PERFORCE

Git Fusion Guide

2017.1

February 2017

Git Fusion Guide 2017.1

February 2017

Copyright © 1999-2017 Perforce Software.

All rights reserved.

Perforce software and documentation is available from `http://www.perforce.com/`. You can download and use Perforce programs, but you can not sell or redistribute them. You can download, print, copy, edit, and redistribute the documentation, but you can not sell it, or sell any documentation derived from it. You can not modify or attempt to reverse engineer the programs.

This product is subject to U.S. export control laws and regulations including, but not limited to, the U.S. Export Administration Regulations, the International Traffic in Arms Regulation requirements, and all applicable end-use, end-user and destination restrictions. Licensee shall not permit, directly or indirectly, use of any Perforce technology in or by any U.S. embargoed country or otherwise in violation of any U.S. export control laws and regulations.

Perforce programs and documents are available from our Web site as is. No warranty or support is provided. Warranties and support, along with higher capacity servers, are sold by Perforce Software.

Perforce Software assumes no responsibility or liability for any errors or inaccuracies that might appear in this book. By downloading and using our programs and documents you agree to these terms.

Perforce and Inter-File Branching are trademarks of Perforce Software.

All other brands or product names are trademarks or registered trademarks of their respective companies or organizations.

Any additional software included within Perforce software is listed in License Statements on page 115.

Table of Contents

About This Manual vii

See also vii
Perforce general resources vii
Please give us feedback vii

Chapter 1 Getting Started 1

What is Git Fusion? 1
Which installation should I use? 2

Chapter 2 Installing Git Fusion using the OVA 3

Prerequisites 3
Installation steps 3
Next Steps 4
Connecting the Git Fusion OVA installation to your Perforce service 6
Next steps 10
Pointing the Git Fusion HTTPS server to your own SSL certificate 10

Chapter 3 Installing Git Fusion using OS-Specific Packages 13

Prerequisites 13
Installation steps 13
Next steps 19

Chapter 4 Setting up Users 21

How do user permissions work? 21
Authentication 21
Authorization 21
Git Fusion users 21
Perforce protections 22
Permission groups 22
Permissions for git-fusion-user 23
Permission validation logic 23
Effect of permissions on push requests 24
What do I have to do? 25
Mapping Git users to Perforce accounts 26
Verify email address match 26
Use the Git Fusion User Map 27
Enable the unknown_git Perforce account 27

Authenticating Git users 28
Use existing HTTPS configuration with a different Perforce Service. 29
Validating your HTTP authentication setup 30
Logs 31
Ubuntu 31
CentOS and Red Hat 31
Authorizing Git users 31
Assign Perforce permissions to Git Fusion users 32
Create the permission groups and group p4 key 32
Populate the permission groups and set the group default p4 key 33
Enable pushes when Git authors lack Perforce permissions 34
Enforce Perforce read permissions on Git pull 35

Chapter 5 Setting up Repos 37

How does Git Fusion map Perforce depots to Git repos? 37
Configuring global defaults for repos 38
Configuring repos 54
Configure repos with a repo configuration file (p4gf_config) 55
Repo configuration file: key definitions and samples 56
Sample repo configuration files 57
Example 1: 58
Example 2: 58
Configure repos from a Perforce workspace 58
Use a Perforce depot path in a Git remote URL 60
Initializing repos on the Git Fusion server 62
Importing existing Git repos into Git Fusion 62
Creating a repo configuration file for import of existing repo 63
Importing an existing repo using a Perforce workspace or repo configuration file 63
Modifying repo configuration files safely 65
Converting a lightweight branch into a fully-populated branch 65
Enabling Git users to create fully-populated branches 66
Create a fully-populated branch only when a Git user explictly chooses to do so 67
Create a fully populated branch every time a Git user pushes a new branch 68
Controlling depot location of pushed branches 69
Examples 69
Example: project/branch hierarchy in Perforce 69
Example: Give each developer their own area in the Perforce depot. 69
Example: {user} without {git_branch_name} 70
Working with Perforce streams 71
Enabling stream import paths as Git submodules 72
Configure and generate submodules from import paths 72
Managing and troubleshooting submodules 73
What are these new virtual streams that appear in the stream depot? 73
How do I change the submodule URL (ssh-url, http-url)? 73
How do I remove submodules generated from import paths? 74
Adding preflight commits to reject pushes 74
Adding preflight hooks to reject pushes 76
Limiting push size and disk usage 78

Limits for a single push 78
Limit total Git Fusion disk usage 78
View current disk usage 79
Detecting Git copy/rename and translating to Perforce 79
Disconnecting a Git Fusion repo from the Perforce service 81
Deleting Git Fusion repos 81

Chapter 6 Additional Administrative Tasks 83

Configuring logging 83
Viewing changelist information 83
Managing Git Fusion p4 keys 84
Managing Git Fusion server IDs 84
Stopping the Git Fusion server 84
Preventing new Git Fusion sessions 85
Backing up and restoring Git Fusion 85
Adding Git Fusion and Perforce server components 86
Add Git Fusion servers 87
Special considerations for P4Broker 87
Git Fusion with Proxies, Replicas, and Edge servers 87
Delete repos on multiple hosts 88
Administering the Git Fusion OVA 88
Authentication and the OVA 88
Perforce Server and the OVA 88
Start and stop scripts 88
SSH key management console 89
Modify Perforce Server Triggers to Ignore Git Fusion 89
p4gf_config2 90
p4gf_environment.cfg 90
Environment Variables 91
Time Zone Configuration 91

Chapter 7 Tips for Git Users 93

Requirements, restrictions, and limitations 93
Providing SSH keys for Git Fusion authentication 93
Referencing Git Fusion repos 94
Sharing new Git branches with Perforce users 94
Referencing Perforce jobs in a commit 94
Using Git Fusion extension commands 95
How permissions affect the @list command 97
Using Swarm for code review 97
Create a Swarm review 97
Amend a Swarm review 98
View reviews created by other Git users 99
View amendments made by other Git users 100
Additional tips 100

Chapter 8 Troubleshooting 101

Clone issues 101
AppleDouble Header not recognized 101
.bashrc source line prevents cloning 101
File cannot be converted to specified charset 101
Missing @repo section 102
Spec depots cannot be mapped 102
General usage issues 102
Cannot terminate active process 102
Connection closed by remote host 102
Case sensitivity conflicts 102
git-fast-import crash 103
Git Fusion submit triggers are not installed 103
headType field does not exist 104
Locked repo caused by active process termination 104
Missing server-id file 105
Unicode-enabled client required 105
Git Fusion OVA issues 105
OVF cannot be parsed 105
P4D cannot be started 106
Push issues 106
Files not in client view 106
Files locked by git-fusion-reviews--non-gf 106
Merge commit requires rebasing 106
Not authorized for Git commit 107
Not permitted to commit 107
Password invalid or unset 107
Pushes prohibited after repo deleted or trigger removed 108
Script issues 108
Updating authorized keys file of multiple servers fails 108

Authenticating Git Users using SSH 109

Set up SSH authentication 109
Use a cron job to copy public keys to Git Fusion 110
Set up SSH authentication using the OVA's SSH key management console 110
Troubleshooting SSH key issues 112
Key or identity not recognized 112
No such Git repo 112
PTY request failed 112
Repo is not a Git repo 112
SSH format issues 113

License Statements 115

About This Manual

This guide tells you how to administer and use Perforce Git Fusion.

This guide is intended for people responsible for installing, configuring, and maintaining a Git Fusion integration with their organization's Perforce service, and assumes that you have intermediate-level Perforce administration experience. This guide covers tasks typically performed by a system administrator (for instance, installing and configuring the software and troubleshooting issues), as well as tasks performed by a Perforce administrator (like setting up Git Fusion users and configuring Git Fusion repos).

See also

For more information, see the following resources available at http://www.perforce.com

- *Perforce System Administrator's Guide*:

  http://www.perforce.com/perforce/doc.current/manuals/p4sag/index.html

- Perforce training courses.

  `http://www.perforce.com/instructor-led-training-overview`

- Video tutorials:

  `http://www.perforce.com/resources/tutorials`

Perforce general resources

To view all Perforce documentation:

- `http://www.perforce.com/documentation`

To obtain online help from within Perforce client programs:

- Command-line Client: Type **`p4 help`** from the command line.
- Graphical client applications: Click **Help** on the main menu bar.

For more information about consulting and technical support, see these web portals:

- Consulting

  `http://www.perforce.com/support-services/consulting-overview`

- Technical Support

  `http://www.perforce.com/support-services`

Please give us feedback

Please send any comments or corrections to `manual@perforce.com`

Chapter 1 Getting Started

This chapter includes the following topics:

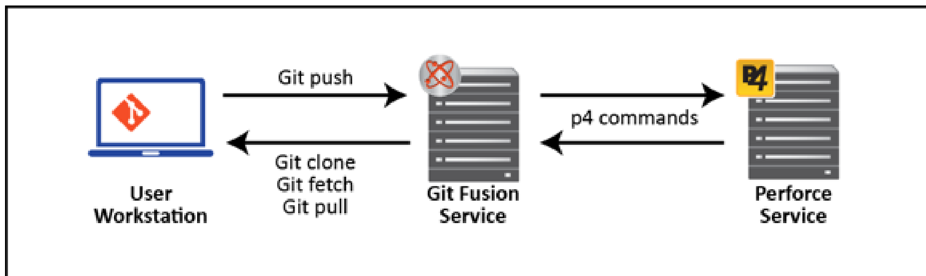
- What is Git Fusion?
- Which installation should I use?

What is Git Fusion?

Git Fusion is a Git remote repository service that uses Perforce Server as its back end.

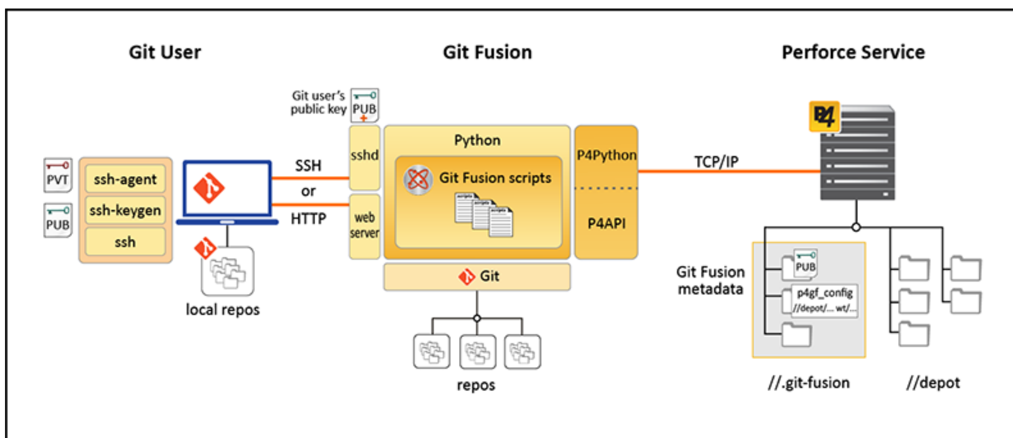
Git users interact with Git Fusion as they do with any other Git remote repository (repo), issuing standard Git commands to clone repos and transfer data. When a Git user pushes changes, Git Fusion translates those Git changes into Perforce changes and submits them to the Perforce depot. When a Git user pulls changes, Git Fusion translates the pull request into Perforce commands to download changes from the Perforce depot.

Figure 1.1. Git Fusion sits between Perforce and the Git user



Under the hood, a series of Python scripts manages Git user authentication and authorization, generates remote Git repos, and translates Git transactions into Perforce commands. The Perforce service is the repository of record not only for the data held in the remote Git repos, but also for the Git Fusion metadata -- including user authentication keys and repo configuration files, which define access and data generation rules for the repos.

Figure 1.2. Git Fusion architecture



For more information about how Git Fusion works, see:

- Setting up Users
- Setting up Repos

Which installation should I use?

There are three ways to install Git Fusion:

- **OVA (`git-fusion.ova`)**: A virtual machine image in the Open Virtual Appliance (OVA) format. This image includes both a Git Fusion and a Perforce Server instance. The Perforce Server instance is preloaded with sample data and configured for use with the included Git Fusion instance. There is a simple set of instructions to turn off this local Perforce service and connect the Git Fusion instance to your own external Perforce service.

  Use the OVA if any of the following apply to you:

  - You want to deploy to a virtual environment, and Ubuntu 12.04 LTS is an acceptable platform.
  - You are not an experienced Linux administrator; this install method requires the least amount of Linux expertise

- **Operating system-specific packages**: OS-specific packages, like RPM, provide a simplified way to install Git Fusion and its dependencies on a supported platform.

  We provide the following packages:

  - RPM package for CentOS/Red Hat 6.x.
  - Debian package for Ubuntu 12.04

  Use an OS-specific package if you want a streamlined, assisted installation on the supported operating systems.

- **Tarball (`git-fusion.tgz`)**: A compressed file that includes Git Fusion source and install scripts for Ubuntu or CentOS/Red Hat.

  Use the tarball if the following applies to you:

  - You are an experienced linux administrator with deep knowledge of configuring and troubleshooting services such as apache, ssh, and syslog.
  - You cannot use the OVA or OS-specific packages.
  - You are required to install and configure dependencies from source.
  - You need very fine control over installation paths.

  Contact Perforce Support for help with manually installing from the tarball.

Chapter 2 Installing Git Fusion using the OVA

Who is this for? The `git-fusion.ova` image includes both a Git Fusion and a Perforce Server instance. The Perforce instance is pre-loaded with sample data and configured for use with the included Git Fusion instance. There is a simple set of instructions to turn off this local Perforce service and connect the Git Fusion instance to your own external Perforce service.

Use the OVA if any of the following apply to you:

- You want to deploy to a virtual environment, and Ubuntu 12.04 LTS is an acceptable platform.
- You are not an experienced Linux administrator, as this installation method requires the least amount of Linux expertise.

Prerequisites

Note	See the Git Fusion release notes for the most comprehensive and recent software and hardware requirements.

- 64-bit operating system on the host computer.
- Virtualization framework that supports the import of `.ova` files.

Installation steps

1. **Download the `git-fusion.ova`:**

   http://www.perforce.com/downloads/git-fusion

   ftp://ftp.perforce.com/perforce/r15.1/bin.noarch/git-fusion.ova

2. **Import the OVA into your virtualization framework.**

   For production use, we recommend at least 4 cores and 16 GB memory.

   Configure as required for your site. Reinitialize the MAC address of all network cards if you are presented with the option.

3. **Start the Git Fusion virtual machine.**

4. **Set Linux account passwords at the prompts.**

   - `root`: root account on the virtual machine
   - `perforce`: service account used by the pre-loaded Perforce service
   - `git`: service account used by Git Fusion

5. **Automatic update to the pre-loaded Perforce service.**

   The Perforce service is installed and updated using a Debian package. At this point, the virtual machine will attempt to connect to package.perforce.com and check for an update to this package. If one is available, it will automatically be installed. No user action is required for this step.

You can update the Perforce service package at a later date with the following command:

```
$ sudo apt-get update && sudo apt-get install helix-p4d-base
```

6. **Enter a Git Fusion server ID or accept the default.**

   Server IDs are required to enable multiple Git Fusion instances to connect to the same Perforce service.

Note	If you want to change your Server ID at a later time, you can run `p4gf_super_init.py` with the `--id` option. For more information, see "Managing Git Fusion server IDs" on page 84.

When you have entered a server ID or accepted the default, the Git Fusion virtual machine completes its startup process. You now have a running Git Fusion instance connected to a local, pre-loaded Perforce service.

Make note of the IP address displayed in the console window. You can use it to perform your first **`git clone`** and to access the online SSH key management console. For information about the SSH key management console, see SSH key management console.

Next Steps

If you are using the OVA to install Git Fusion against another Perforce service:

- Connect your Git Fusion installation to your Perforce service.

  See "Connecting the Git Fusion OVA installation to your Perforce service" on page 6.

- (Optional) Point Git Fusion to your own signed SSL certificate.

  We deliver Git Fusion in the OVA with a self-signed SSL certificate. If you will be using this Git Fusion installation for anything other than testing and evaluation, we recommend that you reference your own signed SSL certificate in the Apache web server site file. See "Pointing the Git Fusion HTTPS server to your own SSL certificate" on page 10.

If you want to use the Perforce service included in the OVA, your installation is complete.

Now you can perform a **`git clone`** of the Talkhouse sample repo. When Git Fusion receives the clone request, it will create a new Git repo out of existing files in Perforce, and deliver the resulting repo to the Git client. For authorization, you'll first need to create a Perforce user.

1. Create a Perforce user.

   a. Log into the Git Fusion virtual machine as `root` (or open a shell on another system that has the p4 client installed).

   b. Create a Perforce user:

```
p4 -p ipaddress:1666 -u super user -f username
```

where ***ipaddress*** is the IP address displayed in the VM console window (the one you noted when you installed the OVA).

Note that **super** is a pre-configured Perforce super user.

Enter **:wq** to save the new user.

2. Assign a password.

```
p4 -p ipaddress:1666 -u username passwd
```

3. Clone the Talkhouse repo.

   You can perform the clone from any computer with a Git installation and network access to the Git Fusion virtual machine.

   a. First tell Git not to verify the SSL certification.

   ```
   $ export GIT_SSL_NO_VERIFY=true
   ```

   Note

   We deliver Git Fusion in the OVA with a self-signed SSL certificate. Exporting this environment variable tells Git not to verify the SSL certification in the current shell session only. To tell Git never to verify SSL certificates, use the following command:

   ```
   git config --global http.sslVerify false
   ```

   To point Git Fusion to your own signed SSL certificate (recommended if you will be using this Git Fusion installation for anything other than demonstration purposes), see "Pointing the Git Fusion HTTPS server to your own SSL certificate" on page 10.

   b. Perform the clone operation using the IP address (*without the port number*) displayed in the Git Fusion VM console window (the one you noted when you installed the OVA.

   ```
   git clone https://ip_address/Talkhouse
   ```

   When prompted, enter the user name and password you created in Step 1.

To learn more about adding users and setting up repos see:

- Setting up Users

- Setting up Repos

To learn more about how to work with the Perforce service included with the OVA, see: "Perforce Server and the OVA" on page 88

Connecting the Git Fusion OVA installation to your Perforce service

Who is this for? You want to use the Git Fusion instance that you installed with the OVA against a Perforce service on another machine, such as your existing production Perforce service. For this installation, you need some Perforce and Linux administration experience.

Prerequisites for the Perforce service:

Note	See the Git Fusion release notes for the most comprehensive and recent software and hardware requirements.

- See the Git Fusion release notes for current Perforce Server (P4D) version requirements.
- You must have root level access to the server(s) that host(s) your Perforce service, as well as Perforce `super` user access.
- Python 2.6+, 3.2+, or 3.3+ on the server hosting the Perforce service triggers.

1. **Turn off the local Perforce service.**

   Log into the Git Fusion virtual machine as `root` and run:

   ```
   # service p4d stop
   # update-rc.d p4d disable
   ```

2. **Update the Apache web service site configuration file to add your Perforce service.**

Note	If you prefer to use SSH rather than HTTPS authentication, skip this step and see "Set up SSH authentication using the OVA's SSH key management console" on page 110.

   a. Stop the Apache web service.

   ```
   # service apache2 stop
   ```

   b. Open the `git-fusion-ssl` Apache site configuration file.

   ```
   # vi /etc/apache2/sites-available/git-fusion-ssl
   ```

   c. Edit the `AddExternalAuth` line to include the full hostname, port, and P4CHARSET of your Perforce service.

```
AddExternalAuth p4_auth "/opt/perforce/git-fusion/libexec/
p4auth.sh myperforceserver:port charset"
```

d. Save your changes and exit **vi**.

```
:wq
```

e. Start the Apache web service.

```
# service apache2 start
```

3. Run the `configure-git-fusion.sh` script.

```
# /opt/perforce/git-fusion/libexec/configure-git-fusion.sh
```

The script prompts you for the following:

- Whether you want to connect to an existing Perforce service or create a new one:

  Type **remote** to use an existing Perforce service on another host.

- How you want to handle the Perforce change owner for git commits authored by non-Perforce users.

  Enter **reject** to reject push which contains commits authored by non-Perforce users (default)

  Enter **pusher** to accept commits authored by non-Perforce users and set the change owner to the pusher

  Enter **unknown** to accept commits authored by non-Perforce users and set the change owner to 'unknown_git'. This option will also create 'unknown_git' Perforce user.

Note	Note: The actual pusher, author, and committer are always recorded in the Perforce changelist description.

- Perforce service's hostname and port (`P4PORT`).

- Perforce **super** user name and password to enable Git Fusion to run administrative `p4` commands.

- Git Fusion time zone, in Olson format.

  Set it to your Perforce service time zone or accept the default, which is the Git Fusion host machine's time zone.

Git Fusion uses the Olson time zone format, as recognized by `pytz` (for example, `US/Pacific` rather than `PST`).

- Whether you want to configure HTTPS authentication

  Enter `no`, since HTTPS authentication is already configured on the Git Fusion OVA

- Single password to be shared by any new Perforce users that Git Fusion creates to enable it to interact with the Perforce service.

  The first time you install and configure a Git Fusion instance for use with any given Perforce service, the script creates the users `git-fusion-user`, `git-fusion-reviews-`*`server-id`*, `git-fusion-reviews--non-gf`, and `git-fusion-reviews--all-gf`.

Note

You can set individual passwords after the configuration script is finished by issuing the following command:

```
p4 -p myperforceserver:port -C charset -u user_name passwd
```

When the script is finished, it congratulates you and suggests that you configure your Perforce service to use Git Fusion's atomic push triggers.

For detailed information about the functions performed by the configuration script, along with information about rerunning it to change your initial configuration settings, see `configure-git-fusion.sh --help`.

4. **Configure Git Fusion triggers in the Perforce service to support atomic pushes.**

Important

We recommend that any submit triggers running on your Perforce service exclude changes that are submitted by `git-fusion-user`. These include `change-submit, change-commit,`and `change-content` triggers that enforce a local policy, like requiring jobs, specific content, or specific formatting in the changelist description. Such triggers can interrupt Git Fusion in the middle of a push, which will damage the repository as replicated within Perforce.

If you cannot exclude `git-fusion-user` from these triggers, you can instead create preflight hooks that reject git pushes based on local policies derived from your current submit triggers. For more information, see Adding preflight commits to reject pushes.

a. Copy the `p4gf_submit_trigger.py` script and the high performance wrapper `p4gf_submit_trigger_wrapper.sh` from `/opt/perforce/git-fusion/libexec` to the server hosting the Perforce service.

Note

The wrapper script is written in Bash, which has a much lower startup overhead than Python. The wrapper quickly determines if the triggered event is related to a Git Fusion operation, in which case a call to the Git Fusion Python submit trigger script is avoided entirely. Although the

> Python trigger script can also be called directly, this arrangement improves Git Fusion performance. When used, the Bash trigger wrapper path is inserted before the Python trigger path and its arguments.

> **Note**
>
> If your Perforce service is hosted on a platform supported by Git Fusion packages, you can install the `helix-git-fusion-trigger` package on it to satisfy this step.

b. Log in to the server hosting the Perforce service as a user with `sudo` privileges.

c. Run the `p4gf_submit_trigger.py` script to install and configure the triggers.

```
$ p4gf_submit_trigger.py --install  myperforceserver:port perforce_super_user password
```

The script does the following:

- Creates login tickets for the Perforce users `git-fusion-user`, `git-fusion-reviews-`*`server-id`*, `git-fusion-reviews--non-gf`, and `git-fusion-reviews--all-gf`.
- Creates a trigger configuration file, `p4gf_submit_trigger.cfg`, in the same directory as the trigger script, that holds your `P4PORT` and `P4CHARSET` variables, as well as the path to the `P4` binary.
- Adds Git Fusion trigger entries to the Perforce Triggers table. If the high performance Bash wrapper is present, triggers will be configured to call it, otherwise they will be configured to call the Python script directly.
- If your Perforce service is SSL-enabled, generates the `p4gf_submit_trigger.trust` file in the same directory as the trigger script to manage the trust of the SSL connection.
- Sets the p4 key that verifies that the trigger was installed.

If you are upgrading an existing Git Fusion installation, the script replaces your old Git Fusion triggers with new ones. It does not touch any other triggers.

For more details about `p4gf_submit_trigger.py`, see `p4gf_submit_trigger.py --help`.

For more information about triggers, see the *Perforce System Administrator Guide*, "Scripting Perforce: Triggers and Daemons."

5. **Verify the configuration from the Git Fusion server.**

Switch to the Git Fusion service account (`git`) on the Git Fusion server and run `p4gf_super_init.py`.

```
# su - git
$ p4gf_super_init.py --user perforce_super_user
```

The script should report that everything has been created, already exists, or is up to date.

Next steps

You are now ready to:

- (Optional) Point the Git Fusion HTTPS server to your own SSL certificate. See "Pointing the Git Fusion HTTPS server to your own SSL certificate" on page 10.
- Set up users. See Chapter 4, "Setting up Users" on page 21.
- Set up repos. See Chapter 5, "Setting up Repos" on page 37.

Pointing the Git Fusion HTTPS server to your own SSL certificate

We deliver Git Fusion in the OVA with a self-signed SSL certificate. If you will be using this Git Fusion installation for anything other than testing and evaluation, we recommend that you use your own signed SSL certificate. If you are using SSH for authentication, you can skip this task.

Note	If you keep the default self-signed SSL certificate, you must tell Git not to verify the SSL certification when you perform Git commands against Git Fusion repos, either on a per-session basis (by running `export GIT_SSL_NO_VERIFY=true`) or for all sessions (by running `git config --global http.sslVerify false`).

To enable Git Fusion to use your own signed SSL certificate:

1. **Stop the Apache web service:**

   Log into the Git Fusion virtual machine as `root` and run:

   ```
   # service apache2 stop
   ```

2. **Open the `git-fusion-ssl` Apache site configuration file.**

   ```
   # vi /etc/apache2/sites-available/git-fusion-ssl
   ```

3. **Edit the lines `SSLCertificateFile` and `SSLCertificateKeyFile` to point to your signed SSL certificate and key.**

   ```
     SSLCertificateFile /path/to/your_certificate_file
     SSLCertificateKeyFile /path/to/your_private_key_file
   ```

4. **Save your changes and exit `vi`.**

   ```
   :wq
   ```

5. **Start the Apache web service.**

```
# service apache2 start
```

Chapter 3 Installing Git Fusion using OS-Specific Packages

Who is this for? Operating system-specific packages provide a simplified way to install Git Fusion and its dependences on a supported platform.

We provide the following packages:

- RPM packages for CentOS/Red Hat.
- Debian packages for Ubuntu.

Prerequisites

Note	See the Git Fusion release notes for the most comprehensive and recent software and hardware requirements.

- Requirements for the Git Fusion server
  - Linux Intel x86_64
    - Ubuntu 12.04 LTS
    - Ubuntu 14.04 LTS
    - CentOS or Red Hat 6.x
    - CentOS or Red Hat 7.x not recommended: http setup too difficult
  - For production use, we recommend at least 4 cores and 16 GB memory.
  - You must have root level access to the server that hosts Git Fusion.
  - Internet connection.
- Requirements for the Perforce service
  - See the Git Fusion release notes for current Perforce Server (P4D) version requirements.
  - You must have root level access to the server(s) that host(s) your Perforce service, as well as Perforce `super` user access.
  - Python 2.6+, 3.2+, or 3.3+ on the server hosting the Perforce service triggers.

Installation steps

Note	On CentOS/Red Hat, default SELinux security policies may deny Git Fusion packages access to resources that they need to install. If your organization's security

policy permits it, disabling SELinux may help to simplify installation. If you require SELinux, contact Perforce customer support for assistance.

1. **Import the Perforce package signing key.**

   As root (or a user with `sudo` privileges), run one of the following:

   For RPM:

```
# rpm --import https://package.perforce.com/perforce.pubkey
```

   For Debian:

```
$ wget -qO - https://package.perforce.com/perforce.pubkey | sudo apt-key add -
```

   For information about how to verify the authenticity of the signing key, see: https://www.perforce.com/perforce-packages.

2. **Add the Perforce package repository.**

   - For RPM packages, create a file called `/etc/yum.repos.d/perforce.repo` with the following content:

```
[perforce]
name=Perforce
baseurl=http://package.perforce.com/yum/rhel/{version}/x86_64/
enabled=1
gpgcheck=1
```

   Note Replace {version} with the major version of your distribution, either '6' or '7'. CentOS and Red Hat 7.x. not recommended: http setup too difficult.

   - For the Debian package, create a file called `/etc/apt/sources.list.d/perforce.list` with the following line:

```
deb http://package.perforce.com/apt/ubuntu {version} release
```

   Note Replace {version} with the codename of your distribution, either 'precise' (Ubuntu 12.04) or 'trusty' (Ubuntu 14.04).

   Update the package repository:

```
$ sudo apt-get update
```

3. **Install the Git Fusion package.**

There are two package files to choose from:

- `helix-git-fusion`: installs the most recent stable version of Git Fusion and creates a Git Fusion service account named **git**. This is the Unix account that Git users will use when they run a Git command against Git Fusion using SSH. It is also the account that administrators will use to run Git Fusion utility scripts. The package will create the account with home directory in `/opt/perforce/git-fusion/home/perforce-git-fusion`.

  After installation, you should immediately set a strong password for this new **git** user. Your SSH users will use SSH keys for access, and so they will not need to know this password.

  For CentOS and Red Hat, run:

  ```
  # yum install helix-git-fusion
  # passwd git
  ```

  For Ubuntu, run:

  ```
  $ sudo apt-get install helix-git-fusion
  $ sudo passwd git
  ```

- `helix-git-fusion-base`: installs the most recent stable version of Git Fusion and does *not* create a **git** user.

  This package enables you to configure a Git Fusion service account with whatever name you want when you run the `configure-git-fusion.sh` script (in the next step).

  For CentOS and Red Hat, run:

  ```
  # yum install helix-git-fusion-base
  ```

  For Ubuntu, run:

  ```
  $ sudo apt-get install helix-git-fusion-base
  ```

The packages install Git Fusion and its dependencies under `/opt/perforce/git-fusion` to prevent conflicts with any system versions of Git and Python.

4. **Run the `configure-git-fusion.sh` script.**

```
# /opt/perforce/git-fusion/libexec/configure-git-fusion.sh
```

The script prompts you for the following:

- Whether you want to connect to an existing Perforce service or create a new one.

Enter **new** to install and configure a new Perforce service on the machine that hosts Git Fusion.

Enter **local** to connect to a Perforce service existing on the same machine hosting Git Fusion.

Enter **remote** to connect to a Perforce service existing on another machine.

- How you want to handle the Perforce change owner for git commits authored by non-Perforce users.

  Enter **reject** to reject push which contains commits authored by non-Perforce users (default)

  Enter **pusher** to accept commits authored by non-Perforce users and set the change owner to the pusher

  Enter **unknown** to accept commits authored by non-Perforce users and set the change owner to 'unknown_git'. This option will also create 'unknown_git' Perforce user.

Note	Note: The actual pusher, author, and committer are always recorded in the Perforce changelist description.

- Perforce service's hostname and port (**P4PORT**).
- Perforce **super** user name and password to enable Git Fusion to run administrative **p4** commands.
- If the script is creating a new Perforce service, the path to your preferred location for the Perforce Server root directory.
- Git Fusion time zone, in Olson format.

  Set it to your Perforce service time zone or accept the default, which is the Git Fusion host machine's time zone.

  Git Fusion uses the Olson time zone format, as recognized by **pytz** (for example, **US/Pacific** rather than **PST**).

- Whether you want to configure HTTPS authentication

  Enter **yes** if you installed the **helix-git-fusion** package and satisfy the requirements for HTTPS configuration.

  Enter **no** if you do not wish to use HTTPS authentication, installed **helix-git-fusion-base** package, or do not satisfy any of the requirements for HTTPS configuration.

Note	See "Authenticating Git users" on page 28 and *Git Fusion release notes* for HTTPS configuration requirements.

- If you installed Git Fusion using the **helix-git-fusion-base** package, the name and password of the system account that will run Git commands for Git Fusion (**git**, by default).

If you installed Git Fusion using the `helix-git-fusion` package, which creates the Git Fusion service account `git`, the script configures that user without prompting you.

- Single password to be shared by any new Perforce users that Git Fusion creates to enable it to interact with the Perforce service.

  The first time you install and configure a Git Fusion instance for use with any given Perforce service, the script creates the users `git-fusion-user`, `git-fusion-reviews-`*`server-id`*, `git-fusion-reviews--non-gf`, and `git-fusion-reviews--all-gf`.

Note

You can set individual passwords after the configuration script is finished by issuing the following command:

```
$ p4 -p myperforceserver:port -C charset -u user_name passwd
```

When the script is finished, it congratulates you and suggests that you configure your Perforce service to use Git Fusion's atomic push triggers.

For detailed information about the functions performed by the configuration script, along with information about rerunning it to change your initial configuration settings, see `configure-git-fusion.sh --help`.

5. **Configure Git Fusion triggers in the Perforce service to support atomic pushes.**

Important

We recommend that any submit triggers running on your Perforce service exclude changes that are submitted by `git-fusion-user`. These include `change-submit`, `change-commit`, and `change-content` triggers that enforce a local policy, like requiring jobs, specific content, or specific formatting in the changelist description. Such triggers can interrupt Git Fusion in the middle of a push, which will damage the repository as replicated within Perforce.

If you cannot exclude `git-fusion-user` from these triggers, you can instead create preflight hooks that reject git pushes based on local policies derived from your current submit triggers. For more information, see Adding preflight commits to reject pushes.

a. Copy the `p4gf_submit_trigger.py` script and the high performance wrapper `p4gf_submit_trigger_wrapper.sh` from `/opt/perforce/git-fusion/libexec` to the server hosting the Perforce service.

Note

The wrapper script is written in Bash, which has a much lower startup overhead than Python. The wrapper quickly determines if the triggered event is related to a Git Fusion operation, in which case a call to the Git Fusion Python submit trigger script is avoided entirely. Although the Python trigger script can also be called directly, this arrangement improves Git Fusion performance. When used, the Bash trigger wrapper path is inserted before the Python trigger path and its arguments.

> **Note**
>
> If your Perforce service and Git Fusion are hosted on the same server, you don't need to copy the scripts.

> **Note**
>
> If your Perforce service is hosted on a platform supported by Git Fusion packages, you can install the **helix-git-fusion-trigger** package on it to satisfy this step.

b. Log in to the server hosting the Perforce service as a user with **sudo** privileges.

c. Run the **p4gf_submit_trigger.py** script to install and configure the triggers.

```
$ p4gf_submit_trigger.py --install  myperforceserver:port perforce_super_user password
```

The script does the following:

- Creates login tickets for the Perforce users **git-fusion-user**, **git-fusion-reviews-*server-id***, **git-fusion-reviews--non-gf**, and **git-fusion-reviews--all-gf**.
- Creates a trigger configuration file, **p4gf_submit_trigger.cfg**, in the same directory as the trigger script, that holds your **P4PORT** and **P4CHARSET** variables, as well as the path to the **P4** binary.
- Adds Git Fusion trigger entries to the Perforce Triggers table. If the high performance Bash wrapper is present, triggers will be configured to call it, otherwise they will be configured to call the Python script directly.
- If your Perforce service is SSL-enabled, generates the **p4gf_submit_trigger.trust** file in the same directory as the trigger script to manage the trust of the SSL connection.
- Sets the p4 key that verifies that the trigger was installed.

If you are upgrading an existing Git Fusion installation, the script replaces your old Git Fusion triggers with new ones. It does not touch any other triggers.

For more details about **p4gf_submit_trigger.py**, see **p4gf_submit_trigger.py --help**.

For more information about triggers, see the *Perforce System Administrator Guide*, "Scripting Perforce: Triggers and Daemons."

6. **Verify the configuration from the Git Fusion server.**

Switch to the Git Fusion service account (**git**) on the Git Fusion server and run **p4gf_super_init.py**.

```
# su - git
$ p4gf_super_init.py --user perforce_super_user
```

The script should report that everything has been created, already exists, or is up to date.

Next steps

You are now ready to:

- Set up users. See Setting up Users
- Set up repos. See Setting up Repos

Chapter 4 Setting up Users

After you install Git Fusion, you must map your Git users to Perforce accounts and set permissions for them.

This chapter discusses the following topics:

- How do user permissions work?
- What do I have to do?
- Mapping Git users to Perforce accounts
- Authenticating Git users
- Authorizing Git users

How do user permissions work?

Git Fusion authenticates users through HTTP or SSH and authorizes them for pull and push transactions through Perforce group membership and permissions.

Authentication

Git Fusion uses HTTP or SSH to authenticate Git client requests (such as **`git clone`**, **`git pull`**, and **`git push`**). In a standard Git implementation, each Git user connects to a remote repo by establishing an individual account on the server that hosts the repo. In Git Fusion, all of your organization's Git users gain access through a Git Fusion service user UNIX account (**`git`**, in the default OVA installation) on the Git Fusion server, where either a web server or SSH daemon performs the authentication and invokes a python script that redirects the request to Git Fusion.

Authorization

While authentication to the Git Fusion server is handled by HTTP or SSH, access to Git Fusion repos is handled by Perforce permissions and groups.

If you are not familiar with Perforce permissions functionality, see the *Perforce System Administrator Guide*, Administering Perforce: Protections.

Git Fusion users

When we discuss Git Fusion permissions, it is helpful to understand the following Git roles -- and to understand that a single Git user can occupy more than one of these roles in any given Git transaction:

- **Git author:** A user who changes a file. Typically this is an individual developer or contributor.
- **Git committer:** A user who checks a change into Git. Usually this is the same user as the Git author, but in some workflows, a Git author lacks easy network or write access to the main Git repo, so the author emails a patch or sends the change to a coworker, who then commits that change to the repo on behalf of the author.
- **Git puller:** A user who runs git clone, git pull, or git fetch to download data into their own Git repo from another Git repository such as Git Fusion.

- **Git pusher:** A user who runs git push to send changes from their own Git repo to a Git remote repo such as Git Fusion. The changes being pushed are often authored and committed by the same person doing the pushing, but not always; it is common for Git users to pull changes from coworkers, so the pushed changes might be authored or committed by anyone.

It is also important to understand that, while Git Fusion maps Git users to Perforce users for authorization, Git Fusion connects to Perforce as a single user, **`git-fusion-user`**, which functions as `P4USER` for all Perforce operations.

Perforce protections

Any Git user who pushes changes to a Git Fusion remote repo must have **`write`** access to the Perforce depot locations that comprise the repo. By default, Git pull transactions do not require **`read`** access. Permission to pull from Git Fusion remote repos is handled instead by membership in a Git Fusion pull group (see Permission groups on page 22). However, there is an option to require that all pull transactions check that the puller has Perforce read permissions for the depot locations included in the repo. For more information, see Enforce Perforce read permissions on Git pull .

The `git-fusion-user` must have write access to all of the Perforce depot locations that include Git Fusion repo content, as well as the `//.git-fusion` depot, where Git Fusion metadata is stored.

Permission groups

Git Fusion uses Perforce *groups* to enforce what the user can push and pull. Each Git puller and pusher maps to a corresponding Perforce user, and that Perforce user must (with exceptions, noted below) be a member of a pull or push permissions group. Pushers must also have write access to the Perforce depot locations included in the repo.

Git Fusion provides three mechanisms for determining pull and push permissions:

- **Repo-specific permission groups:** grant a Perforce user pull (`git-fusion-`*`repo_name`*`-pull`) or pull and push (`git-fusion-`*`repo_name`*`-push`) permissions for a specific Git Fusion repo view (which represents a specified set of files in the Perforce depot).

- **Global permission groups:** grant a Perforce user pull (`git-fusion-pull`) or pull and push permissions (`git-fusion-push`) for *all* Git Fusion repos.

- **Default permissions p4 key:** grants all Perforce users the ability to pull or push -- or prohibits users from doing either action -- by use of a p4 key, (`git-fusion-permission-group-default`).

  If you do not assign a user to either a repo-specific or global group, Git Fusion automatically assigns the user the permission specified by the p4 key. If you use the p4 key to remove access from all users, you can restrict users to repo-specific permissions. You can also use it to give access to all users when you have no need for repo-specific permissions.

Pull groups enable **`git clone`**, **`git fetch`**, and **`git pull`**. Push groups add **`git push`**.

When determining a user's pull and push permissions, Git Fusion iterates through these mechanisms, from the repo-specific groups to the global groups to the p4 key, continuing until it finds and returns the first matching permission.

The global groups and the default permissions p4 key are automatically generated by the User and Client Initialization script (`p4gf_init.py`). The repo-specific permission groups are automatically

generated by the Repo Initialization script (`p4gf_init_repo.py`). You can run these scripts any time after Git Fusion has been initialized into your Perforce service with the Super User Initialization script (`p4gf_super_init.py`). If `p4gf_init.py` has not been run, `p4gf_init_repo.py` will invoke it automatically. If neither has been run, the first push or clone against the Git Fusion server invokes them both automatically.

Permissions for git-fusion-user

The Super User Initialization script (`p4gf_super_init.py`) automatically creates the `git-fusion-user` account, which performs all transactions with the Perforce service. The script grants `admin` privileges to this account and inserts the `git-fusion-user` in the bottom row of the protections table. Consequently, Git users are able to `read` (pull) and `write` (push) based on the permissions set for their corresponding Perforce user and also the permissions assigned to the `git-fusion-user`.

Note that the `git-fusion-user` must be the owner of all global and repo-specific permission groups.

Permission validation logic

Git Fusion and the Perforce service validate pull and push requests using the following logic:

1. Prior to processing a pull or push request, the Perforce service verifies that the `git-fusion-user` has the appropriate permissions for that action. If not, the Perforce service rejects the request.

2. Git Fusion verifies that the Git user maps to a Perforce user with appropriate pull or push permission for that Git Fusion repo.

   Git Fusion iterates through the repo-specific permission groups, the global permission groups, and the default permission p4 key until it finds and returns the first matching permission.

3. If the request is a pull (**`git clone`**, **`git fetch`**, **`git pull`**), permission group membership or the default permission p4 key value determines access to the repo; Git Fusion does not check the Perforce protections table for the puller's `read` access to the files in the Perforce depot, unless an administrator has enabled the option to require a read-access check for all pull transactions. For more information, see Enforce Perforce read permissions on Git pull .

4. If the request is **`git push`**, the Git Fusion server verifies that both the Git author and the Git pusher have Perforce `write` permissions set for the files in the depot. If either user does not, the Git Fusion server rejects the push.

   The requirement that the Git author have write permission is subject to some exceptions. The push can succeed even if the Git author has no associated Perforce user -- or if the Git author's Perforce user does not have `write` permission -- if one or more of the following criteria are met:

   - The **`unknown_git`** user exists and has write permissions.

   - The **`ignore-author-permissions`** property is set to `Yes` in the repo configuration file.

   - The **`change-owner`** property is set to `pusher` in the repo configuration file.

   For more information about `unknown_git`, `ignore-author-permissions`, and `change-owner`, see Enable pushes when Git authors lack Perforce permissions

Effect of permissions on push requests

The following table shows how Git Fusion handles push requests, depending on permissions set for the Git pusher, the Git author, and the `unknown_git` user, along with the `ignore-author-permissions` property for the repo.

Note that this table does not display the effect of setting the `change-owner` property to `pusher`. This is because that setting makes the other settings irrelevant: as long as the Git pusher has `write` access to the correct locations in the Perforce depot, the change will be submitted successfully, with the Git pusher as changelist owner and the author's, committer's, and pusher's names appearing in the changelist description.

A dash (-) indicates that the column has no significance in a row where it appears; the value could be Yes or No.

Table 4.1. Effect of permissions on push requests

Git pusher has write access	P4USER unknown_git exists	P4USER unknown_git has write access	Git author is P4USER	Git author has write access	ignore-author-permissions flag is set	Result
Yes	-	-	Yes	Yes	-	Changelists appear as submitted by Git author's Perforce user ID, and the author's, committer's, and pusher's names appear in the changelist.
Yes	-	-	Yes	No	Yes	Changelists appear as submitted by Git author's Perforce user ID, and the author's, committer's, and pusher's names appear in the changelist.
Yes	Yes	Yes	No	-	-	Changelists appear as submitted by `unknown_git`, and the author's, committer's, and pusher's names appear in the changelist.
Yes	Yes	No	No	-	Yes	Changelists appear as submitted by `unknown_git`, and the author's, committer's, and pusher's names appear in the changelist.

Git pusher has write access	P4USER unknown_git exists	P4USER unknown_git has write access	Git author is P4USER	Git author has write access	ignore-author-permissions flag is set	Result
Yes	-	-	Yes	No	No	Git Fusion prohibits the push and displays the following error message: `remote: import failed: user` *`Git Author's P4USER`* `not authorized to submit file(s) in git commit`
Yes	Yes	No	No	-	No	Git Fusion prohibits the push and displays the following error message: `remote: import failed: user unknown_git not authorized to submit file(s) in git commit`
Yes	No	-	No	-	-	Git Fusion prohibits the push and displays the following error message: `remote: import failed: user` *`Git Author's email`* `not permitted to commit`
No	-	-	-	-	-	Git Fusion prohibits the push and displays the following error message: `remote: import failed: user` *`Pusher's P4USER`* `not authorized to submit file(s) in git commit`

What do I have to do?

To enable Git authors to use Git Fusion, you must:

- Map your Git users to Perforce user accounts. See Mapping Git users to Perforce accounts
- Set up authentication. See Authenticating Git users

- Authorize your Git users using Perforce permissions. See Authorizing Git users

Mapping Git users to Perforce accounts

In a standard Git Fusion implementation, each Git author who pushes or pulls Git Fusion repos must map to a Perforce account. By default, this mapping is made by comparing the email address of the Git author to the email address associated with the Perforce account, or by looking for the email address in a Git Fusion User Map file.

Note

Configure the Git author's email address to be matched against the Perforce user account's email address by issuing a git config command on the git client system.

To configure the email address for a single repository issue the following command from the git repository workspace directory.

```
git config  --local --replace-all user.email jo@example.com
```

To configure the email address for all repositories issue the following command.

```
git config --global --replace-all user.email jo@example.com
```

There are a number of options that enable alternatives to this default:

- The `author-source` property in the global and repo-specific repo configuration files enables you to derive the Perforce account from the `user.name` field in the Git commit or the account portion (the part that precedes @) of the Git author's email address.

  For more information, see Table 5.1, “Global configuration file: keys and default values” on page 38.

- Git authors who perform commits, but not pushes, do not necessarily need to map to Perforce accounts.

  You can extend the ability to author Git commits to Git users who do not have a Perforce account by enabling the `unknown_git` Perforce user. For more information about how `unknown_git` affects Git Fusion pushes, see Effect of permissions on push requests.

- If the Git author is not the same as the user who performs the Git push, you can set Git Fusion to ignore the Git author's Perforce permissions entirely, relying instead on the Perforce permissions of the Git user who performs the push.

  Set the `change-owner` property in the global or repo-specific repo configuration file to `pusher`. For more information, see Table 5.1, “Global configuration file: keys and default values” on page 38.

Verify email address match

Whether you are mapping Git users to existing Perforce accounts or adding new Perforce accounts, the simplest way to map the users is to ensure that the email address associated with the Git user is identical to the email address for their Perforce account.

Use the Git Fusion User Map

In most implementations, establishing the association between your Git users and their Perforce accounts will involve no more than verifying that there is a one-to-one correspondence between the Git account email address and the Perforce account email address. In some cases, however, you may want to map multiple Git email accounts to a single Perforce user or use generic email accounts to mask Perforce user names.

For those scenarios, use the Git Fusion User Map (`p4gf_usermap`), a text file of tuples that enables you to do the following:

- Map multiple Git email accounts to a single Perforce user.

  ```
  p4bob bill@sandimas.net "Bill Preston"
  p4bob bpreston@corporate.com "Bill S. Preston, Esquire"
  ```

- Mask Perforce user names to generic names.

  To mask a company's employee list, run **`p4 users`** and edit the results to map each Perforce user name to a generic email account and name. Add unique identifiers to the email address and name to ensure that each commit maps to the correct user. Otherwise, commits are attributed only to the first user in the list.

  ```
  p4geddy user1@company.com  "Company employee 1"
  p4alex  user2@company.com  "Company employee 2"
  p4neil  user3@company.com  "Company employee 3"
  ```

The map file is automatically created by the User and Client Initialization script (`p4gf_init.py`). The script creates the file in Perforce at `//.git-fusion/users/p4gf_usermap`. You can run this script any time after Git Fusion has been initialized into your Perforce service with the Super User Initialization script (`p4gf_super_init.py`). If `p4gf_init.py` has not been run, `p4gf_init_repo.py` will invoke it automatically. If neither has been run, the first push or clone against the Git Fusion server invokes them both automatically.

Enable the unknown_git Perforce account

If you enable the Perforce user `unknown_git`, commits by Git authors who do not have a Perforce user account can be pushed to Git Fusion repos. The changelist for the Perforce submit will record the submitter as *unknown_git*. For more information about how Git Fusion handles Git authors without Perforce user accounts, see Effect of permissions on push requests. Note that, regardless of whether or not `unknown_git` exists, Git users who perform *pushes* must have a Perforce account.

To allow commits from Git users without a Perforce account:

1. Run **`p4 user`** to create a Perforce account for the user `unknown_git`.
2. Grant permissions to `unknown_git` using Git Fusion's permission groups and Perforce's **`p4 protect`** table.

Authenticating Git users

This guide assumes that you want to use HTTP to authenticate Git users. If you prefer SSH authentication, see Authenticating Git Users using SSH on page 109.

In HTTP authentication, a web server manages authentication for all git client requests. Instead of directly running `git-http-backend`, the standard Common Gateway Interface (CGI) program that implements server-side Git processes over HTTP, Git Fusion HTTP implementations run a Git Fusion script, `p4gf_http_server.py`, that is invoked by CGI. The script does the following:

- Reads the Git Fusion environment to get the `P4PORT` and other options specified in the Git Fusion environment configuration file.
- Reads the CGI environment variables to get the user, repo, and request.
- Checks that the user is authenticated by the web server.
- Checks that the user has Git Fusion authorization for the operation.
- If the operation is a push, assigns the associated Perforce user as the pusher.
- Proceeds with the translation of Git and Perforce commands.
- Invokes `git-http-backend` to manage the rest of the request.

`configure-git-fusion.sh` will optionally configure HTTPS authentication for you. Please see *Git Fusion release notes* for requirements.

Note	HTTPS authentication on Centos 7.x or RedHat 7.x is not supported.

Note

`configure-git-fusion.sh` will configure HTTPS authentication with self-signed SSL certificates.

- If verifiable certificates are available, put them
  - on Ubuntu 12.04 or 14.04 in: `/etc/apache2/ssl/apache.crt` and `/etc/apache2/ssl/apache.key`
  - on CentOS 6.x or RedHat 6.x in: `/etc/httpd/ssl/apache.crt` and `/etc/httpd/ssl/apache.key`
- If you decide to use self-signed certificates, Git users should *not* attempt to verify certificates. Instruct your Git users to do one of the following:
  - Tell Git never to verify SSL certificates:

    ```
    $ git config --global http.sslVerify false
    ```

  - Tell Git not to verify SSL certification in the current shell session only:

```
$ export GIT_SSL_NO_VERIFY=true
```

Note

If you have attempted to configure HTTPS authentication for Git Fusion before, or if you have a working HTTPS authentication, `configure-git-fusion.sh` will not fix or re-configure your existing setup.

Use existing HTTPS configuration with a different Perforce Service.

If you have an existing working HTTPS configuration, but would like to use it with a different Perforce Service, please follow these steps:

- On Ubuntu 12.04 or Ubuntu 14.04:

  a. Stop the Apache web service.

  ```
  # service apache2 stop
  ```

  b. Open the `git-fusion-ssl` or `git-fusion-ssl.conf` Apache site configuration file.

  ```
  # vi /etc/apache2/sites-available/git-fusion-ssl
  ```

  Or

  ```
  # vi /etc/apache2/sites-available/git-fusion-ssl.conf
  ```

  c. Edit the `AddExternalAuth` line to include the full hostname, port and P4CHARSET of your Perforce service.

  ```
  AddExternalAuth p4_auth "/opt/perforce/git-fusion/libexec/
  p4auth.sh myperforceserver:port charset"
  ```

  d. Save your changes and exit `vi`.

  ```
  :wq
  ```

  e. Start the Apache web service.

  ```
  # service apache2 start
  ```

- On CentOS 6.x or RedHat 6.x:

a. Stop the Apache web service.

```
# service httpd stop
```

b. Open the `git-fusion-ssl.conf` Apache site configuration file.

```
# vi /etc/httpd/conf.d/git-fusion-ssl.conf
```

c. Edit the `AddExternalAuth` line to include the full hostname, port, and P4CHARSET of your Perforce service.

```
AddExternalAuth p4_auth "/opt/perforce/git-fusion/libexec/
p4auth.sh myperforceserver:port charset"
```

d. Save your changes and exit `vi`.

```
:wq
```

e. Start the Apache web service.

```
# service httpd start
```

- Run `configure-git-fusion.sh` to re-configure Git Fusion to run against a different Perforce Service, and select `no` when prompted to set up HTTPS configuration.

Validating your HTTP authentication setup

Note	Before verifying HTTPS authentication make sure Git Fusion submit triggers are updated on your Perforce Server.

There are multiple ways to validate that your HTTP setup succeeded:

- From the command line, run:

```
curl -k --user perforce_user:perforce_user_password https://mygitfusionserver/@info
```

- From a browser: go to `https://mygitfusionserver/@info` and log in as a Perforce user.

  The page displays your server information:

```
Perforce - The Fast Software Configuration Management System.
Copyright 2014 Perforce Software.  All rights reserved.
Rev. Git Fusion/2014.2/896875 (2014/07/23).
SHA1: 19786d97b2de1ace6e3694a6937aecf076455e89
Git: git version 1.8.2.3
Python: 3.3.2
P4Python: Rev. P4PYTHON/LINUX35X86_64/2014.1/895961 (2014.1/821990 API) (2014/07/21).
Server address: ssl:1666
```

- From a machine other than the Git Fusion server, clone a repo using HTTP authentication.

```
$ export GIT_SSL_NO_VERIFY=true
$ git clone https://mygitfusionserver/repo_name
```

  The system prompts you to log in as a Perforce user.

Logs

The following logs can be helpful when you need to troubleshoot your HTTP configuration:

Ubuntu

- `/var/log/apache2/error.log`
- `/var/log/apache2/gf-error.log`
- `/var/log/syslog` (default Git Fusion log)

CentOS and Red Hat

- `/var/log/httpd/error_log`
- `/var/log/httpd/gf-error.log`
- `/var/log/messages` (default Git Fusion log)
- `/var/log/audit/audit.log` (for SELinux denials)

Authorizing Git users

To authorize Git users to perform transactions with Git Fusion, you use the `p4 protect` table, Git Fusion repo-specific and global permission groups, and the default group p4 key.

For more information, see How do user permissions work?

To set up authorization:

- Assign Perforce permissions to Git Fusion users

- Create the permission groups and group p4 key
- Populate the permission groups and set the group default p4 key
- (Optional) Enable pushes when Git authors lack Perforce permissions
- (Optional) Enforce Perforce read permissions on Git pull

Assign Perforce permissions to Git Fusion users

Run **`p4 protect`** to verify or add `write` permissions for all Perforce users associated with the Git users who will push changes to the Git Fusion repos.

To successfully perform a push, the Git pusher's Perforce user must have `write` permissions to the affected files. The Git author must also have `write` permissions, unless you use the `unknown_git` user, the `ignore_author_permissions` property, or the `change-owner` property to circumvent that requirement (for more information, see Enable pushes when Git authors lack Perforce permissions).

Note	As of the first 2014.1 patch, you can also configure a branch to be read-only, regardless of a user's Perforce permissions. See Repo configuration file: key definitions and samples.

Git Fusion does not check the `p4 protect` table for `pull` transactions, unless you enable the global `p4gf_config` property to require a read-access check for all pull transactions (see Enforce Perforce read permissions on Git pull). If you do not enable this option, you do not need to assign permissions in the `p4 protect` table for users who are only performing pulls.

Create the permission groups and group p4 key

1. **Run the User and Client Initialization script (`p4gf_init.py`).**

   The global groups and the default permission p4 key are automatically generated by the User and Client Initialization script (`p4gf_init.py`). By default, the group owner is set as `git-fusion-user`. Do not change the owner.

   You can run this script any time after Git Fusion has been initialized in your Perforce service with the Super User Initialization script (`p4gf_super_init.py`). If `p4gf_init.py` has not been run, `p4gf_init_repo.py` will invoke it automatically. If neither has been run, the first push or clone against this Git Fusion server will invoke them both automatically.

Important	The default setting for the `git-fusion-permission-group-default` p4 key is `push`. Change this setting to `none` or `pull`, using **`p4 key`**, if you want to prevent authenticated users who are not members of a permission group from having `push` access to all Git Fusion repos by default. Note that `0` (zero) has the same effect as setting it to `push`. If you set the p4 key to `none`, you must run `p4gf_init_repo.py`.

2. **Run the Repo Initialization script (`p4gf_init_repo.py`) for each repo.**

   `p4gf_init_repo.py` ***`repo_name`***

This script creates the Git Fusion push and pull permission groups for each repo you run it for. By default, the group owner is set as `git-fusion-user.` Do not change the owner.

You can run this script any time after Git Fusion has been initialized in your Perforce service with the Super User Initialization script (`p4gf_super_init.py`). If `p4gf_init.py` has not been run, `p4gf_init_repo.py` will invoke it automatically. If neither has been run, the first push or clone against this Git Fusion server will invoke them both automatically.

For more information about the `p4gf_init_repo.py` script and options, see Setting up Repos.

Populate the permission groups and set the group default p4 key

The way you use the Perforce permission groups and the group default p4 key depends on your needs.

Important

By default, pull requests only check the `p4 protects` table to confirm that the `git-fusion-user` has access to the Perforce depot location; the Git puller's read access to the Perforce location is not checked unless you have enabled the global `p4gf_config` property to require a read-access check for all pull transactions (see Enforce Perforce read permissions on Git pull). Therefore, if you have not enabled this option, you must do **one** of the following to prevent authenticated Git Fusion users from pulling from a particular Perforce depot location, :

- Add all Git Fusion users to repo-specific pull and push permission groups and set the `git-fusion-permission-group-default` p4 key to `none`.
- Use `p4 protects` to deny the `git-fusion-user` (and therefore all Git Fusion users) access to that depot location.

The following are some options:

- **Restrict access strictly by repo.**
  a. Enable users to push by adding them to the `git-fusion-`*`repo_name`*`-push` group for each repo they need to push to. Membership in this group also grants pull permission. Ensure that these group members also have `write` access to the Perforce depot locations associated with the repo being pushed.
  b. Enable users to pull by adding them to each `git-fusion-`*`repo_name`*`-pull` group they need to pull from.
  c. To prevent the global pull and push groups (`git-fusion-pull` and `git-fusion-push`) from granting access to users who are not in a repo-specific group, keep these groups empty.
  d. To prevent the `git-fusion-permission-group-default` p4 key from giving access to users who are not in a repo-specific group, set it to `none`.
- **Provide pull access to all repos, restricting push access.**
  a. Add users to the `git-fusion-`*`repo_name`*`-push` group for each repo they need to push to. Ensure that these group members also have `write` access to the Perforce depot locations associated with the repo being pushed.

b. Add all users to the global `git-fusion-pull` group or set the `git-fusion-permission-group-default` p4 key to `pull`.

- **Open push access to all Git Fusion repos for all authenticated users.**

  Add all users to the global `git-fusion-push` group or set the `git-fusion-permission-group-default` p4 key to `push`. If you want to enable all members to pushes to all repos, ensure that these group members also have `write` access to the Perforce depot locations associated with all Git Fusion repos.

Git Fusion creates global groups `git-fusion-pull` and `git-fusion-push` as part of its configuration script, `configure-git-fusion.sh`. It creates repo groups `git-fusion-`*`repo_name`*`-pull` and `git-fusion-`*`repo_name`*`-push` during the first push or pull for that repo.

For more information about setting group permissions and p4 keys in Perforce, see the *Perforce System Administrator's Guide*, *Administering Perforce: Protections*.

Enable pushes when Git authors lack Perforce permissions

The Git pusher is not always the same Git user as the author and committer of the changes being pushed. While the pusher must always be a licensed Perforce user with write permission for the depot locations being pushed to, you may not need all of your Git authors to be mapped to a licensed Perforce user. Git Fusion provides the following tools to enable pushes when the Git author is not a Perforce user:

- **`unknown_git` user**

  Create this Perforce user and give it Perforce `write` permission for the depot locations associated with all repos for which you want to allow pushes when the Git author has no Perforce account. If your `git-fusion-permission-group-default` p4 key is set to `pull` or `none`, add `unknown_git` to the global `git-fusion-push` group or the relevant repo-specific push groups.

  When a Git push request is made, Git Fusion checks to see if the Git author has a mapped Perforce account. If not, and `unknown_git` has `write` permissions, the push goes through. If the author exists, the author is still recorded as the submitter in the Perforce changelist description. If the author does not exist, the submitter is recorded as *unknown_git*.

- **`ignore-author-permissions` property**

  Set this configuration property to `Yes` in a repo-specific configuration file to enable pushes to go through even when the Git author does not have `write` (push) permissions for the depot locations associated with the repo.

- **`change-owner` property**

  Set this configuration property to `pusher` to make the changelist owner (submitter) the Git pusher rather than the Git author (which is the default). Regardless of which user is set as the changelist submitter, the full information from the Git commit is logged in the changelist description field, including information about the Git committer, Git author, and Git pusher. You can set this configuration property in the global configuration file or a repo-specific configuration file.

For more information about repo configuration files, see Setting up Repos

Enforce Perforce read permissions on Git pull

By default, Git Fusion checks Perforce permissions only for Git push transactions, relying on user authentication to the Git Fusion server and membership in `git-fusion-pull` permission groups to control Git pull (read) access to Git Fusion repos. However, if you want to enforce the permissions that you have set up in the Perforce `protects` table on all Git pull transactions as well, you can do so by setting the `read-permission-check` property in the global `p4gf_config` file. See Global configuration file: keys and default values

Chapter 5 Setting up Repos

After you install Git Fusion, you must configure your repos.

This chapter discusses the following topics:

- "How does Git Fusion map Perforce depots to Git repos?" on page 37
- "Configuring global defaults for repos" on page 38
- "Configuring repos" on page 54
- "Initializing repos on the Git Fusion server" on page 62
- "Importing existing Git repos into Git Fusion" on page 62
- "Modifying repo configuration files safely" on page 65
- "Converting a lightweight branch into a fully-populated branch" on page 65
- "Enabling Git users to create fully-populated branches" on page 66
- "Working with Perforce streams" on page 71
- "Enabling stream import paths as Git submodules" on page 72
- "Adding preflight commits to reject pushes" on page 74
- "Adding preflight hooks to reject pushes" on page 76
- "Limiting push size and disk usage" on page 78
- "Detecting Git copy/rename and translating to Perforce" on page 79
- "Disconnecting a Git Fusion repo from the Perforce service" on page 81
- "Deleting Git Fusion repos" on page 81

How does Git Fusion map Perforce depots to Git repos?

To populate a repo hosted by Git Fusion, a Perforce administrator creates a configuration file (`p4gf_config`) that identifies the scope of Perforce depot data contained in the repo, along with character encoding and branching directions. The map of the Perforce depot location to the Git repo uses the same syntax as standard workspace (client) views, and is referred to in this guide as the *view* or *repo view*. In the following repo view, the path to the left represents a Perforce depot location, and the path to the right represents the Git repo work tree:

```
//depot/main/your_project/foo/... foo/...
```

In this case, the contents of Perforce depot directory `//depot/main/your_project/foo/`maps to the `foo/` directory in the Git repo.

You can also represent the top level of the Git repo work tree with an ellipsis:

```
//depot/main/your_project/foo/... ...
```

Repo configuration files enable you to define multiple branch mappings. Git users can push commits that have linear history or merged history, including two-parent merges and octopus (3+ parent-) merges.

Git Fusion uses two types of repo configuration files:

- The global configuration file, which is generated automatically and stored in the top level of the `//.git-fusion` depot in Perforce.

  You edit this file to provide global defaults, specifically character set preferences, branch enablement, preflight commit scripts to enforce local policy, and author permissions requirements.

- Repo-specific configuration files, which the administrator creates from templates provided in the OVA or distribution package, and which are stored in repo-specific directories in the `//.git-fusion` depot in Perforce.

  Any preferences that you do not specify in a repo-specific configuration file default to those in the global configuration file.

Note

> You can choose not to create a repo configuration file, and instead map your repo to Peforce depot locations by creating a Perforce workspace specification and letting Git Fusion create the configuration file.

For more information, see Configuring repos.

Configuring global defaults for repos

The User and Client Initialization script (`p4gf_init.py`) creates the global configuration file and stores it in Perforce at `//.git-fusion/p4gf_config`. You can edit any of its key values that you want repo configuration files to inherit by default.

Note

> You can run this script any time after Git Fusion has been initialized in your Perforce service with the Super User Initialization script (`p4gf_super_init.py`). If `p4gf_init.py` has not been run, `p4gf_init_repo.py` will invoke it automatically. If neither has been run, the first push or clone against this Git Fusion server will invoke them both automatically.

View the file header comments or see the table below for details.

Table 5.1. Global configuration file: keys and default values

Section Headers or Keys	Definition	Default Value	Valid Values
`[repo-creation]`	Section header for settings that control how Git Fusion creates new repos.	*NA*	Enter the section header exactly as shown.
`charset`	Defines the default Unicode setting that Git Fusion applies to new repos. This setting is	`charset: UTF-8`	Any `P4CHARSET` value; run **`p4 help charset`** for a list of valid values.

Section Headers or Keys	Definition	Default Value	Valid Values
	valid only when Git Fusion interacts with a Unicode-enabled Perforce server.		

`depot-path-repo-creation-enable`	Allow Git users to create new repos by pushing/pulling a git url which specifies a Perforce depot path. This is similar to creating a repo from a p4 client.	`no`	`Yes` equivalent (`Yes, On, 1, True`) or `No` equivalent (`No, Off, 0, False`).
			
			`No`: Automatic repo creation from a depot path is disallowed.
			
			`Yes`: Automatic repo creation from a depot path is allowed. Under the following conditions a new repo will be created:
			
			• The repo name is formated as: ***`depotname/reponame/branchname`***
			• ***`depotname`*** is a defined Perforce depot of `type='local'`
			• No p4gf_config nor p4 client exists with the translated name: `depotname_0xS_reponame_0xS_branchname`
			
			If the conditions are not met, the push/pull will fail with the expected error message reporting the repo is not defined.
			
			The newly created repo p4gf_config will contain:
			
			`[@repo] description = Created from 'depotname_0xS_reponame_0xS_branchname'`
			
			`[Hzb5rdffTRGEsjotvTLoHg==]`
			`git-branch-name = master`
			`view = //depotname/reponame/branchname/... ...`
			
			For a clone/pull situation, any files under ***`//depotname/repo/branch`*** will be imported into a new Git repo's master branch.
			
			For a push situation, any files in the pushed Git branch will be imported into a new Perforce depot path.

Section Headers or Keys	Definition	Default Value	Valid Values
`depot-path-repo-creation-p4group`	Restrict which authenticated Git pushers are allowed to create new repos when depot-path-repo-creation-enable is enabled.	`Unset/None`	`Unset/None`: No restriction: all Git pushers can create new repos from depot paths if `depot-path-repo-creation-enable` is enabled. Set this to the name of an existing Perforce **`p4 group`** to restrict this feature to members of that group. You can also use **`p4 protect`** to grant/deny write permission to areas of the Perforce depot.
`[git-to-perforce]`	Section header for settings that define how Git commits are converted to Perforce changes (submits).	*NA*	Enter the section header exactly as shown.
`change-owner`	Defines whether Git Fusion assigns either the Git commit author or the Git pusher as the owner of a pushed change (submit).	`author`	Either `author` or `pusher`.
`enable-git-branch-creation`	Defines whether Git Fusion creates a new branch of Perforce depot file hierarchy for each copied branch of Git workspace history, including Git task branches as Git Fusion *anonymous branches*. See Git branch and merge: effect of configuration key values for more information about setting this key.	`Yes`	`Yes` equivalent (`Yes, On, 1, True`) or `No` equivalent (`No, Off, 0, False`). When set to `No`, Git Fusion prohibits the copy of any new Git branches to Perforce that are not defined in the repo's configuration file, and also translates Perforce file hierarchy merges to Git as file edits, not as Git merge commits. However, Git Fusion will still copy Git merge commits between Perforce branches that are defined in the repo's configuration file. To permit Git Fusion to create new branches for Swarm reviews, you must also enable `enable-swarm-reviews`.
`enable-swarm-reviews`	Permits branch creation for Swarm reviews, even when `enable-git-branch-creation` is disabled. See Using Swarm for code review for more information about Swarm reviews.	`Yes`	`Yes` equivalent (`Yes, On, 1, True`) or `No` equivalent (`No, Off, 0, False`). `Yes` enables Git Fusion to create a new branch of Perforce depot file hierarchy for each new Swarm review and permits merge commits in the review history, which become *anonymous branches* in Perforce.

Section Headers or Keys	Definition	Default Value	Valid Values
			This setting overrides `enable-git-branch-creation` and `enable-git-merge-commits` for Swarm reviews. `No` disables the creation of branches for Swarm reviews, effectively disabling the ability to push Swarm reviews from Git.
`enable-git-merge-commits`	Defines whether Git Fusion copies merge commits and displays them in Perforce as integrations between Perforce branches. See Git branch and merge: effect of configuration key values for more information about setting this key.	`Yes`	`Yes` equivalent (`Yes, On, 1, True`) or `No` equivalent (`No, Off, 0, False`). `No` means that Git Fusion rejects all merge commits; integrations and merges between Perforce branches must be performed using Perforce.
`enable-git-submodules`	Defines whether Git Fusion allows Git submodules to be pushed to Perforce.	`Yes`	`Yes` equivalent (`Yes, On, 1, True`) or `No` equivalent (`No, Off, 0, False`). `No` prevents Git submodules from being introduced into Git Fusion. If any submodules have already been pushed to Git Fusion, they will be left intact and be reproduced through clone/pull.
`ignore-author-permissions`	Defines whether Git Fusion evaluates both the author's and pusher's Perforce `write` permissions during a push or evaluates only the pusher's permissions.	`No`	`Yes` equivalent (`Yes, On, 1, True`) or `No` equivalent (`No, Off, 0, False`). When set to `yes`, Git Fusion evaluates only the pusher's permissions.
`preflight-commit`	Enables you to trigger preflight commit scripts that enforce local policy for Git pushes. This can be especially useful if you have Perforce submit triggers that could reject a push and damage the repository. For more information about setting this key, see Adding preflight commits to reject pushes.	`none`	`Pass` passes all pushes that Git Fusion would otherwise permit, and `Fail` rejects all pushes; these values are primarily intended for temporarily disabling a preflight commit. You can add a path to a message as an argument to either of these values. To enable a preflight commit script, use the syntax ***`command argument`***, where command is the path to the script. Arguments can include Git Fusion and Perforce trigger variables, as in the following example:

Section Headers or Keys	Definition	Default Value	Valid Values
			`preflight-commit = /home/git/myscript.sh %repo% %sha1%`
`preflight-push`	Enables you to trigger preflight push scripts that enforce local policy for Git pushes. This can be especially useful if you have Perforce submit triggers that could reject a push and damage the repository. For more information about setting this key, see In addition to preflight hooks that operate on a per commit basis, as described in the previous section, it is also possible to run a command to evaluate each pushed reference. This can be useful for rejecting the deletion of a branch, for example, or rejecting any commits to a branch outside of a certain time period. To enable a preflight push hook: 1. **Create the script and save it to the server that hosts Git Fusion.** Guidelines include the following: • Exit code `0` = pass (the push goes through), `1` = fail (reject the push) • The script must be run by the same UNIX account that runs Git Fusion (the Git Fusion service account), under the same environment.	`none`	`Pass` passes all pushes that Git Fusion would otherwise permit, and `Fail` rejects all pushes; these values are primarily intended for temporarily disabling a preflight check. You can add a message as an argument to either of these values. To enable a preflight push script, use the syntax ***`command argument`***, where command is the path to the script. Arguments can include Git Fusion and Perforce trigger variables, as in the following example: `preflight-push = /home/git/myscript.sh %repo% %sha1%`

Section Headers or Keys	Definition	Default Value	Valid Values

	• The script is not invoked with a full shell, but it has access to the following environment variables: `CWD`
	
	`P4PORT`
	
	`P4USER`
	
	`P4CLIENT`
	
	• The script can consume the following Git Fusion variables: `repo`
	
	`user`		

Section Headers or Keys	Definition	Default Value	Valid Values

	`ref`
	
	`git-action`
	
	• The script can consume the following standard Perforce trigger variables: `client`		

Section Headers or Keys	Definition	Default Value	Valid Values
	clienthost		
	command		
	serverport		
	quote		
	user		

Section Headers or Keys	Definition	Default Value	Valid Values

	`formname`
	
	`formtype`
	
	For more information about Perforce trigger scripts and variables, see the *Perforce System Administrator Guide*, "Scripting Perforce: Triggers and Daemons".
	
	2. Add the script to the global configuration file or a repo-specific file, using the `preflight-push` key.
	Use the syntax ***`command argument`***, where		

Section Headers or Keys	Definition	Default Value	Valid Values
	command is the path to the script. Arguments can include any of the variables listed above, using the convention *%variable%*, as in the following example: `[@repo] preflight-push = /home/git/myscript.sh %repo% %user%` Multiple scripts may be run in the configured order. All scripts must pass or the commit is rejected. `[@repo] preflight-push = /home/git/myscript1.sh %repo% %user% /home/git/myscript2.sh %repo% %user% %ref%` For more information about global and repo-specific configuration files, see Configuring global defaults for repos and Configure repos with a repo configuration file (`p4gf_config`). .		
`read-permission-check`	Enables you to require that Git clone, pull, or fetch requests check the Perforce protections table for the puller's read permission on the files being pulled.	`group`	`Group` bypasses a Perforce permissions check on pull transactions, relying on membership in a Git Fusion pull permission group for access to the files. `User` enables a check that the puller's Perforce user has Perforce read permission for all files within the repo. For more

Section Headers or Keys	Definition	Default Value	Valid Values
			information, see Enforce Perforce read permissions on Git pull
`git-merge-avoidance-after-change-num`	If the Perforce service includes any changelists submitted by Git Fusion 13.2 or earlier, you can prevent unnecessary merge commits by setting this key to the number of the last changelist submitted before your site upgraded to a later version of Git Fusion.	`p4 counter change`	Keep the default value, `p4 counter change`, if you have no commits from earlier instances of Git Fusion (13.2 or earlier). At the first initialization of a Git Fusion repo, Git Fusion writes the changelist number to the global configuration file. If you do have commits from Git Fusion 13.2 or earlier, provide the number of the the last changelist submitted before your site upgraded to a later version of Git Fusion.
`job-lookup`	Set the format for entering Perforce jobs in Git commit descriptions so that they are recognized by Git Fusion and appear in Perforce changelists as fixes. By default, job IDs whose string starts with "`job`" (as in `job123456`) are passed through to the changelist description and job field. Use this option if you want Git Fusion to recognize *additional* expressions, such as JIRA issue IDs. For more information about including jobs in Git commit descriptions, see "Referencing Perforce jobs in a commit" on page 94.	`none`	Enter an expression that Git Fusion will pass to **`p4 jobs -e`** to look for matching jobs. You can add multiple fields, one per line. For example, let's say your job specification includes the field DTG_DTISSUE for JIRA issue IDs. If you set `job-lookup: DTG_DTISSUE={jobval}`, then Git Fusion runs **`p4 jobs -e DTG_DTISSUE=XY-1234`** when it sees a Git commit message that includes `Jobs: XY-1234`. You do not need to add a value for standard Job IDs, stored in the job spec's `Job` field, whose string starts with "`job`" (as in `job123456`). These are passed through by default. For more information about the **`p4 jobs`** command and the expressions that you can pass using `-e`, see the *Perforce Command Reference*.
`depot-branch-creation-enable`	Allow Git users to create new fully-populated depot branches within Perforce.	`no`	`no`: Any new branches pushed by Git users go into `//.git-fusion/branches/` as lightweight depot branches.

Section Headers or Keys	Definition	Default Value	Valid Values
	For more information, see "Enabling Git users to create fully-populated branches" on page 66		`explicit`: Push to special remote branch reference `depot-branch/branch_name`. This creates a new fully-populated depot branch in Perforce. For example, **`git push origin mybranch:depot-branch/research`** creates a new Perforce depot branch under `//depot/myrepo/research/`. `all`: Each new Git branch pushed by Git users goes into a new fully-populated depot branch in Perforce. For example, **`git push origin mybranch:research`** creates a new Perforce depot branch under `//depot/myrepo/research/`.
`depot-branch-creation-p4group`	Restrict the authenticated Git pushers who are allowed to create new fully-populated depot branches, if `depot-branch-creation-enable` is enabled. For more information, see "Enabling Git users to create fully-populated branches" on page 66	`None`	Set to the name of an existing Perforce p4 group to restrict this feature to members of that group. `Unset/None`: No restriction. All Git pushers can create new fully-populated depot branches if `depot-branch-creation-enable` is enabled. You can unset this property and use the **`p4 protect`** command to fine-tune Perforce user and group access to specific areas of the Perforce depot.
`depot-branch-creation-depot-path`	Tell Git Fusion where to create new fully-populated depot branches, if `depot-branch-creation-enable` is enabled. Default path is `//depot/{repo}/{git_branch_name}` For more information, see "Enabling Git users to create fully-populated branches" on page 66	*(at left)*	Use the following string substitutions to set the location of new branches: `{repo}`: returns the name of the Git Fusion repo receiving this push. `{git_branch_name}` : returns the name of the pushed branch reference, such as, for example, `myfeature` in the command **`git push master:depot-branch/myfeature`**. Perforce path rules apply: `@`, `#`, `%`, `*` , `//`, `...` are prohibited; `/` is permitted. This substitution must be included somewhere in the string, or it becomes impossible for Git users to create more than one branch to a single repo. `{user}`: returns the Perforce user ID of the pusher.

Section Headers or Keys	Definition	Default Value	Valid Values
`depot-branch-creation-view`	Set how the depot-path set in `depot-branch-creation-depot-path`should appear in Git. For more information, see "Enabling Git users to create fully-populated branches" on page 66	`... ...`	Enter a Perforce view specification that maps Perforce depot paths (left side) to Git work tree paths (right side). Perforce depot paths are relative to the root set in `depot-branch-creation-depot-path`. The default "`... ...`" maps every file under the `depot-branch-creation-depot-path` root to Git. Right side paths must match the right side for every other branch already defined within a repo.
`enable-git-find-copies`	When Git reports a **`copy`** file action, store that action in Perforce as a **`p4 integ`**. Often set in tandem with `enable-git-find-renames`. For more information, see "Detecting Git copy/rename and translating to Perforce" on page 79	`No`	`No/Off/0%`: Do not use Git's copy detection. Treat all possible file copy actions as **`p4 add`** actions. `1%-100%`: Use Git's copy detection. Value passed to `git diff-tree --find-copies=n`. Git Fusion also adds `--find-copies-harder` whenever adding `--find-copies`.
`enable-git-find-renames`	When Git reports a **`rename`** (also called **`move`**) file action, store that in Perforce as a **`p4 move`**. Often set in tandem with `enable-git-find-copies`. For more information, see "Detecting Git copy/rename and translating to Perforce" on page 79	`No`	`No/Off/0%`: Do not use Git's rename detection. Treat all possible file rename actions as independent **`p4 delete`** and **`p4 add`** actions. `1%-100%`: Use Git's rename detection. Value passed to `git diff-tree --find-renames=n`.
`[perforce-to-git]`	Section header for settings that define how Perforce changes (submits) are converted to Git commits.	*NA*	Enter the section header exactly as shown.
`enable-stream-imports`	Enables you to convert Perforce stream import paths to Git submodules when you clone a Git Fusion repository. If set to `Yes`, you must also set either `http-url` or `ssh-url`.	`No`	Set to `Yes` equivalent (`Yes, On, 1, True`) to enable Git Fusion to convert compatible stream import paths to Git submodules. Set to `No` equivalent (`No, Off, 0, False`) to have import paths and their history incorporated in the Git repo for the stream.

Section Headers or Keys	Definition	Default Value	Valid Values
	For more information, see "Enabling stream import paths as Git submodules" on page 72.		
`http-url`	The URL used by Git to clone a repository from Git Fusion over HTTP. This property is required if you want to use Perforce stream import paths as git submodules and you use HTTP(S).	`none`	You can enter the full host name that you use to clone a repo from Git Fusion, or you can include variable placeholders that will be replaced by values from the Git Fusion environment: `{host}`: returns the fully qualified hostname of the Git Fusion host computer, as fetched by the Linux function **`gethostname()`**. If this does not resolve to a value that is recognized by the client (a hostname that can be used to perform Git commands against the Git Fusion repos), then use the actual, full hostname rather than the variable. `{repo}`: returns the name of the Git Fusion repository. Required, must be at the end of the string. Example with only variable placeholders: `http://{host}/{repo}` Example with hostname provided: `http://p4gf.company.com/{repo}` For HTTPS installations, use the `https://` prefix.
`ssh-url`	The "URL" used by Git to clone a repository from Git Fusion using SSH. This property is required if you want to use Perforce stream import paths as git submodules and you use SSH.	`none`	You can use the following variable placeholders that will be replaced by values from the Git Fusion environment: `{user}`: returns the SSH user performing the Git clone. If a user name is not found, this value defaults to `git`. `{host}`: returns the fully qualified hostname of the Git Fusion host computer, as fetched by the Linux function **`gethostname()`**. If this does not resolve to a value that is recognized by the client (a hostname that can be used to perform Git

Section Headers or Keys	Definition	Default Value	Valid Values
			commands against the Git Fusion repos), then use the actual, full hostname rather than the variable. `{repo}`: returns the name of the Git Fusion repository. Required, must be at the end of the string. Example with only variable placeholders: `{user}@{host}:{repo}` Example with hostname provided: `{user}@p4gf.company.com:{repo}`
`[authentica`	Section header for settings that define authentication options.	*NA*	Enter the section header exactly as shown.
`email-case-sensitivity`	Defines whether Git Fusion pays attention to case when matching Git user email addresses to Perforce user account email addresses during the authorization check. For more information about how Git Fusion uses email addresses to authorize users, see Mapping Git users to Perforce accounts.	`no`	`Yes` equivalent (`Yes, On, 1, True`) or `No` equivalent (`No, Off, 0, False`). `Yes` enforces email address case sensitivity.
`author-source`	Defines the source that Git Fusion uses to identify the Perforce user associated with a Git push. For more information about how Git Fusion associates Git authors with Perforce users, see Mapping Git users to Perforce accounts.	`git-email`	Use any one of the following values: • `git-email`: Use the email address of the Git author to look for a Perforce user account with the same email address. Git Fusion consults the `p4gf_usermap` file first, and if that fails to produce a match, it scans the Perforce user table. • `git-user`: Use the `user.name` field in the Git commit. This is the part of the `author` field before the email address. • `git-email-account`: Use the account portion of the Git author's email address. If the Git author's email value is `<samwise@the_shire.com>`, Git Fusion uses the Perforce account `samwise`.

Section Headers or Keys	Definition	Default Value	Valid Values
			You can also tell Git Fusion to iterate through multiple source types until it finds a matching Perforce account. Specify the source types in order of precedence, separated by commas. For example: `git-user, git-email-account, git-email`.
`[quota]`	Section header for settings that define push limit options.	*NA*	Enter the section header exactly as shown.
`limit_space`	Natural number representing the number of megabytes of disk space that can be consumed by any single repo. This value does not include the spaced consumed on the Perforce server.	`0`	If the value is zero or less, the limit is not enforced.
`limit_commi`	Natural number representing the maximum number of commits allowed in a single push.	`0`	If the value is zero or less, the limit is not enforced.
`limit_files`	Natural number representing the maximum number of files allowed in a single push.	`0`	If the value is zero or less, the limit is not enforced.
`limit_megab`	Natural number representing the maximum number of megabytes allowed in a single push.	`0`	If the value is zero or less, the limit is not enforced.

The table below shows how the values you select for the `enable-git-branch-creation` and `enable-git-merge-commits` keys affect Git users' ability to perform branches and merges. Inform your Git users if you implement Scenarios 2, 3, or 4, because these scenarios will restrict their normal use of Git's branch and merge functionality.

Table 5.2. Git branch and merge: effect of configuration key values

Scenario	`enable-git-branch-creation` Value	`enable-git-merge-commits` Value	Result
1	Yes	Yes	This scenario has the least impact on Git users' usual workflow. Any Git user with a corresponding valid Perforce user (either his or her own user or `unknown_git`) can create and push

Scenario	enable-git-branch-creation Value	enable-git-merge-commits Value	Result
			branches and merge commits as they normally do in Git.
2	No	No	This is the most restrictive scenario for Git users. They cannot push any new Git branches that are not expressly defined in a repo's configuration file, and also must ensure that they push commits that have a linear history.
3	Yes	No	This scenario has a moderate impact on Git users. They can push new Git branches to Perforce but they must ensure that all commits have a linear history. If they attempt to push a merge commit, Git Fusion displays the error message: `remote: Merge commits are not enabled for this repo`. Only Perforce users can perform merge and integration work using a Perforce workspace.
4	No	Yes	This scenario has a moderate impact on Git users. They can push merge commits between Perforce branches that are defined in a repo's configuration file, but cannot push new Git branches to Perforce. If they attempt to push a new Git branch, Git Fusion displays the error message: `remote: Git branch creation is prohibited for this repo`.

Configuring repos

To specify the parts of a Perforce depot that are accessible to a Git Fusion repo, you can use any of the following:

- "Configure repos with a repo configuration file (p4gf_config)" on page 55

  Repo configuration files let you define an unlimited number of branches, and repo-specific options including Unicode character sets.

- "Configure repos from a Perforce workspace" on page 58

  This approach can be convenient if you already have workspace definitions that you want to use as Git repos. You use an existing a workspace or create a new one, and Git Fusion generates the repo configuration file using the workspace view. The global configuration file file provides the default options such as branching preferences and charset definitions. This approach does not allow you to define branches when the configuration file is first created, but you can edit the file later to set repo-specific options and add branch definitions (within certain limitations).

- "Use a Perforce depot path in a Git remote URL" on page 60

If enabled, this approach lets some or all Git users define repo configurations by simply supplying a Perforce depot path as part of the remote URL to **git clone**, **pull**, or **push** commands. Git Fusion will generate the repo configuration file using the path supplied. The global configuration file file provides the default options such as branching preferences and charset definitions. This approach does not allow you to define branches when the configuration file is first created, but you can edit the file later to set repo-specific options and add branch definitions (within certain limitations).

Important

Git Fusion does not automatically handle changes to repo definitions throughout the system. Once a repo has been cloned from Git Fusion, only limited modifications should be made to the repo configuration file. For more information about how to modify repos safely, see Modifying repo configuration files safely.

Important

While Git Fusion supports sharing depot paths across repos, in Git, resources shared by different repos, are generally maintained in their own repository. The separate projects can then use the shared repo by including it as a submodule or other means.

Configure repos with a repo configuration file (p4gf_config)

1. **Copy the repo configuration file template, `p4gf_config.repo.txt`, to create a new config file.**

   If you installed Git Fusion using the OVA or operating system specific packages, the template is in the `/opt/perforce/git-fusion/libexec` directory. If you installed using the distribution tarball, the location is the directory where you elected to install Git Fusion.

2. **Enter the key values and Perforce-to-Git mapping (view) for your repo.**

   Ensure that the Git work tree path notation on the **view** field's right side matches for all branches.

   Note

   Views can include overlay and exclusionary mappings, but note that the Git Fusion submit triggers (which enable atomic pushes) ignore exclusionary mappings, because the scope of submit triggers is limited to areas that are potentially of interest to Git Fusion. Exclusionary mappings are ignored for the calculation of areas of interest, because one repo's exclusions could conflict with another's inclusion.

   Note

   If in a given repo configuration, there is Perforce path that is mapped to two or more Git branches, then that path is a "shared" path and thus read-only from the Git perspective.

   For detailed information about the repo configuration keys and **view** syntax, see the repo configuration file's detailed header comments and Repo configuration file: key definitions and samples.

3. **Submit the repo configuration file to Perforce.**

   Save the file as `p4gf_config`.

Submit the file to `//.git-fusion/repos/`*`repo_name`*`/p4gf_config`

The ***`repo_name`*** can include the forward slash (/) and colon (:) characters, but these characters must be encoded as `_0xS_` and `_0xC_`

Note

For example: repo name foo/bar:zee is created by submitting the following p4gf_config

`//.git-fusion/repos/foo_0xS_bar_0xC_zee/p4gf_config`

4. **Initialize (populate) the repo.**

   See Initializing repos on the Git Fusion server

Repo configuration file: key definitions and samples

A repo-specific configuration file can include (and override) any property included in the global configuration file, in addition to the following.

Table 5.3. Repo-specific configuration files: keys and default values

Section Headers or Keys	Definition	Default Value	Valid Values
`[@repo]`	Section header for the repo configuration file. You can override any global configuration property by adding it to this section.	*NA*	Enter the section header exactly as shown.
`description`	Repo description returned by the `@list` command	*NA*	Enter a concise repo description.
`read-only`	Prohibit git pushes that introduce commits on any branch in this repository.	`No`	`Yes` equivalent (`Yes, On, 1, True`) or `No` equivalent (`No, Off, 0, False`). `Yes` makes the repo read-only and prevents users from committing changes.
`[`*`git-fusion-branch-id`*`]`	Section header to define a unique Git Fusion branch.	*NA*	Each branch must have a unique ID in the form of an alphanumeric string. *Do not edit this value after you clone the repo.*
`git-branch-name`	Defines a name specified in a local repo for a Git branch.	*NA*	A valid Git branch name. *Do not edit this value after you clone the repo.*
`view`	Defines a Perforce workspace view mapping that maps	*NA*	Correctly formed mapping syntax; must not include any

Section Headers or Keys	Definition	Default Value	Valid Values
	Perforce depot paths (left side) to Git work tree paths (right side).		Perforce stream or spec depots, and all depot paths on the right side must match exactly across all branch definitions. *You can add and remove only certain types of Perforce branches from this view after you clone the repo*. See Modifying repo configuration files safely
`stream`	Defines a Perforce stream that maps to the Git branch.	*NA*	Provide a stream name using the syntax *`//streamdepot/mystream`*. A Git Fusion branch can be defined as a **`view`** or a **`stream`** *but not both*. If your branch is defined as stream, it can include *only one stream*. For more information, see Working with Perforce streams.
`read-only`	Prohibit git pushes that introduce commits to the branch.	`No`	`Yes` equivalent (`Yes, On, 1, True`) or `No` equivalent (`No, Off, 0, False`). `Yes` makes the branch read-only and prevents users from committing changes.

Sample repo configuration files

Here are two examples of repo configuration files:

Important

The Git work tree path notation on the **`view`** field's right side must match exactly for all branches defined in a repo configuration file to enable merge commits. Otherwise, Git Fusion will fail during a merge between the branches and report the error **`file(s) not in client view`**.

Example 1:

```
[@repo]
description = A repo configuration file that maps two branches, master and release, into the top
 level of the Git repo.

[master]
git-branch-name = master
view = //depot/main/your_project/... ...


[release]
git-branch-name = release
view = //depot/release/your_project/... ...
```

Example 2:

```
[@repo]
description = A repo configuration file that maps portions of two branches, master and release,
 into subdirectories in the Git repo
charset = utf8

[master]
git-branch-name = master
view = //depot/main/your_project/foo1/... foo1/...
       //depot/main/your_project/bar1/... bar1/...

[release]
git-branch-name = release
view = //depot/release/your_project/foo1/... foo1/...
       //depot/release/your_project/bar1/... bar1/...
```

Configure repos from a Perforce workspace

You can use a Perforce workspace (client) to map a single fully-populated Perforce branch to a Git Fusion repo and let Git Fusion generate the repo configuration file for you. The global configuration file file provides the default options such as branching preferences and charset definitions. This approach does not allow you to define branches when the configuration file is first created, but if your global default is to enable branching, you can edit the file later to add branch definitions. For more information, see Modifying repo configuration files safely.

This approach can be convenient if you already have workspace definitions that you want to use as Git repos.

1. **Create a Perforce workspace.**

   The workspace name becomes the repo name.

   Note that the `Client` name can include the forward slash (/) and colon (:) characters. However slash (/) must be encoded as `_0xS_`. The resulting internal Git Fusion repo name will also encode the colon (:) as `_0xC_`. The public git repo name will retain any slash (/) and colon (:) characters.

> **Note**
>
> For example: `Client foo_0xS_bar:zee` will result in the internal repo p4gf_config `//.git-fusion/repos/foo_0xS_bar_0xC_zee/p4gf_config`.
>
> The public git url will use the repo name `foo/bar:zee`.

Use the `View` field to define a single branch mapping. The mappings determine what portions of the Perforce depot are translated into Git repo branches and vice versa.

You can create simple and complex mappings that have the following:

- Exclusionary and overlay mappings.
- Different permissions for each depot path; for example, one depot path that includes files with `read` and `write` permissions, and another depot path that includes files with only `read` permissions.

The example below shows a workspace view with the key fields defined: `Client`, `Owner`, `Root`, and `View`. Note that only the `Client` and `View` fields are meaningful to Git Fusion.

```
Client: project_repo
Owner: p4bob
Root: /home/bob

# View that maps into the top level of the Git repo
View:
   //depot/main/your_project/... //project_repo/...

# View that maps into a sub directory in the Git repo
View:
   //depot/main/your_project/foo1/... //project_repo/foo1/...
```

2. **Save the workspace.**

3. **Initialize (populate) the repo using either of these methods:**

   - **If you issue a Git command like `git clone` using the Perforce workspace name for the repo name, Git Fusion will automatically initialize the new repo, and then pass it off to Git for transfer to the Git client.**

```
$ git clone https://gfserver/Jam
Cloning into 'Jam'...
Perforce: Copying files: 84
Perforce: 100% (23/23) Copying changelists...
Perforce: Submitting new Git commit objects to Perforce: 24
remote: Counting objects: 125, done.
remote: Compressing objects: 100% (69/69), done.
remote: Total 125 (delta 51), reused 89 (delta 33)
Receiving objects: 100% (125/125), 174.80 KiB, done.
Resolving deltas: 100% (51/51), done.
$ cd Jam
```

- **Administrators can also initialize the new repo explicitly. This is often useful for large workspace views that take some time to be turned into Git repos. See Initializing repos on the Git Fusion server.**

Git Fusion uses the workspace view only once, using defaults from the global configuration file, to create a `p4gf_config` file for the repo that it automatically stores in `//.git-fusion/repos/`*`repo_name`*`/p4gf_config`. Because Git Fusion only uses the workspace view once to generate a `p4gf_config` file, you can delete it from the Perforce depot after repo initialization.

For more information about how to define Perforce workspace views, see the *P4 User Guide*, "Configuring P4."

To delete invalid or outdated repo views, see `p4gf_delete_repo.py --help`.

Use a Perforce depot path in a Git remote URL

If enabled, this approach lets some or all Git users define repo configurations by simply supplying a Perforce depot path as part of the remote URL to **`git clone`**, **`pull`**, or **`push`** commands. Git repos can be created from existing Perforce depot paths, and Perforce depot paths can be populated from existing Git repos.

- You can instruct Git Fusion to create a new Git repo simply by running **`git clone`**, or **`git pull`** with a URL that matches an existing Perforce depot path. For example, if your Perforce depot is organized with the `main` branch of the `Jam` project in `//depot/Jam/MAIN`, then you can quickly create a Git repo for that branch of the project by supplying the depot path to **`git clone`**:

```
$ git clone https://gfserver/depot/Jam/MAIN
Cloning into 'MAIN'...
remote: Counting objects: 2070, done.
remote: Compressing objects: 100% (1379/1379), done.
remote: Total 2070 (delta 1218), reused 1074 (delta 325)
Receiving objects: 100% (2070/2070), 600.52 KiB, done.
Resolving deltas: 100% (1218/1218), done.
$
$ cd MAIN
$ git branch
* master
$ ls
src
$
$ p4 dirs //depot/Jam/MAIN
//depot/Jam/MAIN/src
```

- The depot path supplied will be automatically mapped to Git branch **master**, with all files and history for that path immediately available in the new repo.
- You can also push an existing Git repo's branch to a particular Perforce depot path:

```
$ cd myrepo
$ git push https://gfserver/depot/myproject/main master
$
$ p4 dirs //depot/*/*
//depot/myproject/main
```

- Git Fusion uses the supplied Perforce depot path, along with defaults from the global configuration file, to create a **p4gf_config** file for the repo that it automatically stores in:

  `//.git-fusion/repos/`*`depotname`*`_0xS_`*`reponame`*`_0xS_`*`branchname`*`/p4gf_config`.

- While the above examples use HTTPS, SSH URLs are also supported.
- This functionality must be enabled with the Git Fusion configuration option, **depot-path-repo-creation-enable**.
- You can restrict this functionality to a specific group of Perforce users by setting **depot-path-repo-creation-p4group** equal to a Perforce group name.
- The branch name is optional ("main" in the examples above).
- Stream depots are not supported.
- Like most options, these can be set in either the global or repo config file. For detailed usage of the above configuration options, see Table 5.1, “Global configuration file: keys and default values” on page 38.

Initializing repos on the Git Fusion server

Once you have created a repo configuration file or workspace that maps Perforce depot locations to your repo, you or your Git users can perform the initial clone that populates the Git Fusion server:

- If you, the administrator, perform the initial clone, you can absorb the time cost of initializing large repos and fix any repo configuration issues.

  The time the initial clone takes to complete depends on many factors, like the amount of Perforce data Git Fusion must translate into Git data and the amount of network traffic. For large depots, the initial clone can take several hours.

- If you choose to let your Git users initialize new repos, simply distribute the Git repo URLs to your users; the first **`git clone`** transaction will populate the repo on the Git Fusion server.

For administrators, the Repo Initialization script (`p4gf_init_repo.py`) provides a convenient means of initializing new repos.

`p4gf_init_repo.py --start` ***`n repo_name`***

Use `--start` *`n`* to copy history as of a particular changelist. The ***`repo_name`*** can be either the subdirectory name in `//.git-fusion/repos/`*`repo_name`*`/p4gf_config` or the name of a workspace.

The example below initializes a repo named "winston" with history starting at changelist 144656:

```
$ p4gf_init_repo.py --start 144656 winston
```

For information about additional options available when you run this script, see `p4gf_init_repo.py --help`.

Importing existing Git repos into Git Fusion

There are three approaches to importing existing Git repos into Git Fusion. All result in Git branches being available to Perforce users.

Two approaches can generally be executed by end users without administrative intervention (beyond initial configuration). These methods offer simplified steps when only a single branch needs to be imported.

- Use Git itself to push a single branch of a repo to a specified Perforce depot path by supplying a Perforce depot path as part of the remote URL to **`git push`**. This method is not enabled by default. It may be enabled by an administrator for some or all Git users. For information and examples for how to use this method, see "Use a Perforce depot path in a Git remote URL" on page 60.

- Create a Perforce workspace (client) to map a single branch of a repo to a specified Perforce depot path, then run **`git push`** using the workspace name as the repo name. For information and examples for how to create a Perforce workspace for this purpose, see "Configure repos from a Perforce workspace" on page 58. Once you've created a workspace, see "Importing an existing repo using a Perforce workspace or repo configuration file" on page 63.

A third approach generally requires an administrator, but offers the most configuration options.

- Define a new repo configuration file (`p4gf_config`). This approach allows importing multiple Git branches into Git Fusion. To use this approach, follow the example steps in "Creating a repo configuration file for import of existing repo" on page 63 and "Importing an existing repo using a Perforce workspace or repo configuration file" on page 63

Creating a repo configuration file for import of existing repo

The following example is a repo configuration file with a view mapping that defines a repo that does not currently exist in Perforce. It should be submitted to `//.git-fusion/repos/git_project/p4gf_config`. Based on this path, the Git Fusion repo will be named `git_project`. When the Git user pushes their project's master branch to this Git Fusion repo for the first time, Git Fusion will populate the Perforce depot at `//depot/vperry/git_project/`. When the Git user pushes other branches, Git Fusion will store changes on lightweight branches, under `//.git-fusion/branches/`.

Additional notes about creating such repo configuration files:

- The right-side **view** mapping should contain only a workspace root or an ellipsis (…). Do not specify any subdirectories. Git Fusion will create the appropriate subdirectories in the Perforce depot upon initialization.
- If the existing Git repo contains multiple branches, you have the option to map each one to a Perforce depot path, although it is not required.

  Assuming that you have configured your repo configuration file to allow for pushing branches, any unmapped branches that are pushed to Git Fusion will automatically be stored on lightweight branches, under `//.git-fusion/branches/`. At least one branch (for example, `master`) should be mapped to a Perforce depot path.

```
[@repo]
description = Git Fusion repo created from git_project
charset = utf8
enable-git-branch-creation = yes
ignore-author-permissions = no

[master]
git-branch-name = master
view = //depot/vperry/git_project/... ...
```

For more information about configuring repos, see "Configuring repos" on page 54.

Importing an existing repo using a Perforce workspace or repo configuration file

1. **Push the original existing Git repo to Git Fusion.**

   a. Clone the existing repo and **cd** into the resulting local repo directory.

   b. Retain a link to the upstream repo to enable updates.

```
git remote rename origin upstream
```

c. Ensure that you check out all branches of the repo locally.

```
git checkout -b branch upstream/branch
```

d. Establish remote tracking between the local repo and Git Fusion.

> **Note**
>
> For the ***repo_name***, subtitute either a workspace name if using a Perforce workspace, or the `p4gf_config` parent folder name if you defined a new repo configuration file (`git_project` in the example above).

```
git remote add origin https://Git_Fusion_Server/repo_name
```

> **Note**
>
> For really huge repos avoid an HTML timeout by configuring the remote origin using the SSH protocol.
>
> ```
> git remote add origin unixacct@Git_Fusion_Server:repo_name
> ```
>
> For more information about SSH authentication , see Authenticating Git Users using SSH on page 109.

e. Push the local repo's branches to Git Fusion individually, or all at once as in the command below.

```
git push -u --all origin
```

2. Verify the imported data.

a. Log in to the Git Fusion server and remove the ***repo_name*** directory from the *P4GF_HOME*`/views` directory (if you installed using the `configure-git-fusion.sh` script and accepted all defaults, this would be `~git/.git-fusion/views/`*repo_name*).

This step forces Git Fusion to rebuild the Git repo from data stored in Perforce, and is only necessary during this verification.

b. Clone the repo back from Git Fusion.

Be sure to save the repo in a different directory with a different name than the original local repo.

```
git clone unixacct@Git_Fusion_Server:repo_name newdir
```

c. Run **`git log --stat > log.new`** in the clone you created in the previous step.

d. Run **`git log --stat > log.orig`** in your original local repo.

e. Compare the two logs for any data differences.

If the logs do not match, email the data to Perforce Support (`support@perforce.com`) for assistance.

Modifying repo configuration files safely

Once a Git repo has been cloned, any changes to that repo configuration can invalidate its history in ways that prevent an identical rebuild of the Git repo after deleting it.

The following changes, however, are safe:

- **Add a Perforce branch that has no impact on Git history;** that is, a Perforce branch that does not merge into a Perforce branch already mapped to a Git branch.

  You *can* add a Perforce branch that merges into a Perforce branch that is already mapped to a Git branch, *as long as you do not delete the Git repo and try to recreate it.*

  The history reflected in the Git repo will *not* match what is in Perforce: any merges into the pre-existing branch will have been recorded as edits. That said, the content in the Git repo will match what is in Perforce, and Git Fusion will record any *future* merge actions correctly. If you delete the Git repo (using `p4gf_delete_repo.py`) and then recreate it using the repo configuration, then these edit commits will become merge commits and result in a new Git repo that is not compatible with the previous version.

  Example

  Let's say you have `//depot/main` and `//depot/dev`. There is history between the two, with changes originating in `//depot/dev` and merges into `//depot/main`. If you map `//depot/main` to `master` and initialize a new Git Fusion repo, then the merges from `//depot/dev` to `//depot/main` are recorded as edits in Git history. If you go on to add a new mapping for `//depot/dev`, you will get Git history for `//depot/dev` but it will not change those edit commits in any way. If you delete and recreate this Git repo, it will be incompatible with the original generated Git repo, because the edit commits will be regenerated as merge commits.

- **Remove a Perforce branch that touches no other Git history;** that is, a Perforce branch that never merges into another surviving branch.

  A Perforce branch that merges into any surviving branch is a parent to Git merge commits. Removing it would break history: a rebuild of history would convert merges into edits.

  Note that a push from a clone of this repo that contains additional commits on the deleted branch would recreate the Perforce branch as a lightweight branch.

Any other edits to a repo configuration file -- including changes to branch mappings -- require that you create a new repo configuration, distribute the new repo to the affected users, and delete the original. Ensure that all affected Git users push any pending commits to the original repo before you create its replacement.

Important

Whenever you remove a branch from a repo configuration file, you should also run **`p4gf_submit_trigger.py --rebuild-all-gf`** ***`myperforceserver:port [super]`***.

Converting a lightweight branch into a fully-populated branch

When you push a new Git branch that is not mapped to a Perforce branch in a repo configuration file, that new branch is submitted to Perforce as a lightweight branch, under `//.git-fusion/branches/`.

For efficiency, such branches only contain the minimal set of integrations and changes required to represent the pushed history. These branches are transparent to Git users, but their sparse nature may hinder collaboration with Perforce users.

A Git branch that is mapped to a Perforce branch in a repo configuration file will be fully-populated at the depot path specified. Again, the choice of branch treatment is transparent to Git users. When Git users and Perforce users need to share a persistent branch, it is ususally best to use a fully-populated branch. This can be accomplished in one of two ways:

- Use Git to merge changes from an unmapped branch (lightweight in Perforce) to a mapped branch (fully-populated in Perforce), and push.
- Convert a lightweight branch to a fully-populated branch, using the steps below.

1. **Add the new, as-yet unpopulated target branch to the repo's `p4gf_config` file.**

```
[my_new_branch]
git-branch-name : my_new_branch
view : //depot/my_project/my_new_branch/... ...
```

2. **In Git, create the new branch, pointing to the branch reference where you want to start commit history.**

```
$ git branch my_new_branch <branch ref or commit sha1>
```

3. **Push the new branch through Git Fusion to Perforce.**

```
$ git push origin my_new_branch
```

The branch is now fully populated in Perforce, and both Perforce and Git users can work in it.

Enabling Git users to create fully-populated branches

As discussed in "Converting a lightweight branch into a fully-populated branch" on page 65, fully-populated Perforce branches are the best choice when Git and Perforce users need to collaborate on the same branch.

An administrator or user with access may add a new branch mapping to the repo-specific `p4gf_config` file, so that when a Git user pushes to the new branch, it is fully-populated in Perforce. For more information on adding new branches, see "Modifying repo configuration files safely" on page 65

The two approaches below let Git users push fully-populated branches without administrative intervention (apart from initial configuration).

Create a fully-populated branch only when a Git user explictly chooses to do so

You can enable Git users to push some branches as fully-populated branches, for sharing with Perforce users, while letting others be pushed as lightweight branches:

1. **Enable depot branch creation in the global or repo-specific `p4gf_config` file with the `explicit` value.**

   Use the global configuration file to enable this option for all repos:

   ```
   [git-to-perforce]
   depot-branch-creation-enable = explicit
   ```

   Alternatively, use a repo-specific configuration file to enable it repo-by-repo:

   ```
   [@repo]
   depot-branch-creation-enable = explicit
   ```

   For additional configuration options, see the `depot-branch-creation-*` keys in Table 5.1, "Global configuration file: keys and default values" on page 38

2. **In Git, create a new branch.**

   ```
   $ git checkout -b new_git_branch
   ```

3. **Explicitly push the new branch using the following syntax to create and map a fully-populated branch in Perforce:**

   ```
   $ git push origin new_git_branch:depot-branch/new_p4_branch
   ```

   This creates a new fully-populated branch in the Perforce depot and maps it to the Git branch, `new_git_branch`:

   ```
   //depot/my_project/new_p4_branch/...
   ```

   In the command above, `depot-branch` is a keyword which instructs Git Fusion to create a new fully populated branch and map it to the Git branch being pushed. Similar to pushing a Swarm review, the remote branch reference `depot-branch/new_p4_branch` is never created on the remote Git Fusion server. Instead, a new remote reference is created for `new_git_branch`. After using this method, it is necessary to fetch the new remote reference, and remove the non-existent `depot-branch` reference. This can be accomplished in one step:

4. **Fetch the newly created Git branch reference, and remove local remnants from the previous operation:**

```
$ git fetch --prune origin
```

> **Note** A typical **git push** will continue to create a lightweight branch in Perforce. In this case, no pruning is necessary.

5. **Review the repo-specific p4gf_config file to see the new branch mapping created by Git Fusion.**

Create a fully populated branch every time a Git user pushes a new branch

If you want Perforce users to be able to instantly use all new branches pushed by Git users, you can elect to create fully-populated branches in Perforce whenever Git users push a new branch.

1. **Enable depot branch creation in the global or repo-specific p4gf_config file with the all value.**

   Use the global configuration file to enable this option for all repos or a repo-specific configuration file to enable it repo-by-repo.

```
[git-to-perforce]
depot-branch-creation-enable = all
```

   For additional configuration options, see the `depot-branch-creation-*` keys in Table 5.1, "Global configuration file: keys and default values" on page 38

2. **In Git, create a new branch.**

```
$ git checkout -b new_branch
```

3. **Push the new branch as usual to create a fully-populated branch in Perforce:**

```
$ git push [--set-upstream] origin new_branch
```

   This creates a new fully-populated branch in the Perforce depot and maps it to the Git branch, `new_branch`:

```
//depot/my_project/new_branch/...
```

   The optional **--set-upstream** connects local branch reference `new_branch` to remote `new_branch` to reduce the amount of typing required for future pulls or pushes.

   A push to a branch name that already exists must be a fast-forward push, the same as pushes to `master` or any other branch. Otherwise the push is rejected. A Git user who unknowningly pushes

a branch name that already exists must choose a different name, or rebase their new history on top of the existing branch's head.

> **Note** Lightweight branches may still be created where needed for accurate representation and recreation of Git merge commits.

4. **Review the repo-specific `p4gf_config` file to see the new branch mapping created by Git Fusion.**

Controlling depot location of pushed branches

Git Fusion uses `depot-branch-creation-depot-path` to determine where within Perforce to create the new branch. Git Fusion takes the value for this setting, performs string substitutions, and uses the result as the root to hold the pushed files.

There are three string substitutions:

- `{repo}`: The name of the Git Fusion repo receiving this push.
- `{git_branch_name}`: The name of the pushed branch reference.
- `{user}`: The Perforce user ID of the pusher.

Examples

Example: project/branch hierarchy in Perforce

The common and default path omits user, uses the project name as the container for all branches on that project:

```
depot-branch-creation-depot-path = //depot/{repo}/{git_branch_name}

dirk@syrinx$ git push https://syrinx/project task:dirks_task
^^^^                                 ^^^^^^^      ^^^^^^^^^^^
user                                 repo         git_branch_name

... creates ...

[pjfda2uh4rgzdixseowlk4zfki]
git-branch-name = dirks_task
view = //depot/project/dirks_task/... ...
               ^^^^^^^ ^^^^^^^^^^
               repo    git_branch_name
```

Example: Give each developer their own area in the Perforce depot.

Let each Git user have their own area under `//dev/{user}/...` to hold their own branches:

```
depot-branch-creation-depot-path = //dev/{user}/{repo}/{git_branch_name}

dirk@syrinx$ git push https://syrinx/project task:dirks_task
^^^^                                ^^^^^^^      ^^^^^^^^^^^
user                                repo         git_branch_name

... creates ...

[bzkrk5p3yvcjzjxo6ikfajzoaq]
git-branch-name = dirks_task
view = //dev/dirk/project/dirks_task/... ...
             ^^^^ ^^^^^^^ ^^^^^^^^^^
             user repo    git_branch_name
```

Creates a new branch **dirks_task** in the Git Fusion repo, visible to all Git users, and mapped to Perforce location **//dev/dirk/project/dirks_task/...**.

Example: {user} without {git_branch_name}

It is possible, but highly unlikely, that you want depot paths to include the user who created them, but *not* the Git branch name:

```
depot-branch-creation-depot-path = //depot/{repo}/{user}

dirk@syrinx$ git push https://syrinx/project task:dirks_task
^^^^                                ^^^^^^^      ^^^^^^^^^^^
user                                repo         git_branch_name

... creates ...

[tk5kurabcfeovhkadywgdxv6xq]
git-branch-name = dirks_task
view = //depot/project/dirk/... ...
               ^^^^^^^ ^^^^
               repo    git_branch_name
```

The lack of **{git_branch_name}** limits each user to a single branch per repo, because any attempt to create a second branch will map to the same path as the first, and Git Fusion will reject the push because Perforce already has files on that path:

```
$ git push https://syrinx/project task2:dirks_task2
Username for 'https://syrinx': dirk
Password for 'https://dirk@syrinx':
Counting objects: 3, done.
Writing objects: 100% (3/3), 226 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote: Perforce: Cannot create new depot branch for ref 'refs/heads/dirks_task2':
  Git depot root '//depot/project/dirk' already contains Perforce changelists.
To https://syrinx/project
 ! [remote rejected] task2 -> dirks_task2 (pre-receive hook declined)
error: failed to push some refs to 'https://syrinx/project'
```

Working with Perforce streams

You can expose a Perforce stream as a branch view within a Git Fusion repo, allowing Git users to work with that branch and have that work submitted to Perforce as work within that stream. There are two ways to map a Git Fusion repo to a stream:

- **Clone using the stream name as the repo name.**

  ```
  git clone https://git_fusion_server/stream_name
  ```

  If there is no Git Fusion repo with the name `stream_name`, Git Fusion searches for:

  - An existing Perforce workspace (client) with the name `stream_name`.
  - An existing Perforce stream with the name `//stream_name`.

  If Git Fusion finds a workspace, and that workspace is a stream workspace, then Git Fusion creates a repo with a single branch that maps that stream.

  If Git Fusion finds a stream, then Git Fusion creates a repo with a single branch that maps that stream.

  Note that Git would be confused by the `//` stream name prefix, so you must omit it from the clone command. Git can handle the internal `/`, but it will be translated to `_0xS_` when the repo name is constructed. For example, if you clone using stream `//flow/mainline`, you use `git clone https://gfserver/flow/mainline` and get a repo named `flow_0xs_mainline`.

- **Add a stream to the branch definition in a repo's `p4gf_config` file.**

  ```
  [my_new_branch]
  git-branch-name : my_new_branch
  stream : //streamdepot/my_stream
  ```

  Note that a stream branch does not include a view (there is no path on the "right hand side"), because the view is determined by the stream. A branch definition containing both stream and view will be rejected by the config file validator.

You must consider the following when you use streams with Git Fusion:

- You can include *only one stream* per branch (although you can use a stream that imports other streams).
- You *can* include both standard view-based branches and stream-based branches in the same repo.
- You cannot base a git branch on a task stream.
- You cannot change the stream view of a stream that is mapped to a Git branch.

  Git Fusion rejects pushes to Git branches whose stream view has changed since the repo was initialized.

- Git users can merge between standard view-based branches and stream-based branches.

  This means that you can "drive through the hedges," merging and copying between streams that do not have a parent-child relationship.

- Every branch mapping in Git Fusion must have the same right-hand side. Streams (other than `mainline`) with exclusionary lines or other remapping operations tend to produce different right-hand sides.

Enabling stream import paths as Git submodules

Git Fusion lets you represent stream import paths as Git submodules. In Perforce streams, import paths enable you to source files into a stream from different locations in the Perforce repository. Files included by import can be synced but not submitted, merged, or copied. Import paths are intended for situations when you want to include external libraries that you do not need to edit, such as those required for builds. Git submodules fill a similar role, allowing foreign repositories to be embedded within a dedicated subdirectory of the source tree.

Some considerations:

- Submodules generated from import paths are read-only; you cannot push changes to them.

- The process does not work in reverse: adding a submodule to a stream-based branch in Git does not add an import path to the stream.

- For environments with multiple Git Fusion instances, be aware that submodules generated from import paths use a single Git Fusion instance as their remote.

  Ensure that users of a repo containing such a submodule can access the Git Fusion instance that is set as the submodule's remote.

Configure and generate submodules from import paths

To enable the conversion of stream import paths to Git submodules:

1. **Set the `enable-git-submodules` option to `Yes` in the repo configuration file.**

   To enable import paths as submodules for all Git Fusion repos, set the option in the global configuration file. For individual repos, set the option in the repo-specific file.

   For more information, see "Configuring global defaults for repos" on page 38 and"Configure repos with a repo configuration file (p4gf_config)" on page 55.

2. **Add the SSH or HTTP address you use to clone Git Fusion repos to the repo configuration file.**

   Set the `ssh-url` or `http-url` property in the global configuration file if you are enabling submodules for all Git Fusion repos. For individual repos, set the property in the repo-specific file.

Important	• For any given repo, you can select only one protocol (SSH or HTTP) at a time.

> If at any point you need to switch from one protocol to another, you can update this configuration, but you must also edit the `.gitmodules_*stream-name*` file in the Perforce depot.
>
> - If you use the `{host}` variable in the URL property, submodule processing will use the hostname returned by the Linux function **`gethostname()`**. Verify that the value returned is the correct URL for running Git commands against the Git Fusion repo. Some network topologies can result in the return of unexpected values. Use the full hostname rather than the variable placeholder if you are not confident that the value returned will be correct.

3. **Define a repo branch using a stream with an import path.**

   The stream must observe the following rules:

   - It must include a `share...` path.
   - It cannot include nested or overlapping import paths.

     If the stream imports from another stream that itself includes an import path or includes multiple import paths that share the same directories, Git Fusion treats these nested or overlapping paths as ordinary stream paths and does not convert them into submodules.
   - The stream depot path must be populated and end with `/...` .
   - You cannot change the stream root after the Git repo is initialized.

   For more information about defining repo branches using streams, see "Working with Perforce streams" on page 71.

4. **To generate the submodules, clone the repo you created in the previous step.**

   The repo will include submodules with names derived from the depot path. The naming convention is to drop the depot path's initial `//` and terminal `/...` and replace any internal slashes with `_0xS_`. For example, a submodule generated from the import path `//foo/bar/...` would have the name `foo_0xS_bar`.

Managing and troubleshooting submodules

What are these new virtual streams that appear in the stream depot?

Git Fusion uses virtual streams as an intermediary in the creation of submodules from import paths. The virtual stream is created with the same name as its parent, with the addition of a `_p4gfv` suffix. Do not remove these virtual streams from the stream depot.

How do I change the submodule URL (ssh-url, http-url)?

If the value of `ssh-url` or `http-url` in the repo configuration file returns the wrong URL, Git Fusion cannot create submodules that work.

To fix the URL:

1. **Set `ssh-url` or `http-url` in the repo configuration file to the correct URL.**

   If you are having issues generating submodules from stream import paths, it is often because the `{host}` variable placeholder is returning the wrong hostname. Use the full hostname rather than the variable placeholder.

   For more information, see "Configure and generate submodules from import paths" on page 72.

2. **Edit the `.gitmodules` file to update the submodule URL.**

   The `.gitmodules` file is located in the top-level directory of your Git working tree and at the stream root in the Perforce. In Perforce, the file is stored with the suffix `_`*`stream-name`*.

3. **Update your clone by pulling and running `git submodule update`.**

   Perform this command for each Git client that has attempted to clone the repo.

How do I remove submodules generated from import paths?

If an import path is removed from the stream definition, Git Fusion removes the associated submodule from the Git repo the next time a user pulls from that repo.

Adding preflight commits to reject pushes

If your Perforce service is configured with submit triggers that enforce a local policy, like requiring jobs, specific content, or specific formatting in the changelist description, these triggers can interrupt Git Fusion in the middle of a push, which will damage the repository as replicated within Perforce. You could simply exclude changes that are submitted by `git-fusion-user` from these submit triggers, but you can also create preflight commits (scripts that fire when a user attempts to push a commit to a Git Fusion repo) that reject git pushes before they have a chance to set off a potentially damaging submit trigger.

Preflight commit scripts can be written much the same way as Perforce trigger scripts, which gives you the option to reuse trigger scripts (or revise them minimally) to enforce local policy before Git Fusion submits the push to Perforce.

To enable a preflight commit hook:

1. **Create the script and save it to the server that hosts Git Fusion.**

   Guidelines include the following:

   - Exit code `0` = pass (the push goes through), `1` = fail (reject the push)
   - The script must be run by the same UNIX account that runs Git Fusion (the Git Fusion service account), under the same environment.
   - The script is not invoked with a full shell, but it has access to the following environment variables:

`CWD`	Git work tree directory (parent of `.git` directory)

`P4PORT`	Perforce service (***`myperforceserver:port`***)
`P4USER`	**`git-fusion-user`**
`P4CLIENT`	**`git-fusion-`*`repo_name`***

- The script can consume the following Git Fusion variables:

`repo`	Name of the pushed repo
`sha1`	Full 40-character hexadecimal **`sha1`** of a single commit
`branch_id`	Unique identifier for the Git Fusion branch view receiving this commit
`git-branch-name`	Git branch ref (if any) associated with above branch view

- The script can consume the following standard Perforce trigger variables:

`client`	The client issuing the command. Always **`git-fusion-`*`repo`***.
`clienthost`	Hostname of the client. Always the SSH client connection.
`serverport`	IP ***`address:port`*** of the server. Always the **`P4PORT`** that Git Fusion uses.
`quote`	A double quote character
`user`	User issuing the command. This is the **`P4USER`** associated with the commit's changelist owner: either Git author or Git pusher, depending on the repo configuration options.
`formfile`	Path to temp file containing form.
`formname`	The form's name (branch name, etc). Always **`new`**.
`formtype`	The type of form (branch, etc) Always **`change`**.
`jobs`	List of job names for fix triggers.

See the **`preflight-commit-require-job.py`** and **`preflight-commit-require-case.py`** sample scripts in your **`libexec`** directory for examples.

For more information about Perforce trigger scripts and variables, see the *Perforce System Administrator Guide*, "Scripting Perforce: Triggers and Daemons".

2. **Add the script to the global configuration file or a repo-specific file, using the `preflight-commit` key.**

   Use the syntax ***`command argument`***, where command is the path to the script. Arguments can include any of the variables listed above, using the convention ***`%variable%`***, as in the following example:

```
[@repo]
preflight-commit = /home/git/myscript.sh %repo% %sha1%
```

Multiple scripts may be run in the configured order. All scripts must pass or the commit is rejected.

```
[@repo]
preflight-commit = /home/git/myscript1.sh %repo% %sha1%
                   /home/git/myscript2.sh %repo% %sha1% %git-branch-name%
```

For more information about global and repo-specific configuration files, see Configuring global defaults for repos and Configure repos with a repo configuration file (`p4gf_config`).

Adding preflight hooks to reject pushes

In addition to preflight hooks that operate on a per commit basis, as described in the previous section, it is also possible to run a command to evaluate each pushed reference. This can be useful for rejecting the deletion of a branch, for example, or rejecting any commits to a branch outside of a certain time period.

To enable a preflight push hook:

1. **Create the script and save it to the server that hosts Git Fusion.**

   Guidelines include the following:

   - Exit code `0` = pass (the push goes through), `1` = fail (reject the push)
   - The script must be run by the same UNIX account that runs Git Fusion (the Git Fusion service account), under the same environment.
   - The script is not invoked with a full shell, but it has access to the following environment variables:

`CWD`	Git work tree directory (parent of `.git` directory)
`P4PORT`	Perforce service (*`myperforceserver:port`*)
`P4USER`	`git-fusion-user`
`P4CLIENT`	`git-fusion-`*`repo_name`*

   - The script can consume the following Git Fusion variables:

`repo`	Name of the pushed repo
`user`	Name of the Perforce user performing the push
`ref`	Git branch/tag reference

`git-action`	Git branch action: **`new`** for a new branch or tag, **`update`** for new commits on an existing branch, and **`delete`** if deleting a branch or tag.

- The script can consume the following standard Perforce trigger variables:

`client`	The client issuing the command. Always **`git-fusion-`*`repo`***.
`clienthost`	Hostname of the client. Always the SSH client connection.
`command`	Always **`user-submit`**
`serverport`	IP ***`address:port`*** of the server. Always the **`P4PORT`** that Git Fusion uses.
`quote`	A double quote character
`user`	User issuing the command. This is the **`P4USER`** associated with the commit's changelist owner: either Git author or Git pusher, depending on the repo configuration options.
`formname`	The form's name (branch name, etc). Always **`new`**.
`formtype`	The type of form (branch, etc) Always **`change`**.

For more information about Perforce trigger scripts and variables, see the *Perforce System Administrator Guide*, "Scripting Perforce: Triggers and Daemons".

2. **Add the script to the global configuration file or a repo-specific file, using the `preflight-push` key.**

   Use the syntax ***`command argument`***, where command is the path to the script. Arguments can include any of the variables listed above, using the convention ***`%variable%`***, as in the following example:

```
[@repo]
preflight-push = /home/git/myscript.sh %repo% %user%
```

   Multiple scripts may be run in the configured order. All scripts must pass or the commit is rejected.

```
[@repo]
preflight-push = /home/git/myscript1.sh %repo% %user%
                 /home/git/myscript2.sh %repo% %user% %ref%
```

   For more information about global and repo-specific configuration files, see Configuring global defaults for repos and Configure repos with a repo configuration file (`p4gf_config`).

Limiting push size and disk usage

In certain cases, large pushes from Git to Git Fusion can have undesireable effects.

- Pushes to Git Fusion are immediately stored in Perforce, where history is purposely immutable. Extensive history can be easily added, but difficult to back out.
- A large push can impact performance of the Git Fusion server and Perforce server, that possibly affects other users.
- A large push can consume a significant amount of disk space on the Git Fusion server and Perforce server, and lead to longer backup, data replication, and maintenance times.
- In a SaaS environment, extra resource usage can incur direct costs.

To curtail these effects, an administrator may wish to constrain the amount of data that can be pushed from Git to Git Fusion at any one time, or in total.

Limits for a single push

Git Fusion offers the ability to restrict pushes which exceed various quotas defined in Git Fusion configuration files.

- These quotas are set by using the following configuration options, under the `[quota]` category:
  - `limit_space_mb`
  - `limit_commits_received`
  - `limit_files_received`
  - `limit_megabytes_received`
- Any combination of these configuration options is allowed, however each additional metric requires some additional processing time.
- The defaults for all quotas are zero, effectively disabling quota enforcement.
- Like most options, these can be set in either the global or repo config file. For detailed usage of the above configuration options, see Table 5.1, “Global configuration file: keys and default values” on page 38.

Limit total Git Fusion disk usage

It is also possible to limit the total size of all repos managed by Git Fusion. To facilitate this limitation, an administrator sets, and Git Fusion respects, a Perforce **`p4 key`** named `git-fusion-space-remaining-mb` that defines the remaining number of megabytes permitted to be pushed to Git Fusion.

- The format of the value may be either a natural number or a decimal fraction. That is, the administrator may set this to a natural number, and as Git Fusion processes push operations, fractional amounts will be subtracted from this value.

- This value is only ever decreased by Git Fusion, and only when a push has been successfully completed.
- Note that Git Fusion only measures the Git repository usage, and not the disk usage in the Perforce server. There are simply too many factors involved to permit making any reasonable estimate, and as such, only an administrator will have the necessary information to determine and set any usage limit.

To set a value for the `git-fusion-space-remaining-mb` key:

1. **Disable new Git Fusion sessions on all instances for all repos**

   `$ p4 key -i git-fusion-prevent-sessions-all`

2. **Wait for all pending pull/push operations to finish (i.e. no keys are returned by the command below)**

   `$ p4 keys -e git-fusion-view-*-lock`

3. **Set the `git-fusion-space-remaining-mb` key to the desired overall space limit (measured in megabytes)**

   `$ p4 key git-fusion-space-remaining-mb 1000`

4. **Re-enable new Git Fusion sessions**

   `$ p4 key -d git-fusion-prevent-new-sessions`

View current disk usage

In addition to standard shell and Git commands already available to administrators, Git Fusion offers two convenient ways to view repo disk usage.

- Perforce **`p4 keys`** store the current size in megabytes of each repo that Git Fusion manages, as well as the size of any push which is currently going on. To view these sizes, run the following commands from any host with access, substituting the desired repo name:

```
$ p4 key git-fusion-view-repo-total-mb
$ p4 key git-fusion-view-repo-pending-mb
```

- The script `p4gf_push_limits.py` can be run interactively on any Git Fusion server to display what is known about the available Git Fusion repositories, including the values in the total and pending keys as well as the disk usage of the repo on the Git Fusion host on which the script is currently running. Additionally, this script can be used to update the total and pending keys to reflect the current reality.

Detecting Git copy/rename and translating to Perforce

You can elect to honor Git's reported file actions for **`copy`** and **`rename`** when pushing repos into Perforce via Git Fusion.

By default Git Fusion does not detect Git copy/rename.

- A file copied in Git will result in a **p4 add** in Perforce.
- A file renamed in Git will result in a **p4 delete** and a **p4 add** in Perforce.

Git itself does not record copy/rename actions.

- Git records file *state*, not file *actions* to change that state.
- To report file actions, Git compares before/after states, then deduces file actions.
- To detect copy/rename actions, Git scans before/after file lists looking for matching content, and if found, reports as a copy or rename.

Git copy/rename detection and translation into Perforce is enabled by two configuration options.

- With these options enabled, Git Fusion uses Git's **--find-copies** and **--find-renames**.
- Git provides detection of copy/rename for less than identical files by setting the options to values < 100% and Git Fusion translates the results into the corresponding Perforce actions.
- Perforce retains file history across depot branches. Copy or rename actions remain recorded in the branch where they occurred.
- These options are disabled by default.
- Like most options, these can be set in either the global or repo config file. For additional configuration options, see the `enable-git-find-copies` and `enable-git-find-renames` keys in Table 5.1, "Global configuration file: keys and default values" on page 38.

What happens when Git guesses incorrectly?

- **False Negative: Git misses a copy or rename action.** The intention is lost. No integration between associated files is added to Perforce.
  - Copy is recorded as **p4 add**.
  - Rename is downgraded to a **p4 delete** and **p4 add** pair.
- **False Positive: Git reports a copy or rename where none was intended.** An association is inferred where none was intended. Perforce records an integration between two files that are similar in content.
  - Copy creates a **p4 copy** link between a new file and a similar existing file.
  - Rename creates a **p4 move** link between a new file and a similar existing file that is deleted in the same commit.
- **Guesses cannot be backed out.** Perforce history is purposely immutable. Once Git Fusion records a commit with Git's bad guess, the erroneous integration (or lack of integration) is part of history forever. Editing past history in Perforce is difficult: **p4 obliterate**, checkpoint surgery, and/or a call to Perforce Support is required.

Disconnecting a Git Fusion repo from the Perforce service

You can sever a Git Fusion repo's connection to Perforce and retain the repo.

To sever the repo’s connection:

1. **Copy the Git Fusion repo directory from `~/.git-fusion/views/`*`repo_name`*`/git` to a location outside of `.git-fusion`.**

2. **Delete `.git/hooks/pre-receive` from the repo copy.**

After you copy the Git Fusion repo directory, you can delete the original repo from Git Fusion by running the Git Fusion Delete Repo script (`p4gf_delete_repo.py`).

Deleting Git Fusion repos

To delete Git Fusion repos from Perforce, use the Git Fusion Delete Repo script (`p4gf_delete_repo.py`).

Important	Whenever you run `p4gf_delete_repo.py`, you should also run **`p4gf_submit_trigger.py --reset`** ***`myperforceserver:port`***.

Chapter 6 Additional Administrative Tasks

This chapter discusses the following administrative tasks:

- Configuring logging
- Viewing changelist information
- Managing Git Fusion p4 keys
- Managing Git Fusion server IDs
- Stopping the Git Fusion server
- Preventing new Git Fusion sessions
- Backing up and restoring Git Fusion
- Adding Git Fusion and Perforce server components
- Administering the Git Fusion OVA

Configuring logging

Git Fusion provides a logging configuration file, `git-fusion.log.conf`, that contains preset defaults and comprehensive information on establishing logs.

The `configure-git-fusion.sh` script puts this logging configuration file in the `/etc` directory, configures `syslog`, and configures automatic log rotation.

If you do not want to use the default logging configuration, you can customize your logging options using the `git-fusion.log.conf` file. For additional logging configuration information, see `http://answers.perforce.com/articles/KB_Article/Configuring-Git-Fusion-Logging`

Viewing changelist information

Git Fusion stores pushed Git commits in the Perforce service as Perforce changelists. Perforce users can view Git users' changes using Perforce tools. Note that a single Git push can contain multiple commits, and therefore can spawn multiple changelists.

Each changelist resulting from a Git commit includes the author's name as the changelist owner (if you used the default value for `change-owner` in your repo configuration), in addition to the following Git commit information, which appears in the **Description** field of the Perforce changelist:

- The Git commit message text.
- The phrase `Imported by Git`.
- Git author, Git committer, and `SHA1` information.
- The pusher's name, if the Git user who pushed the change is not the author.

  The pusher is always the user who authenticated with HTTP or SSH.

- A `push-state` field with a value of `complete` or `incomplete`.
- Branch information.
- Perforce Jobs information, if the Git user includes a job number with the commit.

  If you are using P4V, also see the changelist's **Job** field.

To determine which Git Fusion repo pushed a change to Git Fusion, refer to the **Client** field (**Workspace** field in P4V) of the appropriate Perforce changelist.

The **Date** field **(Date submitted** field in P4V) includes a date and timestamp of when the Git commit was successfully pushed to Git Fusion. This is not the date the author or committer pushed the commit to his or her local Git repo; you must use Git to review this information.

Managing Git Fusion p4 keys

All Git Fusion p4 keys start with `git-fusion-`. To find all Git Fusion p4 keys on a Perforce service, run:

- All Git Fusion p4 keys: **`p4 keys -e git-fusion-*`**
- Submit Triggers: **`p4 keys -e git-fusion-*-submit-*`**

Managing Git Fusion server IDs

Each Git Fusion instance must have its own unique server ID. Git Fusion uses the computer's hostname as a default ID value; however, sites where multiple Git Fusion instances run on the same host must specify a unique server ID for each instance.

To change the server ID, log in as the Git Fusion service account and run:

```
p4gf_super_init.py --id new_server_ID
```

Git Fusion stores the server ID under `P4GF_HOME/server-id`, where `P4GF_HOME` is the Git Fusion working directory specified in the Git Fusion environment configuration file (`~/p4gf_environment.cfg`).

For more information, run:

```
p4gf_super_init.py -h
```

Important	Do not change a server ID while Git Fusion is processing a Git request.

Stopping the Git Fusion server

Git Fusion runs only when your users access it through SSH or HTTP(S). If you are using HTTP(S) authentication, you must stop the web server to stop Git Fusion. If you are using SSH authentication, you must disable the authorized keys update process to stop Git Fusion.

1. **Log into the Git Fusion UNIX service account.**

2. **Disable the update authorized keys process.**

    - Move the `~/.ssh/authorized_keys` or `~/.ssh2/authorized` keys file (or both, if applicable to your implementation) to another location to prevent users from activating Git Fusion.

    - Disable any `cron` jobs or triggers that automatically run the Update Authorized Keys script (`p4gf_auth_update_authorized_keys.py`).

      See Use a `cron` job to copy public keys to Git Fusion

Preventing new Git Fusion sessions

New git sessions/connections may be prevented for all of Git Fusion, for a specific instance, or for a specific repo.

- Disable all new Git Fusion sessions on all instances

```
$ p4 key -i git-fusion-prevent-sessions-all
```

- Disable all new Git Fusion sessions on a specific instance

```
$ p4 key -i git-fusion-prevent-sessions-instance--serverid
```

- Disable all new Git Fusion sessions on all instances for a specific repo

```
$ p4 key -i git-fusion-prevent-sessions-repo--reponame
```

- Disable all new Git Fusion sessions on a specific instances for a specific repo

```
$ p4 key -i git-fusion-prevent-sessions-instance-repo--serverid--reponame
```

Delete the applicable p4 key to re-enable new sessions.

Note: new sessions for p4 users who are members of p4 group 'git-fusion-admin' are NOT prevented by these keys.

Backing up and restoring Git Fusion

Git Fusion can restore Git history from Perforce using the standard Perforce backup and recovery process—as long as that Git history *was* stored in Perforce. You cannot restore commits and branches that have not been pushed, since they exist only in a Git user's local repo.

To back up Git Fusion, use standard Perforce backup procedure to take a checkpoint and back up the `//.git-fusion` depot and the depot locations that map to your Git Fusion repos.

To recover from a backup:

1. **Reinstall Git Fusion.**

When you reach the installation step in which you run `p4gf_super_init.py`, you have the option to set the Git Fusion server ID to be the same as the original Git Fusion server or set a new Git Fusion server ID.

To use the same Git Fusion server ID:

a. Get the server ID of the original server (in this example, '`gf`'):

```
$ p4gf_super_init.py --user perforce_super_user --showid
Git Fusion server IDs [('gf.example.com', 'gf')]
```

b. Set the server ID:

```
$ p4gf_super_init.py --user perforce_super_user --id gf
```

To specify a different server ID (in this example, '`gf2`'), run:

```
$ p4gf_super_init.py --user perforce_super_user --id gf2
```

If you use a new server ID, you must remove any unused service users, clients, or p4 keys that belong to the original, failed Git Fusion server ID.

2. **Copy your users' public SSH keys from Perforce to the `authorized_keys` file.**

   As the Git Fusion service account (`git`), run `p4gf_auth_update_authorized_keys.py`.

3. **Initialize the Git Fusion repos.**

   See Initializing repos on the Git Fusion server

For more information about backing up and restoring a Perforce service, see the *Perforce System Administrator's Guide*, *Supporting Perforce: Backup and Recovery*.

Adding Git Fusion and Perforce server components

You can incorporate multiple Git Fusion servers into your implementation.

For optimal performance, Git Fusion instances should generally be connected directly to the Perforce *Master* or *Commit* server.

- A low-latency connection is always recommended between Git Fusion instances and the Perforce server.
- WAN connections should be avoided, and co-location is strongly advised.
- Git users at remote sites should push across the WAN to a Git Fusion server co-located and directly connected to the *Master* or *Commit* server.

In Perforce Clusters, the *Router* must be configured to direct all Git Fusion instance requests directly to the *Depot* server.

Add Git Fusion servers

You can implement Git Fusion on multiple hosts that connect to a single Perforce service. Simply repeat installation for each Git Fusion instance. Functionality within Git Fusion handles the coordination among the instances. You do need to be aware of the following:

- Each Git Fusion instance has a unique server ID.

  The server ID is set during installation by running either of `configure-git-fusion.sh` or `p4gf_super_init.py`.

  For more information about how the Super Initialization script (`p4gf_super_init.py`) handles Git Fusion server IDs see `p4gf_super_init.py --help`.

- Each Git Fusion instance has a separate Perforce workspace (client) with the name `git-fusion--`*`server-id`*.

- Each Git Fusion instance has a separate Git Fusion service user with the name `git-fusion-reviews-`*`server-id`*.

- Multiple Git Fusion instances can act as the remote for the same Git repo.

Special considerations for P4Broker

P4Broker may interfere with Git Fusion operations by rewriting commands. If Git Fusion is connected to a P4Broker, the broker must not alter the semantics of any commands issued by Git Fusion. For assistance using Git Fusion with P4Broker, contact Perforce Technical Support at `support@perforce.com`.

For more information, see *Distributing Perforce Guide*, *The Perforce Broker*.

Git Fusion with Proxies, Replicas, and Edge servers

In certain topologies, Git Fusion instances may also be connected to *P4Proxy*, *Replica*, and *Edge* servers. Please contact Perforce Technical Support at `support@perforce.com` for assistance and considerations with such server components. The following guidelines apply:

- The Git Fusion Atomic Push submit triggers must always be implemented on the *Master* or *Commit* server.

- All **`git push`** operations should still be done against a Git Fusion server co-located and directly connected to the *Master* or *Commit* server.

- Any Git Fusion server connected to an *Edge* or *Replica* server should be configured for "read-only" mode (set `READ_ONLY` in `p4gf_environment.cfg`).

Once Git changes have been translated into Perforce, they can be replicated and distributed in the same way as changes that originated in Perforce.

Delete repos on multiple hosts

Invoke the Delete Repo script (`p4gf_delete_repo.py`) on each Git Fusion host to remove local files and the associated object cache client.

Note	Avoid running the Delete Repo script on multiple hosts simultaneously.

Administering the Git Fusion OVA

This section discusses administration tasks specific to the Git Fusion OVA.

Authentication and the OVA

The OVA installation of Git Fusion supports both HTTPS and SSH authentication. If you want to use SSH authentication, note the following:

- You cannot log into the Git Fusion virtual machine as `git` using SSH, because the SSH configuration will try to invoke Git Fusion.
- There is a `cron` job for the `git` user that polls for new SSH public key information every minute.

For more information about using HTTP(S) and SSH authentication for Git Fusion, see:

- Chapter 4, "Setting up Users" on page 21
- "Referencing Git Fusion repos" on page 94
- "Providing SSH keys for Git Fusion authentication" on page 93

Perforce Server and the OVA

You have the option of using the Perforce service included with the OVA or your own external Perforce service. If you are using the included Perforce service, note the following:

- The included Perforce service is running in the 20/20 license mode. The system is limited to 20 clients once the number of files exceeds 1000.
- The pre-configured Perforce accounts (`admin`, `super`, and `git-fusion-user`) have not been assigned passwords.
- The Perforce service is running on port 1666.

For information about using an external Perforce service, see "Connecting the Git Fusion OVA installation to your Perforce service" on page 6.

Start and stop scripts

The Git Fusion OVA includes the following shell scripts to simplify maintenance of its internal Perforce service:

- `/etc/init.d/p4d`: This initialization script enables you to start, stop, restart, and get the status of the Perforce Server.
- `/etc/p4d.conf`: This configuration script stores and sets environment variables that are used by the `/etc/init.d/p4d` initialization script.

You can use the standard wrapper script (`/usr/sbin/service`) to invoke these scripts:

- **`$ sudo service p4d start`**
- **`$ sudo service p4d stop`**

SSH key management console

The OVA supports both SSH and HTTPS authentication. If you choose to use SSH, the OVA's online SSH key management console enables you to do the following:

- Upload user SSH keys using the **Git Fusion Config: Upload User Key File** page.

  For more information, see Set up SSH authentication using the OVA's SSH key management console
- If you are using Git Fusion with your own external Perforce service, rather than the one included with the OVA, change the Perforce service connection using the **Git Fusion Config: Perforce Server** page.
- View system information using the **System** tab.
- Shut down and reboot the Git Fusion service using the **System** tab.
- Configure time zone settings using the **System** tab.
- View and refresh network status information using the **Network** tab.
- Change network address settings using the **Network** tab.
- Set a network proxy server using the **Network** tab.

To access the SSH key management console, go to the the IP address displayed in the window that appears when you start the OVA VM and log in as `root`, using the password you assigned during installation. For more information, see Installation steps in the chapter, Installing Git Fusion using the OVA.

Modify Perforce Server Triggers to Ignore Git Fusion

If your Perforce service is configured with triggers that enforce a local policy, these triggers must be modified to allow Git Fusion to operate freely.

The easiest check for Git Fusion operations is to look for `git-fusion-user`.

Some operations that Git Fusion performs which your triggers must not prevent:

- Submit changelists whose descriptions fail formatting or content policies.

  See Adding preflight commits to reject pushes for how you can configure Git Fusion to enforce these policies on Git users.

- Create clients with wide-open views.

p4gf_config2

For most repos, Git Fusion creates a second configuration file `p4gf_config2`.

This file holds Git Fusion internal data:

- branch defintions to hold Git commits that do not fit into any of the branches defined in `p4gf_config`
- original view mappings of stream-based branches so that Git Fusion can detect changes to stream views.

Git Fusion controls this file. Do not edit this file.

p4gf_environment.cfg

Git Fusion loads many environment variables from the Git Fusion environment configuration file `~/p4gf_environment.cfg`.

Typically this file is created or written by `configure-git-fusion.sh` as part of initial setup. You can edit this file with additional customizations.

This file contains settings that tell Git Fusion

- where to find Git Fusion's home directory: `P4GF_HOME`
- how to talk to Perforce, such as `P4PORT` and `P4CHARSET`
- where within Perforce to find data, such as `P4GF_DEPOT`
- where to find Git itself: `GIT_BIN`.
- Git Fusion server-specific configuration such as `READ_ONLY` and `MAX_TEMP_CLIENTS`

How Git Fusion loads its environment:

1. Git Fusion reads environment variable `P4GF_ENV` if defined. This is the path to `p4gf_environment.cfg`. The file can have any name, not just `p4gf_environment.cfg`.

   This is rare. Most installations leave `P4GF_ENV` undefined.

2. Git Fusion reads `~/p4gf_environment.cfg`

3. Git Fusion loads the environment variables defined in the above environment config file.

Environment Variables

Git Fusion uses several environment variables. Both the OVA and OS package installs configure these automatically for you and set them in the Git Fusion environment configuration file (`~/p4gf_environment.cfg`).

- `P4GF_TMPDIR`: where Git Fusion temporarily stores files as it extracts them from one versioning system and adds them to the other. Defaults to `tmp` in `P4GF_HOME`. Translation tends to run faster if `P4GF_TMPDIR` point to the same disk as the Git Fusion working directory `P4GF_HOME`.

Time Zone Configuration

Git Fusion uses a p4 key named `git-fusion-perforce-time-zone-name` to determine the time zone in use on the system. This is normally set during configuration via the `configure-git-fusion.sh` script. The value is used when converting Perforce changes to Git commits. Acceptable values are in the Olson format.

- `P4GF_TMPDIR`: where Git Fusion temporarily stores files as it extracts them from one versioning system and adds them to the other. Defaults to `tmp` in `P4GF_HOME`. Translation tends to run faster if `P4GF_TMPDIR` point to the same disk as the Git Fusion working directory `P4GF_HOME`.

Chapter 7 Tips for Git Users

This chapter provides information to help Git users who are working with Git Fusion repos.

- "Requirements, restrictions, and limitations" on page 93
- "Providing SSH keys for Git Fusion authentication" on page 93
- "Referencing Git Fusion repos" on page 94
- "Sharing new Git branches with Perforce users" on page 94
- "Referencing Perforce jobs in a commit" on page 94
- "Using Git Fusion extension commands" on page 95
- "Using Swarm for code review" on page 97

Requirements, restrictions, and limitations

The Git client version must be able to connect to a Git 1.8.2.3 server.

Git Fusion does not support:

- Localized messages.

  Git Fusion messages appear in US English only.

- Perforce file types `apple` and `resource`.
- Perforce Labels functionality.

  Git users of Git Fusion do not have the ability to read or edit any project-related Perforce Labels maintained by Perforce team members. Git commit tags are supported (and stored in Perforce at `//.git-fusion/objects/repos/`*`repo_name`*`/tags/`) but are not translated into Perforce Labels.

- Renaming detection functionality.

  Git Fusion does not use the `-M`, `-C`, and `--find-copies-harder` flags when copying from Git to Perforce. Instead, it handles and logs file renaming as a file add and delete.

- Perforce file locks on files that Git users might also edit.

  Git Fusion cannot copy pushed commits into Perforce if those commits modify Perforce files that are locked by 'p4 lock' or exclusive open filetype +l. If Git Fusion encounters a file that was locked after a **`git push`** has started, Git Fusion unlocks the file and submits Git content on top of the previously locked file.

Providing SSH keys for Git Fusion authentication

To enable SSH authentication to the Git Fusion server, users generate SSH private-public key pairs locally and provide their public key to Git Fusion. This must be done either by sending the public key

to a Git Fusion administrator or by using a Perforce client application to submit the public key file directly to `//.git-fusion/users/user_name/keys/` in Perforce.

For more information about how Git Fusion uses SSH authentication, see Authentication.

Referencing Git Fusion repos

The Git Fusion repo URL follows the standard Git command convention to access a remote repo, except that the repo is referenced by repo name, rather than a directory that holds the repo.

Using HTTP(S) authentication, the syntax is `https://git-fusion.example.com/repo_name`, where `repo_name` is the name of the Git Fusion repo.

A command to clone the repo "winston" using HTTP(S) authentication therefore might look like this:

```
$ git clone https://git-fusion.example.com/winston
```

Using SSH authentication, the syntax is `unixacct@hostname:repo_name`:

- `unixacct` is the UNIX account that runs Git Fusion on the host server.
- `hostname` is the IP address or host name of the server that hosts Git Fusion.
- `repo_name` is the name of the Git Fusion repo.

For example: `git@git-fusion.example.com:winston` or `fusion@ub:bltwub`.

A git clone command using SSH authentication therefore might look like this:

```
$ git clone git@git-fusion.example.com:winston
```

Sharing new Git branches with Perforce users

Git users can interact with Git Fusion just like they would with any other Git server. When new branches are pushed to Git Fusion, they are immediately available for use by other Git users. To see how Git users can create new branches to share with Perforce users, see “Enabling Git users to create fully-populated branches” on page 66.

Referencing Perforce jobs in a commit

If Git users receive Perforce job information from team members, they can include the job’s alphanumeric identifier in a commit. Git Fusion adds the Perforce jobs to the Perforce changelist in the Description and Job fields, recording the job as fixed.

To include Perforce jobs in a commit message, enter the field `Jobs:` followed by the job’s alphanumeric identifier. You can include or omit a space before the identifier. If you are noting multiple jobs in the commit, enter each job on a separate line. For example:

```
Jobs: jobA00876B
   jobA00923C
```

By default, Git Fusion expects the values you enter to be in the format "**job*nnnnn***"as defined in the **Job** field in the Perforce job specification, but your administrator can enable other field values to be recognized and passed to the changelist description.

For example, if your organization uses the **DTG-DTISSUE** field in the job specification to associate jobs with JIRA issues, you administrator can enable Git Fusion to recognize JIRA issue identifiers, and you can enter JIRA issue IDs in your Git commits, like this:

```
Jobs: TPUB-1888
   TPUB-1912
```

For more information, see the description of the **job-lookup** option in Table 5.1, “Global configuration file: keys and default values” on page 38.

Using Git Fusion extension commands

Git Fusion includes the following commands that extend Git command functionality:

- **@help**: Shows Git Fusion Git extension command help.
- **@info**: Shows Git Fusion version information.
- **@list**: Shows repos and actions available to you, depending on your group permissions.

  For more information about how permissions determine what you can view using @list, see “How permissions affect the @list command” on page 97.
- **@status@repo**: Reports a message indicating the status of the push operation for a particular repository.

Note	If a push failure occurs very early in Git Fusion's process, the failure may not be recorded by @status. In this case, @status will report the status of the previous push.

- **@status@repo@pushID**: Reports a message indicating the status of a particular push operation, identified by the push ID number, for a particular repository.

  The push ID is displayed in the output of the client when performing a push to Git Fusion.
- **@wait@repo**: Reports a message when all push operations have completed for a particular repository.

Note	The @wait command will attempt to acquire the lock for the repository, which will cause it to wait for any push operation to complete before returning control to the client.

- **`@wait@repo@pushID`**: Reports a message when a particular push operation, identified by the push ID number, has completed for a particular repository.

  The push ID is displayed in the output of the client when performing a push to Git Fusion.

To use a Git Fusion extension command with SSH authentication, run **`git clone`** with the command in place of the repo name. For example:

```
$ git clone git@git-fusion.example.com:@info
Cloning into '@info'...
Perforce - The Fast Software Configuration Management System.
Copyright 2012-2015 Perforce Software.  All rights reserved.
Rev. Git Fusion/2015.1/997473 (2015/02/02).
SHA1: 12b9e102a892e0fd3cb6be246af4da2626ff1b24
Git: git version 1.8.2.3
Python: 3.3.2
P4Python: Rev. P4PYTHON/LINUX32X86_64/2014.1/925900 (2014.1/821990 API) (2014/08/26).
...
fatal: Could not read from remote repository.
```

Because a Git Fusion extension command is not a valid repo, Git terminates with the sentence: `fatal: The remote end hung up unexpectedly`.

If you are using HTTP authentication, extra output from Git Fusion is discarded by the Git client, which means that these special commands fail to return the information you want.

```
$ git clone https://git-fusion.example.com/@info

Cloning into '@info'...
fatal: https://git-fusion.example.com/@info/info/refs not valid: is this a git repository?
```

You can use your web browser to view Git Fusion output, or use `curl`, as in the following example:

Note	Git Fusion is not installed with `curl`. You will need to run the following command on another machine or install `curl` on the Git Fusion machine.

```
$ curl --user p4bob https://git-fusion.example.com/@info

Enter host password for user'p4bob':
Perforce - The Fast Software Configuration Management System.
Copyright 2012-2015 Perforce Software.  All rights reserved.
Rev. Git Fusion/2015.1/997473 (2015/02/02).
SHA1: 12b9e102a892e0fd3cb6be246af4da2626ff1b24
Git: git version 1.8.2.3
Python: 3.3.2
P4Python: Rev. P4PYTHON/LINUX32X86_64/2014.1/925900 (2014.1/821990 API) (2014/08/26).
...
```

How permissions affect the @list command

The repos returned by the **@list** command are determined by your repo permissions. You must have at least pull permissions, granted by membership in a repo-specific pull or push group, the global push or pull group, or the `git-fusion-permission-group-default` key, to return repos with the **@list** command:

- If you have pull permissions for the repo, the repo will be listed with the note, "pull."
- If you have push permissions for the repo, the repo will be listed with the note, "push."
- If you have neither pull nor push permissions, the repo does not appear in the list.

If the `read-permission-check` property is set to `user` in the global configuration file, then you must have pull access (through membership in a repo's pull or push group, the global pull or push group, or by virtue of the `git-fusion-permission-group-default` key setting) *and* have read access in the Perforce Protects table to all of the depot locations that map to that repo's branch definition.

Note that if the `git-fusion-permission-group-default` key is set to `pull` or `push`, all users can list all repos using the **@list** command.

For more information about how Git Fusion permissions work, see "Authorization" on page 21.

Using Swarm for code review

If your organization's Perforce implementation includes Swarm 2014.1 or above and you are licensed to use Swarm, Git Fusion lets you initiate and amend pre-commit Swarm code reviews using Git.

For additional information about how to use Swarm, see the Swarm help.

Create a Swarm review

To create a Swarm review branch:

1. **Create the new review branch using the following syntax:**

```
$ git push origin task1:review/master/new
```

Task1 is the current Git task branch and *master* is the target branch.

Note	The target branch must be mapped to a named Perforce branch in the Git Fusion repo configuration.

When the command completes, the output indicates the new review id (in this case `1234`):

```
Counting objects: 11, done.
Delta compression using up to 24 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 1.76 KiB, done.
Total 6 (delta 5), reused 0 (delta 0)
remote: Perforce: 100% (1870/1870) Loading commit tree into memory...
remote: Perforce: 100% (1870/1870) Finding child commits...
remote: Perforce: Running git fast-export...
remote: Perforce: 100% (2/2) Checking commits...
remote: Processing will continue even if connection is closed.
remote: Perforce: 100% (2/2) Copying changelists...
remote: Perforce: Swarm review assigned: review/master/1234
remote:
remote: Perforce: Submitting new Git commit objects to Perforce: 3
To https://gfprod.perforce.com/gfmain
 * [new branch]      task1 -> review/master/new
```

The process of initiating a Swarm review creates a new pending changelist in Perforce, with a changelist number identical to the review ID. This changelist contains shelved file integrations to the depot files in the branch view.

Note

While the Git Fusion output reports the new review ID, Git itself does not know about this review branch, so it reports `task1 -> review/master/new` instead of `task1 -> review/master/1234`. To get the review branch ID, use `git fetch --prune`, followed by `branch -a` or `-r` to view the branch list.

2. **View the review in Swarm.**

   Go to Swarm and browse the activity stream to find the review.

Note

If the target Git branch maps to a Perforce branch that is included in a Swarm project, all members of that project receive a notification email when you create a review. The email includes a link to the review in Swarm.

Use Swarm to approve, reject, or comment on reviews.

Amend a Swarm review

You can amend an existing review using `git fetch` and `git checkout`. You must follow these steps even if you are the Git user who initiated the review.

1. **Fetch the review head.**

   In the following example, the target branch is `master`, the review ID is `1234`, the Git Fusion server hostname is `gfserver`, and the remote repo name is `p4gf_repo`:

```
$ git fetch --prune origin
From gfserver:p4gf_repo
 * [new branch]      review/master/1234 -> origin/review/master/1234
 x [deleted]         (none)     -> origin/review/master/new
```

The `--prune` option lets the local Git repo delete the unwanted `review/master/new` reference created by the initial `git push origin task1:review/master/new` command.

2. **Check out the review head.**

```
$ git checkout review/master/1234
```

3. **Make your changes.**

4. **Push your changes to the review.**

```
$ git push origin review/master/1234
```

Note

If you get review feedback that is better expressed as a git rebase and cleaned up history, you can make your changes and push them as a new review.

You *cannot* rebase, clean up history, and then push your changes to the *same* review.

View reviews created by other Git users

You can view all reviews that were initiated in Git. First you need to fetch the existing branches in the current Git Fusion repo:

```
$ git fetch --prune origin
```

Then you can list all branches, including review branches, for the current Git Fusion repo:

```
$ git branch -a
  dev
* master
  remotes/origin/master
  remotes/origin/task1
  remotes/origin/review/master/1234
  remotes/origin/review/master/1236
  remotes/origin/review/master/1358
  remotes/origin/review/task1/1235
  remotes/origin/review/task1/1244
  remotes/origin/review/task1/1347
```

Note

Git users cannot see Swarm reviews initiated in Perforce.

View amendments made by other Git users

To view review amendments made by other Git users, fetch the Git reference for the review. If you want to work with the review, assign it to a local reference.

```
$ git fetch origin review/master/1234:myreview
From mac-bob:myrepo
 * [new branch]      review/master/1234 -> myreview

$ git checkout myreview
Switched to branch 'myreview'
```

Note

> Git users cannot see amendments to Git-initiated reviews if those amendments were made in Perforce.
>
> Indeed, a Perforce user should not amend a Swarm review initiated in Git, because if a Git user attempts to make an amendment after the Perforce user does, the Git user's new file actions could overwrite ("clobber") the shelved file actions performed by the Perforce user.

Additional tips

Be aware of the following when you create Swarm reviews with Git Fusion:

- You should not create Swarm reviews targeted for lightweight branches.

  The target branch must be mapped to a named Perforce branch in the Git Fusion repo configuration.

- You cannot delete a Swarm review.
- You can delete the remote Git branch mapped to the Swarm review from the Git Fusion repo.

  ```
  $ git push origin :review/master/1234
  ```

- You cannot approve, reject, or comment on reviews using Git; you perform the review itself in Swarm.

  You can *effectively* accept and submit a review using Git by merging the review into its destination branch and pushing that merge. Swarm, however, will not know about what you have done. You can close the review in Swarm by manually marking the review as approved.

- Git Fusion reviews do not display the individual task branch commits that make up the review; only the merged commit diffs are shown.

Chapter 8 Troubleshooting

This chapter provides solutions to some common error messages and issues you may encounter when administering or using Git Fusion; however, it does not provide a definitive listing of all possible error scenarios.

For help with using Git Fusion, contact your Perforce Technical Support representative at support@perforce.com.

Clone issues

AppleDouble Header not recognized

During a clone, the following message appears:

```
Unable to read AppleDouble Header
```

The client view contains Perforce-specific file types not supported by Git Fusion, like `apple` and `resource`. Update the view to exclude the files and run the Git Fusion Delete Repo script to delete the outdated repos; see `p4gf_delete_repo.py --help`.

.bashrc source line prevents cloning

During a clone, the following message appears:

```
git clone git@server:@info Cloning into @info bash: p4gf_auth_server.py: command not found
fatal: could not read from remote repository
```

This error may indicate that, if you are using a `.git-fusion-profile` file, the line `source .git-fusion-profile` cannot be read or found within the `.bashrc` file. We recommend putting this line at the top of the `.bashrc` file so that it can be correctly parsed during setup.

For more information, see Connecting the Git Fusion OVA installation to your Perforce service .

File cannot be converted to specified charset

During a clone, a message similar to the following appears:

```
error: failed winansi conversion for //depot_path/file_name
```

This error indicates that one or more files cannot be converted to the Unicode `charset` value specified for the repo, which prevents the repo from being cloned.

To correct this issue, you must do one of the following:

- Use exclusionary mappings in the repo's `p4gf_config` file to omit nonconvertible files from the clone request.
- Change the `charset` value of the repo's `p4gf_config` file to a value that enables all files to be converted.

For more information, see Repo configuration file: key definitions and samples.

Missing @repo section

During a clone, the following message appears:

```
No section: '@repo'
```

This error indicates either an undefined `@repo` section or a typographical error in the `@repo` section header of configuration file, like `@repos` instead of `@repo`. Review the configuration file and correct any errors; see Repo configuration file: key definitions and samples.

Spec depots cannot be mapped

During a clone, the following message appears:

```
spec depots cannot be mapped in client view fatal: The remote end hung up unexpectedly
```

The client view maps to at least one spec depot; however, spec depots are not supported in Git Fusion. Update the client view to remove all spec depots. You do not need to run the Git Fusion Delete Repo script (`p4gf_delete_repo.py`) because Git Fusion did not allow the clone to complete.

General usage issues

Cannot terminate active process

You issued a **Ctrl+C** or similar command to terminate an active process (like **git clone**) and the SSH connection, and the following message appears:

```
Processing will continue even if connection is closed.
```

During an active process, Git Fusion reaches a point where interruptions will cause data corruption in Perforce. Consequently, Git Fusion will continue processing even if you attempt to terminate it or the SSH connection.

Connection closed by remote host

The remote host abruptly closes the connection and displays the following message:

```
ssh_exchange_identification: Connection closed by remote host
```

This message indicates that the load on Git Fusion exceeds the `MaxStartups` value set in the `sshd_config` file. Adjust this setting to resolve this issue; we recommend a minimum value of at least 100.

```
MaxStartups 100
```

The `sshd_config` file is normally located in the directory `/etc/ssh/sshd_config`.

Case sensitivity conflicts

To resolve case sensitivity conflicts when you are operating Git Fusion in a mixed OS environment, do one or all of the following:

- Convert the Perforce server to run in case-sensitive mode.
- If users are on a case-insensitive platform like Windows, instruct them to configure their Git clients with the `core.ignorecase` option of the **`git configure`** command to prevent case inconsistencies.
- Configure Git Fusion with a preflight commit check to reject uppercase. See Adding preflight commits to reject pushes and `preflight-commit-require-case.py`

Resolving case sensitivity conflicts is beyond the scope of this documentation. Contact Technical Support at `support@perforce.com` for help.

Combining Git and a case-insensitive Perforce server can lead to trouble that is difficult to work around or repair. **You can end up with Git histories that can never be pushed into Perforce.**

Note	A Git user can create a history where two different files exist simultaneously, paths differing solely by case. Pushing that history into Perforce will either merge the two files into one, or fail.

Git is case-sensitive at its core, regardless of `core.ignorecase`.

- **`git log`** and other commands ignore `core.ignorecase`, remain case-senstive: **`git log FILE`** reports entirely different results than **`git log file`**.
- A single file in Perforce can split into two separate files in Git, path differing solely by case.

git-fast-import crash

When a `git-fast-import` crash occurs, Git Fusion records the resulting crash report to its logs as an error. To locate these errors, search the logs for the following text strings that appear at the beginning of the report:

`date/time stamp` `p4gf_fastimport ERROR git fast_import_crash_`*`process identifier value`*

`fast-import crash report:`

If a `git-import-crash` occurs during the initial clone of a repo, Git Fusion recovers from the import failure by deleting the entire working files directory for this repo. If the crash occurs during a push or pull request for an existing repo, Git Fusion deletes only the working files for the specific request. In either case, because Git Fusion automatically deletes its working files, you must redo the request.

Git Fusion submit triggers are not installed

The remote host abruptly closes the connection and displays the following message:

```
Git Fusion submit triggers are not installed.
```

This message indicates one or more errors in your implementation of submit triggers in the Perforce service for the Atomic Push feature, which is necessary for the correct operation of the branching functionality. Inform all Git users to discontinue their use of Git Fusion, and do the following:

- Review the installation procedure and verify that you have correctly implemented this functionality; see Step 4 on page 8 of installation steps.

- Run **`p4gf_submit_trigger.py --set-version-p4key`** ***`serverport`***to set the `P4PORT` value; see `p4gf_submit_trigger.py --help`.

headType field does not exist

The following error message appears:

`Field headType doesn't exist`.

This error may occur when a commit contains added and deleted files that have similar names.

Look for the file noted in the directory path after the following phrase, and revise the client view to exclude the file:

```
[P4#run] Errors during command execution ( "p4 fstat -TheadType path/.../filename")
```

Locked repo caused by active process termination

Issuing `Ctrl+C` or a similar command to abort an active client-side process (like a **`git clone`**) to the Git Fusion service may permanently disable further use of the repo; the following message (or similar) may appear:

```
# git-upload-pack: Perforce gi-copy-p2g failed. Stopping
```

```
fatal: The remote end hung up unexpectedly
```

Git Fusion may not always receive terminate signals that are run in a Git client. This has two consequences:

- Any active server-side Git processes may continue to run.
- The Git Fusion repo will become locked and unusable.

To correct this situation and unlock the repo, do the following:

1. **Wait one minute to allow Git Fusion time to delete the locked repo.**
2. **Delete the key that Perforce uses as a mutex. Git Fusion regenerates the key at the next Git command (clone, fetch, pull, or push) to this repo.**

   Any user that has at least Perforce review access permission can run the following command from a workstation:

   `p4 key -d git_fusion_` *`repo_name`* `_lock`

If (for any reason) you cannot delete the key, you can also correct this issue by doing the following:

1. **End the active server-side processes.**
2. **Delete the locked Git Fusion repo.**

   Run the Git Fusion Delete Repo script (`p4gf_delete_repo.py`) to delete the locked repo.

3. **Recreate the repo.**
4. **Inform users that they need to reclone the repo.**

We recommend that you delete the key, because this does not require users of the affected repo to reclone. See the *Perforce Command Reference*, `p4 key`, and `p4 protect` for `review` permission information.

Missing server-id file

The following message appears:

```
Git Fusion is missing server-id file /home/git/.git-fusion/server-id. Please contact your administrator.
```

This error indicates that the `server-id` file is missing or has been deleted. Run `p4gf_super_init.py` with the `--id` option to rebuild this file; see `p4gf_super_init.py --help`.

Unicode-enabled client required

The following message appears:

```
Unicode server permits only unicode enabled clients
```

This message indicates that the repo is interacting with a Unicode-enabled Perforce service but the Unicode setting in `.bashrc` or the repo configuration file is missing or invalid.

1. Verify that you have correctly set the `P4CHARSET` environment variable in your `.bashrc` file.

   For example, for a Perforce service using UTF-8, `.bashrc` should include the line `export P4CHARSET=utf8`.

   If you installed Git Fusion using the OVA, this setting should be in `~/.git-fusion-profile`. For more information about setting Git Fusion environment variables, see Installing Git Fusion using OS-Specific Packages or Installing Git Fusion using the OVA.

2. If the `P4CHARSET` environment variable is set correctly in your environment, check the `p4gf_config` file for the repo.

   The `charset` value should be set to the correct `P4CHARSET` for your Perforce service.

   If `charset` is incorrect, you must delete the existing repo and recreate it with the correct value. See Setting up Repos

Git Fusion OVA issues

OVF cannot be parsed

When attempting to convert the Git Fusion OVA image, the following message appears:

```
The OVF descriptor file could not be parsed.
```

You are trying to convert the OVA image with an unsupported virtual machine version. See the release notes for the supported versions of Oracle VM VirtualBox and VMWare.

P4D cannot be started

The Perforce service in the OVA does not start.

Run either of the following commands to reboot the Perforce service:

- `sudo service p4d start`
- `sudo /etc/init.d/p4d start`

Push issues

Files not in client view

When you or a Git user attempts to push a new file or a merge commit to Git Fusion, the push fails and the following error message appears:

```
File(s) not in client view.
```

This message appears for the following two scenarios:

- When the Git repo includes files that are not in the client view.
- When a Git work tree path notation does not match the notation of all other Git work tree paths in the repo configuration file.

To resolve this error, recreate the repo definition with a broader view, and ensure that the Git work tree paths in the `view` field are formatted correctly for all branch definitions. See Sample repo configuration files for examples of correct Git work tree path notation.

Files locked by git-fusion-reviews--non-gf

When a user attempts to push to Git Fusion or submit to Perforce, the push or submit fails and the following error message appears:

```
Files in the push are locked by [git-fusion-reviews--non-gf]
```

This indicates the Atomic Push functionality is active and preventing other users from performing an action that currently conflicts with another user's push. Instruct the user to wait a few minutes and attempt the push or submit again.

Merge commit requires rebasing

When a Git user attempts a push, the following message appears:

```
Merge commit <SHA1> not permitted. Rebase to create a linear history.
```

This indicates that you have implemented a Git Fusion repo without Perforce branching functionality, and that the user is attempting to push a commit that has non-linear history. You must do the following:

- Instruct the user to run `git rebase` and verify that the commit has a linear history with a single parent before attempting another push.
- Review the Git Fusion repo configuration files' definition, determine if you need to enable Perforce branch support, and contact your Perforce Technical Support representative at `support@perforce.com` for help with this conversion.

Not authorized for Git commit

When you or a Git user attempts a push, the following error message appears:

`user name not authorized to submit file(s) in git commit`

The Perforce service enforces read and write permissions on files. Review the **read** and **write** permissions you have assigned to the user and determine if there are any errors.

To resolve the user's issue, instruct the user to do the following

1. **Run `git filter-branch` to remove prohibited files from the local Git repo.**
2. **Attempt the push again.**

To minimize this issue, provide your users with a list of their specific directory and file permissions.

Not permitted to commit

When you or a Git user attempts a push, the following error message appears:

`User 'email_address' not not permitted to commit`

This indicates that the Git author does not map to any Perforce user. Git Fusion is unable to assign a changelist owner to the changelist for such a commit.

To allow this push to go through, either:

- See Mapping Git users to Perforce accounts.
- See Enable the `unknown_git` Perforce account.

Password invalid or unset

When you or a Git user attempts a push, the following error message appears:

`[Error]: Perforce password (P4PASSWD) invalid or unset`

`Please make sure you have the correct access rights and the repository exists.`

This indicates one of the following two scenarios:

- You have not set a password for `git-fusion-user` as required by your Perforce service.

  Set a password and repeat the login procedure.

- You have not logged Git Fusion into the Perforce service.

  To log the Git Fusion server into the Perforce service:

  1. **Log into the Git Fusion service account (`git`) on the Git Fusion server.**
  2. **Run `p4 login git-fusion-user`**
  3. **Run `p4 login git-fusion-reviews-`*`serverid`***

     The *`serverid`* is the Git Fusion server's ID. Git Fusion sets this ID when you run `p4gf_super_init.py` and also records the ID in `~git/.git-fusion/server-id`.

Pushes prohibited after repo deleted or trigger removed

After you delete a repo, remove a branch from a repo config file, or remove a trigger that affects a Git Fusion implementation, Git users cannot push commits.

To resolve this situation, run the Submit Trigger script `p4gf_submit_trigger.py` against the Perforce Server. This resets data and re-enables users to perform pushes. See `p4gf_submit_trigger.py --help`

Script issues

Updating authorized keys file of multiple servers fails

You run the `p4gf_auth_update_authorized_keys.py` on a series of Git Fusion servers, but some or most of the servers still do not have updated key information.

To successfully run this script on multiple Git Fusion servers, each server must have a unique server ID. Use the `p4gf_super_init.py` script with `--showids` to view existing IDs and `--id` to assign new IDs.

Appendix Authenticating Git Users using SSH

SSH uses public and private key pairs to perform authentication. Git Fusion provides a method for managing SSH keys, wherein each user's public key is versioned in the Perforce depot under `//.git-fusion/users/p4user/keys`. Users either send the Git Fusion administrator their public keys to submit to Perforce, or users submit them directly to the Perforce depot, depending on your organization's workflow preferences.

The Git Fusion Update Authorized Keys script (`p4gf_auth_update_authorized_keys.py`) copies the public keys from the Perforce depot to the Git Fusion server and performs the following tasks:

- Inserts a call to `p4gf_auth_server.py` in the key.

  When a user issues a Git command against the Git Fusion server, the embedded command in that user's public key invokes the `p4gf_auth_server.py` script, which authenticates the user and routes the request to Git Fusion.

- Writes the modified key to the Git Fusion service user account's authorized keys file (or SSH2 authorization files).

Git Fusion supports the following SSH implementations:

- OpenSSH and other SSH implementations that use the same public key and `~/.ssh/authorized_keys` format.
- SSH2 implementations that use RFC 4716-style SSH2 key files and `~/.ssh2/authorization` files.

  Git Fusion can work with SSH2 keys that have been converted from OpenSSH keys.

Set up SSH authentication

Note	You can use any SSH key management method that you like, as long as SSH keys are modified to call `p4gf_auth_server.py`. For more information about the `p4gf_auth_server.py` script, run **`p4gf_auth_server.py -h`** as the Git Fusion service account (`git`).

To manage SSH keys using the method provided by Git Fusion:

1. **Create a workspace (client) for submitting public keys to Perforce.**

   The workspace view should map `//.git-fusion/users/...` to your workspace root.

2. **Add users' public keys to Perforce.**

   a. Obtain the user's public key.

      Key files can have any name. Be sure to store only *public* keys in Perforce.

   b. Submit the user's public key to the `//.git-fusion` Perforce depot.

```
$ p4 -u user_name -c client add //.git-fusion/users/p4user/keys/keyname
$ p4 submit -d “add new keyname”
```

3. **Run the Git Fusion Update Authorized Keys script (`p4gf_auth_update_authorized_keys.py`).**

   You must run it as the Git Fusion service account (`git`).

   You can run the script manually, but it is better to use a `cron` job to run the script automatically. The `configure-git-fusion.sh` script creates this `cron` job for you. For more information, see Use a `cron` job to copy public keys to Git Fusion.

> **Note**
>
> If you want to let Git users administer their own keys in the Perforce service, you must give them write permissions on their `//.git-fusion/users/`*`p4user`*`/keys` directory.
>
> When you add a permissions row to the `p4 protect` form, enter the Git Fusion server's IP address as the `host` value. You can represent this IP address as an asterisk (*), unless you are using CIDR notation:
>
> ```
> $ p4 protect Protections:
> ...
>      write   user   p4joe   *   //.git-fusion/users/p4joe/keys/...
> ```

> **Note**
>
> Git Fusion supports multiple keys for the same user and stores them in the user's `keys` directory. If users are maintaining multiple keys, ensure that they do not store them in separate subdirectories for each key. These keys are shared across all Git Fusion instances.

Use a cron job to copy public keys to Git Fusion

Git Fusion uses the Git Fusion Update Authorized Keys script (`p4gf_auth_update_authorized_keys.py`) to identify new SSH keys in Perforce, modify them, and copy them to Git Fusion. The `configure-git-fusion.sh` script creates a `cron` job to run the script every minute, letting you avoid having to run the script manually every time a user adds or changes their public key.

The `configure-git-fusion.sh` script creates the `cron` job in `/etc/cron.d/perforce-git-fusion`.

You can modify this `cron` job to add `p4gf_auth_update_authorized_keys.py` script options, such as `--file` or `--ssh2`, as needed. For more information, see `p4gf_auth_update_authorized_keys.py --help`.

Set up SSH authentication using the OVA's SSH key management console

If you implement Git Fusion using the OVA, the OVA's SSH key management console simplifies the authentication setup process for you. When you upload a key using the online management console, Git Fusion automatically places the key in the correct directory and runs the Git Fusion Update Authorized Keys script (`p4gf_auth_update_authorized_keys.py`).

Note

The SSH key management console works out-of-the-box when you use the Perforce Server instance that was installed with the OVA. If you are connecting to an existing, external Perforce service from the Git Fusion OVA, you must provide your Perforce service's hostname and port (`$P4PORT`) on the **Git Fusion Config: Perforce Server** page.

Important

If you have assigned a password to `git-fusion-user`, you must update this password in the SSH key managment console before you can upload SSH keys:

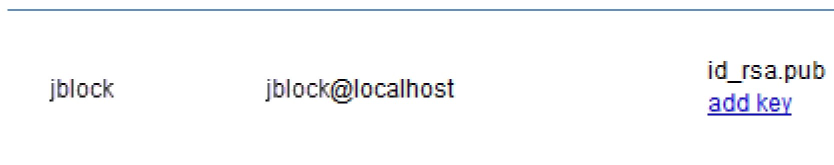
1. Go to the **Perforce Server** tab in the online management console.
2. Enter the password you set for `git-fusion-user` and click **Apply**.

To add a new public SSH key using the online management console:

1. On the Git Fusion online management console, go to the **Git Fusion Config** tab and click the **User Key Management** button to access the **Upload User Key File** page.
2. Enter a Perforce **User ID** and browse to select its public SSH key.
3. The **Authentication Required** window displays.

   Enter `root` and the password you established for `root`.
4. Click **Upload File**.

   This adds the key to the correct directory location and runs the Git Fusion Update Authorized Keys script (`p4gf_auth_update_authorized_keys.py`), which copies the key to enable the Git user's access to Git Fusion. On the **Upload User Key File** page, the Git user's information displays *without* the question mark icon and *with* an email account:

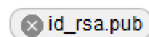
   

   Adding a Git user's SSH public key does not automatically add that user as a Perforce user. A Git user's name that displays *with* a question mark icon and *without* an email account does not yet exist in the Perforce service. You must create a Perforce user account for the Git user, then click the **Refresh** button on the **Upload User Key File** page.

To remove a key:

1. Mouse over the public key file.

   The **Delete** icon displays on the left side of the public key file name.

2. Click the **Delete** icon and click **OK**.

The following confirmation message appears: `File` *`filename`*`.pub was deleted.`

Troubleshooting SSH key issues

Key or identity not recognized

Git user's new or changed keys do not seem to be working, or when you use the identity file (`-i`) option with the **`ssh`** command, the following message appears:

```
Too few arguments.
```

To resolve the issue:

- Wait a few minutes for Git Fusion to update the authorized keys directory automatically.
- Run **`ssh-add -D`** to clear your computer's authentication cache and force SSH (or git-over-ssh) to honor your keys for the next command.

  The authentication software on your computer maintains a small cache of keys and identities for you. However, note that if you regularly switch between different SSH keys, the cache occasionally uses an older key for an SSH session even if you specify a different or newer key by running **`ssh -i`** ***`identity_file`***

No such Git repo

When a Git user connects to Git Fusion using SSH, it prompts the user for a password and then displays the following error message:

```
There's no such Git repository.
```

Logging into Git Fusion using SSH does not require a password. This issue usually indicates an error in the SSH key configuration, like Git Fusion having an incorrect key pair.

PTY request failed

The following error message appears:

```
PTY allocation request failed on channel 0.
```

You used an SSH key associated with Git Fusion when attempting to perform a non-Git operation such as SSH or SCP.

Keys checked into `//.git-fusion/users/`*`p4user`*`/keys/`*`filename`* are reserved solely for Git Fusion operations like **`git clone`** and **`git push`**. You cannot use these keys for SSH, SCP, or other operations.

Repo is not a Git repo

The following error message appears:

repo **does not appear to be a git repository.**

You used an SSH key that is not associated with Git Fusion when attempting to perform a Git operation such as **`git clone`** or **`git push`**.

SSH format issues

If you encounter key issues, verify that the key is in a supported format and stored in the correct directory. Git Fusion supports keys formatted and stored as follows:

- OpenSSH, stored in `~/.ssh`
- SSH2, stored in `~/.ssh2`

Appendix License Statements

Perforce software includes software developed by the University of California, Berkeley and its contributors. This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (`http://www.openssl.org/`).

Perforce software includes software from the Apache ZooKeeper project, developed by the Apache Software Foundation and its contributors. (`http://zookeeper.apache.org/`)