GitWaterFlow: A Successful Branching Model and Tooling, for Achieving Continuous Delivery with Multiple Version Branches

Rayene Ben Rayana Sylvain Killian Nicolas Trangez Arnaud Calmettes

Scality, Release Engineering Dept.
11 rue Tronchet, Paris, France
firstname.lastname@scality.com

ABSTRACT

Collaborative software development presents organizations with a near-constant flow of day-to-day challenges, and there is no available off-the-shelf solution that covers all needs. This paper provides insight into the hurdles that Scality's Engineering team faced in developing and extending a sophisticated storage solution, while coping with ever-growing development teams, challenging — and regularly shifting — business requirements, and non-trivial new feature development.

The authors present a novel combination of a Git-based Version Control and Branching model with a set of innovative tools dubbed *GitWaterFlow* to cope with the issues encountered, including the need to both support old product versions and to provide time-critical delivery of bug fixes.

In the spirit of Continuous Delivery, Scality Release Engineering aims to ensure high quality and stability, to present short and predictable release cycles, and to minimize development disruption. The team's experience with the *GitWaterFlow* model suggests that the approach has been effective in meeting these goals in the given setting, with room for unceasing fine-tuning and improvement of processes and tools.

CCS Concepts

•**Software and its engineering** → **Software configuration management and version control systems; Software evolution; Software version control;** *Agile software development; Software testing and debugging;*

Keywords

branching model; version control; continuous integration; gatekeeper; workflow automation; concurrent release cycles

1. INTRODUCTION

Software development has been an inherently collaborative effort since the advent of computing, with applications, libraries, and whole products often developed by teams of contributors spread across multiple locations (e.g., in the Open Source sphere). With this come many challenges, related to establishing and maintaining the quality and stability of the delivered work, coordination of changes and maintaining multiple versions delivered to customers.

Scality RING is a distributed system providing object and cloud storage allowing one to store and access billions of objects or even PB-sized objects across standard x86 hardware to meet the most demanding digital business, cloud and application requirements.

At the start, the small-scale team of Scality engineers working to develop the RING product employed CVS and later Subversion in an ad-hoc fashion, and collaboration happened 'on the spot'.

The engineering team pushed features and bug fixes into a shared *trunk* branch. This branch was slated to become the next 'major' release of the product, though overeager integration of partially-delivered features often resulted in the branch being in a non-shippable state.

Scality RING is at the basis of many business-critical information systems. Therefore, customers tend to minimize the number of feature rich upgrades of their systems. This has a major impact on the development process since it requires continuous maintenance of multiple version branches.

The process to port bug fixes to relevant branches ('backporting'), sometimes requiring changes to the patch itself was fully manual. When the change rate of the codebase reached a certain level, this turned out to be a bottleneck in the development process. Furthermore, being a manual process, there was a continuous risk these backports introduced accidental bugs or regressions. Creation of backport commits on various version branches also destroyed relationships between semantically equivalent changesets, which could only be recovered through information kept in commit messages or the ticketing system, again relying on humans doing the right thing.

The developer workflow was, in practice, interrupt-driven: one could be requested to assist in the backporting of a change made weeks ago whilst already working on an entirely unrelated feature. Also, as a consequence of a 'nightly build' testing procedure, there could be significant delays between introduction of a change and feedback on its correctness. Furthermore, due to the nature of 'nightly builds' which test the aggregate of multiple distinct changes in one go, failure of these test runs could be caused by any (unrelated) change that landed within the same timeframe.

RELENG'16, November 18, 2016, Seattle, WA, USA
ACM. 978-1-4503-4399-2/16/11...
http://dx.doi.org/10.1145/2993274.2993277

2. GOALS

To overcome the deficiencies and drawbacks of our former development process, we set to radically change our approach, easing the workflow for developers as well as ensuring correctness of meta-information. Our high-level goals can be summarized as follows:

Full automation of all processes reducing room for error, and ensuring information retained in various systems is kept in sync by construction.

Keep version branches in shippable state increasing trust in the ability to meet delivery dates within a team, as well as within the company as a whole. As an engineer, this implies one can at all times start to develop a new feature on a 'stable base'.

Simplified workflow with less interruptions improving understanding for all contributors, especially new ones, reducing room for human mistakes. Reducing the number of interactions a developer is to have with the system reduces the number of expensive 'context switches' during day-to-day development work.

Retain meta-information and ensure its correctness by beeping links between semantically equivalent patches which allows to answer questions like "In which versions is this patch contained?" with confidence. Automation (see above) also ensures this meta-information is in sync between multiple systems (including VCS, code review, ticketing system, etc.,).

3. RELATED WORK

3.1 Branching Practices

Most version control management user manuals describe basic branching functionality (e.g., [2]). In addition, multiple branching and merging patterns are identified and thoroughly explained in [1] and [6]. Also, in [3] the authors conducted a survey that served to provide a broad view of modern practices in collaborative development, whereas in [4] a similar study is made at company level to identify branching practices and their impact on parallel development in an Agile environment.

One popular branching pattern is to maintain ***one branch per release***. The main drawback of this method is the difficulty in maintaining the succession of active branches without proper tooling, as this involves a large degree of merging and manual conflict resolution. A noteworthy project that takes this one branch per release route is the standard Python distribution[1].

A second pattern, one that allows for arbitrarily complex multi-level validation processes, is to define ***one branch per promotion*** or stage reflecting the validation process by maintaining branches for Development, Testing and Release with increasing quality levels.

Yet another pattern is to create ***one temporary branch per task*** (feature, bugfix, improvement, etc.) and to merge these branches into the common, permanent, development branch once the task is completed. This method eases concurrent development and keeps the development branch in a relatively good, shippable state at all times (provided the task branches are validated prior to being merged using gatekeeping techniques).

Vincent Drissen combined the two latter patterns to design *gitflow*[2], consisting mainly of using a permanent development branch as the mainline branch of the "temporary branch per task" pattern. The branch is then staged to a release branch when a release is feature-complete and fully validated revisions are merged into the *master* branch. This workflow has grown quite popular in recent years, however it assumes the software to be in a rolling release cycle and does not allow for multiple release generations to be concurrently alive at the same time.

3.2 Gatekeeping and Merging Tools

In traditional continuous integration, automated testing occurs once the code is merged into the mainline branch. This practice allows regressions and bugs to be merged assuming that it is usually easy to find and fix a breaking changeset which is true for small and co-located teams, however as teams grow in size this generates a lot of useless communication and frustration, especially when the team is dispersed across different timezones.

Gatekeeping is the process of attempting to prevent regressions from being merged by running tests on the software before changes are merged (and thus keeping the mainline branch working for all developers). A number of tools are available with which a team can automate this process, and thus prevent human errors, including: Gerrit, Bors, Homu, and Zuul.

Initially forked from Rietveld[3], Gerrit[4] is a popular code review tool for Git. It acts as a gatekeeper on a public git repository by adding a review process that is triggered when a developer pushes a new changeset to the repository. This review process can be split into two steps, both of which require a positive score prior to merging the change into the mainline branch: typically automated verification and peer reviewing.

An integration robot written for the Rust project, Bors[5] automates the merging of pull requests against the master branch of a GitHub project. It is written as an idempotent stateless function that continuously polls for pull request changes, walking them through different states (unreviewed, approved, tested, closed).

Homu[6] is a gatekeeper robot that implements the same state machine as Bors. Unlike that tool, however, Homu is stateful and trigger-based, maintaining a state in a local database and using webhooks instead of regularly polling pull requests, in order to limit the bandwidth and API calls its work requires.

Zuul[7] is OpenStack's gatekeeping system. Its most noteworthy feature is how it handles parallel testing of ordered dependent changesets across projects and repositories. Zuul uses an optimistic queue system: dependent changesets are merged into a shared integration branch, thus if changesets A, B, and C are submitted in rapid succession C will be tested with the prior changes included, even though those changes are currently being processed by the testing pipeline.

[1] https://hg.python.org/cpython/branches

[2] http://nvie.com/posts/a-successful-git-branching-model/

[3] https://github.com/rietveld-codereview/rietveld

[4] https://www.gerritcodereview.com/

[5] https://github.com/graydon/bors

[6] http://homu.io

[7] http://docs.openstack.org/infra/zuul/

Such an approach is characterized as optimistic because it relies on the assumption that changesets A and B will pass all the tests to validate C (and as such, once the assumption is verified all of the changes can be merged at once, hence a dramatic increase of the rate at which changes are merged onto the mainline branch).

4. GitWaterFlow

GitWaterFlow (GWF) is a combination of a branching model and its associated tooling, featuring a transactional view on multi-branch changesets supported by none of the tools and models previously described. GWF tends to ban "backporting" in favor of "(forward) porting".

4.1 Porting is Better than Backporting

The term "porting" is employed to describe the act of developing a changeset on an old — yet active — version branch and subsequently merging it on newer ones. It is considered better than "backporting" for multiple reasons cited in [3], "Porting" also makes merge automation trivial. In fact, changes that are merged in an old version branch, whether fixes or improvements, must also land in newer ones, otherwise there is a risk of regression. A bot can use this assumption to prepare and then execute the merge on newer branches, thus offloading the developer.

4.2 Development Branches

GWF comes with a versioning scheme that is inspired by semver [5]. Basically, version numbers are in the form *major.minor.patch*. *patch* is incremented only when backward-compatible bug fixes are being added, *minor* is incremented when backward-compatible features are added, and *major* is incremented with major backward incompatible changes.

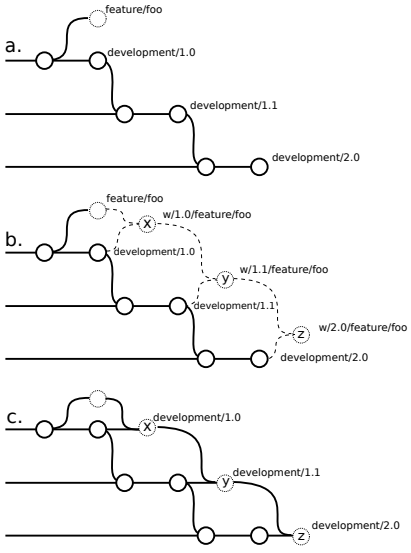

Figure 1: Steps to merge a feature branch with GitWaterFlow

In GWF, every living *minor* version has a corresponding *development/major.minor* branch, each of which must be included in newer ones. In fig.1.a *development/1.0* is included into *development/1.1*, which in turn is included in *development 2.0*. Consequently, a GWF-compliant repository has a waterfall-like representation, hence the name "GitWaterFlow".

4.3 Feature Branches

As GWF is based on 'porting', feature branches do not necessarily start from the latest development branch. In fact, prior to start coding a developer must determine the oldest *development/** branch his code should land upon (refer to fig.1.a). Once ready to merge, the developer creates a pull request that targets the development branch from which he started. A gating and merging bot will ensure that the feature branch will be merged not only on the destination but also on all the subsequent development branches.

4.4 Transactional Multi-Branch Changes

The fact that every pull request can concurrently target more than one mainline branch can dramatically affect the approach that developers take in addressing issues. For instance, it is not uncommon that conflicts exist between the feature branch targeting version *n*, and version *n+1*. In our setup, this class of conflicts must be detected and fixed *prior to merging the pull request*. The code that resolves such conflicts is considered part of the change, in fact, and must be reviewed at the same time. Also, it is a requirement that a pull request be merged only once it has passed the tests on *all* targeted versions.

In short, the changes brought to the software on multiple branches is a single entity and should be developed, reviewed, tested, and merged as such.

4.5 Bert-E

Bert-E is the gatekeeping and merging bot Scality developed in-house to automate GWF, its purpose being to help developers merge their feature branches on multiple development branches. The tool is written in Python and designed to function as a stateless idempotent bot. It is triggered via Bitbucket webhooks after each pull request change occurrence (creation, commit, peer approval, comment, etc.).

Bert-E helps the developer prepare his pull request for merging. It interacts directly with the developer through Bitbucket's comment system via the pull request's timeline, pushing contextualized messages on the current status and next expected actions. In Scality's case, Bert-E asks the developer to 1/ ensure the JIRA *fixVersion* field correctness with regard to target branches, 2/ obtain a couple of peer approvals, and 3/ approve his own pull request (refer to fig.2.).

Bert-E usually replies in less than 50 seconds, thus creating a trial-and-error process with a fast feedback loop that is ideal in onboarding newcomers to the ticketing process.

4.6 Integration Branches

In parallel with the previously described process, Bert-E begins trying to merge on the subsequent development branches by creating integration branches named *w/major.minor/feature/foo*, after both the originating feature branch and the target development branch (refer to fig.1.b).

Every time Bert-E is triggered, it checks to ensure that the *w/** branches are ahead of both the feature branch and the corresponding development branches (updating them following the same process when this is not the case).

Every change on a *w/** branch triggers a build/test session on a Jenkins instance. When the pull request fulfills all the

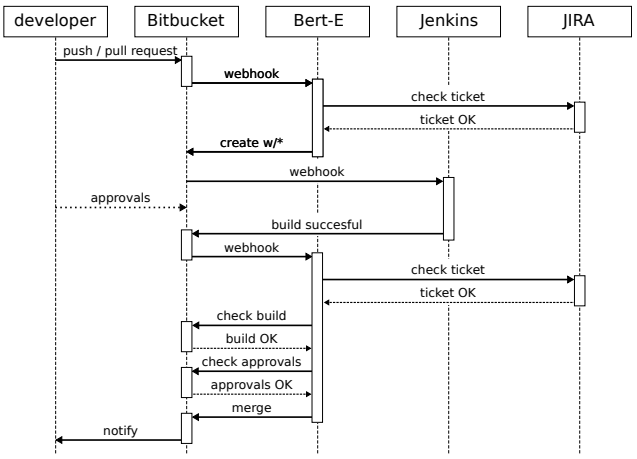

Figure 2: Merging a feature branch with Bert-E - Best case scenario

requirements previously described, and when the builds are green on all the *w/** branches, Bert-E fast-forwards all the development branches to point to the corresponding *w/** branches in an atomic transaction, as depicted in fig.1.c. Note that if another pull request is merged in the interim, Bert-E will not be able to push and must re-update its *w/** branches and repeat the build/test process.

4.7 Managing Conflicts and Adapting Code to Newer Versions

Bert-E may encounter conflicts in the creation of the integration branches, in which case it comments out the pull request and gives the developer the Git commands required to reproduce the conflict on his machine. The developer then fixes the conflict on an integration branch, and pushes it to trigger Bert-E again and proceed further in the merge process.

4.8 Other Branches

In addition to the branches previously described, there are also *release/major.minor* branches that point to the latest release for every version branch and *hotfix/** branches that have the same purpose as in gitflow (usually starting from *release/**).

Finally, Scality has added *stabilization/major.minor.patch*, which we optionally create to protect the branch from receiving last-minute risky changesets. These are similar to *development/* branches*, the difference being that they do not automatically receive merges from the upstream cascade.

5. EXPERIENCE REPORT

5.1 Better 'Definition of DONE' and Smoother Developer Process

In use at Scality for nearly a year, we can testify that the main GWF benefit is its atomic multi-branch merge property. In this context, 'DONE' means merged and fully tested on all target branches, and there is no additional backporting phase wherein it is discovered that the backport is more complex than the fix itself. Target branch conflicts are detected early and are dealt with prior to merging.

Peer reviews/approvals aside, the development process is smoother and allows the developer to push his changeset to completion without depending on third parties to merge. Changeset ownership reverts back to the author and does not vary across time. Thus, the developer is responsible for it up until the merge.

Also, the metadata in the git repository is much clearer, and now a simple `'git branch --contains <commit>'` will indicate within which branch a change has been merged.

Due to gatekeeping, the development branches are always in a shippable state, which has greatly improved Scality's accuracy in predicting delivery dates. Hand-in-hand with that, the amount of overall engineering work in progress has been reduced due to the GWF deployment, and as a direct result Scality is shipping faster.

5.2 GWF and Long Running Test Suites

Although the switch to GWF has been positive on the overall, some aspects still require refinement, with one major issue being a consequence of Scality's extensive test suite. In GWF, when two builds start and finish at approximately the same time, and both succeed, only the first one successfully merges. Regarding the second build, Bert-E detects that the development branch(es) have been updated and will subsequently update the *w/** branch, which triggers a new build cycle. As a consequence, the theoretical daily merge capacity is 24h divided by the duration of the test suite. To overcome this limitation, we have begun developing a second version of GitWaterFlow that performs optimistic merge queuing in a Zuul-like fashion (refer to 3.2) but with the support of multiple development branches.

6. ACKNOWLEDGMENTS

The authors would like to thank Pierre-Louis Bonicoli and Maxime Vaude for their valuable contributions to Bert-E's source code, and Kory Kessel for editing the paper.

7. REFERENCES

[1] B. Appleton, S. Berczuk, R. Cabrera, and R. Orenstein. Streamed lines: Branching patterns for parallel software development. In *Proceedings of PloP*, volume 98, 1998.

[2] S. Chacon. *Pro Git*. Apress, 2014.

[3] S. Phillips, J. Sillito, and R. Walker. Branching and merging: an investigation into current version control practices. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 9–15. ACM, 2011.

[4] R. Premraj, A. Tang, N. Linssen, H. Geraats, and H. van Vliet. To branch or not to branch? In *Proceedings of the 2011 International Conference on Software and Systems Process*, pages 81–90. ACM, 2011.

[5] Semantic versioning 2.0.0. http://web.archive.org/web/20160630232710/http://semver.org/. Accessed: 2016-06-30.

[6] C. Walrad and D. Strom. The importance of branching models in scm. *Computer*, 35(9):31–38, 2002.