BACHELORARBEIT

zur Erlangung des akademischen Grades

„Bachelor of Science in Engineering“ im Studiengang

Informatik/Computer Science

Vergleich verschiedener Workflows im Versionierungssystem Git

Ausgeführt von: Felix Bauer
Personenkennzeichen: 1610257002

1. Begutachter: Dipl.-Ing. (FH) Arthur Michael Zaczek

Stockerau, 26.01.2018

FH University of Applied Sciences
TECHNIKUM WIEN

Eidesstattliche Erklärung

„Ich, als Autor und Urheber der vorliegenden Arbeit, bestätige mit meiner Unterschrift die Kenntnisnahme der einschlägigen urheber- und hochschulrechtlichen Bestimmungen (vgl. Urheberrechtsgesetz idgF sowie Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien idgF).

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig angefertigt und Gedankengut jeglicher Art aus fremden sowie selbst verfassten Quellen zur Gänze zitiert habe. Ich bin mir bei Nachweis fehlender Eigen- und Selbstständigkeit sowie dem Nachweis eines Vorsatzes zur Erschleichung einer positiven Beurteilung dieser Arbeit der Konsequenzen bewusst, die von der Studiengangsleitung ausgesprochen werden können (vgl. Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien idgF).

Weiters bestätige ich, dass ich die vorliegende Arbeit bis dato nicht veröffentlicht und weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt habe. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht.“

Ort, Datum	Unterschrift

Kurzfassung

Alle bekannten Workflows haben ihre Stärken und Schwächen. Während der GitHub Flow gerade für Software-as-a-Service-Produkte gut geeignet ist, hat er große Limitierungen hinsichtlich des Release Managements. Um eine Lösung für diese zu bieten hat GitLab mehrere Workflows geschaffen. Der GitLab Flow mit Production Branch und der mit Environment Branches sind besonders gut geeignet, um eine stärkere Kontrolle über den Zeitpunkt der Releases zu erhalten. Der GitLab Flow mit Release Branches ermöglicht das simultane warten mehrerer aktiver Releases. Die Stärke von Git-Flow hingegen liegt in großen Projekten, bei denen kurz vor dem Release letzte Änderungen vorgenommen und Metadaten angepasst werden müssen.

Schlagwörter: Git, Workflow, Release Management, Continuous Integration, Continuous Delivery

Abstract

All popular workflows have their strengths and weaknesses. While GitHub Flow is especially well-suited for Software as a service projects, it has major limitations in terms of release management. To provide a solution for this, GitLab has created several workflows. The GitLab Flow with production branch or environment branches are particularly well-suited for greater control over the timing of releases. The GitLab Flow with release branches enables the simultaneous maintenance of multiple active releases. The strength of Git-Flow, on the other hand, lies in large projects, where final changes have to be made and metadata has to be adjusted shortly before release.

Keywords: Git, Workflow, Release Management, Continuous Integration, Continuous Delivery

Inhaltsverzeichnis

1 Einleitung

1.1 Problemstellung

Die Auswahl des richtigen Workflows ist eine grundlegende Aufgabe, die zu Beginn eines Projekts getroffen werden muss, aber die Vielzahl der Möglichkeiten erschweren dies. Bei der Evaluierung eines Workflows müssen einige Kriterien beachtet werden. [1] Es ist allerdings schwierig, eine Übersicht über verschiedene Workflows und deren Vor- und Nachteile zu finden, sodass nicht immer der optimale Workflow ausgewählt wird.

1.2 Zielsetzung

Das Ziel dieser Bachelorarbeit ist es, einen Überblick über die gängigsten Workflows zu verschaffen. Des Weiteren sollen Kriterien aufgezeigt werden, wonach die Eignung der Workflows bewertet werden kann sowie festgestellt werden kann, welcher Workflow für welche Art von Projekten am besten geeignet ist.
Ausgehend von der Problemstellung und der Zielsetzung kann folgende Forschungsfrage formuliert werden:

> Was sind die populärsten Git Workflows, was sind ihre Vor- und Nachteile und für welche Projekte sind sie am besten geeignet?

1.3 Aufbau der Arbeit

Um die Workflows vergleichen zu können, werden zunächst in Kapitel 2 Kriterien aufgezählt, nach denen Workflows bewertet werden können. Nur durch das unabhängige und unvoreingenommene Sammeln von Kriterien können die Workflows objektiv miteinander verglichen werden.
Anschließend werden in Kapitel 3 die bekanntesten Workflows vorgestellt und nach den zuvor gesammelten Kriterien bewertet.
In der Conclusio werden die gefundenen Erkenntnisse zusammengefasst sowie eine Übersicht geboten, worin die Vorteile der einzelnen Workflows liegen. Abschließend wird eine Empfehlung abgegeben, welcher Workflow für welche Art von Projekten am besten geeignet ist.

2 Kriterien

2.1 Continuous Integration

CI (Continuous Integration) beschreibt die Praxis, Änderungen im Code so schnell und oft wie möglich zu testen, um die Funktion und Integration regelmäßig sicher zu stellen. [2, Kap. What is continuous integration?] Das passiert dadurch, dass bei jeder Änderung des Quellcodes die Software automatisch gebuildet und getestet wird. [3] Die Art und Weise, in der das geschieht, ist allerdings je nach Branching Modell unterschiedlich [4] und verschiedene Workflows sind dafür unterschiedlich gut geeignet. [5, Kap. Einleitung]

2.2 Continuous Delivery

CD (Continuous Delivery) beschreibt die Möglichkeit, Änderungen der Software wie neue Funktionen und Fehlerbehebungen schnell und sicher in die Produktionsumgebung zu migrieren. [6, Kap. What is Continuous Delivery?] Um CD praktizieren zu können, muss zunächst CI implementiert werden. CD ist also eine Erweiterung zu CI [7, Abs. 8], [8, Kap. Conclusion: How to Get Started with CI/CD]. Dabei ist zu beachten, dass CD nicht bedeutet, dass jede Änderung automatisch ausgeliefert wird, sondern nur, dass jederzeit die Möglichkeit besteht, aktuelle Änderungen auszuliefern. [9, Kap. Continuous Integration, Delivery and Deployment]

Eine weitere Möglichkeit für CD ist es, regelmäßig auf Staging- oder QA-Umgebungen (Quality Assurance) auszurollen, um dort stets die aktuelle Version der Software zu testen bzw. Acceptance Tests durchzuführen. [10, Kap. #6: Deploy to staging], [11, Abs. 3]

Der Prozess des Automatisierten CI und CD wird häufig als CI/CD-Pipeline bezeichnet. [12]

2.3 Skalierbarkeit

Ein grundlegender Faktor der Skalierbarkeit ist das Eliminieren von Bottlenecks im Allgemeinen. Ein Bottleneck ist eine Stufe eines Prozesses, die das Fortschreiten bzw. Weiterkommen behindert oder erschwert. [13] Es gilt hier also jene Schritte zu identifizieren, die in weiterer Folge den Fortschritt verlangsamen können. Gerade in Kombination mit CI bzw. CD kann es zu Problemen mit der Skalierbarkeit kommen, wenn schneller neue Aufträge in die Pipeline kommen, als diese sie verarbeiten kann.

2.4 Protected Branches

Bestimmte Workflows erlauben oder erfordern es, bestimmte Branches zu schützen, sodass nur bestimmte Benutzer oder Gruppen auf diese Branches committen bzw. mergen können. Das ermöglicht, dass viele Entwickler in einem Repository entwickeln, aber nur eine Person oder Personengruppe letztendlich darüber entscheidet, welche Beiträge in die protected Branches übernommen werden. Bei Open Source Projekten geschieht das in der Regel durch Integration Managers. [14, Kap. Integration-Manager Workflow]

Einige Tools wie GitLab und GitHub bieten die Funktionalität Branches zu protecten an. [15] Auch ohne diese Tools kann diese Funktionalität mittels serverseitiger Hooks zur Verfügung gestellt werden. [16, Kap. Enforcing a User-Based ACL System]

Um entwickeln zu können ist es wichtig, dass Entwickler auch Branches zur Verfügung haben, die nicht protected sind. Nach abgeschlossener Arbeit müssen sie dann einen MR (merge request) oder PR (pull request) erstellen. Der Request muss wiederum von einem Benutzer mit entsprechender Berechtigung bzw. dem Integration Manager angenommen werden. [14, Kap. Integration-Manager Workflow], [15], [17]

2.5 Komplexität

Vor allem, wenn nur wenige Entwickler an einem Projekt arbeiten, kann die zusätzliche Komplexität mehr Aufwand bedeuten. In diesem Fall ist es möglicherweise ratsam, weniger komplexe Workflows zu wählen, um den zusätzlichen Aufwand zu vermeiden.
Außerdem sind unterschiedliche Workflows für neue Entwickler unterschiedlich leicht zu erlernen und beherrschen. Gerade wenn Entwickler neu in ein Projekt einsteigen oder als Freelancer für mehrere Projekte arbeiten kann das ein Kriterium darstellen. [17, Kap. Issues with git-flow]

2.6 Mehrere Environments / aktive Releases

Für manche Projekte ist es notwendig, mehrere aktive Releases zu warten. Ein Beispiel dafür ist PHP, dessen aktuelle Version 7.2 ist. Aktiven Support gibt es allerdings auch für Version 7.1 und Sicherheitsupdates gibt es zusätzlich für die Versionen 7.0 sowie 5.6. [18]
Ein Beispiel für Software, die in mehreren Umgebungen (in verschiedenen Versionen) ausgerollt ist, ist das Spiel League of Legends. Zusätzlich zur Produktionsumgebung gibt es dort eine öffentliche Beta-Umgebung (PBE). Auf dieser werden neue Funktionen getestet bevor sie in der Produktionsumgebung ausgerollt werden. [19]

3 Workflows

3.1 GitHub

Der GitHub Flow sieht vor, dass für neue Features neue Branches erstellt werden. Diese gehen stets vom Master Branch aus und sollten beschreibende Namen tragen. Auf diesem Branch steht es einem nun frei zu arbeiten und zu committen. Ein PR kann im GitHub Flow jederzeit geöffnet werden. So ist es auch möglich, einen PR zu öffnen, wenn man noch keinen oder wenig Code geschrieben hat aber beispielsweise Ideen ins Projekt einbringen möchte. Eine andere Möglichkeit ist, PRs erst zu öffnen, wenn man mit seiner Arbeit fertig ist und der Code bereit ist reviewed zu werden. Im Anschluss startet eine Diskussion zu den Änderungen, man erhält Feedback und kann weiterhin arbeiten, committen und pushen. Erst wenn durch ausreichendes Testen überprüft ist, dass die Änderungen keine negative Auswirkung auf die Produktion haben, wird der Branch gemerged und der PR damit geschlossen. Im Idealfall wird sofort ein Deployment durchgeführt. [20], [21, Kap. Github Flow]

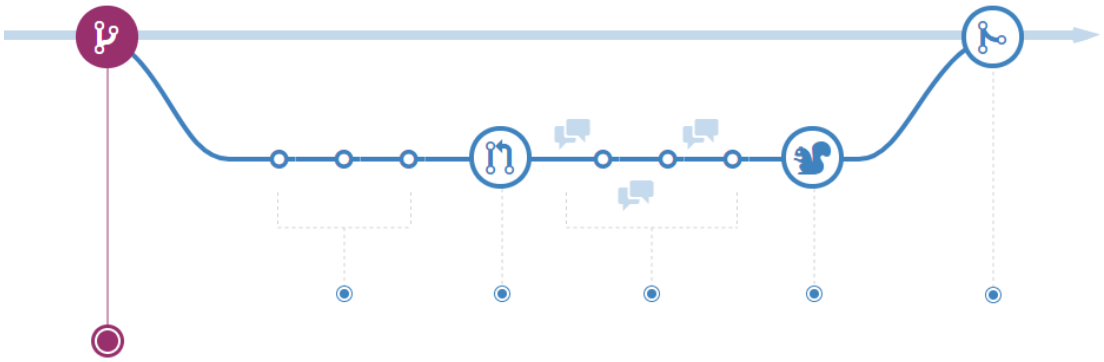

Abbildung 1: GitHub Flow [20]

Continuous Integration / Continuous Delivery

Da der Master Branch zu jeder Zeit eine stabile Version enthalten muss, ist dieser prinzipiell zu jeder Zeit auslieferbar und im Idealfall jederzeit in der Produktionsumgebung ausgerollt.

Skalierbarkeit

Der GitHub Flow sieht vor, dass alle Features, die in den Master Branch gemerged werden, sofort automatisiert deployed werden. Sofern die Dauer eines Deployments relativ kurz ist, stellt dies kein Problem dar. In Projekten, in denen ein Deployment allerdings viel Zeit in Anspruch nimmt (z.B. technologiebedingt) und neue Features in kurzen Intervallen in den Master Branch gemerged werden, kann das automatisierte Deployment jedes Merges ein Bottleneck darstellen.

Protected Branches

Da der GitHub Flow lediglich den Master Branch als allgemeinen, stabilen Branch ansieht, ist es ratsam diesen zu schützen. GitHub bietet zu diesem Zweck ein Berechtigungssystem, dass nur bestimmten Benutzern erlaubt auf diesen Branch zu committe. [15] Für Open Source Projekte, die den GitHub Flow mit einem Fork & Pull Modell arbeiten, sind PRs die einzige Möglichkeit für Außenstehende Code in das öffentliche Repository einzupflegen. [22, Kap. Einleitung]

Komplexität

Der GitHub Flow zielt darauf ab so einfach wie möglich zu sein. [17, Kap. Issues with git-flow] Aufgrund der wenigen Branches ist dieser Workflow sehr simpel und daher äußerst beliebt und einfach zu erlernen. [21]

Mehrere Environments / aktive Releases

Der GitHub Flow kennt lediglich einen stabilen Branch, weshalb es mit diesem Workflow nicht möglich ist, verschiedene Versionen gleichzeitig ausgerollt zu haben.

3.2 GitLab

3.2.1 Production Branch

GitLab hat den GitHub Flow um einen Production Branch erweitert (siehe Abbildung 2). Dieser ist besonders für Applikationen nützlich, die nicht als Saas (Software as a Service) betrieben werden und man nur eingeschränkte Kontrolle über die Releases hat. Beispiele

dafür sind iOS Applikationen, die erst von Apple abgenommen werden müssen [23, Kap. Einleitung] oder Software die nur in bestimmten Zeitfenstern released werden kann. Die Idee hinter dem Workflow ist, dass immer, wenn ein Release durchgeführt wurde, der Master Branch mit dem Production Branch gemerged wird und am Production Branch daher stets die Version zu finden ist, die gerade in der Produktionsumgebung ausgerollt ist. Dennoch sollte der Master Branch jederzeit deploybar sein, da die Zeitpunkte der Releases eventuell nicht exakt planbar sind. [24]

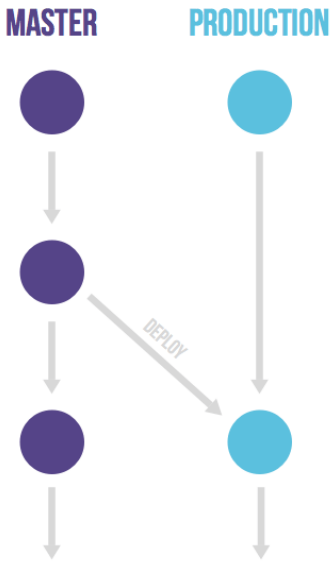

Abbildung 2: GitLab Flow mit Production Branch [24]

Continuous Integration / Continuous Delivery

Dieser Workflow ist explizit für Szenarien entwickelt worden, bei denen Deployments nicht automatisierbar sind oder nicht regelmäßig durchgeführt werden können.

Skalierbarkeit

Dadurch, dass bei diesem Workflow nicht jeder Merge in den Master Branch deployed wird, entfällt hier das in Kapitel 3.1 vorgestellte Bottleneck, dass Deployments zu Spitzenzeiten zu lange dauern können.

Protected Branches

Ebenso wie beim GitHub Flow sollte auch hier der Master Branch geschützt werden. Besonderen Schutz sollte der Production Branch genießen, da auf diesen nur gemerged werden darf, wenn released wird. Dementsprechend dürfen auch nur die Personen mergen, die auch releasen dürfen. Abgesehen von diesen beiden Branches gibt es zahlreiche Feature Branches, für die kein besonderer Schutz vorgesehen ist.

Komplexität

Durch das Hinzufügen eines weiteren Branches zum GitHub Flow steigt die Komplexität. Da der zusätzliche Branch allerdings nicht für die Entwicklung, sondern lediglich für das Release Management gedacht ist, ist der Komplexitätsunterschied verglichen zum GitHub Flow marginal.

Mehrere Environments / aktive Releases

Durch das Einführen eines Branches für die Produktionsumgebung, ist es möglich, den Master Branch in einer Testumgebung auszurollen. Das bedeutet allerdings, dass immer, wenn ein Feature Branch in den Master Branch gemerged wird, erneut deployed wird und nicht eine bestimmte Version für längere Zeit zum Testen zur Verfügung stehen kann.
Außerdem wird dadurch das potentielle Bottleneck wie es in Kapitel 3.1 beschrieben ist wiedereingeführt.

3.2.2 Environment Branches

In Szenarien, in denen die Anwendung in mehreren Umgebungen (Production, Staging, Pre-Production, …) ausgerollt ist, schlägt GitLab vor, zusätzlich zum Master- und Production Branch weitere long-running Branches anzulegen (siehe Abbildung 3).

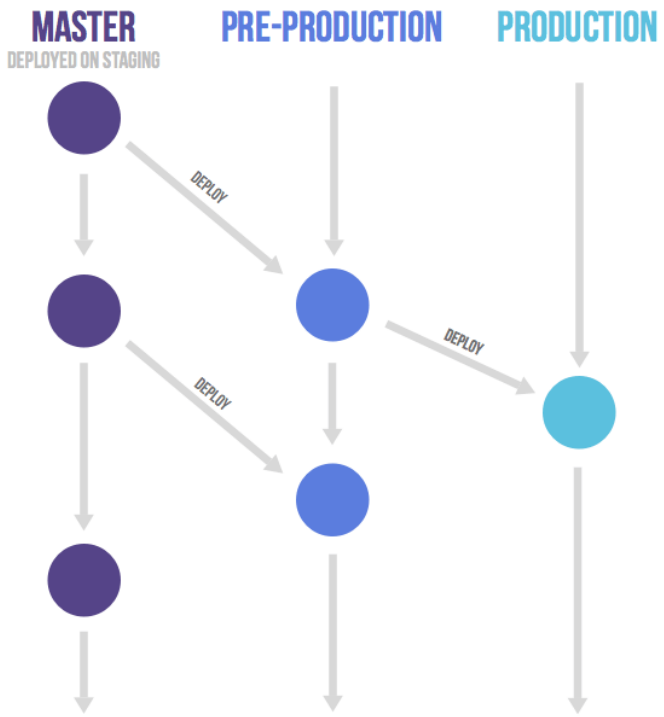

Abbildung 3: GitLab Flow mit Environment Branches [24]

Der aktuelle Commit des Production Banches ist dabei in der Produktionsumgebung ausgerollt. Aktuelle Commits in den anderen long-running Branches sind in der jeweiligen Umgebung ausgerollt. [24, Kap. Environment branches with GitLab flow]

Continuous Integration / Continuous Delivery

Während CI bei diesem Workflow kein Problem darstellt, wurde er speziell entwickelt, um verschiedene Umgebungen zu haben, in denen unterschiedliche Versionen ausgerollt sind. Damit widerspricht es der Idee der CD, aktuelle Versionen möglichst schnell und automatisiert in Produktion auszurollen. Was allerdings möglich und empfehlenswert ist, ist das automatische Deployment des Master Branches in eine Test- oder QA-Umgebung.

Skalierbarkeit

Da in diesem Workflow, ebenso wie im GitHub Workflow, stets der aktuelle Stand des Master Branches ausgerollt ist, ist dies ein potentielles Bottleneck. Das Bottleneck der Produktionsumgebung wir daher lediglich in die Staging- oder QA-Umgebung verschoben aber nicht eliminiert.

Protected Branches

Ebenso wie beim GitLab Flow mit Production Branch, sollten auch hier die ausgerollten Branches geschützt werden. Nur berechtigte Benutzer sollten Deployments anstoßen können und daher auf die entsprechenden Branches mergen können.

Komplexität

Verglichen mit dem GitLab Flow mit Production Branch gibt es hier zumindest einen weiteren Branch, der die Komplexität erhöht. Ebenso wie der Komplexitätsunterschied zwischen dem GitHub Flow und dem GitLab Flow mit Production Branch, ist auch jener zwischen GitLab Flow mit Production Branch und GitLab Flow mit Environment Branches gering da der zusätzliche Branch lediglich dem Release Management dient.

Mehrere Environments / aktive Releases

Durch das Hinzufügen eines oder mehrerer Pre-Production Branches ist es möglich, Test-, Staging- oder andere Pre-Production Umgebungen ausgerollt zu haben.

Das Warten mehrerer aktiver Releases ist mit diesem Workflow nicht möglich.

3.2.3 Release Branches

Oft ist es in der Softwareentwicklung so, dass mit mehreren aktiven, ausgerollten Versionen umgegangen werden muss. Das ist der Fall, wenn nicht alle Verwender immer sofort zur aktuellen Version wechseln können. [25, S. 199]

In diesem Fall sieht der GitLab Flow vor, dass für jedes Minor Release ein neuer Branch angelegt wird, der seinerseits vom Master Branch abzweigt, wie es in Abbildung 4 zu sehen ist. Diese Release Branches sollen so spät wie möglich abgezweigt werden, um nachträgliche Bugfixes zu vermeiden. [24, Kap. Release branches with GitLab flow]

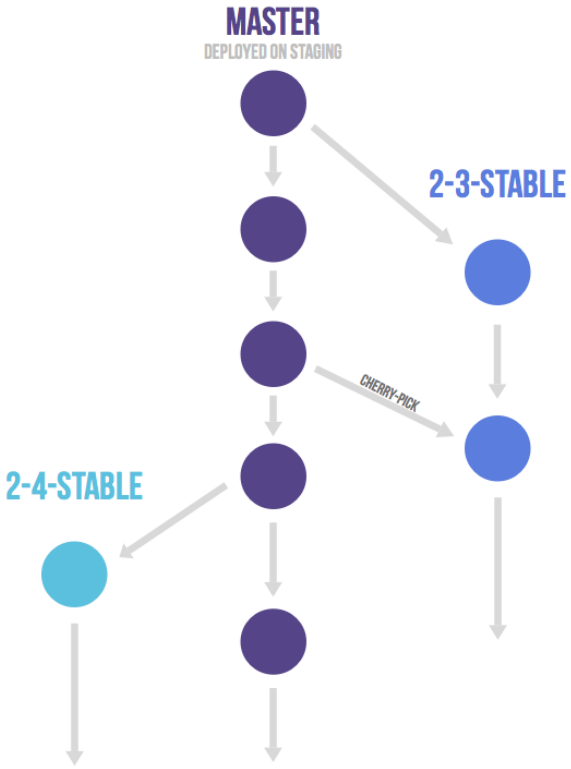

Abbildung 4: GitLab Flow mit Release Branches [24]

Für den Fall, das kritische Fehler in einem aktiven Release gefunden werden, werden diese auf einem eigenen Branch entwickelt und im Anschluss in den master gemerged. Dieser Merge Commit wird dann in Branches aller aktiven Releases per Cherry Pick übertragen. [24, Kap. Release branches with GitLab flow] Gemäß Semantic Versioning muss die Patch Nummer mit dem Cherry Pick erhöht werden. [26] Das geschieht in diesem Workflow durch das taggen der jeweiligen Commits in den Branches. [24, Kap. Release branches with GitLab flow]

Dadurch, dass der Commit vom master aus auf die jeweiligen Releases übertragen wird, wird vermieden, dass zukünftige Releases wieder den gleichen Fehler enthalten. Diese Technik heißt „Upstream First“ und wird in dieser Form auch von Google für Bugfixes aus dem Linux Kernel für Chromium OS verwendet. [27]

Continuous Integration / Continuous Delivery

Dieser Workflow wurde speziell für Szenarien entwickelt, in denen es mehr als nur einen aktiven Release gibt, was eine Grundvoraussetzung für Conuous Delivery wäre. [25, S. 176] Commits auf den jeweiligen Release Branches stellen trotzdem alle eine veröffentlichte Version dar und können bzw. sollten durch eine Release Pipeline verarbeitet werden. Commits auf dem master Branch stellen einen grundsätzlichen Release Candidate dar und muss daher immer stabil sein.

Skalierbarkeit

Ebenso wie der GitLab Flow mit Environment Branches ist auch hier das Bottleneck von der Produktionsumgebung in die Staging-Umgebung verschoben aber nicht eliminiert.

Protected Branches

Wie auch bei den anderen Workflows, sollte der Master Branch geschützt werden. Besonderen Schutz sollten jedoch die Branches genießen, die ausgerollt sind. Gerade, wenn diese von einer Release Pipeline verarbeitet werden, muss darauf geachtet werden, dass nicht zu viele Personen die Berechtigung haben, die Branches zu ändern. [28, Kap. Trey Research]

Komplexität

Durch den Fokus auf Releases, entsteht insofern ein weiterer Komplexitätsfaktor, als das entschieden werden muss, welche Features bzw. Bugfixes in bereits bestehende Versionen übernommen werden. Außerdem muss bei diesen Fixes darauf geachtet werden, dass diese rückwärtskompatibel mit allen Versionen sind, in die der Fix übernommen werden soll. Das bedeutet wiederum, dass der Programmierer beim Fixen keine Funktionalitäten verwenden darf, die erst in späteren Releases hinzugekommen sind.

Außerdem spielt es in diesem Workflow, anders als bei anderen, aufgrund der Release-orientierung eine wesentliche Rolle, welche Features zu welchem Zeitpunkt fertig gestellt werden, damit bestimmte Features in ein Release übernommen werden, während andere Features für spätere Releases noch übernommen sein sollten.

Mehrere Environments / aktive Releases

Dieser Workflow wurde eigens entwickelt, damit mehrere aktive Releases unterstützt werden. Da jeder Branch seinerseits ein Release darstellt, können einzelne Commits aus dem Master Branch in die jeweiligen Release Branches übernommen werden, um beispielsweise Sicherheitslücken und kritische Bugs zu beheben.

Andererseits ist es auch möglich, sofern das eine Anforderung des Projekts ist, diese Versionen gleichzeitig, in verschiedenen Environments, ausgerollt zu haben.

3.3 Git-Flow

Git-Flow beschreibt ein Workflow, der von Vincent Driessen im Jahr 2010 vorgestellt wurde, den er in privaten und beruflichen Projekten nach eigenen Angaben sehr erfolgreich verwendet. [29, Abs. 1]

Prinzipiell geht der Workflow von zwei grundlegenden Branches aus. Der Master Branch ist jederzeit stabil und ausrollbar. Der Develop Branch ist jener, auf dem Integration Tests durchgeführt werden. Immer, wenn sichergestellt ist, dass der Status am Develop Branch stabil ist, sollte mit dem Master Branch gemerged werden. [29, Kap. The main branches]

Zusätzlich zu diesen zwei Branches gibt es mehrere Supporting Branches. Diese können prinzipiell in Feature Branches, Release Branches und Hotfix Branches unterteilt werden. [29, Kap. Supporting branches]

Feature Branches sind ähnlich wie die aus anderen Workflows. Sie zweigen wie in Abbildung 5 zu sehen ist aus dem Develop Branch ab. Typischerweise existieren diese Branches nur

lokal und werden nicht zum Server gepushed. Nach der Fertigstellung werden Feature Branches wieder in den Develop Branch gemerged. Das geschieht ohne Fast-Forward um die Historie klar zu halten. Nach dem Merge soll der Feature Branch gelöscht werden. [29, Kap. Feature branches]

Release Branches zweigen aus dem Develop Branch ab. Sie werden verwendet um letzte Vorbereitungen vor einem Release zu treffen. Das können beispielsweise kleine Bugfixes, Versionsnummern oder andere Metadaten sein. Der Zeitpunkt, zu dem der Release Branch abgezweigt wird sollte jener sein, an dem der Develop Branch den erwarteten Status des Releases wiederspiegelt. Erst zu diesem Zeitpunkt sollte festgelegt werden, welche Versionsnummer der nächste Release trägt. Nachdem alle erforderlichen Änderungen durchgeführt wurden, wird der Branch sowohl in den Master Branch als auch in den Develop Branch gemerged. Der resultierende Merge commit im Master Branch sollte anschließend mit der jeweiligen Versionsnummer getaggt und der Release Branch schließlich gelöscht werden. [29, Kap. Release branches]

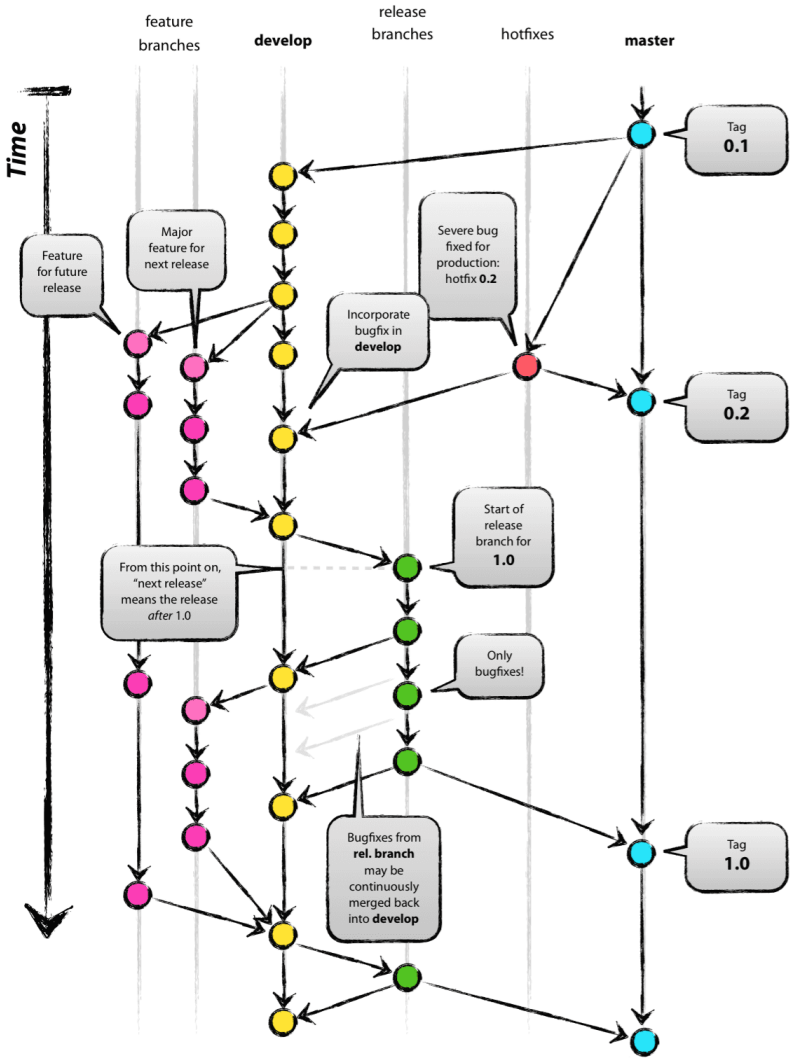

Abbildung 5: Git-Flow [29]

Im Fall, dass ein kritischer Bug in der Produktionsumgebung auftritt, wird ein Hotfix Branch erstellt. Dieser ähnelt sehr stark einem Release Branch, ist sind allerdings nicht geplant. Nachdem die kritischen Fehler auf diesem Branch behoben wurden, wird er erneut in den Master Branch gemerged und ein entsprechender Tag mit der Versionsnummer angelegt. Der Branch wird außerdem in den Develop Branch gemerged und anschließend gelöscht. [29, Kap. Hotfix branches]

Continuous Integration / Continuous Delivery

Git-Flow geht davon aus, dass Feature Branches große Änderungen enthalten und die Integration des Features aufwendig ist. Dafür wird bei Git-Flow ein Develop Branch benutzt. Die Tatsache, dass Features nach der Implementierung zusätzliche Änderungen in Form eines Release Branches benötigen oder zumindest benötigen können widerstrebt der Idee der Continuous Delivery, dass jedes Feature nach der Fertigstellung sofort ausgeliefert werden kann. [30, Kap. The current state of things (spoiler: git-flow)]

Skalierbarkeit

Da hier der Develop Branch prinzipiell nicht in einer Testumgebung ausgerollt ist, entfällt das Bottleneck, dass Deployments zu lange dauern könnten, wie es beim GitHub Flow sowie beim GitLab Flow mit Environment- oder Release Branches existiert.

Außerdem wird hier der Release Branch vom Develop Branch abgezweigt bevor ein Release tatsächlich durchgeführt wird, sodass letzte Anpassungen durchgeführt werden können. Da diese Änderungen auf einem eigenen Branch durchgeführt werden, wird hier der Develop Branch nicht beeinflusst und die Entwicklung kann ungehindert fortgesetzt werden. Ohne Release Branches müssten diese Änderungen potentiell am Develop Branch durchgeführt werden und die eigentliche Entwicklung dafür angehalten werden.

Protected Branches

Die Rolle, die in den anderen Workflows der Master Branch inne hatte, übernimmt hier der Develop Branch. Dementsprechend sollte dieser geschützt werden. Ebenso geschützt werden sollten die Release Branches, da diese den letzten Schritt vor dem eigentlichen Release darstellen und besondere Aufmerksamkeit verdienen. Der Master Branch, auf dem jeder Commit einen produktiven Release darstellt, darf nur Commit enthalten, deren Autoren berechtigt sind, Releases durchzuführen.

Komplexität

Durch das Einführen der Supporting Branches steigt die Komplexität. Diese Komplexität war einer der Gründe für das Entstehen der GitLab Flows. [24] Um Entwickler bei der Arbeit mit Git-Flow zu unterstützen, hat Vincent Driessen eine Erweiterung für Git entwickelt. Mit dieser können Operationen im Git-Flow mit einem höheren Abstrahierungsgrad durchgeführt werden. [31]

Mehrere Environments / aktive Releases

Git-Flow sieht vor, dass der letzte Commit am Master Branch das aktuelle Release darstellt. Dadurch ist es nicht möglich, mehrere aktive Releases zu warten. Bei jeder Änderung des Master Branches ist ein neuer Release durchzuführen. Das heißt, dass beispielsweise

Fehlerbehebungen stets auf dem letzten aktiven Release basieren und die Versionsnummer zwingendermaßen erhöhen.

4 Conclusio

Wichtig ist, dass es keinen Workflow gibt, der für alle Anwendungszwecke optimal ist. Stattdessen sollte sich der Workflow an den Geschäftsanforderungen orientieren. [1] In Tabelle 1 finden sich die Erkenntnisse über Stärken und Schwächen der jeweiligen Workflows wieder.

	CI/CD	Protected Branches	Geringe Komplexität	Skalierbarkeit	Mehrere Releases	Mehrere Environments
GitHub	✓	✓	✓	~	×	×
GitLab Production Branch	~	✓	✓	✓	×	×
GitLab Environment Branches	~	✓	✓	~	×	✓
GitLab Release Branches	~	✓	✓	~	✓	×
Git-Flow	×	✓	×	✓	×	×

Tabelle 1: Übersicht über die beschriebenen Workflows

Anhand der Erkenntnisse lässt sich feststellen, dass sofern es eine Anforderung des Projekts ist, mehrere aktive Releases zu warten, prinzipiell nur der GitLab Flow mit Release Branches in Frage kommt.

Für den Fall, dass verschiedene Versionen in verschiedenen Umgebungen ausgerollt sein müssen, kommt prinzipiell nur der GitLab Flow mit Environment Branches in Frage.

Projekte, in denen CD eine wesentliche Rolle spielt, sollten mit dem GitHub Flow durchgeführt werden. Dieser verfolgt als einziger die Idee Features sofort nach der Fertigstellung auszuliefern. [30, Kap. Enter GitHub Flow]

Für besonders große Projekte, bei denen mit Performance-Engpässen im Zusammenhang mit CD gerechnet werden muss, bietet es sich an einen Workflow zu wählen, bei dem nicht jeder Commit auf den Master- bzw. Develop Branch zu einem Deployment führt. Daher ist der GitHub Flow auszuschließen.

Git-Flow wurde in der Vergangenheit mehrfach für seine Komplexität kritisiert. [32] Dementsprechend wurde bei der Entiwcklung neueren Workflows darauf geachtet besonders simpel und leicht erlernbar zu sein, statt weitere Komplexität in die Softwareentwicklung hinzuzufügen. [24, Kap. Git flow and its problems], [29]

Ausgehend von den Vor- und Nachteilen der Workflows ist ferner festzustellen, dass sich in der Regel je nach Art des Projekts ein Workflow besonders gut anbietet.
Für SaaS-Projekte ist dies der GitHub Flow, da hier in der Regel jederzeit deployed werden kann, sofern die Qualität der Software durch umfangreiches automatisiertes testen sichergestellt ist. [4, Kap. Continuous delivery workflow for SaaS products] Wenn es notwendig ist, das Produkt in einer Testumgebung auszurollen und manuelle Tests durchzuführen, können auch der GitLab Flow mit Production Branch oder der GitLab Flow mit Environment Branches verwendet werden.
Für Software, die installiert werden muss, ist prinzipiell zu differenzieren, ob in Zukunft mehrere Releases gewartet und mit Updates versorgt werden müssen oder jeweils nur ein aktiver Release existiert. Im Fall, dass mehrere Releases gewartet werden müssen empfiehlt sich der GitLab Flow mit Release Branches. [4, Kap. Continuous delivery workflow for installed products] Anderenfalls bieten sich der GitLab Flow mit Production Branch sowie Git-Flow an. Einer der wesentlichen Unterschiede zwischen den beiden Workflows ist der Release Branch, den Git-Flow als Support Branch sieht. Während dieser die Möglichkeit bietet letzte Anpassungen vorzunehmen, führt er auch zusätzliche Komplexität ein und erschwert Releases durch die zusätzlich vorzunehmende Arbeit.
Am Beispiel des Linux Kernel ist ersichtlich, dass für ein derart großes Projekt, bei dem eine einzelne Person nicht alle Änderungen überblicken kann, keiner der obigen Workflows geeignet ist. Daher wird für diesen ein Dictator and Lieutenants Workflow verwendet. [33, Kap. 2.3: HOW PATCHES GET INTO THE KERNEL] Das bedeutet, dass die Software in mehrere Subsysteme unterteilt wird und jeweils ein Lieutenant verantwortlich für ein Subsystem ist. Der Dictator akkumuliert dann die Änderungen der Lieutenants und veröffentlicht diese in seinem Repository. [14, Kap. Dictator and Lieutenants Workflow]

Abschließend ist zu sagen, dass die vorgestellten Workflows lediglich als Vorlagen dienen und an die konkreten Bedürfnisse eines Projekts angepasst werden können.

Literaturverzeichnis

[1] Atlassian, „Git Workflow | Atlassian Git Tutorial“, *Atlassian*. [Online]. Verfügbar unter: https://www.atlassian.com/git/tutorials/comparing-workflows. [Zugegriffen: 12-Nov-2017].

[2] Atlassian, „Continuous integration, explained | Continuous Delivery“, *Atlassian*. [Online]. Verfügbar unter: https://www.atlassian.com/continuous-delivery/continuous-integration-intro. [Zugegriffen: 12-Nov-2017].

[3] „Was ist Continuous Integration? | DevOps“. [Online]. Verfügbar unter: https://www.visualstudio.com/de/learn/what-is-continuous-integration/. [Zugegriffen: 12-Nov-2017].

[4] Atlassian, „Feature Branching Workflows for Continuous Delivery | Atlassian Continuous Delivery“, *Atlassian*. [Online]. Verfügbar unter: https://www.atlassian.com/continuous-delivery/continuous-delivery-workflows-with-feature-branching-and-gitflow. [Zugegriffen: 12-Nov-2017].

[5] „Continuous Integration – strategies for branching and merging | Art of Software Engineering“. [Online]. Verfügbar unter: http://eugenedvorkin.com/continuous-integration-strategies-for-branching-and-merging/. [Zugegriffen: 12-Nov-2017].

[6] „What is Continuous Delivery? - Continuous Delivery“. [Online]. Verfügbar unter: https://continuousdelivery.com/. [Zugegriffen: 12-Nov-2017].

[7] „Continuous Integration - Continuous Delivery“. [Online]. Verfügbar unter: https://continuousdelivery.com/foundations/continuous-integration/. [Zugegriffen: 12-Nov-2017].

[8] V. Power, „What’s the difference between CI and CD, anyway?“ [Online]. Verfügbar unter: http://blog.wercker.com/what-is-ci-and-cd. [Zugegriffen: 12-Nov-2017].

[9] „Continuous Integration (CI) Best Practices with SAP: CI/CD Practices | SAP“. [Online]. Verfügbar unter: https://www.sap.com/developer/tutorials/ci-best-practices-ci-cd.html. [Zugegriffen: 12-Nov-2017].

[10] S. Smith, „Practical continuous deployment: a guide to automated software delivery“, *Atlassian Blog*, 17-Feb-2017. [Online]. Verfügbar unter: https://www.atlassian.com/blog/continuous-delivery/practical-continuous-deployment. [Zugegriffen: 12-Nov-2017].

[11] A. B. Lukas Christoph, „Eine Einführung in Continuous Delivery, Teil 3: Acceptance Test Stage“, *heise Developer*. [Online]. Verfügbar unter: https://www.heise.de/developer/artikel/Eine-Einfuehrung-in-Continuous-Delivery-Teil-3-Acceptance-Test-Stage-2457023.html. [Zugegriffen: 12-Nov-2017].

[12] „Vorgehensweise beim Einrichten einer fortlaufenden Integrations- und Bereitstellungspipeline (CI/CD)“, *Amazon Web Services, Inc.* [Online]. Verfügbar unter: //aws.amazon.com/de/getting-started/projects/set-up-ci-cd-pipeline/. [Zugegriffen: 25-Jan-2018].

[13] „Bottleneck | Define Bottleneck at Dictionary.com“. [Online]. Verfügbar unter: http://www.dictionary.com/browse/bottleneck. [Zugegriffen: 21-Jan-2018].

[14] „Git - Distributed Workflows“. [Online]. Verfügbar unter: https://git-scm.com/book/en/v2/Distributed-Git-Distributed-Workflows. [Zugegriffen: 21-Nov-2017].

[15] „Repository permission levels for an organization - User Documentation“. [Online]. Verfügbar unter: https://help.github.com/articles/repository-permission-levels-for-an-organization/. [Zugegriffen: 21-Nov-2017].

[16] „Git - An Example Git-Enforced Policy“. [Online]. Verfügbar unter: https://git-scm.com/book/uz/v2/Customizing-Git-An-Example-Git-Enforced-Policy. [Zugegriffen: 12-Nov-2017].

[17] „GitHub Flow“. [Online]. Verfügbar unter: https://githubflow.github.io/. [Zugegriffen: 21-Nov-2017].
[18] „PHP: Supported Versions“. [Online]. Verfügbar unter: http://php.net/supported-versions.php. [Zugegriffen: 13-Jan-2018].
[19] „FAQ zum Testserver (PBE)“, *Riot Games Support*. [Online]. Verfügbar unter: http://support.riotgames.com/hc/de/articles/201751904-FAQ-zum-Testserver-PBE-. [Zugegriffen: 13-Jan-2018].
[20] „Understanding the GitHub Flow · GitHub Guides“. [Online]. Verfügbar unter: https://guides.github.com/introduction/flow/. [Zugegriffen: 21-Nov-2017].
[21] lucamezzalira, „Git Flow vs Github Flow“, *Luca Mezzalira*, 10-März-2014. [Online]. Verfügbar unter: https://lucamezzalira.com/2014/03/10/git-flow-vs-github-flow/. [Zugegriffen: 21-Nov-2017].
[22] „Forking Projects · GitHub Guides“. [Online]. Verfügbar unter: https://guides.github.com/activities/forking/. [Zugegriffen: 21-Nov-2017].
[23] „App Review - App Store - Apple Developer“. [Online]. Verfügbar unter: https://developer.apple.com/app-store/review/. [Zugegriffen: 28-Dez-2017].
[24] „GitLab Flow“, *GitLab*. [Online]. Verfügbar unter: https://about.gitlab.com/2014/09/29/gitlab-flow/. [Zugegriffen: 28-Dez-2017].
[25] R. Preißel und B. Stachmann, *Git*, 4. Aufl. Heidelberg: dpunkt.verlag.
[26] T. Preston-Werner, „Semantic Versioning 2.0.0“, *Semantic Versioning*. [Online]. Verfügbar unter: https://semver.org/. [Zugegriffen: 31-Dez-2017].
[27] „Upstream First - The Chromium Projects“. [Online]. Verfügbar unter: http://www.chromium.org/chromium-os/chromiumos-design-docs/upstream-first. [Zugegriffen: 31-Dez-2017].
[28] „4 - Automating the Release Pipeline“. [Online]. Verfügbar unter: https://msdn.microsoft.com/en-us/library/dn449951.aspx. [Zugegriffen: 26-Jan-2018].
[29] V. Driessen, „A successful Git branching model“, *nvie.com*. [Online]. Verfügbar unter: http://nvie.com/posts/a-successful-git-branching-model/. [Zugegriffen: 01-Jan-2018].
[30] W. Buchwalter, „A Git Workflow for Continuous Delivery“, *A Git Workflow for Continuous Delivery*, 21-Juni-2016. [Online]. Verfügbar unter: https://blogs.technet.microsoft.com/devops/2016/06/21/a-git-workflow-for-continuous-delivery/. [Zugegriffen: 08-Jan-2018].
[31] V. Driessen, „gitflow: Git extensions to provide high-level repository operations for Vincent Driessen’s branching model“, 14-Jan-2018. [Online]. Verfügbar unter: https://github.com/nvie/gitflow. [Zugegriffen: 14-Jan-2018].
[32] „Git Flow is superflous and complex“, *Alan’s amusing and surprising homepage*, 19-Juli-2016. [Online]. Verfügbar unter: https://www.franzoni.eu/git-flow-is-superflous-and-complex/. [Zugegriffen: 25-Jan-2018].
[33] „How to Participate in the Linux Community“, *Linux.com | The source for Linux information*, 01-Aug-2008. [Online]. Verfügbar unter: https://www.linux.com/publications/how-participate-linux-community. [Zugegriffen: 13-Jan-2018].

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

CD	Continuous Delivery
CI	Continuous Integration
MR	Merge Request
PR	Pull Request
QA	Quality Assurance
SaaS	Software as a Service