BRANCHING MODELLE

Bestimmung einer Strategie für jedes Projekt

INHALT

- Motivation
- Vorstellung Branching Modelle
- Kriterien/Einflussfaktoren beim Branching/Merging
- Fazit
- Fragen/Diskussion

MOTIVATION

Problem

- das Vorhandensein einer geeigneten Branching-Strategie sorgt beim Softwareentwicklungsprozess für eine effizientere Arbeitsweise im Team
- es existiert eine Vielzahl von Branching-Modellen
- Wie wählt man ein geeignetes Branching-Modell aus?

Ziel

Erarbeiten einer Vorgehensweise, mit welcher es möglich ist, das am besten geeignete Branching-Modell für ein Projekt anhand seiner Eigenschaften zu bestimmen.

BRANCHING-MODELLE

- regeln den Umgang mit Versionsverwaltungssystemen beim gemeinsamen Entwickeln im Team
- allgemeine Vorgaben für das Anlegen und Löschen von Branches, Tags, usw.
- konkrete Naming-Patterns, Integrationsmethode, usw.
- Eigenschaften:
  - Möglichkeit, eine stabile Codebasis zu pflegen
  - Unterstützung von kontinuierlicher Integration/kontinuierlichem Deployment
  - Möglichkeit, verschiedene Versionen parallel zu pflegen
  - Möglichkeit, isolierte/parallele Featureentwicklung zu betreiben
  - Möglichkeit, kritische Bugfixes zeitnah in Produktion zu überführen
  - Möglichkeit, einen Rollback auf eine vorhergehende Version durchzuführen

ZENTRALES MODELL

- ein öffentlicher Branch, auf den alle Entwickler committen
- kommt oft bei zentraler Versionsverwaltung zum Einsatz
- wird oft in Verbindung mit Continuous Integration verwendet
- Trunk Based Development[1]

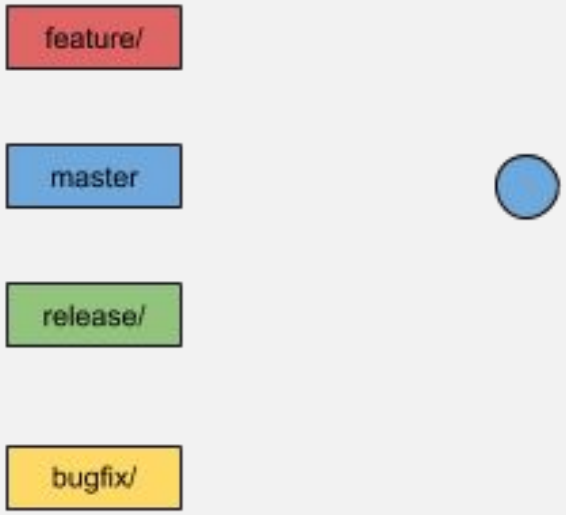
[1] https://trunkbaseddevelopment.com/

TOPIC BRANCHES

- ein öffentlicher Hauptzweig (master)
- lokale Branches werden nach Aufgabengebiet angelegt und nach deren Bearbeitung in den Haupt-Branch gemergt
- Features, Hotfixes, Releases
- OneFlow[2], GitHubFlow[3]

[2] http://endoflineblog.com/oneflow-a-git-branching-model-and-workflow
[3] http://scottchacon.com/2011/08/31/github-flow.html

ONEFLOW – FEATURE BRANCHES



ONEFLOW – FEATURE BRANCHES

feature/

master

release/

bugfix/

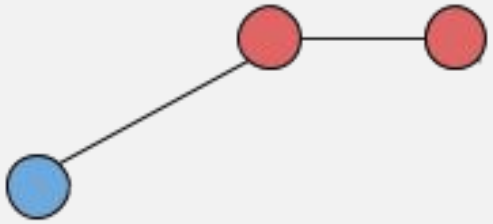
ONEFLOW – FEATURE BRANCHES

feature/

master

release/

bugfix/



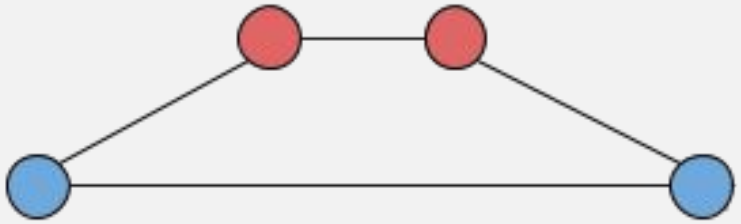
ONEFLOW – FEATURE BRANCHES

feature/

master

release/

bugfix/



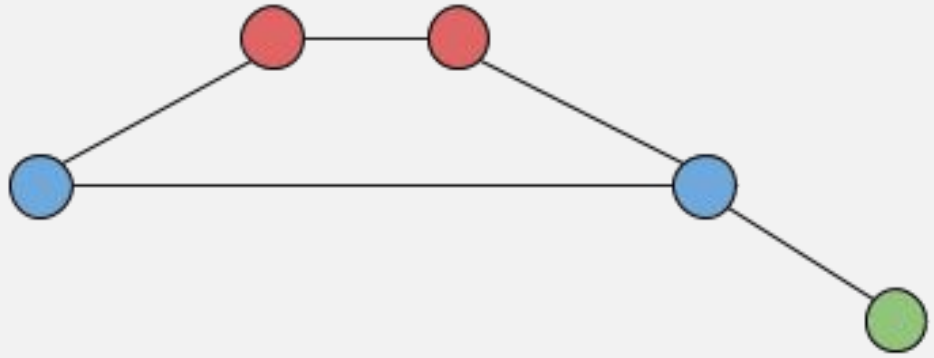
ONEFLOW – RELEASE BRANCHES

feature/

master

release/

bugfix/



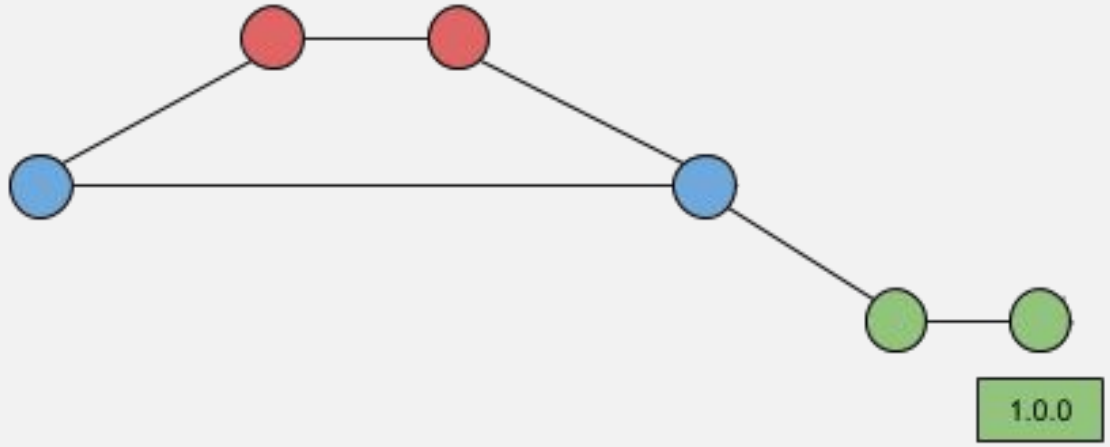
ONEFLOW – RELEASE BRANCHES

feature/

master

release/

bugfix/



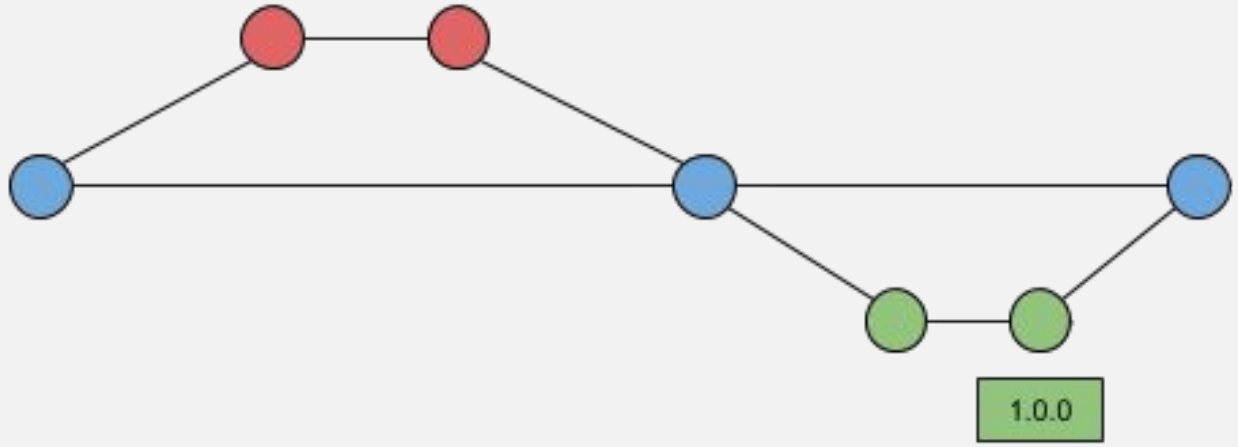
ONEFLOW – RELEASE BRANCHES

feature/

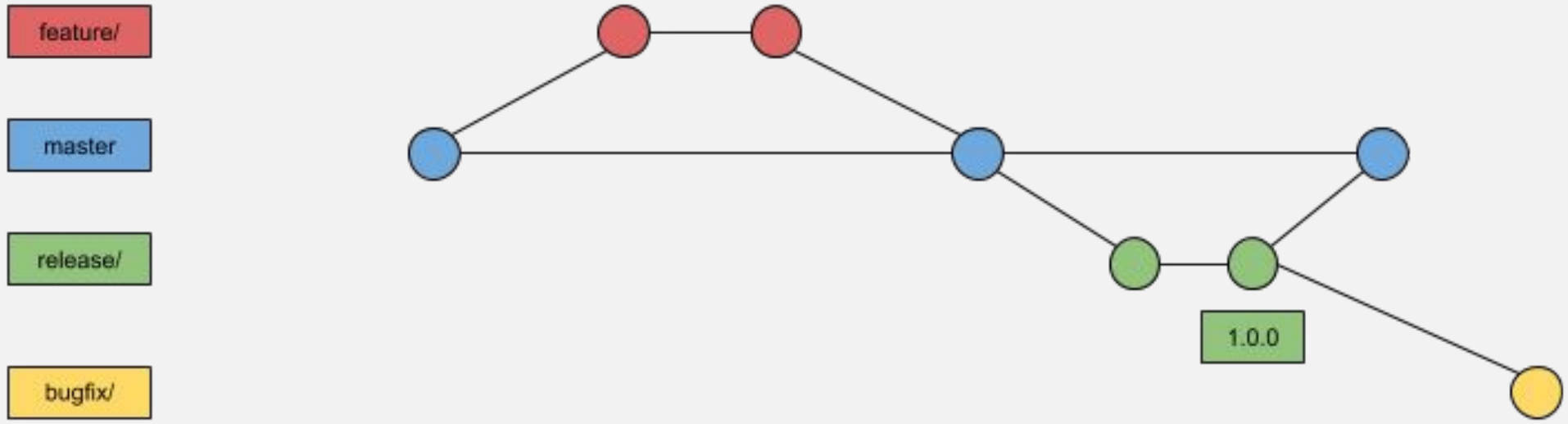
master

release/

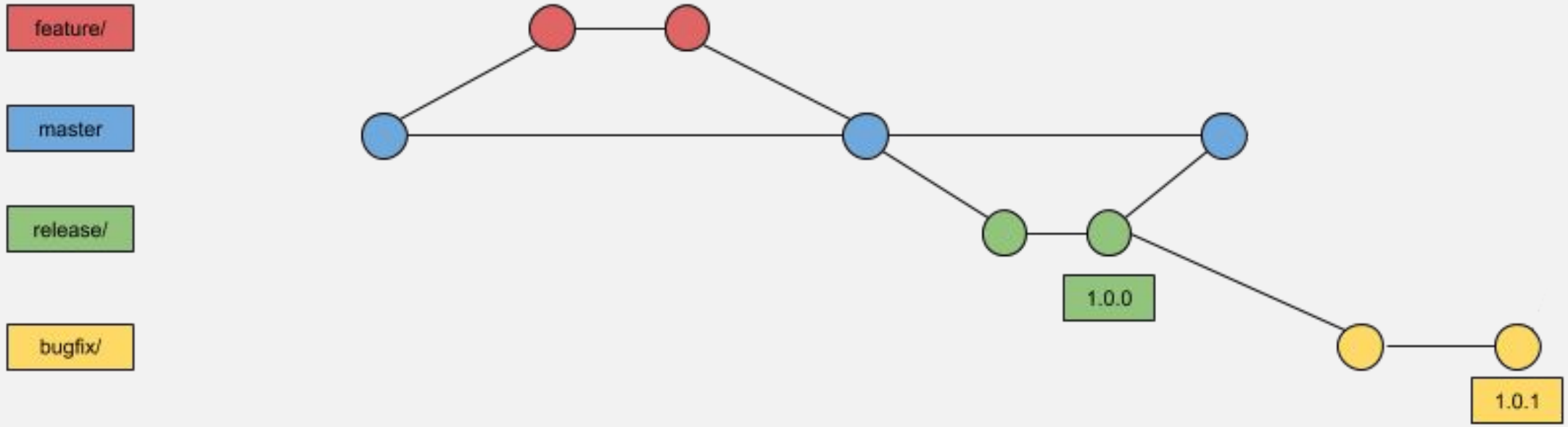
bugfix/



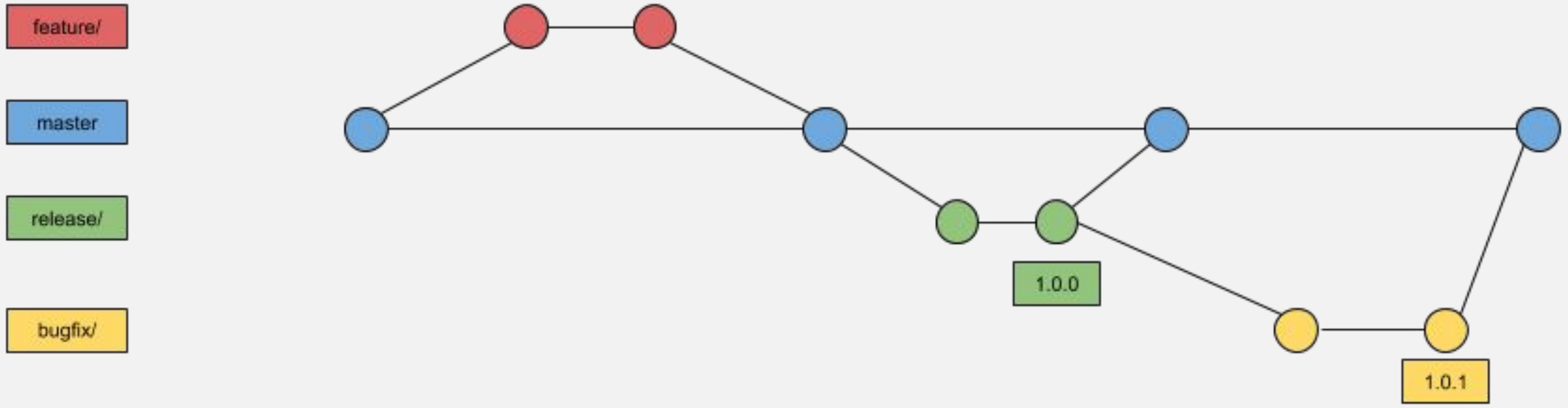
ONEFLOW – HOTFIX BRANCHES



ONEFLOW – HOTFIX BRANCHES



ONEFLOW – HOTFIX BRANCHES



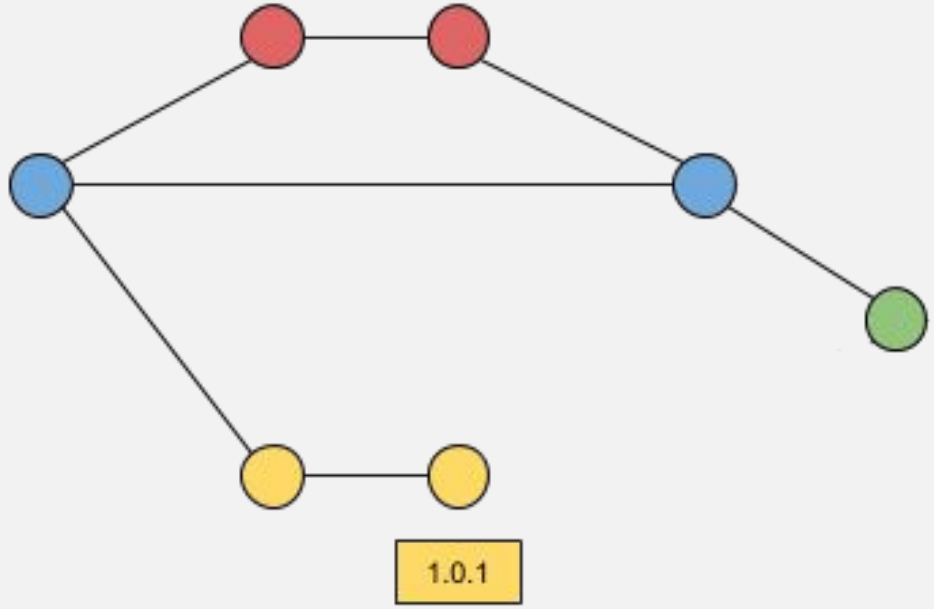
ONEFLOW – HOTFIX BRANCHES

feature/

master

release/

bugfix/



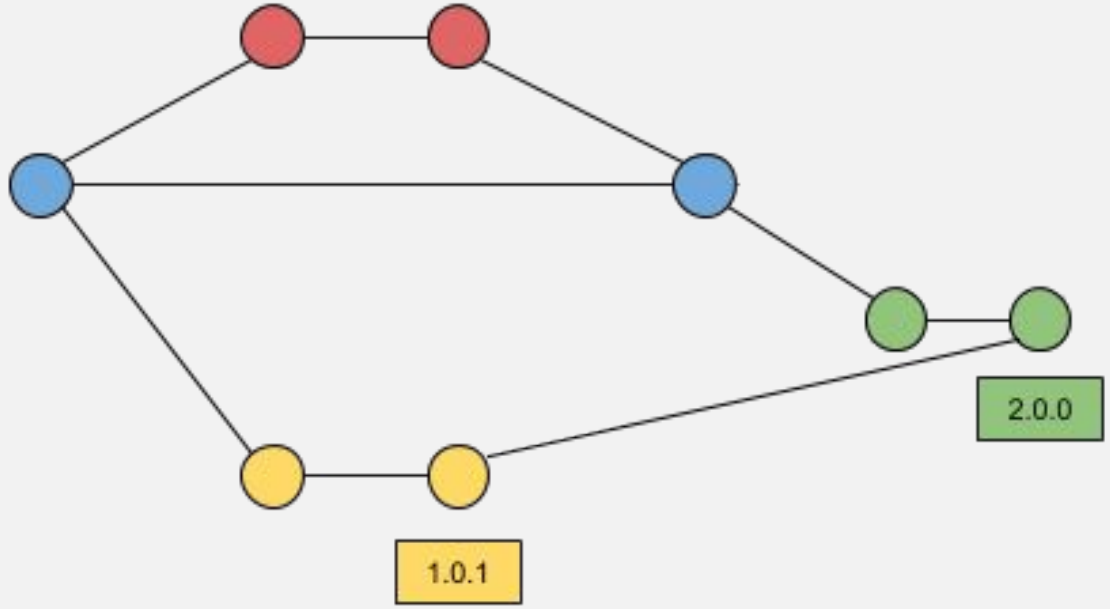
ONEFLOW – HOTFIX BRANCHES

feature/

master

release/

bugfix/



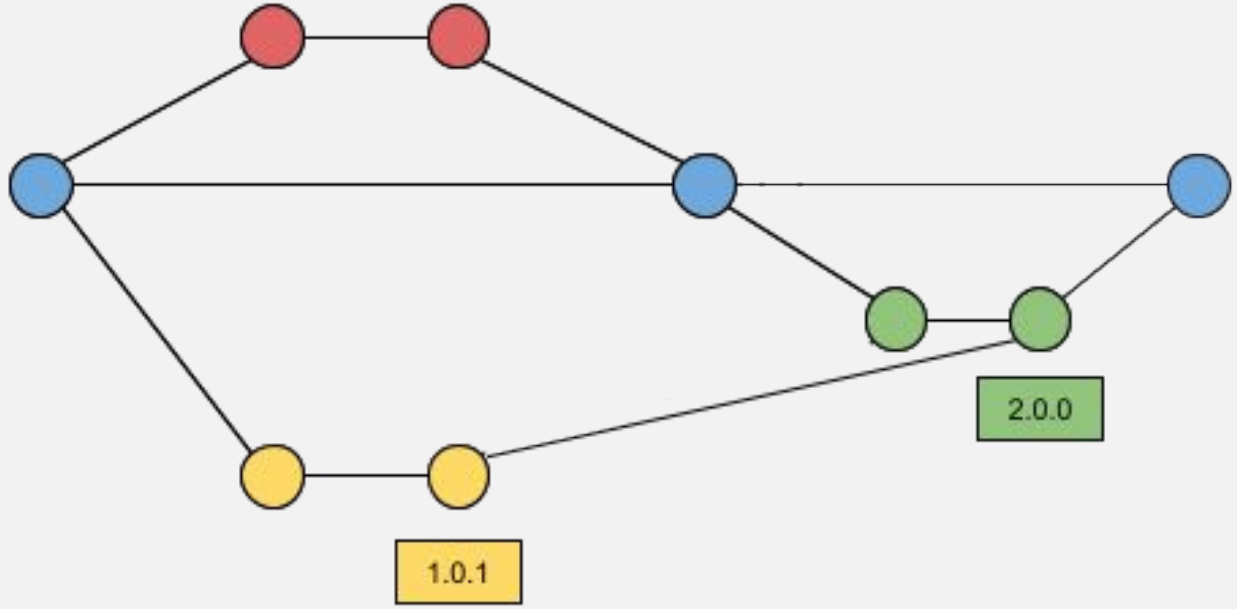
ONEFLOW – HOTFIX BRANCHES

feature/

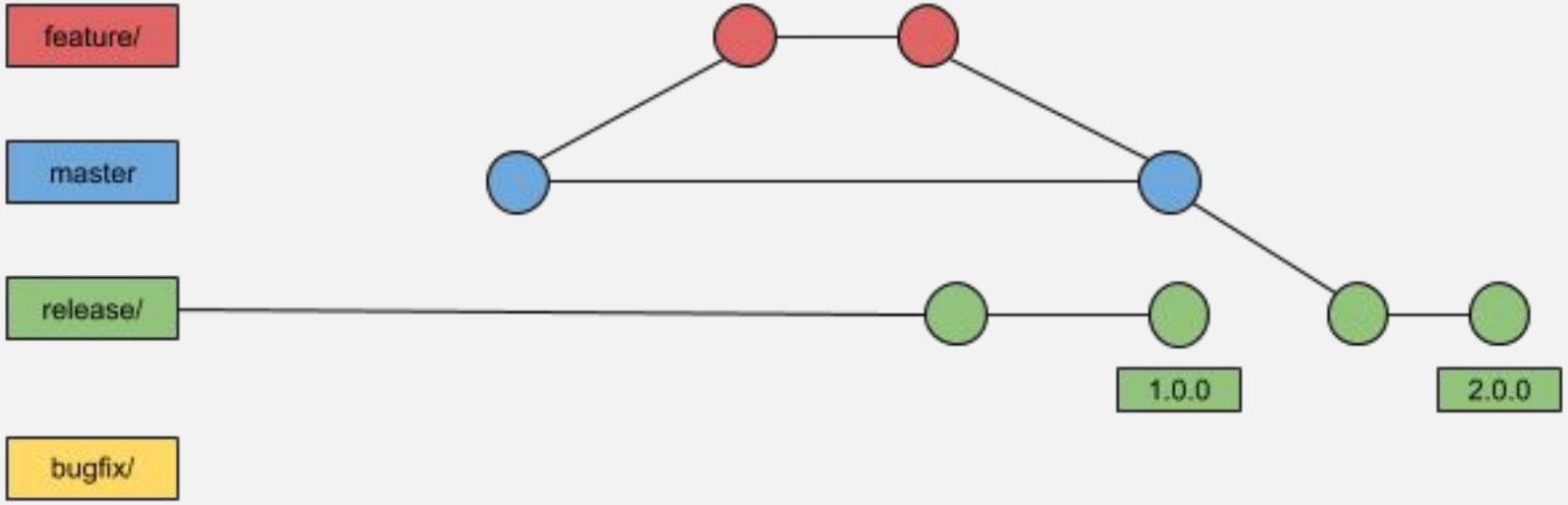
master

release/

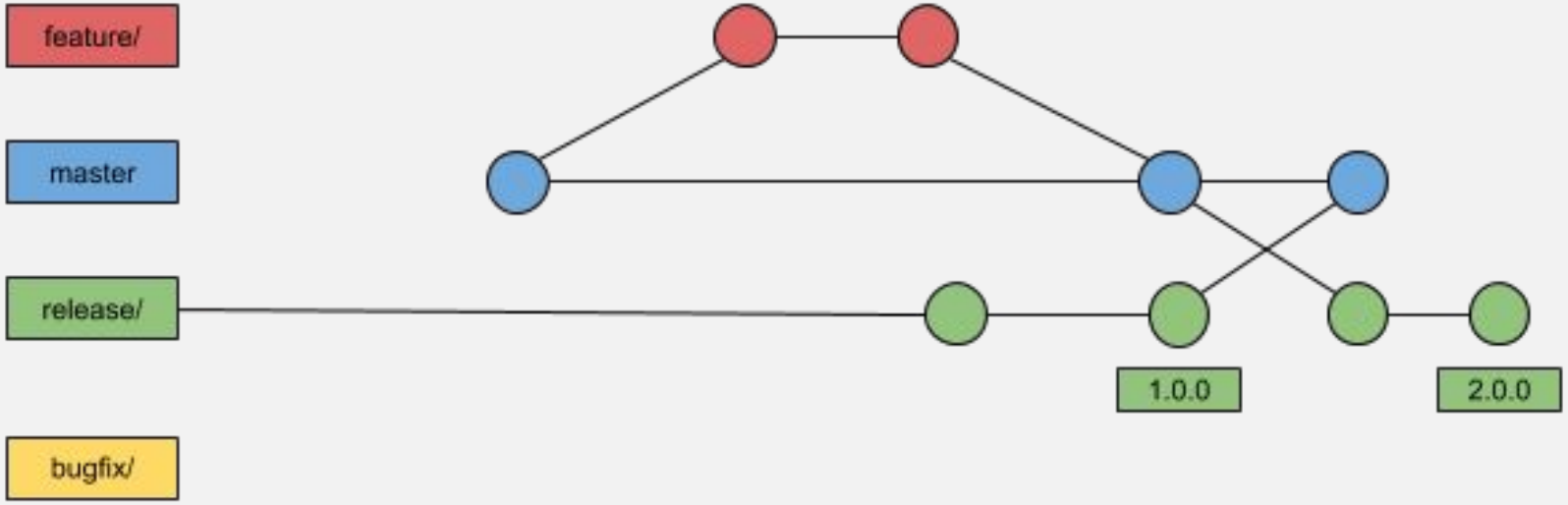
bugfix/



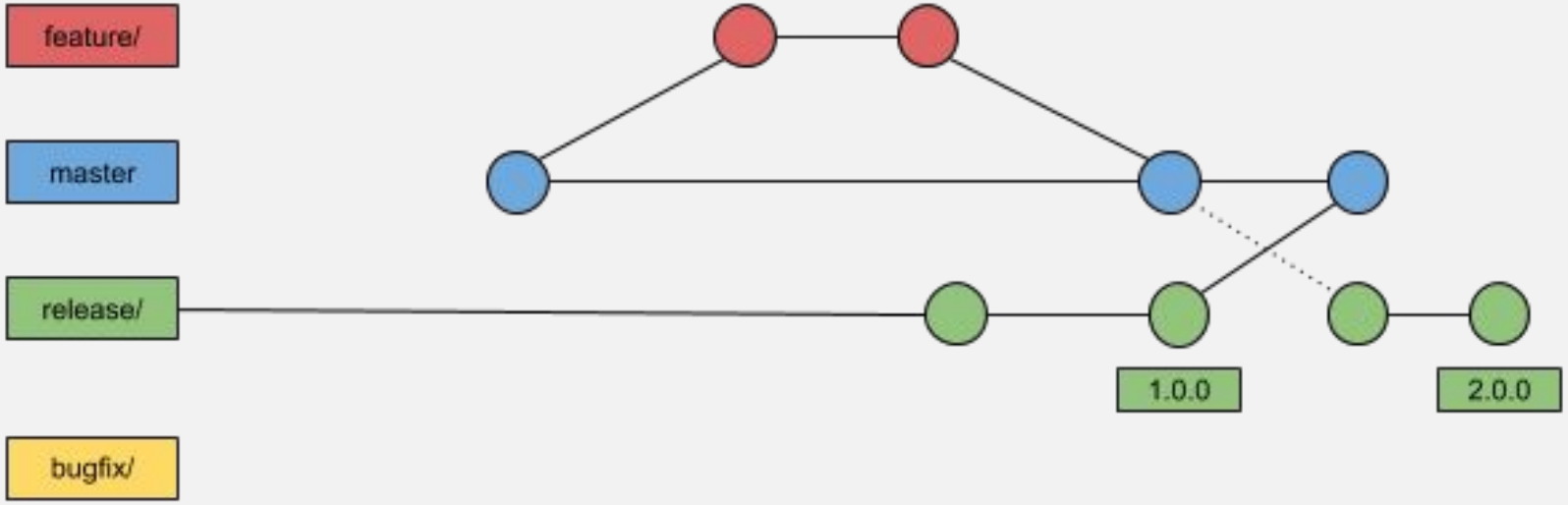
ONEFLOW – PARALLELE RELEASES



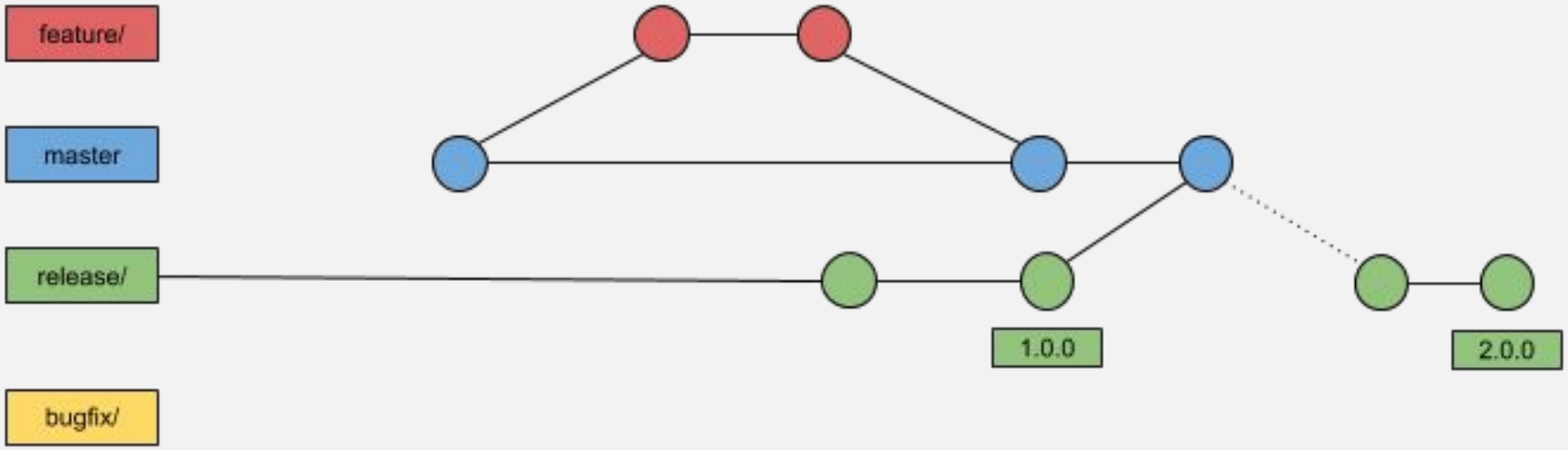
ONEFLOW – PARALLELE RELEASES



ONEFLOW – PARALLELE RELEASES



ONEFLOW – PARALLELE RELEASES



ONEFLOW ZUSAMMENFASSUNG

Vorteile:

- einfaches & vielseitig einsetzbares Branching Modell
- problemlose Integration in Issue Tracking Tools

Nachteile:

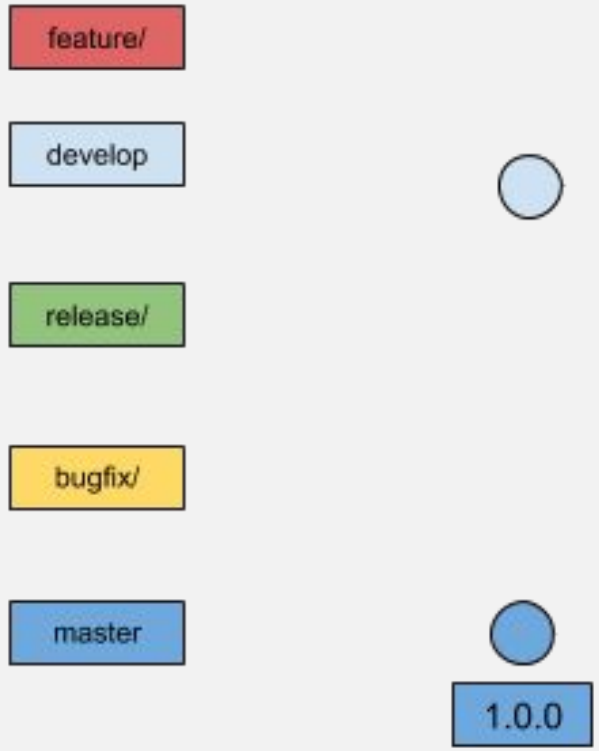
- master Branch kann schnell instabil werden
- für eine saubere Git Historie sollte mit Rebases gearbeitet werden
- keine Unterstützung für mehrere Releases

STABLE/DEV BRANCHES

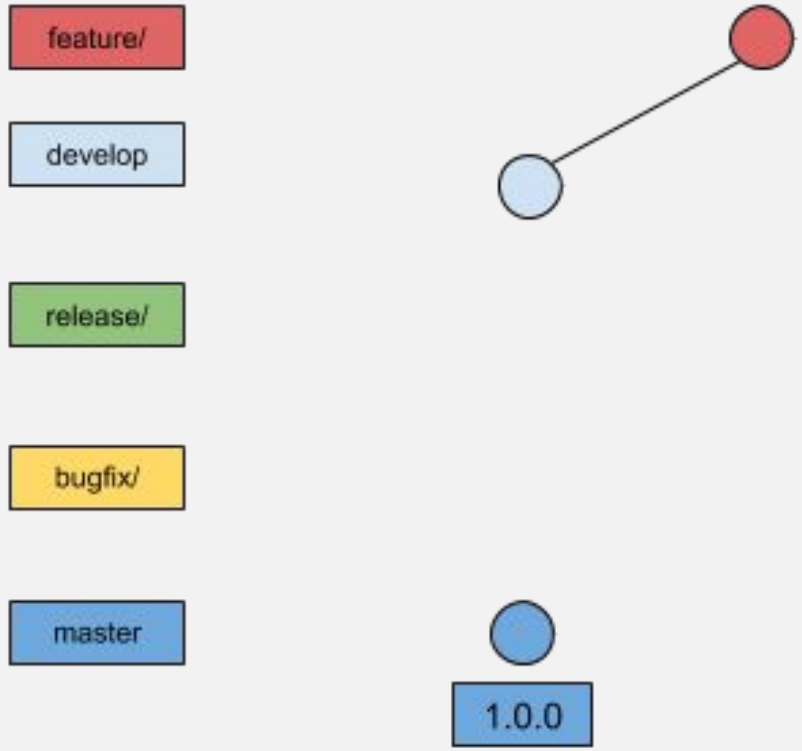
- ein öffentlicher Hauptzweig (stable) & ein öffentlicher
- Entwicklungszweig (develop)
- bei Release wird der develop- in den stable-Branch gemergt
- Bugfixes werden auf dem stable-Branch angelegt
- GitFlow[6]

[6] https://nvie.com/posts/a-successful-git-branching-model/

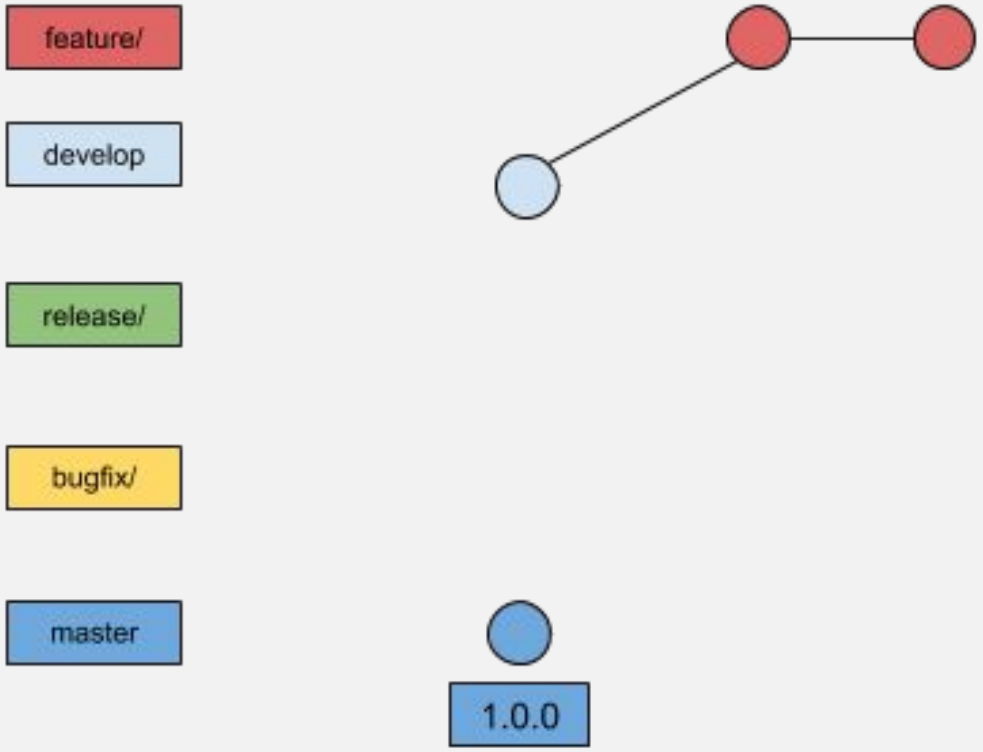
GITFLOW
FEATURE ENTWICKLUNG



GITFLOW
FEATURE ENTWICKLUNG



GITFLOW
FEATURE ENTWICKLUNG



GITFLOW
FEATURE ENTWICKLUNG

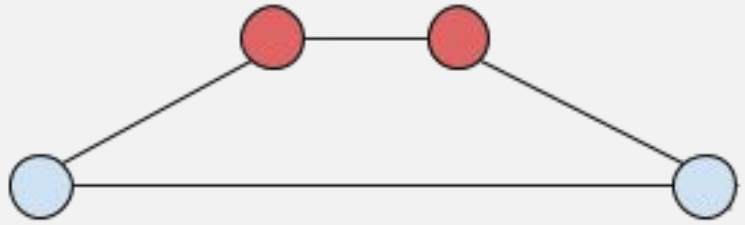
feature/

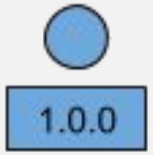
develop

release/

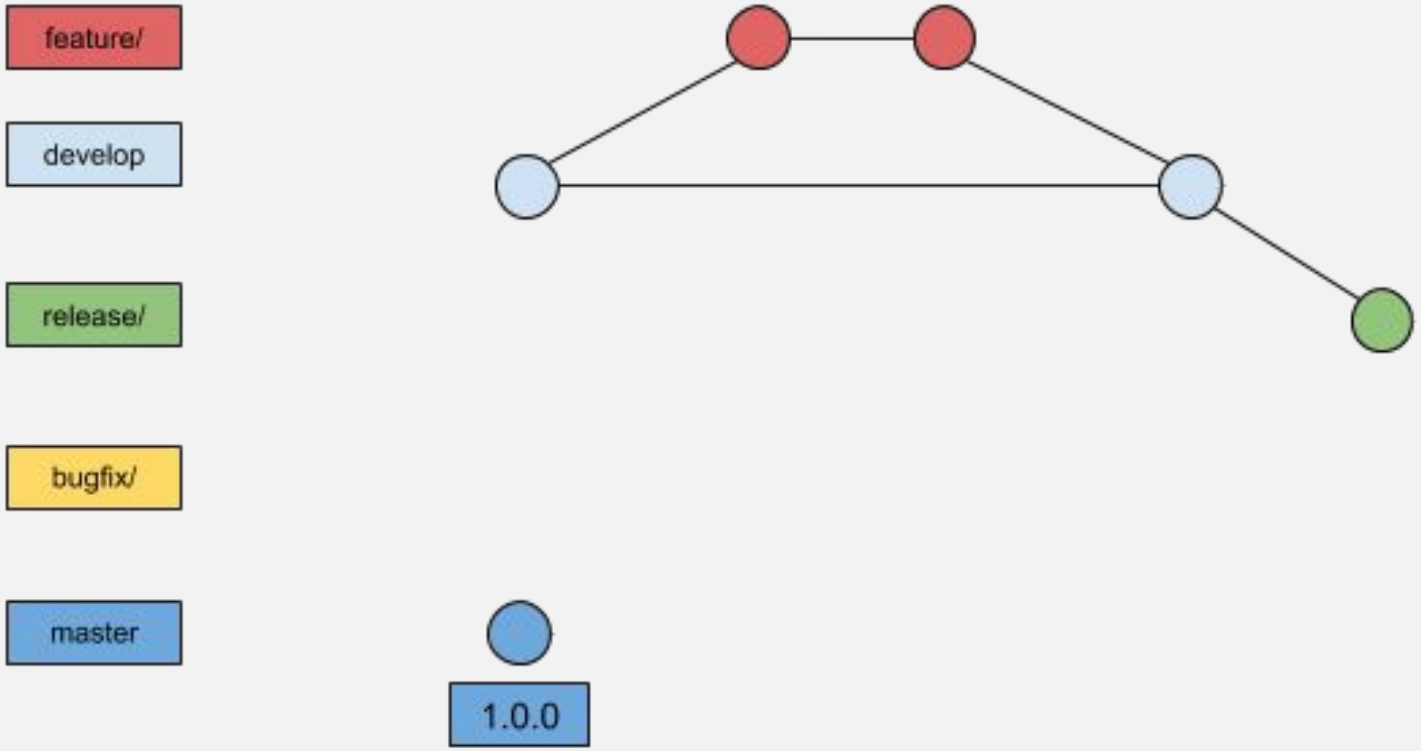
bugfix/

master

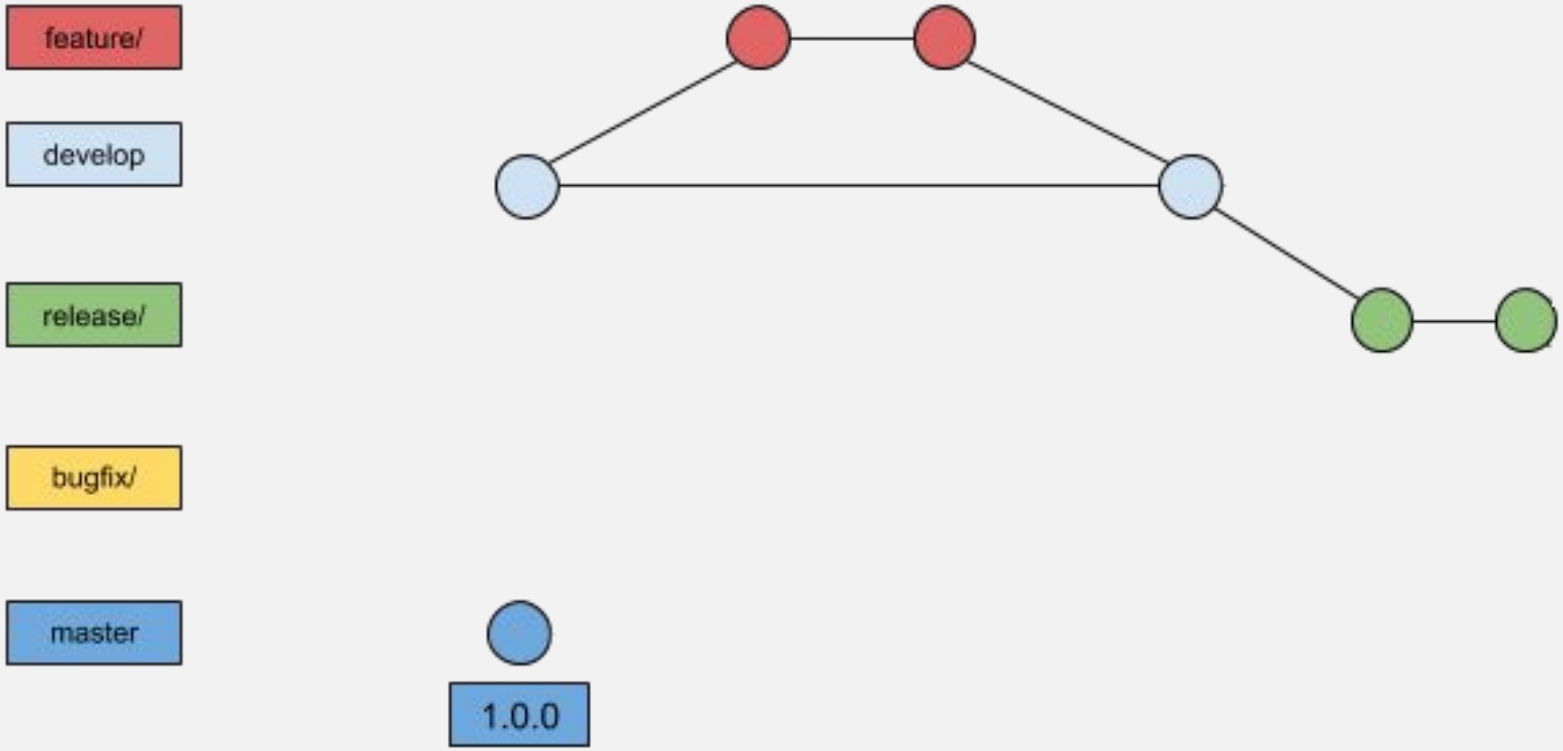




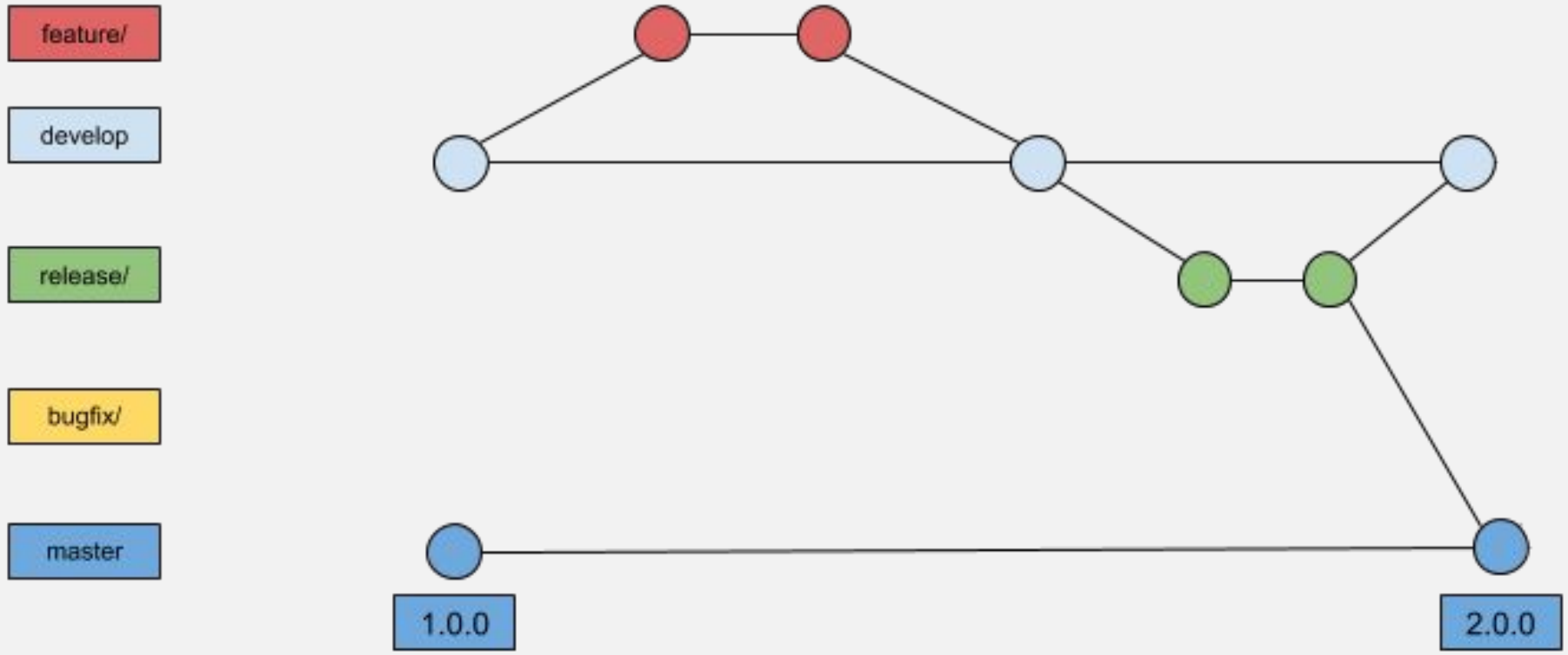
GITFLOW
RELEASE VORBEREITUNG



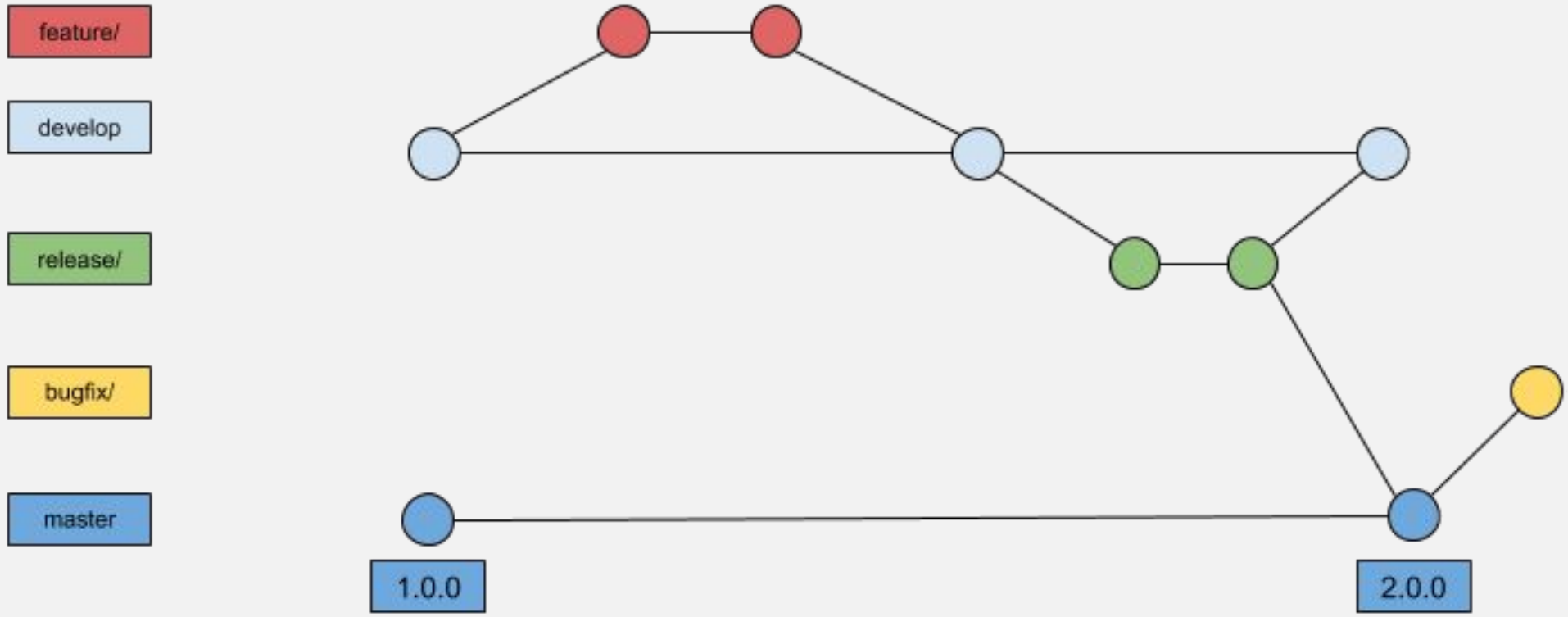
GITFLOW
RELEASE VORBEREITUNG



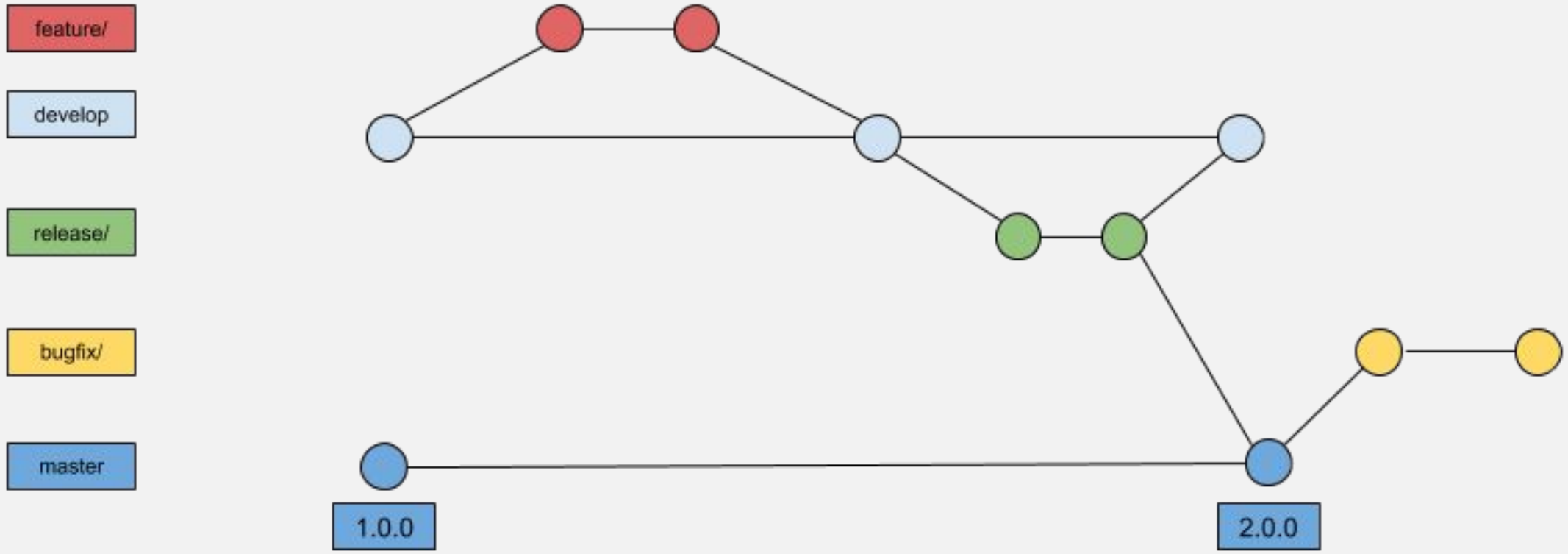
GITFLOW
RELEASE



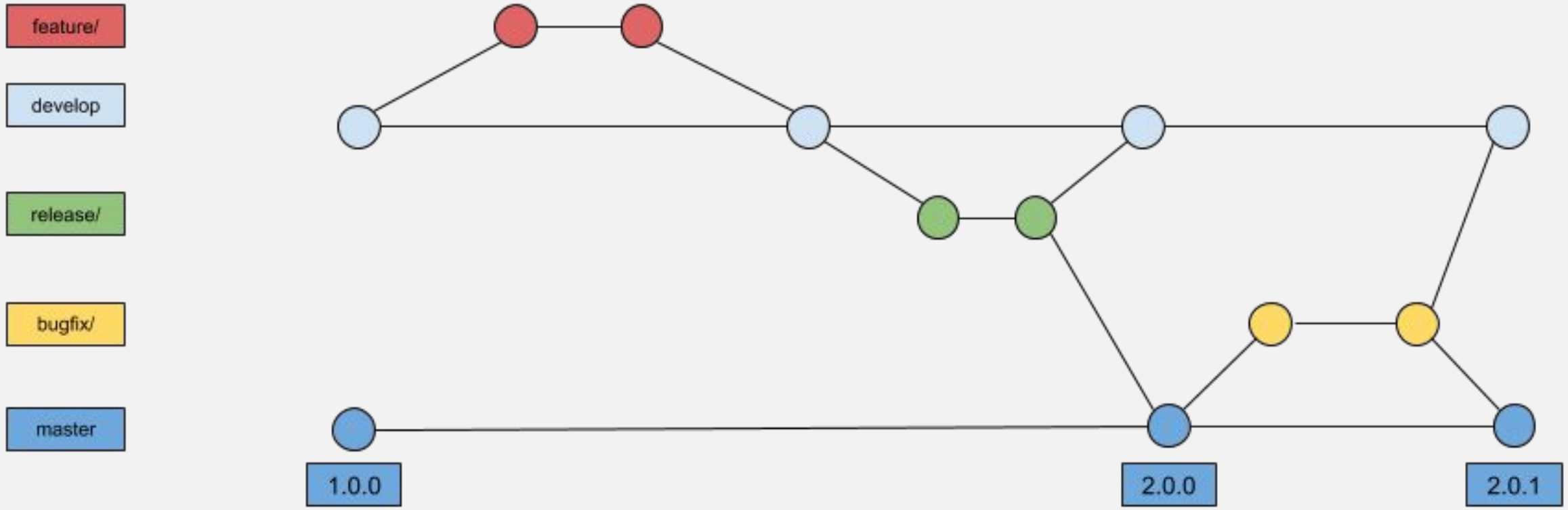
GITFLOW
HOTFIX



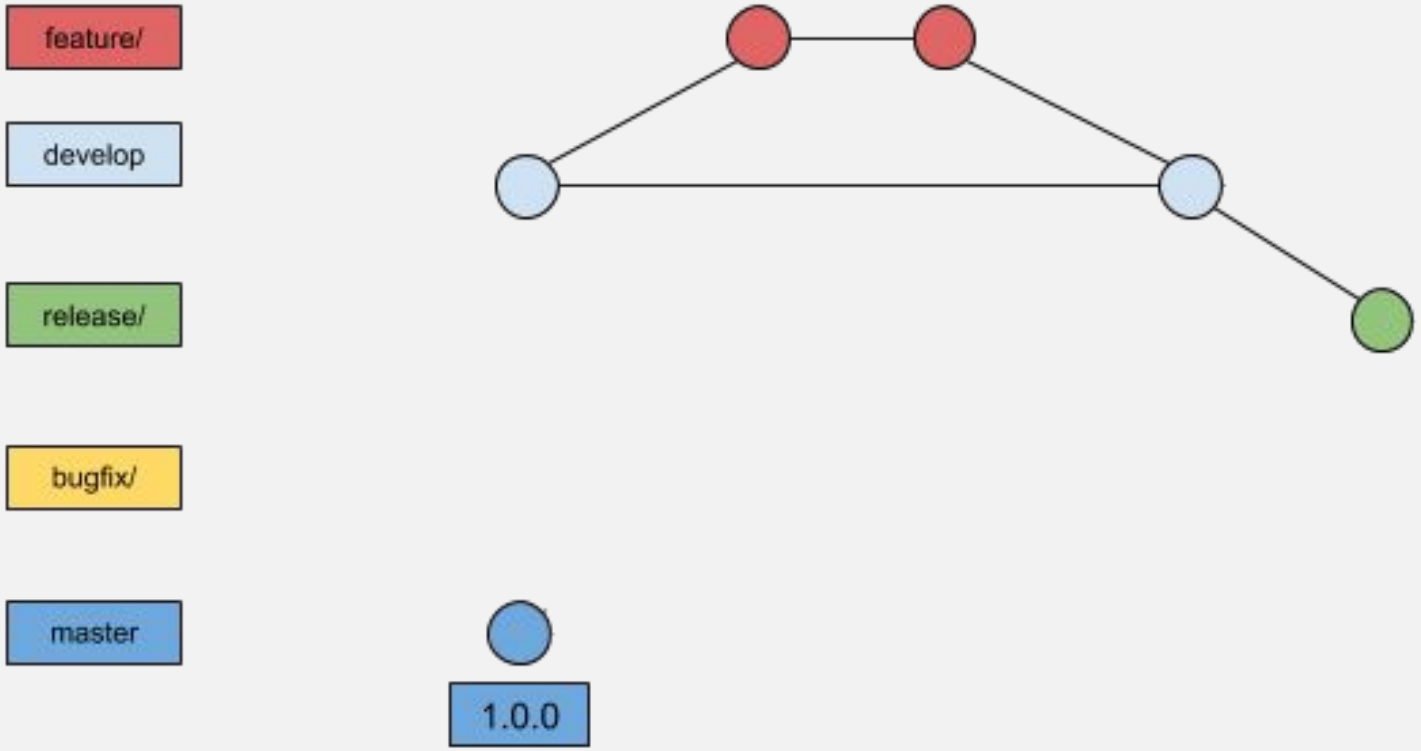
GITFLOW
HOTFIX



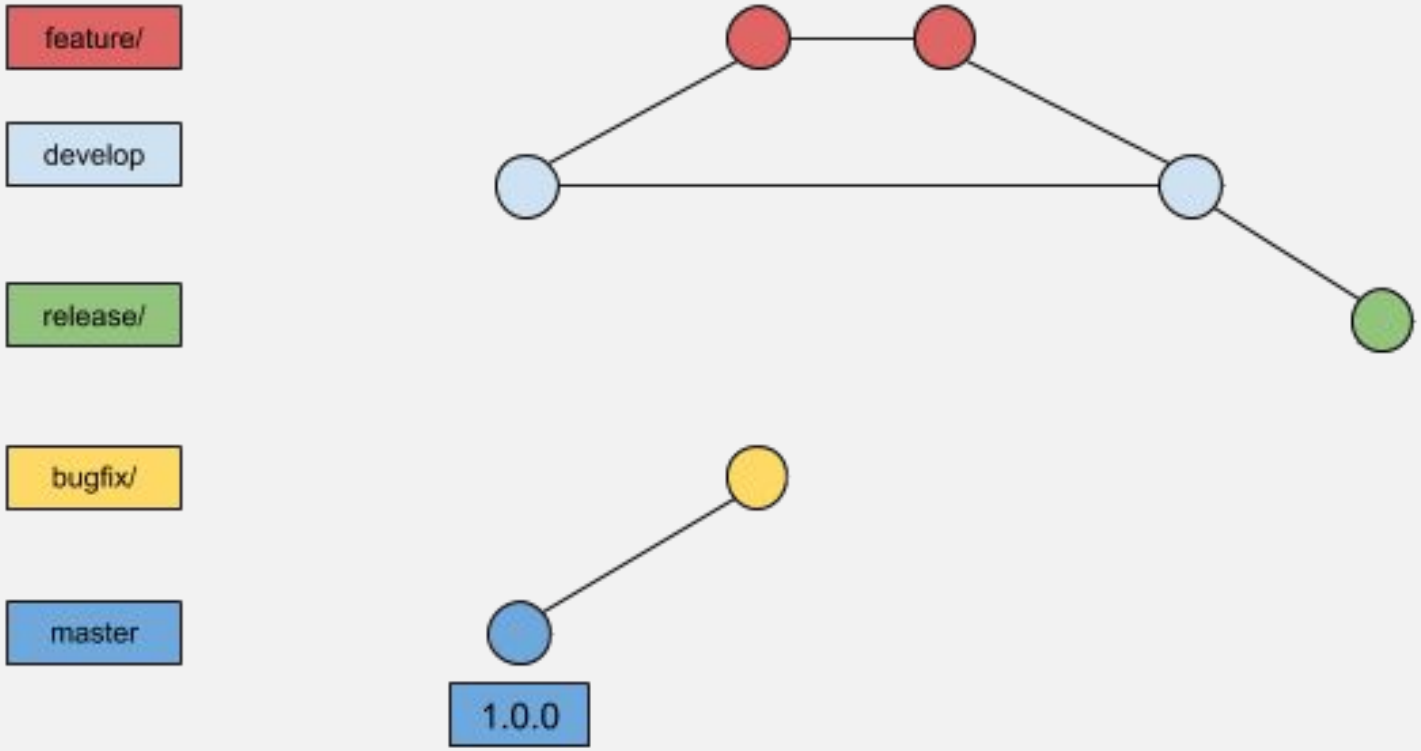
GITFLOW
HOTFIX RELEASE



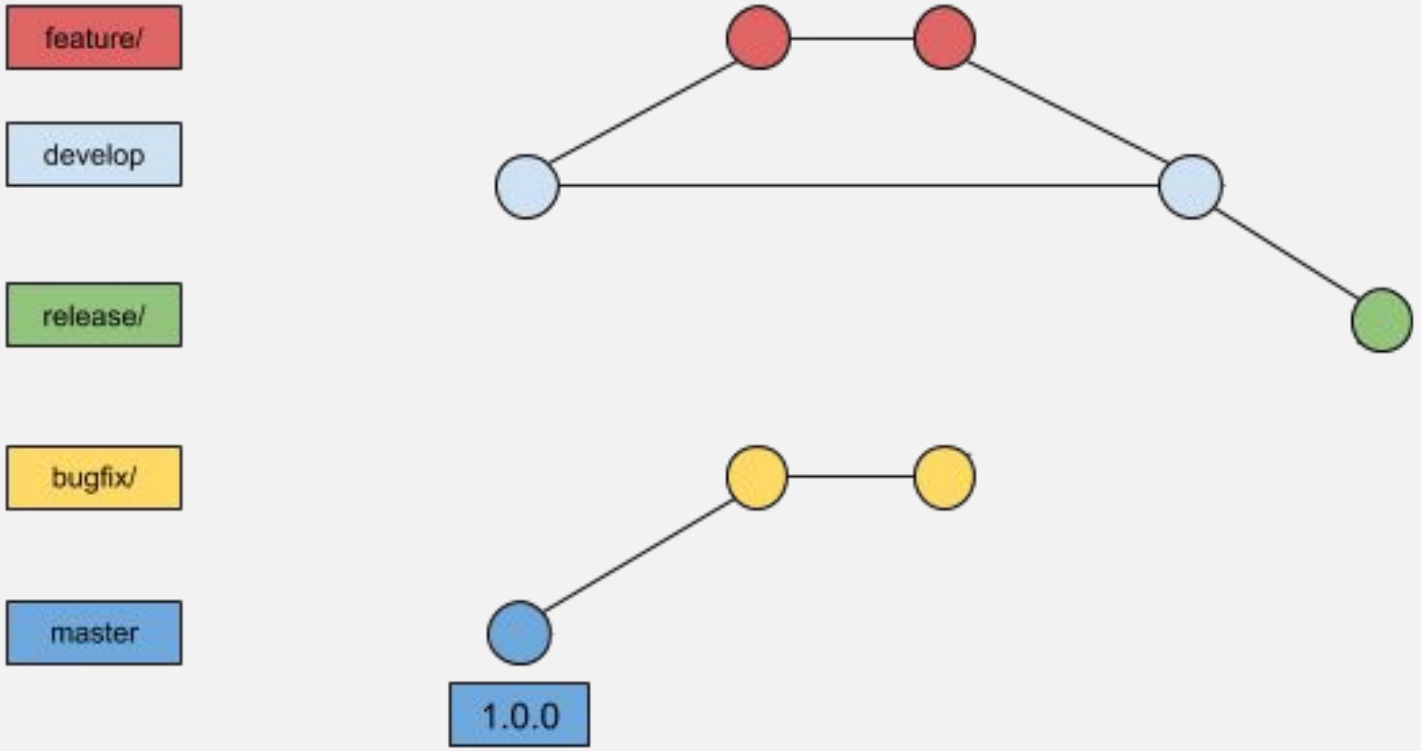
GITFLOW
HOTFIX BRANCHES



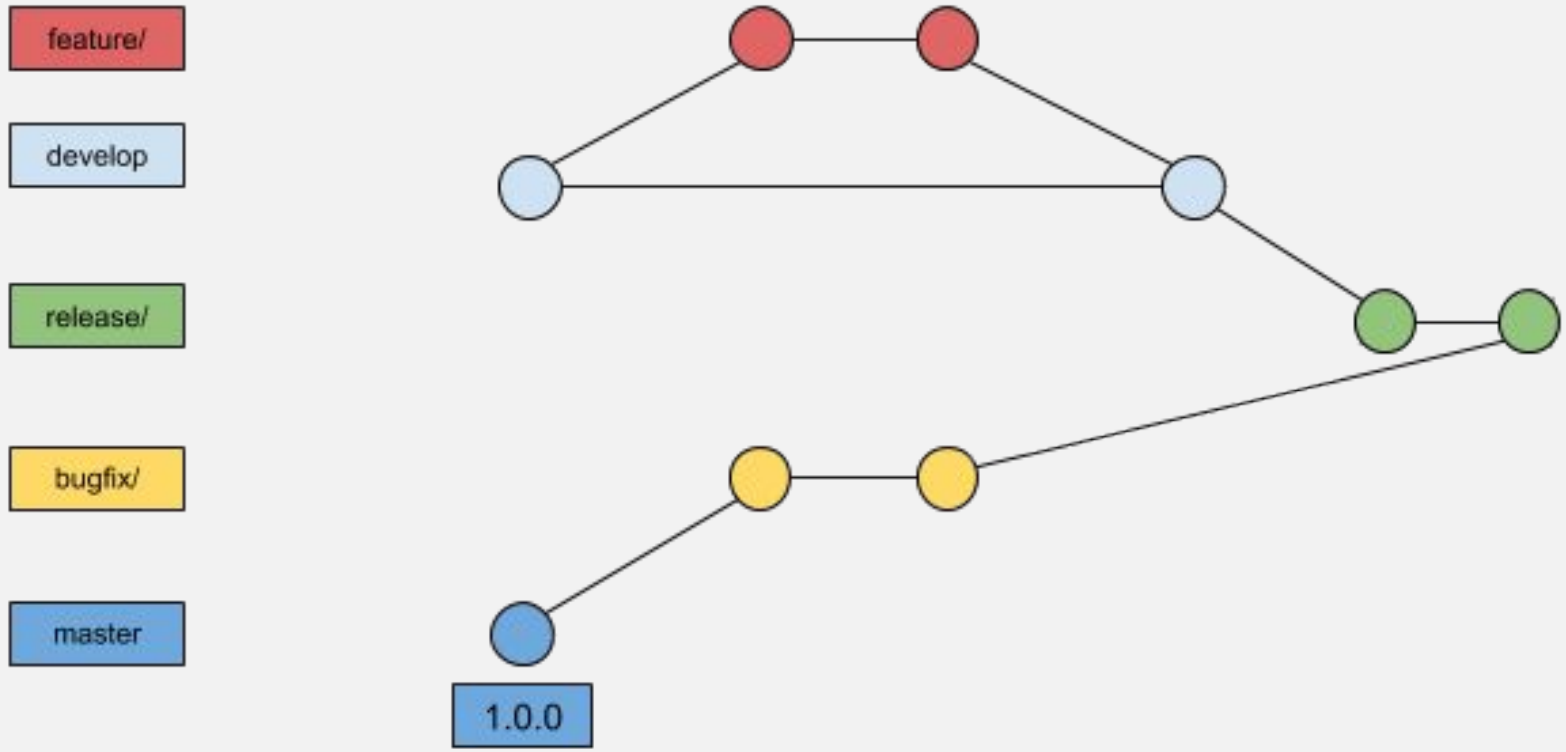
GITFLOW
HOTFIX BRANCHES



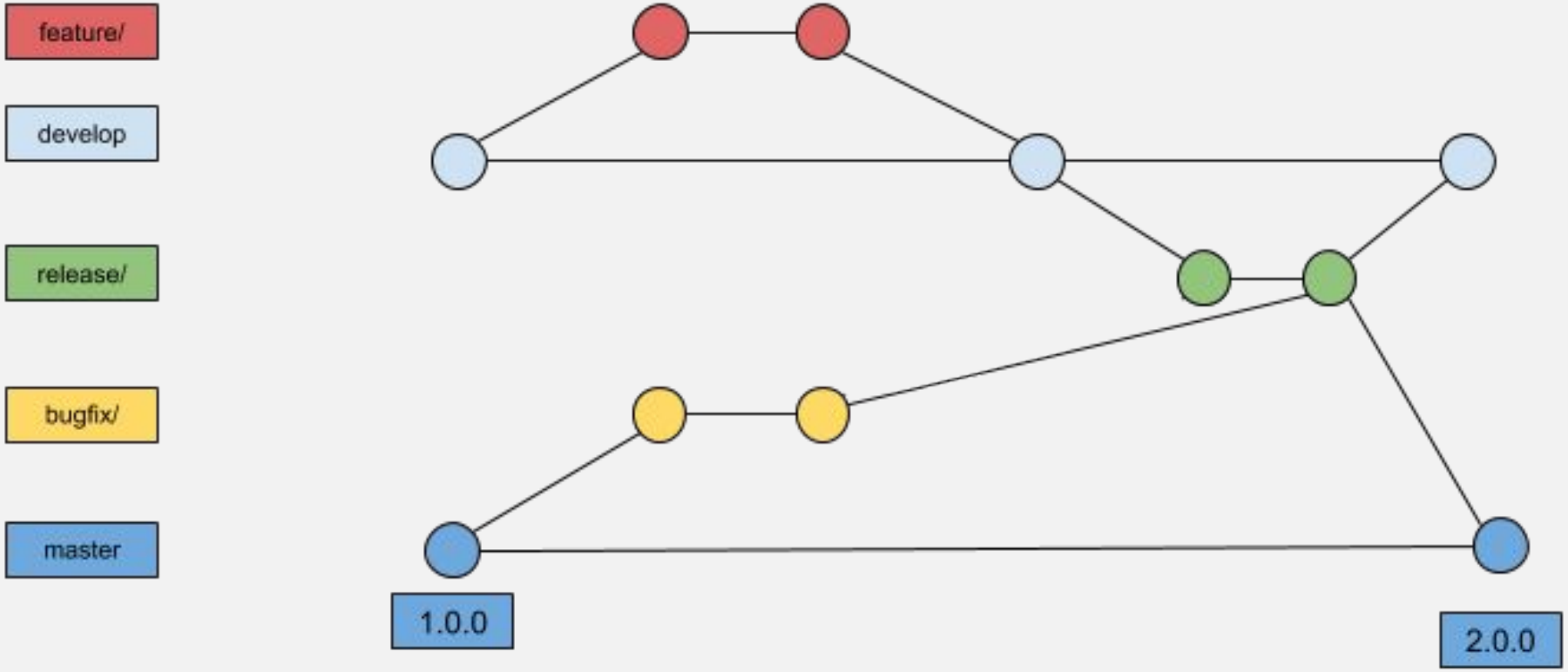
GITFLOW
HOTFIX BRANCHES



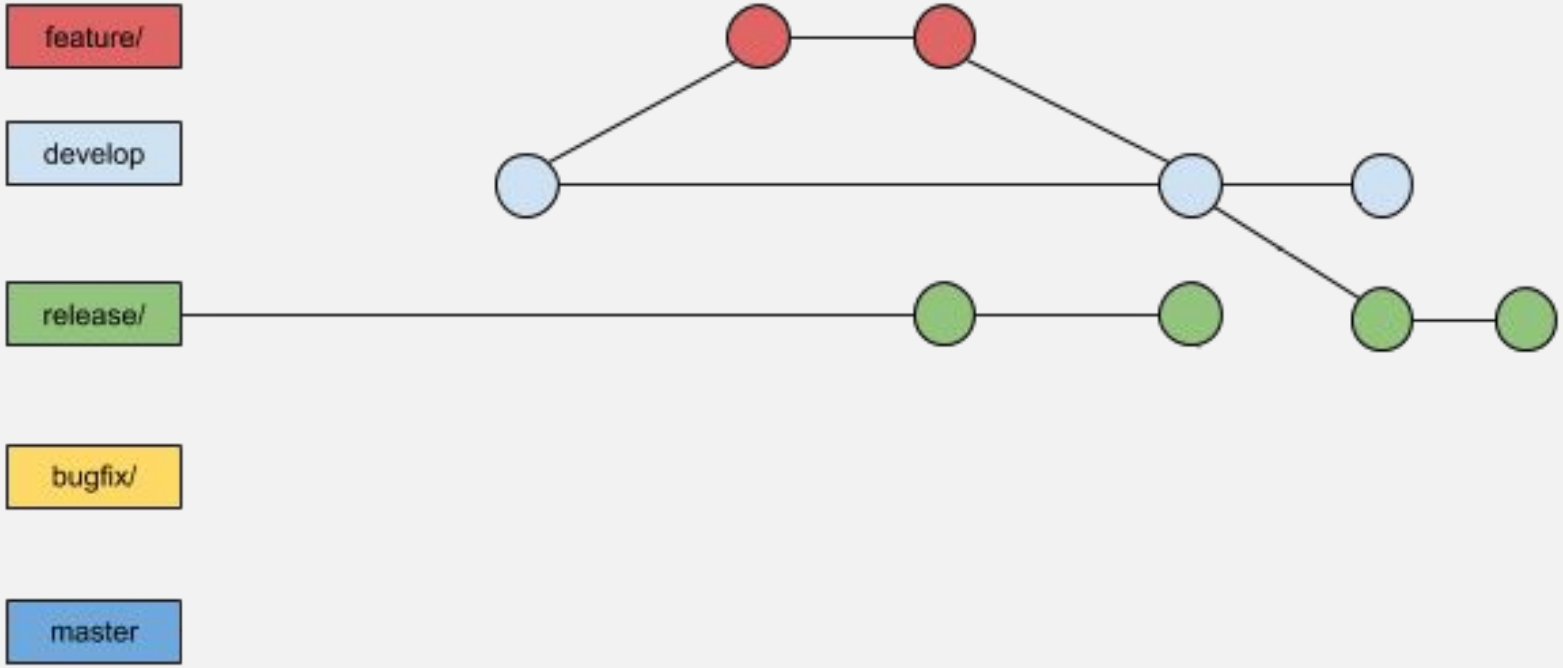
GITFLOW
HOTFIX BRANCHES



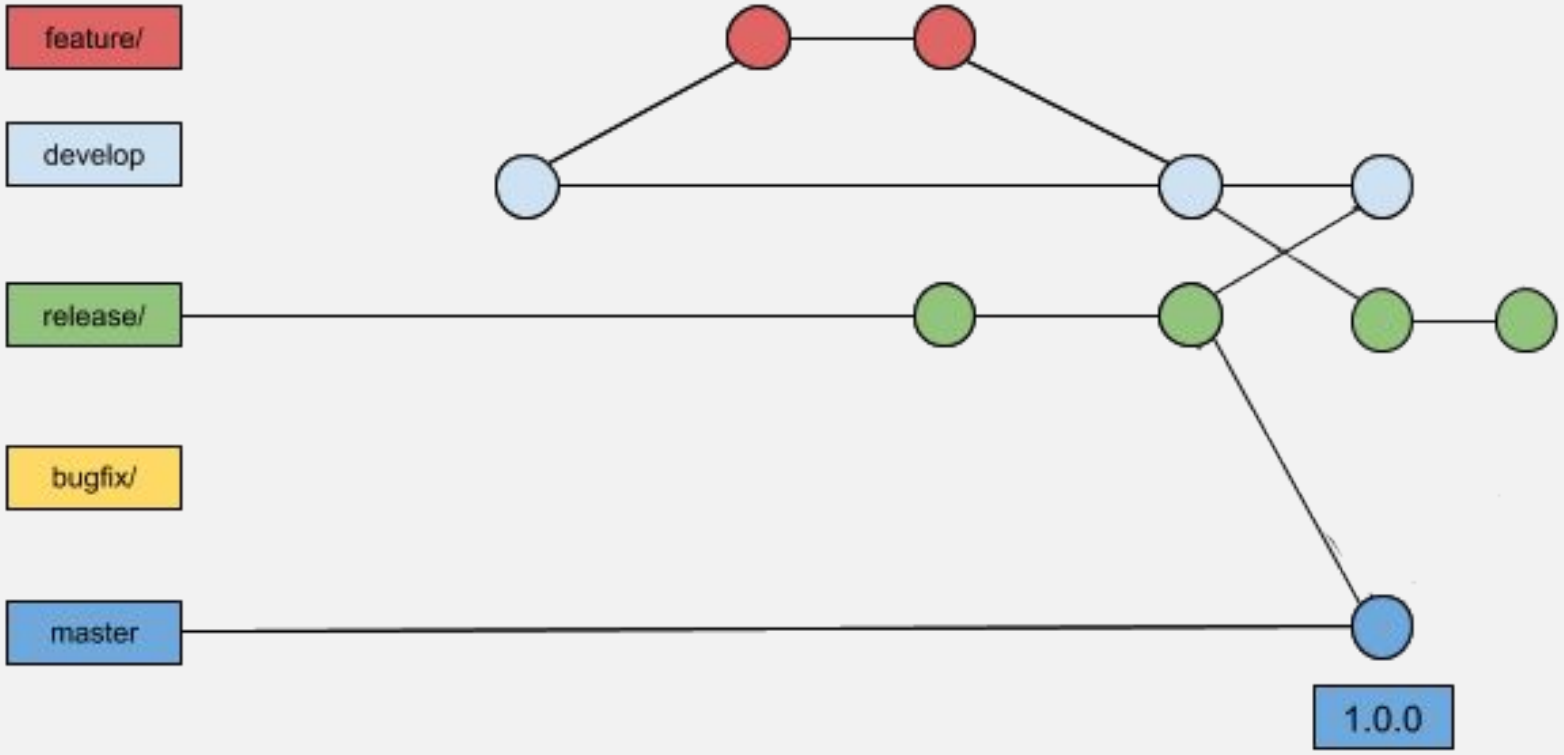
GITFLOW
HOTFIX BRANCHES



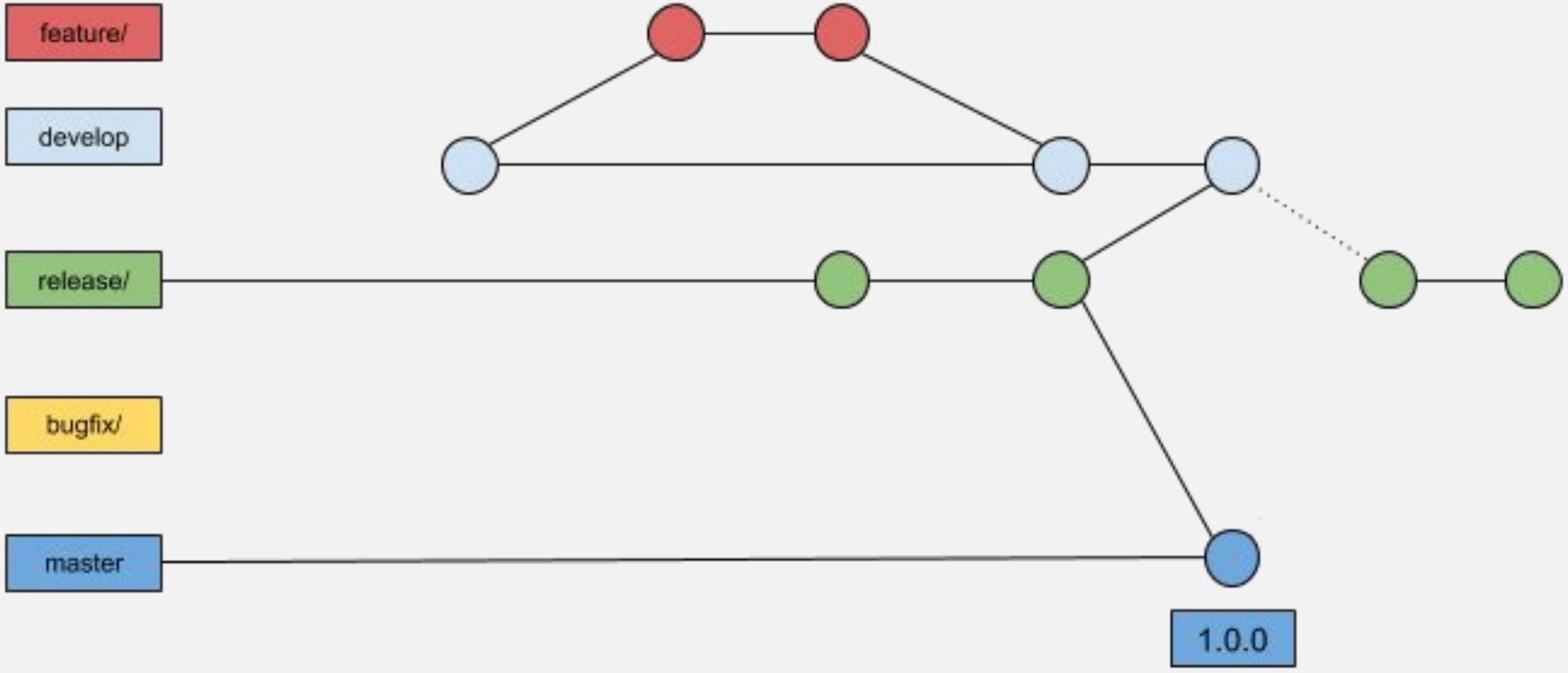
GITFLOW
PARALLELE RELEASE BRANCHES



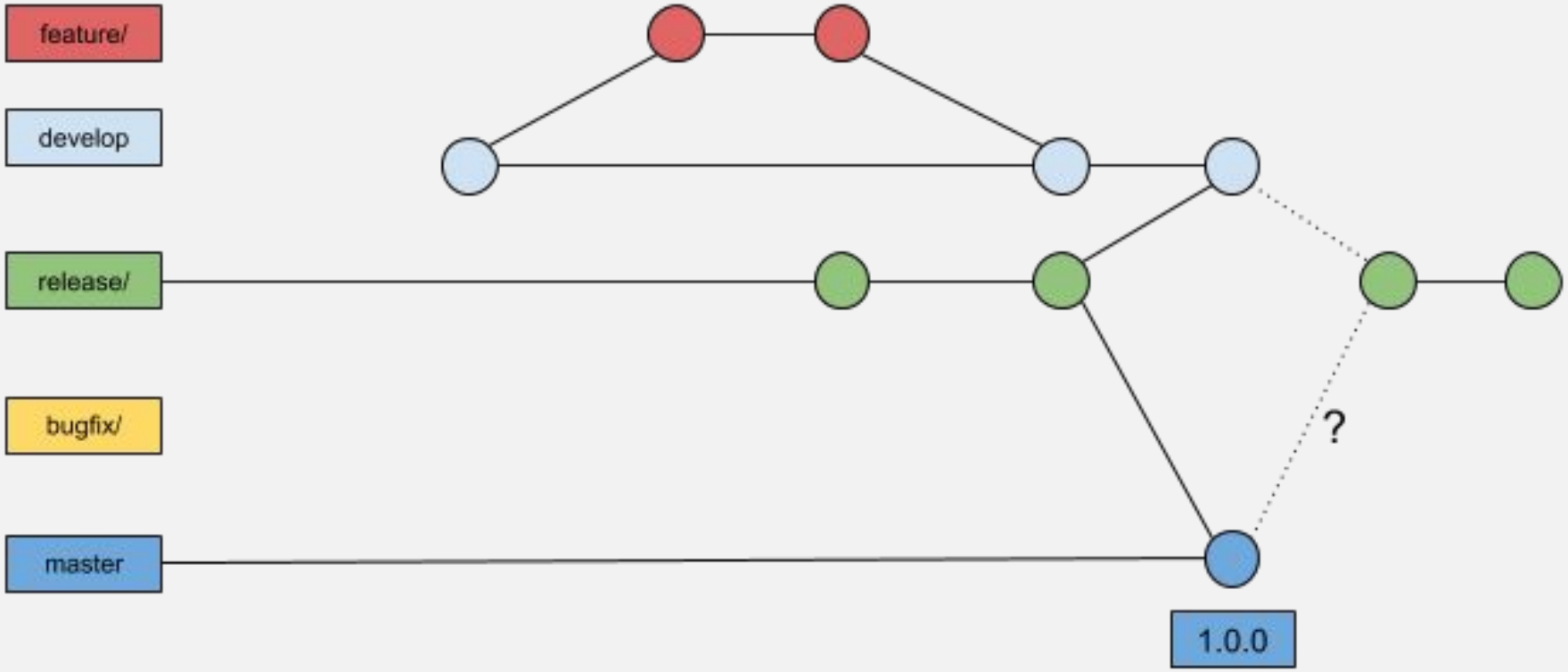
GITFLOW
PARALLELE RELEASE BRANCHES



GITFLOW
PARALLELE RELEASE BRANCHES



GITFLOW
PARALLELE RELEASE BRANCHES



GITFLOW ZUSAMMENFASSUNG

Vorteile:

- klar strukturiertes Branchingmodell
- hohe Unterstützung durch Plugins für Build-Tools, Build-Server, Code-Hosting-Services etc.
- gut nachvollziehbare Git Historie (wenn auch aufgebläht)

Nachteile:

- Redundanzen beim Integrieren von Release- & Hotfix Branches
- getaggte Version entspricht nie dem Stand auf master
- keine Unterstützung von mehreren Releases

RELEASE BRANCHES

- ein öffentlicher Hauptzweig
- bei Release wird ein öffentlicher Branch mit der jeweiligen Release-Nummer erstellt
- Änderungen am Code werden in den Release-Branch cherry-picked
- Release-Branches werden nie in den Haupt-Branch zurückgemergt
- Cactus Model[4], Stable Mainline[5]

[4] https://barro.github.io/2016/02/a-succesful-git-branching-model-considered-harmful/
[5] http://www.bitsnbites.eu/a-stable-mainline-branching-model-for-git/

CACTUS MODEL
FEATURE ENTWICKLUNG

CACTUS MODEL
FEATURE ENTWICKLUNG

feature/

master

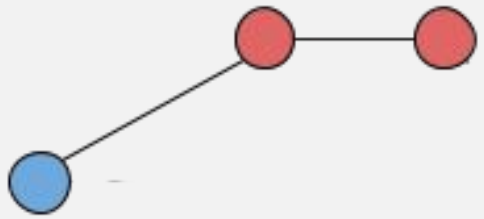
release/

CACTUS MODEL
FEATURE ENTWICKLUNG

feature/

master

release/

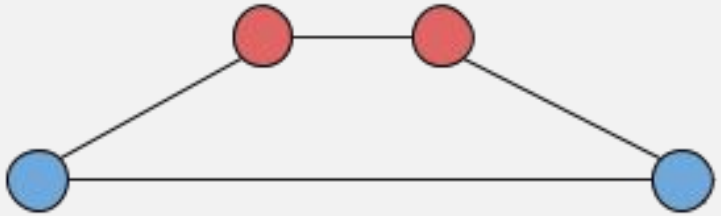


CACTUS MODEL
FEATURE ENTWICKLUNG

feature/

master

release/

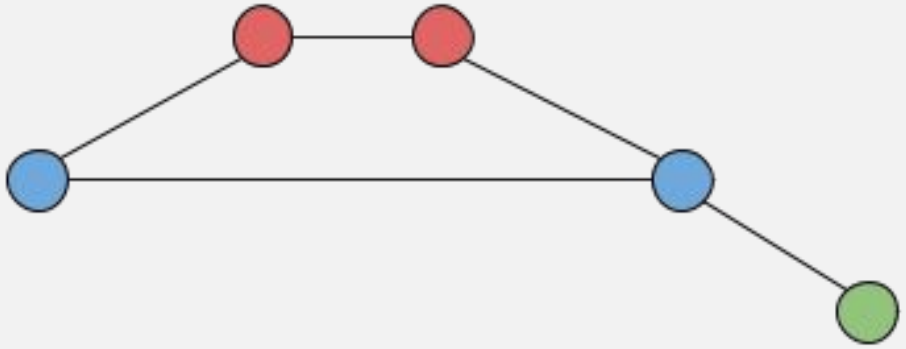


CACTUS MODEL
RELEASE

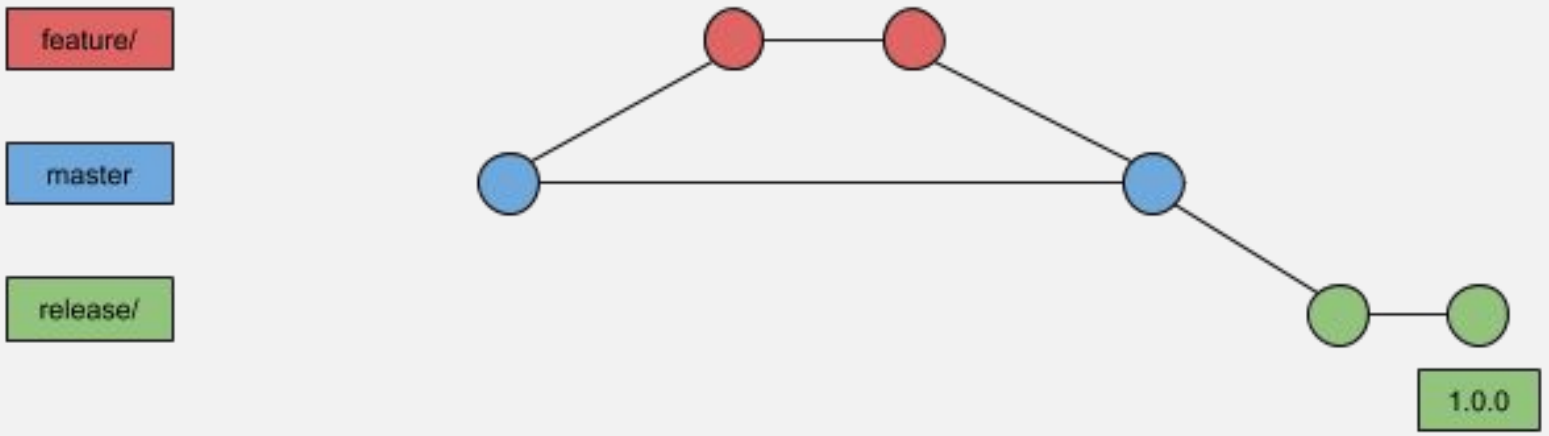
feature/

master

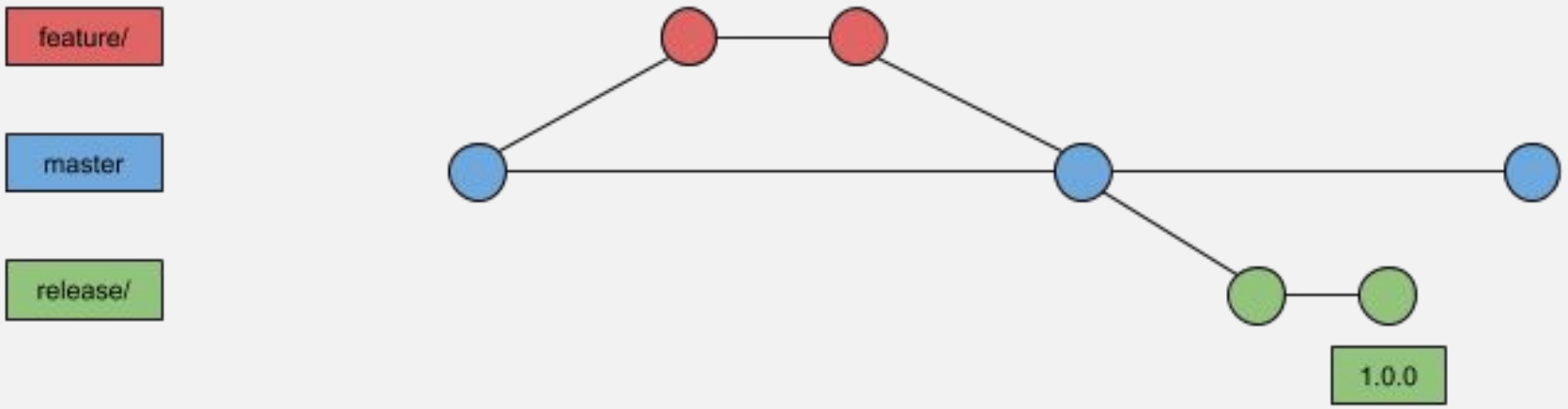
release/



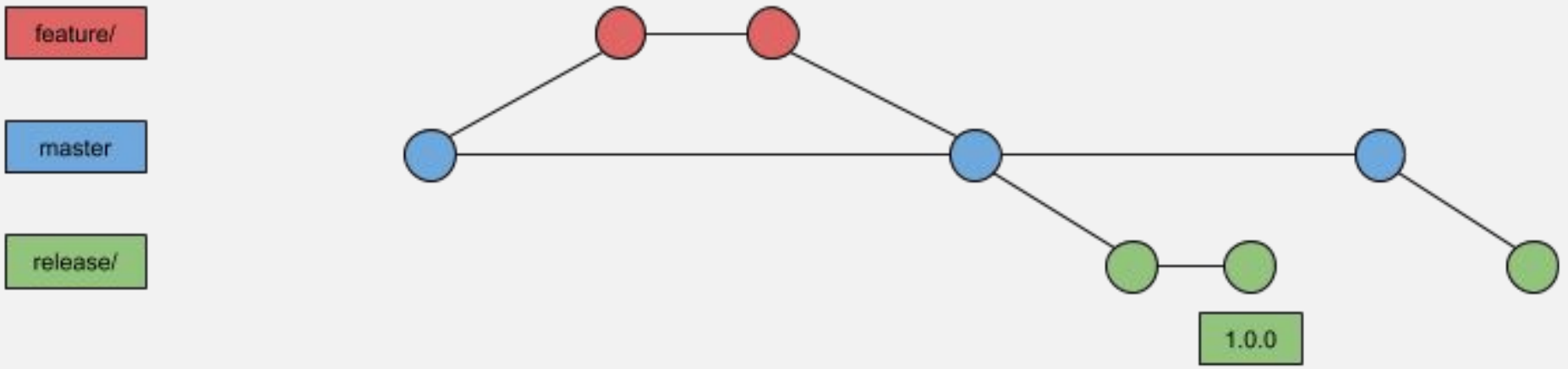
CACTUS MODEL
RELEASE



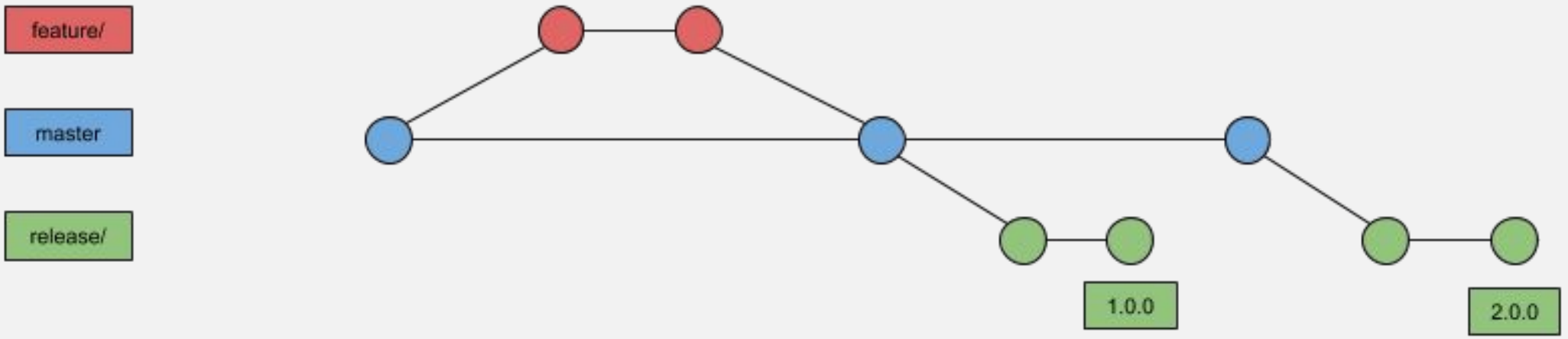
CACTUS MODEL
RELEASE



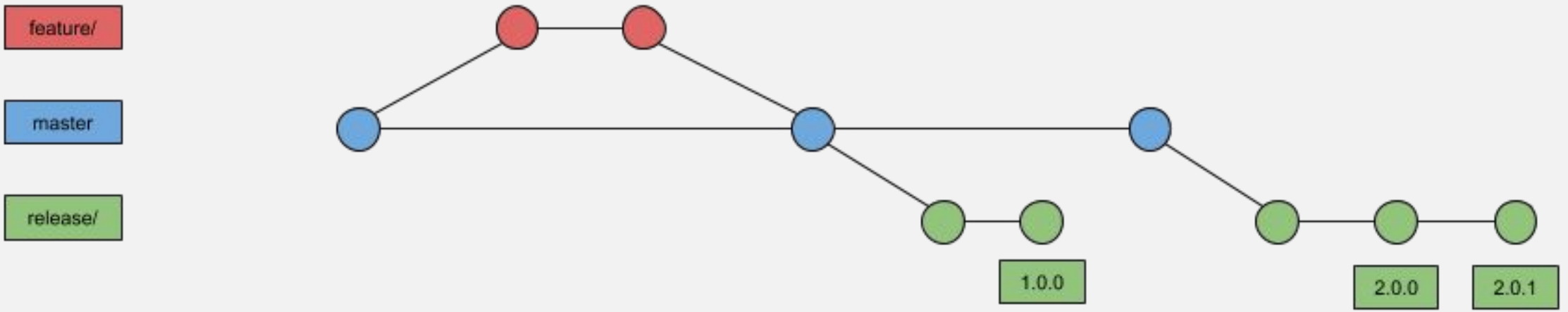
CACTUS MODEL
RELEASE



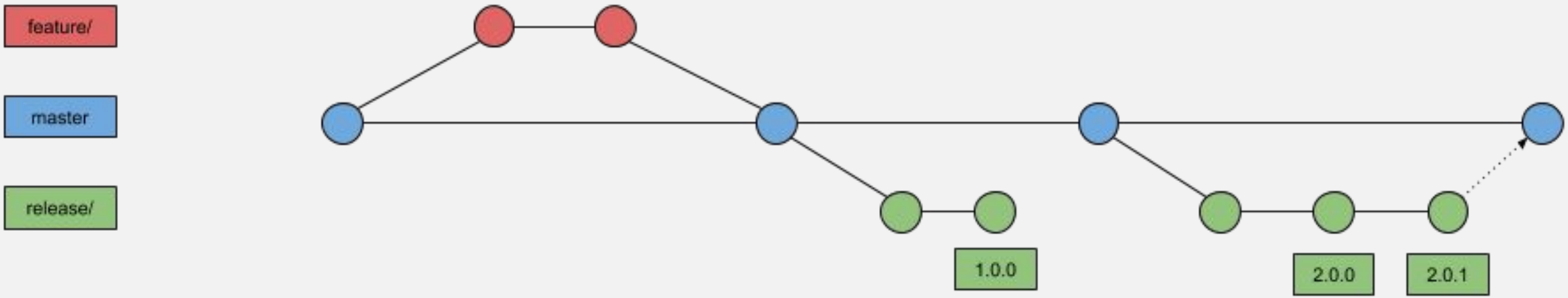
CACTUS MODEL
RELEASE BRANCHES



CACTUS MODEL
HOTFIX



CACTUS MODEL
HOTFIX



CACTUS MODEL ZUSAMMENFASSUNG

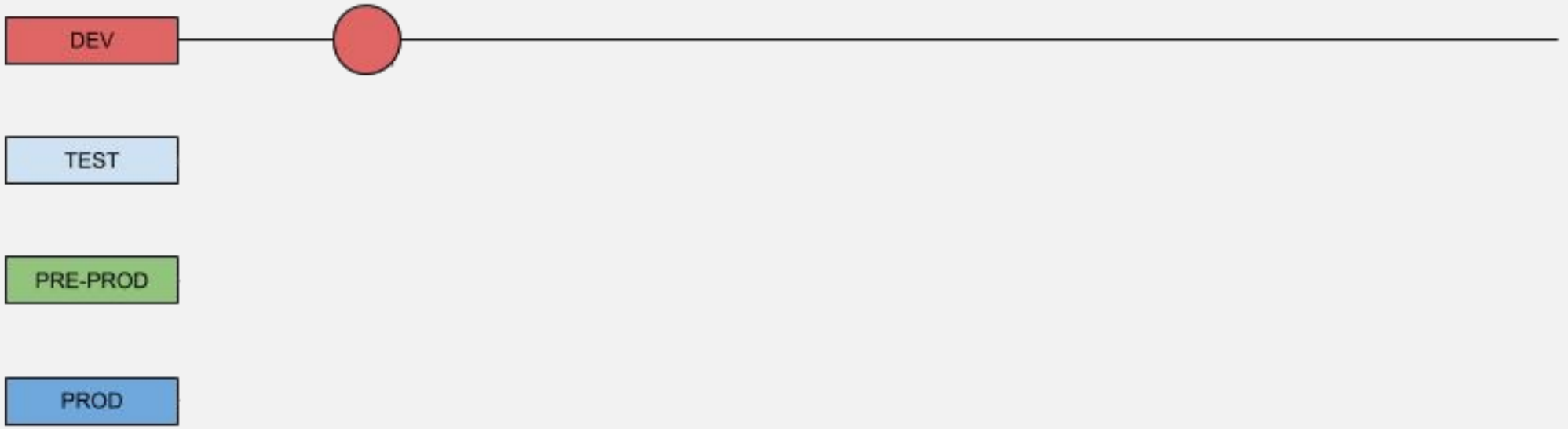
Vorteile:

- Branching Modell mit eindeutiger Aufgabe
- Unterstützung mehrerer Release Branches in parallel

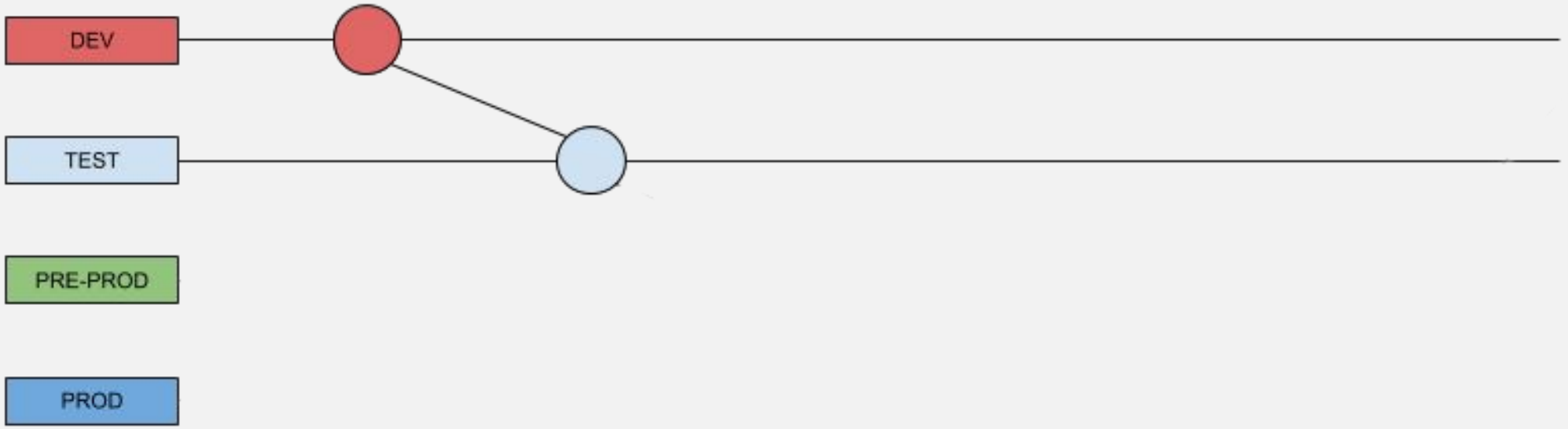
Nachteile:

- Cherry Picking notwendig
- Erzeugen doppelter Commits

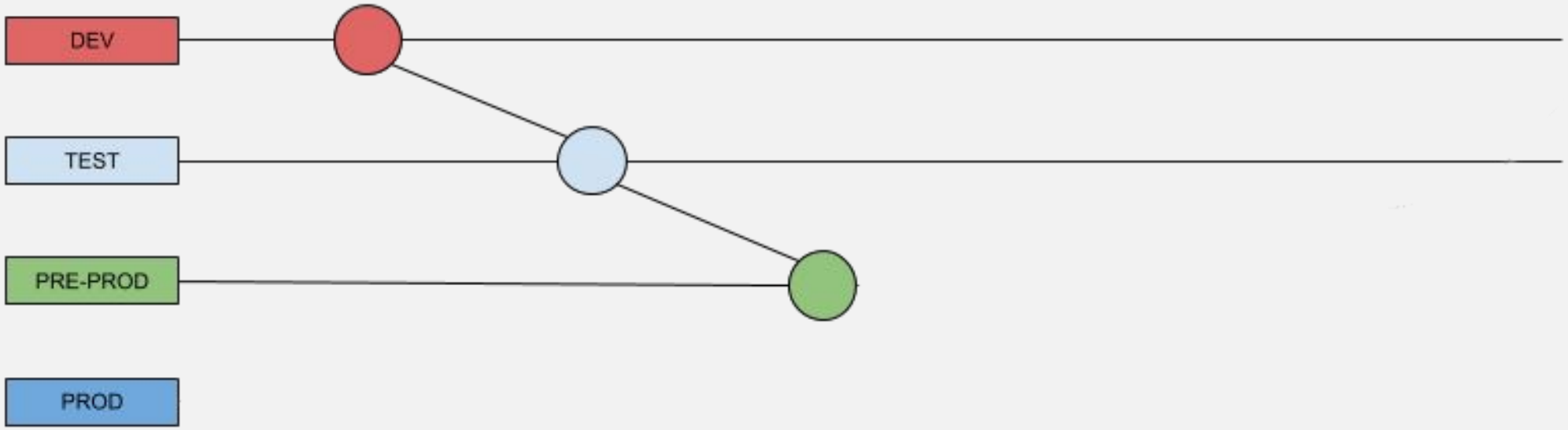
ENVIRONMENT BRANCHES
FEATURE ENTWICKLUNG



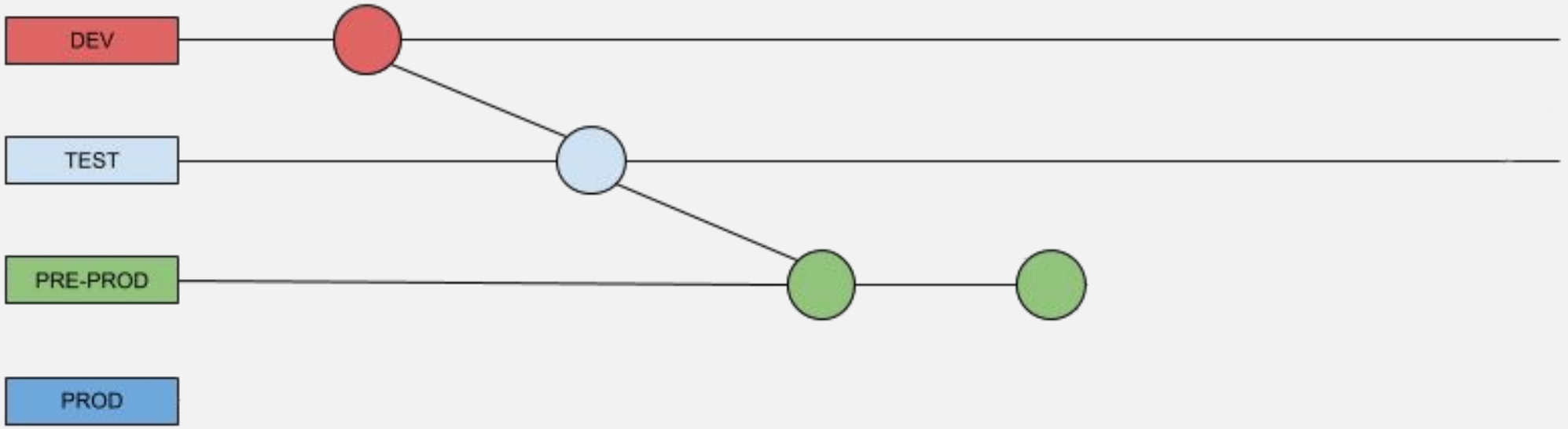
ENVIRONMENT BRANCHES
FEATURE TEST



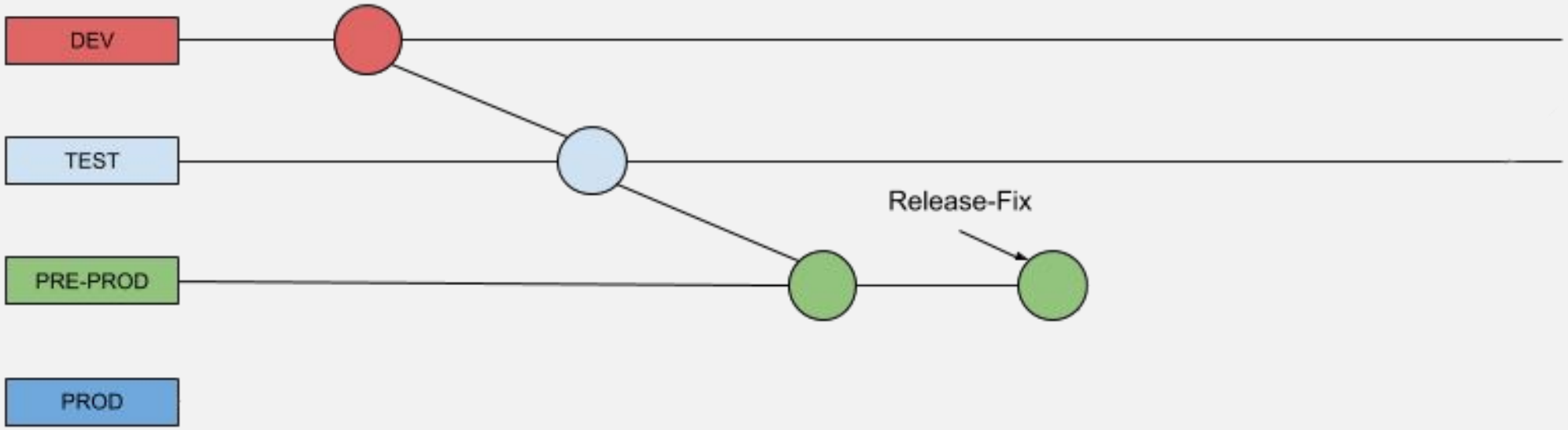
ENVIRONMENT BRANCHES
RELEASE VORBEREITUNG



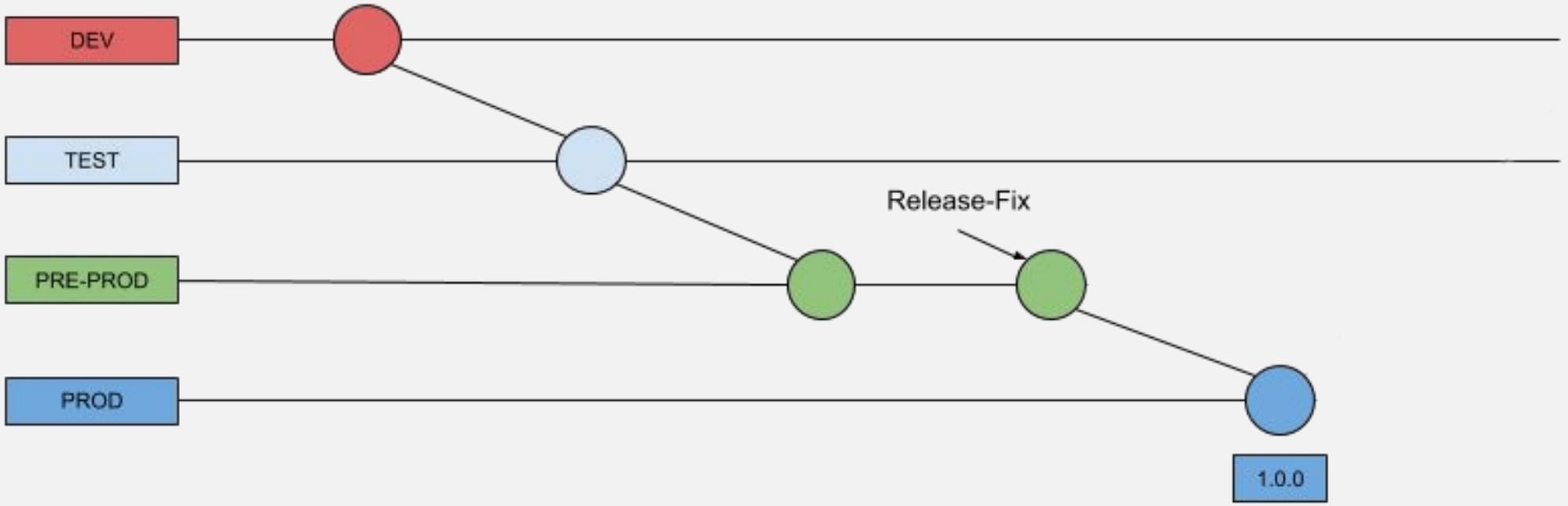
ENVIRONMENT BRANCHES
RELEASE FIX



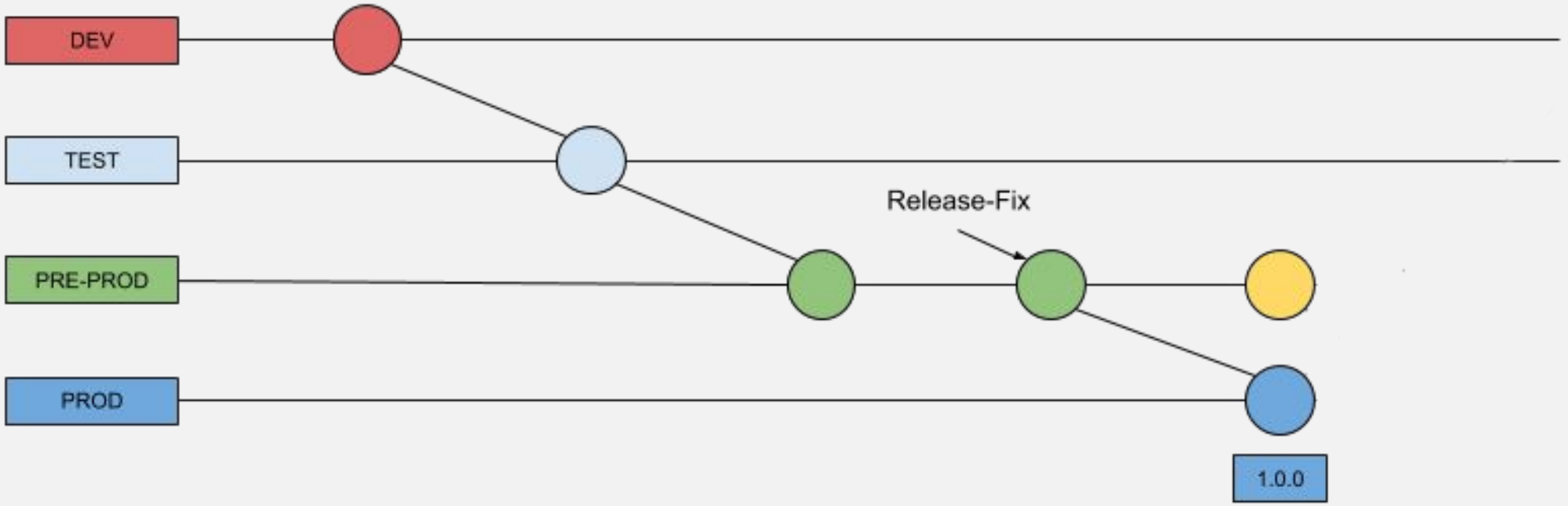
ENVIRONMENT BRANCHES
RELEASE FIX



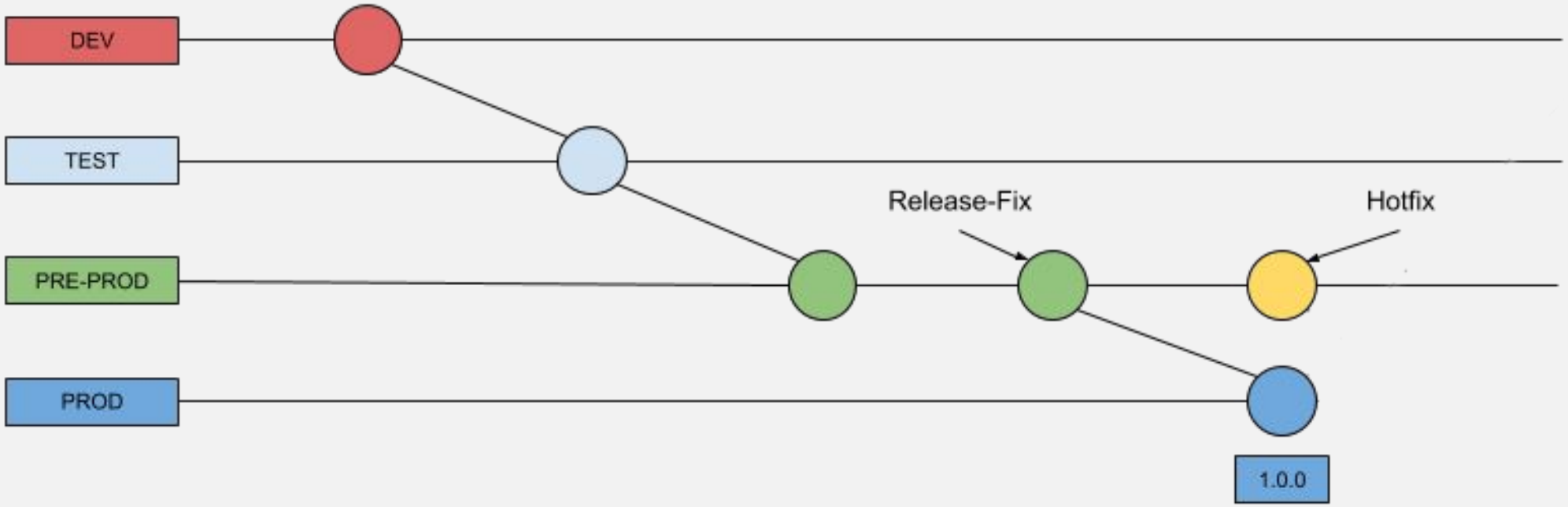
ENVIRONMENT BRANCHES
RELEASE



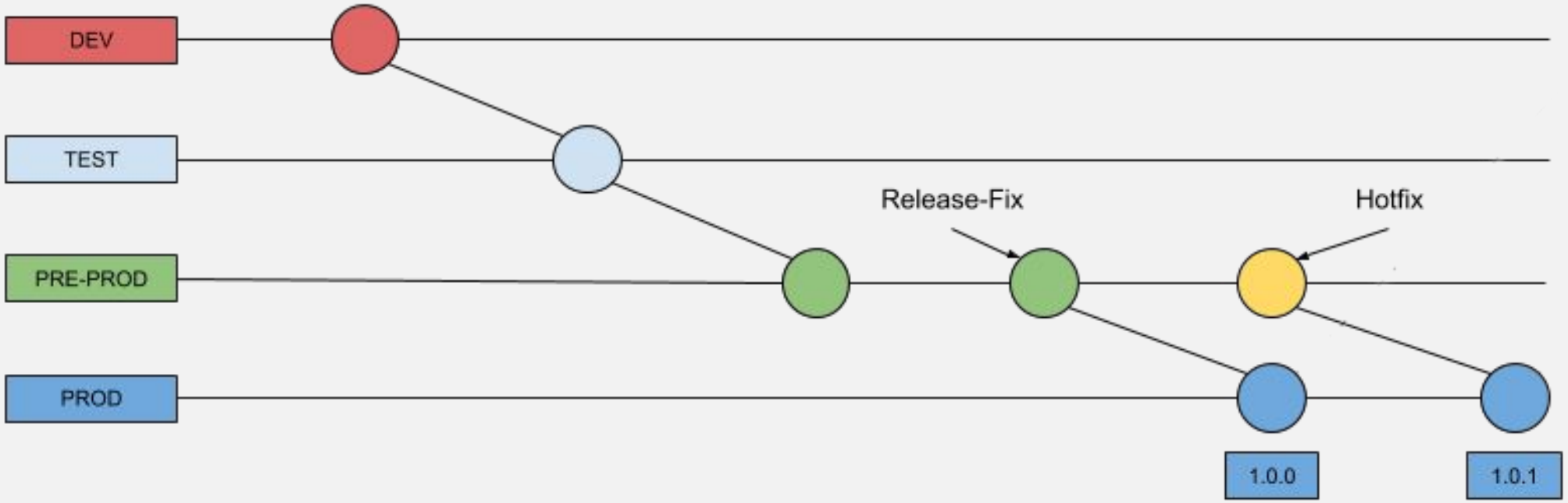
ENVIRONMENT BRANCHES
HOTFIX



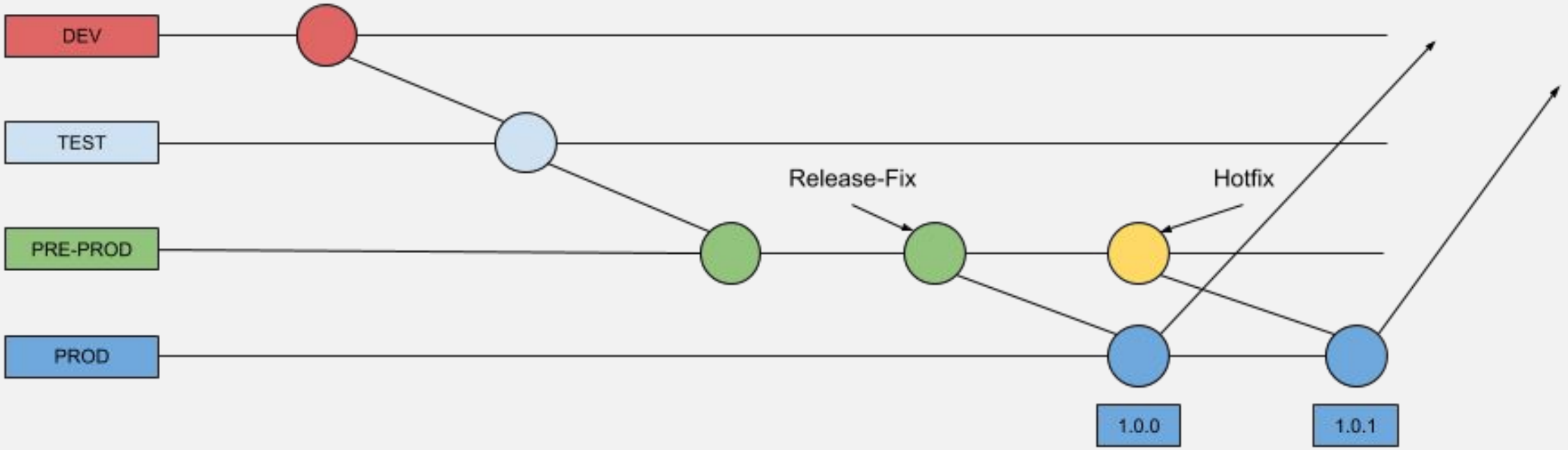
ENVIRONMENT BRANCHES
HOTFIX



ENVIRONMENT BRANCHES
HOTFIX RELEASE



ENVIRONMENT BRANCHES
HOTFIX RELEASE



ENVIRONMENT BRANCHES ZUSAMMENFASSUNG

Vorteile:

- eindeutige Zuordnung von Branch zu Umgebung

Nachteile:

- Viele Merge-Operationen notwendig
- nicht ohne Automatisierung von Merges praktikabel

KRITERIEN

KRITERIEN

- Wahl des Versionsverwaltungssystems

WAHL DES VERSIONSVERWALTUNGSSYSTEMS

- verteilte Versionsverwaltungssysteme werden bevorzugt
- verteilte Workflows
- selbe Leistungsfähigkeit, wie CVCS, jedoch
  - lokales Arbeiten mit Branches möglich
  - Drei-Wege-Merges
  - Pull Requests
  - Rebasing
- Git gilt heutzutage als Quasi-Standard
- Open-Source-Software
- die meisten Branching-Modelle sind für Git konzipiert
- Infrastruktur, wie Code-Hosting-Services unterstützen hauptsächlich Git

KRITERIEN

- Wahl des Versionsverwaltungssystems
- Continuous Integration VS Feature Branches

CONTINUOUS INTEGRATION VS FEATURE BRANCHES

Feature Toggle

- Production Readiness eines Features wird durch Variable ausgedrückt
- Frameworks, wie Togglz[1] können zur Unterstützung verwendet werden
- Vorteile:
  - Feature kann vorab „ausprobiert“ werden
  - bei Release ist kein Deployment notwendig
- Nachteile:
  - Konfiguration im Code
  - Aufwand durch Entfernen des Toggles

```
private static boolean featureXEnabled = false;

public void useFeatureX(){
    ...

    if (featureXEnabled) {
        featureX();
    } else {
        doSomethingElse();
    }

    ...

    }

public void featureX(){
    // TODO: feature implementation
}
```

[1] https://www.togglz.org/

CONTINUOUS INTEGRATION VS FEATURE BRANCHES

Feature Toggle

- Production Readiness eines Features wird durch Variable ausgedrückt
- Frameworks, wie Togglz[1] können zur Unterstützung verwendet werden
- Vorteile:
  - Feature kann vorab „ausprobiert“ werden
  - bei Release ist kein Deployment notwendig
- Nachteile:
  - Konfiguration im Code
  - Aufwand durch Entfernen des Toggles

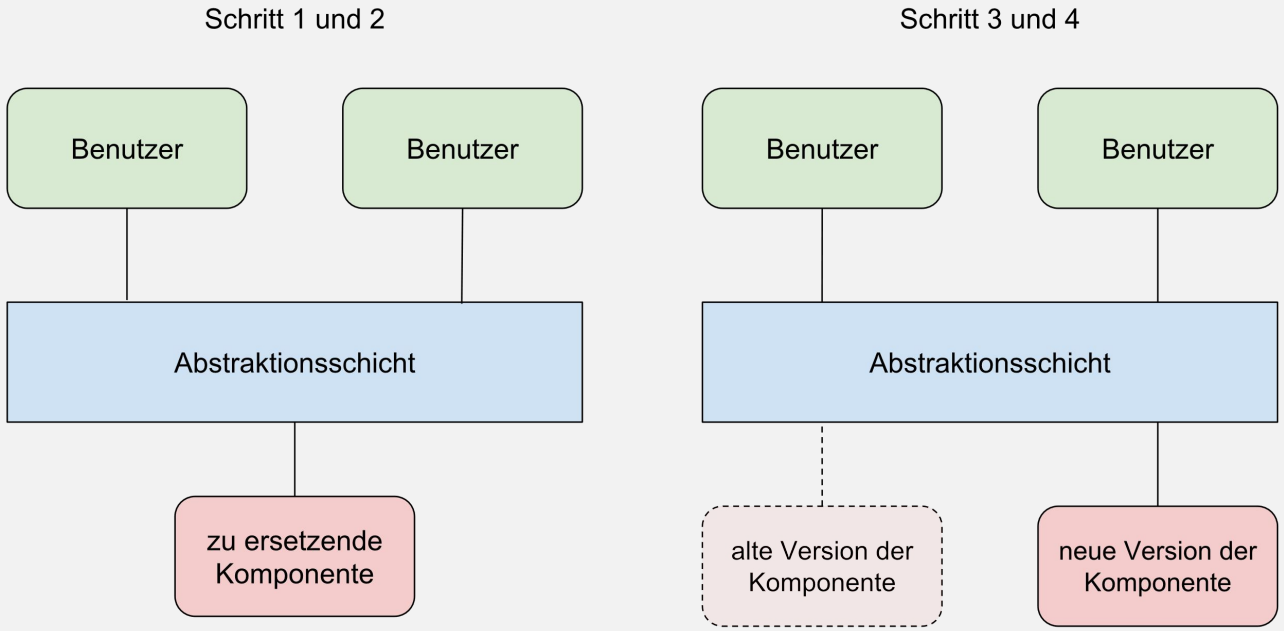
[1] https://www.togglz.org/

```
public enum MyFeatures implements Feature {

    @Label("First Feature")
    FEATURE_ONE,

    @Label("Second Feature")
    FEATURE_TWO;

    public boolean isActive() {
        return FeatureContext.getFeatureManager().isActive(this);
    }

}

public void someBusinessMethod() {

  if( MyFeatures.FEATURE_ONE.isActive() ) {
    // do new exciting stuff here
  }

  [...]

}
```

CONTINUOUS INTEGRATION VS FEATURE BRANCHES

Branch By Abstraction

1. Schritt:
   Anlegen einer Abstraktionsschicht
2. Schritt:
   Bestehendes System refaktorisieren
3. Schritt:
   Implementierung der neuen Komponente
4. Schritt:
   Löschen der alten Komponente



CONTINUOUS INTEGRATION VS FEATURE BRANCHES

Einen CI-Server zu verwenden, heißt *nicht*, dass **Continuous Integration** praktiziert wird.

- Anwenden von Schlüsselpraktiken, wie z.B.:
  - Builds automatisieren
  - tägliches Committen auf den Haupt-Zweig
  - Isolation von Features mittels Feature-Toggle oder Branch-By-Abstraction

Praktiken und Antipatterns, bei deren Beachtung eine Kombination der Vorteile von Continuous Integration und Feature-Branches möglich ist:

- CI-Builds in der Mainline dürfen nicht fehlschlagen
- Rebases regelmäßig durchführen
- Große Änderungen kommunizieren

KRITERIEN

- Wahl des Versionsverwaltungssystems
- Continuous Integration VS Feature Branches
- Beschaffenheit des Teams

KRITERIEN

- Wahl des Versionsverwaltungssystems
- Continuous Integration VS Feature Branches
- Beschaffenheit des Teams

KRITERIEN

- Wahl des Versionsverwaltungssystems
- Continuous Integration VS Feature Branches
- Beschaffenheit des Teams
  - Anzahl der Teammitglieder

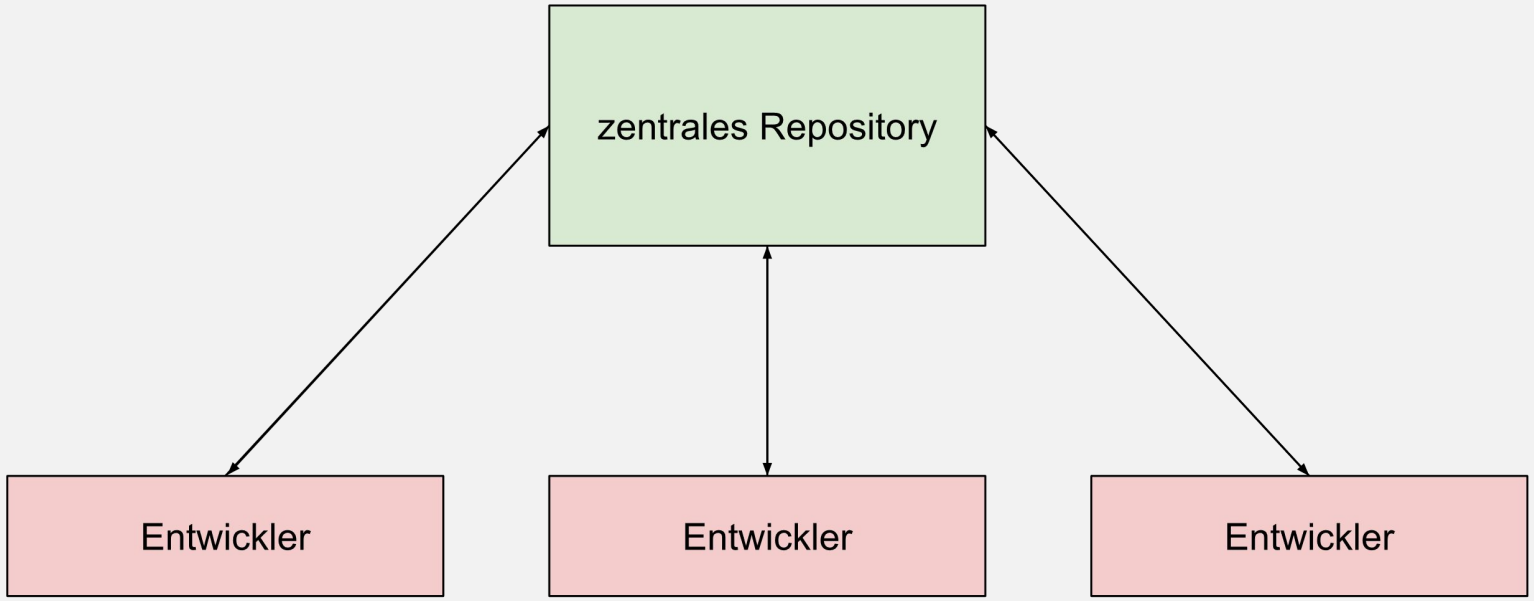
KRITERIEN

- Wahl des Versionsverwaltungssystems
- Continuous Integration VS Feature Branches
- Beschaffenheit des Teams
  - Anzahl der Teammitglieder
  - Disziplin der Teammitglieder

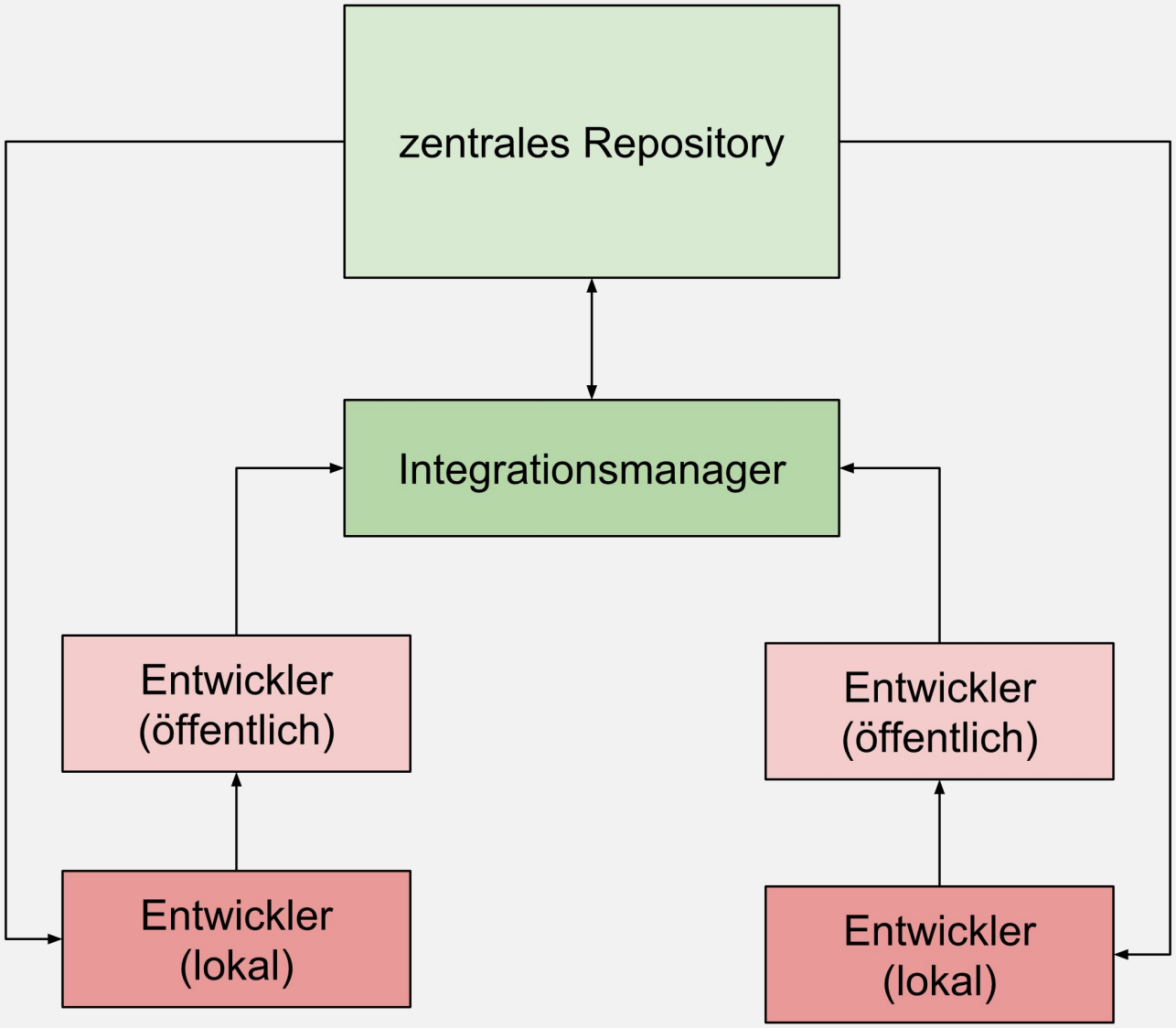
KRITERIEN

- Wahl des Versionsverwaltungssystems
- Continuous Integration VS Feature Branches
- Beschaffenheit des Teams
  - Anzahl der Teammitglieder
  - Disziplin der Teammitglieder
  - Verteilung der Teammitglieder

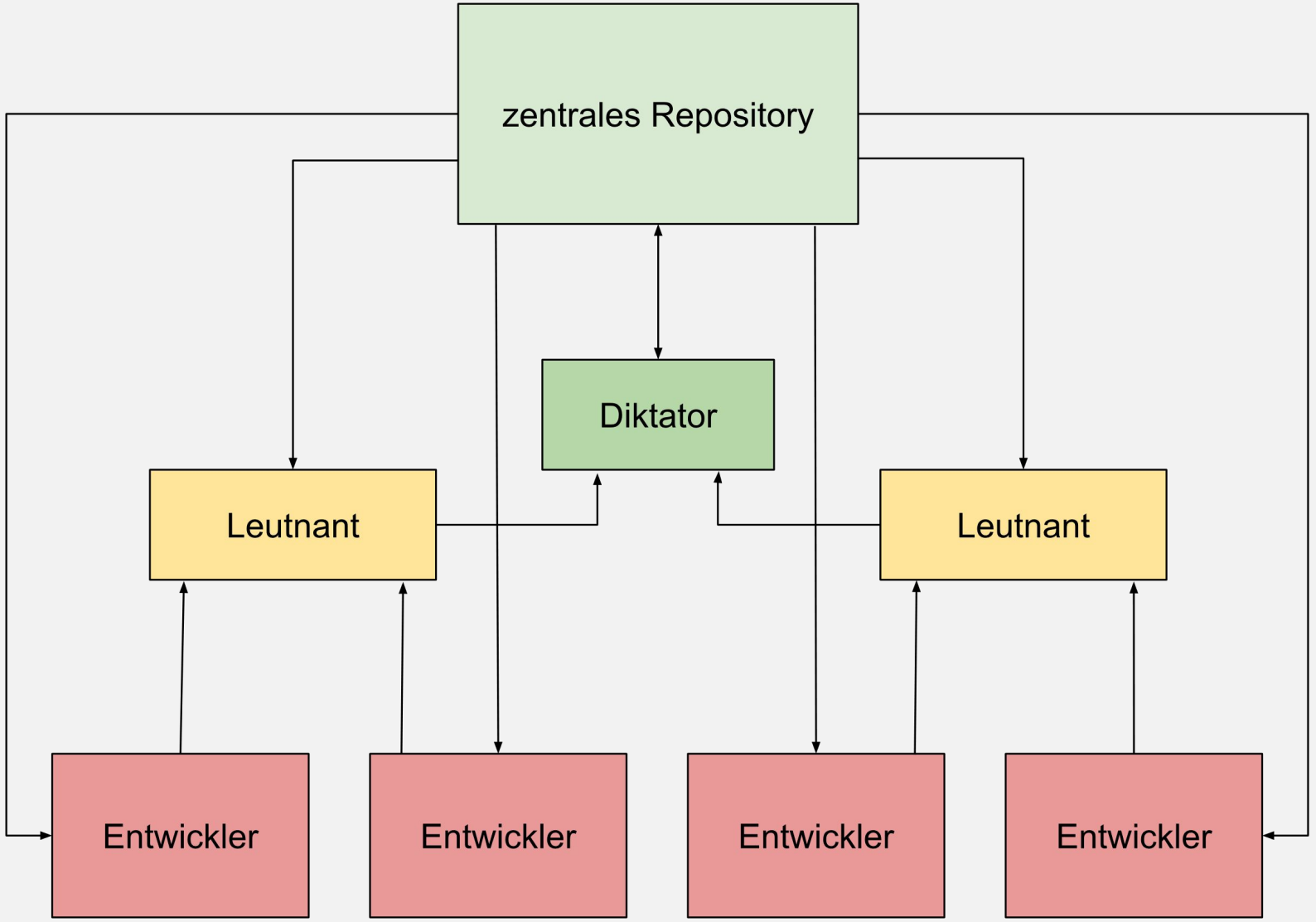
VERTEILUNG VON TEAMMITGLIEDERN ZENTRALISIERTER WORKFLOW



VERTEILUNG VON TEAMMITGLIEDERN INTEGRATIONSMANAGER WORKFLOW



VERTEILUNG VON TEAMMITGLIEDERN DIKTATOR UND LEUTNANTS WORKFLOW



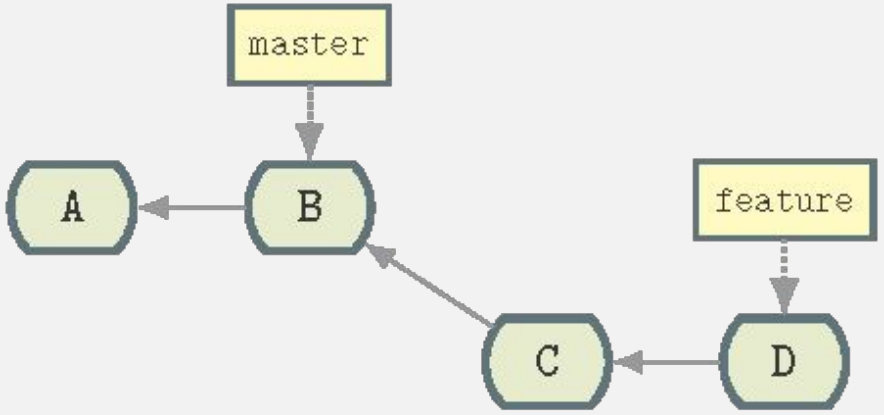
KRITERIEN

- Wahl des Versionsverwaltungssystems
- Continuous Integration VS Feature Branches
- Beschaffenheit des Teams
  - Anzahl der Teammitglieder
  - Disziplin der Teammitglieder
  - Verteilung der Teammitglieder
- Verwendung eines einzelnen Repository oder Pflege mehrerer Repositories notwendig?

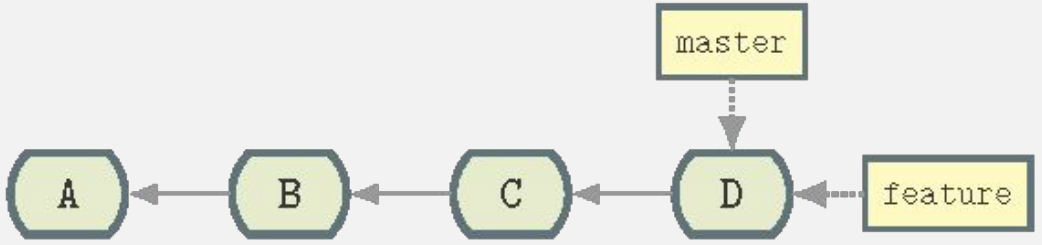
KRITERIEN

- Wahl des Versionsverwaltungssystems
- Continuous Integration VS Feature Branches
- Beschaffenheit des Teams
  - Anzahl der Teammitglieder
  - Disziplin der Teammitglieder
  - Verteilung der Teammitglieder
- Verwendung eines einzelnen Repository oder Pflege mehrerer Repositories notwendig?
- Anforderungen an die Git Historie

ANFORDERUNGEN AN DIE GIT HISTORIE
FAST-FORWARD-MERGE

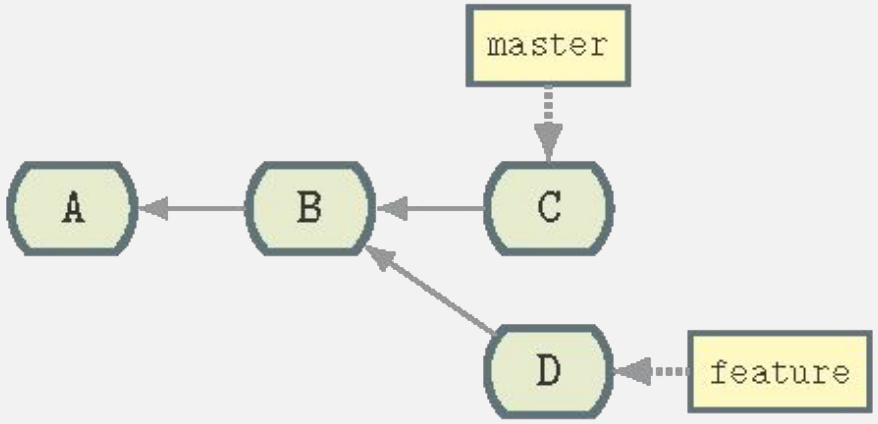


vor dem Merge

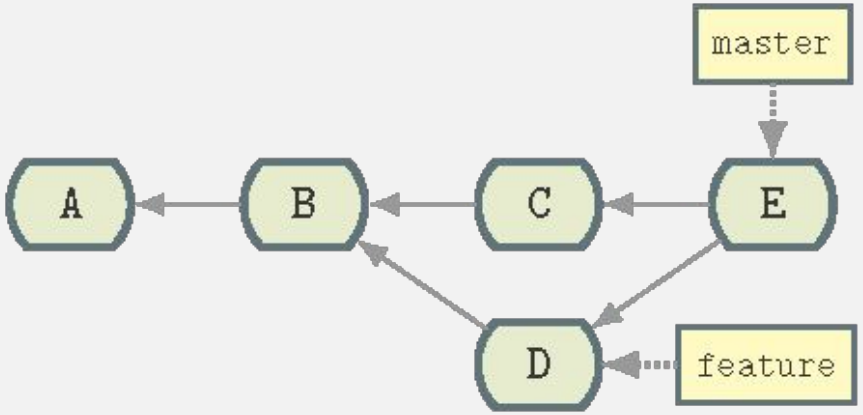


nach dem Merge

ANFORDERUNGEN AN DIE GIT HISTORIE
3-WEGE-MERGE

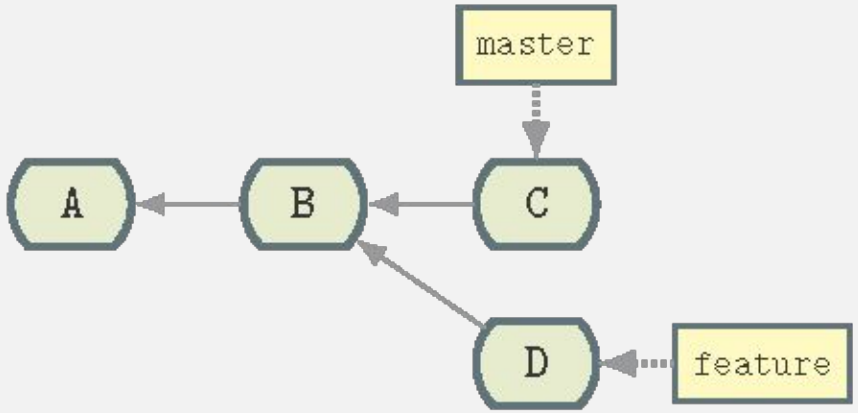


vor dem Merge

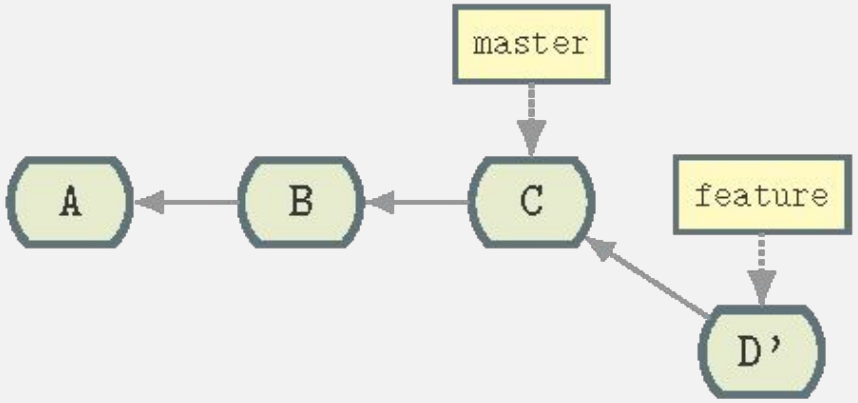


nach dem Merge

ANFORDERUNGEN AN DIE GIT HISTORIE
REBASE



vor dem Rebase



nach dem Rebase

FAZIT

Welche Branching Strategien sollte ich verfolgen?

- Ich möchte ein zentrales Versionsverwaltungssystem verwenden
  - Trunk Based Development
  - (Cactus Model)
- Ich möchte Continuous Integration betreiben
  - Trunk Based Development
  - (Cactus Model)

FAZIT

Welche Branching Strategien sollte ich verfolgen?

- Habe ich mehrere Versionen zu pflegen?
  - Cactus Model
  - (OneFlow)

FAZIT

Welche Branching Strategien sollte ich verfolgen?

- Führe ich regelmäßige formale Releases durch?
  Ja
  - OneFlow
  - GitFlow
  - (Environment Branches)

  Nein
  - GitHub Flow
  - (Trunk Based Development)

FAZIT

Welche Branching Strategien sollte ich verfolgen?

- Möchte ich eine Git Historie, die keine unnötigen oder doppelten Operationen beinhaltet?
  - OneFlow

- Gibt es eine hohe Verteilung der Teammitglieder?
  - Verwendung vom Integrationsmanager-Workflow
- Gibt es eine hohe Verteilung und eine hohe Anzahl der Teammitglieder?
  - Verwendung vom Diktator & Leutnants Workflow

FAZIT

Welche Branching Strategien sollte ich verfolgen?

- Führe ich regelmäßige formale Releases durch?
  Ja
  - OneFlow
  - GitFlow
  - (Environment Branches)

  Nein
  - GitHub Flow
  - (Trunk Based Development)

FRAGEN