HelixCore

Solutions Overview: Helix Version Control System

2017.1

May 2017

PERFORCE

www.perforce.com

Contents

How to Use this Guide

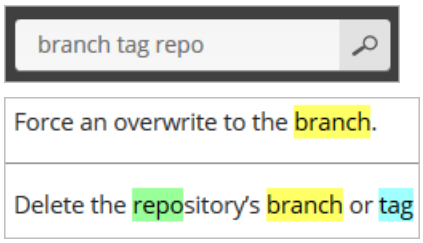
This manual introduces Helix, a secure, scalable, and highly available version control system that supports parallel development. You should read this document before you start working with Helix.

This document:

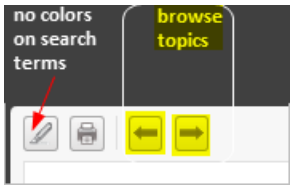
- introduces the basic concepts and tasks of version control
- explains how you can configure Helix to improve performance and to scale the system
- suggests some ways you can extend and customize Helix
- explains how you can use Helix with other products to get additional functionality
- discusses use cases for Helix
- introduces resources that can help you use Helix

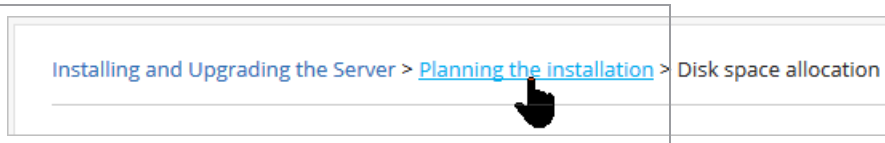
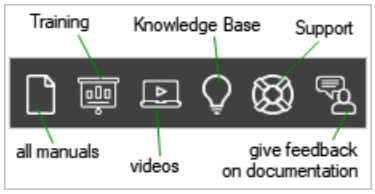
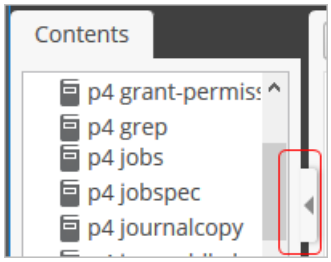
If you are familiar with version control systems, you can skip ahead to "Helix as a version control implementation" on page 9.

Search

Use quotes for an exact phrase:	
To search within a page that is long, use Command-F (Mac) or CTRL+F (PC)	
If you enter multiple search terms: the hits show a different color for each term:	

Navigation

You can remove the colors, as well as browse from topic to topic with the arrow buttons:	

The top of each page indicates where the topic is:	
The bottom of every page has links to resources:	
You can use links using standard URLs, such as `https://www.perforce.com/perforce/doc.current/manuals/cmdref/#CmdRef/p4_add.html#Examples` even if you see extra characters after that.	
You can resize the Content pane	

Feedback

How can we improve this manual? Email us at manual@perforce.com.

Other Helix documentation

See https://www.perforce.com/support/self-service-resources/documentation.

Syntax conventions

Helix documentation uses the following syntax conventions to describe command line syntax.

Notation	Meaning
`literal`	Must be used in the command exactly as shown.
italics	A parameter for which you must supply specific information. For example, for a *serverid* parameter, supply the ID of the server.
[`-f`]	The enclosed elements are optional. Omit the brackets when you compose the command.

Notation	Meaning
...	■ Repeats as much as needed: • `alias-name[[$(arg1)... [$(argn)]]=transformation` ■ Recursive for all directory levels: • `clone perforce:1666 //depot/main/p4... ~/local-repos/main` • `p4 repos -e //gra.../rep...`
element1 | *element2*	Either *element1* or *element2* is required.

The Basics of Version Control

When you work alone on a document, the latest is usually the greatest: you successively open the document, make changes, and save the document. Each time you save, you overwrite the existing copy. The situation is different when you are working with a large, globally distributed team on a project consisting of hundreds, or even thousands, of files. In this case, it is important to track authorship and changes, and to resolve conflicts when users make conflicting changes to the same file. Version control systems allow you to do this. You can track and manage changes to any large collection of digital assets: documents, source code, web sites, audio files, and so on.

One technique of version control systems is versioning. Rather than overwriting earlier versions of a file when it is saved, each saved copy of the file is versioned and assigned a number or letter that reflects the order in which it was saved.

In addition to identifying a file version within a sequence of versions, a version control system automatically associates certain information with each version: it records who made the change, when the change was made, and why the change was made. This information provides an audit trail that you can always consult to understand how a project developed and when specific changes were made. Because no version of a file is overwritten, when bugs arise, it is possible to identify the point at which the bug was introduced. This can be critical in fixing bugs that cannot be reproduced. In addition, looking at file history and understanding why certain decisions were made can help project participants stay on track or find appropriate options for future directions.

Sharing data under version control requires a certain amount of gatekeeping to determine who can access the data and how conflicts are resolved when two users make changes to the same file. To support this gatekeeping function, version control systems introduce the additional step of checking out a file and checking in or submitting a file.

The basic version control workflow looks like this:

1. Assets under version control are placed in a specified repository.
2. Assets are associated with specific permissions that enable users to read or modify them.
3. A user checks out a working copy of an asset and makes changes.
4. Another user checks out a working copy of the same asset and makes changes.
5. The first user saves changes to the local working copy and checks in that copy.
6. The second user saves changes to the local working copy and attempts to check in that copy.
7. The version control system detects the fact that the same asset was changed in parallel, and it asks the second user to merge changes with those of the first user before the second user's changes can be checked in. The work of comparing and merging changes is called *resolving*.

In this way, the version control system makes sure that changes are predictable, manageable, and auditable.

Version control systems are traditionally either centralized or distributed:

- *Centralized version control systems* use a single repository from which users check out one or more files to work on locally.

- *Distributed version control systems* allow users to host repositories locally, check out entire repositories with all history—or, in the case of Helix, a subset of repositories—work independently of one another, and combine their work through merging when necessary.

Helix supports either model, as well as a hybrid of the two.

Version control systems can be used as stand-alone applications or they can be incorporated into development or authoring tools as a means of managing the assets produced by these tools.

Helix as a version control implementation

Helix uses a client-server architecture to implement version control management.

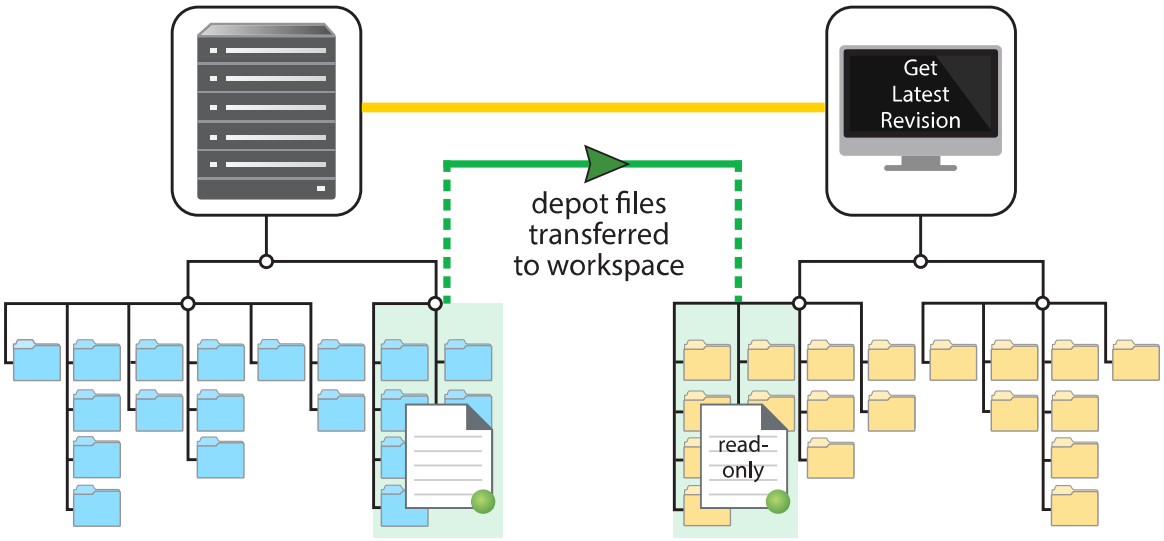
- The Helix*server* (also known as the Helix Core or `p4d`) manages shared file repositories, or depots, that contain every revision of every file under version management. Files are organized into directory trees. The server also maintains a database to track data associated with files and client activity: logs, user permissions, metadata, configuration values, and so on.
- Helix*clients* provide an interface that allows you to check files in and out of the depot, resolve conflicts, track change requests, and more.

  Helix clients include a command-line client, a graphical user interface client, and various plugins that work with commercial IDEs and productivity software. A Helix server can provide services to a mix of Helix clients.

  You also use Helix clients to manage a special area of your computer called a workspace. Directories in the depot are mapped to directories in your workspace, which contain local copies of managed files. You always work on managed files in your workspace:

  1. You check the files out of the depot (and into your workspace).
  2. You modify the files.
  3. You check them back into the depot.
  4. If the changes you try to submit conflict with changes that other users, working in parallel with you, have already submitted, you must resolve conflicts as needed.

The next figure shows the mapping between depot files (shown on the left) and workspace files (shown on the right). Until files are checked out from the depot, they remain as read-only in the workspace. To have Helix update your workspace so that it reflects current work on the depot, synchronize your workspace to the depot by getting the latest revision of the files.



We have explained about checking files in and out of the depot, suggesting that single files may be checked in and out. In fact, the means we use to check files in and out of the depot is the *changelist*. A changelist must contain at least one file and may contain tens of thousands. A changelist is numbered and allows you to track all changes with respect to the contents of the depot: file modifications, the addition of a file, or the deletion of a file.

The changelist is the simplest way to organize your work. A changelists also represents the atomic unit of work in Helix: if a changelist includes several files, changes for all the files are committed to the depot or none of the changes are. For example, if a network connection between the client and the server fails during changelist submission, the entire submit fails.

Multiple user access to a set of files

Version control systems must address the fundamental need for multiple users to work on the same project simultaneously. Helix offers two ways to do this:

- File locking: Helix locks a file while someone is working in it. This controls access to the file: if several users want to edit the same file, it is possible to merge changes into one mutually acceptable version.
- Branching and merging: By branching streams and then merging them later, multiple users can work on the same files simultaneously. See "Balancing stability and innovation: the mainline model" below.

Balancing stability and innovation: the mainline model

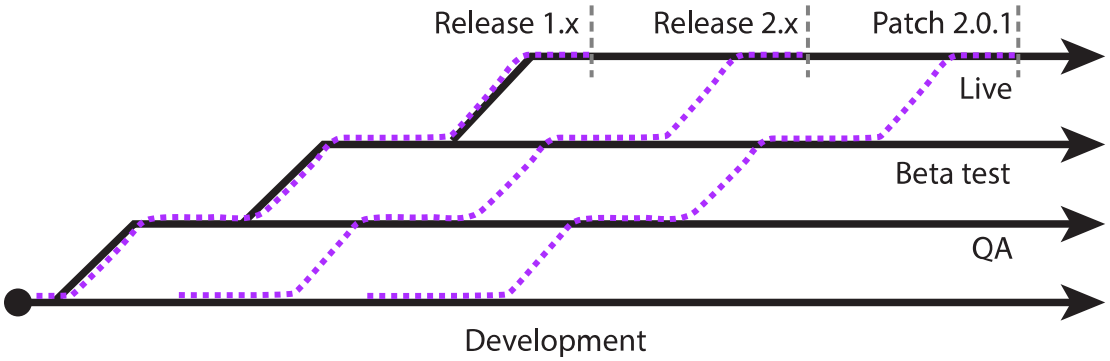
So far, we have explained how version control systems handle the problem of different users working on the same file at the same time: when the different copies of the file are checked in, all but the first user must resolve changes by deciding which changes will be preserved in the latest version of the file.

This is the simplest use case for parallel development. More complicated cases arise when large development projects require many people to work in parallel both because they must support multiple releases and because they involve multiple functional teams working together to create the desired product.

For example, a game development company depends on the combined efforts of artists, musicians, programmers, testers, and build engineers to create a release. One way to organize this effort is to split the set of files that make up the project into multiple parallel branches, allowing development and testing to occur along each branch. Integration then occurs across branches to promote everyone's work into a releasable product.

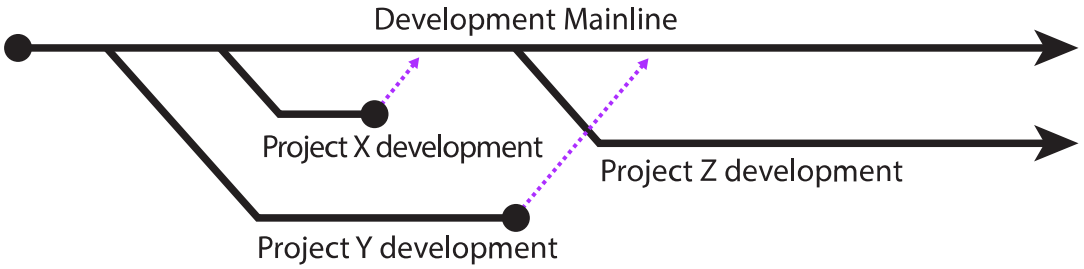
Consider the following cases:

- Extremely short release cycles, such as occur with fast changing web content, require overlapping cycles of testing and release.



  To handle this case, we move code through the branches shown above. Development occurs along the development line. When a milestone is reached, code is copied up to the QA line where it is thoroughly tested. After it clears all tests, it is copied up to the Beta test line where it is subjected to real-world use. Having satisfied Beta users, it can then be copied up to a release line. Note the purple dashed lines, which indicate the flow of code as it is copied up through the branch hierarchy.

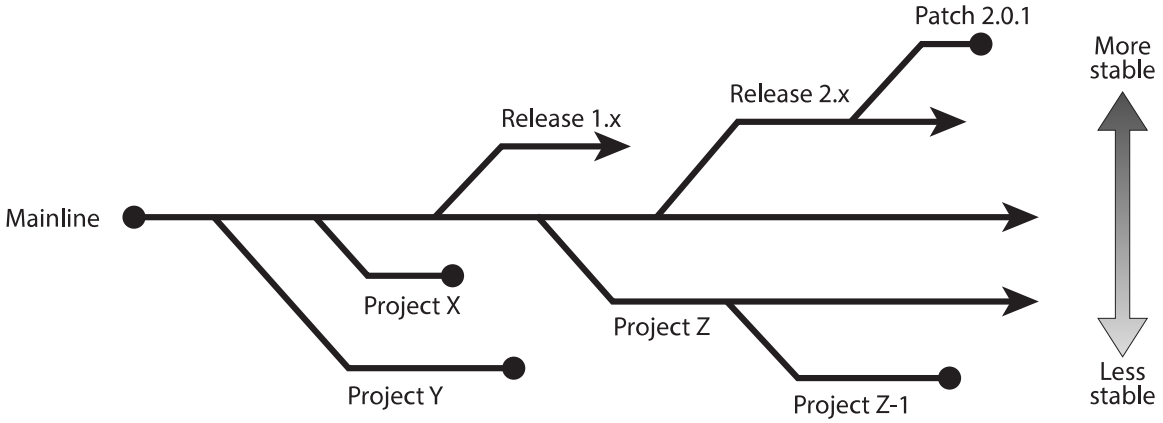
- Unforeseen delays cause some features under development to miss deadlines. Not wanting to imperil the project as a whole, the project is released while development continues for the laggard components.



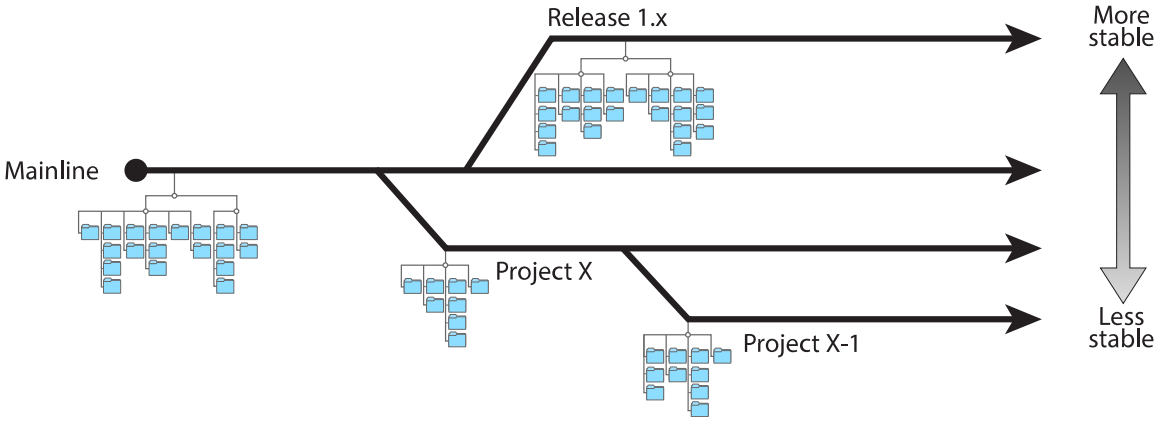
  In the model shown above, projects X, Y, and Z have been branched off from the development mainline and worked on independently. When work has completed and projects X and Y have passed development tests, they are copied up to the mainline. Project Z however could not be completed and work continues on that project without putting the larger release at risk.

What does branching have to offer? It allows us to balance the need for stability with the need for innovation. On the one hand, we have release branches that hold the most tested and stable code; on the other hand, we have development branches that allow for experimentation and exploration without putting the release at risk.

Branches can be organized in a variety of ways: you can create branches for different platforms, you can create branches along organizational lines, and so on. One common model used for product development is the Mainline model shown below:



The Mainline forms the trunk from which release branches and development branches are created. Each branch normally contains a subset of the Mainline files. Release branches might contain fewer files because files needed for testing are excluded; development branches might contain a different subset of files because the projects they represent focus on discrete product features. In the Mainline model, the "up" direction indicates increased stability or confidence.

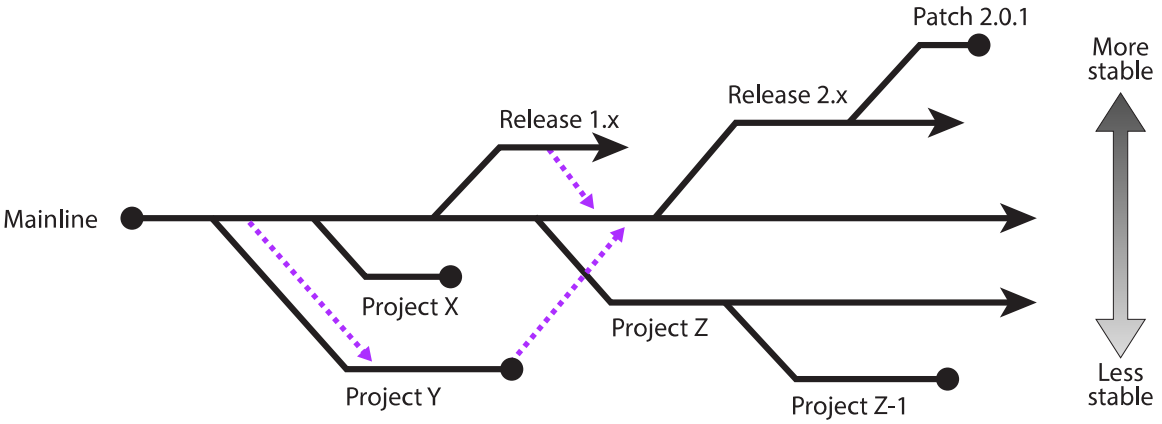


When you create branches, you are free to integrate changes in any direction you like. Unfortunately, this can lead to big problems if you inadvertently integrate untested changes into an otherwise stable branch. For this reason, in addition to defining branches that isolate changes, the Mainline model is most useful when it can implement some protocols that limit what changes can be made and in what direction.

Streams

Helix streams implement the Mainline model, adding intelligence that determines what changes can be made and in what order they must be made.

Let's look at the Mainline example again and add some information to indicate flow of change:

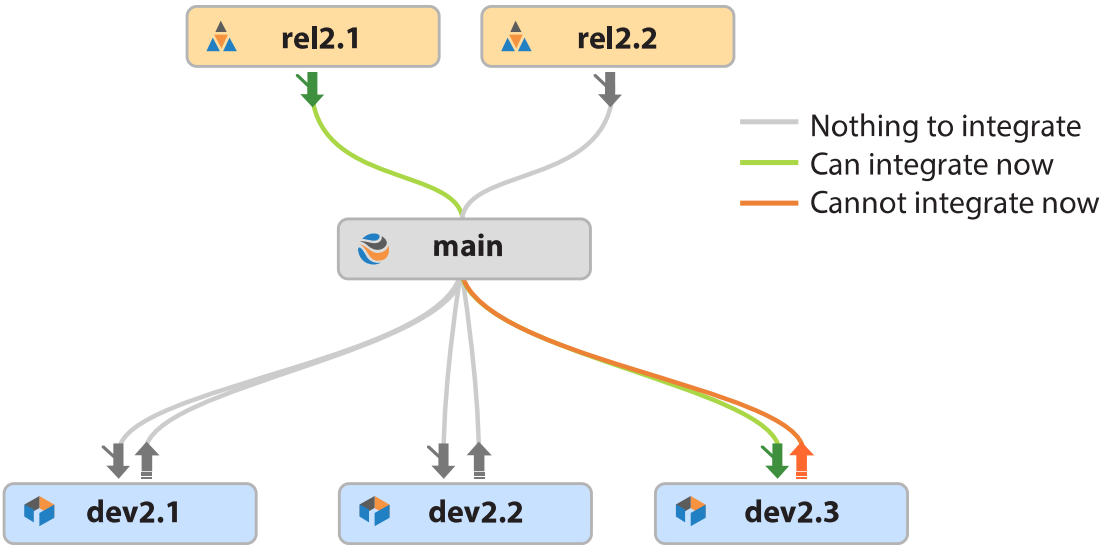


- Project Y has been branched from Mainline; work and testing continues until it is complete. It is then ready to be copied back up to the mainline. However, while development has taken place in Y, Mainline has continued to change. Before we can copy the contents up to the Mainline, we need to make sure that Project Y files reflect changes that have been made in Mainline; we merge those changes into Project Y before we copy Project Y files to Mainline.
- A bug is found in Release 1.x. The bug is fixed and tested. We now want the bug fix in Mainline, so we merge files from Release 1.x down to Mainline. We do not copy anything up because Release 1.x should not include any features added after it has been branched.

The Mainline model arranges branches in terms of stability: the most stable branches are at the top; the least stable branches are at the bottom. The flow of change needs to support this model by merging changes down and copying up.

Typically, when you work with streams, you define and populate the mainline first. You then create development streams and release streams as children of the Mainline stream. The type of a stream and its relationship to other streams determines what sort of changes can be made and in what order they are made.

Rather than using a timeline, the streams GUI—found in the Perforce Visual Client (P4V)--represents related streams as shown below:



The children of Main are shown both above and below Main. Release type streams are at the top; development and task streams are at the bottom. Stability grows as streams near the top of the diagram. The direction and color of arrows linking streams indicate both the direction of flow and the order of flow.

When you create a stream, you specify its type, its relationship to other streams, and how files are to be treated for merging and branching. The information you provide is then used by the streams application to encourage good behavior.

Streams provide visual clues for where and how to branch and merge. They guide behavior that supports stability and innovation. Using streams eliminates much of the work needed to define branches, to create workspaces, and to manage integrations.

An additional advantage of using streams is that when you switch from one stream to another, the contents of your workspace are updated automatically to reflect the contents of the current stream.

Streams automate branching, but you do not have to use them. You can create your own branches and manage them as you see fit. Custom branching gives you finer grained control but you lose the convenience of built-in flow control and workspace updating.

For information on streams, see the *Helix Versioning Engine User Guide*.

Organizing your work: jobs and labels

In addition to using changelists and streams to organize your work, you can use two other methods: jobs and labels.

- *Jobs* provide lightweight issue tracking that integrates well with third party defect tracking and workflow systems. They allow you to track the status of a bug or an enhancement request. Jobs have a status and a creator and are associated with changelists that implement the bug fix or the enhancement. An administrator can customize the type of information tracked by jobs add more fine grained status values, or define additional fields for information to be tracked: which customer

the enhancement is for; what was done to test the fix, and so on.

You can integrate the jobs function with third-party defect tracking and workflow systems. For more information, see the *Defect Tracking Gateway* page.

- *Labels* are sets of tagged file revisions that allow you to handle a heterogeneous group of files as one unit. While a changelist refers only to the contents of a given set of files at the time they were submitted, a label can refer to a group of file revisions from different points in time. You might want to use labels to define the group of files contained in a particular release, to sync a set of files, to populate a workspace, or to specify a set of file revisions to be branched. You can also use a label as an alias for a changelist number, which makes it easier to remember the changelist and easier to refer to it in issuing commands.

For information about jobs and labels from a user's perspective, see the *Helix Versioning Engine User Guide*.

For information about managing jobs and labels, see the *Helix Versioning Engine Administrator Guide: Fundamentals*.

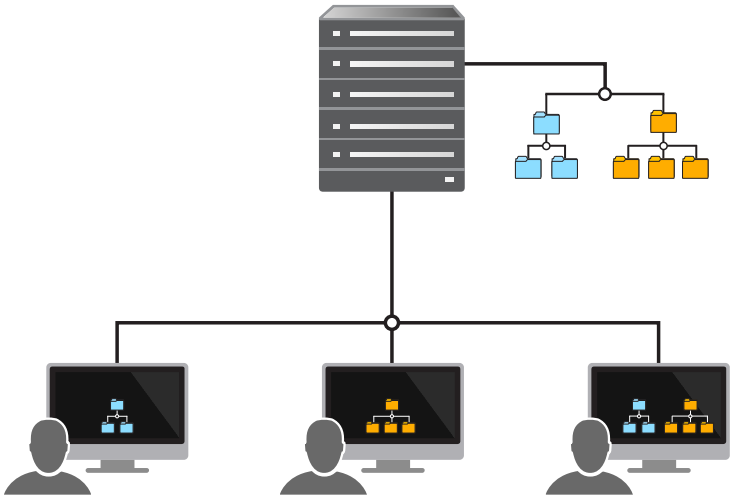
Working together and working apart: centralized and distributed development

We mentioned earlier that version control systems can implement either a centralized model or a distributed model:

- *Centralized version control systems* use a single repository from which users check out one or more files to work on in their local directories.
- *Distributed version control systems* allow users to host repositories locally, check out entire repositories with history—or, in the case of Helix, a subset of repositories—work independently of one another, and combine their work through merging when necessary.

Helix supports either model, as well as a hybrid of the two.

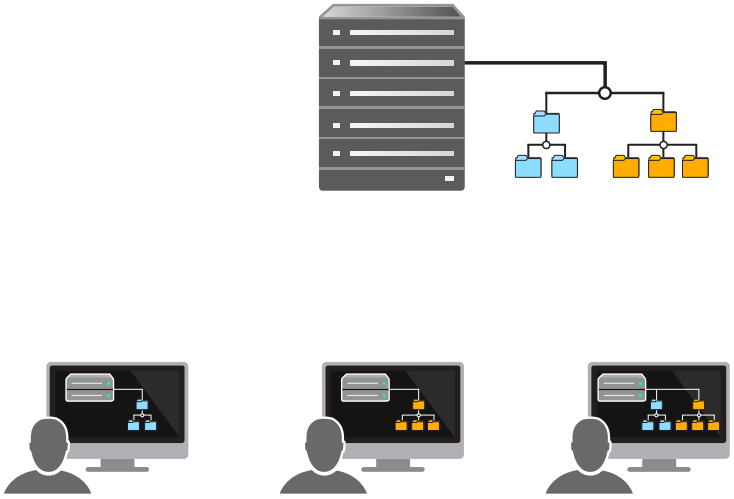
In the centralized model, clients work with a depot on a shared server. A mapping of files from the depot to their workspace determines which files they are able to work with in their workspace. The following figure illustrates the centralized model:



Users check files out of the same depot, work on them, and check in their changed files. If multiple users work on the same file, they use merging and conflict resolution to make sure the resulting version is satisfactory to all authors. Although users can disconnect from the shared server and continue to work on the files in their workspace, some manual work is required to sync back to the server and to check in files when the user reconnects. For information on working with Helix using this model, see the *Helix Versioning Engine User Guide*.

In the distributed model, users work with a depot on personal servers that are then connected to a shared server. The depot on their personal server might contain a subset or the entire set of files on the shared server. Each user can work disconnected from the shared server but still be able to access all the files in their workspace and place these under version control using their personal server. Each user can access the entire history of a file locally, can rewrite and revise history, and can manage the files and streams on their local machine without interacting with the shared server.

The following figure illustrates this model:



When users decide to share their code or digital assets with other users, they connect to and then push their content to the shared server. This allows other users to fetch that content from the shared server and work with it on their own personal server. Users might need to merge content before pushing if their changes conflict with changes made by others.

In addition to supporting these two models, Helix also allows for a hybrid architecture in which some users connect directly to the shared server while others connect to personal servers that are connected to the shared server.

For more information about distributed development and file management, see *Using Helix Core for Distributed Versioning*.

Performance, scaling, and high availability

Version control systems are key to managing large projects: with Helix, “large” can be large indeed. With enterprise-level features that you can use to fine tune and improve performance, Helix lets you scale your system to accommodate a global workforce, and to automate failover for a highly available system. For example, Helix can accommodate the needs of a gaming development company whose files might take up hundreds of terabytes or even petabytes of data; or it can support the work of a software company, whose activity level includes massive automated testing as well as focused, analytic bug fixing and tracking work.

To support these tasks, Helix uses the following additional server types:

- *Proxies* are used where bandwidth to remote sites is limited; they mediate between remote clients and the versioning service. By caching frequently used files, the proxy reduces demand on the server and keeps network traffic to a minimum.
- *Brokers* mediate between clients and servers to implement policies that solve routing or security problems.
- *Replicas* duplicate server data. They can be used to provide a warm standby server or to reduce load on a primary server.

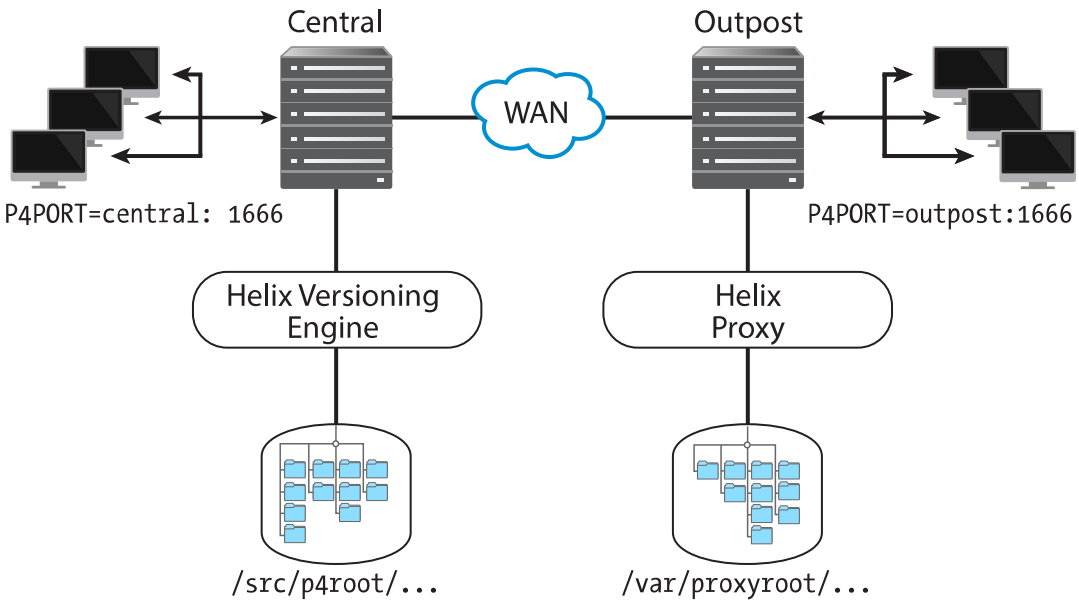
The following sections explain how these servers are used singly or how they are combined to provide enterprise-level performance. For complete information about using proxies, brokers, and replicas, see *Helix Versioning Engine Administrator Guide: Multi-Site Deployment*.

See also "Git at scale" on page 24.

Using proxies to improve performance

To improve performance for users accessing a shared Helix repository across a WAN, you can configure a proxy on the side of the network close to the users and configure the users to access the service through the proxy; then configure the proxy to access the master Helix service.

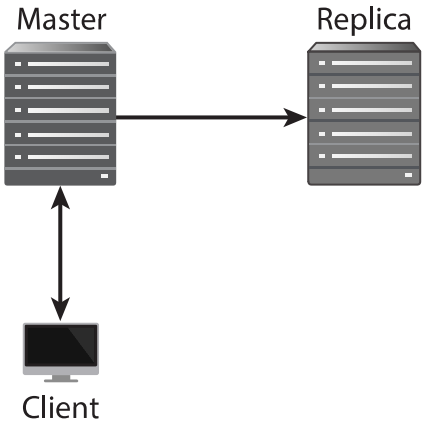
The following diagram illustrates a typical proxy configuration:



In this configuration, file revisions requested by users at a remote development site are fetched first from a shared server (`p4d` running on Central) and transferred over a relatively slow WAN. Subsequent requests for that same revision, however, are delivered from the Helix Proxy, (`p4p` running on Outpost), over the remote development site's LAN; this architecture reduces both network traffic across the WAN and CPU load on the shared server.

Using a replica for disaster recovery

Replication is the duplication of server data from one Helix server to another. The replicated server is called the master server; its replica is called a replica server. You can use replication to provide a warm standby server, or to reduce load and downtime on a primary server when performing builds. The following figure shows how you set up a replica to provide a warm standby to aid in disaster recovery.



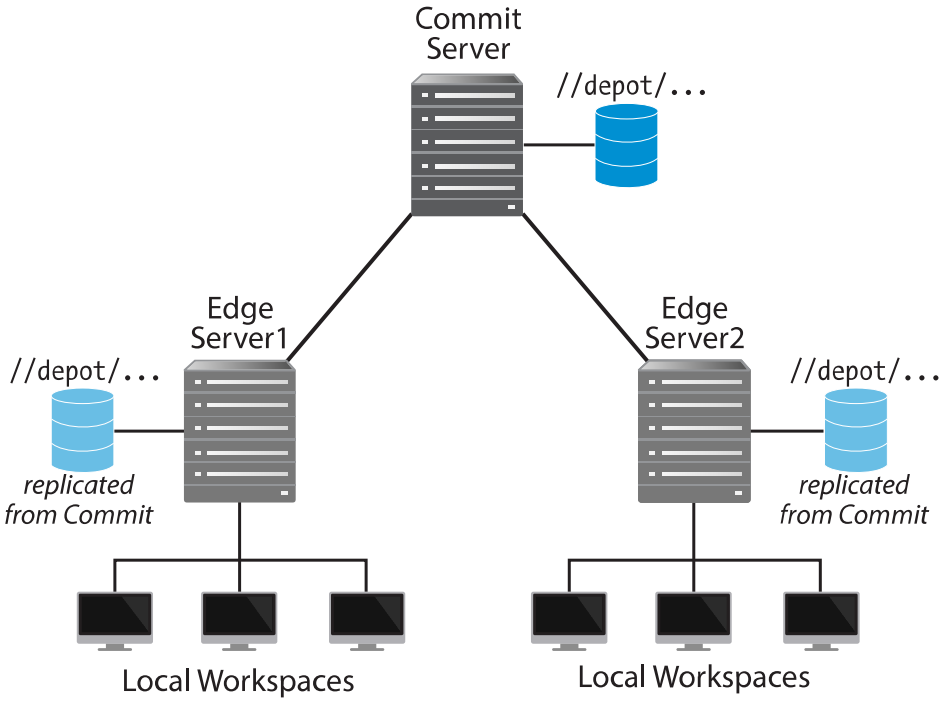
When you create the replica, you specify which server it should get its data from. The replica then periodically updates itself by copying files and metadata from the master. If the master fails, all you need do is reconfigure the replica to be the new master and then reconnect clients to communicate with it.

Edge servers and workspace servers, described in the following sections, are special examples of replica servers.

Commit-edge architecture

This architecture supports optimal performance for geographically distributed work groups. At a minimum it is made up of the following kinds of servers:

- A *commit* server that stores archives and metadata.
- An *edge* server that contains a replicated copy of the commit server data and a unique, local copy of some workspace and work-in-progress information. This server can handle read-only operations and operations that only write to the local data. You can connect multiple edge servers to a commit server as shown in the next figure.



Since an edge server can handle most routine operations locally, the edge-commit architecture offloads a significant amount of processing work from the commit server and reduces data transmission between commit and edge servers. This greatly improves performance.

For more information about these options, see *Helix Versioning Engine Administrator Guide: Multi-Site Deployment*.

Securing the system

Independent of the Helix architecture you use, secure communication is guaranteed both with respect to communication between clients and servers as well as communication between servers.

- User authentications can be done using passwords or tickets, and the strength of the password can be defined by an administrator. Users can be authenticated against an Active Directory or LDAP server, or against an internal Helix user database.
- Communication between clients and servers can be secured using the SSL protocol, which you specify when connecting to the server.
- Communication between servers in a distributed environment can be secured using a trust file and by setting permissions for the service users that own the different servers in the environment.

In addition to user authentication, digital assets are further secured by a protection scheme to prevent unauthorized access. Protections determine which Helix commands can be run, on which files, by whom, and from which host. This scheme provides the finest grained control possible.

For more information on security, see the *Helix Versioning Engine Administrator Guide: Fundamentals* and the *Helix Versioning Engine Administrator Guide: Multi-Site Deployment*.

Clients, IDE's, builds

Helix products also include tools and software packages that allow you to work with and analyze your data as well as seamlessly integrate Helix into interactive development environments and build solutions.

Helix client applications include:

- A command line interface available for all platforms
- A GUI interface for Mac OS X, UNIX, Linux, and Windows
- Integrations, or plug-ins, that work with commercial IDEs and productivity software

Interactive development environment integrations include:

- The P4VS plugin embeds the power of Perforce Helix features into the Microsoft Visual Studio IDE.
- The P4Eclipse plugin integrates the strengths of Perforce Helix with Eclipse's powerful IDE.
- P4Connect, the Perforce Helix Plugin for Unity, enables you to perform Perforce Helix operations directly from within Unity.

  You also have access to many community-built integration plugins, which are summarized on the Perforce website, on the Helix Clients & Integrations page.

Build and reporting integrations include:

- The Jenkins integration allows Jenkins users to use Helix repositories directly. The plugin makes it simple to pull code, automate reviews, apply labels, and so on. Helix4Git can be part of a environment that uses Jenkins.
- The P42DB integration replicates Helix metadata to open source and commercial SQL databases.

Customizing and Extending Helix

APIs

- Helix Core C/C++ API
- Helix Core API for .NET
- Helix Core API for Java
- Helix Core API for Perl
- Helix Core API for Ruby
- Helix Core API for Python
- Helix Core API for PHP

Triggers Triggers are user-written programs or scripts that are called by a Helix Core when certain operations are performed, such as changelist submissions, changes to forms, or attempts by users to log in or change passwords. If the script returns a value of `0`, the operation continues. If the script returns any other value, the operation fails. For example:

- a pre-submit trigger to ensure that whenever `file1` is submitted in a changelist, `file2` is also submitted.
- form triggers generate a customized default workspace view when users run the `p4 client` command
- a `graph-push-complete` trigger can fire when mirroring into Helix from a third-party Git server. (See "Git at scale" on page 24)

See *Helix Versioning Engine Administrator Guide: Fundamentals* .

Helix Broker The Helix Broker enables you to implement local policies in your Helix environment by allowing you to restrict the commands that can be executed, or redirect specific commands to an alternate (replica or edge) Helix Core.

The Helix Broker is a server process that mediates between Helix client applications and Helix services, including proxy servers. For example, Helix client applications can connect to a proxy server that connects to the broker, which then connects to a Helix Core. Or, Helix client applications can connect to a broker configured to redirect reporting-related commands to a read-only replica server, while passing other commands through to a master server.

See *Helix Versioning Engine Administrator Guide: Multi-Site Deployment* .

Use Cases

The Ace Rocket company develops software to manage the production and distribution of their space rockets and rocket launchers. They have a variety of technical needs, all of which can be met by members of the Helix product family. The following sections describe these needs and the Helix products that address them.

Software development

Ace Rocket uses a variety of programming languages to develop the components and products that they build. Because they use open source projects, they have numerous developers that use Git. For their internal development teams, they prefer the flexibility of Helix Core. Using Helix4Git, Ace Rocket can track their Git commits in the Helix Core. This makes the Helix Core a single source of truth for all of their assets and provides an unbroken history.

Ace's software developers use both Swarm to collaborate and communicate about the code they write. Ace Rocket's Windows developers take advantage of the native Visual Studio integration (P4VS), which makes it easy for them to kick off code reviews from within their IDE. Ace's Java developers get the full power of Helix inside of their IDE of choice: Eclipse.

Using the Jenkins plugin, the Ace Rocket testing team continually tests code as it is checked into a single place by both P4 and Git developers.

Digital asset management

The Ace Rocket Company tracks design versions of their products using Helix. They track versions of their CAD/CAM files as well as large test datasets. Ace takes advantage of Helix's file locking to ensure that multiple people do not attempt to change a model at the same time.

Using configurable storage, Ace Rocket chooses to limit its storage consumption to only 20 versions of each file. Using Helix's multi-site deployment architecture, Ace seamlessly replicates these large files across the globe to improve access times for their engineers.

Hybrid product development

With both source code and CAD/CAM files tracked in Helix, Ace Rocket can easily track dependencies between their code and their models. Developers take advantage of Helix's fluid distributed workflows, while Ace's model engineers use locking to obtain the tight control they need to collaborate with each other.

Ace Rocket's IT team takes advantage of the numerous Helix APIs to build custom tools that meet their unique needs and to integrate Helix into all of their systems. In addition, they enjoy the benefits of Helix's native Active Directory support to make it easy to manage all of their users.

Another type of hybrid product development is combining Helix4Git with classic Helix. See Helix4Git Administration.

Ease of use for Helix administration

Ace Rocket employs an outside agency for contractors. They want to give these contractors access to only a certain subset of the depot, while they want to give in-house developers full access. Because Helix Core permissions are highly configurable, it's easy for an administrator to use finely grained access control to manage access for different types of users.

Growing into the future

As Ace Rocket grows and develops offices in Shanghai, Buenos Aires, and Johannesburg, they deploy edge servers and replicas to support thousands of developers in these remote offices. Both developers and digital asset managers in remote offices benefit from having local access to their software or assets.

Helix is keeping one unified view, worldwide, of digital assets content and history so the distributed teams can collectively work as one.

Git at scale

Ace Rocket benefits from having scalable Git environments in **one** place. Helix4Git, with its Git Connector component, enables Ace Rocket to have rapid mirroring of Git repos into Helix, and supports real-time visibility into Ace Rocket's Continuous Integration (CI) activities with an automated build tool, such as Jenkins.

Support for Global Teams

- Ace Rocket enjoys good performance and usability across distributed geographic locations.
- Ace Rocket's enterprise teams have fast and stable development environments, wherever they are in the world.

Reduced Overhead and Tooling

- No danger of repo sprawl.
- Scalability for large digital assets, large numbers of files, and large total repo sizes are supported with streamlined repository management.

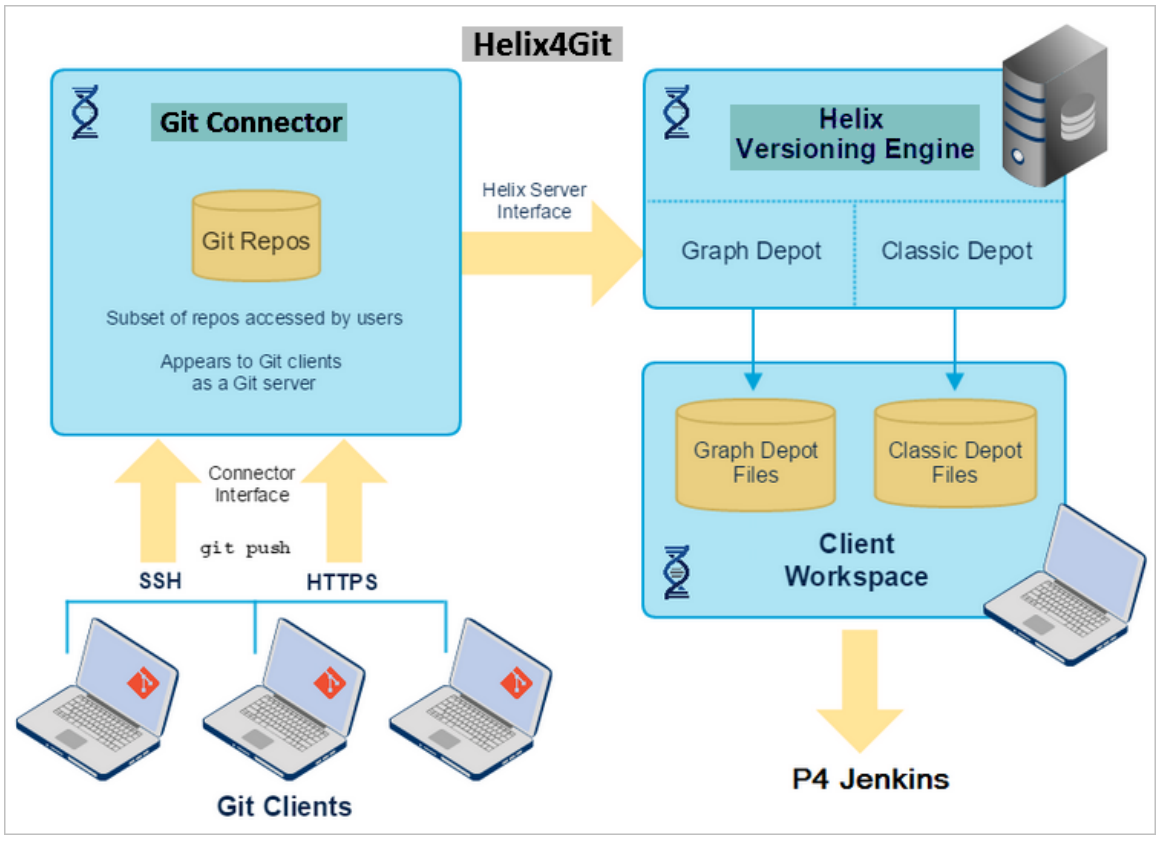
Centrally Manage All Digital Assets

- The graph depot accepts repos from multiple sources.
- Users can centrally manage their assets with multi-repo support and traceability.

Git Continuous Integration and Delivery

- Automatic mirroring into Helix from third-party Git servers, such as GitHub, GitLab, and Gerrit, help Ace Rocket achieve rapid release iterations and development benchmarks.
- Helix4Git boosts performance with simplified builds of multi-repo, multi-asset projects on top of a federated architecture.

For more information, see https://www.perforce.com/solutions/git-scale.



To learn more about Helix

To ...	See ...
Download Helix products	http://www.perforce.com/downloads
View tutorials explaining how Helix products work	http://www.perforce.com/resources/tutorials
Get online help from within Helix applications	■ Use the help menu from within graphical Perforce applications ■ Type `p4 help` from the command line for help with the Command-Line Client
Access the Helix Knowledge Base, which has Support articles	http://answers.perforce.com
Access the Helix Forums where users ask and answer questions	http://forums.perforce.com/
Request Support by phone or email	https://www.perforce.com/support

This list is a subset of the documentation for users, developers, and administrators at: https://www.perforce.com/documentation:

For information about …	See …
Perforce's code review and code collaboration tool	*Helix Swarm Guide*
Administering the Helix versioning engine to support git repos and LFS files	*Helix4Git Administration*
Using the command-line interface to perform software version management and codeline management Working with Helix streams; jobs, reporting, and scripting	*Helix Versioning Engine User Guide*
Workflows using P4V, the cross-platform Helix desktop client	*P4V User Guide*
Working with personal and shared servers Understanding the distributed versioning features of the Helix Versioning engine	*Using Helix Core for Distributed Versioning*
p4 command line	*P4 Command Reference*, `p4 help`
Installing and administering the Helix Core, including user management, security settings	*Helix Versioning Engine Administrator Guide: Fundamentals*

For information about …	See …
Installing and configuring Helix servers in a distributed environment: ■ proxies ■ replicas ■ edge servers	*Helix Versioning Engine Administrator Guide: Multi-Site Deployment*
Helix plug-ins and integrations	IDEs: *Using IDE Plug-ins* Defect trackers: *Defect Tracking Gateway Guide* Others: online help from the Helix menu or www.perforce.com
Developing custom Helix applications using the Helix C/C++ API	*C/C++ API User Guide*
Working with Helix in Ruby, Perl, Python, and PHP	*APIs for Scripting*

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/) Perforce software includes software developed by the OpenLDAP Foundation (http://www.openldap.org/). Perforce software includes software developed Computing Services at Carnegie Mellon University: Cyrus SASL (http://www.cmu.edu/computing/).