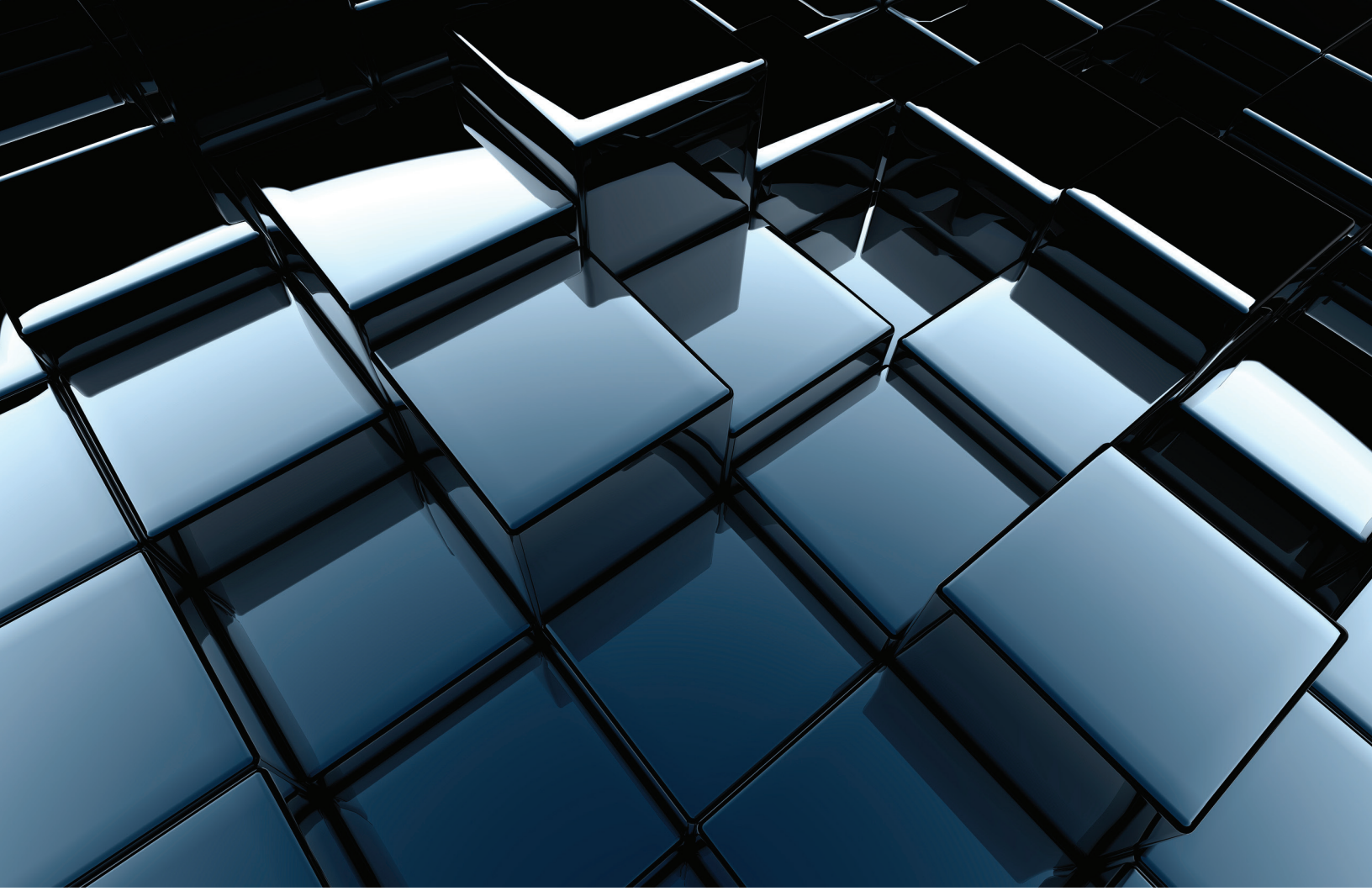


Practical Git

Go2Group
Complexity Made Simple

Practical Git

Revised September 2014

Practical Git

Practical Git

Introduction

Software development teams need a way to guarantee that everyone knows which code is the latest, ensure that no one writes over someone else's changes, and prevent errors. In the old days, this was done by carefully managing files, using simple systems to communicate between developers. As teams grow, simple systems become unworkable, especially when development teams span the globe, work different hours, speak different languages, work at different companies, and combine new development and maintenance tasks.

Today, we use Source Code Management (SCM) systems. Git is the defacto SCM system for software development. Just as vi is simply the editor you use on Linux, Git is the SCM tool you use for new software projects.

Your developers learn Git through GitHub as they start coding in college and on open source projects. Development managers now see Git as a talent retention strategy.

This book explains how to get the most out of a Git deployment.

Definition of Terms

Source Code Management (SCM) systems allow diverse teams to manage complex projects.

Version Control Systems (or VCS, also known as **source control** or **revision control**) is a system that tracks changing code, files, data, and documents. It identifies the latest elements and allows changes to be reversed. It manages the work of multiple authors, so that one person doesn't accidentally lose changes made by someone else.

Repository: The data structure where resources reside, managed by the VCS.

Git and **Subversion (SVN)** are the two dominant open source SCM tools. Collectively they represent about 90% of the SCM market for open source projects and about 60% of the market for enterprise SCM tools.

GitHub: A web-based host for Git-managed projects. GitHub is the most popular code repository site for open source projects

Management Console: The user interface that controls the version control system. They provide basic repository management: creating repositories, managing access control, and in some cases coordinating backups and replication. They provide social and collaboration features: activity streams, projects, code review, merge requests, and integration with build and test systems.

Stash: A Git repository management and collaboration product from Atlassian. Stash serves as the master repository for Git projects and as a project portal.

Need Help?

Go2Group
Complexity Made Simple

Go2Group is expert in software development tools, including Git and Stash. A global provider of consulting services in Application Lifecycle Management (ALM) systems, Go2Group has implemented thousands of enterprise-level migrations. We specialize in complex integration projects involving multiple ALM platforms, such as HP, IBM, Perforce, Salesforce, TFS, ClearQuest, Rally, Seapine and SugarCRM. Our goal: *Make it easy.*

An Enterprise and Platinum Atlassian Expert, we offer full services for all Atlassian products. We are the world's largest reseller of Atlassian tools.

Our ConnectALL product effortlessly and seamlessly connects multiple ALM systems so your teams can use the tools of their choice and still work together.

Git and SVN: SCM Tools of Choice

Git and Subversion (SVN) are the two dominant open source tools for Source Code Management (SCM). Collectively they represent about 90% of the SCM market for open source projects and about 60% of the market for enterprise SCM tools. Git and SVN are both solid SCM tools with a solid pedigree, strong community support, and good support from other tool vendors. Both tools will serve well in an enterprise setting. Choosing between SVN and Git is largely a matter of style and preference.

This document will lay out a decision matrix for selecting between Git and SVN.

Background

SVN was started around the year 2000 as a replacement for CVS (Concurrent Versions System or Concurrent Versioning System). The initial goal of the project was to overcome deficiencies in CVS, a goal that was largely met by 2004. SVN rapidly became the open source SCM tool of choice in both the open source and enterprise communities. A key point in SVN's favor is its reliance on the Apache web server as a platform. Building on Apache gives it strong security and access control tools, good performance, and even advanced replication features.

Git was started in 2005 as the SCM replacement for the Linux kernel project. It was designed to fit the workflow of that project: a large but distributed pool of potentially anonymous contributors, guided by a small core of module owners. Git entered the enterprise setting in 2007 when it was adopted for the Android project. That move brought Git into companies with thousands of developers, and spurred the creation of new tools like Gerrit. Git accelerated further with the launch of GitHub in 2009, making it the de facto public hosting site for all types of projects.

Futures

Both SVN and Git will maintain a strong presence in the enterprise market for many years to come. SVN has reached the late majority phase, meaning that the supporting tools and infrastructure are solid enough to be accepted by the vast majority of enterprise software and IT departments. SVN is mature, has commercial sponsorship, and is an important tool for many large companies.

Git is reaching the early majority phase. It is well entrenched in Silicon Valley, and the supporting tools and infrastructure are slowly reaching maturity. As it was designed for an open source workflow, Git has no built-in access control, replication, or HA/DR features. All of these shortcomings are now addressed by open source and commercial add-ons. Git is hugely popular in the developer community in North America, and the lagging markets (EMEA and AsiaPac) are catching up. The only real question about Git is how large its eventual market share will be.

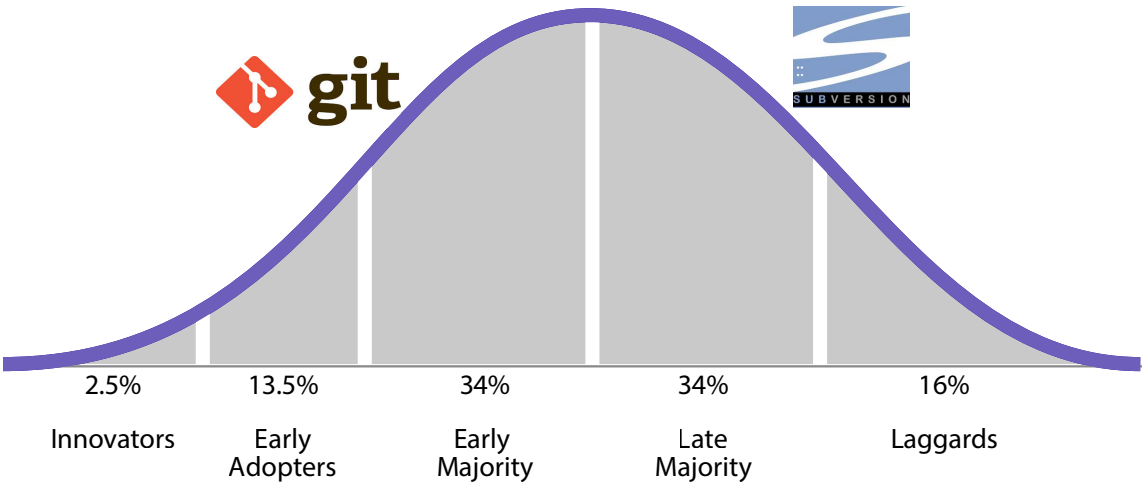


FIGURE 1: ADOPTION CURVE

Key Features

Subversion	Git
Central server	Distributed version control
Apache security	Needs external security system
Stable and mature	Enterprise tools maturing
Best for mainline model	Supports several workflows

- Much is made of Git's distributed nature. In an enterprise setting that fact is less relevant than it appears at first glance. Almost without fail Git will be used in a centralized manner (i.e. with a designated and controlled master repository) to provide for security, availability, and integration with build systems.
- Carefully consider how to provide for security, access control, backup and recovery, and scalability for Git. These are always difficult questions but the answers are more well defined for SVN. Many vendors provide simple security and access control mechanisms for Git, but do not cover replication and scalability. When evaluating solutions, consider how well the solution will support 1,000 developers at several sites working on a hundred repositories.
- The collaboration tools around Git are very popular with developers. Essentially these tools provide code review tightly integrated with build automation and branch management, with a layer of social features. There are more choices available in this area for Git. For SVN, only Phabricator provides a similar workflow.
- The learning curve for Git is shallow initially but becomes very steep very quickly. Strongly consider training for Git administrators and power users who wish to take full advantage of the tool.

- Collaborating beyond the firewall is much easier with Git, as the history in Git is not strongly tied to a particular repository.

Atlassian Integration

Atlassian has announced publicly that they are strongly supporting Git in all of their tools. Stash and Bitbucket are dedicated Git hosting and collaboration tools for on-premise and cloud use. Stash provides effective repository hosting and access control along with a useful 'pull request' collaboration workflow. Like most Atlassian products, it has an attractive user interface and is relatively easy to deploy and configure.

Atlassian is clustering Confluence and JIRA with their Confluence Data Center and JIRA Data Center products. JIRA Data Center includes features beyond clustering, such as an archive.

We expect a Stash/Git clustered product in September 2014 and expect the name to be Stash Data Center. We do not expect the product to handle geo-dispersed clustering in the initial release.

While SVN is still supported in all of their other tools like JIRA and Fisheye (and that support will likely remain indefinitely), the Git support will likely receive more advanced features. For instance, the pull requests offered in Stash are far more powerful than the equivalent code review workflow in Crucible.

Alternatives to Stash include GitHub Enterprise (which is considerably more expensive) and GitLab, an open source project. More distantly related cousins include Gitblit and Phabricator. GitLab in particular is worth investigating, as it is a more open system than Stash. However it is not as tightly integrated into the other Atlassian tools.

Also consider the availability of vendor support and services. Atlassian does not provide field support or consulting, instead relying on a network of partners.

Git Administration Basics

Protocols

Git repositories can be accessed remotely via four protocols.

- The *git* protocol is fast and suitable for anonymous read access.
- The *file* protocol can be used for read and write access but offers no access control. It is most often used for peer-to-peer collaboration. It is also useful to facilitate server-side sharing of object stores between large Git repositories.
- The *ssh* protocol is often used to provide secure authenticated read and write access to Git repositories.
- The *http(s)* protocol offers authenticated read and write access. It is gaining in popularity as it avoids the need for SSH key management and can tie into Apache's library of popular authentication mechanisms.

Access Control

Access control can be performed at three levels as seen in the table below.

Control level	Implementation	Protocols supported
Repository level (read/write)	SSH keys or HTTP(S) security	SSH, HTTP(S)
File or directory level (write)	Pre-receive or update hooks	SSH, HTTP(S)
Branch/tag (ref) level (write)	Pre-receive or update hooks	SSH, HTTP(S)

As you can see, it is generally not feasible to perform read access checks at anything other than the repository level. It may be feasible to encrypt data in a Git repository and automatically decrypt it for selected users using smudge and clean filters.

Repository Structure

Git repositories are designed to contain a single project or component. Most Git operations like branching and merging happen at the workspace, not file, level. Additionally it would be difficult to use more than one mainline in a Git repository.

On a related note, many Git operations like the *status* command reflect information about every file in a repository. This puts a practical limit on the size of individual files and the repository as a whole.

For both of these reasons, larger projects are generally split into several Git repositories and managed independently. Dependencies are tracked at the source level using submodules or subtrees, or at the binary level using tools like Maven and Nexus.

This design can come as a surprise to those used to large monolithic SCM repositories. However, using several smaller Git repositories offers some advantages, and using a true dependency management tool (with strong versioning applied to the declarative file) is considered a best practice.

Auditing

Audit logs should contain a record of each read or write access to a Git repository. Write operations can be captured using the post-receive hook, while read operations must be handled by the access control system. In practice this means that auditing requires a combination of SSH or Apache logs and the output of an auditing hook.

Backup Strategies

Like any critical system Git requires a layered backup strategy. Note that clones that exist on end user workstations are generally not sufficient as they do not necessarily contain a clean and full set of history, and will always lack hooks and access control data.

Starting from the last line of defense, full file system copies of Git repositories should be made on a regular basis. Tools like *rsync*, advanced storage level backups, or even shared network storage can all be used effectively.

The next level of backups may make use of Git bundles, which allow for periodic transfer of new transactions to remote repositories.

Finally, read-only Git mirrors can provide more of a hot spare capability.

Maintenance

Git performs some routine maintenance (e.g. garbage collection) automatically. Git administrators may occasionally be called upon to use the reflog to undo a bad rebase operation, but that's usually end user assistance. In rare cases a Git administrator will need to use the *filter-branch* technique to permanently remove data from Git, at the expense of forcing each end user to clone a fresh copy of the repository.

Performance monitoring should be done routinely. Besides hardware statistics like RAM and CPU usage, monitor:

- The total size of each repository. Repositories greater than 1 GB in size may be unwieldy for end users.
- The presence of any individual files larger than 100 MB.
- The total number of files in the repository. Repositories with more than 100,000 files may cause performance difficulties.
- The total depth of history. A repository with more than 15,000 commits may take a long time to clone.

Repository Configuration

Almost every Git bare repository should be configured to deny forced updates (non-fast-forward pushes). Otherwise the permanent record of a Git repository may be damaged.

Optional configuration may disallow deleting branches and tags, or perform an integrity check on every push.

Hooks are quite useful for enforcing policy, but note that an administrator should only be concerned with server-side hooks. Client-side hooks can provide guidance, but only server-side hooks can enforce policy. The most useful server-side hooks are:

- *Pre-receive* and *update* hooks can perform access control checks, ensure that commit messages contain bug numbers, and otherwise choose to allow or deny writes.
- *Post-receive* hooks can response to writes and send emails, trigger builds, and take other actions.

Git Workflows

In the following discussion, it is useful to understand a few popular Git workflows.

Fork and pull

In this model, a developer will fork (clone) a Git repository and work independently on their own server-side copy. When the developer has a change ready to contribute, she will ask the upstream maintainers to pull her changes into the original repository.

This model works very well in open source projects, where contributors may not even know one another and the upstream maintainers need to review any contributions.

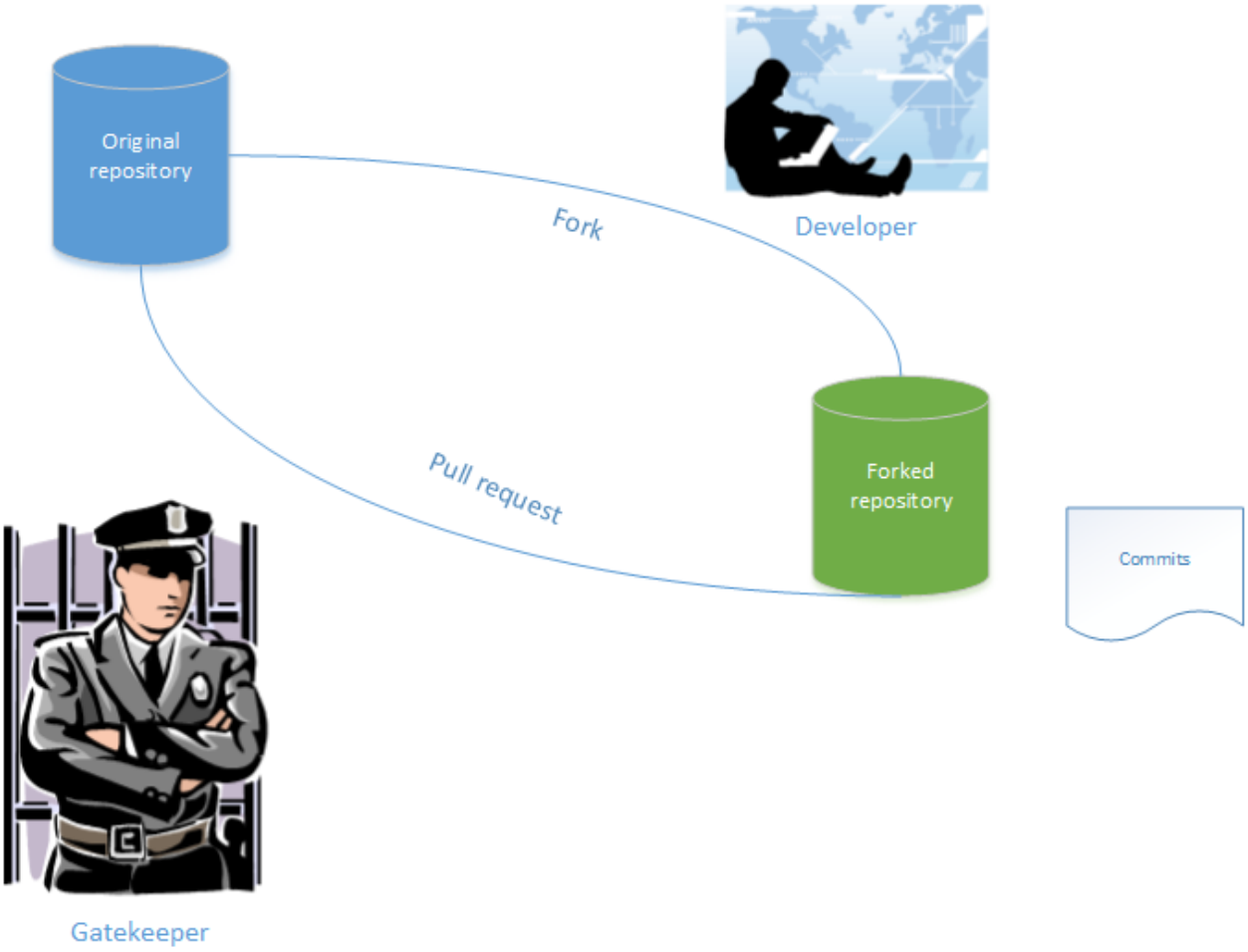


FIGURE 2: FORK AND PULL WORKFLOW

Topic branches

In this model new branches are made for each task (topic) and are shared on occasion. When changes are approved they are merged to the mainline (master) branch.

This model suits many small business teams, as they are able to collaborate in a single shared repository yet still isolate new work. Functionally it is very similar to fork-and-pull.

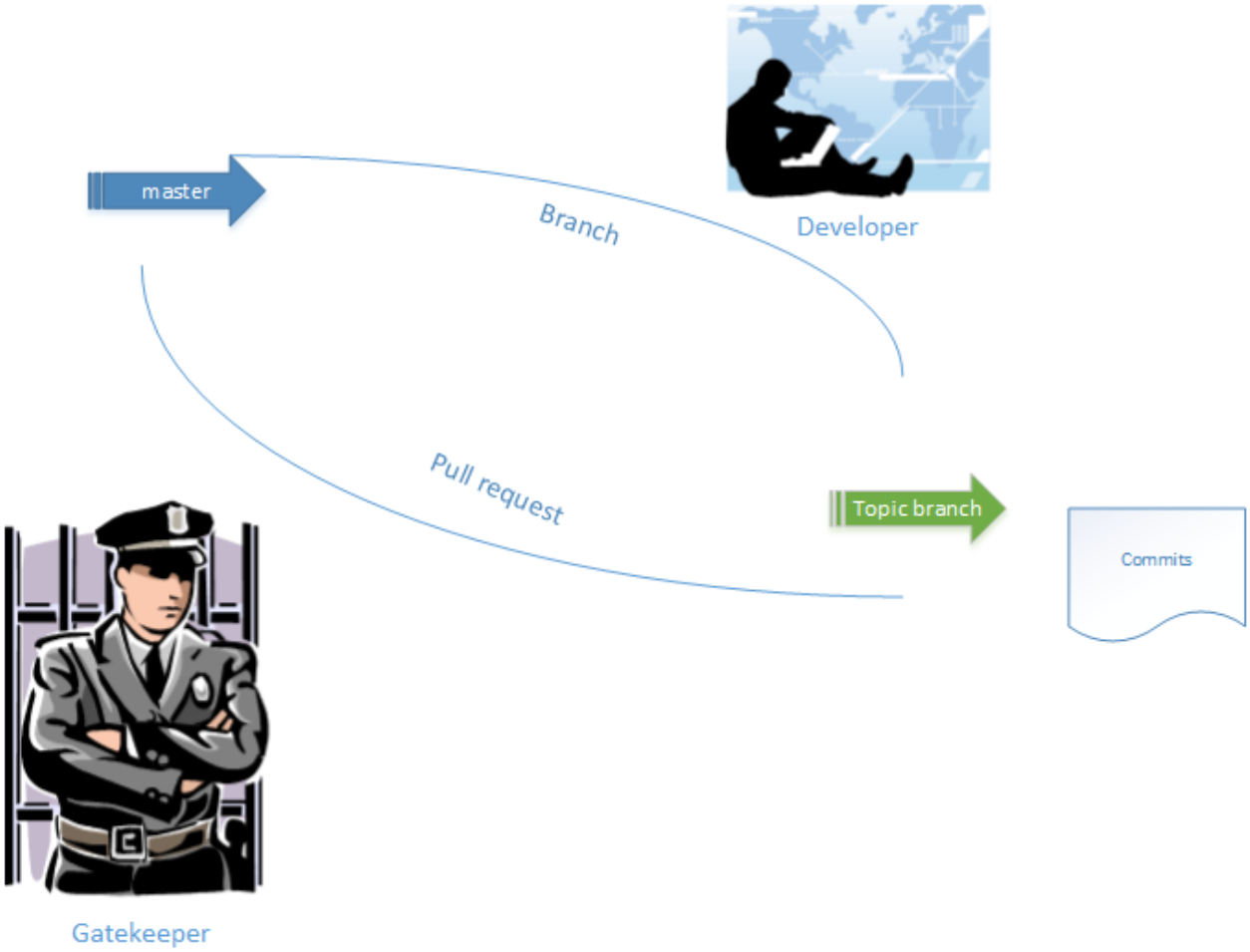


FIGURE 3: TOPIC BRANCH WORKFLOW

Mainline model

In this model most work is committed directly to the mainline (master) branch. There are few, if any, long lived branches less stable than the mainline. Long lived branches are sometimes used for release maintenance. Developers are encouraged to commit to the mainline frequently, perhaps daily. Local branches and stashes can be used for pre-flight review and build, but are not promoted to the shared repository.

This model is preferred by adherents of continuous integration and continuous delivery, and has proven to scale to large teams working in enterprise settings.

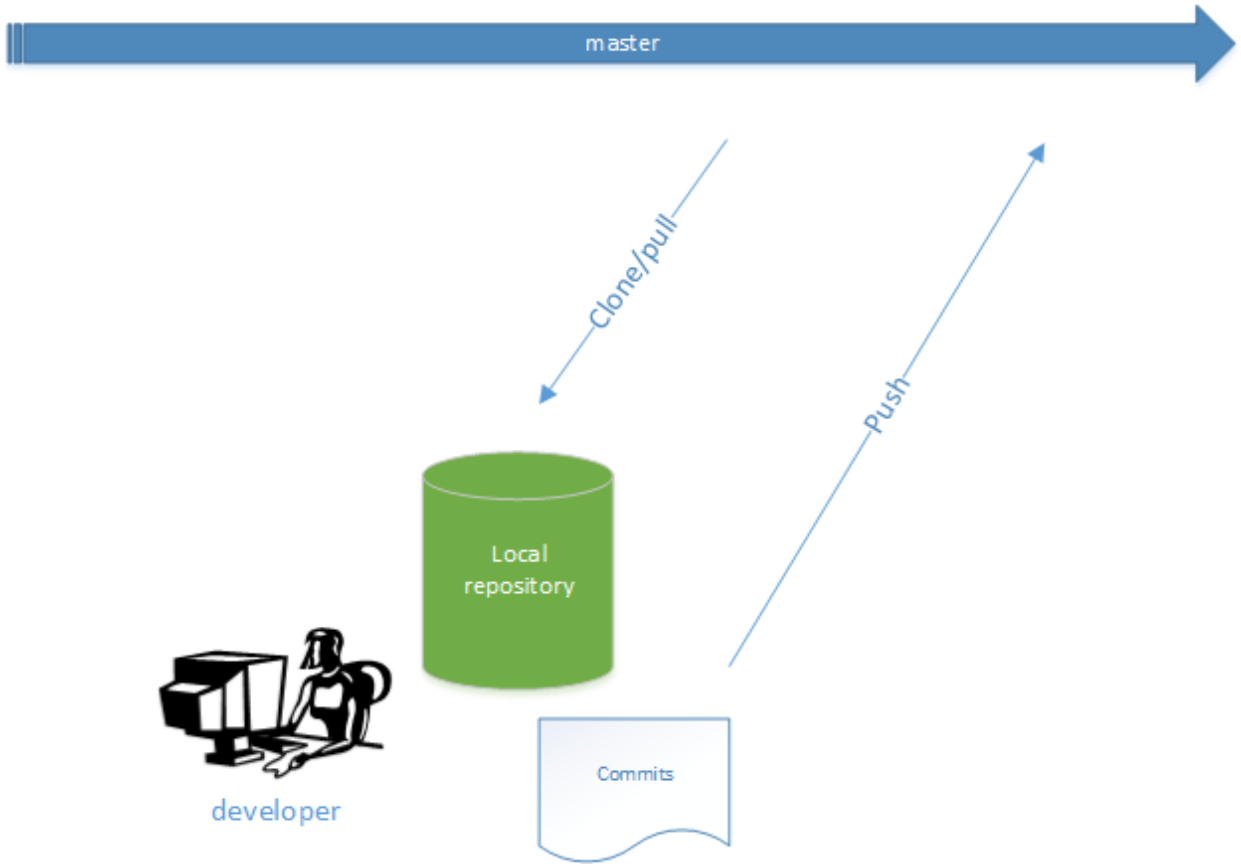


FIGURE 4: MAINLINE WORKFLOW

Git Flow

"Git Flow" is a popular model developed by Vincent Driessen[1]. It recommends a long lived development branch containing work-in-progress, a stable mainline, and topic, hot fix, and release branches as necessary. It is somewhat similar to a mainline model with long lived integration branches and topic branches.

This model violates some of the precepts of continuous integration. Notably, work may be left on the development branch or topic branches, and not integrated with the latest changes on the mainline, for a long period of time. Nonetheless this model is often a comfortable transition for teams new to Git and continuous integration.

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

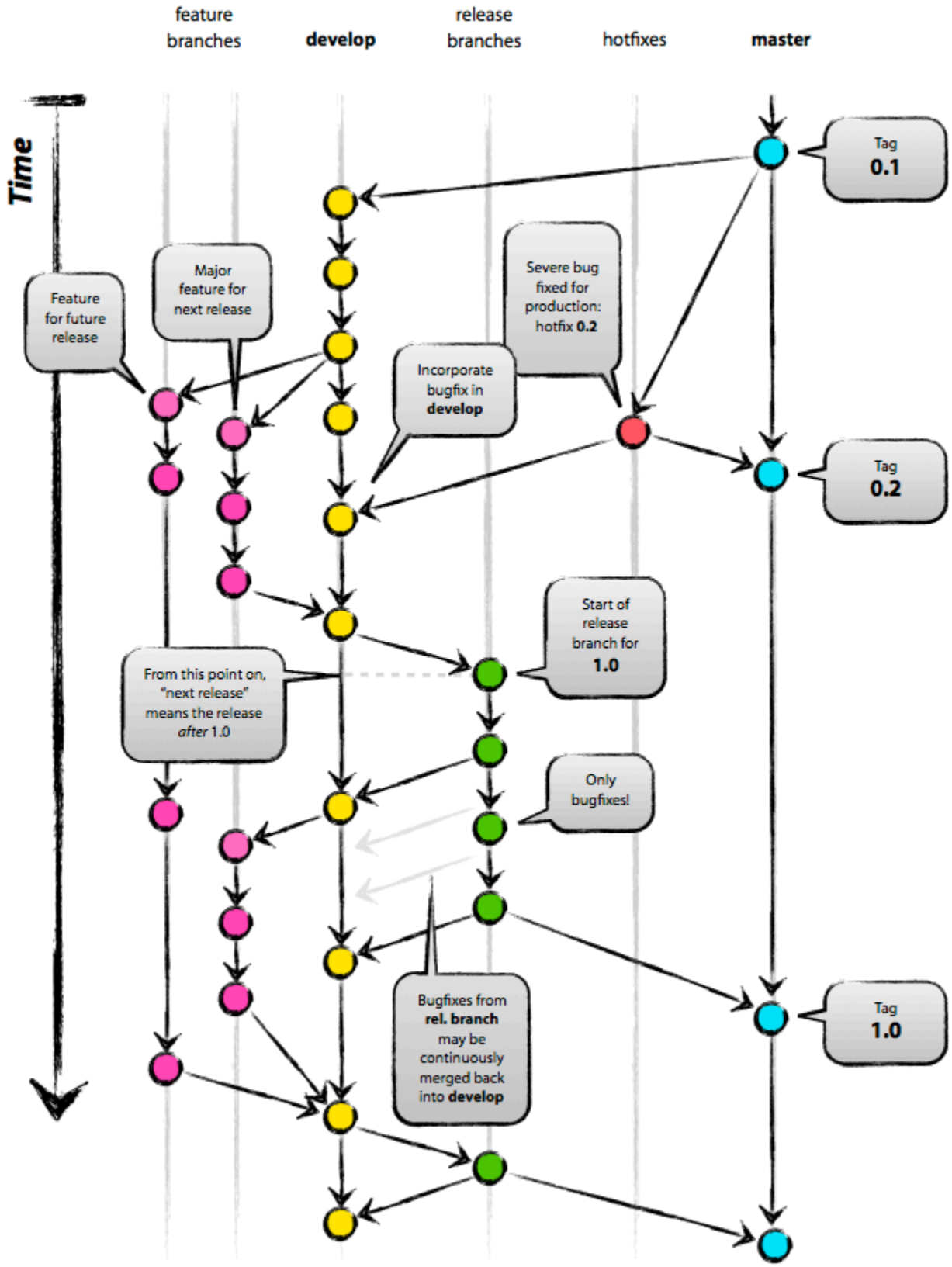


FIGURE 5: GIT FLOW

Git Management Consoles

Git management consoles serve two distinct purposes. First, they provide basic Git repository management: creating repositories, managing access control, and in some cases coordinating backups and replication. Second, they provide social and collaboration features: activity streams, projects, code review, merge requests, and integration with build and test systems.

This document reviews several Git management consoles, highlighting their relative strengths and weaknesses. There is currently no 'best' choice; rather, the selection process should be driven by your business priorities. However, after reviewing all aspects of Git management consoles and their trajectory over the past year, Stash is now becoming the leader.

Stash is reinforced as the Git management console of choice when comparing tertiary, but important aspects such as a well integrated ALM suite, community adoption, roadmap, available integrations and consulting, price, performance, and support. Perhaps as importantly, Stash's flexible architecture lets it subsume the best features of other tools like Gerrit.

Overview

Of the six consoles reviewed in this document, GitHub and Stash are competing directly as standalone consoles focused on social coding and collaboration. Gerrit is similar in spirit to GitHub and Stash but takes a much different approach. Git Fusion and TeamForge take a more enterprise spin on Git management.

GitHub is far and away the trend leader among the six vendors according to a Google Trend analysis[2]:

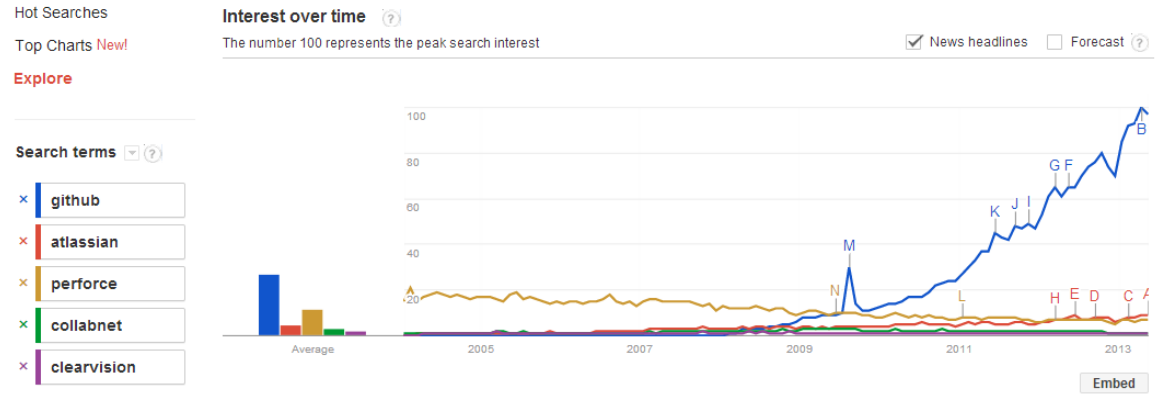


FIGURE 6: GIT MANAGEMENT VENDOR TRENDS

GitHub

Delivery model	SaaS, on-premise

2 http://www.google.com/trends/explore?q=github#q=github%2C%20atlassian%2C%20perforce%2C%20collabnet%2C%20clearvision&cmpt=q

Unique strengths	• Social coding (e.g. network graph) • Extensive integration set • Large user community for public projects
Average capabilities	• Basic Git repository management • Simple access control • Task review and promotion
Weaknesses	• Enterprise administration • Complex projects • Poor on-premise deployment model • No data replication
Pricing	• $5,000 annually per 20 users (on-premise) • Business plans from $25-$200/month (hosted)

Capsule Review

GitHub blazed the trail for social coding and has built a dynamic online community for their public hosting site. Their collaboration and codeline tools are impressive, while their basic Git repository management features satisfactory. GitHub Enterprise has seen early success, being adopted by small groups in large companies. However, GitHub Enterprise faces resistance from enterprise IT departments due to its closed nature and lack of administration tools.

Collaboration

GitHub excels in this area. It offers built-in commenting and review tools, simple task and reporting capabilities, a wiki with offline access, notifications, and Instant Message (IM) and discussion capabilities. Repository and branch browsing are also included. Few of these tools are best-of-breed, but the value lies in the simplicity and accessibility of the total package. One area where GitHub does shine is the network graph (Figure 7), which shows relationships and pending work between related repositories.

Codeline Workflows

GitHub made merge requests famous. A merge request is a formal request, managed in a ticket, for someone with write access to review and accept a contribution to a repository.

A merge request (also known as a pull request) is used in either the fork-and-pull workflow or the topic branch workflow. It is especially valuable in the fork-and-pull model, as it lets the owner of the other repository know that there is an outstanding change to review and merge. In the topic branch workflow it is still useful as a way to enforce a gatekeeper on the master branch.

As noted earlier, the mainline workflow is often preferable in a large enterprise. GitHub supports this model as well, and a merge request can still be used as a code review mechanism.

Integrations

GitHub offers a simple API and several ways to tie external systems into the system. For example, it is easy to invoke a continuous integration (CI) process as part of a review, or deploy

a commit to a web server for testing. As with the collaboration tools, the approach emphasizes simplicity and openness over tighter and heavier integrations.

Administration

GitHub provides simple and effective tools for Git repository management. Creating repositories, managing teams, and setting up simple read/write/admin access control is easy. Access control is managed via SSH keys or, less commonly, by user names and passwords for HTTPS access. LDAP and corporate email integration are available.

GitHub is organized around projects, each containing a single repository and related tools like the wiki. The access control setting affects the entire project.

A consistent criticism of GitHub Enterprise is the closed nature of the application. It is shipped as a virtual machine (VM) with very limited access to the internals; it is not possible to access the VM via SSH, for example. That means that administrators are not able to diagnose performance problems, install monitoring tools, or implement flexible backup and recovery strategies. If there is a performance issue, you can reboot the VM; if you want to perform a backup, you can snapshot the VM.

Screenshots

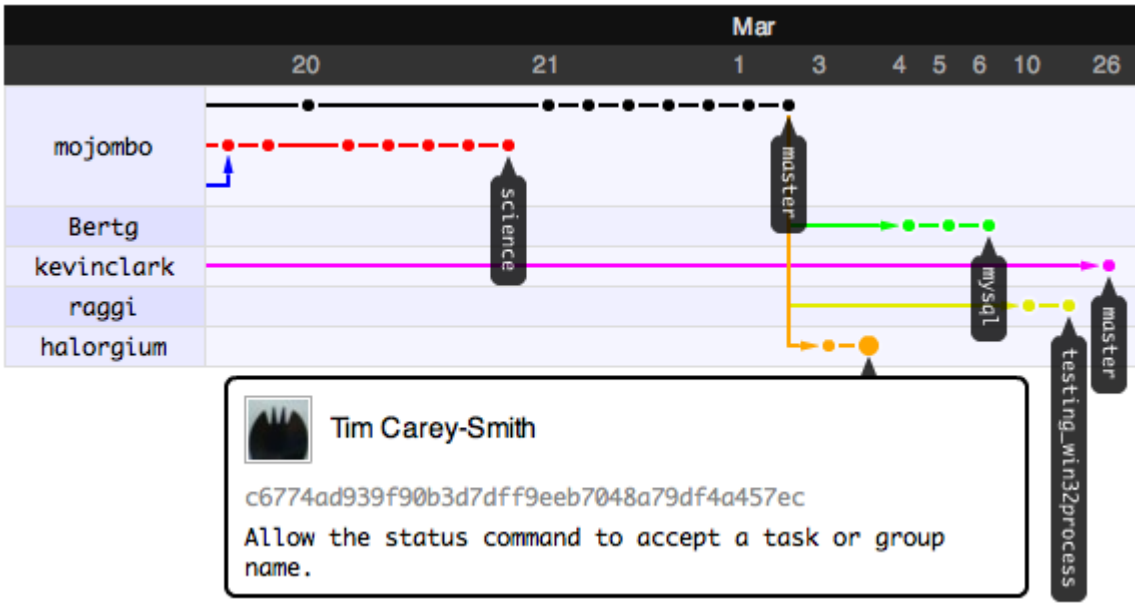
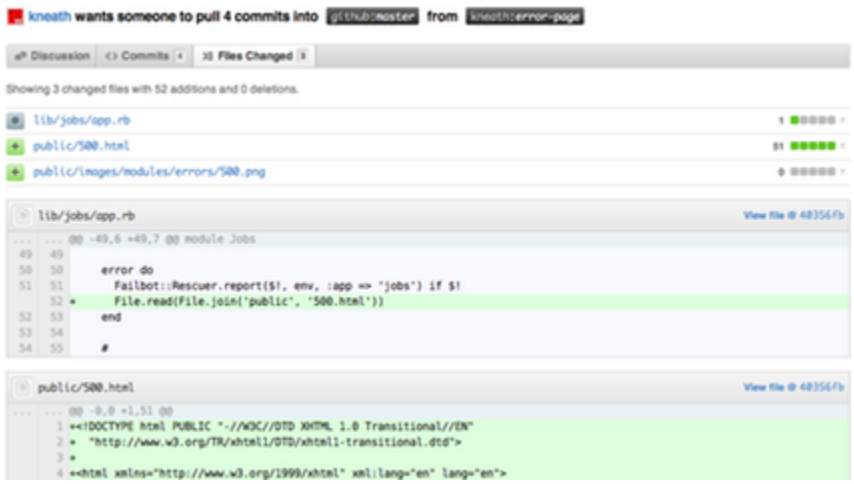


FIGURE 7: GITHUB NETWORK GRAPH



Stash

Delivery model	On-premise (Bitbucket is a more standalone SaaS service also owned by Atlassian)
Unique strengths	• More flexible project model with support for several repositories • Strong integration with other Atlassian products and plugin framework (e.g. JIRA, Crowd SSO and two factor PIV and CAC) • Simple administration for on-premise deployment • Personal repositories • Customized workflows (can emulate Gerrit)
Average capabilities	• Basic Git repository management • Simple access control • Task review and promotion (pull requests)
Weaknesses	• Complex projects • No data replication
Pricing	• $10 for 5 users, rising to $12,000 for 500 users.

Capsule Review

Stash is a strong competitor to GitHub. It offers a strong set of codeline workflow and collaboration tools, and integrates with Atlassian's other tools such as JIRA for a more complete offering. Stash is evolving rapidly and Atlassian is heavily committed to a Git strategy in their developer tools.

Stash is more familiar to enterprise IT departments, as the deployment and licensing model is similar to other Atlassian products. However, like all Atlassian tools, there is no out-of-the-box

replication solution, and scalability requires a significant hardware investment. Stash also requires integrating with other Atlassian tools in order to match GitHub's total feature set. However, these challenges are largely understood by Atlassian's large enterprise user community, and will not pose a particular burden.

Collaboration

Stash offers simple collaboration tools like '@mentions' and Markdown support. It does not have the rich set of collaboration tools found in GitHub, although some of them can be added by integrating with other Atlassian products.

Stash has also staked a claim at more comprehensive enterprise repository management, by virtue of allowing projects to contain multiple repositories. That implies that teams working on related components of a larger project can still share pull requests and JIRA ticket data. This data model is more conducive to enterprise work than the 'project = repository' model found in GitHub.

Codeline Workflows

Stash originally supported merge requests for the topic branch workflow, and it now supports the fork-and-pull workflow. Stash can also support the mainline model, particularly when used in conjunction with Atlassian's build system, Bamboo. Bamboo offers a compelling feature set for continuous delivery, and features like template build plans provide pre-flight build capability for Git developers.

With appropriate configuration and a plugin offered by Go2Group, Stash can closely emulate Gerrit's workflow, offering an interesting option for those who need Gerrit's power but are put off by its complexity.

Stash also offers some recognition of submodules in a repository and link to a submodule hosted in another project.

Integrations

Stash integrates well with other Atlassian products like JIRA and Bamboo. Bamboo in particular offers powerful CI features for Stash repositories. Bamboo can apply build plans for the master branch to new topic branches, providing something similar to a pre-flight capability for a mainline workflow.

JIRA provides more complete ticketing and Agile project management capabilities that can be found in GitHub. Teams looking for a balance of social coding with powerful project management may be swayed by the combination of Stash and JIRA.

Third party tools can be integrated using Stash's API, Git hooks, and like other Atlassian products Stash has a strong plugin community. The maturity of the Atlassian plugin ecosystem drives development of more powerful and well supported extensions than found in many products.

Administration

Stash is built around projects that can contain several repositories. Like GitHub, managing repositories, teams, and access control is straightforward. Stash extends access control down to the branch level.

Access control is managed via SSH keys for repositories and user names and passwords for the web site. Atlassian has invested heavily in supporting several types of user directories, including internal directories, directories managed by their other applications, and corporate (LDAP) directories. All of these are available with Stash, along with a web SSO capability.

Stash is a web app and can be deployed and managed fairly easily. Performance at scale will be a concern, and Stash may require a powerful machine to support a busy site. Backup and recovery is handled with database and file system backup tools.

Screenshots

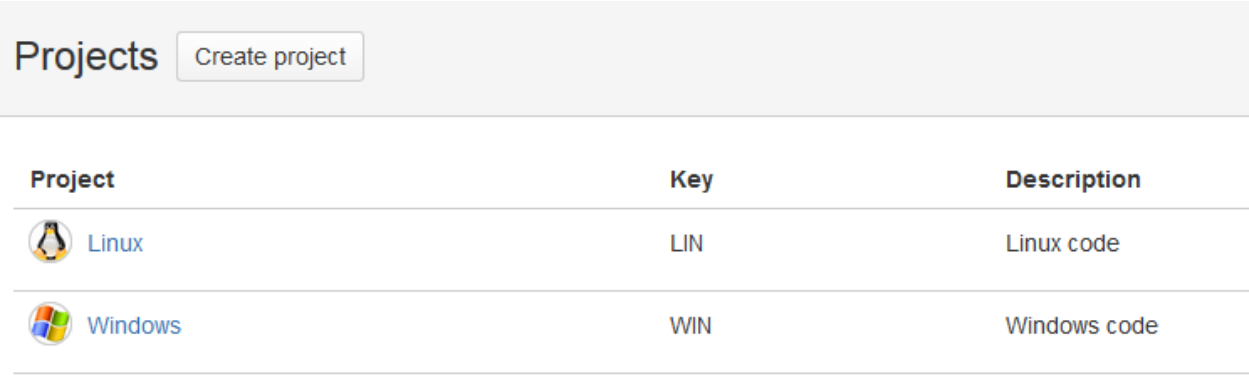


FIGURE 9: STASH DASHBOARD AND PROJECTS

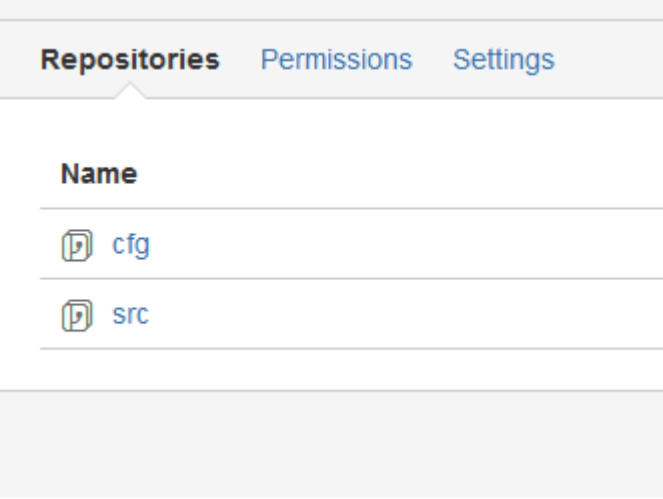


FIGURE 10: REPOSITORIES IN A PROJECT

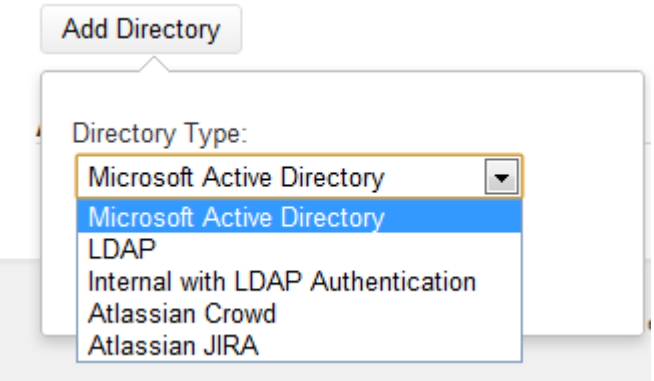


FIGURE 11: STASH DIRECTORY AUTHENTICATION OPTIONS

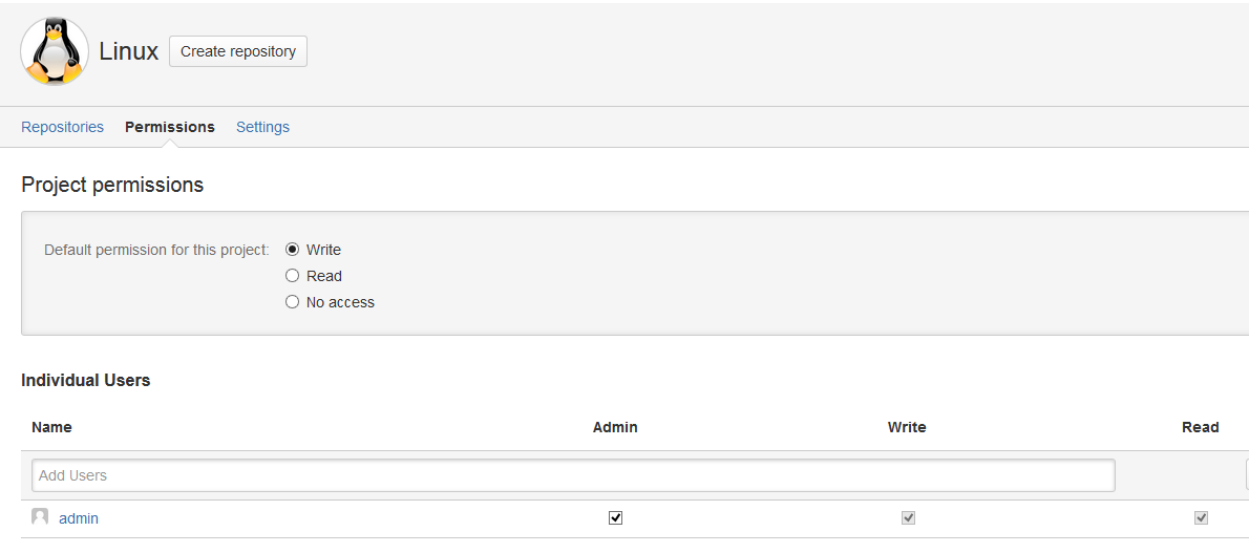


FIGURE 12: STASH PROJECT ACCESS CONTROL

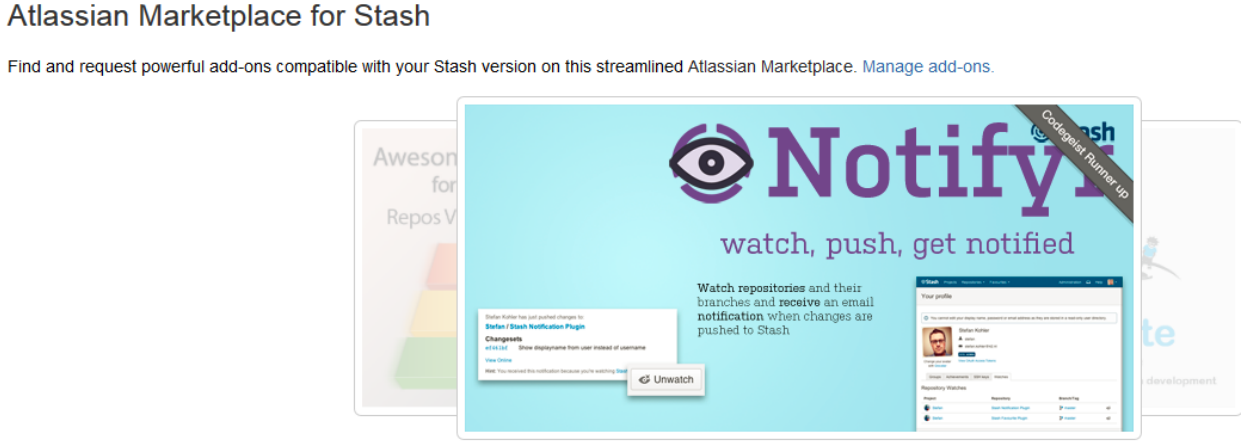


FIGURE 13: STASH PLUGIN MARKETPLACE

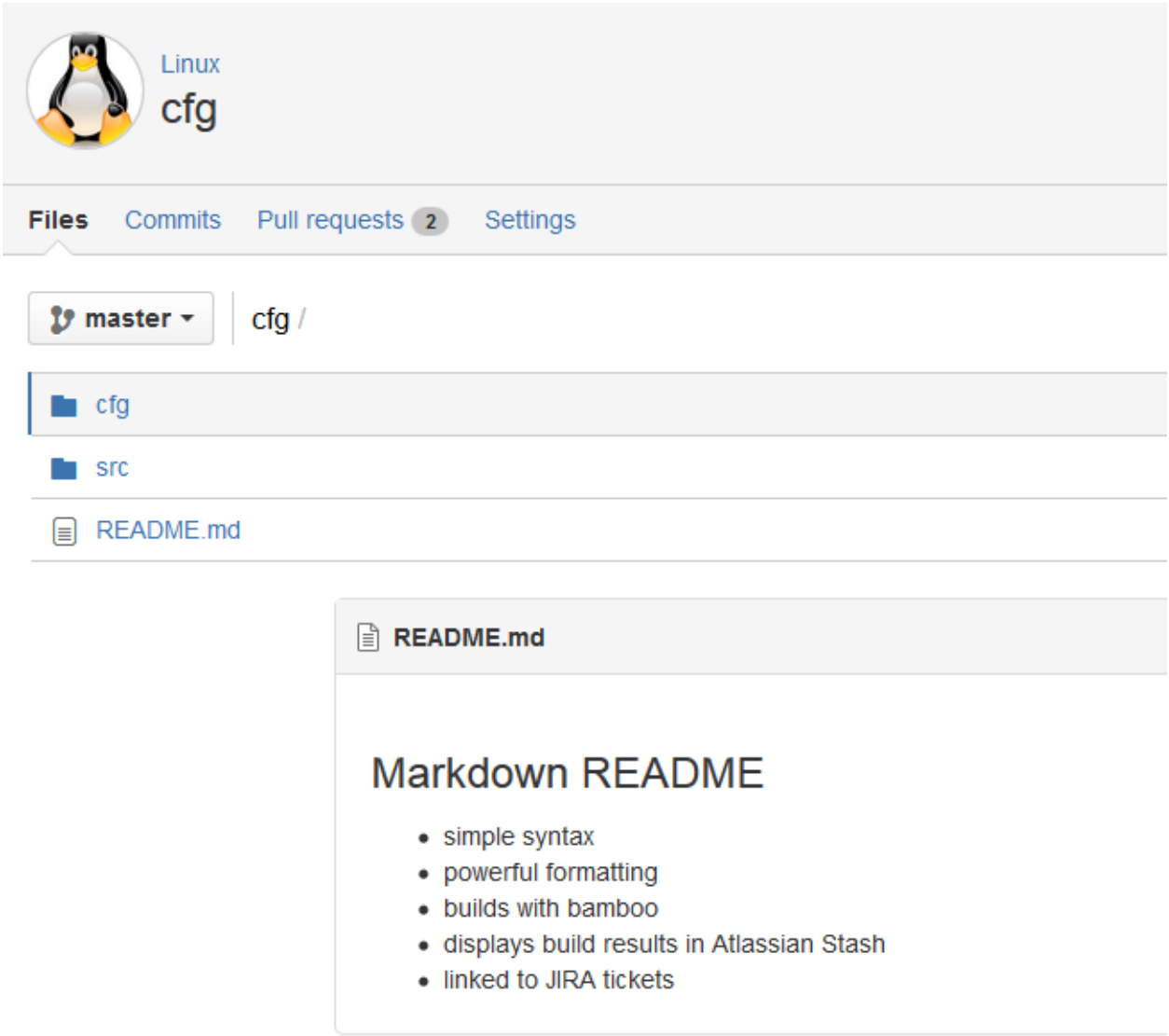


FIGURE 14: REPOSITORY DASHBOARD

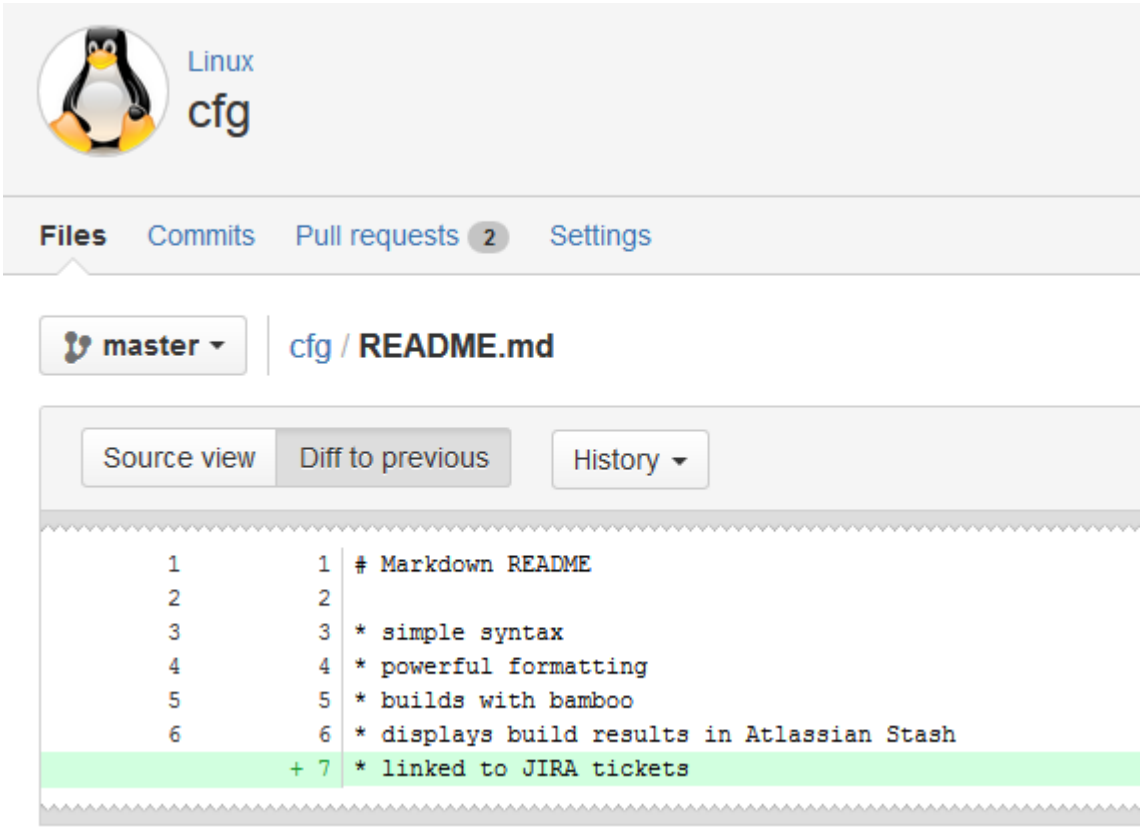


FIGURE 15: REPOSITORY BROWSER

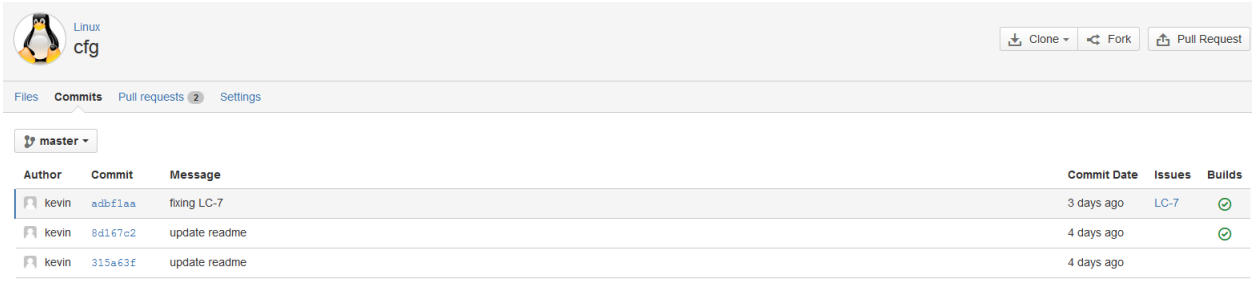


FIGURE 16: COMMITS WITH TICKETS AND BUILDS

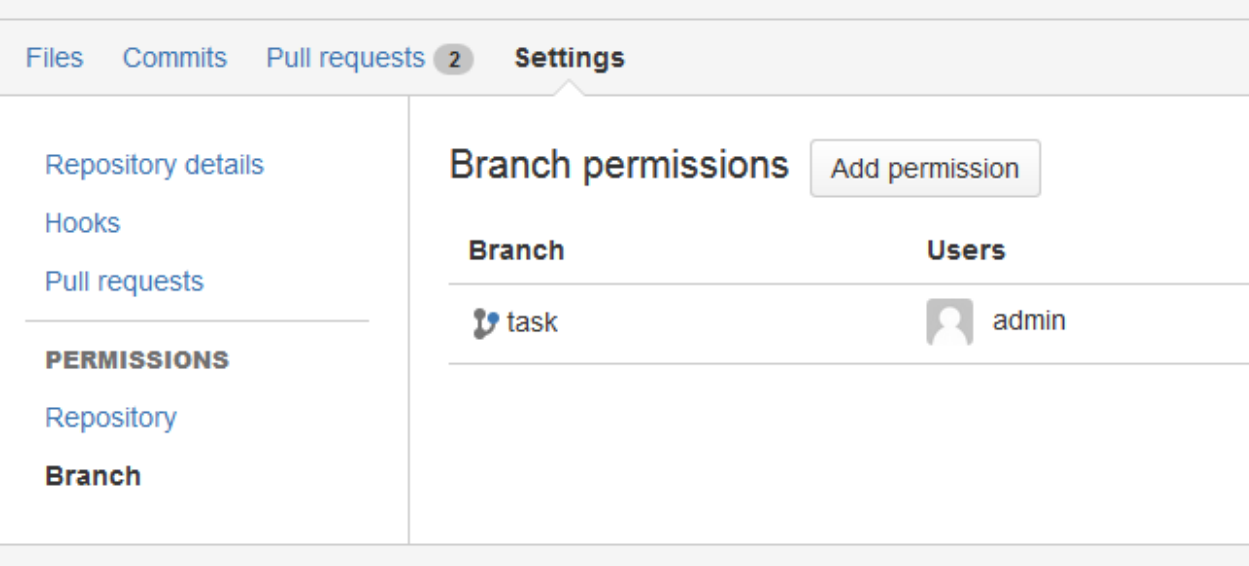


FIGURE 17: BRANCH PERMISSIONS

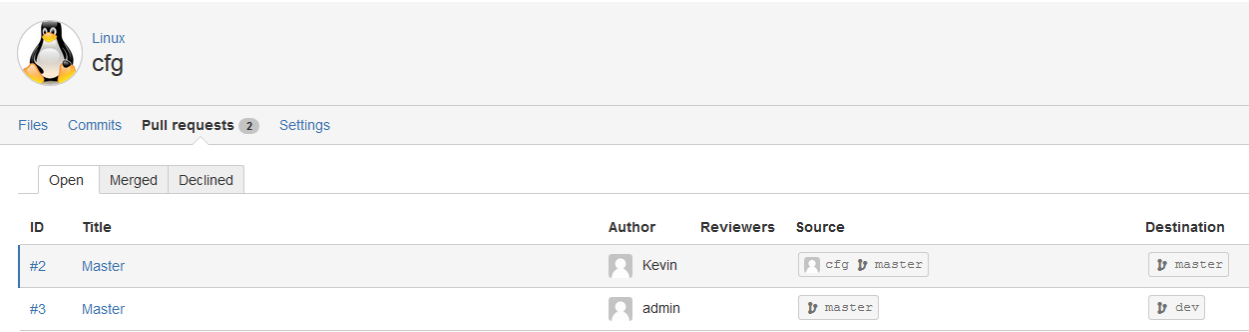


FIGURE 18: PULL REQUESTS

Delivery model	On-premise (virtual appliance available)
Unique strengths	• Very flexible project model with support for complex projects and large data sets backed by Perforce repository • Fine-grained permission system, strong auditing and compliance features • Administered as part of an enterprise Perforce deployment • Advanced data replication • Enables collaboration with cross functional teams (e.g. video game artists)

Average capabilities	• Git-specific user management (delegated from Perforce access control)
Weaknesses	• Relatively new product, still maturing • Social coding and collaboration features delegated to other new products • Setup and administration are complex
Pricing	• Perforce pricing starts at $25 per user per month • Swarm pricing starts at $5 per user per month

FIGURE 19: PULL REQUEST IN PROGRESS

Perforce Git Fusion

Capsule Review

Perforce is a centralized SCM system with a long pedigree and an impressive customer base. It has a flexible architecture that lends itself to complicated projects and non-traditional data such as media assets for video games.

Git Fusion is a middleware product that exposes slices of a Perforce repository as one or several Git repositories. At the cost of speed and complexity, Git Fusion allows dynamic remapping of data into new Git repositories. For example, a mobile development team could use two Git repositories that share the artwork but have separate front ends for iOS and Android.

Although Git Fusion has an interesting value proposition, it remains to be seen whether Perforce data can be seamlessly translated into Git data for all use cases. Additionally, Perforce does not yet offer a social and collaborative front end, although one is being introduced in Q3 2013. Finally, administering Git Fusion requires expert knowledge of both Perforce and Git – an unusual skill set.

Collaboration

Perforce offers no native client applications or collaboration tools for Git. Perforce is introducing a social and collaborative web application this year, but the first iteration does not support Git Fusion.

Codeline Workflows

Perforce is built around the mainline model and will support any Git workflow. However, it does not yet offer any additional support like merge requests.

Git Fusion translates Git activities into equivalent Perforce operations, and vice versa. Be aware that the translation is sometimes (of necessity) imperfect. For example, Perforce supports the idea of a partial merge, where 3 out of 5 files in a change set are merged and the others ignored. Perforce will record the merge history at the file level. But, Git does not support partial merges, and so this activity is recorded as a regular merge in Git.

Also note that not all Git data structures are supported in Git Fusion. Tags, submodules, and subtrees are not handled.

Integrations

Perforce has integrations with over 50 third party tools in several ALM categories. However, these integrations are targeted at native Perforce data.

In terms of integrating with Git Fusion, most tools that support Git repositories will 'just work' with Git Fusion. For example, the Phabricator code review system can operate with Git Fusion repositories. There may be limitations in some cases and these third party tools are not optimized for use with Git Fusion.

Administration

Perforce administration is primarily done on the command line. It offers a Perforce-centric administration GUI but only a subset of Perforce administration is accessible.

Perforce offers no visual tools for Git Fusion administration. Setting up repositories requires advanced knowledge of Perforce data structures and how they translate to Git Fusion functionality. Likewise, Git Fusion user and access control management requires knowledge of Perforce user and access control settings, and is not straightforward. Notably, the powerful Perforce access control model is partially and inconsistently exposed in Git Fusion. The same user may experience different access control settings when accessing the same data through Perforce versus Git Fusion.

Perforce supports SSH access but does not support other Git protocols. It does not offer any LDAP integration beyond password checking.

Git Fusion is a set of Python scripts with a long list of dependencies. Installation and maintenance is non-trivial, although a virtual appliance format is provided.

An interesting alternative is simply using Perforce as a backup and replication solution for another Git management console. For example, if you configure Git Fusion to allow anonymous (non-licensed) pushes and automatic branch creation, you can configure Gerrit to replicate all commits to Git Fusion. A similar strategy could be employed with other tools if you use Git hooks to trigger the replication.

Performance

Git Fusion performs extensive data translation during some operations. Accordingly, performance is sometimes much worse than a comparable native Git system. Some operations experience performance degradation of one or more orders of magnitude.

Screenshots

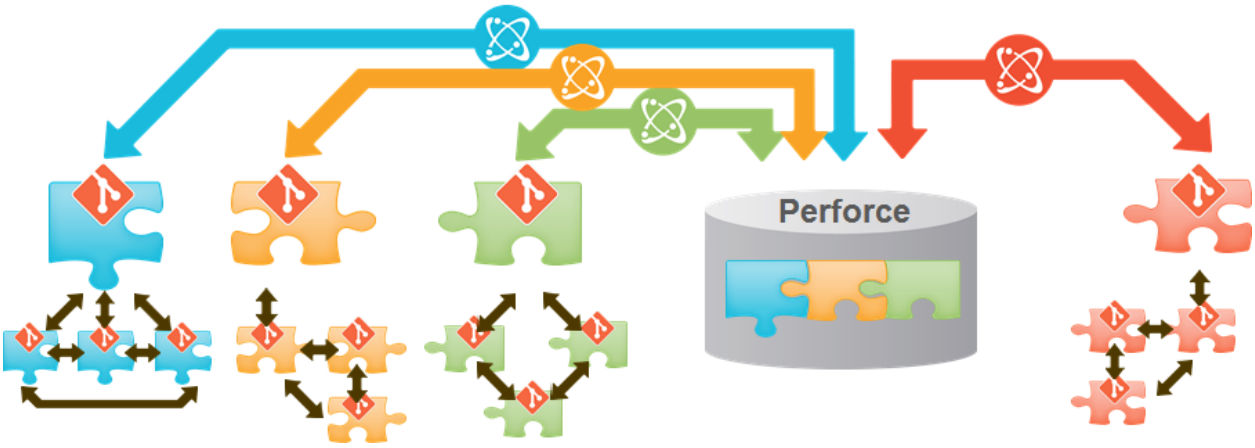


FIGURE 20: GIT FUSION "REPOSITORY REMAPPING"

CollabNet TeamForge

Delivery model	SaaS, on-premise
Unique strengths	• Full ALM stack • Capable Jenkins integration • Embeds Gerrit
Average capabilities	• Repository management • Access control
Weaknesses	• Dated UI • Traditional ALM focus (legacy workflows)
Pricing	• On-premise: $5,000 annually for 25 users • Hosted: $25,000 annually for 25 users • Additional users: $295 annually

Capsule Review

TeamForge is an ALM stack consisting of version control, defect tracking, document handling, collaboration features, and project management. The interface is focused on projects. Repository management and browsing capabilities are simple and straightforward.

TeamForge will be most appropriate for teams seeking a traditional ALM project management system. Aside from embedding Gerrit it does not offer the modern Git tools and workflows seen in GitHub and Stash.

The user interface is dated and will not be as attractive to developers as GitHub or Stash.

Collaboration

TeamForge offers a simple set of collaboration features including activity feeds, forums, wiki, and watch lists. It is geared more towards traditional project management, and its features in that area include project groups, project templates, release management, and search. Project can be categorized based on development process (Agile or waterfall) and development language. Projects can use a simple wiki to create home pages.

The ticketing system tracks several types of records including defects, stories, tasks, and tests.

It also has simple reporting capabilities, both graphical and ad-hoc database queries. Reports include Agile charts (e.g. burndown report) and simple project charts (e.g. defects grouped by priority).

Codeline Workflows

Each project can contain several repositories of various types. Again, TeamForge's emphasis is on projects, not repositories.

There is no special support for any of the Git workflows. That being said, any of the Git workflows can be used with TeamForge. It really just provides bare repository hosting; most of the workflow is subsumed into the ALM tools. The advantage to that approach is that consistent workflow and policies can be applied across several repositories hosted in Git and Subversion.

Alternatively, since TeamForge embeds Gerrit, the Gerrit workflow can be used.

Integrations

TeamForge has an extension mechanism which can be used to provide custom event handlers and other capabilities. It offers a SOAP API.

TeamForge supplies a powerful Jenkins integration module that goes beyond merely triggering builds. This module provides SSO and integrated management of releases, artifacts, and documents.

Other notable integrations include Black Duck Code Sight.

Administration

TeamForge is distributed as a virtual appliance for on-premise use.

TeamForge is a web application built on JBoss. It encapsulates Subversion, CVS, Git, Gerrit, and its own ALM tools.

User management has a traditional user-group-role structure. Access control applies at the project level.

Screenshots

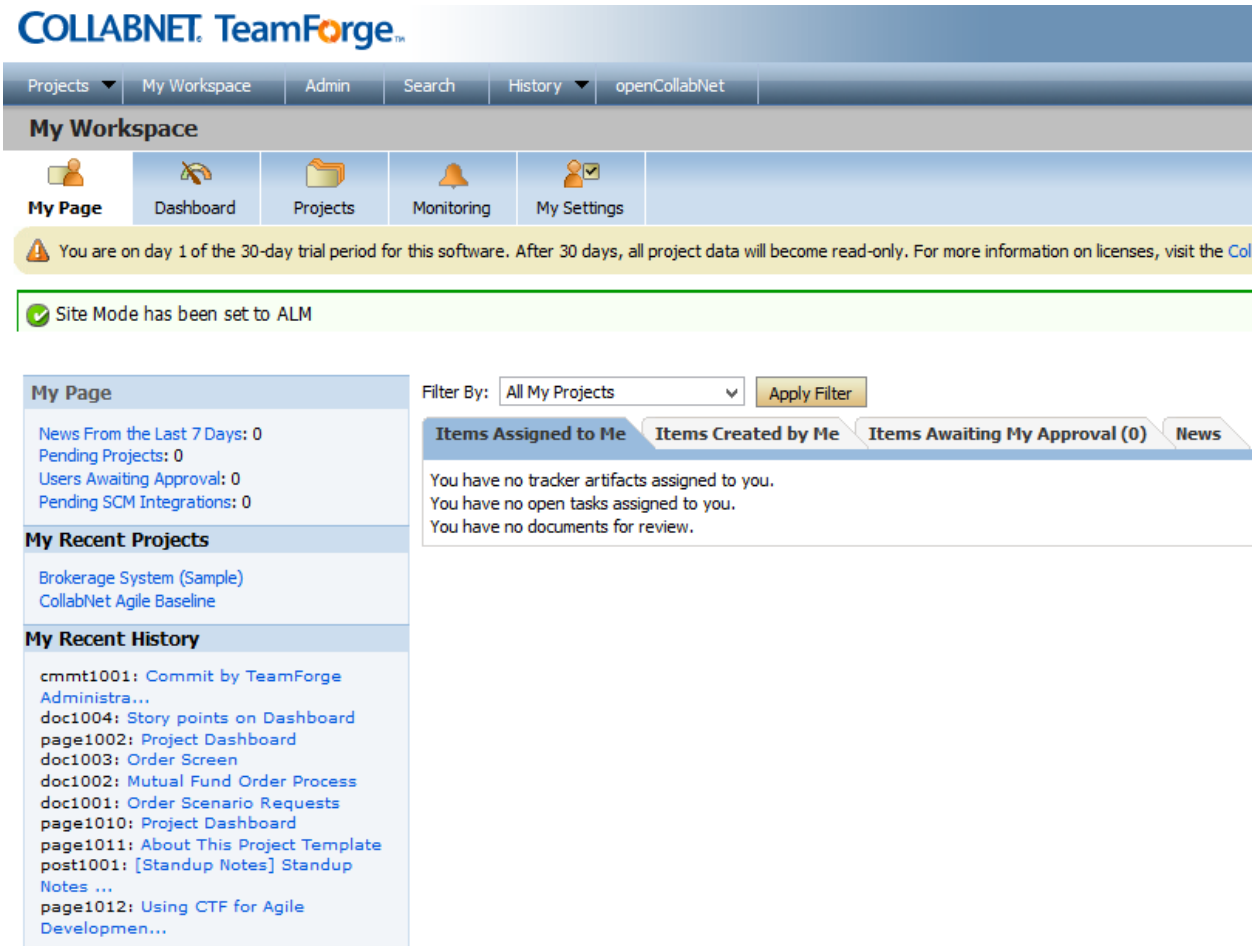


FIGURE 21: PERSONAL DASHBOARD

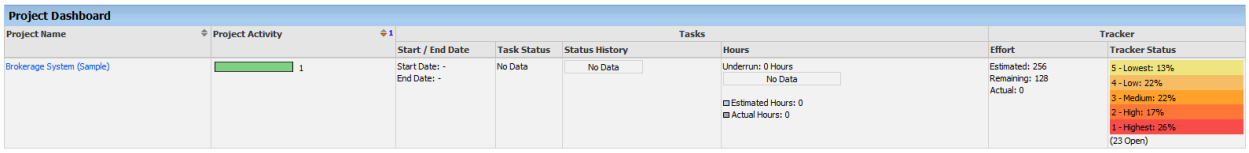


FIGURE 22: PROJECT DASHBOARD

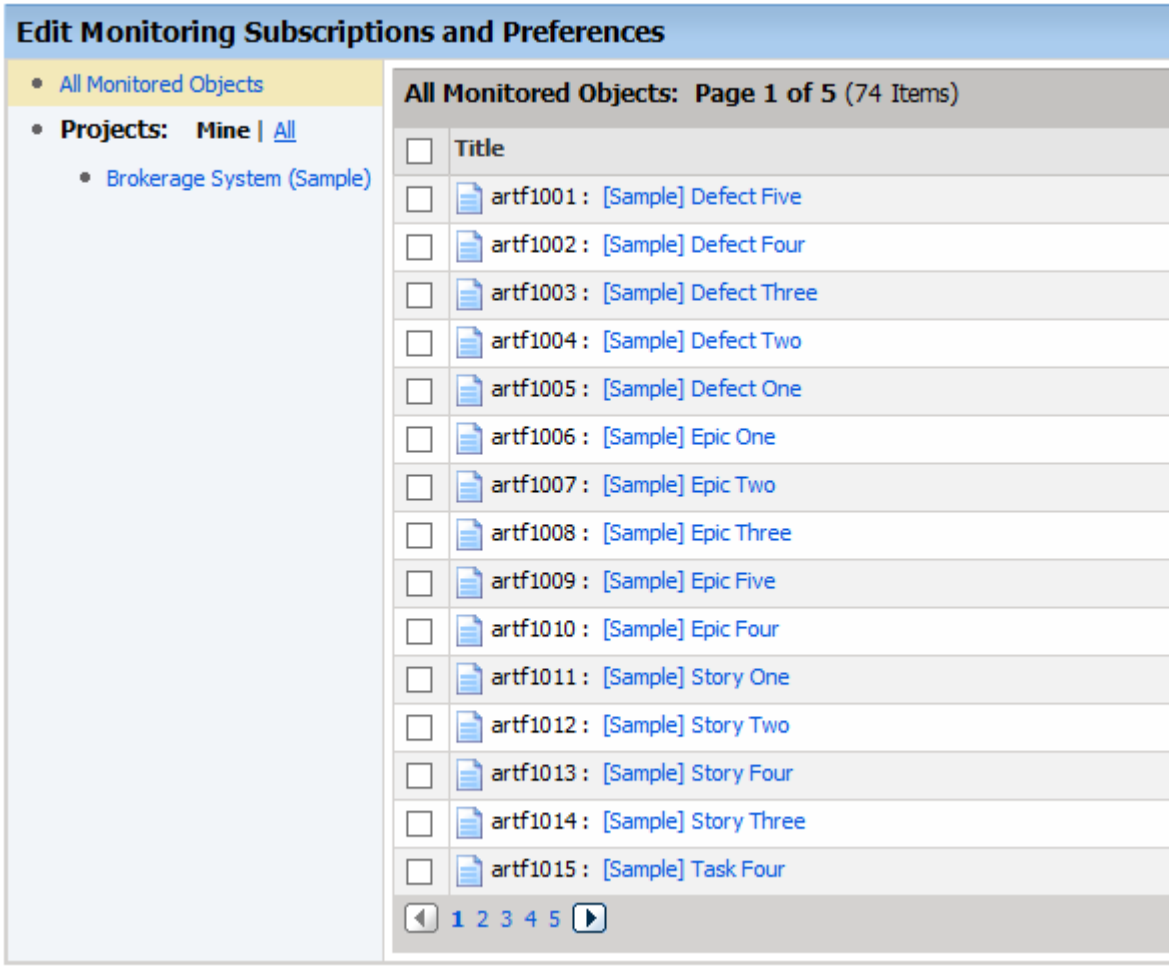


FIGURE 23: WATCH LIST

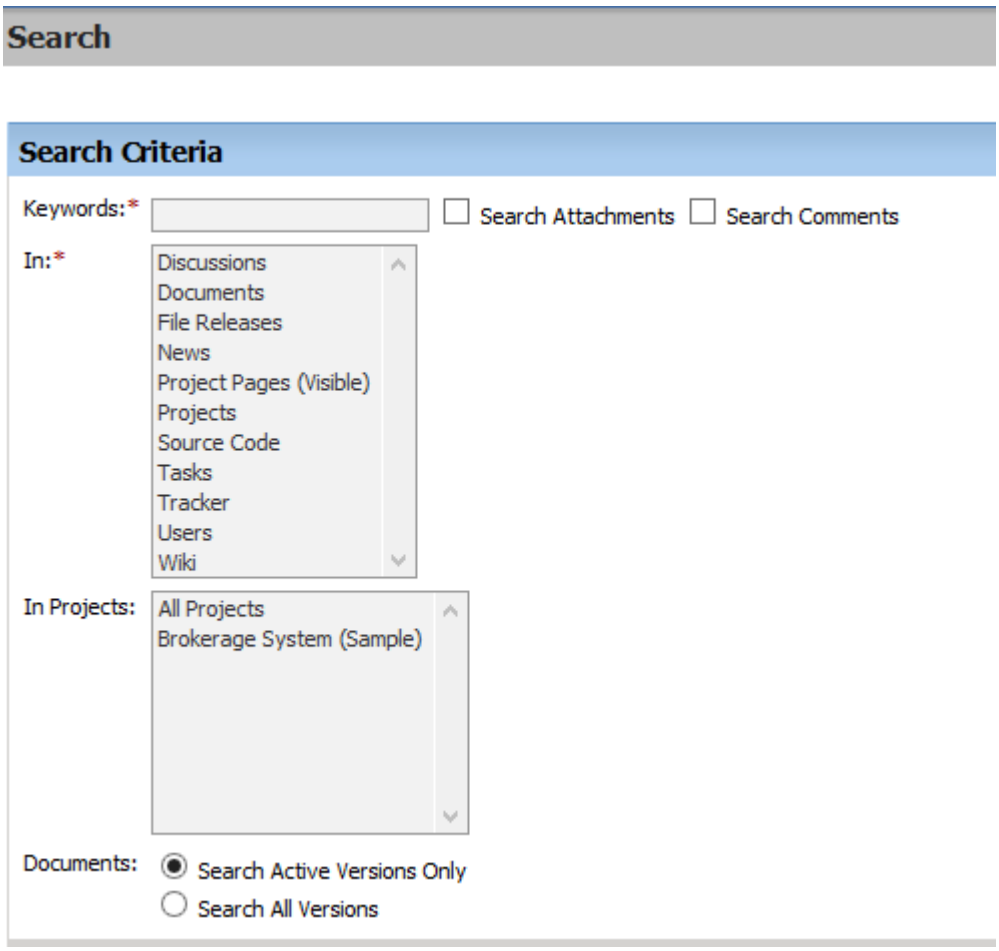


FIGURE 24: SEARCH TOOL

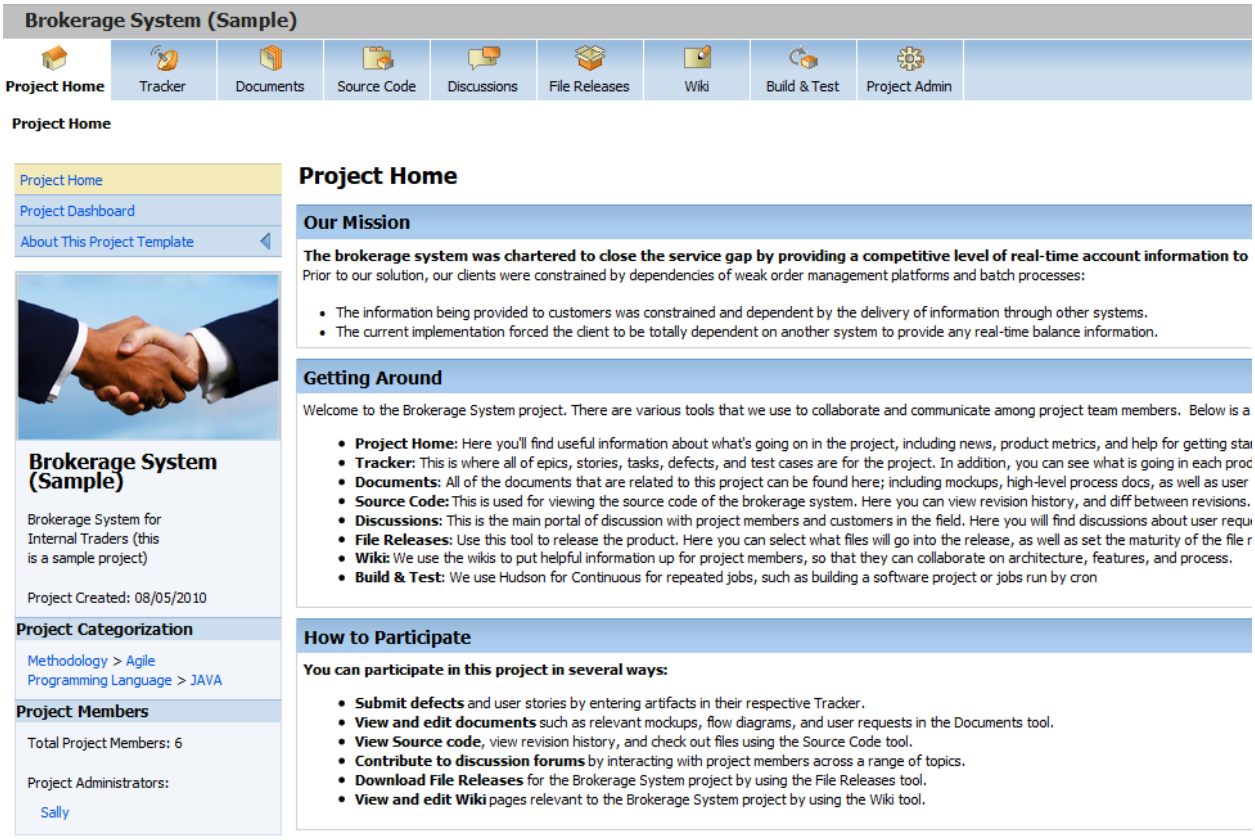


FIGURE 25: PROJECT HOME

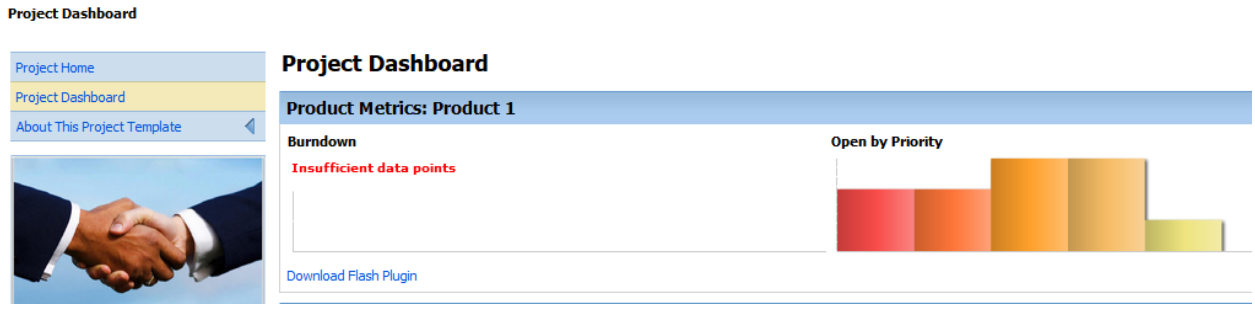


FIGURE 26: PROJECT REPORTING

Trackers > Defects > **View Artifact**

Artifact artf1053 : search by co name is broken

Tracker:	Defects
Title:	search by co name is broken
Description:	Returns a generic error--causing a major problem for the brokers and needs to be addressed asap!
Submitted By:	TeamForge Administrator
Submitted On:	08/05/2010 11:52 AM IST
Last Modified:	08/05/2010 11:52 AM IST

FIGURE 27: DEFECT RECORD

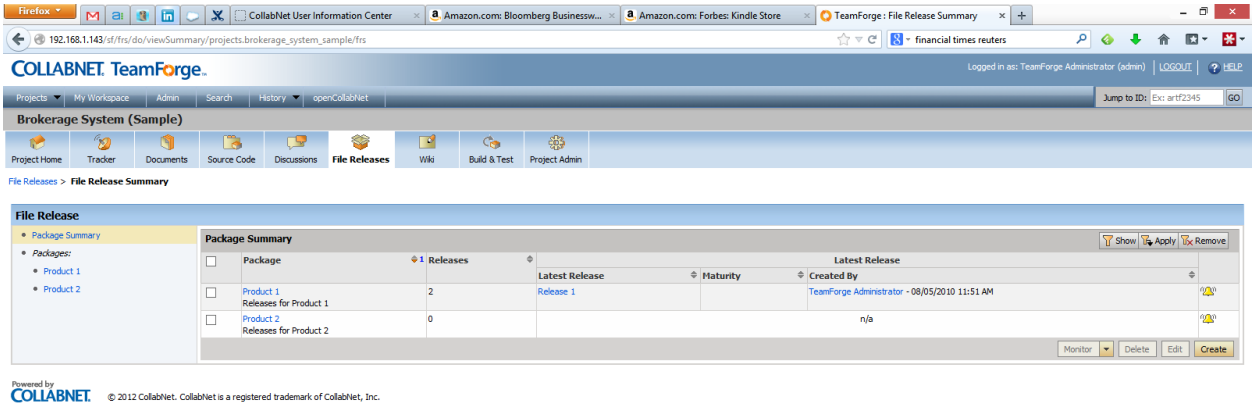


FIGURE 28: FILE RELEASES

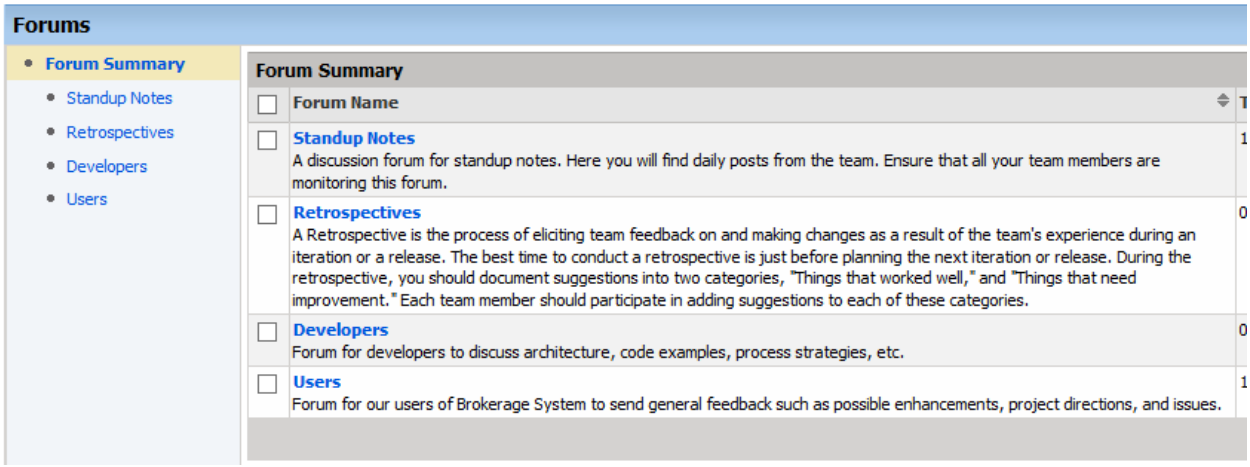


FIGURE 29: FORUMS

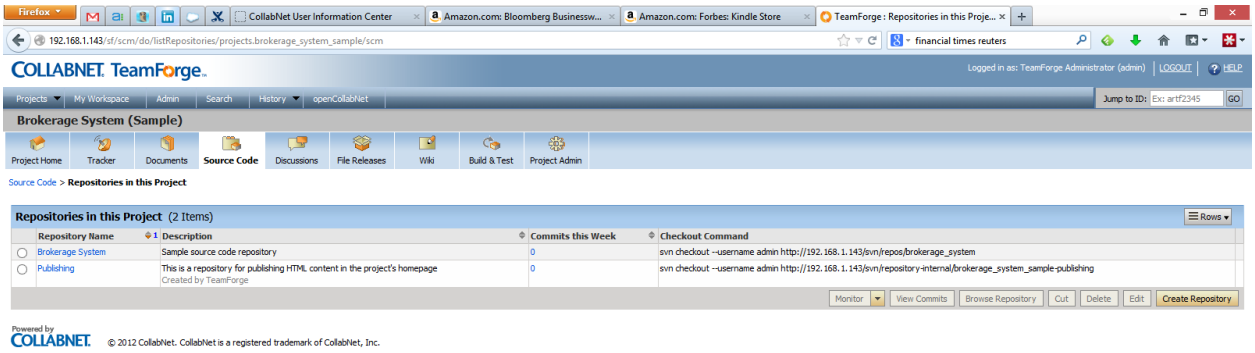


FIGURE 30: PROJECT REPOSITORIES

FIGURE 31: REPO BROWSING

Gerrit

Delivery model	On-premise
Unique strengths	• Pre-flight mainline model workflow with human and machine voting • Flexible security and access control • Key part of Android ecosystem • Proven to scale to large number of contributors • Simplified submodule handling
Average capabilities	• Plugin system • Simple repository replication
Weaknesses	• Administration complexity • Unattractive user interface • Ease of use • Few social features beyond code review
Pricing	• Free

Capsule Review

Gerrit evolved from code review tools developed internally at Google, and is now a key part of the Android ecosystem. It is used at large and busy sites.

Gerrit on the surface is not an attractive product. The user interface is clunky, the learning curve can be steep, the data model is very straightforward, and repository management is rather simplistic. But, Gerrit has proven to be a fair project and review system for Git repositories.

Gerrit has a unique workflow. By default, any change pushed to a repository is intercepted and put into a review branch. Review branches are pseudo-branches created dynamically and managed entirely by Gerrit. During a review both humans and build processes vote on the change. Approved changes can be merged into the real branch.

This workflow offers three compelling features:

- Every change must be reviewed.
- Successful build results may be required.
- It avoids long-lived feature branches in favor of transient review branches and pre-flight build built into the process.

The workflow is thus in line with continuous integration and continuous delivery best practices.

Gerrit also has a very strong security model, with access control available at the branch and reference level.

This diagram from the Gerrit documentation illustrates how Gerrit fits into a development tool stack:

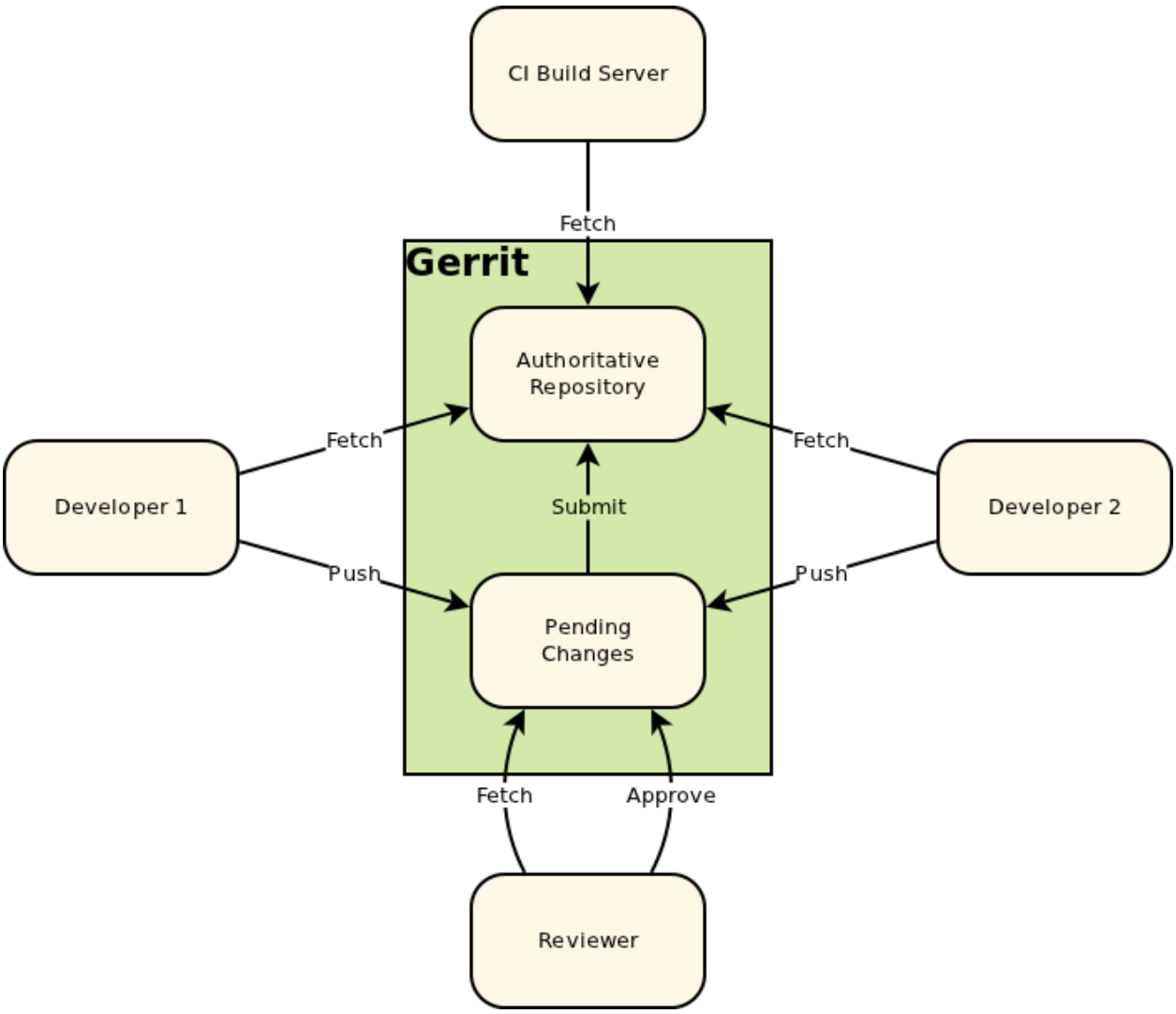


FIGURE 32: GERRIT'S ROLE

The standard Gerrit workflow is seen in the Android patch process, illustrated in the Gerrit documentation as follows:

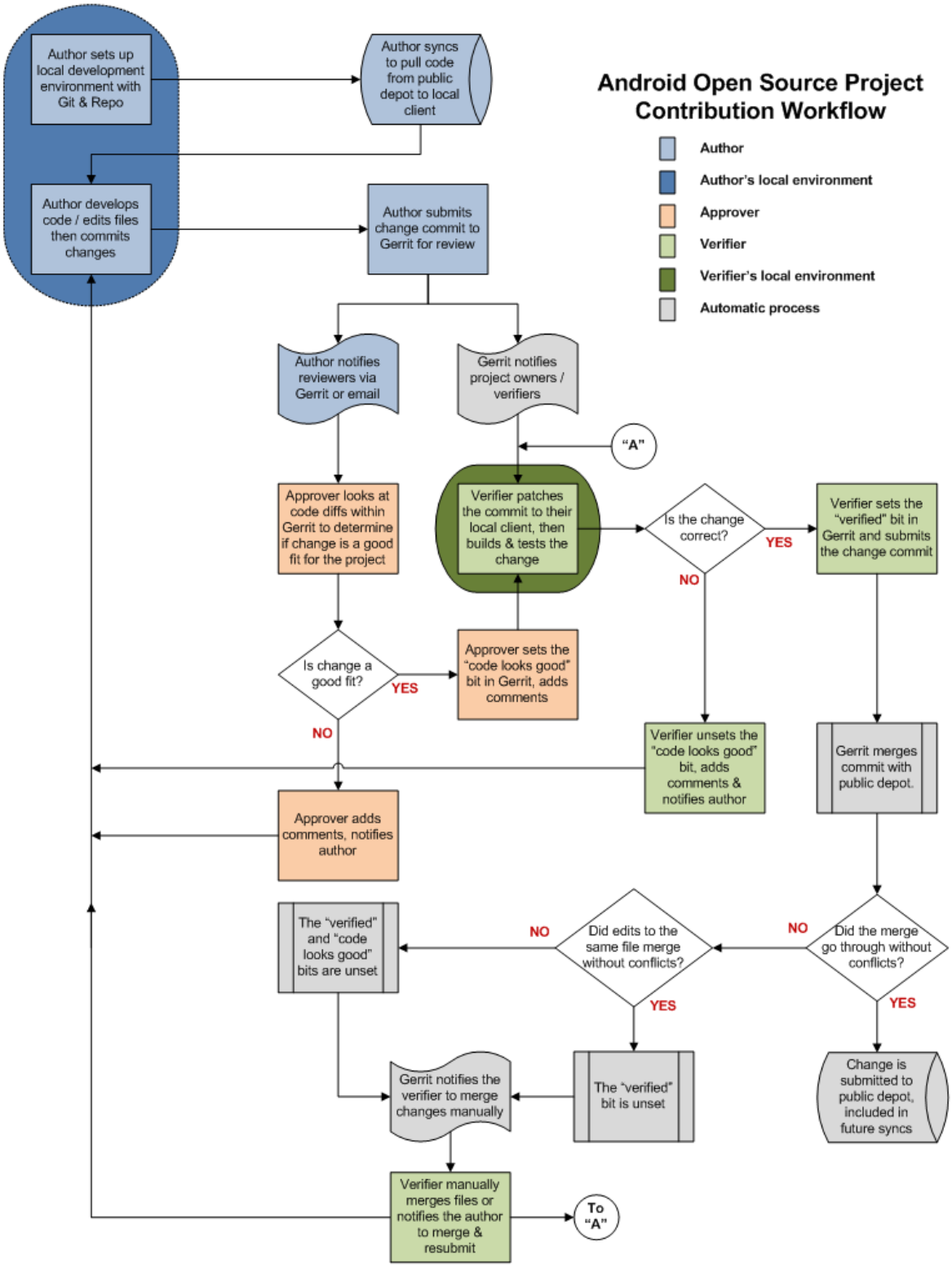


FIGURE 33: ANDROID PATCH PROCESS

Collaboration

Social collaboration features are not a priority for Gerrit. Beyond commenting, email notifications, and voting on code reviews, it offers few collaboration features. Or more accurately, some of the other collaboration features like custom dashboards are not easily created and shared by average users.

Codeline Workflows

Gerrit strongly supports the mainline model as noted earlier. The default Gerrit workflow involves these steps:

- Clone a repository from Gerrit
- Make local changes
- Push to a special address (normally *refs/for/master*)
- Gerrit creates a new branches on the fly for this change
- The commit is reviewed by peers using voting
- The commit is built and the results constitute a vote. A build can be triggered automatically (e.g. with the Jenkins plugin) or manually (e.g. a user pulls the commit in question into a clone and tests manually).
- The review is approved, and merged, or rejected in favor of rework.
- If rework is necessary, a new commit can be published using a special change ID in the commit message.

Gerrit is flexible and additional review steps (known as review labels) can be defined.

A topic branch workflow is feasible in Gerrit, but the fork-and-pull model is not well supported. Gerrit was designed for a site where a large number of contributors are working on a shared codeline.

Git Flow is possible, but Gerrit's default workflow tends to discourage the use of a large number of branches and the corresponding merge activity.

On a per-project basis Gerrit provides several related options, including:

- How 'clean' a merge must be when a reviewed commit is merged into a branch.
- Requiring sign-off messages and other signatures in commits.

Gerrit also supports a flexible search syntax and the ability to more easily manage submodules in a super project.

Integrations

Gerrit offers a REST API and some functions are accessible via SSH commands and a command line client. It now offers a plugin architecture. Most third party tools, particularly build processes, are integrated via simple scripts.

Plugins are of two types, plugins (full access to Gerrit) and extensions (less tightly coupled with less access). They are developed using a Maven framework and can offer both web and SSH functionality. Common uses for plugins include additional validation of commits.

Gerrit also supports a unique set of repository hooks for customized workflow.

Administration

Gerrit makes basic repository management easy. Creating repositories that use the default workflow and security settings is straightforward. Branches can be created in the web UI. The access control system is quite sophisticated and very fine-grained control of repository activity is possible. A Prolog workflow engine is included when tailored review processes are desired.

However, using the advanced access control and workflow features requires some expertise. Gerrit is a complicated system and the administration UI is not intuitive. A site with a large number of projects will do well to invest in building up the expertise necessary to use project templates for easier administration.

Gerrit uses OpenID accounts by default. These include public email systems from Gmail and Yahoo, but in an enterprise setting a compliance internal email domain is required. It manages a simple group structure internally. Several personal identifies can be associated with a single Gerrit account. Some Single Sign-On solutions are supported.

Gerrit is a web application built using Google Web Toolkit. It can be hosted in any J2EE servlet container such as Tomcat or Jetty. It has two data stores, the Git repositories and a relational database. The Gerrit community is trying to move as much data as possible into Git, but at the moment backup and scaling plans need to work with both data stores.

Gerrit can scale out to a large number of contributors and active push requests. However, due to the use of JGit it has scaling issues. Support larger instances, Gerrit will require expensive and powerful and high multi-core servers with tens of gigabytes of RAM and solid state storage[3] to try to obtain acceptable performance. Large installations also use Git mirrors and Gerrit slaves.

[3] http://code.google.com/p/gerrit/wiki/Scaling

Screenshots

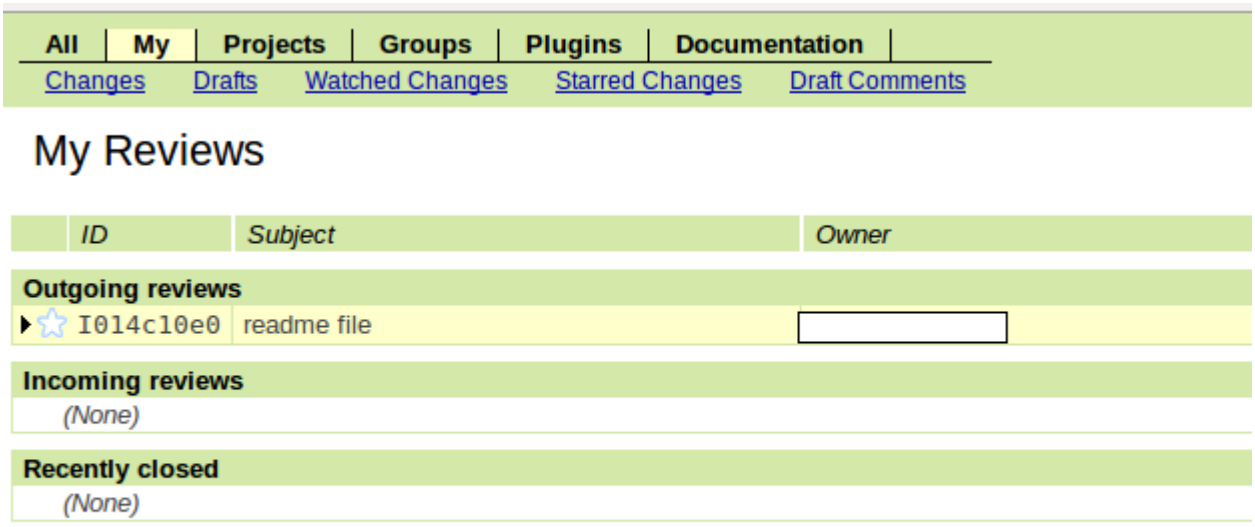


FIGURE 34: PERSONAL DASHBOARD

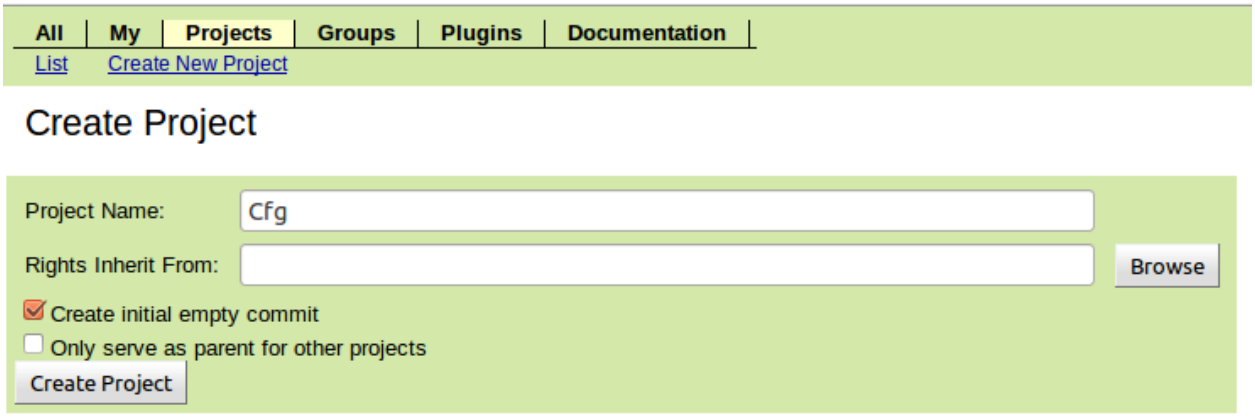


FIGURE 35: GERRIT PROJECT CREATION

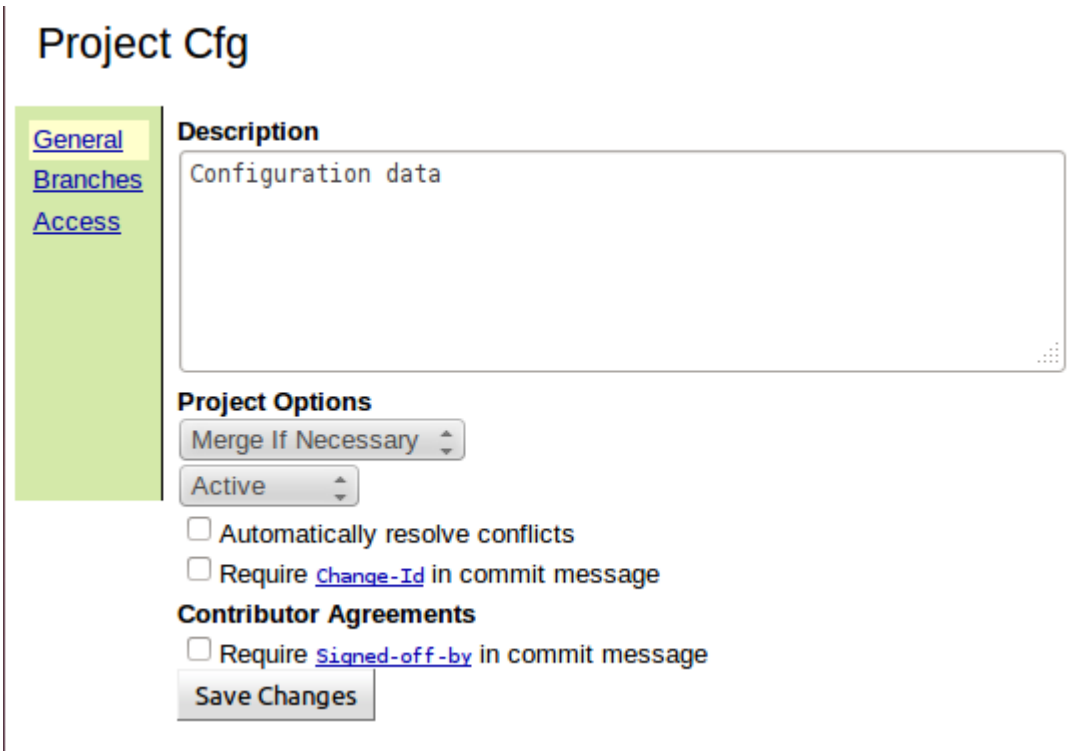


FIGURE 36: GERRIT PROJECT OPTIONS

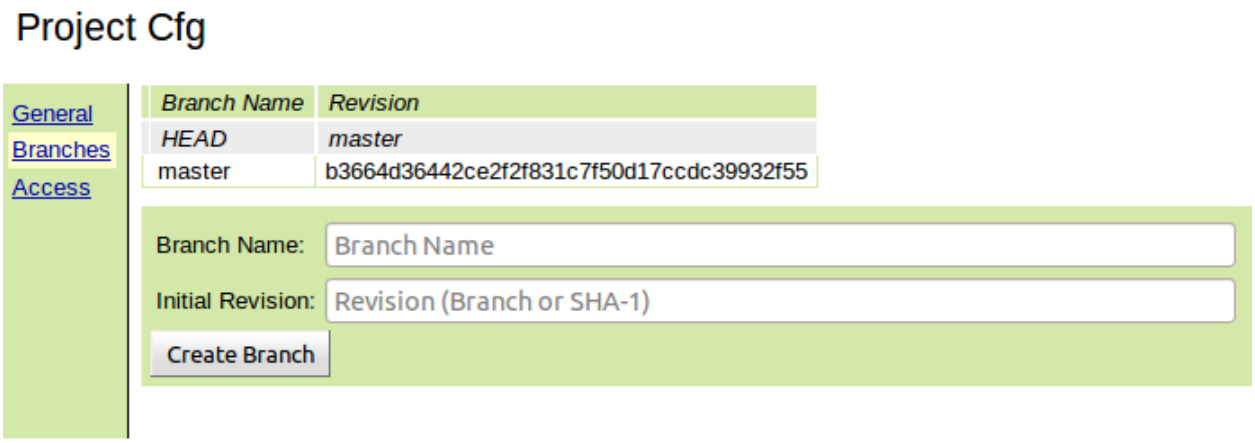


FIGURE 37: BRANCH CREATION

FIGURE 38: DEFAULT ACCESS CONTROL

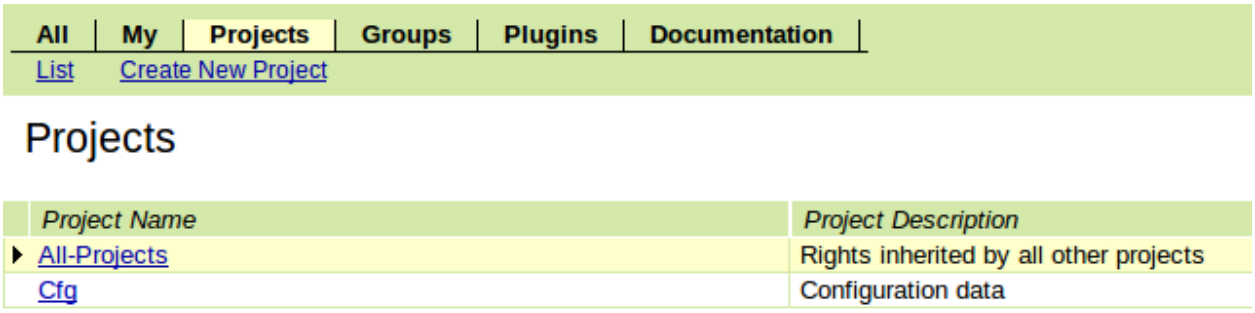
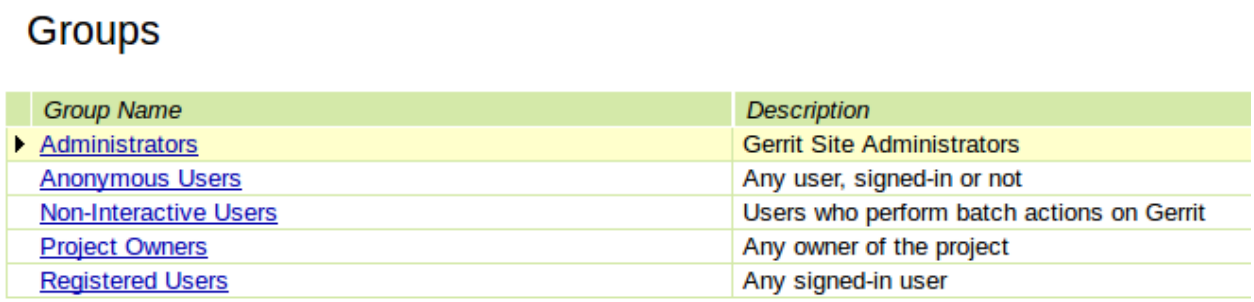


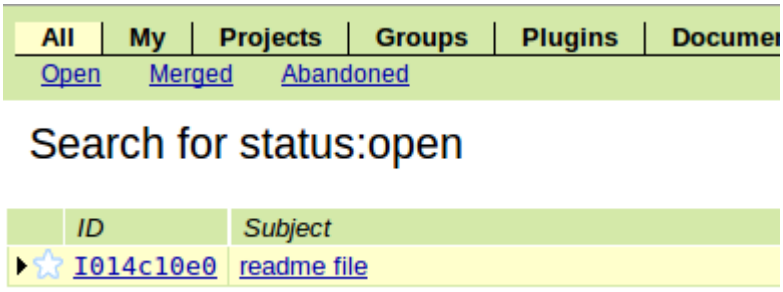
FIGURE 39: PROJECT LIST



Groups

	Group Name	Description
▸	Administrators	Gerrit Site Administrators
	Anonymous Users	Any user, signed-in or not
	Non-Interactive Users	Users who perform batch actions on Gerrit
	Project Owners	Any owner of the project
	Registered Users	Any signed-in user

FIGURE 40: GROUPS



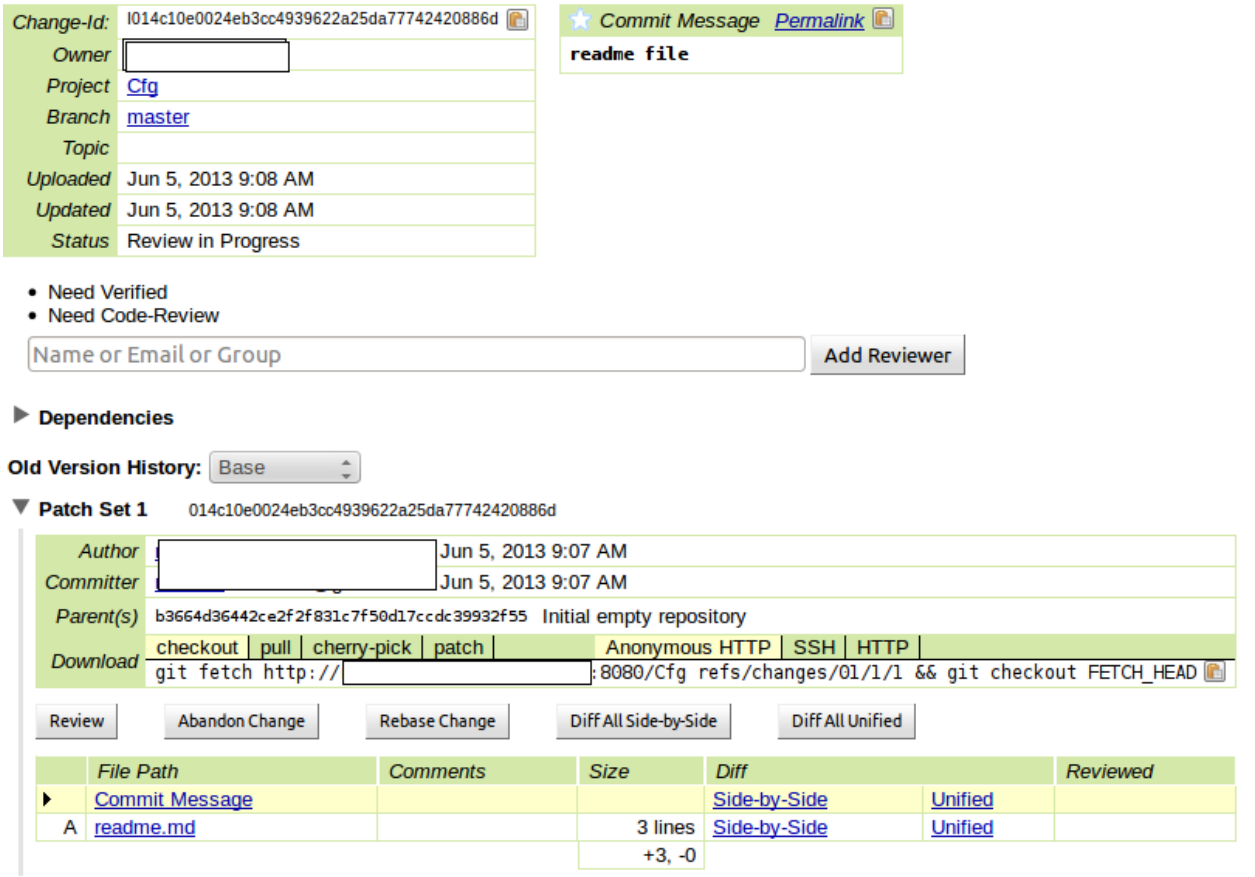
All | My | Projects | Groups | Plugins | Docume

Open Merged Abandoned

Search for status:open

	ID	Subject
▸ ☆	I014c10e0	readme file

FIGURE 41: OPEN REVIEWS



Change-Id:	I014c10e0024eb3cc4939622a25da77742420886d
Owner	[redacted]
Project	Cfg
Branch	master
Topic	
Uploaded	Jun 5, 2013 9:08 AM
Updated	Jun 5, 2013 9:08 AM
Status	Review in Progress

☆ Commit Message Permalink

readme file

- Need Verified
- Need Code-Review

Name or Email or Group | Add Reviewer

▶ Dependencies

Old Version History: Base

▼ Patch Set 1 014c10e0024eb3cc4939622a25da77742420886d

Author	[redacted] Jun 5, 2013 9:07 AM
Committer	[redacted] Jun 5, 2013 9:07 AM
Parent(s)	b3664d36442ce2f2f831c7f50d17ccdc39932f55 Initial empty repository
Download	checkout | pull | cherry-pick | patch — Anonymous HTTP | SSH | HTTP
	git fetch http://[redacted]:8080/Cfg refs/changes/01/1/1 && git checkout FETCH_HEAD

Review | Abandon Change | Rebase Change | Diff All Side-by-Side | Diff All Unified

	File Path	Comments	Size	Diff		Reviewed
▸	Commit Message			Side-by-Side	Unified	
A	readme.md		3 lines +3, -0	Side-by-Side	Unified	

FIGURE 42: REVIEW IN PROGRESS

FIGURE 43: VOTING ON A REVIEW

FIGURE 44: KEYBOARD SHORTCUTS

Securing Git: Options for Single Sign On and Two Factor Authentication

Git supports several transport protocols:

- File
- Git
- SSH
- HTTP/S

Only two of these protocols, SSH and HTTP/S, offer real authentication options.

Two Factor Authentication with SSH

SSH is considered a fairly secure protocol although managing SSH keys can be an administration burden. Most often developers create their own SSH keys and register the public key with the Git server.

However, SSH keys can also be password-protected. This approach is not a typical two-factor authentication technique, but it does require possession of a private key and knowledge of a password in order to authenticate.

Enforcing the use of a password on an SSH key may require use of an SSH key management system. Commercial SSH management solutions are available, but another solution is simply to

provide a self-service portal that requires a password before generating an SSH key and registering it with the Git server.

Single sign-on with HTTP/S

Git is often served through the Apache web server (Git embeds *curl*) and can take advantage of many standard Apache authentication modules, including those for LDAP and Active Directory. One interesting possibility is to require client-side certificates. This provides a fairly secure SSO method as a certificate is required to access Git. Certificates can be expired and otherwise managed.

In order to try this approach:

- Create a certificate authority if you don't already have one available.
- Create client certificates for developers. (Client certificates do not have to be provided by a commercial certificate provider; internally generated certificates are fine.)
- Define the CA file in Apache config (e.g. 'SSLCACertificateFile /etc/CA/ca.crt')
- Require client certificates in Apache config (e.g. 'SSLVerifyClient require')
- Configure user name from certificate in Apache config (e.g. 'SSLUserName SSL_CLIENT_S_DN_CN')
- Restart Apache
- Each developer adds their certificate and key to their global Git config

Two-factor authentication and single sign on with Kerberos

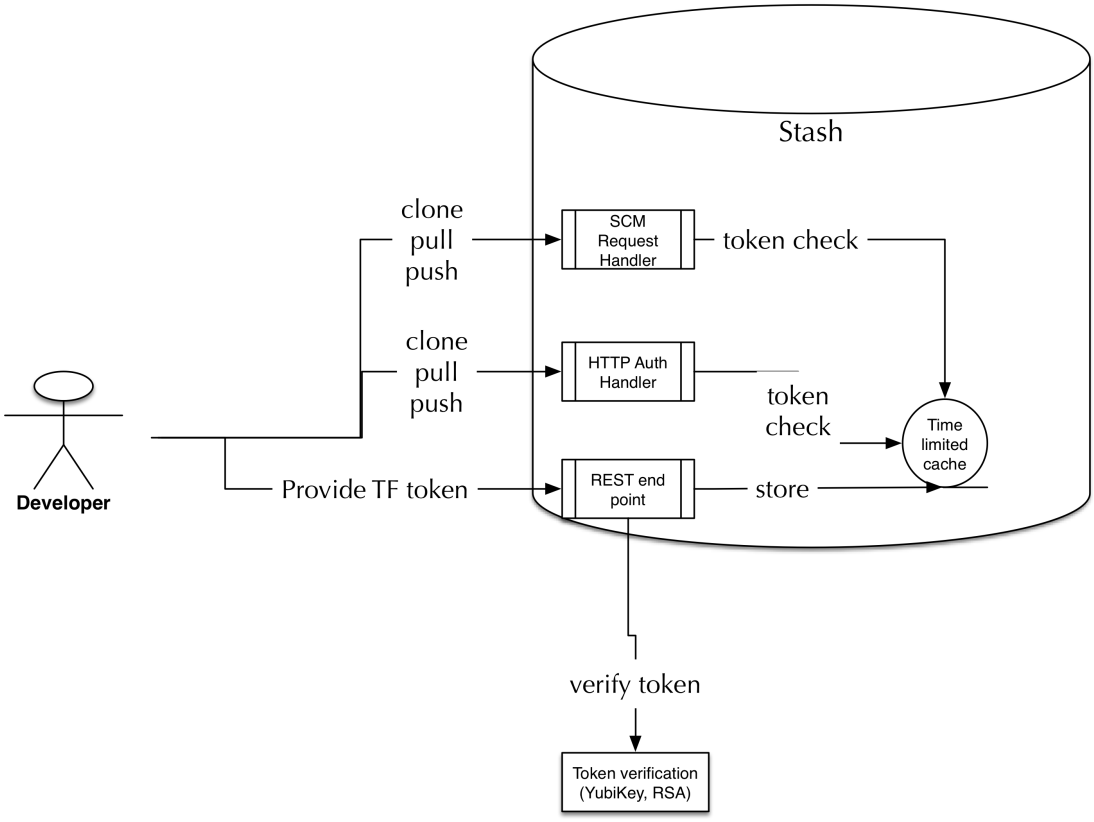
Kerberos is a common cross-platform identity management system. If you use Kerberos for operating system account authentication and require two-factor authentication through Kerberos, you can take advantage of this capability in Git.

You would set up Apache to authenticate with Kerberos tokens. Each developer would use Git's ASK_PASS setting to use a script that would locate their Kerberos token and provide it to Apache. Assuming again that they had to use two-factor authentication with Kerberos, you now have two-factor authentication and SSO for Git.

Two Stage Two Factor Authentication with Stash

This solution is specific to Stash and is modeled on a workflow developed for the Linux kernel.

As depicted below, this solution entails two phases. First, a developer provides a valid second authentication token to Stash before attempting any repository operations. Second, Stash verifies the provided token during any repository access attempts.



Token verification

Before attempting any repository access, a developer must make a call to a custom Stash REST end point. The REST plugin verifies the token against an authentication service such as YubiKey, then stories the verified token in a time-limited memory cache.

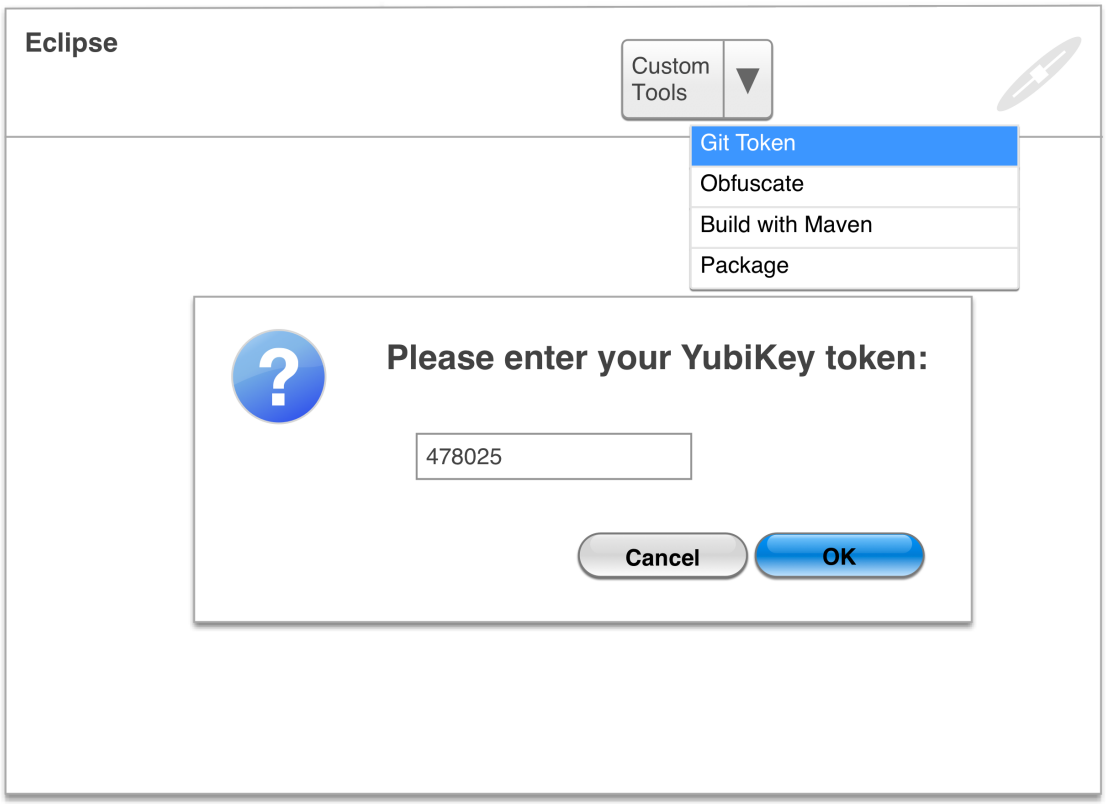
Each developer is provided with a script to run from the command line or to plug in to IDEs such as Eclipse and Visual Studio, as shown below.

Git Bash

> git token

Please provide your YubiKey token:
<478025>
.. Validating …

Accepted! Your token grants access to Stash repositories for 12 hours.



Token check

When a developer attempts to access a repository, custom handlers (for HTTP requests or more general SCM access) check for the existence of a verified token for the developer. If the token is not present or has expired, the attempt is blocked.

Improving Stash

After reading the preceding discussion, you may be struggling to decide which route to take. Atlassian Stash is a popular choice for many: a strong user community backed by a dynamic vendor, low pricing, and a well understood architecture. But is it best-of-breed?

The answer is that Atlassian has invested in flexibility rather than striving to satisfy today's needs perfectly. By taking advantage of this architecture, you can adopt the best of other solutions and maintain the ability to adapt in the future.

In the subsequent sections, we'll examine how to improve Stash in a few key areas.

Simple Stash High Availability Solution

Because Stash serves as the master repository and is the project portal, down time is very costly to a development organization.

This section describes a simple High Availability (HA) and Disaster Recovery (DR) solution for Stash. It reduces outage time in the event of hardware failure on the main Stash server or any of its components and prevents data loss in the event of total local site failure, with reduced down time.

HA does not provide relief for performance difficulties. Stash can usually be scaled with improved hardware and storage solutions, as well as read-only Git mirrors.

Stash Architecture

Stash is a web application that stores configuration and repository data on a file system and other data in a relational database (RDBMS). There are three components to consider in an HA/DR solution:

- The Stash installation directory contains the Stash application, plugins, and customizations. In most Stash installations this directory also contains the application server that hosts Stash.
- The Stash home directory contains the Git repositories and some basic Stash configuration data and log files.
- The RDBMS stores most Stash application data.

Solution Architecture

The HA/DR solution architecture is shown in Figure 45: Stash HA/DR Solution.

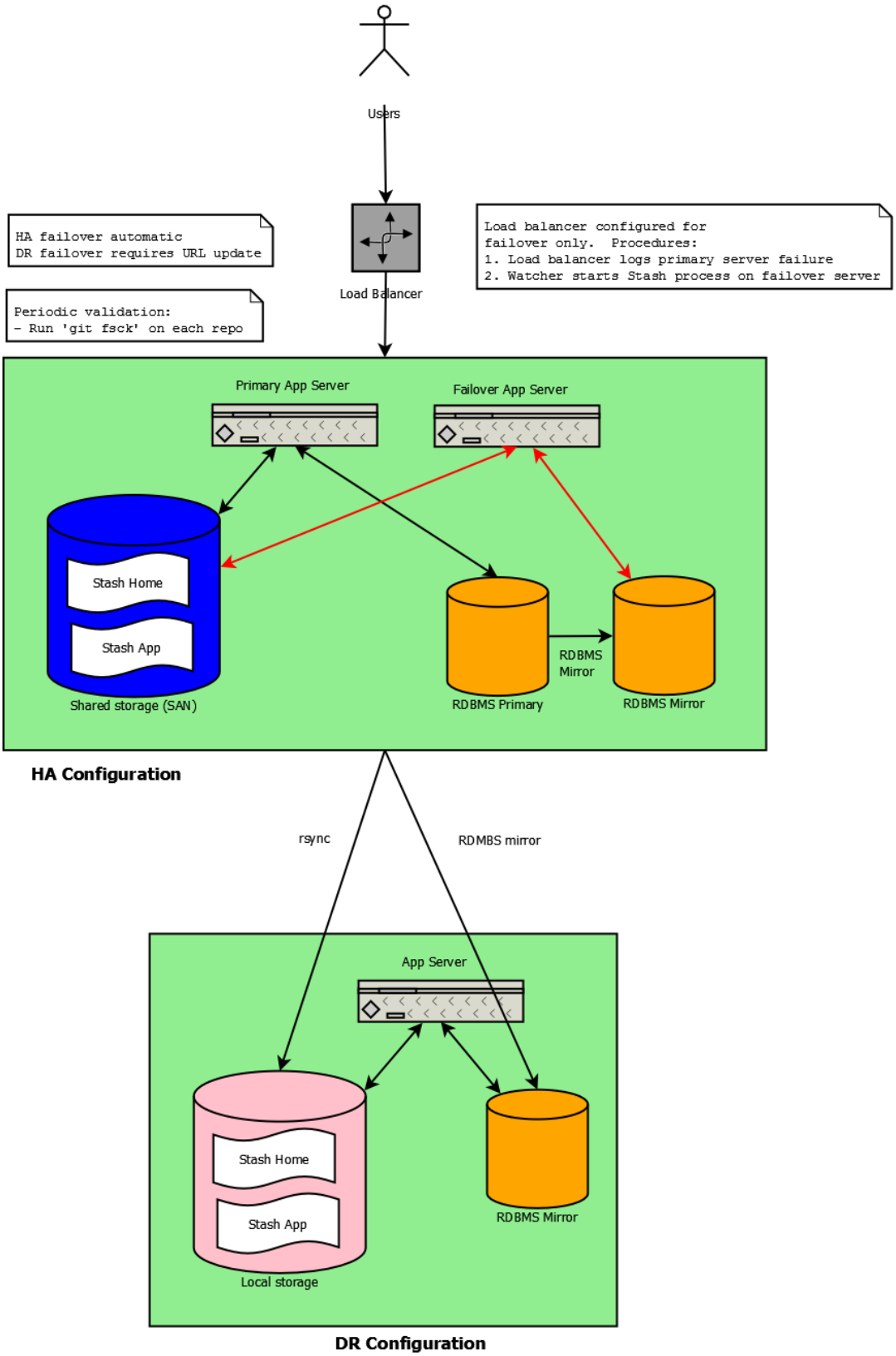


FIGURE 45: STASH HA/DR SOLUTION

In this scenario, two identical pieces of hardware serve as Stash primary and spare (failover) servers. These servers run the Stash application, but do not host any data. As such they are stateless. A monitoring solution is responsible for detecting failure of the primary server and initiating failover to the spare server. The monitoring solution may be an off-the-shelf solution such as HAProxy or a custom program. In any case the monitoring solution must only respond to hardware failures. A software failure (e.g. the Stash application crashing due to a faulty plugin) should be detected and dealt with separately.

The Stash installation and home directories are located on separate LUNs on a SAN. (Using separate LUNs prevents I/O contention and allows for allocation of higher performing storage for the home directory.) For the purposes of this solution the SAN is considered to be fault tolerant. The SAN is mounted to only a single Stash server at a time.

The RDBMS uses native mirroring (replication) technology to provide a spare (slave) database server. Note that the choice of RDBMS will largely dictate the implementation of the mirroring and monitoring, with each RDBMS having greater or lesser capabilities. For example, MySQL provides mirroring but not monitoring, so an off-the-shelf or custom monitoring solution is necessary.

Disaster recovery is provided by a second configuration of servers at a remote site. An application server with local storage runs Stash and hosts the installation and home directories, and is kept up to date with *rsync* or similar tools. Another RDBMS mirror provides the rest of the data.

Failover Scenarios

This section considers failure of each major system component.

RDBMS

In the event of RDBMS failure, the monitoring solution must:

- Stop the Stash application or make it unavailable to users.
- Make the mirror (slave) the new master RDBMS. This can be accomplished with DNS switching or via native RDBMS cluster functionality.
- Start the Stash application or make it available to users again.

HA Storage System

As noted earlier, the SAN is assumed to be fault tolerant. Most enterprise storage solutions already come with failover capabilities in the event of hardware failure.

Stash Server

If the primary server fails, the monitoring solution must:

- Switch the SAN mount point to the spare server.
- Start Stash on the spare server.
- Switch the DNS entry so users and applications will be able to communicate with the spare server.

The monitoring solution may be off-the-shelf or a custom solution, perhaps built around a load balancer.

Total Primary Site Failure (Disaster Recovery)

In this scenario, an administrator should trigger a process that starts the Stash application on the DR app server and configure the DR RDBMS to act as a primary database. Verification checks should then be run on each Git repository.

After the DR site is ready, users must use a new URL to access Stash and their Git remotes.

Failback Scenarios

In the event of failover of any HA component, a new spare or mirror (slave) must be set up. Until it is available the DR site becomes the only failover possibility other than cold backups. In some cases the repaired primary or master can take over this role, while in other cases new hardware must be procured and configured.

If the DR site is used, aggressive cold backup procedures should be instituted until a new primary HA site is available. Performance will be sacrificed for the sake of data integrity.

For the RDBMS, it is assumed that configuring a new mirror (slave) is a known procedure. Using deployment solutions like Puppets can greatly reduce the time required to deploy and configure a new server.

Similarly, configuring a new Stash spare server can be done manually or with a deployment tool like Puppet. Using pre-configured server images is highly recommended as it reduces deployment time and the chance of errors.

After the new spare or mirror (slave) is in place, the monitoring solutions must be made aware of them.

Multiple Redundancy

Additional spare or mirror (slave) servers can be added to this configuration with a minimal change to the monitoring solutions, providing even greater redundancy.

Summary

This solution provides Stash HA with a very good Recovery Point Objective (RPO) and Recovery Time Objective (RTO). The RPO can be assumed to be zero data loss if the SAN is indeed fault tolerant. The RTO will typically be 10 minutes or less, depending on the sophistication of the monitoring solutions and the speed of the DNS switch.

This solution also provides Stash DR with a very good RPO and a reasonable RTO. The RPO will typically involve no more than a few minutes of possible data loss, while the RTO will be bounded by the failover steps required of the administrator.

This solution makes no provision for high performance. High performance for Stash can generally be obtained with improved hardware, storage, and configuration, and possibly Git mirrors.

Advanced Stash Cluster

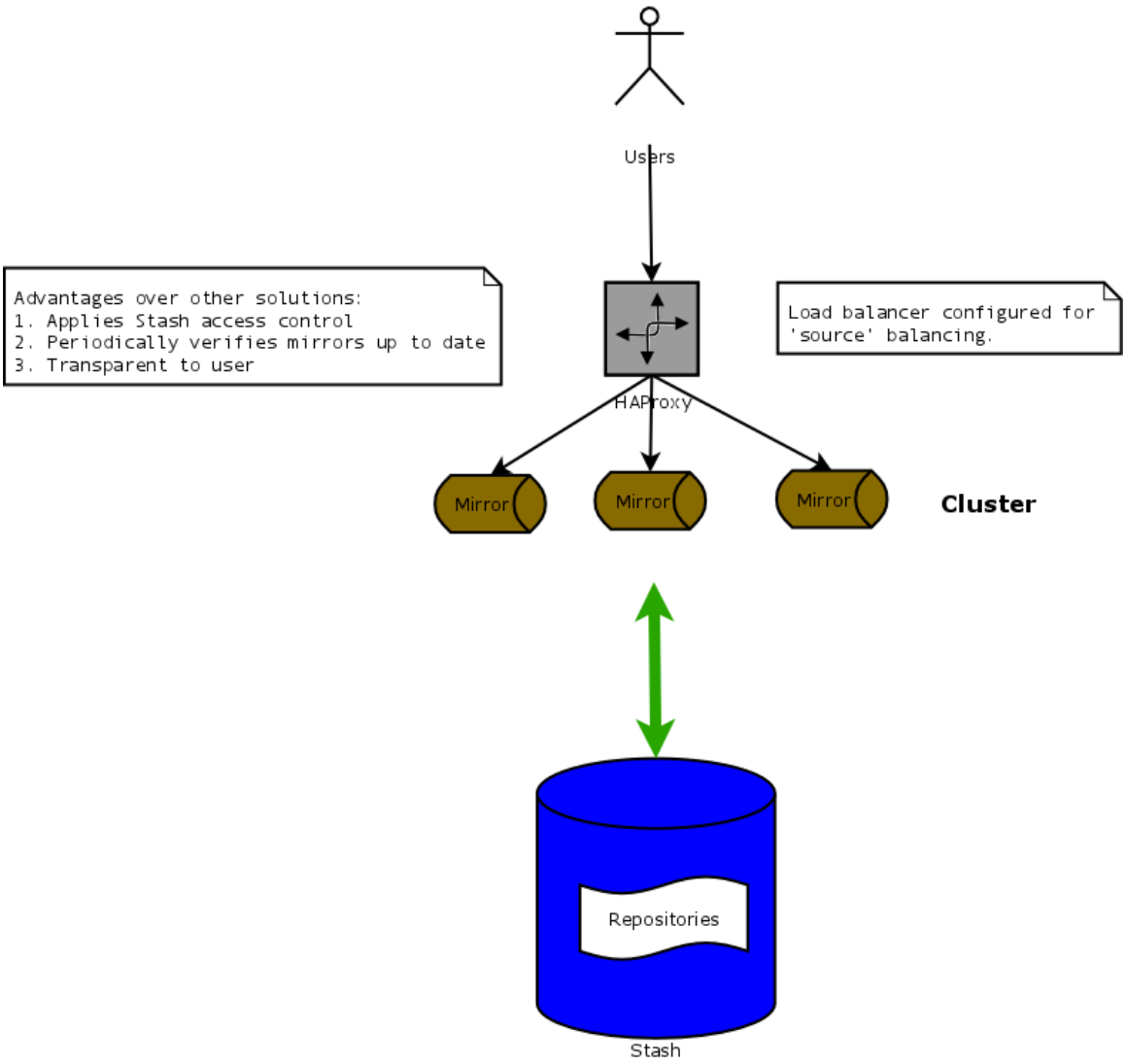
An alternative approach to Stash clustering provides load balancing and high availability. The design places a load balancer (we use HAProxy as our example) in front of all end users. The load balancer directs any read request to one of the Stash mirrors in the cluster, while write requests are sent to Stash directly. The effect is transparent to the end user, who manages a single Git remote as usual.

Authentication is done with Crowd, providing single sign-on to all mirrors and to Stash itself for transparent password authentication and reduced administration overhead. Authorization requests are always handled through Stash, providing consistent permission checks at every node.

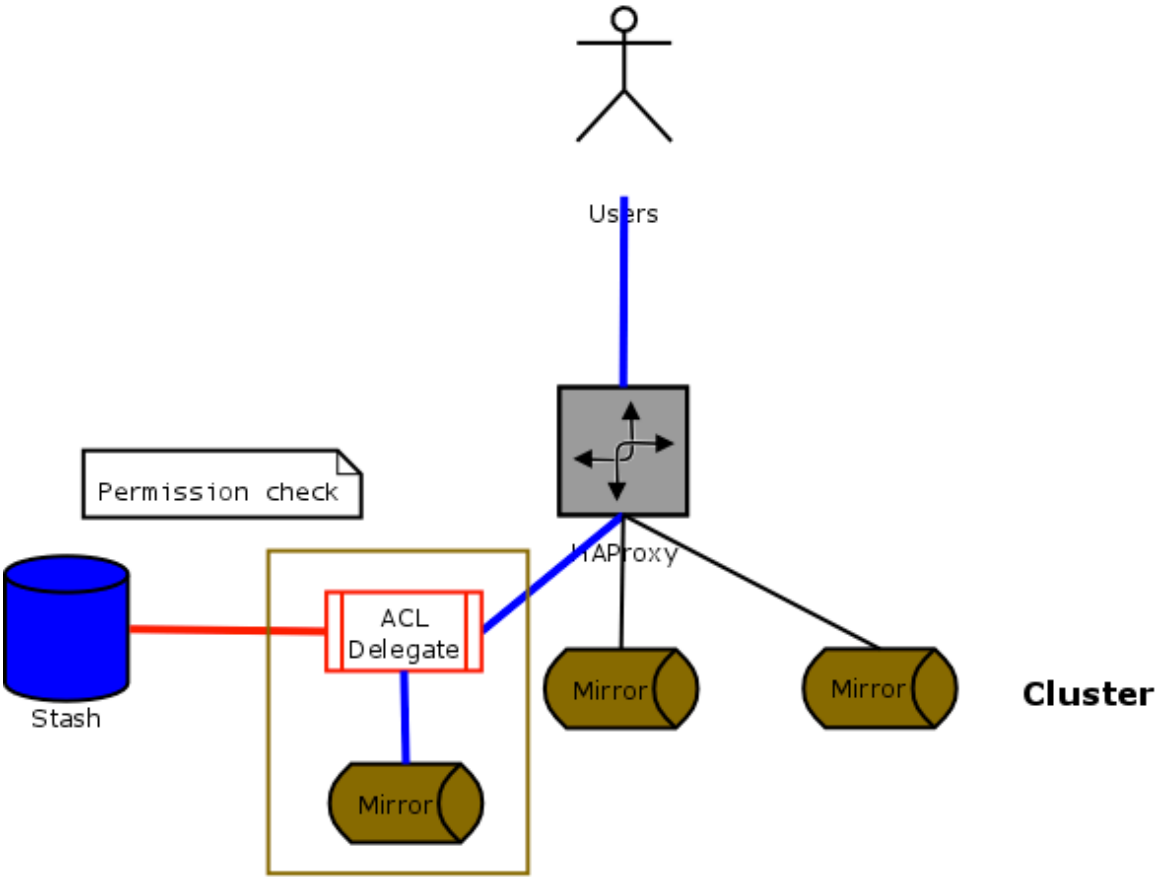
Additional benefits include automatic monitoring of the mirrors to ensure the consistency of repository data at every site. The lightweight monitoring algorithm reduces concern about out-of-sync mirrors.

Design

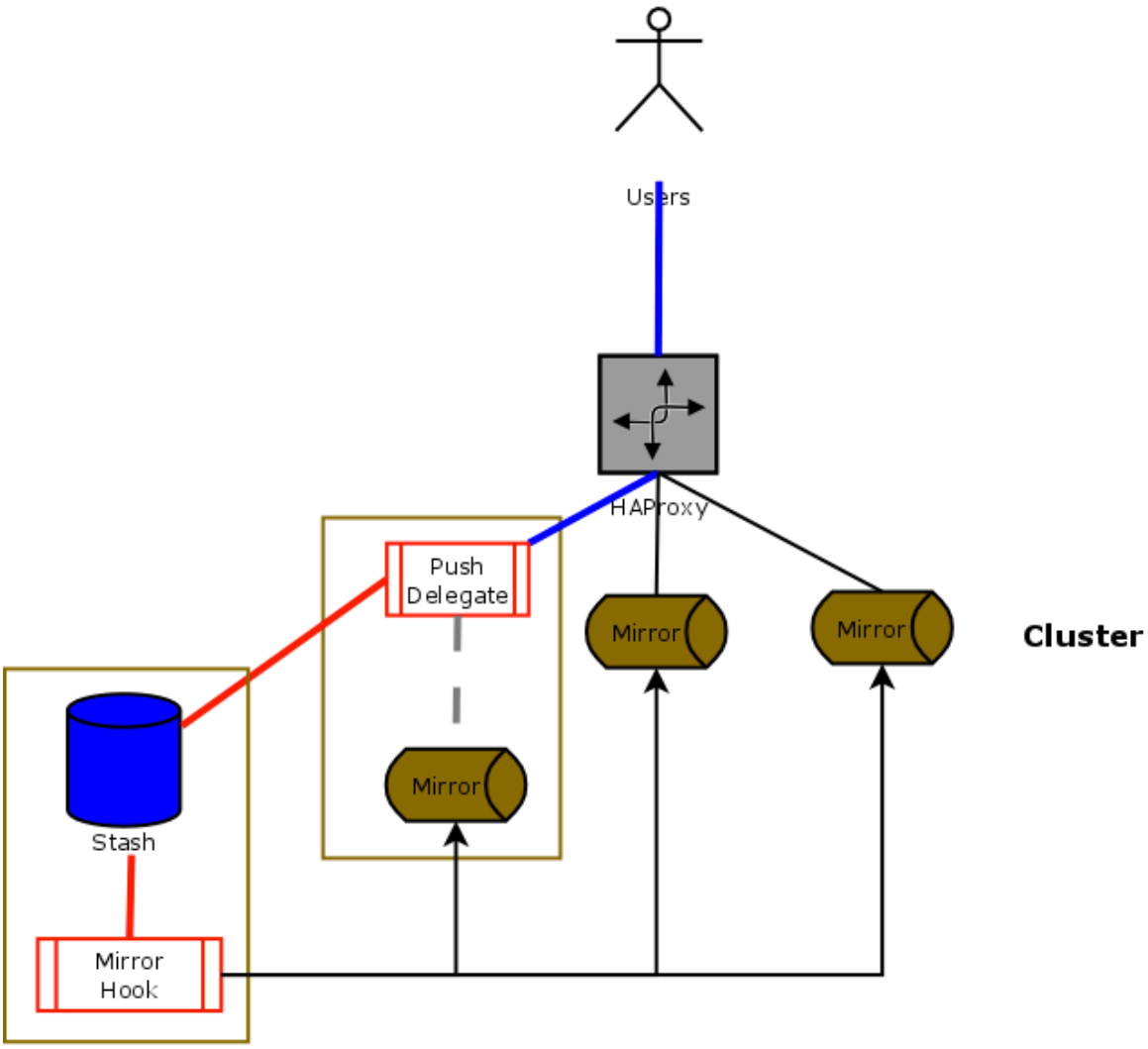
The overall design is shown below.


Users
Advantages over other solutions:
1. Applies Stash access control
2. Periodically verifies mirrors up to date
3. Transparent to user
Load balancer configured for 'source' balancing.
HAProxy
Mirror
Mirror
Mirror
Cluster
Repositories
Stash

An overview of read operations is shown below.



As noted earlier, all write operations are handled directly by Stash for consistency of permission checks and hook use.



Primary Use Cases

- *Load balancing* of read operations. The load balancer transparently distributes read operations (typically 98% of Git repository activity) to the mirrors.
- *Failover*. The cluster can survive the failure of all but one of the mirrors. Stash is still a single point of failure for write operations and access control. Recovered mirrors can be brought up to date simply by fetching all repositories from Stash again.
- *Monitoring*. The verification tools ensure consistency of all mirrors. The tools are efficient and can be run on a scheduled basis.

This design is not intended to handle remote site performance issues.

Go2Group
Complexity Made Simple

Migrating from Gerrit to Stash

Gerrit is a web-based code review and repository management product for Git. It was developed by Google and is used heavily in the Android community. Its success in Android led to adoption at other firms, with its powerful security model and rigorous workflow engine proving popular with enterprise software developers.

However, Gerrit is not intuitive and does not offer much in the way of social coding (collaboration) tools. As Stash has matured it now offers a viable alternative, with strong access control and a flexible workflow system. Indeed, a combination of configuration and plugins lets Stash emulate the most important Gerrit workflow features.

To developers looking to move from Gerrit to Stash, the last remaining roadblock is data migration. This guide describes a plugin that offers:

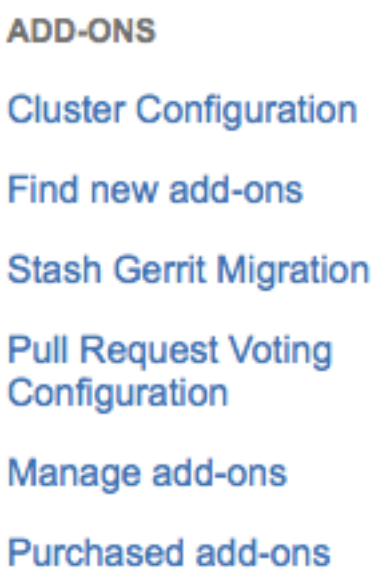
- Full and incremental transfer of Git repository data from Gerrit to Stash
- Configuration of Stash access controls to mimic Gerrit's access control as closely as possible
- Transfer of legacy code review metadata from Gerrit to Stash

Installation

Upload and install the provided plugin (stash-gerrit-migration-1.3.1.jar) per the normal plugin installation method.

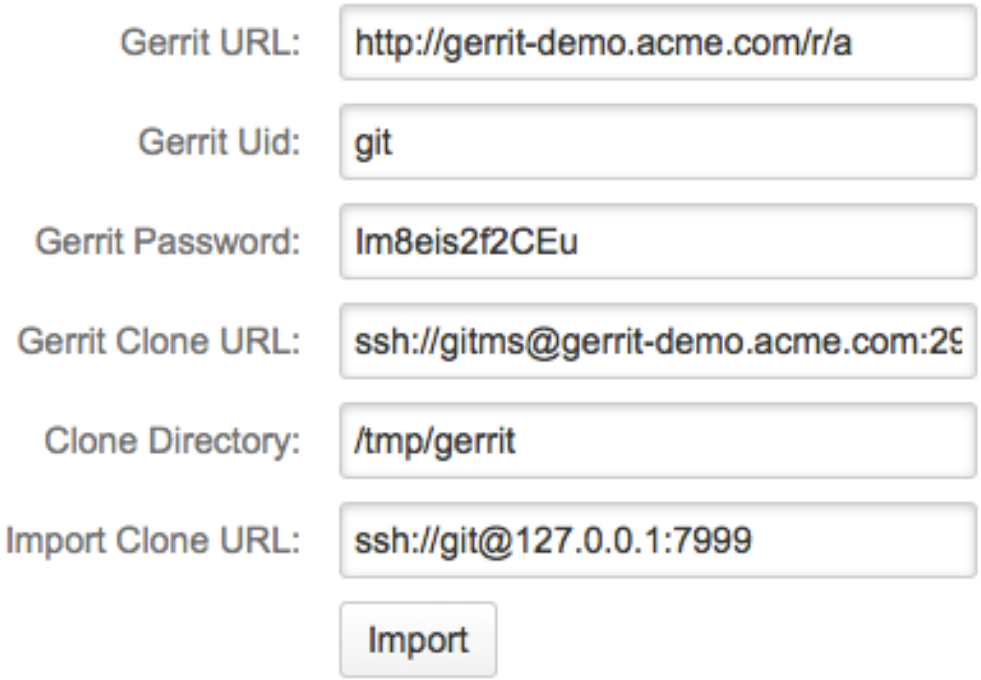
Configuration

Go to the menu item Stash Gerrit Migration.



Next enter the following parameters.

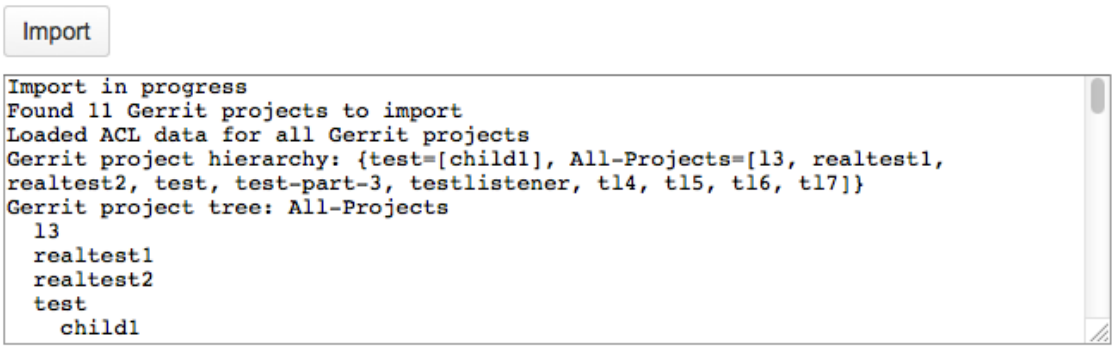
Name	Value
Gerrit URL	The URL for authenticated login to Gerrit's REST API.
Gerrit Uid	The Gerrit user ID to use for REST calls
Gerrit Password	The password to use for authenticated REST calls
Gerrit Clone URL	The SSH URL to use for Gerrit clone operations. It must support password-less authentication and be able to clone any Gerrit repository.
Clone Directory	The file system path to store temporary clones.
Import Clone URL	The Stash URL to use for pushing Git data. Must support password-less authentication.



These values will be saved until overwritten.

Starting Migration

Press the *Import* button on the configuration screen to start importing data. The status window will, upon refresh of the page, display the status of the current import. Note that you cannot start a new migration until the current run is complete.



When the migration is finished, each Gerrit project will exist as a repository in a Stash project, with appropriate access control settings.

Incremental Migration

The plugin supports incremental migration if you leave the cloned repositories on the file system in the temporary clone area. During incremental migration updates are fetched from Gerrit and transferred into Stash. At this time access control settings and code review metadata are not refreshed.

If you intend to use incremental migration, make sure to not push any commits to Stash before the final migration. The plugin will not attempt to merge data from Gerrit and Stash.

If you prefer to start over instead of performing an incremental migration, simply remove all Stash projects and run the tool again.

Migration Internals

Supported versions

The plugin was tested against Gerrit 2.8 and Stash 2.10. It uses only core features so forward compatibility is likely but not guaranteed.

The plugin makes heavy use of Gerrit's REST API to obtain Gerrit metadata. The REST API must be accessible.

Repository import

Gerrit projects exist in a multi-level hierarchy. The root is assumed to be *All-Projects*. Some projects contain data while others are used only to set access control for child projects.

Stash on the other hand uses a relative flat structure where a project can contain one or more repositories. Access control is typically done at the project level although each repository can have its own settings, notably branch-level restrictions.

In order to translate the Gerrit hierarchy to Stash, each Gerrit project that contains data is imported as a Stash repository. The Stash project is selected as the highest level containing Gerrit project, with the exception that *All-Projects* is not considered. (It is used however when importing access control data.) The table below shows examples of this translation.

Gerrit Tree	Stash Project	Stash Repository
All Projects ->	Java	Widget

Java -> Widget		
All Projects -> Java -> Widget -> UI	Java	UI
All Projects -> Java	Java	Java

Users and Groups

Users and groups are created on demand to satisfy access control rule import. Group membership will be configured when necessary to support access control rules.

Group recursion is not yet handled.

All new users have a default password of *stashuser*.

Access control rules

Few of Gerrit's permissions have a direct equivalent in Stash other than the basic permissions to read and write project data. Only the following permissions are directly translated:

- System administrator
- Read
- Write
- Project owner

In the future it may be possible to use heuristics to figure out a more complete translation. For example, those with special code review permissions may be Stash users who can manage protected branches. However any such scheme is unlikely to be bulletproof.

Inheritance is handled by reading all settings from the project and it's parent(s) recursively, walking all the way back to *All-Projects*. Permissions are applied cumulatively.

In terms of *refs* in rules, the following are supported:

* *refs/heads/** or *refs/** implies all branches

* *refs/heads/<pattern>/* implies branch-specific rules

* *refs/for/refs/* will be interpreted as new review-only branches

Other Gerrit *refs* such as *refs/meta/config* are simply ignored as they have no purpose in Stash.

Deny rules from Gerrit are ignored as Stash only has permissive rules.

Review Metadata

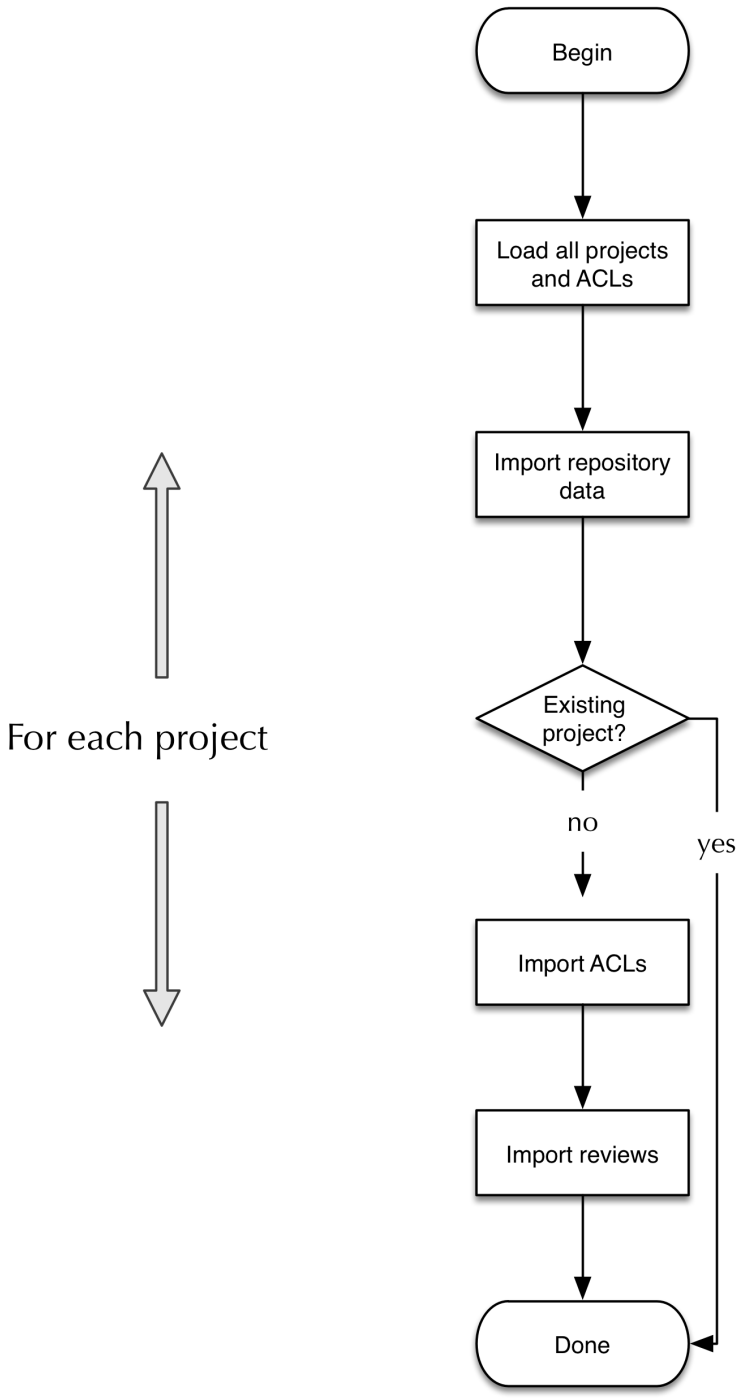
Directly representing Gerrit code reviews as Stash pull requests is difficult. Stash pull requests assume that you are merging from one branch to another, while Gerrit code reviews simply place incoming commits in a holding area pending review. There are no 'official' Git branches created for Gerrit reviews.

As a solution, the import tool creates new branches with the `gerrit-` prefix. A pull request is then created for each merged Gerrit review. The pull request will create a correct link from the parent commit to the new commit, although they will be on these temporary branches. The pull request will also summarize the Gerrit code review comments, approvals, and other data, in a pull request comment.

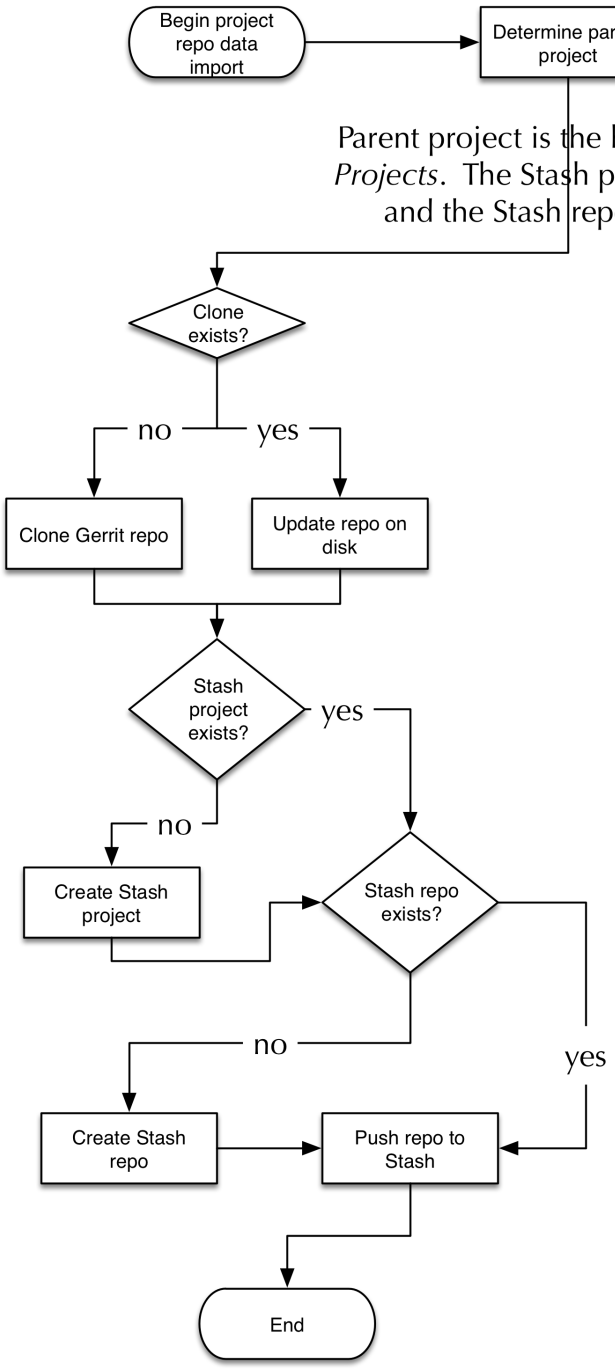
The temporary branches can be easily deleted later.

Process Flow

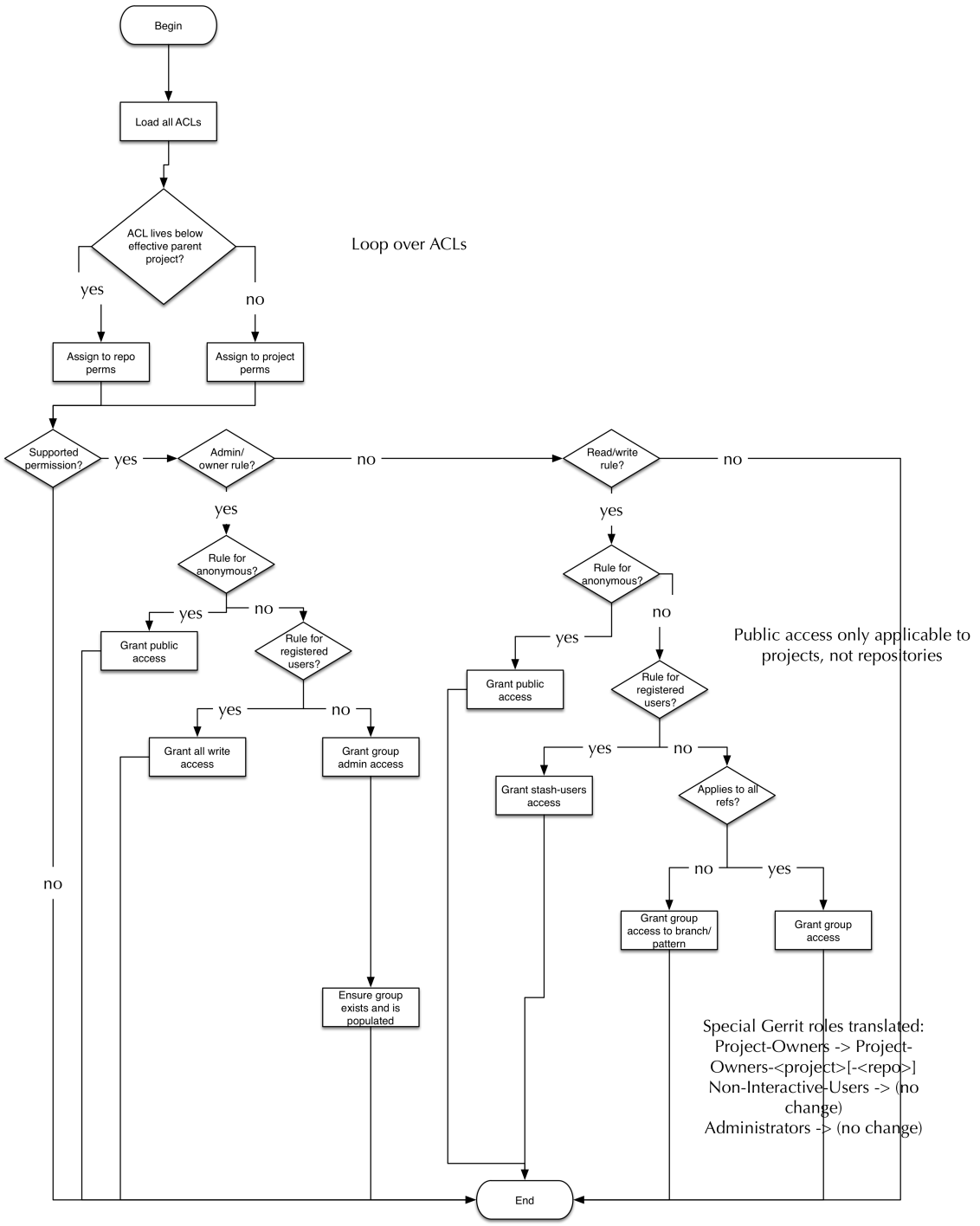
The overall process is shown below.



The process for importing a repository is shown below.



And finally the rule import process is shown below.



The code review import process is very straightforward: simply loop over each Gerrit review in the project and create an associated Stash pull request.

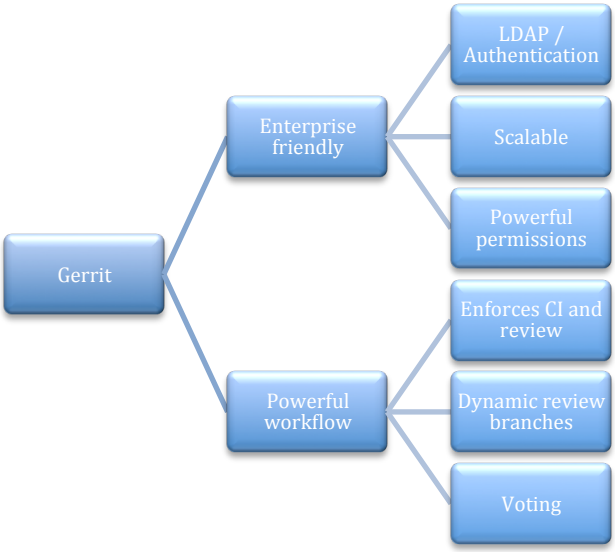
Logging

The plugin logs data to the normal Stash log files. You can get *INFO* or *DEBUG* level logging for the *com.go2group* package if necessary.

Gerrit Workflow in Stash

Gerrit is a popular code review tool introduced by Google to support the Android Open Source Project (AOSP). Despite a deficient user interface and the lack of repository browsing and issue management, it has a large footprint in enterprise Git development.

Why? There are several reasons.



Stash has largely matched Gerrit on being enterprise friendly. However the Stash native workflow is much more free-form than Gerrit.

A combination of configuration and plugins allows Stash to provide a more structured workflow similar to Gerrit. This guide will detail the required settings and plugins.

Parts List

Here's what you'll need to get started.

- A recent (2.5+) version of Stash
- Administrator access to Stash
- One plugin
- Pull request voting plugin provided by Go2Group
- Atlassian JIRA integrated with Stash
- Atlassian Bamboo integrated with Stash

Now on to configuration.

Configuration

First we'll detail the built-in settings, then move on to plugins.

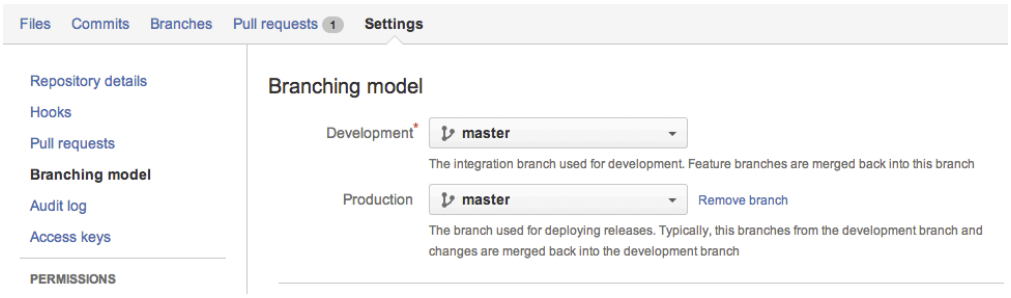
Stash Settings

On the Stash side we need to configure several things:

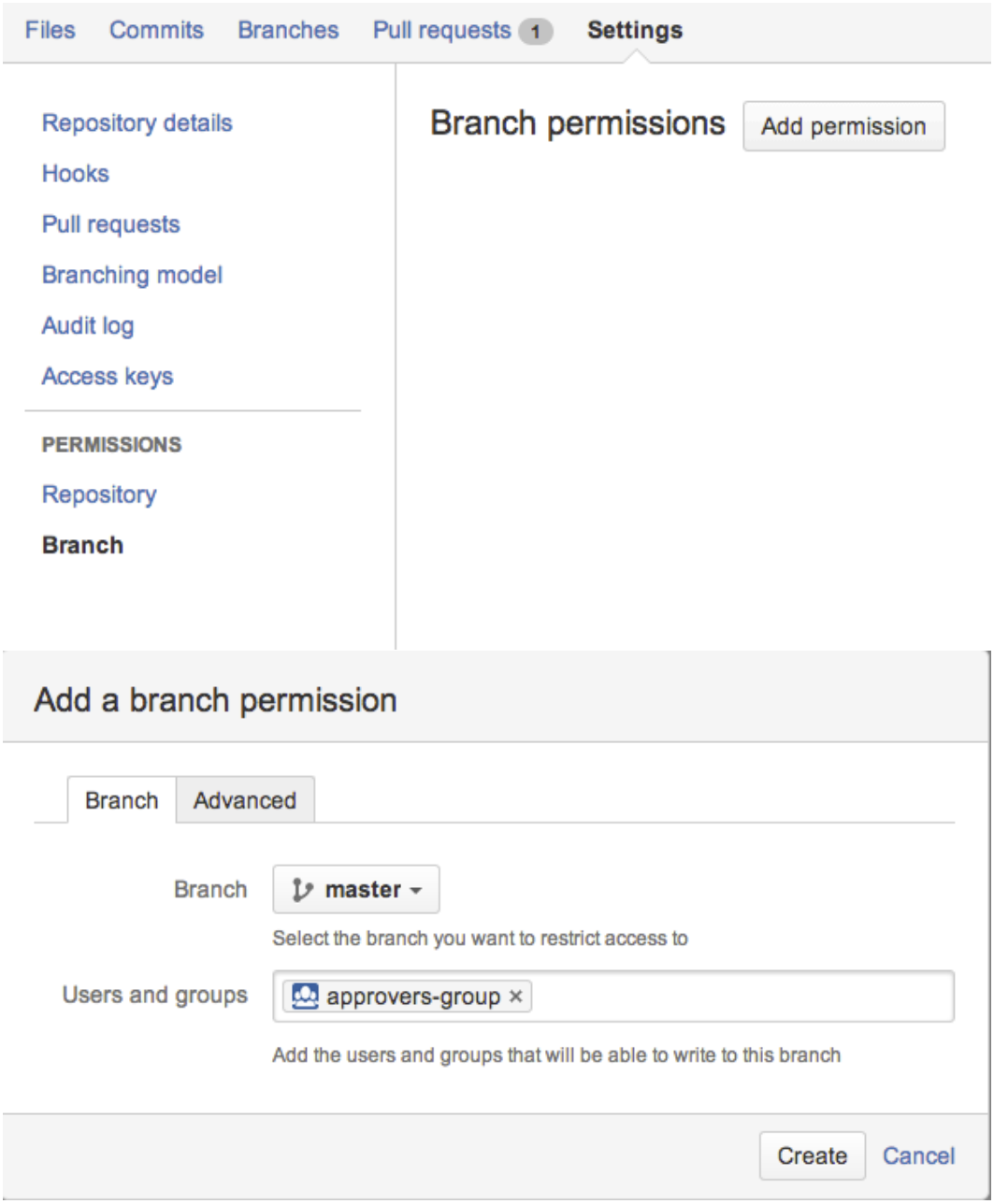
- Branch model and permissions
- Minimum number of approvers and builds
- Dynamic branch creation

BRANCH MODEL AND PERMISSIONS

First, configure Stash branching for each repository that should follow the Gerrit model. To follow the Gerrit model, default development and production releases both happen on `master`.

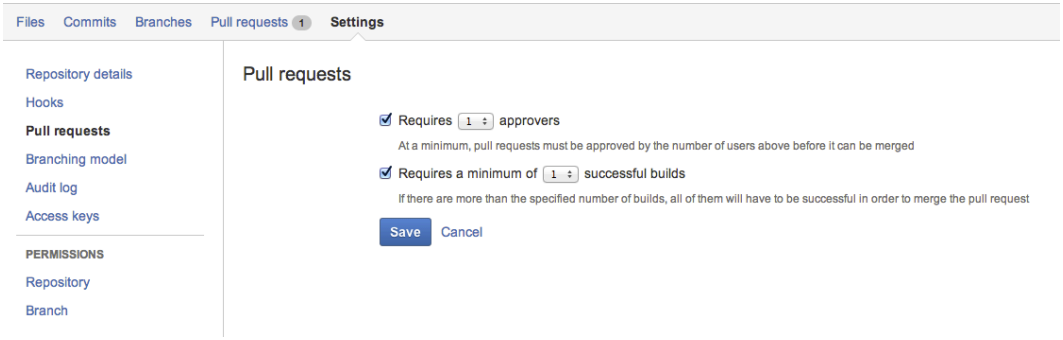


Branch-level permissions restrict writes to the master branch to your designated list of approvers. In Gerrit these would be team members with `+2` permission.



MINIMUM NUMBER OF APPROVERS AND BUILDS

Next, configure how many approvals and successful builds are necessary before a review can be merged. To follow the Gerrit model strictly require one approver and one build.

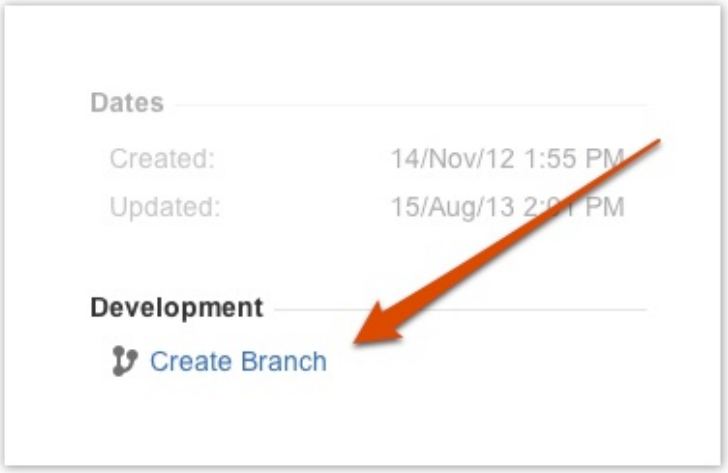


So far we have set up the necessary configuration for Gerrit `+2` style approval plus build verification, with a restricted set of approvers. If you prefer a more flexible voting-based workflow, then you can ignore the required approvers setting here and configure a workflow using the plugin described later in this document.

DYNAMIC BRANCH CREATION

Gerrit creates review branches automatically, a nice feature for developers who may be frustrated that they can't commit directly to `master`. You can achieve the same effect using JIRA integrated with Stash.

First, set up JIRA links to Stash per the Atlassian documentation. Now a developer can create a new branch automatically from JIRA.



Once the branch is created, the developer should run:

- `git remote update`
- `git checkout –b` *`branch`* `origin/`*`branch`*

Now the developer can work as on the new branch and then file a pull request when ready.

Plugins

We need the Go2Group plugin that provides two features:

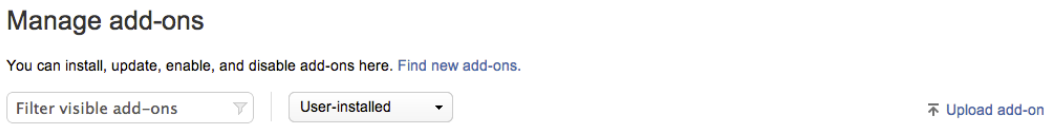
- Allowing voting
- Preventing merge of pull requests if configurable workflow rules are violated

ALLOWING VOTING

Next we'll install a plugin to let non-approvers vote on a pull request. You must configure at least one workflow in order to allow voting.

INSTALLATION

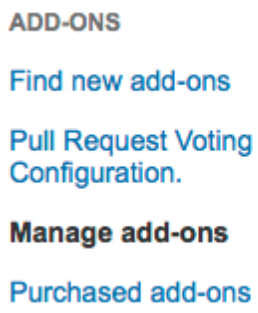
First install the plugin by uploading the plugin JAR file in the plugin manager.



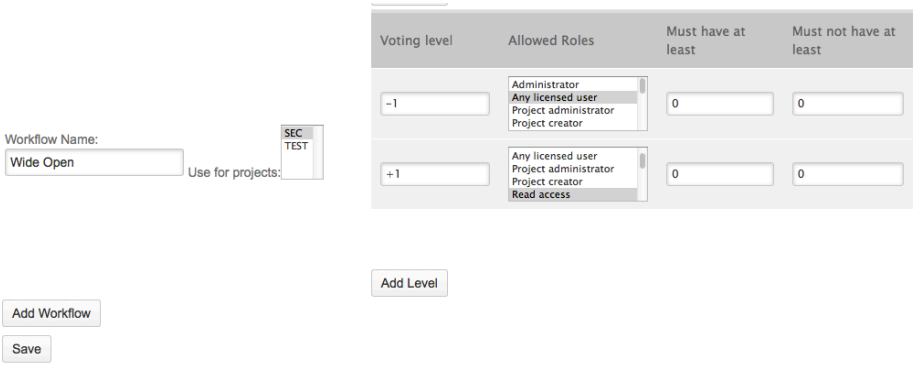
You should now see the plugin as installed with several modules enabled.

WORKFLOWS

Next configure one or more workflows by choosing the `Pull Request Voting Configuration` menu option.

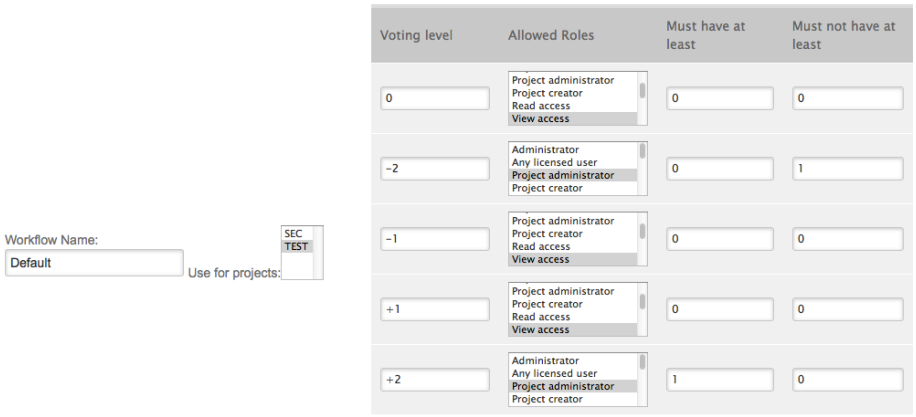


A workflow defines a set of voting levels, who is allowed to cast a vote for each level, and whether certain types of votes are required in order to approve a pull request. An example will help illustrate the concepts.



In the screenshot above we see a simple example. The workflow has a name and is assigned to the `SEC` project. It defines two voting levels, `-1` and `+1`. Any Stash user can cast a `-1` vote, while casting a `+1` vote requires project read access. The votes have no impact on approving the merge request.

Now let's look at a more sophisticated example that emulates the default Gerrit workflow.



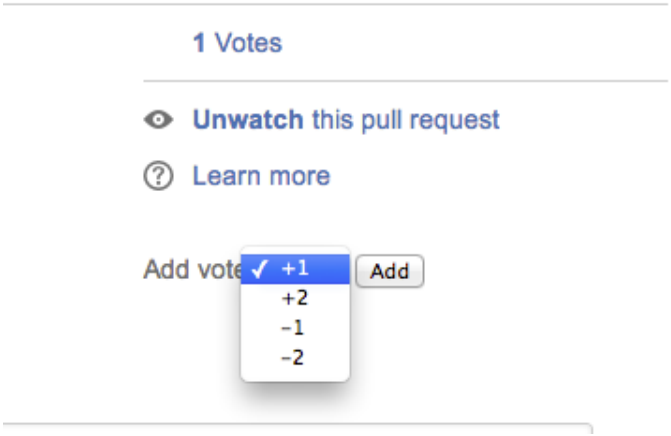
In this example we define a workflow named `Default` and assign it to the `TEST` project. It defines five voting levels: `-2, -1, 0, +1, +2`. Roughly these would correspond to expressing 'reject this', 'needs work', 'neutral', 'looks good', and 'approved'.

Anyone with at least `View` access to the project can cast a `-1, 0, +1` vote. But only project administrators (who are probably team leads) can cast a `-2` or `+2`. These votes are significant – in the final two columns of the configuration table we indicate that 1 `+2` vote is required to approve a pull request (must have at least 1 `+2` vote), but a pull request cannot be approved if it has any `-2` votes (must not have 1 `-2` vote).

You can define as many workflows as you need for different projects.

VOTING

Now on any open pull request you'll see a list of votes and the option to add your vote, assuming you have the proper access level.



Clicking on the vote summary shows a list of all current votes including the name of the person who cast the vote. Duplicate votes are allowed to show the history of team opinion. The votes are sorted with most recent first for clarity. This screen gives you a good summary of what other team members think about this pull request.

You can also remove a vote and vote again.

REJECTING MERGE REQUESTS BASED ON VOTING

Based on the workflows, the plugin may prevent a pull request from being merged if it lacks the right votes (or has the wrong votes).

You will see an information message on pull requests that cannot be merged.

DATA MIGRATION (OPTIONAL)

Transferring data from Gerrit to Stash can be a difficult process. Go2Group offers a flexible migration plugin that supports full and incremental migration of repositories, access control rules, and code review metadata.

Conclusion

By following this guide you'll get the best of Gerrit in Stash's more accessible framework:

- Dynamic review branch creation
- Enforcement of human and build code review for each commit with a designated set of approvers
- Collaborative team voting on pull requests with optional workflow enforcement

About Go2Group

Go2Group is a global provider of consulting services, third-party application integrations, data migrations, software testing, and training services in Application Lifecycle Management (ALM) systems. We've implemented thousands of enterprise-level migrations. We specialize in complex, multi-platform, ALM integration projects.

Our goal: To make it easy.

Our clients say, "We feel like you are part of our team." Go2Group is expert in the entire Atlassian product line and in best of breed solutions, including HP, IBM, Perforce, Salesforce, and SugarCRM environments.

An Enterprise and Platinum Atlassian Expert, we offer a full suite of services for all Atlassian products and are the world's largest reseller of Atlassian tools.

http://www.go2group.com/

Corporate Office, USA: 138 North Hickory Avenue, Bel Air, MD 21014

Hawaii: 7007 Hawaii Kai Drive, Suite C26, Honolulu, HI 96825

Japan: Le Premier Akihabara 11th Floor, 73 Kanda Neribei-cho, Chiyoda-ku, Tokyo 101-0022

Add VA, LA, Indiana.

Telephone:

877-442-4669 (toll free)

Email: sales@go2group.com

Go2Group and GSA

Atlassian products and Perforce SCM are available under the GSA purchasing program. Schedule 70 – Multiple Award Contract GS-35F-0209W. Expiration date: January 20, 2015

www.go2group.com