Branching-Strategien im Vergleich: Git Flow, GitHub Flow und Trunk-Based Development

Die Wahl der richtigen Branching-Strategie ist eine der wichtigsten architektonischen Entscheidungen für dein Projekt. Sie beeinflusst, wie dein Team zusammenarbeitet, wie schnell ihr Features ausliefern könnt und wie stabil eure Releases sind. Lass mich dir die drei populärsten Strategien im Detail vorstellen.

□□ Git Flow – der strukturierte Klassiker

Git Flow wurde 2010 von Vincent Driessen vorgestellt und war lange Zeit *der* Standard für professionelle Softwareentwicklung. Es ist eine ausgeklügelte, aber komplexe Strategie mit klar definierten Branch-Typen und strengen Regeln.

Die Branch-Struktur

```
gitGraph
    commit id: "Initial"
    branch develop
    checkout develop
    commit id: "Setup"
    branch feature/login
    checkout feature/login
    commit id: "Login UI"
```

```
    commit id: "Login Logic"
    checkout develop
    merge feature/login id: "Merge Login"
    branch release/1.0
    checkout release/1.0
    commit id: "Bugfix"
    commit id: "Version bump"
    checkout main
    merge release/1.0 id: "Release 1.0" tag: "v1.0"
    checkout develop
    merge release/1.0 id: "Back-merge"
```

Git Flow definiert **fünf Branch-Typen** mit jeweils spezifischen Aufgaben:

1. **`main`** (früher `master`)
   Der heilige Gral deines Repositories. Dieser Branch enthält *ausschließlich* produktionsreifen Code. Jeder Commit auf `main` repräsentiert eine Release-Version und wird typischerweise mit einem Tag versehen (z.B. `v1.0.0`, `v1.1.0`). Direktes Committen auf `main` ist streng verboten – Änderungen kommen nur durch Merges von Release- oder Hotfix-Branches.
2. **`develop`**
   Der Integrations-Branch für alle neuen Entwicklungen. Hier fließen alle abgeschlossenen Features zusammen. `develop` repräsentiert den aktuellen Entwicklungsstand und ist die Basis für neue Feature-Branches. Man könnte sagen: `main` ist „was der Kunde sieht", `develop` ist „was als nächstes kommt".
3. **`feature/*`** (z.B. `feature/user-authentication`, `feature/shopping-cart`)
   Für jedes neue Feature wird ein eigener Branch von `develop` abgezweigt. Hier findet die eigentliche Entwicklungsarbeit statt. Feature-Branches können beliebig viele Commits enthalten und existieren so lange, bis das Feature fertig ist. Nach Abschluss wird der Branch zurück in `develop` gemergt und gelöscht.
4. **`release/*`** (z.B. `release/1.2.0`)
   Wenn `develop` genügend Features für eine neue Version angesammelt hat, wird ein Release-Branch abgezweigt. Ab diesem Zeitpunkt kommen *keine neuen Features* mehr hinzu – nur noch Bugfixes, Dokumentation und Release-Vorbereitungen (Versionsnummern aktualisieren, Changelog schreiben). Nach Fertigstellung wird der Release-Branch sowohl in `main` als auch zurück in `develop` gemergt.
5. **`hotfix/*`** (z.B. `hotfix/1.0.1-security-patch`)
   Für kritische Bugs in der Produktion, die nicht auf den nächsten regulären Release warten können. Hotfix-Branches werden direkt von `main` abgezweigt, enthalten nur die minimale Änderung zur Fehlerbehebung und werden nach Fertigstellung sowohl in `main` (mit neuem Tag) als auch in `develop` gemergt.

Der typische Workflow in Git Flow

Ein neues Feature durchläuft folgenden Weg:

```
develop → feature/xyz → develop → release/1.0 → main + develop
                                        ↑
                                   (Bugfixes)
```

Und ein Hotfix:

```
main → hotfix/1.0.1 → main + develop
```

Vor- und Nachteile von Git Flow

✅ Vorteile	❌ Nachteile
Klare Trennung zwischen Entwicklung und Produktion	Hohe Komplexität durch viele Branch-Typen
Parallele Arbeit an Release und neuen Features möglich	Lange Feature-Branches führen zu schwierigen Merges
Gut geeignet für versionierte Software mit Releases	Langsamerer Entwicklungszyklus
Stabile `main`-Branch garantiert	Erfordert Disziplin und Tooling
Hotfixes können unabhängig eingespielt werden	Overkill für kleine Teams oder Web-Apps

Wann Git Flow verwenden?

Git Flow eignet sich besonders für:

- **Versionierte Software** mit klaren Release-Zyklen (z.B. Desktop-Apps, Mobile Apps, Libraries)
- **Projekte mit Support für mehrere Versionen** gleichzeitig
- **Größere Teams** (5+ Entwickler) mit spezialisierten Rollen
- **Regulierte Umgebungen**, die Nachvollziehbarkeit und Audits erfordern
- **Software mit längeren Release-Zyklen** (Wochen bis Monate)

□□ GitHub Flow – schlank und kontinuierlich

GitHub Flow wurde von GitHub selbst als Reaktion auf die Komplexität von Git Flow entwickelt. Es ist radikal einfacher: Es gibt nur `main` und Feature-Branches. Die Philosophie ist „ship early, ship

often" – kontinuierliche Auslieferung kleiner Änderungen.

Die Branch-Struktur

```
gitGraph
    commit id: "Initial"
    branch feature/login
    checkout feature/login
    commit id: "Login UI"
    commit id: "Login Tests"
    checkout main
    merge feature/login id: "PR #1" tag: "deployed"
    branch feature/dashboard
    checkout feature/dashboard
    commit id: "Dashboard"
    checkout main
    merge feature/dashboard id: "PR #2" tag: "deployed"
    branch bugfix/header
    checkout bugfix/header
    commit id: "Fix Header"
    checkout main
    merge bugfix/header id: "PR #3" tag: "deployed"
```

Die sechs Regeln von GitHub Flow

1. **`main` ist immer deploybar**
   Der `main`-Branch muss zu *jedem Zeitpunkt* in Produktion deployt werden können. Das bedeutet: Nur getesteter, funktionierender Code kommt hinein.
2. **Für jede Änderung einen Branch erstellen**
   Egal ob Feature, Bugfix oder Experiment – erstelle einen beschreibend benannten Branch von `main` (z.B. `add-user-profile`, `fix-login-timeout`, `experiment/new-checkout`).
3. **Regelmäßig committen und pushen**
   Kleine, häufige Commits mit klaren Nachrichten. Push regelmäßig zu GitHub, damit andere den Fortschritt sehen und du ein Backup hast.
4. **Pull Request erstellen, wenn bereit für Review**
   Sobald deine Änderung bereit für Feedback ist (oder du Diskussion brauchst), öffne einen Pull Request. Dies ist der zentrale Ort für Code Review und Diskussion.
5. **Nach Review und Approval: Merge in `main`**
   Sobald der PR approved ist und alle Tests grün sind, wird er in `main` gemergt. Bei GitHub Flow gibt es keine Zwischenstationen.

6. **Sofort nach dem Merge deployen**
   Der Merge in `main` triggert (idealerweise automatisch) ein Deployment. Das bedeutet, dass jeder Merge innerhalb von Minuten in Produktion ist.

Der typische Workflow in GitHub Flow

```
main → feature-branch → Pull Request → Code Review → main → Deploy
           ↑                                  ↓
      (entwickeln)                (automatische Tests)
```

Vor- und Nachteile von GitHub Flow

✅ Vorteile	❌ Nachteile
Extrem einfach zu verstehen und umzusetzen	Keine native Unterstützung für versionierte Releases
Fördert kontinuierliche Integration	Erfordert sehr gute Testabdeckung und CI/CD
Schnelle Iteration und Feedback-Zyklen	Hotfixes sind „normal" – keine spezielle Behandlung
Pull Requests als zentrale Review-Plattform	Weniger geeignet, wenn mehrere Versionen unterstützt werden müssen
Ideal für Web-Applikationen und SaaS	Kann bei großen Features unübersichtlich werden

Wann GitHub Flow verwenden?

GitHub Flow eignet sich besonders für:

- **Web-Applikationen und SaaS**, die kontinuierlich deployt werden
- **Kleine bis mittlere Teams** (1–10 Entwickler)
- **Projekte mit guter CI/CD-Pipeline** und automatisierten Tests
- **Agile Teams** mit kurzen Sprints und häufigen Releases
- **Startups und schnell iterierende Produkte**

□□ Trunk-Based Development – der radikale Ansatz

Trunk-Based Development (TBD) geht noch einen Schritt weiter als GitHub Flow. Die Idee: Alle Entwickler committen direkt in einen einzigen Branch (den „Trunk", typischerweise `main`). Feature-Branches existieren entweder gar nicht oder sind extrem kurzlebig (maximal 1–2 Tage).

Die Branch-Struktur

```
gitGraph
    commit id: "Feature A - Part 1"
    commit id: "Feature A - Part 2"
    commit id: "Bugfix"
    commit id: "Feature B" tag: "deployed"
    commit id: "Refactoring"
    commit id: "Feature A - Part 3"
    commit id: "Feature A complete" tag: "deployed"
```

Die Kernprinzipien

1. **Ein einziger Branch für alle**
   Es gibt nur `main` (den „Trunk"). Alle Entwickler integrieren ihre Änderungen hier. Lange lebende Feature-Branches sind *verboten*.
2. **Sehr kleine, sehr häufige Commits**
   Statt eines großen Commits am Ende eines Features gibt es viele kleine Commits während der Entwicklung. Jeder Commit muss den Build grün halten.
3. **Feature Flags für unfertige Features**
   Wie bringt man Code für ein halbfertiges Feature in `main`, ohne dass Nutzer es sehen? Durch **Feature Flags** (auch Feature Toggles genannt):

   ```
   if ($featureFlags->isEnabled('new-checkout-process')) {
       return $this->newCheckoutProcess($cart);
   } else {
       return $this->legacyCheckout($cart);
   }
   ```

   Das Feature wird erst „eingeschaltet", wenn es fertig ist – obwohl der Code schon lange in Produktion ist.
4. **Kurzlebige Feature-Branches (optional)**
   Manche Teams erlauben Feature-Branches, aber mit strikter Regel: Sie dürfen maximal 1–2 Tage existieren und müssen dann gemergt werden. Das erzwingt kleine, inkrementelle Änderungen.
5. **Exzellente CI/CD ist Pflicht**

> Da jeder Commit potenziell in Produktion geht, muss die Test-Pipeline *schnell* und *zuverlässig* sein. Typischerweise < 10 Minuten für den kompletten Build.

Branch by Abstraction

Für größere Refactorings, die nicht in 1–2 Tagen erledigt sind, nutzt TBD das Pattern „**Branch by Abstraction**":

1. Erstelle eine Abstraktionsschicht (Interface) vor dem zu ändernden Code
2. Implementiere die neue Version hinter dem Interface
3. Schalte schrittweise um (Feature Flag oder schrittweise Migration)
4. Entferne die alte Implementierung, sobald die neue stabil ist

So bleibt der Code immer lauffähig, obwohl du parallel zwei Implementierungen hast.

Vor- und Nachteile von Trunk-Based Development

✅ Vorteile	❌ Nachteile
Keine Merge-Hölle durch lang lebende Branches	Erfordert erfahrene, disziplinierte Entwickler
Maximale Kontinuierliche Integration	Feature Flags erhöhen Code-Komplexität
Sehr schnelles Feedback	Sehr gute Testabdeckung ist Voraussetzung
Fördert kleine, fokussierte Änderungen	Kann für Junior-Entwickler überfordernd sein
Von Google, Facebook und anderen Tech-Giganten verwendet	Weniger Review-Möglichkeiten vor dem Merge

Wann Trunk-Based Development verwenden?

Trunk-Based Development eignet sich besonders für:

- **Erfahrene, hochperformante Teams** mit starker Ingenieurskultur
- **Unternehmen mit exzellenter CI/CD-Infrastruktur**
- **Projekte, die mehrmals täglich deployen** wollen
- **Teams, die Pair Programming praktizieren** (ersetzt Code Review)
- **Microservices-Architekturen** mit kleinen, fokussierten Repositories

---

□□ Der große Vergleich

Aspekt	Git Flow	GitHub Flow	Trunk-Based
Komplexität	Hoch	Niedrig	Mittel*
Branch-Typen	5 (main, develop, feature, release, hotfix)	2 (main, feature)	1 (main)
Lebensdauer Feature-Branch	Tage bis Wochen	Stunden bis Tage	Keine oder < 2 Tage
Release-Modell	Geplante Versionen	Kontinuierlich	Kontinuierlich
Merge-Frequenz	Selten, dafür größer	Häufig	Sehr häufig (mehrmals täglich)
CI/CD-Anforderung	Optional	Empfohlen	Pflicht
Feature Flags nötig?	Nein	Selten	Ja
Parallele Versionen	Ja	Nein	Nein
Team-Größe	Mittel bis groß	Klein bis mittel	Klein bis mittel (erfahren)
Lernkurve	Steil	Flach	Mittel

**Die Komplexität bei TBD liegt nicht im Branching, sondern in den Praktiken (Feature Flags, kleine Commits, CI/CD).*

□□ Meine Empfehlung für dein PHP-Webprojekt

Du beschreibst ein **mittelgroßes PHP-Webprojekt mit regelmäßigen Releases**. Lass mich die Faktoren analysieren:

Analyse deiner Situation

- **„Mittelgroß"** deutet auf ein Team von 2–8 Entwicklern hin
- **„PHP-Webprojekt"** ist typischerweise eine Web-Applikation, die deployt wird (nicht installierte Software)
- **„Regelmäßige Releases"** können unterschiedlich interpretiert werden:
  - Wöchentlich/zweiwöchentlich → GitHub Flow geeignet
  - Monatlich mit festen Versionen → Git Flow könnte passen

    - Kontinuierlich → Trunk-Based möglich

Meine Empfehlung: **GitHub Flow mit Release-Tags** □□

Für dein Szenario empfehle ich **GitHub Flow**, aber mit einer kleinen Erweiterung für die Versionierung:

```
flowchart LR
    subgraph Entwicklung
        A["Feature-Branch\nerstellen"] --> B["Entwickeln\nund testen"]
        B --> C["Pull Request\nerstellen"]
        C --> D["Code Review"]
        D --> E["Merge in main"]
    end

    subgraph Release
        E --> F{"Release\nfällig?"}
        F -->|Ja| G["Tag erstellen\nv1.2.0"]
        F -->|Nein| H["Weiter entwickeln"]
        G --> I["Deployment"]
    end

    style G fill:#90EE90,color:#000000
    style I fill:#87CEEB,color:#000000
```

Warum GitHub Flow für dich?

1. **Einfachheit:** Du und dein Team müsst nicht fünf verschiedene Branch-Typen jonglieren. Die Regeln passen auf einen Bierdeckel.
2. **Web-Applikation:** PHP-Webprojekte werden deployt, nicht installiert. Du brauchst keine parallelen Versionen zu unterstützen – es gibt nur „was gerade live ist".
3. **Pull Requests als Qualitätssicherung:** Jede Änderung durchläuft einen Review-Prozess. Das ist Gold wert für die Code-Qualität.
4. **Flexibilität bei Releases:** Du kannst so oft oder selten releasen, wie du möchtest. Ein Release ist einfach ein Tag auf `main`.
5. **Einfacher Einstieg:** Falls Teammitglieder noch nicht so Git-erfahren sind, ist GitHub Flow viel leichter zu erlernen als Git Flow.

Wie du „regelmäßige Releases" mit GitHub Flow umsetzt

Anstatt Release-Branches zu nutzen, arbeitest du mit **Tags** und einem **Release-Rhythmus**:

```
# Nach dem Merge aller Features für Version 1.2.0
git tag -a v1.2.0 -m "Release 1.2.0: User Dashboard, Performance Improvements"
git push origin v1.2.0
```

In GitHub kannst du dann einen formalen Release erstellen, der auf diesen Tag verweist – mit Changelog, Release Notes und ggf. Assets.

Dein angepasster Workflow

1. **Für jede Aufgabe:** Erstelle einen Branch von `main`

   ```
   git checkout main
   git pull
   git checkout -b feature/user-dashboard
   ```

2. **Entwickle und committe** regelmäßig mit aussagekräftigen Nachrichten
3. **Erstelle einen Pull Request** auf GitHub, wenn bereit für Review
4. **Nach Approval:** Merge in `main` (ich empfehle „Squash and Merge" für eine saubere Historie)
5. **Für Releases:** Wenn genügend Features gemergt sind, erstelle einen Tag und deploye:

   ```
   git checkout main
   git pull
   git tag -a v1.2.0 -m "Release 1.2.0"
   git push origin v1.2.0
   # Deployment triggern (manuell oder automatisch via GitHub Actions)
   ```

6. **Für Hotfixes:** Behandle sie wie normale Features – Branch erstellen, fixen, PR, mergen. Danach ggf. Patch-Version taggen (`v1.2.1`).

Wann du doch Git Flow in Betracht ziehen solltest

Überdenke meine Empfehlung, wenn *einer* dieser Punkte zutrifft:

- Du musst **mehrere Versionen parallel unterstützen** (z.B. v1.x und v2.x gleichzeitig pflegen)
- Du hast **sehr lange Release-Zyklen** (> 1 Monat) mit umfangreichen QA-Phasen
- Du arbeitest in einer **regulierten Branche** (Medizin, Finanzen) mit strengen Audit-Anforderungen
- Dein Team ist **sehr groß** (> 10 Entwickler) und braucht klare Abgrenzungen

Wann Trunk-Based Development interessant wäre

- Du möchtest **mehrmals täglich deployen**
- Dein Team ist **sehr erfahren** und praktiziert Pair Programming
- Du hast eine **exzellente CI/CD-Pipeline** mit < 10 Minuten Build-Zeit
- Du bist bereit, in **Feature Flags** zu investieren

□□ Zusammenfassung

Strategie	Motto	Ideal für
Git Flow	„Struktur und Kontrolle"	Versionierte Software, große Teams, lange Zyklen
GitHub Flow	„Ship early, ship often"	Web-Apps, kleine-mittlere Teams, kontinuierliche Delivery
Trunk-Based	„Immer integriert"	Hochperformante Teams, mehrfach tägliches Deployment

Für dein mittelgroßes PHP-Webprojekt ist **GitHub Flow** der Sweet Spot: Einfach genug, um schnell produktiv zu sein, aber strukturiert genug für professionelle Zusammenarbeit. Mit Tags und GitHub Releases bekommst du die Versionierung, die du für „regelmäßige Releases" brauchst – ohne die Komplexität von Git Flow.

Starte mit GitHub Flow, und wenn du merkst, dass du mehr Struktur brauchst, kannst du immer noch zu Git Flow wechseln. Der umgekehrte Weg (von komplex zu einfach) ist erfahrungsgemäß schwieriger! □□

Revision #1
Created 2026-06-29 15:25:42 UTC by art10m
Updated 2026-06-29 15:28:31 UTC by art10m