How It Works:

1. Each team owns specific modules
2. All teams share the same code repository
3. Each team creates their tests in separate folders
4. Everyone follows the same coding standards
5. CI/CD runs all tests together

1.3 Key Roles and Responsibilities

1.3.1 Automation Engineer (Junior/Mid Level)

Primary Responsibilities:

- Write automated test cases
- Maintain existing tests
- Execute tests locally
- Report bugs found during automation
- Follow coding standards

Daily Activities:

```
Morning:
  └── Pull latest code from repository
  └── Check if any assigned tests need updates

During Day:
  └── Write new test cases
  └── Fix failing tests
  └── Run tests locally before pushing

End of Day:
  └── Push code to feature branch
  └── Raise Pull Request if work is complete
```

1.3.2 Senior Automation Engineer

Primary Responsibilities:

- Design framework structure
- Review code from junior engineers
- Handle complex automation challenges
- Mentor team members
- Optimize test execution

Additional Duties:

- Create reusable utilities
- Define page objects structure
- Establish coding standards

- Troubleshoot flaky tests

1.3.3 Automation Lead

Primary Responsibilities:

- Plan automation strategy
- Assign work to team members
- Track progress and metrics
- Coordinate with other teams
- Make architectural decisions

Key Metrics Tracked:

Metric	Description
Test Coverage	% of features automated
Pass Rate	% of tests passing
Execution Time	Total time to run all tests
Flaky Rate	% of unstable tests

1.3.4 CI/CD Owner (DevOps Engineer)

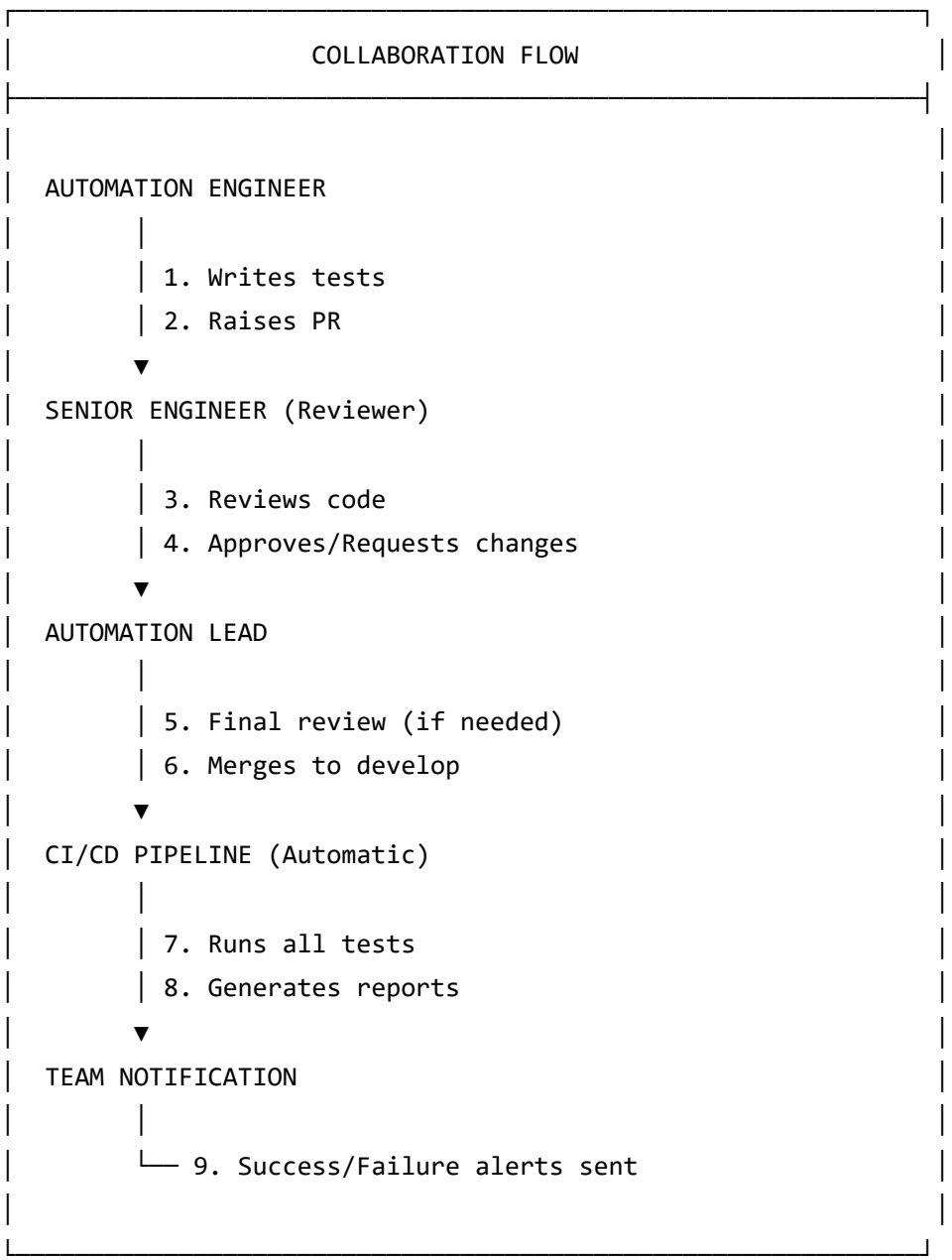
Primary Responsibilities:

- Set up and maintain CI/CD pipelines
- Configure GitHub Actions/Jenkins
- Manage cloud runners
- Handle secrets and credentials
- Monitor pipeline health

Key Tasks:

- Configure parallel execution
- Set up test environments
- Manage artifact storage
- Optimize pipeline speed

1.4 Team Collaboration Model



SECTION 2 — Repository & Branching Strategy

2.1 What is Git?

Git is a version control system that tracks changes to your code over time.

Simple Analogy: Think of Git like "Track Changes" in Microsoft Word, but much more powerful for code.

Why Git is Essential:

- Track all changes made to code
- See who made what change and when
- Revert to previous versions if needed
- Work on multiple features simultaneously
- Collaborate without overwriting others' work

2.2 What is GitHub?

GitHub is a cloud platform that stores your Git repositories online.

Simple Analogy: If Git is like a photo album, GitHub is like Google Photos where you store and share your albums.

Why Enterprises Use GitHub:

Reason	Explanation
Central Storage	All code in one accessible place
Backup	Code is safe even if laptop crashes
Collaboration	Multiple people can work together
Code Review	Pull Requests for quality control
CI/CD	Built-in GitHub Actions
Access Control	Control who can see/edit code
Audit Trail	Complete history of all changes

2.3 Repository Structure

```
playwright-automation-framework/          ← Root Directory
│
├── .github/                              ← GitHub Configuration
│   └── workflows/
│       └── playwright.yml                ← CI/CD Pipeline
│
├── pages/                                ← Page Object Models
│   ├── LoginPage.ts
│   ├── RegisterPage.ts
│   └── DashboardPage.ts
│
├── tests/                                ← Test Files
│   ├── login/
│   │   └── login.spec.ts
│   ├── register/
│   │   └── register.spec.ts
│   └── dashboard/
│       └── dashboard.spec.ts
│
├── test-data/                            ← Test Data
│   ├── loginData.ts
│   └── registerData.ts
│
├── utils/                                ← Utilities
│   ├── helpers.ts
│   └── excelReader.ts
│
├── playwright.config.ts                  ← Playwright Config
├── package.json                          ← Dependencies
├── tsconfig.json                         ← TypeScript Config
└── README.md                             ← Documentation
```

2.4 Branch Types Explained

2.4.1 Main/Master Branch

```
PURPOSE: Production-ready code only
```

Rules:

- ❌ Never commit directly
- ❌ Never push untested code
- ✅ Only merge from develop after full testing
- ✅ Always stable and deployable

Protection: This branch should be protected - no one can push directly.

2.4.2 Develop Branch

```
PURPOSE: Integration branch for all features
```

Rules:

- All feature branches merge here first
- Must pass all tests before merging to main
- Represents "next release" code

2.4.3 Feature Branches (feature/*)

```
PURPOSE: New test cases or features
```

Examples:

- `feature/login-tests`
- `feature/payment-validation`
- `feature/add-excel-reader`

Lifecycle:

```
1. Created FROM: develop
2. Merged INTO: develop
3. Deleted AFTER: merge is complete
```

2.4.4 Bugfix Branches (bugfix/*)

```
PURPOSE: Fix failing tests or broken code
```

Examples:

- `bugfix/login-timeout-issue`

- `bugfix/locator-update-dashboard`

When to Use:

- Test started failing after recent changes
- Locator changed due to UI update
- Logic error found in existing test

2.4.5 Hotfix Branches (hotfix/*)

```
PURPOSE: Emergency fixes for production
```

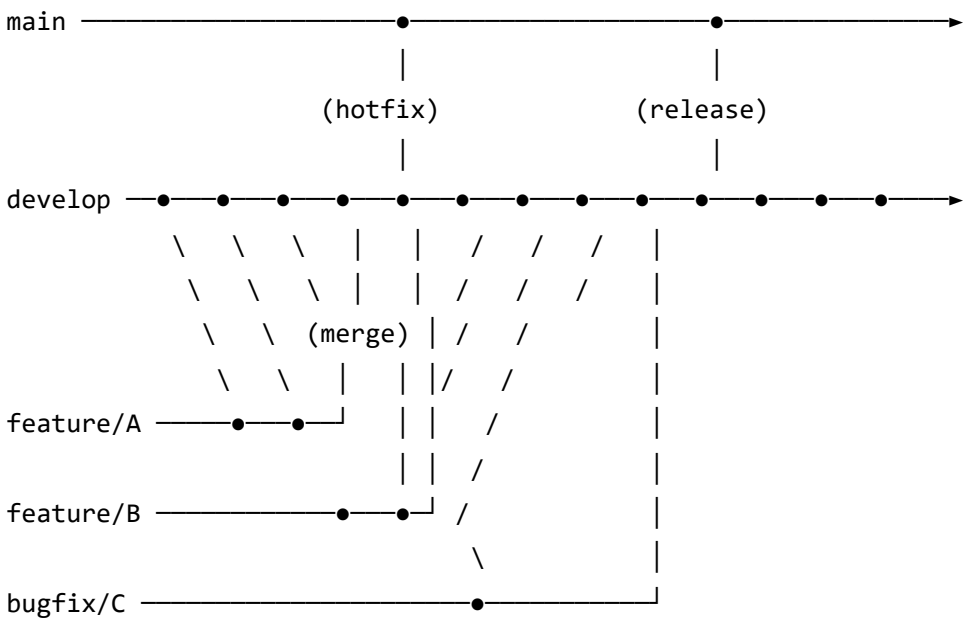
Examples:

- `hotfix/critical-login-failure`
- `hotfix/ci-pipeline-broken`

Special Rules:

- Created from: main (not develop)
- Merged into: BOTH main AND develop
- Used only for urgent issues

2.5 Branch Flow Diagram



LEGEND:

● = commit

─ = time progression

/ = merge

2.6 Branch Naming Conventions

Type	Format	Example
Feature	`feature/<description>`	`feature/login-page-tests`
Bugfix	`bugfix/<issue>`	`bugfix/fix-flaky-checkout`
Hotfix	`hotfix/<critical-issue>`	`hotfix/ci-broken`
Release	`release/<version>`	`release/v1.2.0`

Naming Rules:

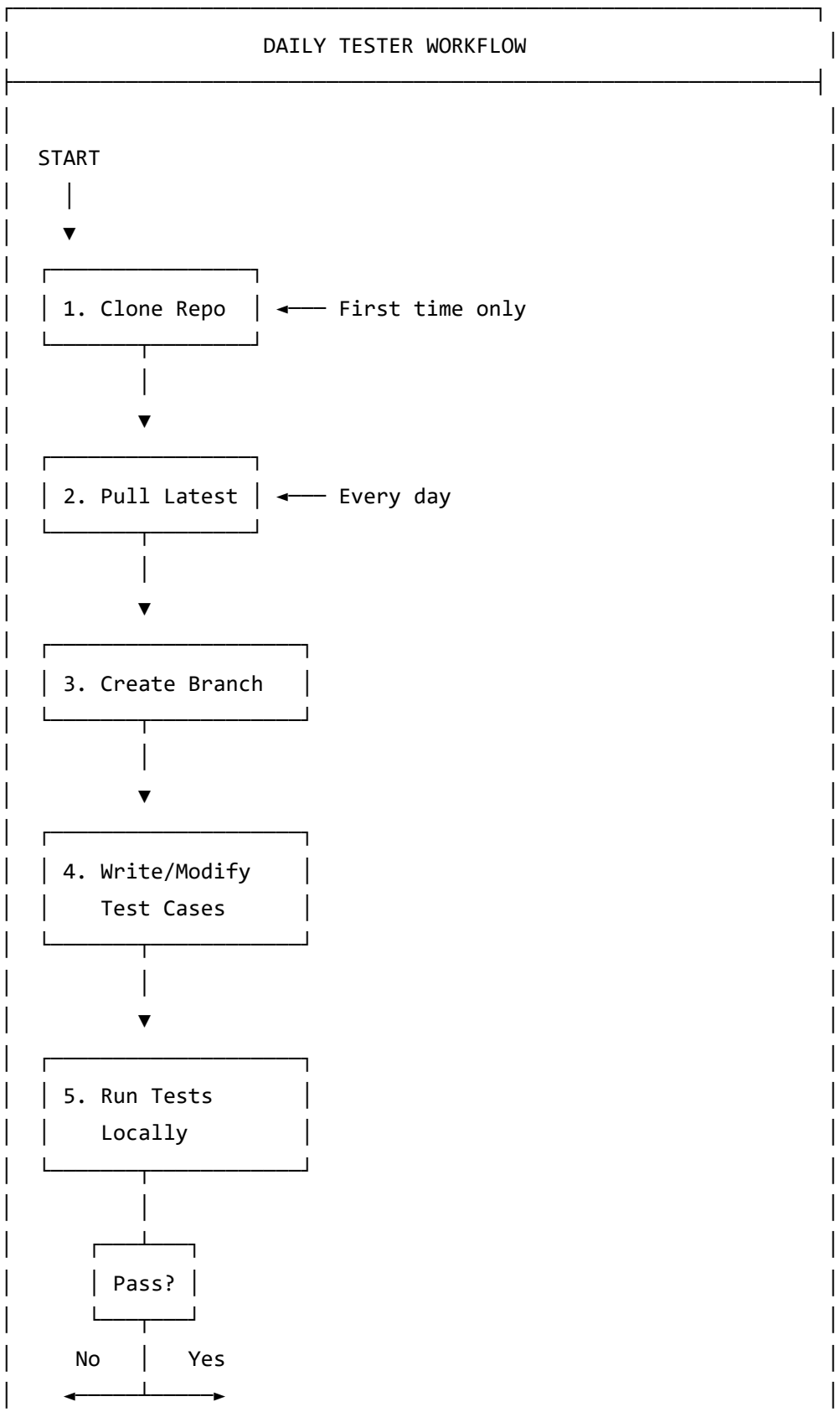
1. Use lowercase letters only
2. Use hyphens (-) not underscores
3. Keep it short but descriptive
4. Include ticket number if available (e.g., `feature/JIRA-123-login-tests`)

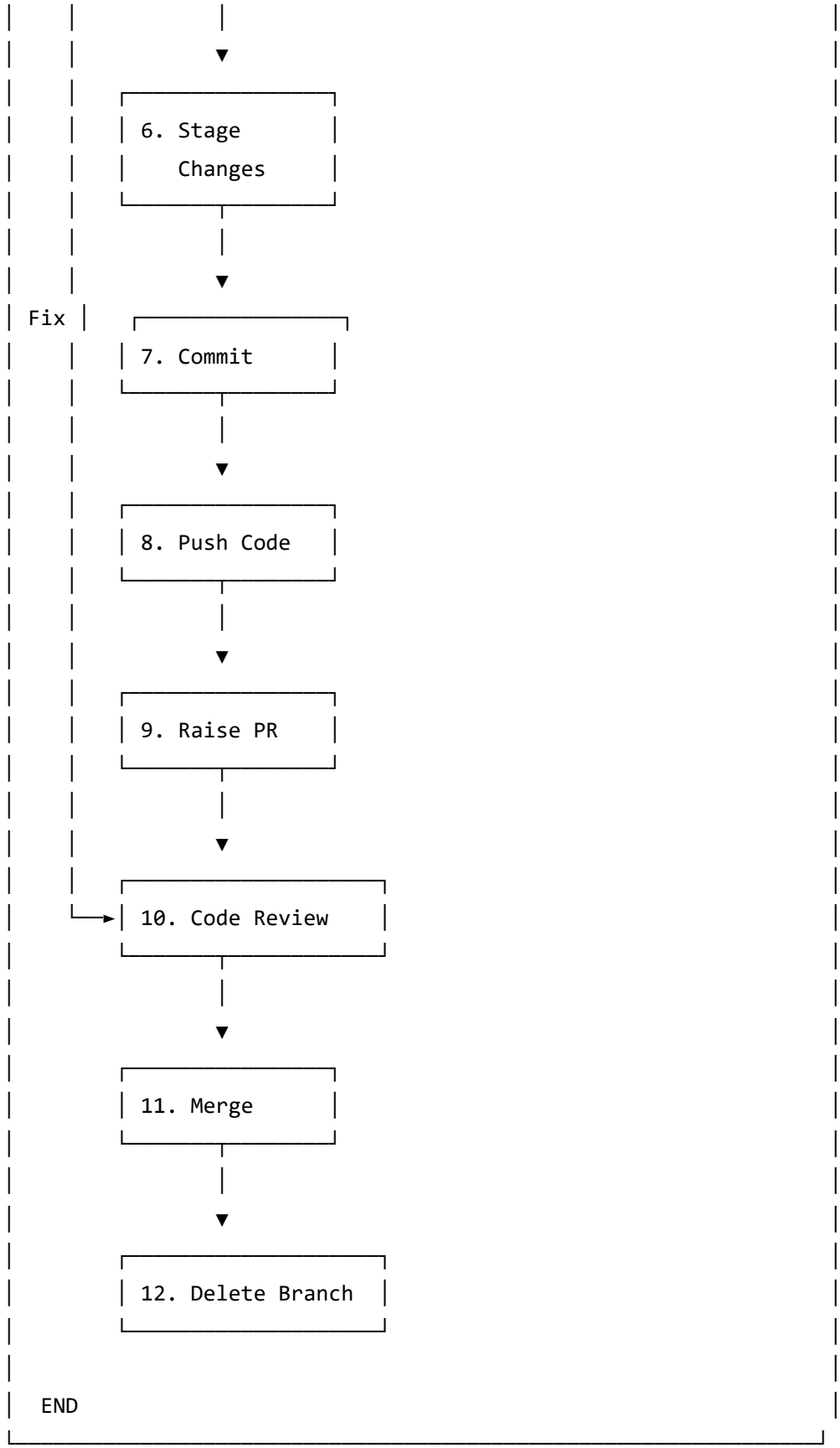
2.7 When to Create Which Branch?

Scenario	Branch Type	Example
Writing new tests for a feature	`feature/*`	`feature/checkout-tests`
Fixing a broken test	`bugfix/*`	`bugfix/login-locator-fix`
Urgent fix needed in production	`hotfix/*`	`hotfix/critical-failure`
Preparing a new release	`release/*`	`release/v2.0.0`

SECTION 3 — Daily Tester Workflow (Step-by-Step)

3.1 Complete Workflow Overview




6. Stage Changes
Fix
7. Commit
8. Push Code
9. Raise PR
10. Code Review
11. Merge
12. Delete Branch
END

3.2 Step 1: Clone Repository (First Time Only)

What is Cloning?

Cloning creates a copy of the GitHub repository on your local computer.

Command:

```
git clone https://github.com/company/automation-framework.git
```

What Happens Internally:

1. Git contacts GitHub server
2. Downloads all files and history
3. Creates a folder with the repository name
4. Sets up connection to GitHub (remote)

After Cloning:

```
cd automation-framework    # Enter the folder
npm install                 # Install dependencies
```

3.3 Step 2: Pull Latest Code (Every Day)

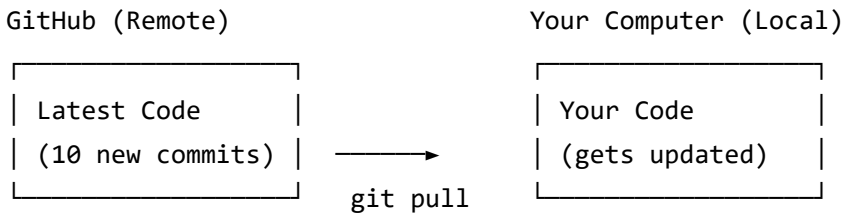
What is Pulling?

Getting the latest changes from GitHub that others have pushed.

Command:

```
# Make sure you're on develop branch
git checkout develop

# Pull latest changes
git pull origin develop
```

What Happens Internally:



Common Mistake:

❌ Forgetting to pull before starting work = working on outdated code

3.4 Step 3: Create Feature Branch

Command:

```
# Create and switch to new branch
git checkout -b feature/login-tests
```

What Happens Internally:

1. Git creates a new branch pointer
2. Copies current develop state
3. Switches your working directory to new branch

Verify Branch:

```
git branch    # Shows all branches, * marks current
```

Output:

```
  develop
* feature/login-tests    ← You are here
  main
```

3.5 Step 4: Add or Modify Test Cases

Example: Creating a new test file

```
// tests/login/login-validation.spec.ts
import { test, expect } from '@playwright/test';
import { LoginPage } from '../../pages/LoginPage';

test.describe('Login Validation Tests', () => {
    test('should show error for empty username', async ({ page }) => {
        const loginPage = new LoginPage(page);
        await loginPage.navigate();
        await loginPage.fillLoginForm('', 'password123');
        await loginPage.submitForm();

        await expect(loginPage.errorMessage).toBeVisible();
    });
});
```

3.6 Step 5: Run Tests Locally

Why Run Locally First?

- Catch errors before pushing
- Faster feedback loop
- Don't break the pipeline for others

Commands:

```
# Run all tests
npx playwright test

# Run specific test file
npx playwright test tests/login/login-validation.spec.ts

# Run with browser visible (headed mode)
npx playwright test --headed

# Run with debug mode
npx playwright test --debug
```

Expected Output:

```
Running 5 tests using 4 workers

  ✓ login-validation.spec.ts:5:5 › should show error for empty username (2s)
  ✓ login-validation.spec.ts:15:5 › should show error for empty password (1s)
  ✓ login-validation.spec.ts:25:5 › should login with valid credentials (3s)
  ✓ login-validation.spec.ts:35:5 › should show error for invalid credentials (2s)
  ✓ login-validation.spec.ts:45:5 › should redirect after successful login (2s)

  5 passed (10s)
```

3.7 Step 6: Stage Changes

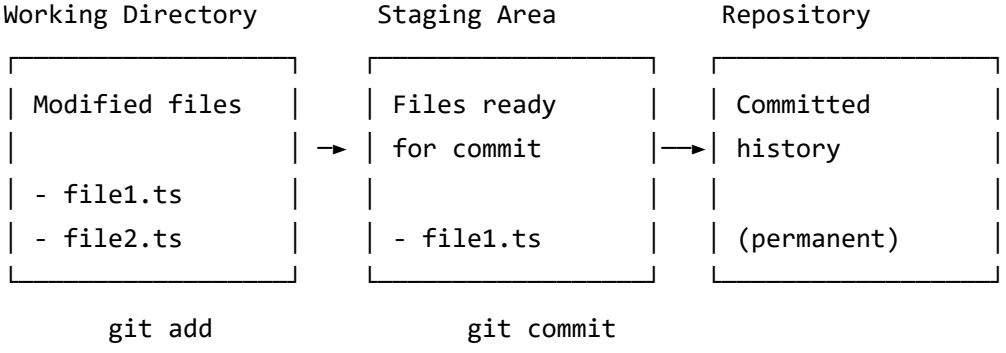
What is Staging?

Marking specific files to be included in the next commit.

Commands:

```bash
# See what files changed
git status

# Stage specific file
git add tests/login/login-validation.spec.ts

# Stage all changed files
git add .

# Stage specific folder
git add tests/
```

What Happens Internally:



3.8 Step 7: Commit Changes

What is a Commit?

A snapshot of your staged changes with a message describing what you did.

Command:

```
git commit -m "Add login validation test cases"
```

Commit Message Standards (Enterprise):

Type	Description	Example
`feat:`	New feature/test	`feat: add login validation tests`
`fix:`	Bug fix	`fix: update login button locator`
`refactor:`	Code improvement	`refactor: simplify page object methods`
`docs:`	Documentation	`docs: update README with setup steps`
`chore:`	Maintenance	`chore: update dependencies`

Good vs Bad Commit Messages:

❌ Bad	✅ Good
`update`	`feat: add checkout page tests`
`fix bug`	`fix: resolve timeout in payment test`
`changes`	`refactor: extract common login helper`
`test`	`feat: add data-driven login tests`

3.9 Step 8: Push Code

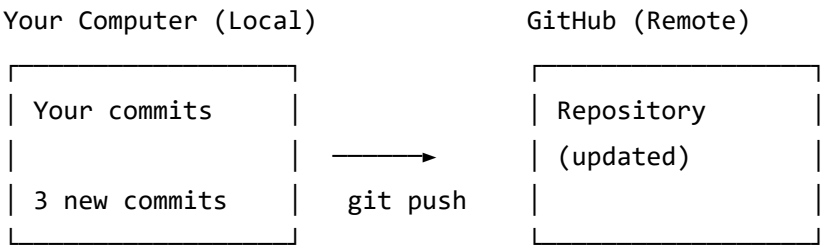
What is Pushing?

Sending your local commits to GitHub.

Command:

```
git push origin feature/login-tests
```

What Happens Internally:



First Time Push (new branch):

```
git push -u origin feature/login-tests
```

The `-u` sets upstream tracking, so future pushes only need `git push`.

3.10 Step 9: Raise Pull Request (PR)

What is a Pull Request?
A request to merge your branch into another branch (usually develop).

Steps to Create PR:

1. Go to GitHub repository
2. Click "Pull Requests" tab
3. Click "New Pull Request"
4. Select: base = `develop`, compare = `feature/login-tests`
5. Fill in title and description
6. Click "Create Pull Request"

PR Title Format:

```
[JIRA-123] feat: Add login validation test cases
```

PR Description Template:

```
## Summary
Added 5 new test cases for login validation.

## Changes
- Added login-validation.spec.ts
- Added new test data for edge cases
- Updated LoginPage with error message locator

## Testing
- All tests pass locally
- Tested on Chrome and Firefox

## Checklist
- [ ] Tests pass locally
- [ ] Code follows standards
- [ ] No hardcoded values
- [ ] Added appropriate test tags
```

3.11 Step 10: Code Review Process

What Happens:

1. PR is created
2. Reviewers are notified
3. Reviewers check the code
4. Comments/suggestions are added
5. Author makes changes if needed
6. Reviewers approve

Responding to Review Comments:

```
# Make requested changes in your code
# Then commit and push

git add .
git commit -m "fix: address review comments"
git push origin feature/login-tests
```

Common Review Feedback:

Feedback	Action
"Add assertion for URL"	Add `expect(page).toHaveURL(...)`
"Remove hardcoded wait"	Replace `waitForTimeout` with proper wait
"Use test data file"	Move data to test-data folder
"Add test tags"	Add `@smoke` or `@regression` tags

3.12 Step 11: Merge Strategy

After Approval:

1. Click "Merge Pull Request"
2. Choose merge type (usually "Squash and merge" for clean history)
3. Confirm merge

Merge Types:

Type	When to Use
Merge commit	Preserve all commit history
Squash and merge	Combine all commits into one (recommended)
Rebase and merge	Linear history, no merge commits

3.13 Step 12: Branch Cleanup

After Merge:

```
# Switch back to develop
git checkout develop

# Pull the merged changes
git pull origin develop

# Delete local feature branch
git branch -d feature/login-tests

# Delete remote branch (if not auto-deleted)
git push origin --delete feature/login-tests
```

3.14 Common Mistakes and Solutions

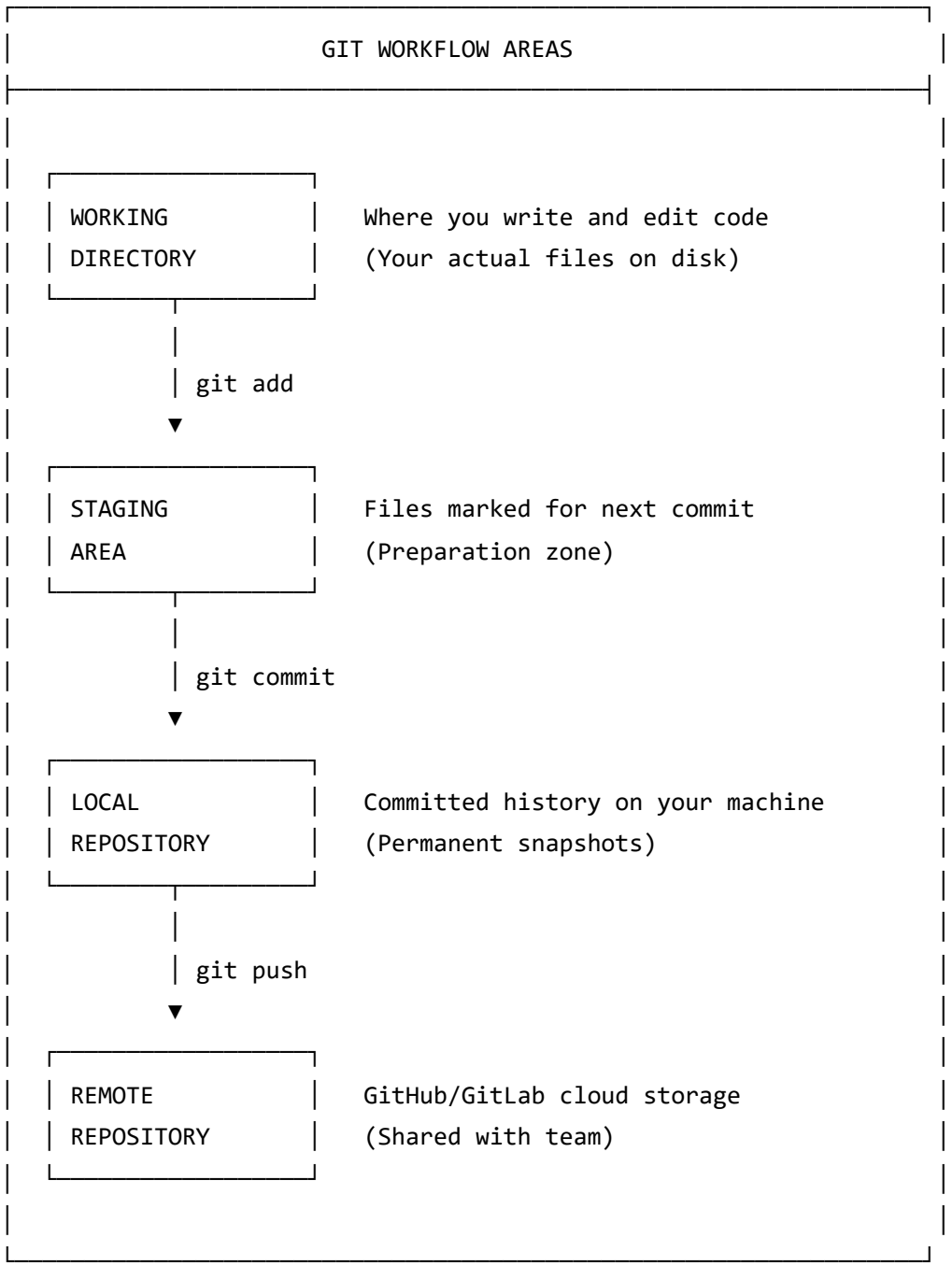
Mistake	Impact	Solution
Not pulling before starting	Merge conflicts later	Always `git pull` first
Committing to wrong branch	Changes in wrong place	Use `git stash` and switch
Forgetting to run tests	Broken pipeline	Make it a habit before push
Vague commit messages	Hard to track changes	Follow commit message standards
Large PRs with many changes	Hard to review	Keep PRs small and focused
Not responding to reviews	PR gets stale	Respond within 24 hours

Continue to Section 4-6...

SECTIONS 4-6: Git Concepts, CI/CD Fundamentals & GitHub Actions

SECTION 4 — Git Concepts Explained Simply

4.1 The Three Areas of Git



4.2 Working Directory

What It Is:

The folder on your computer where you edit files.

Analogy: Your desk where you're actively working on papers.

Key Points:

- Contains actual files you can see and edit
- Changes here are NOT saved to Git yet
- If you delete a file, it's gone (unless committed)

Commands:

```
# See what changed in working directory
git status

# Discard changes in working directory
git checkout -- filename.ts
```

4.3 Staging Area (Index)

What It Is:
A preparation zone for files you want to commit.

Analogy: An "Outbox" on your desk - papers you've reviewed and ready to file.

Why It Exists:

- Pick specific files to commit
- Review changes before committing
- Create focused commits

Commands:

```
# Add file to staging
git add filename.ts

# Add all files
git add .

# Remove from staging (keep changes in working directory)
git reset filename.ts

# See what's staged
git status
```

4.4 Commits

What It Is:
A permanent snapshot of staged changes with a message.

Analogy: Making a photocopy and filing it. The original paper can change, but the copy is permanent.

Key Points:

- Each commit has a unique ID (hash)
- Contains author, date, message, and changes
- You can always go back to any commit

Commands:

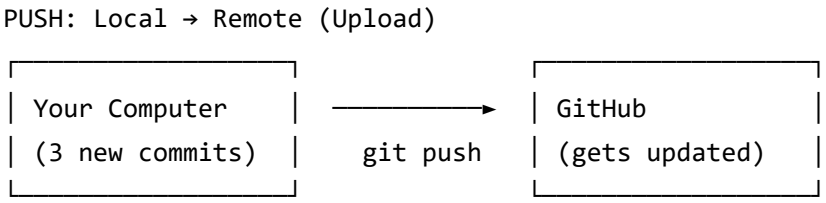
```
# Create a commit
git commit -m "Your message"

# See commit history
git log --oneline

# See details of a commit
git show <commit-hash>
```

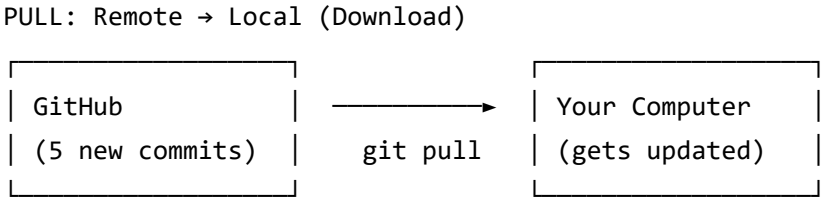
Example Output:

```
a1b2c3d (HEAD -> feature/login) feat: add login tests
d4e5f6g fix: update locator
g7h8i9j refactor: simplify page object
```

4.5 Push vs Pull

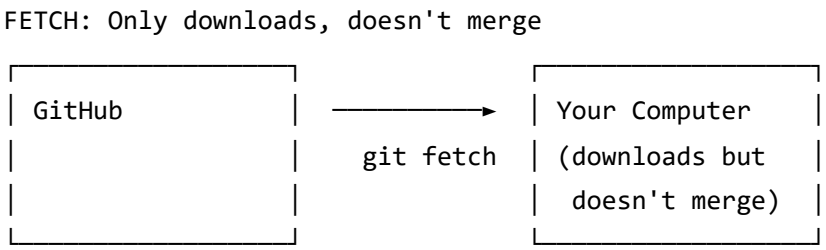
Visual Comparison:





Aspect	Push	Pull
Direction	Local to Remote	Remote to Local
Purpose	Share your work	Get team's work
When	After committing	Before starting work
Command	`git push origin branch`	`git pull origin branch`

4.6 Fetch vs Pull



Aspect	Fetch	Pull
Downloads changes	✅ Yes	✅ Yes

Aspect	Fetch	Pull
Merges changes	❌ No	✅ Yes
Safe	Safer	Can cause conflicts
Use case	Review first	Get latest quickly

When to Use Fetch:

```
git fetch origin
git log origin/develop    # See what's new
git merge origin/develop  # Then merge if happy
```

4.7 Merge vs Rebase

Merge: Creates a merge commit

```
Before merge:
main:    A──B──C
              \
feature:       D──E

After merge:
main:    A──B──C──M (merge commit)
              \     /
feature:       D──E
```

Rebase: Moves commits to new base

```
Before rebase:
main:    A──B──C
              \
feature:       D──E

After rebase:
main:    A──B──C
                  \
feature:           D'──E' (re-applied commits)
```

Aspect	Merge	Rebase
History	Preserves all branches	Linear history
Commit	Creates merge commit	No extra commits
Safety	Safer, non-destructive	Rewrites history
Team	Better for shared branches	Better for local branches
Command	`git merge branch`	`git rebase main`

Enterprise Recommendation:

- Use **merge** for shared branches (develop, main)
- Use **rebase** for local feature branches before PR

4.8 Conflict Resolution

What is a Conflict?

When Git can't automatically merge because the same lines were changed.

When Conflicts Happen:

```
You:      Changed line 10 to "Hello"
Teammate: Changed line 10 to "Hi"
Git:      🤷 Which one to keep?
```

Conflict Markers:

```
<<<<<<< HEAD
// Your version
const greeting = "Hello";
=======
// Their version
const greeting = "Hi";
>>>>>>> feature/other-branch
```

How to Resolve:

1. **Find conflicted files:**

```
git status    # Shows files with conflicts
```

2. **Open file and decide which version to keep:**

```
// After resolution - pick one or combine
const greeting = "Hello World";
```

3. **Remove conflict markers:**
   Delete the `<<<<<<<` , `=======` , and `>>>>>>>` lines.
4. **Stage and commit:**

```
git add filename.ts
git commit -m "fix: resolve merge conflict in greeting"
```

Conflict Resolution Tips:

Tip	Description
Communicate	Talk to teammate who made the change
Understand both	Know why each change was made
Test after	Run tests after resolving
Small commits	Smaller PRs = fewer conflicts
Pull often	Frequent pulls reduce conflicts

4.9 Why Force Push is Dangerous

What is Force Push?

```
git push --force    # ⚠️ DANGEROUS
git push -f         # ⚠️ SAME THING
```

Why It's Dangerous:

```
BEFORE your force push:
Remote: A──B──C──D──E (teammate's work)
Your:   A──B──X──Y

AFTER your force push:
Remote: A──B──X──Y     (D and E are GONE!)
```

When Force Push Destroys Work:

1. Teammate pushed commits D and E
2. You didn't pull their changes
3. You force pushed your version
4. Their commits D and E are **permanently deleted**

Safe Alternatives:

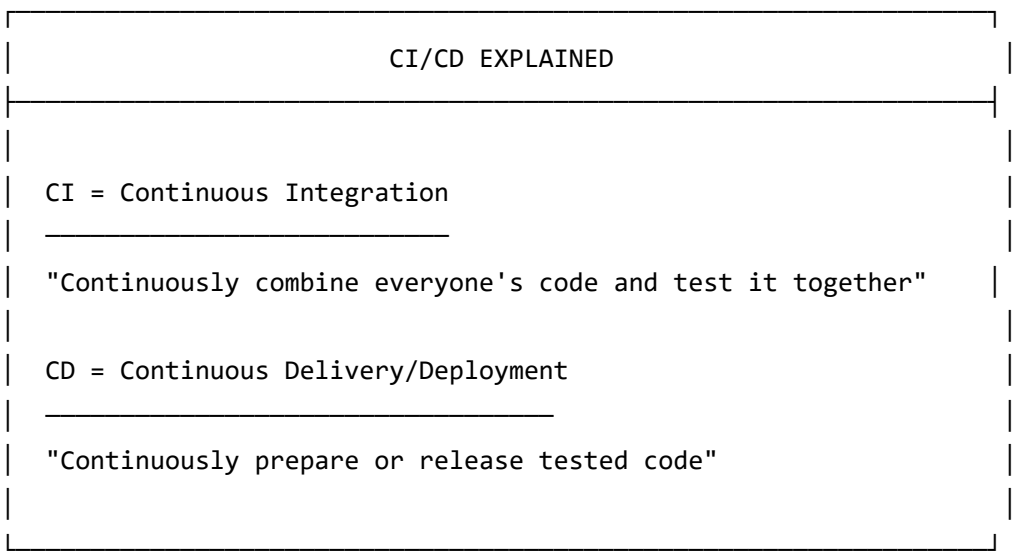
Instead of	Use	Why
`git push --force`	`git push --force-with-lease`	Fails if remote has new commits
Force pushing to shared branch	Merge or rebase properly	Don't rewrite shared history

Enterprise Rule:

> ⛔ NEVER force push to `main` or `develop` branches
> ⚠️ Only force push to your own feature branches when necessary

SECTION 5 — CI/CD Fundamentals (Beginner Friendly)

5.1 What is CI/CD?

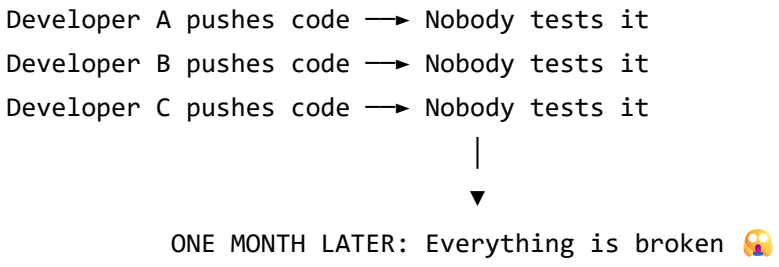


Simple Analogy:

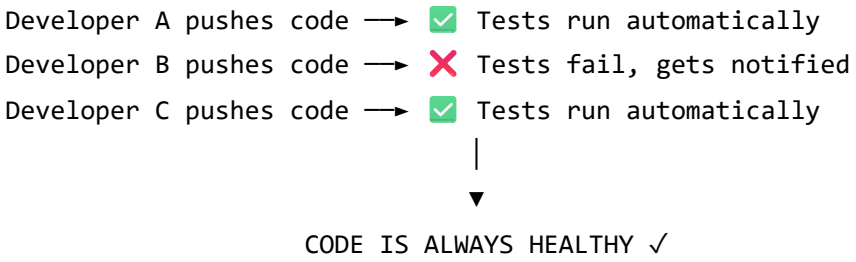
Concept	Analogy
CI	Restaurant kitchen automatically checking every dish before serving
CD	Automatically preparing dishes to be served when ready

5.2 Why Enterprises Need CI/CD

Without CI/CD:



With CI/CD:

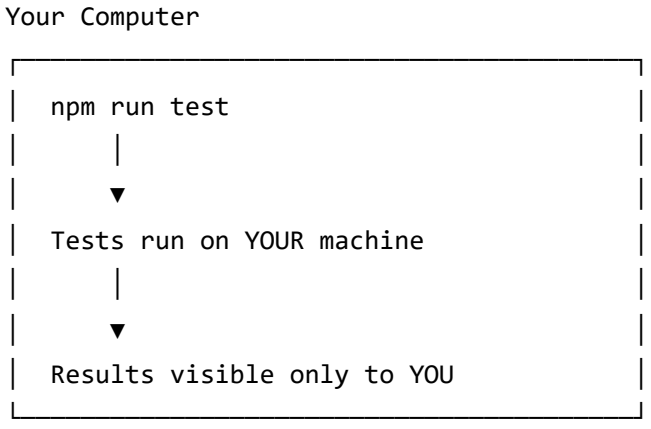


Benefits for Enterprises:

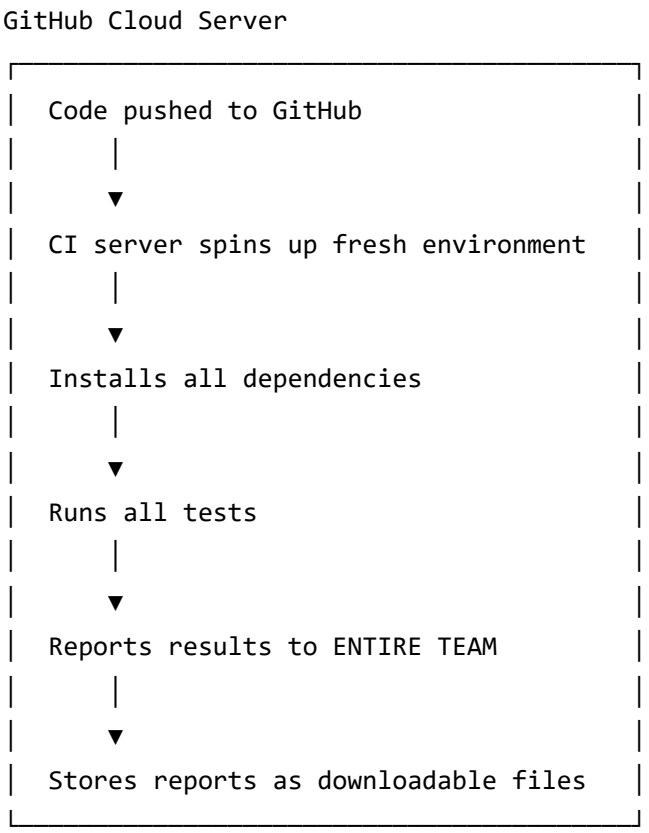
Benefit	Description
Early Detection	Find bugs immediately, not after a month
Consistent Testing	Same tests run every time, in same environment
Time Savings	No manual test execution needed
Quality Assurance	Code can't merge if tests fail
Team Visibility	Everyone sees test results
Audit Trail	Complete history of all test runs

5.3 Local Execution vs CI Execution

Local Execution:



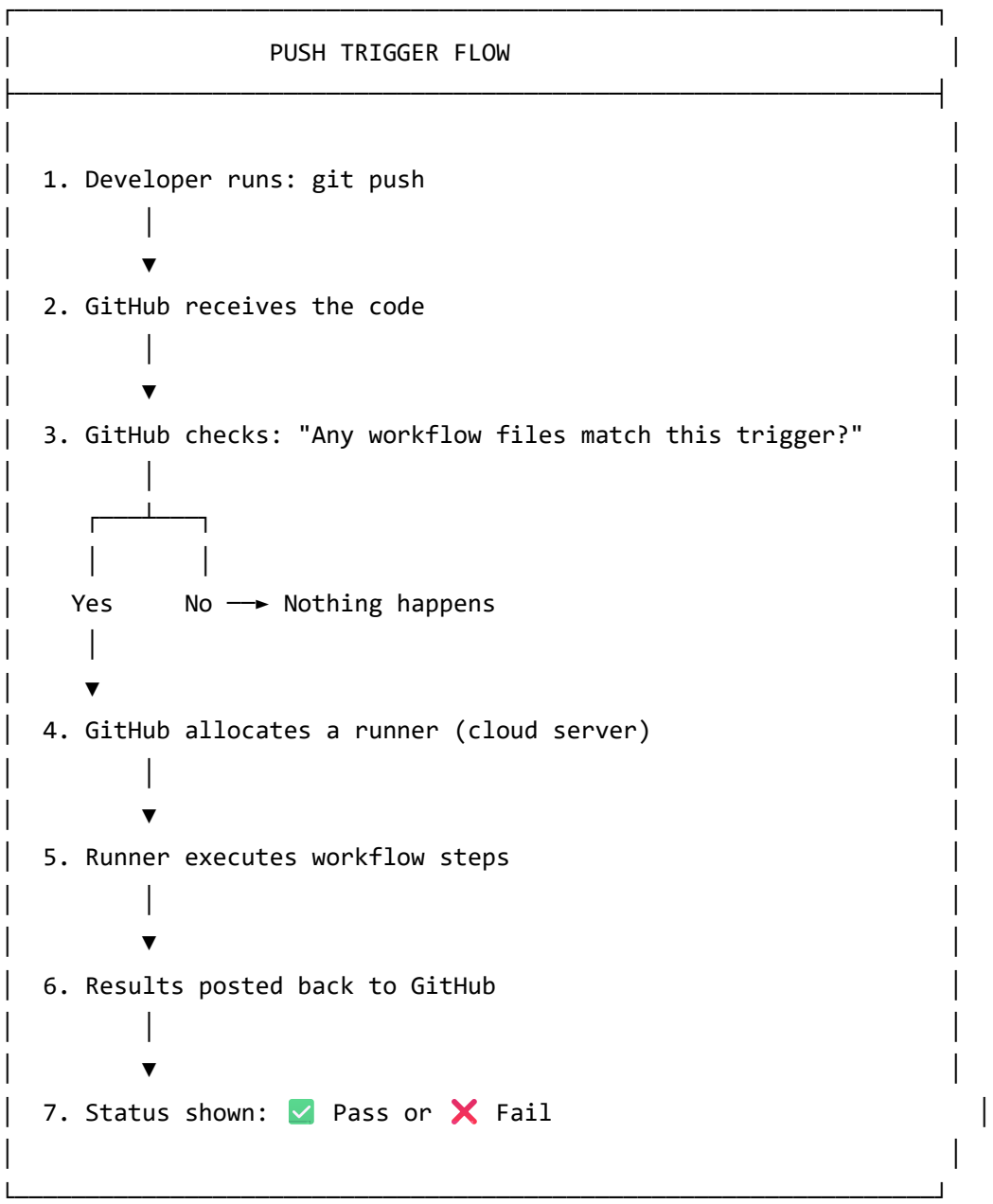
CI Execution:



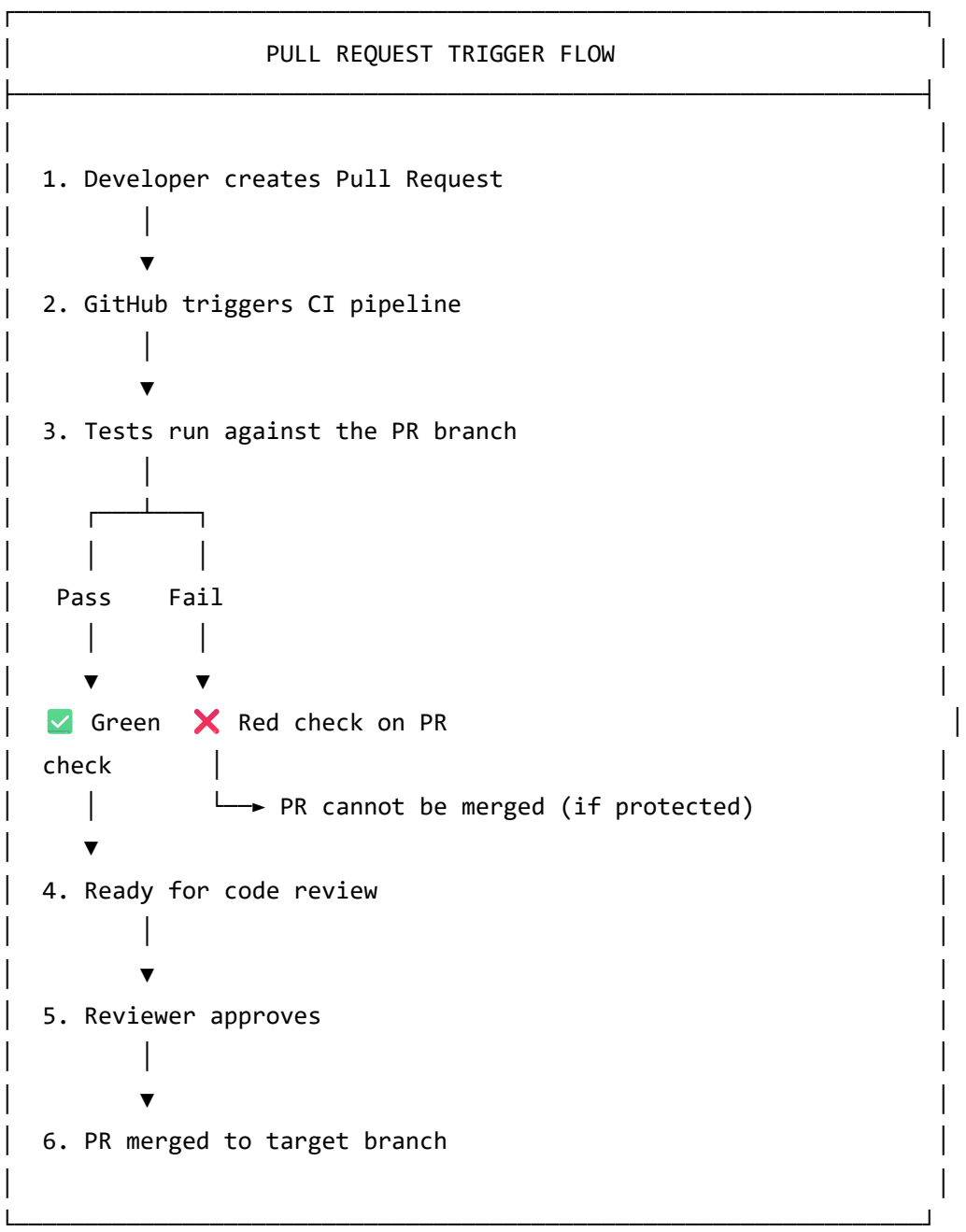
Key Differences:

Aspect	Local	CI
Environment	Your machine	Fresh cloud server
Dependencies	Already installed	Installed fresh each time
Consistency	Varies by machine	Same every time
Visibility	Only you	Entire team
Reports	Local folder	Cloud artifacts
Speed	Faster (no setup)	Slower (includes setup)

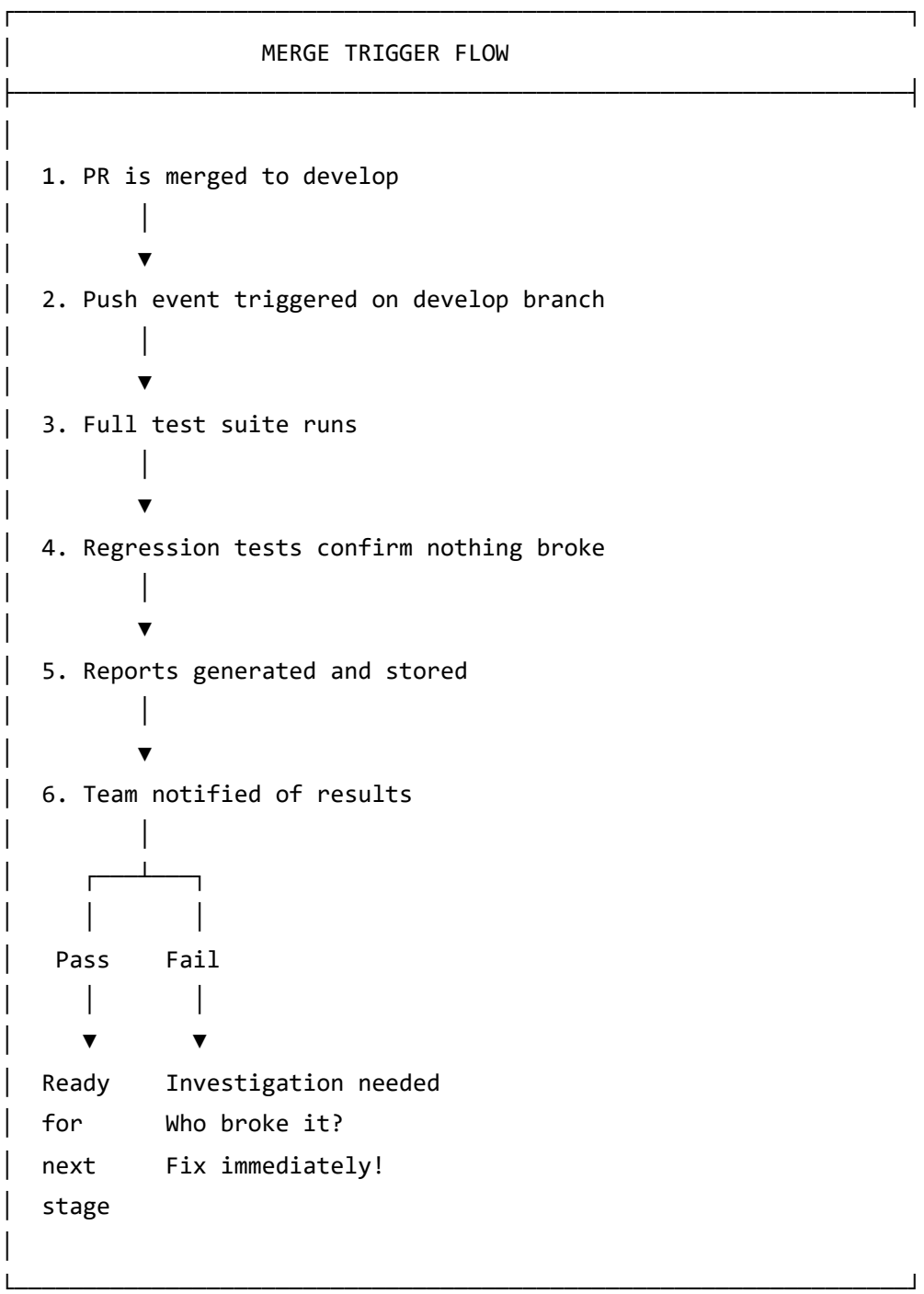
5.4 What Happens When Code is Pushed



5.5 What Happens When PR is Created



5.6 What Happens When PR is Merged



SECTION 6 — GitHub Actions (Deep Dive)

6.1 What are GitHub Actions?

Definition:
GitHub Actions is a CI/CD platform built directly into GitHub that allows you to automate workflows.

Simple Explanation:
"GitHub Actions is like a robot assistant that watches your repository and automatically does tasks when certain events happen."

Examples of Tasks:

- Run tests when code is pushed
- Build and deploy application
- Send notifications
- Generate reports
- Label issues automatically

6.2 Workflow File Structure

Location:

```
your-repo/
└── .github/
    └── workflows/
        └── playwright.yml    ← Workflow file
```

Basic Structure:

```yaml
name: Workflow Name              # Display name

on:                              # Triggers
  push:
    branches: [main]

jobs:                            # What to do
  job-name:
    runs-on: ubuntu-latest
    steps:
      - name: Step 1
        run: echo "Hello"
```

6.3 Complete Workflow Anatomy

```yaml
# ═══════════════════════════════════════════════════════════════
# NAME: What appears in GitHub Actions tab
# ═══════════════════════════════════════════════════════════════
name: Playwright Tests

# ═══════════════════════════════════════════════════════════════
# ON: When should this workflow run?
# ═══════════════════════════════════════════════════════════════
on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]
  workflow_dispatch:
  schedule:
    - cron: '0 0 * * *'

# ═══════════════════════════════════════════════════════════════
# JOBS: What work should be done?
# ═══════════════════════════════════════════════════════════════
jobs:
  test:
    name: Run Tests
    runs-on: ubuntu-latest
    timeout-minutes: 60

    # ═══════════════════════════════════════════════════════
    # STEPS: Individual tasks in sequence
    # ═══════════════════════════════════════════════════════
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: 20

      - name: Install dependencies
        run: npm ci
```

```
      - name: Run tests
        run: npx playwright test
        env:
          BASE_URL: ${{ secrets.BASE_URL }}

      - name: Upload report
        uses: actions/upload-artifact@v4
        if: always()
        with:
          name: playwright-report
          path: playwright-report/
```

6.4 Jobs Explained

What is a Job?

A job is a set of steps that execute on the same runner (server).

```
jobs:
  # Job 1: Run tests
  test:
    runs-on: ubuntu-latest
    steps:
      - run: npx playwright test

  # Job 2: Deploy (only if tests pass)
  deploy:
    needs: test              # Wait for 'test' job
    runs-on: ubuntu-latest
    steps:
      - run: npm run deploy
```

Job Properties:

Property	Description	Example
`runs-on`	Operating system	`ubuntu-latest` , `windows-latest`
`timeout-minutes`	Max job duration	`60`
`needs`	Job dependencies	`needs: test`
`if`	Conditional execution	`if: success()`

Property	Description	Example
`env`	Environment variables	`env: MY_VAR: value`
`strategy`	Matrix configuration	Browser matrix

6.5 Steps Explained

What is a Step?

A single task within a job. Steps run in sequence.

Two Types of Steps:

```yaml
steps:
  # TYPE 1: Action (uses)
  - name: Checkout code
    uses: actions/checkout@v4

  # TYPE 2: Command (run)
  - name: Run tests
    run: npx playwright test
```

Step Properties:

Property	Description	Example
`name`	Display name	`Install dependencies`
`uses`	Action to use	`actions/checkout@v4`
`run`	Command to execute	`npm install`
`with`	Action inputs	`node-version: 20`
`env`	Environment variables	`CI: true`
`if`	Conditional	`if: failure()`
`continue-on-error`	Don't fail job	`true`

6.6 Runners Explained

What is a Runner?

A server (virtual machine) that executes your workflow.

Types of Runners:

Type	Description	Cost
GitHub-hosted	Managed by GitHub	Free (limits apply)
Self-hosted	Your own servers	Your infrastructure

Available GitHub-Hosted Runners:

Runner	OS	Use Case
`ubuntu-latest`	Linux	Most common, fastest
`windows-latest`	Windows	Windows-specific tests
`macos-latest`	macOS	Safari/iOS testing

6.7 Secrets Explained

What are Secrets?

Encrypted environment variables for sensitive data.

How to Create Secrets:

1. Go to repository → Settings
2. Click Secrets and variables → Actions
3. Click "New repository secret"
4. Add name and value

Using Secrets in Workflow:

```yaml
steps:
  - name: Run tests
    run: npx playwright test
    env:
      TEST_USERNAME: ${{ secrets.TEST_USERNAME }}
      TEST_PASSWORD: ${{ secrets.TEST_PASSWORD }}
      API_KEY: ${{ secrets.API_KEY }}
```

Secret Levels:

Level	Scope	Use Case
Repository	Single repo	Repo-specific credentials
Organization	All repos in org	Shared credentials
Environment	Specific env	Dev vs Prod secrets

6.8 Environment Variables

Two Ways to Set:

```yaml
# Level 1: Workflow level
env:
  NODE_ENV: test

jobs:
  test:
    # Level 2: Job level
    env:
      DEBUG: pw:api

    steps:
      - name: Run tests
        # Level 3: Step level
        env:
          HEADLESS: true
        run: npx playwright test
```

Built-in Variables:

Variable	Description
`${{ github.repository }}`	Owner/repo name
`${{ github.ref }}`	Branch/tag ref
`${{ github.sha }}`	Commit SHA
`${{ github.actor }}`	Username who triggered
`${{ github.run_id }}`	Unique run ID

6.9 Triggers Deep Dive

Push Trigger

```yaml
on:
  push:
    branches:
      - main
      - develop
      - 'feature/**'        # Wildcard: all feature branches
    paths:
      - 'tests/**'          # Only if tests changed
      - '!docs/**'          # Ignore docs changes
```

Pull Request Trigger

```yaml
on:
  pull_request:
    branches:
      - main
    types:                  # Types of PR events
      - opened
      - synchronize         # New commits pushed
      - reopened
```

Schedule Trigger (Cron)

```yaml
on:
  schedule:
    - cron: '0 0 * * *'     # Daily at midnight UTC
    - cron: '0 9 * * 1'     # Every Monday at 9 AM UTC
```

Cron Syntax:

```
┌──────────── minute (0 - 59)
│ ┌──────────── hour (0 - 23)
│ │ ┌──────────── day of month (1 - 31)
│ │ │ ┌──────────── month (1 - 12)
│ │ │ │ ┌──────────── day of week (0 - 6, Sun-Sat)
│ │ │ │ │
* * * * *
```

Schedule	Cron	Description
Daily midnight	`0 0 * * *`	Every day at 00:00 UTC
Weekdays 9 AM	`0 9 * * 1-5`	Mon-Fri at 9 AM UTC
Hourly	`0 * * * *`	Every hour
Weekly Monday	`0 0 * * 1`	Every Monday midnight

Manual Trigger (workflow_dispatch)

```yaml
on:
  workflow_dispatch:
    inputs:
      environment:
        description: 'Environment to test'
        required: true
        default: 'staging'
        type: choice
        options:
          - development
          - staging
          - production
      browser:
        description: 'Browser to use'
        required: false
        default: 'chromium'
```

Using Input Values:

```yaml
steps:
  - name: Run tests
    run: npx playwright test --project=${{ inputs.browser }}
```

6.10 YAML Keywords Summary

Keyword	Purpose	Example
`name`	Display name	`name: CI Pipeline`
`on`	Triggers	`on: push`
`jobs`	Job definitions	`jobs: test:`
`runs-on`	Runner OS	`runs-on: ubuntu-latest`
`steps`	Job steps	`steps: - run: test`
`uses`	Use an action	`uses: actions/checkout@v4`
`run`	Run command	`run: npm test`

Keyword	Purpose	Example
`with`	Action inputs	`with: node-version: 20`
`env`	Environment vars	`env: CI: true`
`if`	Conditional	`if: failure()`
`needs`	Job dependency	`needs: build`
`strategy`	Matrix/parallel	`strategy: matrix:`
`secrets`	Access secrets	`secrets.TOKEN`

Continue to Section 7-9...

SECTIONS 7-9: CI/CD Pipeline, Employee Usage & Multi-Team Management

SECTION 7 — CI/CD Pipeline for Playwright

Framework

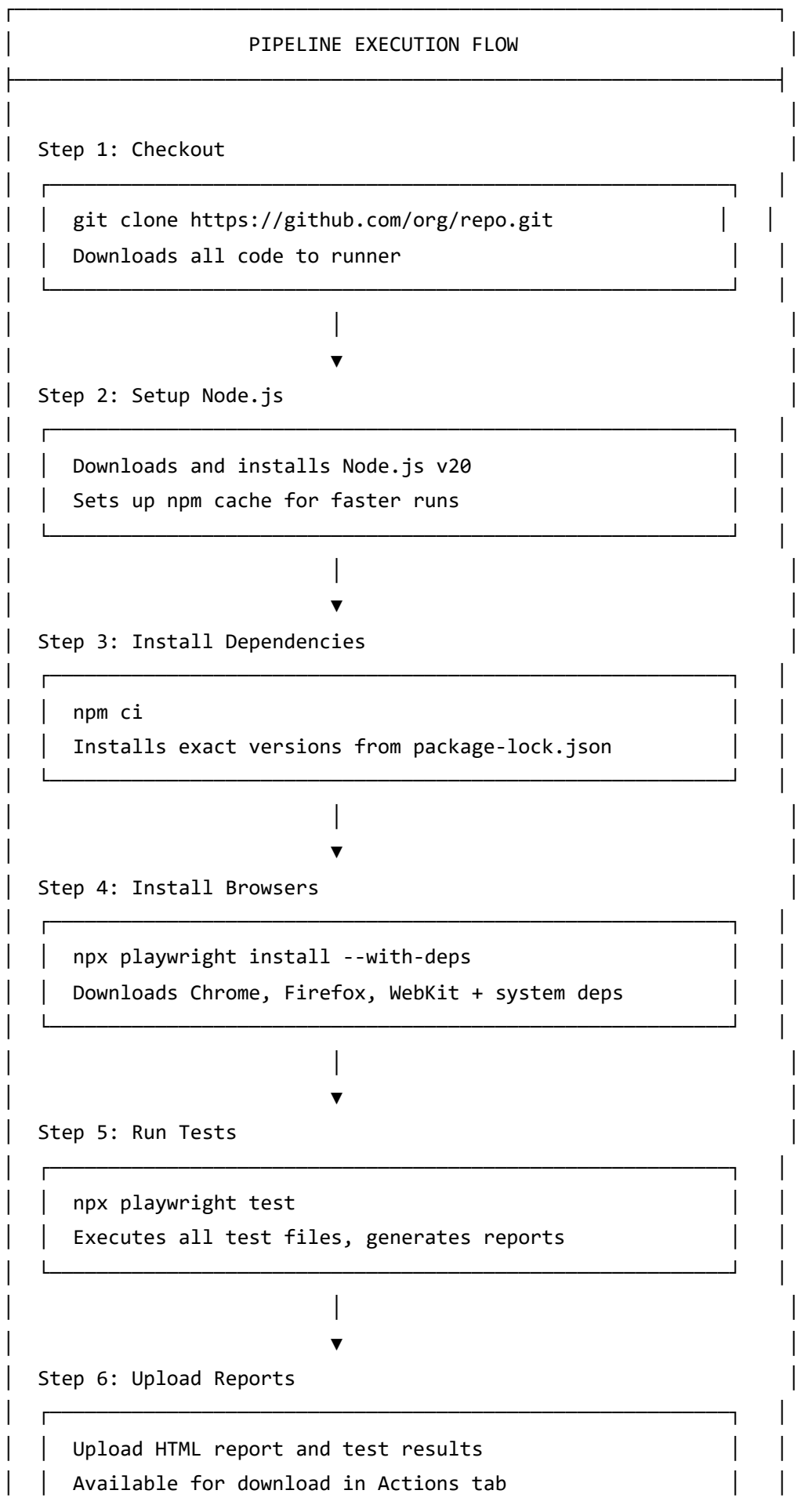
7.1 Complete Pipeline Configuration

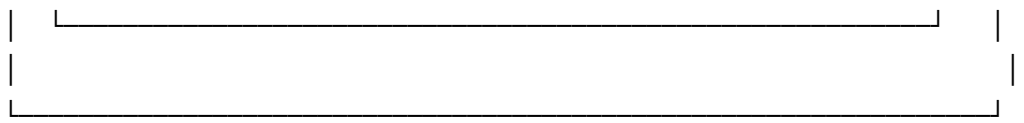
```yaml
# .github/workflows/playwright.yml
name: Playwright Tests

on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main, develop]
  workflow_dispatch:
    inputs:
      environment:
        description: 'Test environment'
        required: true
        default: 'staging'
        type: choice
        options:
          - development
          - staging
          - production

jobs:
  test:
    name: Run Playwright Tests
    runs-on: ubuntu-latest
    timeout-minutes: 60

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: 20
          cache: 'npm'

      - name: Install dependencies
        run: npm ci
```

```yaml
- name: Install Playwright browsers
  run: npx playwright install --with-deps

- name: Run Playwright tests
  run: npx playwright test
  env:
    BASE_URL: ${{ secrets.BASE_URL }}
    TEST_USERNAME: ${{ secrets.TEST_USERNAME }}
    TEST_PASSWORD: ${{ secrets.TEST_PASSWORD }}

- name: Upload HTML report
  uses: actions/upload-artifact@v4
  if: always()
  with:
    name: playwright-report-${{ github.run_id }}
    path: playwright-report/
    retention-days: 30

- name: Upload test results
  uses: actions/upload-artifact@v4
  if: failure()
  with:
    name: test-results-${{ github.run_id }}
    path: test-results/
    retention-days: 7
```

7.2 Running Tests in Pipeline

Step-by-Step Execution:


PIPELINE EXECUTION FLOW
Step 1: Checkout
git clone https://github.com/org/repo.git
Downloads all code to runner
Step 2: Setup Node.js
Downloads and installs Node.js v20
Sets up npm cache for faster runs
Step 3: Install Dependencies
npm ci
Installs exact versions from package-lock.json
Step 4: Install Browsers
npx playwright install --with-deps
Downloads Chrome, Firefox, WebKit + system deps
Step 5: Run Tests
npx playwright test
Executes all test files, generates reports
Step 6: Upload Reports
Upload HTML report and test results
Available for download in Actions tab



7.3 Generating HTML Reports

Playwright Config for Reports:

```
// playwright.config.ts
import { defineConfig } from '@playwright/test';

export default defineConfig({
    reporter: [
        ['html', {
            open: 'never',
            outputFolder: 'playwright-report'
        }],
        ['list'],                    // Console output
        ['json', {
            outputFile: 'test-results.json'
        }],
        ['junit', {
            outputFile: 'junit-results.xml'
        }]
    ],
});
```

Report Types:

Reporter	Output	Use Case
`html`	Interactive HTML	Human review
`list`	Console text	Quick CI feedback
`json`	JSON file	Integrations
`junit`	XML file	CI tools (Jenkins)

7.4 Uploading Reports as Artifacts

Basic Upload:

```yaml
- name: Upload HTML report
  uses: actions/upload-artifact@v4
  if: always()                    # Upload even if tests fail
  with:
    name: playwright-report
    path: playwright-report/
    retention-days: 30
```

Multiple Artifacts:

```yaml
- name: Upload all reports
  uses: actions/upload-artifact@v4
  if: always()
  with:
    name: test-artifacts
    path: |
      playwright-report/
      test-results/
      videos/
      screenshots/
```

Downloading Artifacts:

1. Go to repository → Actions tab
2. Click on completed workflow run
3. Scroll to "Artifacts" section
4. Click artifact name to download ZIP

7.5 Handling Failures

Conditional Steps Based on Failure:

```yaml
steps:
  - name: Run tests
    id: test
    run: npx playwright test
    continue-on-error: true      # Don't stop pipeline

  - name: Upload failure screenshots
    if: failure()                # Only on failure
    uses: actions/upload-artifact@v4
    with:
      name: failure-screenshots
      path: test-results/**/*.png

  - name: Notify team on failure
    if: failure()
    run: |
      echo "Tests failed! Notifying team..."
      # Add Slack/Teams notification here

  - name: Check test results
    if: always()
    run: |
      if [ "${{ steps.test.outcome }}" == "failure" ]; then
        echo "Some tests failed - check artifacts"
        exit 1
      fi
```

Failure Notification Example (Slack):

```yaml
- name: Notify Slack on failure
  if: failure()
  uses: 8398a7/action-slack@v3
  with:
    status: failure
    text: 'Playwright tests failed on ${{ github.ref }}'
    webhook_url: ${{ secrets.SLACK_WEBHOOK }}
```

7.6 Retrying Flaky Tests

Method 1: Playwright Built-in Retries

```
// playwright.config.ts
export default defineConfig({
    retries: process.env.CI ? 2 : 0,    // 2 retries in CI, 0 locally
});
```

Method 2: Test-Level Retries

```
test('flaky test', async ({ page }) => {
    test.info().config.retries = 3;     // This test gets 3 retries
    // test code
});
```

Method 3: Retry Failed Tests Only

```
- name: Run tests (first attempt)
  id: first-run
  run: npx playwright test
  continue-on-error: true

- name: Retry failed tests
  if: steps.first-run.outcome == 'failure'
  run: npx playwright test --last-failed    # Playwright 1.40+
```

7.7 Pipeline Best Practices

Practice	Description
Use `npm ci`	Faster, reliable dependency install
Cache dependencies	Reduce install time
Set timeouts	Prevent stuck jobs
Upload on `always()`	Get reports even on failure
Use matrix for browsers	Test all browsers
Shard for speed	Split tests across runners

Caching Example:

```yaml
- name: Setup Node.js
  uses: actions/setup-node@v4
  with:
    node-version: 20
    cache: 'npm'          # Caches npm dependencies

- name: Cache Playwright browsers
  uses: actions/cache@v3
  with:
    path: ~/.cache/ms-playwright
    key: playwright-${{ runner.os }}-${{ hashFiles('package-lock.json') }}
```

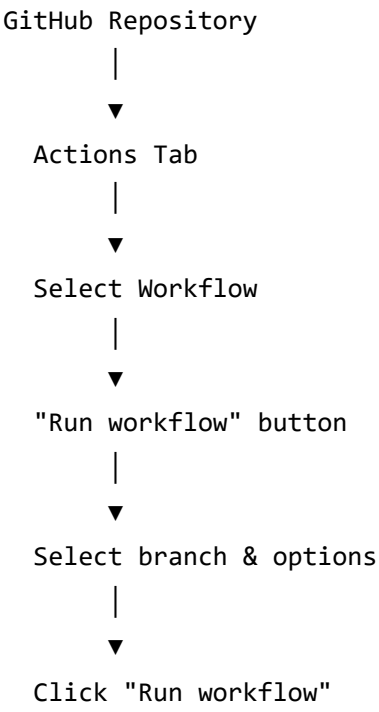
SECTION 8 — How Employees Use CI/CD Pipeline

8.1 How Each Employee Triggers Pipeline

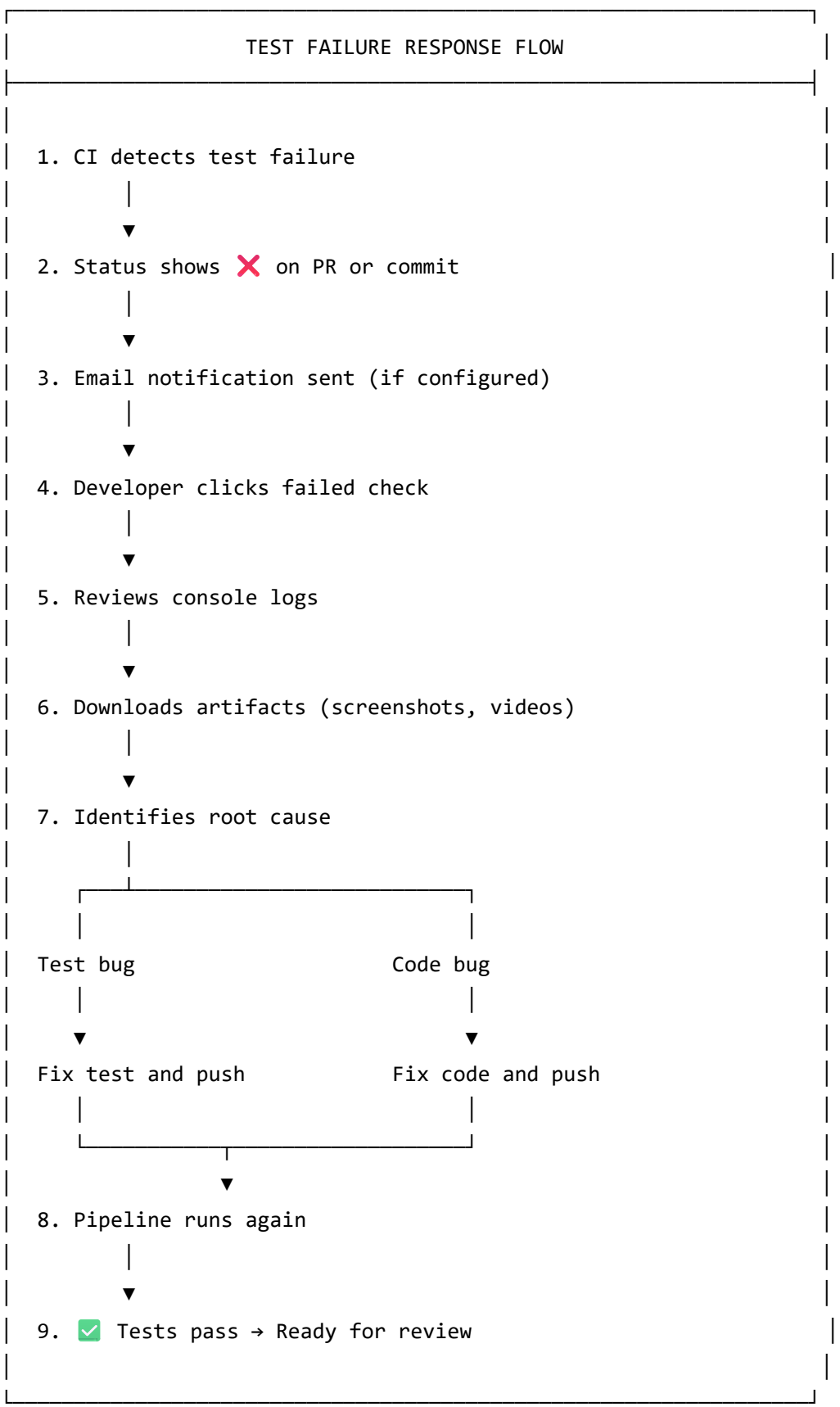
Automatic Triggers:

Action	Trigger	Pipeline Runs
Push to feature branch	`push`	✅ If configured
Create Pull Request	`pull_request`	✅ Yes
Push to develop	`push`	✅ Yes
Merge to main	`push`	✅ Yes

Manual Triggers:


GitHub Repository
Actions Tab
Select Workflow
"Run workflow" button
Select branch & options
Click "Run workflow"

8.2 What Happens When Tests Fail

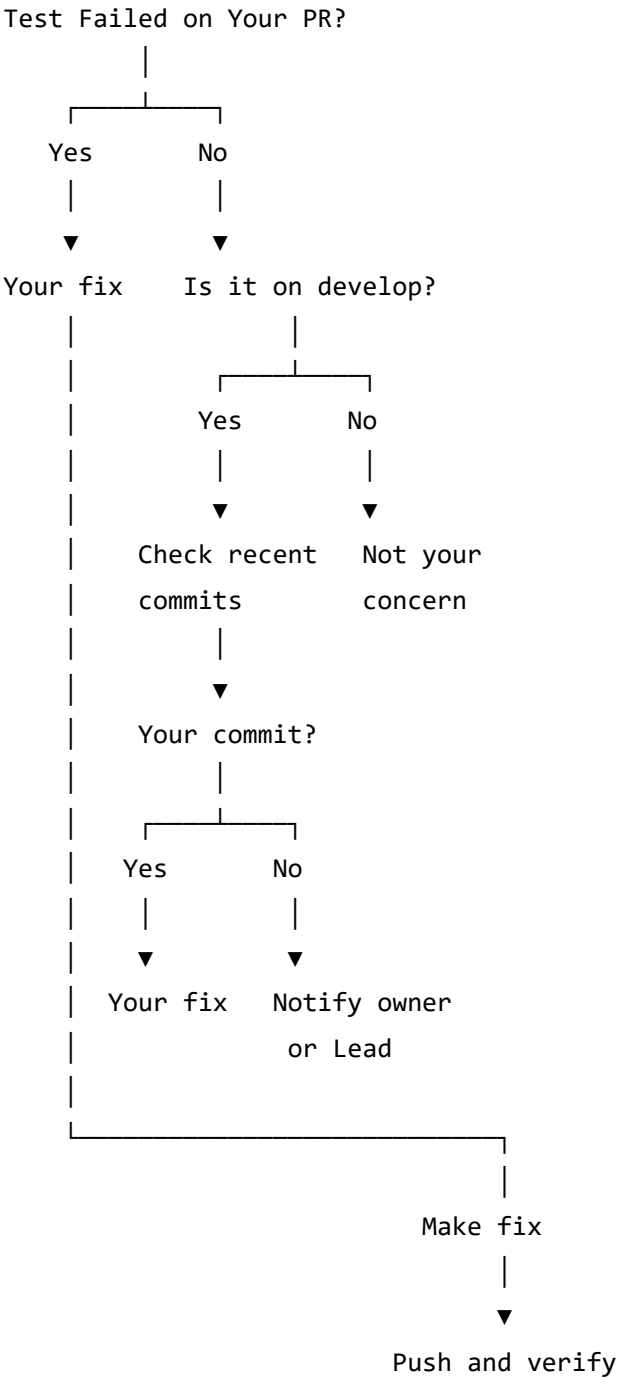


8.3 Who Fixes Failures - Ownership Model

Ownership Rules:

Failure Scenario	Owner	Responsibility
Your PR failed	You	Fix immediately
Develop broken after your merge	You	High priority fix
Develop broken by someone else	That person	You wait or help
Main broken	Automation Lead + Team	Emergency fix
Flaky test failing randomly	Senior Engineer	Investigation

Decision Tree:



8.4 How to Avoid Blocking Others

Best Practices:

Practice	Description
Run tests locally first	Catch failures before push
Fix failures immediately	Don't leave develop broken

Practice	Description
Small PRs	Easier to identify issues
Communicate	Let team know if you broke something
Monitor after merge	Stay online after merging

Anti-Patterns to Avoid:

❌ Don't	✅ Do Instead
Push and leave for the day	Stay to verify pipeline passes
Ignore red pipeline	Fix it or revert
Merge despite failing tests	Wait for green build
Blame others without checking	Investigate your changes first
Skip local testing	Always run `npx playwright test`

8.5 Responding to Pipeline Failures

Step-by-Step Response:

```
# 1. Check if it's your code
git log --oneline -5              # See recent commits

# 2. Run failing tests locally
npx playwright test tests/login.spec.ts --headed

# 3. If you broke it, fix it
# Make your changes...

# 4. Verify fix locally
npx playwright test

# 5. Push the fix
git add .
git commit -m "fix: resolve failing login test"
git push

# 6. Monitor pipeline
# Go to GitHub Actions and watch the run
```

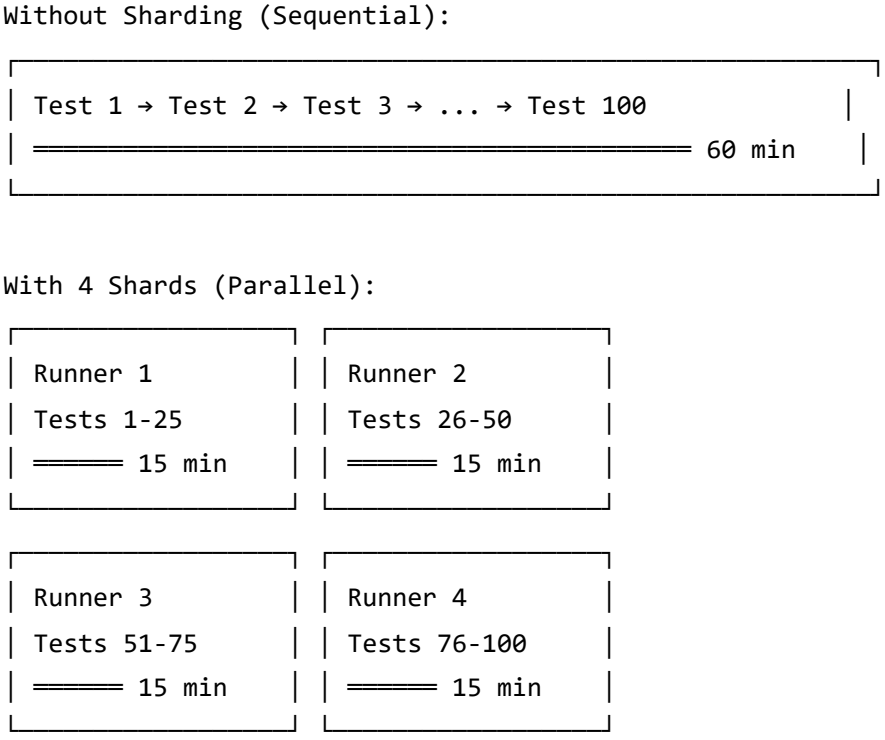
SECTION 9 — Managing CI/CD for Multiple Teams

9.1 Parallel Execution

Running Tests in Parallel:

```yaml
jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      fail-fast: false          # Don't cancel other jobs on failure
      matrix:
        shard: [1/4, 2/4, 3/4, 4/4]

    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 20
      - run: npm ci
      - run: npx playwright install --with-deps
      - run: npx playwright test --shard=${{ matrix.shard }}
```

Visual Explanation:



9.2 Module-Wise Test Execution

Folder Structure by Module:

```
tests/
├── login/              ← Team A
│   ├── login.spec.ts
│   └── login-data.spec.ts
├── registration/       ← Team B
│   ├── register.spec.ts
│   └── register-validation.spec.ts
├── payments/           ← Team C
│   └── checkout.spec.ts
└── reports/            ← Team D
    └── dashboard.spec.ts
```

Running Specific Modules:

```yaml
jobs:
  login-tests:
    runs-on: ubuntu-latest
    steps:
      - run: npx playwright test tests/login/

  payment-tests:
    runs-on: ubuntu-latest
    steps:
      - run: npx playwright test tests/payments/
```

Or Using Matrix:

```yaml
strategy:
  matrix:
    module: [login, registration, payments, reports]

steps:
  - run: npx playwright test tests/${{ matrix.module }}/
```

9.3 Tag-Based Execution

Adding Tags to Tests:

```
// Use annotations in test file
test('login with valid credentials @smoke @login', async ({ page }) => {
    // test code
});

test('complex payment flow @regression @payment', async ({ page }) => {
    // test code
});

test('critical checkout @critical @payment', async ({ page }) => {
    // test code
});
```

Running by Tags:

```
# Run only smoke tests
npx playwright test --grep @smoke

# Run login tests only
npx playwright test --grep @login

# Run everything except slow tests
npx playwright test --grep-invert @slow
```

Pipeline with Tag Selection:

```yaml
on:
  workflow_dispatch:
    inputs:
      test-tag:
        description: 'Test tag to run'
        required: true
        default: '@smoke'
        type: choice
        options:
          - '@smoke'
          - '@regression'
          - '@critical'

jobs:
  test:
    steps:
      - run: npx playwright test --grep ${{ inputs.test-tag }}
```

9.4 Environment Separation

Different Environments:

Environment	Purpose	URL
Development	Latest unstable code	dev.example.com
QA	QA testing	qa.example.com
Staging	Pre-production	staging.example.com
Production	Live site	www.example.com

Configuring Environments in Pipeline:

```yaml
on:
  workflow_dispatch:
    inputs:
      environment:
        description: 'Target environment'
        required: true
        type: choice
        options:
          - development
          - qa
          - staging

jobs:
  test:
    environment: ${{ inputs.environment }}
    env:
      BASE_URL: ${{ vars.BASE_URL }}          # Environment-specific
      TEST_USER: ${{ secrets.TEST_USER }}     # Environment-specific

    steps:
      - run: npx playwright test
```

Setting Up GitHub Environments:

1. Go to repository Settings
2. Click "Environments"
3. Create environment (e.g., "staging")
4. Add environment-specific secrets and variables
5. Optionally add protection rules (approvals)

9.5 Secrets Management

Types of Secrets:

Level	Scope	Example Use
Repository	This repo only	API keys for this project
Organization	All org repos	Shared service accounts
Environment	Specific env	Prod vs staging credentials

Setting Repository Secrets:

```
Repository → Settings → Secrets and variables → Actions → New repository secret
```

Secret Best Practices:

✅ Do	❌ Don't
Use secrets for passwords	Hardcode credentials in code
Use environment variables	Log secret values
Rotate secrets regularly	Share secrets in chat
Use least privilege	Give full access
Audit secret access	Ignore who has access

Accessing Secrets:

```yaml
env:
  DB_PASSWORD: ${{ secrets.DB_PASSWORD }}
  API_KEY: ${{ secrets.API_KEY }}
  TEST_USER: ${{ secrets.TEST_USER }}
```

9.6 Access Control

Team-Based Permissions:

Role	Repository Access	Actions Access
Read	View code	View workflow runs
Triage	Issues/PRs	View workflow runs
Write	Push to branches	Trigger workflows
Maintain	Manage settings	Manage secrets
Admin	Full control	Full control

Protected Branches:

```
Settings → Branches → Add branch protection rule

□ Require a pull request before merging
  □ Require approvals (1 or more)
□ Require status checks to pass
  □ Require branches to be up to date
  □ Status checks: "test" (your workflow job)
□ Do not allow bypassing above settings
```

This ensures:

- No direct pushes to main/develop
- Tests must pass before merge
- At least one reviewer approves

Continue to Section 10-13 and Bonus...

SECTIONS 10-13: Best Practices, Roles, Mistakes & Workflow Summary

SECTION 10 — Enterprise Best Practices

10.1 Code Quality Standards

Coding Standards Checklist:

Area	Standard	Example
Locators	Use data-testid or stable selectors	`[data-testid="login-btn"]`
Waits	Never use hardcoded waits	Use `waitFor()` instead
Assertions	Use specific assertions	`toHaveURL()` not just `toBeTruthy()`
Page Objects	One class per page	`LoginPage.ts`, `HomePage.ts`

Area	Standard	Example
Test Data	External files, not hardcoded	`test-data/loginData.ts`
Naming	Descriptive test names	`should show error for empty password`

Code Review Checklist:

```
□ No hardcoded values (URLs, credentials)
□ Proper error handling
□ Descriptive variable names
□ Page Object pattern followed
□ No unnecessary waits
□ Assertions are specific
□ Test covers stated scenario
□ No console.log statements
□ Follows naming conventions
□ Test tags applied (@smoke, @regression)
```

10.2 Mandatory PR Reviews

PR Requirements:

Requirement	Description
1+ Approval	At least one reviewer must approve
Tests Pass	All CI checks must be green
No Conflicts	Must be mergeable
Updated Branch	Must include latest develop changes

Review Focus Areas:

```
REVIEWER CHECKLIST:

□ Test Logic
  └─ Does test actually verify what it claims?

□ Locators
  └─ Are they stable and maintainable?

□ Test Data
  └─ Is sensitive data handled via secrets?

□ Error Handling
  └─ Will test fail clearly if issue occurs?

□ Readability
  └─ Can another engineer understand this?

□ Performance
  └─ No unnecessary waits or loops?
```

10.3 Linting

ESLint Configuration:

```
// .eslintrc.json
{
    "extends": [
        "eslint:recommended",
        "plugin:@typescript-eslint/recommended",
        "plugin:playwright/recommended"
    ],
    "rules": {
        "@typescript-eslint/no-unused-vars": "error",
        "playwright/no-wait-for-timeout": "error",
        "playwright/prefer-web-first-assertions": "error",
        "no-console": "warn"
    }
}
```

Run Linting in Pipeline:

```
- name: Lint code
  run: npm run lint

- name: Type check
  run: npx tsc --noEmit
```

10.4 Test Tagging

Tag Categories:

Tag	Purpose	Example
`@smoke`	Quick sanity checks	Login, homepage load
`@regression`	Full test suite	All test cases
`@critical`	Business-critical paths	Payment, checkout
`@slow`	Long-running tests	Reports, uploads
`@flaky`	Known unstable tests	Investigate later

Applying Tags:

```
test('verify login @smoke @login', async ({ page }) => {
    // Critical path test
});

test('complex workflow @regression @slow', async ({ page }) => {
    // Long running test
});
```

Running by Tag:

```
npx playwright test --grep @smoke        # Smoke tests only
npx playwright test --grep-invert @slow  # Exclude slow tests
```

10.5 Retry Strategy

Configuration Levels:

```
// playwright.config.ts
export default defineConfig({
    // Global retries
    retries: process.env.CI ? 2 : 0,

    // Project-specific
    projects: [
        {
            name: 'chromium',
            retries: 3,    // More retries for Chrome
        },
    ],
});
```

Test-Level Retry:

```
test('potentially flaky test', async ({ page }) => {
    test.info().config.retries = 5;   // Extra retries
    // test code
});
```

10.6 Flaky Test Handling

What is a Flaky Test?

A test that sometimes passes and sometimes fails without code changes.

Identification:

```
Test Results Over 10 Runs:
Run 1: ✅ Pass
Run 2: ✅ Pass
Run 3: ❌ Fail    ← Flaky!
Run 4: ✅ Pass
Run 5: ✅ Pass
```

Common Causes & Solutions:

Cause	Solution
Race conditions	Add proper waits (`waitFor`)

Cause	Solution
Network timing	Use `networkidle` state
Animation	Wait for animation completion
Test data pollution	Isolate test data
Parallel conflicts	Make tests independent

Handling Strategy:

```
FLAKY TEST WORKFLOW:

1. Identify
   └─ Monitor test results for inconsistency

2. Quarantine
   └─ Add @flaky tag
   └─ Exclude from critical pipelines

3. Investigate
   └─ Run repeatedly: npx playwright test --repeat-each=10
   └─ Add traces: trace: 'on'

4. Fix
   └─ Apply proper fixes
   └─ Remove @flaky tag

5. Monitor
   └─ Watch for recurrence
```

10.7 Documentation Standards

What to Document:

Item	Location	Content
Setup guide	README.md	How to start
Test guide	docs/TESTING.md	How to run tests
Framework guide	docs/FRAMEWORK.md	Architecture

Item	Location	Content
Change log	CHANGELOG.md	What changed

Test File Headers:

```
/**
 * Login Page Test Suite
 *
 * Tests the login functionality including:
 * - Valid credential login
 * - Invalid credential handling
 * - Empty field validation
 * - Remember me feature
 *
 * @module login
 * @tags @smoke @regression
 */
```

10.8 Security Practices

Security Checklist:

Practice	Implementation
No hardcoded secrets	Use GitHub Secrets
API keys	Store in secrets
Test data	Use fake data generators
Screenshots	Don't capture sensitive data
Logs	Mask sensitive information

Secure Secret Usage:

```
// BAD ❌
const password = 'myP@ssw0rd123';

// GOOD ✅
const password = process.env.TEST_PASSWORD;
```

10.9 Audit & Compliance

Audit Trail Maintenance:

```
MAINTAIN HISTORY OF:

□ All test runs (CI/CD keeps these)
□ Code changes (Git keeps these)
□ PR approvals (GitHub keeps these)
□ Deployment records
□ Access changes
```

Compliance Requirements:

Requirement	Implementation
Traceability	Link tests to requirements
Evidence	Store test reports
Access Control	Restrict sensitive access
Change Approval	Require PR reviews

SECTION 11 — Roles & Responsibilities

11.1 Junior Automation Engineer

Experience Level: 0-2 years

Day-to-Day Activities:

```
Morning:
├── Pull latest code
├── Check assigned tickets
└── Review any feedback on PRs

During Day:
├── Write new test cases
├── Fix assigned bugs
├── Run tests locally
└── Ask questions when stuck

End of Day:
├── Push completed work
├── Update ticket status
└── Note blockers
```

Key Responsibilities:

Responsibility	Priority
Write new tests	High
Fix failing tests	High
Follow standards	High
Learn framework	Medium
Document work	Medium

Skills to Develop:

- Playwright fundamentals
- TypeScript basics
- Git workflow
- Page Object Model
- Debugging skills

11.2 Senior Automation Engineer

Experience Level: 2-5 years

Responsibilities:

Area	Tasks
Technical	Design complex tests, handle edge cases
Quality	Review PRs, enforce standards
Mentoring	Guide junior engineers
Troubleshooting	Debug flaky/complex failures

Decision-Making Authority:

```
CAN DECIDE:
├── Test implementation approach
├── Which locators to use
├── Test data structure
└── Code review outcomes

SHOULD ESCALATE:
├── Framework architecture changes
├── New tool adoption
├── Major refactoring
└── Test coverage gaps
```

11.3 Automation Lead

Experience Level: 5+ years

Strategic Responsibilities:

Area	Focus
Planning	Test strategy, sprint planning
Coordination	Work with dev/QA teams
Reporting	Metrics, coverage reports
Technical	Architecture decisions
Team	Hiring, training, reviews

Metrics Owned:

```
AUTOMATION LEAD DASHBOARD:

Test Coverage:     ███████░ 80%
Pass Rate:         ████████ 92%
Execution Time:    35 minutes
Flaky Rate:        ██░░░░░░ 5%
Open Defects:      12
Sprint Velocity:   15 tests/sprint
```

11.4 DevOps / CI Owner

Focus Area: Infrastructure & Pipelines

Responsibilities:

Area	Tasks
CI/CD	Setup and maintain pipelines
Infrastructure	Manage runners, environments
Secrets	Handle credentials securely
Monitoring	Pipeline health, alerts
Optimization	Reduce execution time

Technical Ownership:

```
OWNS:
├── .github/workflows/
├── GitHub Actions configuration
├── Secret management
├── Environment setup
├── Runner management
└── Pipeline optimization
```

11.5 QA Manager

Strategic Focus:

Area	Responsibility
Strategy	Overall quality approach
Resources	Team allocation, hiring
Budget	Tools, infrastructure costs
Stakeholders	Progress reporting
Process	Methodology improvements

Reporting Expectations:

```
MONTHLY REPORT TEMPLATE:

Executive Summary
├── Overall test health
├── Coverage improvements
├── Key risks

Metrics
├── Test execution trends
├── Defect trends
├── Automation ROI

Roadmap
├── Next month priorities
├── Resource needs
└── Risk mitigation
```

SECTION 12 — Common Enterprise Mistakes & Solutions

12.1 Merge Conflicts

Common Scenarios:

Scenario	Cause	Prevention
Same file edited	Two people worked on same file	Communicate, smaller files
Outdated branch	Didn't pull before pushing	Always pull before work
Long-lived branches	Branch existed too long	Merge frequently

Resolution Steps:

```
# 1. Pull latest develop
git checkout develop
git pull origin develop

# 2. Go back to your branch
git checkout feature/your-branch

# 3. Merge develop into your branch
git merge develop

# 4. If conflicts appear, open conflicted files
# 5. Resolve conflicts manually (remove markers)
# 6. Stage resolved files
git add .

# 7. Complete the merge
git commit -m "fix: resolve merge conflicts with develop"

# 8. Push
git push origin feature/your-branch
```

12.2 Broken Pipelines

Troubleshooting Table:

Error	Likely Cause	Solution
"npm ci failed"	Missing package-lock.json	Commit package-lock.json
"Browser not found"	Missing install step	Add `playwright install --with-deps`
"Timeout exceeded"	Tests too slow or stuck	Increase timeout, fix stuck test

Error	Likely Cause	Solution
"Out of memory"	Too many parallel workers	Reduce worker count
"Permission denied"	File permission issue	Check file permissions
"Secret not found"	Secret not configured	Add secret in settings

Debugging Pipeline:

```
# Add debug output
- name: Debug info
  run: |
    echo "Node version: $(node --version)"
    echo "NPM version: $(npm --version)"
    echo "Current directory: $(pwd)"
    ls -la
```

12.3 Unstable Tests

Stability Checklist:

Issue	Check	Fix
Timing issues	Are you using hardcoded waits?	Use `waitFor()`
Selector problems	Is selector specific enough?	Use data-testid
Test isolation	Does test depend on others?	Make independent
Data issues	Is data shared between tests?	Isolate test data
Race conditions	Are you waiting for async ops?	Add proper waits

Making Tests Stable:

```
// BAD ❌ - Hardcoded wait
await page.waitForTimeout(3000);

// GOOD ✅ - Wait for element
await page.locator('#result').waitFor({ state: 'visible' });

// BAD ❌ - Flaky selector
await page.locator('.btn').click();

// GOOD ✅ - Stable selector
await page.locator('[data-testid="submit-btn"]').click();
```

12.4 Hardcoded Data

Problem:

```
// BAD ❌
const username = 'testuser123';
const password = 'P@ssw0rd!';
const apiUrl = 'https://api.prod.example.com';
```

Solution:

```
// GOOD ✅ - Use environment variables
const username = process.env.TEST_USERNAME;
const password = process.env.TEST_PASSWORD;
const apiUrl = process.env.API_URL;

// GOOD ✅ - Use test data files
import { loginData } from '../test-data/loginData';
await loginPage.login(loginData.username, loginData.password);
```

12.5 Poor Branching

Anti-Patterns:

❌ Bad Practice	✅ Best Practice
Commit to main directly	Use feature branches

❌ Bad Practice	✅ Best Practice
Never merge develop	Merge frequently
Huge feature branches	Small, focused branches
Vague branch names	Descriptive names
Never delete branches	Clean up after merge

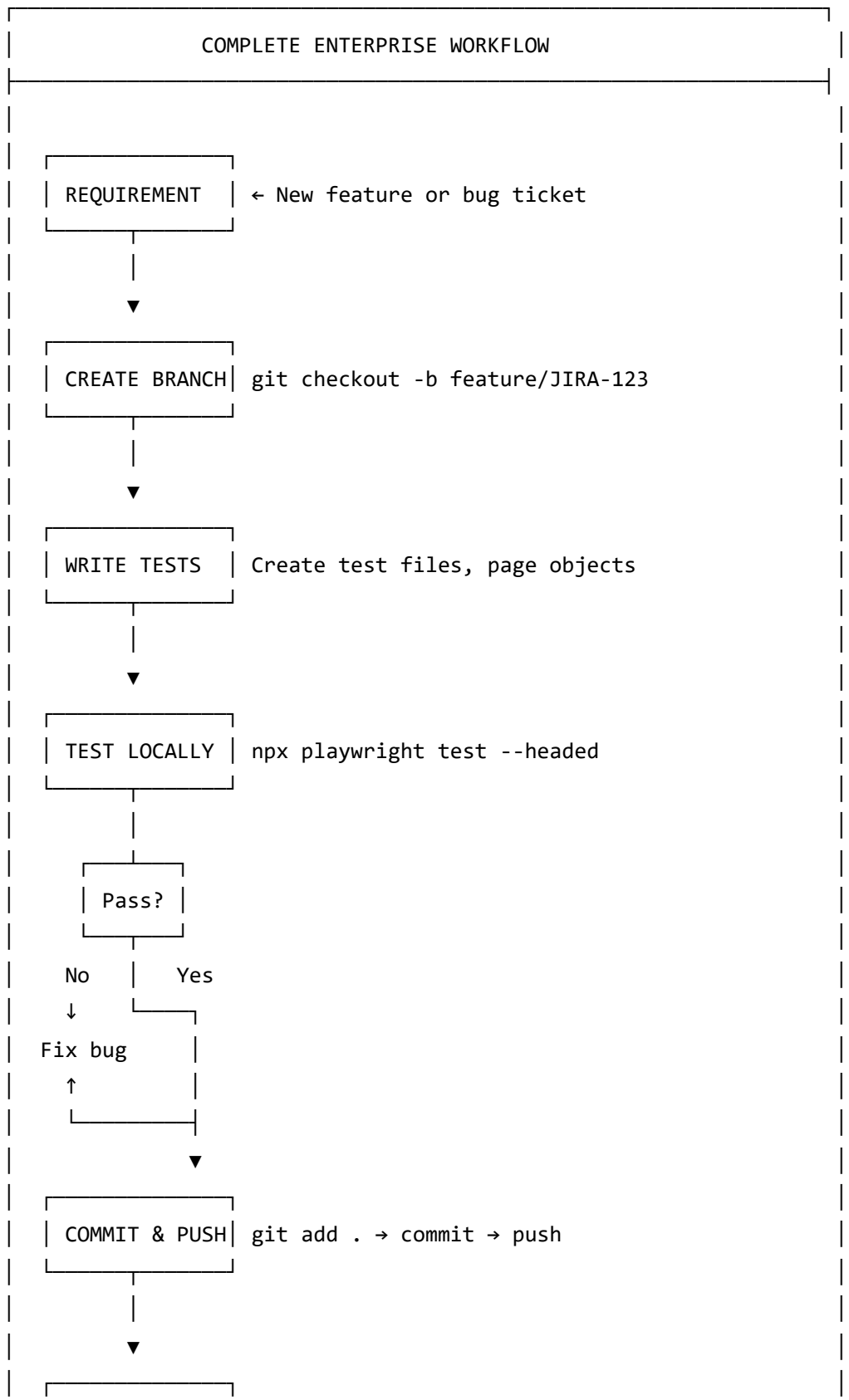
Branch Hygiene:

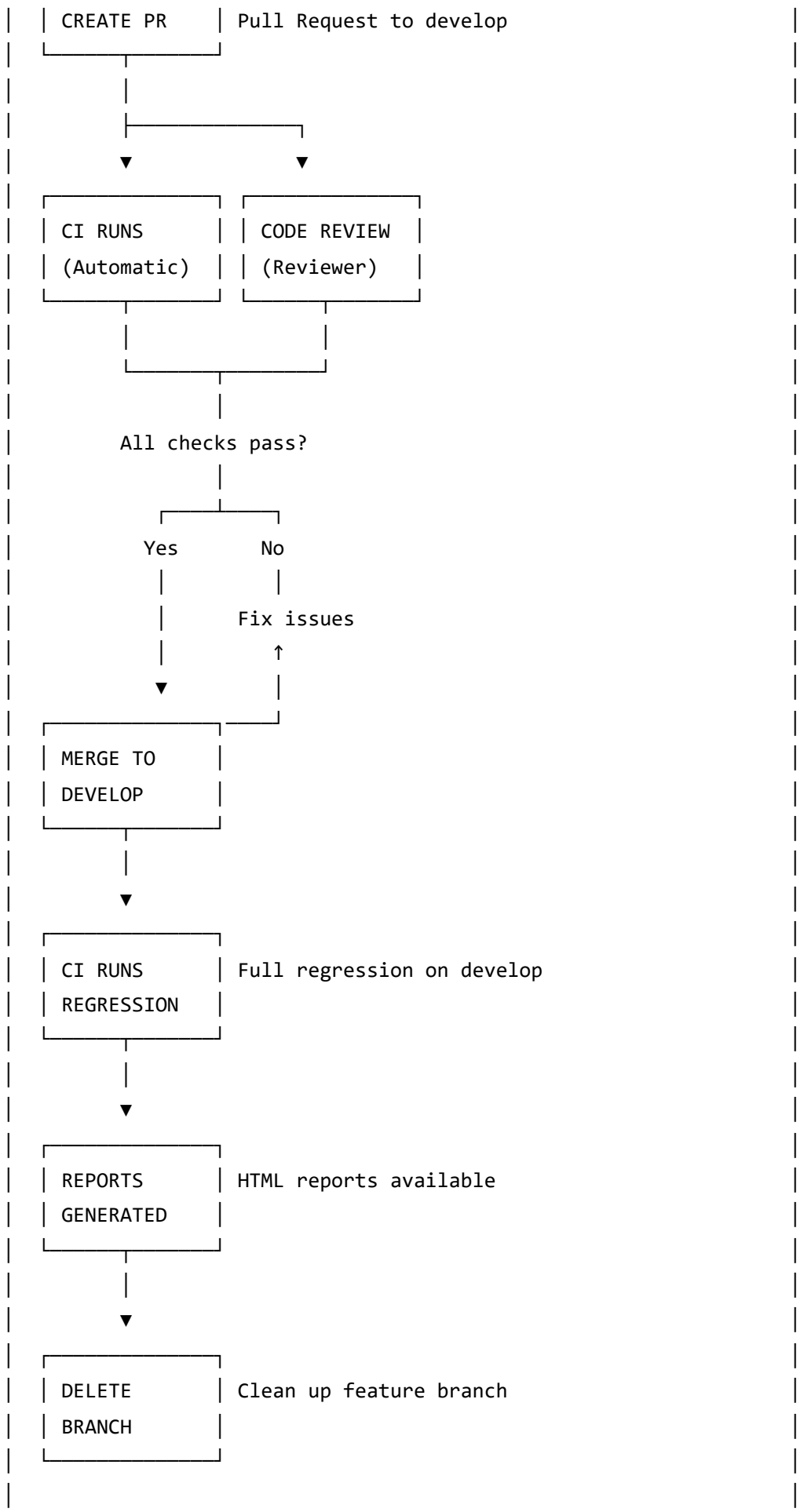
```
# List all branches
git branch -a

# Delete merged local branches
git branch --merged | grep -v "main\|develop" | xargs git branch -d

# Delete remote merged branch
git push origin --delete feature/old-branch
```

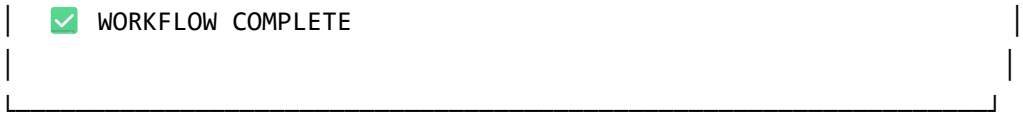
SECTION 13 — Real-World Enterprise Workflow

Summary

13.1 End-to-End Flow




CREATE PR
Pull Request to develop
CI RUNS
(Automatic)
CODE REVIEW
(Reviewer)
All checks pass?
Yes
No
Fix issues
MERGE TO
DEVELOP
CI RUNS
REGRESSION
Full regression on develop
REPORTS
GENERATED
HTML reports available
DELETE
BRANCH
Clean up feature branch


✅ WORKFLOW COMPLETE

13.2 Quick Reference Commands

```
# ═══════════════════════════════════════════════════════════════
# DAILY WORKFLOW COMMANDS
# ═══════════════════════════════════════════════════════════════

# Start of day
git checkout develop
git pull origin develop

# Create feature branch
git checkout -b feature/JIRA-123-description

# Check status
git status

# Stage changes
git add .

# Commit with message
git commit -m "feat: add login test cases"

# Push branch
git push origin feature/JIRA-123-description

# ═══════════════════════════════════════════════════════════════
# TEST COMMANDS
# ═══════════════════════════════════════════════════════════════

# Run all tests
npx playwright test

# Run headed (see browser)
npx playwright test --headed

# Run specific file
npx playwright test tests/login.spec.ts

# Run by tag
npx playwright test --grep @smoke

# Debug mode
npx playwright test --debug
```

```
# Show report
npx playwright show-report

# ═══════════════════════════════════════════════════════════════
# AFTER MERGE
# ═══════════════════════════════════════════════════════════════

# Update local develop
git checkout develop
git pull origin develop

# Delete local branch
git branch -d feature/JIRA-123-description
```

13.3 Summary Checklist

Before Starting Work:

- ☐ Pull latest develop
- ☐ Create feature branch with proper naming
- ☐ Understand the requirement

While Working:

- ☐ Follow coding standards
- ☐ Write descriptive test names
- ☐ Use Page Object pattern
- ☐ No hardcoded values
- ☐ Add appropriate test tags

Before Pushing:

- ☐ Run tests locally
- ☐ All tests pass
- ☐ Code is clean (no console.log)
- ☐ Commit message follows standards

Pull Request:

- ☐ PR has descriptive title
- ☐ PR description explains changes

- [ ] Linked to ticket/requirement
- [ ] Assigned reviewers

After Merge:

- [ ] CI pipeline passes on develop
- [ ] Delete feature branch
- [ ] Update ticket status
- [ ] Notify team if needed

Continue to Bonus Section...

BONUS SECTION: Interview Tips, Troubleshooting & Enterprise Examples

BONUS 1 — Interview Perspective

Key Topics Interviewers Ask

Git & GitHub Questions:

Question	Key Points to Mention
Explain Git workflow	Clone → Branch → Code → Commit → Push → PR → Merge
Difference between merge and rebase	Merge preserves history, rebase creates linear history
How to resolve conflicts	Pull latest, open files, resolve markers, commit
What is a Pull Request	Code review mechanism, CI runs, approval required
Why use branching	Isolation, parallel work, safe experimentation

CI/CD Questions:

Question	Key Points to Mention
What is CI/CD	Continuous Integration & Delivery, automated pipelines
Benefits of CI/CD	Early bug detection, consistent testing, faster releases
Explain GitHub Actions	YAML-based workflows, triggers, jobs, steps, runners
How to handle secrets	GitHub Secrets, environment variables, never hardcode
What happens when tests fail in CI	Notification sent, PR blocked, fix required

Framework Questions:

Question	Key Points to Mention
Explain Page Object Model	Separate locators from tests, maintainability, reusability
How to handle flaky tests	Retries, proper waits, stable locators, investigation
Data-driven testing	External data files, parameterization, multiple scenarios
Parallel execution	Workers, sharding, independent tests
Reporting	HTML reporter, artifacts, CI integration

Do's and Don'ts in Interviews

Technical Demonstration:

✅ Do	❌ Don't
Explain your thought process	Just give one-word answers
Mention real-world experience	Only speak theoretically
Acknowledge what you don't know	Pretend to know everything
Use proper terminology	Use vague language
Give specific examples	Be too generic

When Asked About Your Framework:

```
GOOD ANSWER STRUCTURE:

1. Start with architecture overview
   "Our framework uses Page Object Model with..."

2. Explain key components
   "We have separate folders for pages, tests, and utilities..."

3. Mention CI/CD integration
   "Tests run automatically on every PR using GitHub Actions..."

4. Highlight best practices
   "We enforce code reviews, use data-driven testing..."

5. Share a challenge you solved
   "We had flaky tests, so I implemented..."
```

BONUS 2 — Troubleshooting Tables

Git Troubleshooting

Error Message	Cause	Solution
`fatal: not a git repository`	Not in git folder	`cd` to correct folder or `git init`
`error: failed to push`	Remote has new commits	`git pull` first, then push
`CONFLICT in file.ts`	Same lines changed	Resolve conflicts manually
`Your branch is behind`	Need to pull updates	`git pull origin develop`
`Permission denied`	Auth issue	Check SSH keys or use HTTPS
`detached HEAD`	Not on a branch	`git checkout branch-name`

Pipeline Troubleshooting

Error	Cause	Solution
`npm ci failed`	Missing lock file	Commit `package-lock.json`

Error	Cause	Solution
`playwright install failed`	Missing deps	Use `--with-deps` flag
`Timeout 60000ms exceeded`	Test too slow	Increase timeout or fix test
`Browser closed unexpectedly`	Memory issue	Reduce parallelism
`Secret not found`	Not configured	Add in repository settings
`Artifact upload failed`	Path doesn't exist	Check directory path

Test Troubleshooting

Issue	Cause	Solution
Element not found	Locator changed	Update locator
Click intercepted	Element covered	Scroll to element first
Test timeout	Slow page/network	Increase timeout, check network
Random failures	Race condition	Add proper waits
Works locally, fails in CI	Environment diff	Check headless mode, viewport

BONUS 3 — Do's and Don'ts

Code Writing

✅ Do	❌ Don't
Use data-testid selectors	Use fragile XPath
Externalize test data	Hardcode values
Write independent tests	Create test dependencies
Use page objects	Write everything in test file
Add meaningful assertions	Skip verification

✅ Do	❌ Don't
Handle expected failures	Let tests fail silently

Git Usage

✅ Do	❌ Don't
Pull before pushing	Force push to shared branches
Write clear commit messages	Use vague messages like "fix"
Create focused branches	Put multiple features in one branch
Review PR feedback promptly	Ignore review comments
Delete merged branches	Leave stale branches

CI/CD

✅ Do	❌ Don't
Fix broken builds immediately	Leave pipeline red
Use secrets for credentials	Hardcode passwords
Monitor test results	Ignore CI output
Keep pipelines fast	Add unnecessary steps
Upload reports as artifacts	Lose test evidence

Team Collaboration

✅ Do	❌ Don't
Communicate blockers early	Wait until deadline
Help teammates debug	Blame without helping
Share knowledge	Keep solutions to yourself
Document your work	Leave others guessing

✅ Do	❌ Don't
Follow agreed standards	Create your own rules

BONUS 4 — Real Enterprise Examples

Example 1: E-Commerce Test Suite Structure

```
ecommerce-tests/
├── .github/
│   └── workflows/
│       ├── smoke.yml           # Quick tests on every PR
│       ├── regression.yml      # Full suite nightly
│       └── production.yml      # Production monitoring
│
├── tests/
│   ├── cart/
│   │   ├── add-to-cart.spec.ts
│   │   ├── cart-quantity.spec.ts
│   │   └── cart-removal.spec.ts
│   ├── checkout/
│   │   ├── guest-checkout.spec.ts
│   │   ├── member-checkout.spec.ts
│   │   └── payment-methods.spec.ts
│   └── search/
│       ├── product-search.spec.ts
│       └── filter-products.spec.ts
│
├── pages/
│   ├── CartPage.ts
│   ├── CheckoutPage.ts
│   ├── ProductPage.ts
│   └── SearchPage.ts
│
└── test-data/
    ├── products.json
    ├── users.json
    └── addresses.json
```

Example 2: Multi-Environment Pipeline

```yaml
# .github/workflows/multi-env.yml
name: Multi-Environment Tests

on:
  workflow_dispatch:
    inputs:
      environment:
        description: 'Environment'
        required: true
        type: choice
        options:
          - dev
          - qa
          - staging
          - prod

jobs:
  test:
    runs-on: ubuntu-latest
    environment: ${{ inputs.environment }}

    steps:
      - uses: actions/checkout@v4

      - uses: actions/setup-node@v4
        with:
          node-version: 20

      - run: npm ci
      - run: npx playwright install --with-deps

      - name: Run tests against ${{ inputs.environment }}
        run: npx playwright test
        env:
          BASE_URL: ${{ vars.BASE_URL }}
          API_URL: ${{ vars.API_URL }}
          TEST_USER: ${{ secrets.TEST_USER }}
          TEST_PASS: ${{ secrets.TEST_PASS }}

      - uses: actions/upload-artifact@v4
        if: always()
```

```
      with:
        name: ${{ inputs.environment }}-report
        path: playwright-report/
```

Example 3: Team Workflow Scenario

Scenario: Adding new checkout tests

```
DAY 1 - Monday

Engineer A (you):
├── 09:00 - Pull latest code
│   git checkout develop
│   git pull origin develop
│
├── 09:15 - Create feature branch
│   git checkout -b feature/JIRA-456-checkout-tests
│
├── 09:30 - Start writing tests
│   Created CheckoutPage.ts
│   Created checkout.spec.ts
│
├── 17:00 - Run tests locally
│   npx playwright test tests/checkout/ --headed
│   3 tests pass, 1 fails
│
├── 17:30 - Fix failing test
│   Updated locator for payment button
│
└── 18:00 - Push work
    git add .
    git commit -m "feat: add checkout page tests (WIP)"
    git push origin feature/JIRA-456-checkout-tests

DAY 2 - Tuesday

├── 09:00 - Pull latest develop
│   git checkout develop
│   git pull origin develop
│   git checkout feature/JIRA-456-checkout-tests
│   git merge develop
│
├── 09:30 - Complete remaining tests
│   Added 3 more test cases
│
├── 15:00 - Final local run
│   npx playwright test tests/checkout/
│   All 7 tests pass ✅
│
├── 15:30 - Push and create PR
│   git push origin feature/JIRA-456-checkout-tests
```

```
│   Created PR on GitHub
│
├── 16:00 - CI runs
│   Pipeline started
│   Tests running...
│   ✅ All checks pass
│
└── 16:30 - Request review
    Assigned Senior Engineer as reviewer

DAY 3 - Wednesday

├── 10:00 - Reviewer leaves comments
│   "Please add test for error handling"
│   "Use data-testid instead of class"
│
├── 11:00 - Address review comments
│   Added error handling test
│   Updated locators
│
├── 11:30 - Push updates
│   git add .
│   git commit -m "fix: address review comments"
│   git push origin feature/JIRA-456-checkout-tests
│
├── 14:00 - Review approved
│   Reviewer approved PR ✅
│
├── 14:15 - Merge to develop
│   Clicked "Squash and merge"
│   All 7 tests merged to develop
│
└── 14:30 - Cleanup
    git checkout develop
    git pull origin develop
    git branch -d feature/JIRA-456-checkout-tests

RESULT:
├── 7 new checkout tests added
├── CI pipeline passing
├── Code reviewed and approved
└── Branch cleaned up
```

Quick Reference Card

```
┌─────────────────────────────────────────────────────────────┐
│              PLAYWRIGHT ENTERPRISE QUICK REFERENCE          │
├─────────────────────────────────────────────────────────────┤
│                                                             │
│  ▶ DAILY WORKFLOW                                           │
│    git pull → branch → code → test → commit → push → PR     │
│                                                             │
│  ▶ ESSENTIAL COMMANDS                                       │
│    npx playwright test           # Run all tests            │
│    npx playwright test --headed  # See browser              │
│    npx playwright test --debug   # Debug mode               │
│    npx playwright show-report    # View report              │
│                                                             │
│  ▶ GIT COMMANDS                                             │
│    git checkout -b feature/name  # Create branch            │
│    git add . && git commit -m "" # Stage and commit         │
│    git push origin branch        # Push to GitHub           │
│                                                             │
│  ▶ TEST TAGS                                                │
│    @smoke      - Quick sanity tests                         │
│    @regression - Full test suite                            │
│    @critical   - Must-pass tests                            │
│                                                             │
│  ▶ COMMIT PREFIXES                                          │
│    feat:     - New feature                                  │
│    fix:      - Bug fix                                      │
│    refactor: - Code improvement                             │
│    docs:     - Documentation                                │
│                                                             │
│  ▶ BRANCH NAMING                                            │
│    feature/JIRA-123-description                             │
│    bugfix/JIRA-456-fix-name                                 │
│    hotfix/critical-issue                                    │
│                                                             │
│  ▶ PIPELINE STATUS                                          │
│    ✅ Green = Ready to merge                                │
│    ❌ Red   = Fix before merging                            │
│    🟡 Yellow = Running                                      │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

Document Information

Item	Details
Created	December 31, 2024
Purpose	Enterprise onboarding & training
Audience	Automation engineers (all levels)
Framework	Playwright + TypeScript
Scope	Complete workflow from basics to enterprise practices

End of Enterprise Automation Handbook