PERFORCE

WHITE PAPER

How to Automate Your Branching Strategy

Perforce Streams Adoption Guide

Introduction

Branching and merging are how code changes get coordinated in a version control system. Development teams working on the same branch need to coordinate efforts to avoid duplication and overriding other team members' changes.

Developers should merge their code changes early and often to ensure the shared branch stays up to date. Code can then be integrated frequently for builds and tests, leading to increased velocity and higher-quality releases. But as development products grow and become more complex, so do the number of branches teams need to manage.

Contents

Parallel development can get complicated at scale. To support multiple teams, releases, components and/or a large number of users, enterprises need to implement development processes. Teams could be divided up by programming languages, function (frontend and backend), and/or features, for example. The branching strategy coordinates how changes are made, promoted, tested, and eventually released across teams.

For many, branching, merging, and tagging procedures are not automated. If not automated, there are going to be challenges. Review this white paper to discover how you can automate your branching strategy to help teams move fast.

Branching and Merging Challenges

Developers want to spend time innovating, not figuring out where to branch and merge. Without automation, resources are spent managing:

- Relationships Between Branches
- Branching Documentation
- Complicated Scripts

RELATIONSHIPS BETWEEN BRANCHES

As developers branch to make changes, they are usually diverging code from a shared branch. For most version control systems, once a branch is created, there is no systematically defined relationship to the originating branch. Developers need to remember what branch they need to merge their changes back into.

But this is tricky at scale. Developers could be working on several branches throughout the day. It's hard to remember how branches are related. Even if developers do merge their changes back into the right branch, there is a risk they do not have all of the latest changes, causing another conflict. Instead of moving onto the next task, time needs to be spent reviewing merge conflicts or worse yet, backing out their changes.

BRANCHING DOCUMENTATION

Branching strategies only work if they are communicated effectively. They need to be concise. Otherwise, it can be too difficult for developers to follow, leading to a need to focus on activities other than writing new code and thus decreasing overall team velocity.

For some teams, strategies are outlined using an external tool. They may be written down on a whiteboard, wiki, or confluence page. Or they exist in the head of the most senior member of the team. Over the years, this documentation is rarely updated, or even worse, it is lost.

When a new team member joins, they need to figure out how to not break everything. Relying on a physical whiteboard or external tool to track how code should flow leads to more errors and confused developers. Some may even delay merging to avoid such conflicts. But late-stage integrations can cause time-zapping regressions that could even delay a release.

COMPLICATED SCRIPTS

What many companies do to force relationships between branches is write scripts on top of their version control system. Anytime scripts are written on top of a product, it locks developers into a very specific pattern. It's difficult to maintain over time and also makes it challenging to upgrade.

These scripts would need to handle hundreds to thousands of branches for a large enterprise-scale project. They can quickly become out of date. Scripts are costly to maintain. Companies need to hire people to just manage those scripts. If an admin leaves the company on short notice, their knowledge goes with them.

Enterprises end up sacrificing speed in order to support their development process. But there is a better way. The right version control system can automate your process, keeping velocity high and teams productive.

Automate With Perforce Streams

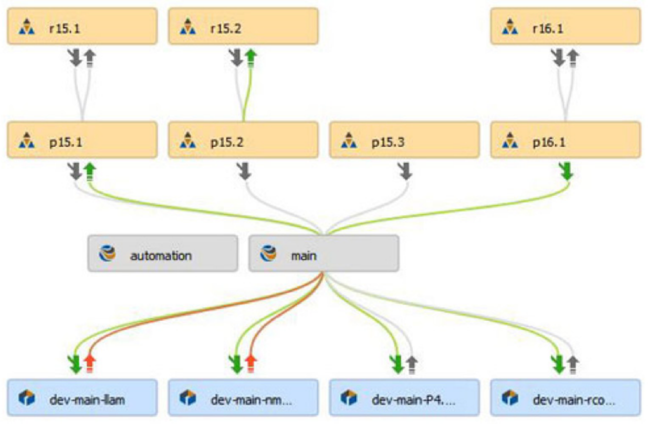
Perforce Streams — the branching mechanism in Helix Core version control — solves development challenges. It can…

- Automate any development process.
- Implement flexible workflows.
- Maintain control.
- Simplify complex environments.

HOW STREAMS WORKS

Regardless of how complex your process is, developers can easily get started with Streams without needing to understand all the details. All they need is the name of the stream to work against. Then they use either Helix Visual Client (P4V) or Helix Command-Line Client (P4) and get to work.

Streams tracks the relationship between the originating stream/branch (the parent) and the new child stream/branch. It's easy to visualize the relationship between streams in the Stream Graph.



Stream Graph in Helix Visual Client (P4V).

It's easy to understand the flow of change. Different colored arrows on the Stream Graph indicate new changes that need to be addressed:

- Grey = no merge or copy required.
- Green = a merge or copy operation is available.
- Orange = stream must be updated, after which merge or copy is available.

Streams prevents developers from copying up their changes until they have merged down all necessary updates. This built-in best practice encourages developers to merge early and often. Plus, it separates mature code from immature code, keeping your codebase stable. Because Streams tracks relationships, merges will always go to the correct branch.

Streams also works with Helix Core features such as exclusive checkouts. Teams won't waste hours or days on duplicated efforts. When it comes to the build process, Streams integrates with any preferred build runner to automatically test. With Federated Architecture, Helix Core servers can be set up around the globe. This ensure that team members get their feedback and files fast, no matter where they're located.

With Streams, your teams no longer need a white-board or heavy scripts. Developers can merge with confidence and avoid time-zapping merges and late-stage regressions.

How to Use Perforce Streams

There are several use cases where Streams can help automate development. Examples include:

- Component-Based Development
- Development Pipeline/Maturity Model/Shift Left
- Multiple Teams, Parallel Releases, and Multiple Variants

Component-Based Development

Component-Based Development (CBD) breaks up products into reusable components. This can include shared/common code, microservices, or a producer consumer model.

CBD approaches development like an assembly line filled with building blocks. Each block is worked on independently. These blocks are put together to create a larger block. Then those larger blocks are put together to create systems.

COMPONENT-BASED DEVELOPMENT CHALLENGES

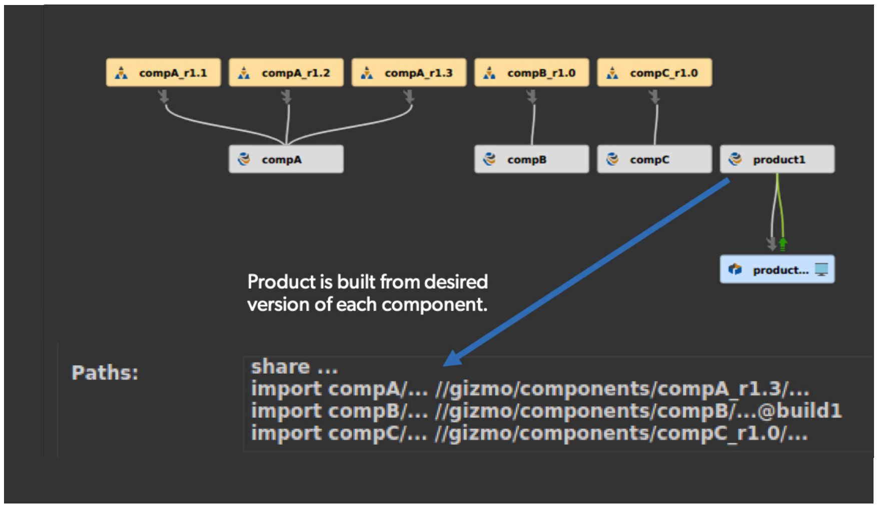
Each component needs to fit together. If a project has ten components, it's pretty simple to track how things are interrelated. But as projects scale to hundreds or potentially thousands of components, it can be almost impossible to track dependencies. To avoid time-consuming testing across components, developers may not merge changes as frequently. But late-stage integrations just cause more conflicts.

Teams may also have multiple different sub-components and products using different versions of a common library. Once a defect is found, propagating fixes to larger building blocks (potentially across versions) is a monumental task. This sort of dependency chain management is a tremendously painful task if done manually. Scripting layers can only do so much. They need to be updated to account for each new branch, release, and defect.

The point of component-based development is to be able to reuse components. But if developers are spending more time managing these relationships, teams may not see the benefits. Automation allows teams to just work on code while decreasing the overall complexity of managing dependencies.

SIMPLIFY COMPONENT-BASED DEVELOPMENT WITH STREAMS

Using Streams for component-based development simplifies and encourages reuse. All components are managed independently and graphically. Teams can version product configurations just like code.



```
share ...
import compA/... //gizmo/components/compA_r1.3/...
import compB/... //gizmo/components/compB/...@build1
import compC/... //gizmo/components/compC_r1.0/...
```

Component-based development Stream Graph.

In this example, product1 combines three different components — component A version 1.3 and version 1.0 of component B and C. The stream path indicates what components, and their versions, are included in the product release.

Let's say there is a defect in component C and a new version is released. The import path can be automatically updated to reflect the change, no external scripts required. Now product version 1.1 will include the new version.

Because product configurations can be versioned, teams do not need to remember what products are dependent on specific components. This gives teams ultimate traceability. Using Streams helps teams reuse code more efficiently, eliminate regressions, and easily track down defects.

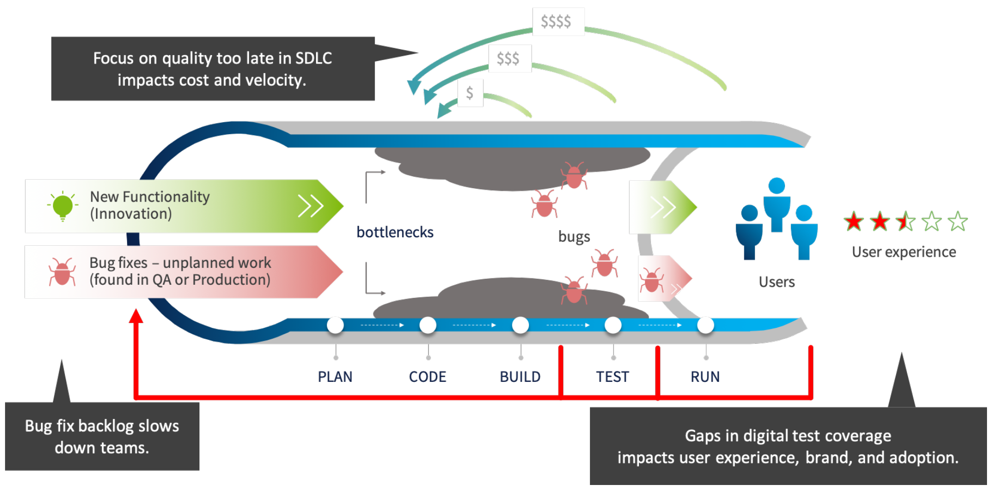
Development Pipeline/ Maturity Model/Shift Left

The later a defect is found, the more expensive it is to fix. To help remove quality issues in production, teams may look to evolve their development pipeline to address gaps in testing. Implementing a development pipeline maturity model — also referred to as shift-left testing — requires code to be integrated earlier in the process.

EARLY TESTING CHALLENGES

To implement a development pipeline, teams need to address testing at every stage. In the beginning, most developers work in isolation. There is little visibility into what everyone is working on and how it will eventually come together. Test coverage is therefore limited.

Teams might implement a pre-commit trigger as a best practice. Therefore, when changes are committed, a build is initiated. Developers might receive feedback in as little as a few minutes. If the build is successful, the changes are passed along. With each new build, test coverage grows. As a change moves through the pipeline, it is combined with changes across multiple teams.


Focus on quality too late in SDLC impacts cost and velocity.
$$$$
$$$
$
New Functionality (Innovation)
Bug fixes – unplanned work (found in QA or Production)
bottlenecks
bugs
Users
User experience
PLAN
CODE
BUILD
TEST
RUN
Bug fix backlog slows down teams.
Gaps in digital test coverage impacts user experience, brand, and adoption.

Development pipeline without Streams.

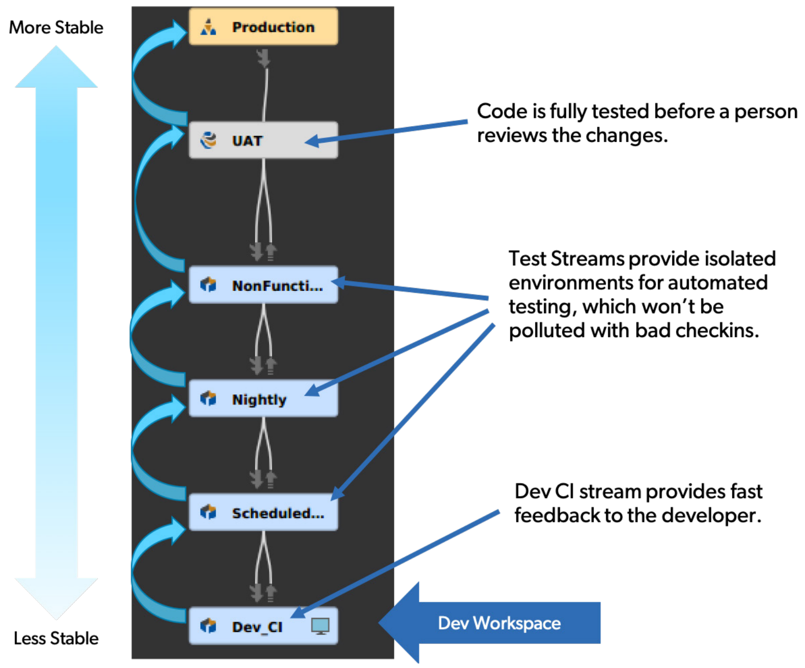
This model can create gaps in test coverage that can reveal bugs in QA or production. It slows teams down. Now they need to put new functionality on hold to address quality concerns.

If teams are working off a mainline, a developer could commit and push code that has only been tested on their own machine. Now immature changes are mixed with mature code. With most version control systems, it can be incredibly difficult to detangle.

Automating your testing environments allows you to save the most expensive resource — humans working on code — to deal with more high-value tasks. Building a solid pipeline should focus on getting people feedback fast. By the time a change enters the codebase and is handed off to QA, it should be well-tested and work well with other changes. This keeps your pipeline moving as fast as possible.

SHIFT LEFT WITH STREAMS

Streams supports your development pipeline by giving teams visibility. By automating each testing cycle unit – static, continuous integration, regression, etc. — Streams separates immature changes from your mature codebase. This ensures your mainline does not get polluted by bad check-ins.


More Stable
Less Stable
Production
UAT
NonFuncti...
Nightly
Scheduled...
Dev_CI
Code is fully tested before a person reviews the changes.
Test Streams provide isolated environments for automated testing, which won't be polluted with bad checkins.
Dev CI stream provides fast feedback to the developer.
Dev Workspace

Test Streams integrations.

In the Stream Graph, developers can see how their code moves through the pipeline. They can check out a stream, make changes, and test in isolation. When changes have passed an isolated build, code can be checked in. Now other team members can see the code and work from the latest build. Because the code has already been tested, it minimizes the risk that a build will break. Now you can test specific components, see how code evolves, and trace issues.

Streams removes potential bottlenecks in testing, increasing coverage with every step the chain. Developers get faster feedback. And by the time changes get to production, teams can be confident they are releasing only the highest-quality code.

Multiple Teams, Parallel Releases, and Multiple Variants

As complex projects scale, development can expand to include multiple teams, parallel releases, and/ or multiple variants. Organizing and orchestrating changes proposes several challenges for teams.

- Multiple Teams
- Parallel Releases
- Multiple Variants

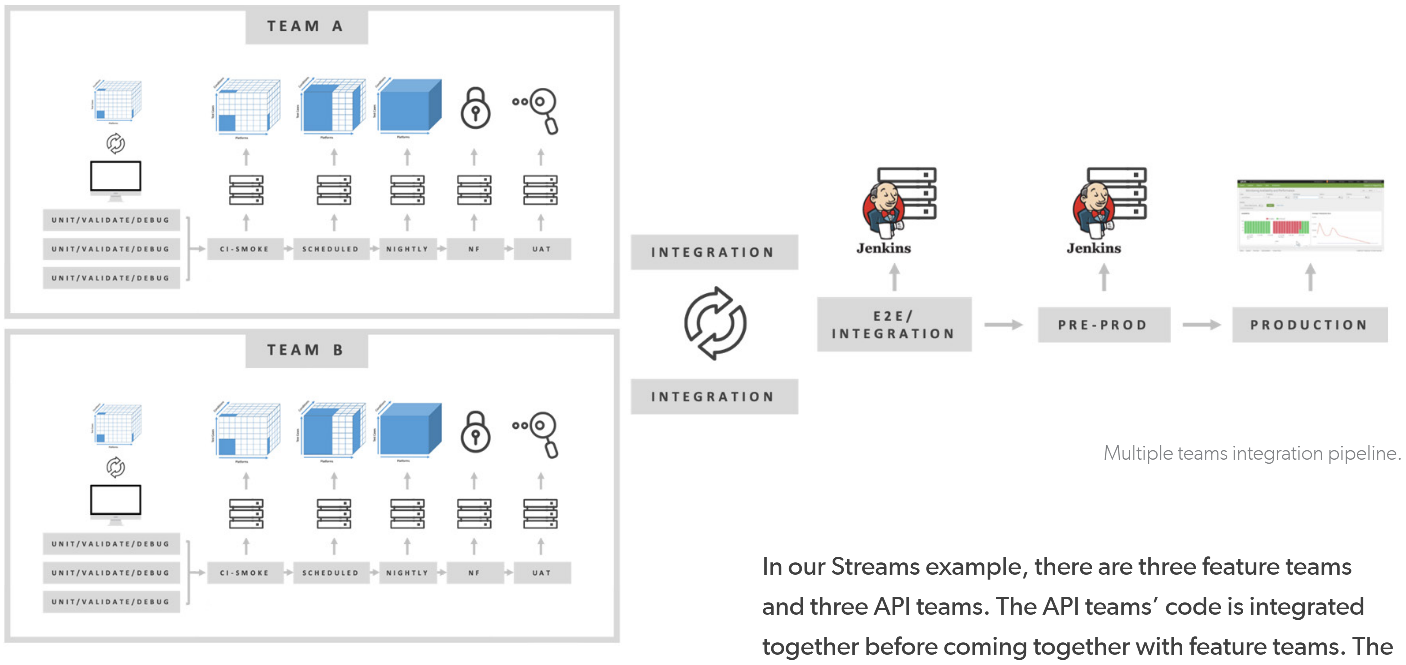
MULTIPLE TEAMS

Enterprises with large, complex products often break code into multiple component or feature teams in an attempt to move faster. For example, they may be split up by front end and backend. Depending on scope, a product may also require a separate API team or another working on a database layer. Communication within a team can be tough. Now it needs to work across teams.

Challenges With Multiple Teams

Although code is interdependent, developers don't know how their changes are connected. Teams may defer integration and testing as it is too difficult. Waiting until the end to combine everything can cause delays in production, massive rework, and quality issues.

Much like "merge early, merge often" is a best practice within a team, "integrate early, integrate often" is a best practice across teams. Tools and scripts struggle to accommodate ever-growing scale. Late integration can cause release tail. Without a single unified platform to incorporate all the changes, the developer feedback loop can quickly lose speed.

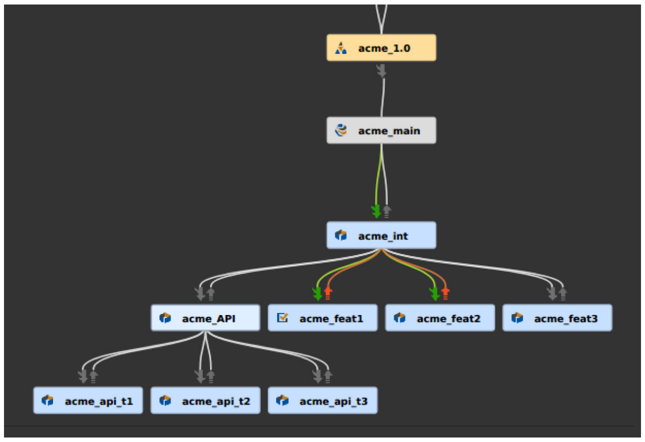


Multiple teams integration pipeline.

For example, Team A is writing a new feature based on a handful of rest API calls. But Team B changed that rest API call for something else. Now when the code comes together, it's broken. This integration was delayed due to complexity, so Team A has already moved on to the next feature. It's taken weeks to discover everything is broken, and now there is a new feature! Teams now need to halt their work to identify the root causes of the failure. Automating integrations eliminates this issue.

Multiple Teams + Streams

Integration across teams is easier with Streams. It's because Streams guides developers and increases visibility into the entire development process.



Stream Graph to integrate code across teams.

In our Streams example, there are three feature teams and three API teams. The API teams' code is integrated together before coming together with feature teams. The acme_int stream's colors on the Stream Graph indicate there some changes in the acme_int stream that need to be merged down into the acme_feat1 and acme_feat2 streams. These streams also have changes that can be copied up after the merge.

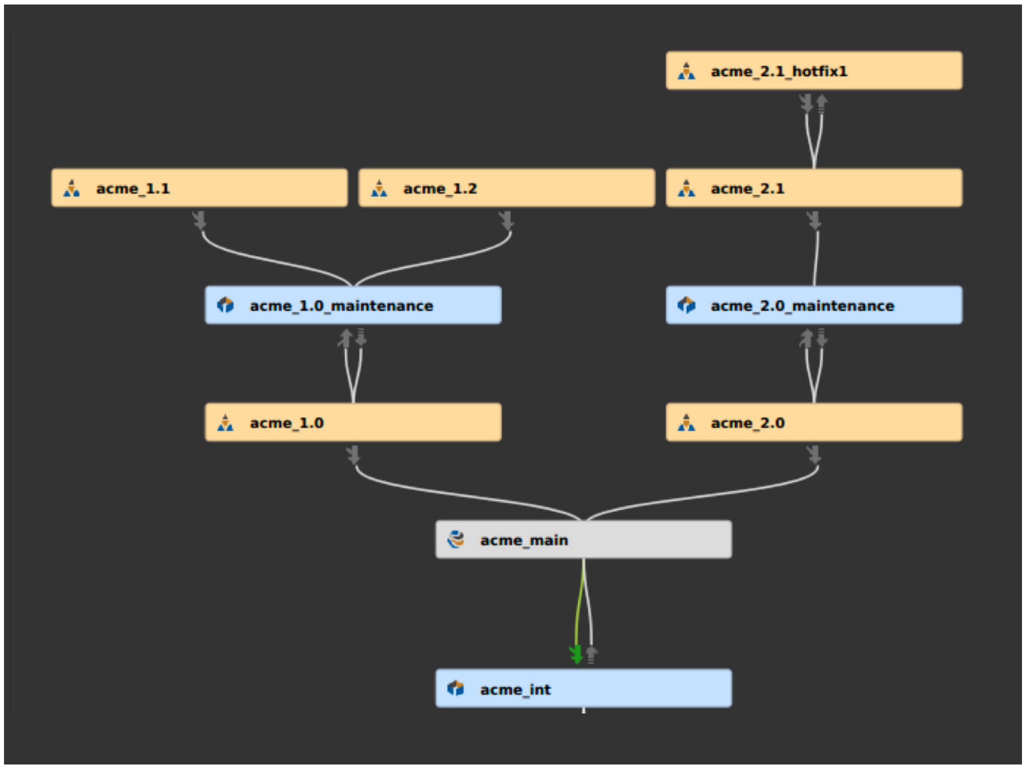
Developers always know they have the most recent changes before merging, encouraging them to integrate early and often. Because each step is automated, there are no bottlenecks integrating code across teams.

PARALLEL RELEASES

Supporting multiple product releases requires a lot out of teams. If a product has four major and eight minor releases a year, and supports releases over the previous three years, this means teams need to support 36 releases.

Challenges With Parallel Releases

Patching defects across multiple releases is very difficult. Usually, the person in charge needs to determine where to patch, and which releases and branches are impacted. Often, it's a manual process that offers
zero traceability. This can cause a lot of problems for products that need to meet regulatory standards.



Multiple release and maintenance streams in Stream Graph.

MULTIPLE VARIANTS

Companies often need to produce slightly different versions of a given product. Most of the code stays the same, but each variant differs. Managing these slight variations can be difficult, especially at scale.

Challenges With Multiple Variants

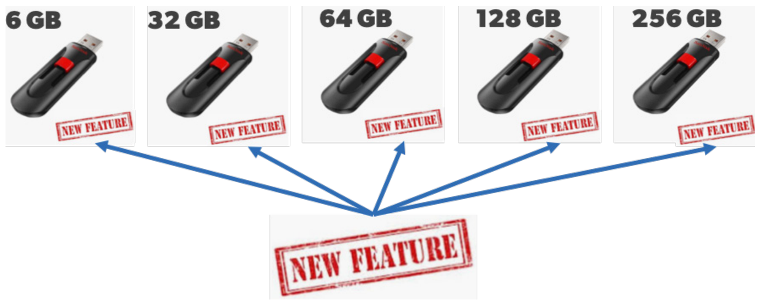
When managing multiple variants, fixes or new feature code must be propagated across products. This could potentially involve hundreds of variants. Each must be maintained and tested every time a change is made to the codebase.

Most of the time, teams just need to hope a patch is merged correctly. Hope is never a good strategy.

Parallel Releases + Streams

Streams allows you to visualize all your releases. It helps teams more easily propagate bug fixes and maintain higher-quality code.

At the bottom of the Stream Graph, there is a stream for integrations that is tied to main. The main stream has acme release 1.0 and 2.0. These release steams here are supported by maintenance streams. Using the graph, you can see how hot fixes would be propagated down into a release. This process happens automatically. The system itself knows exactly how these branches are related to each other and how the flow of change should go. Using Streams helps you maintain these releases with higher quality.



New feature being applied to several USB drives.

Propagating changes across variants has the same problems as supporting multiple parallel releases. It's hard to determine where the changes need to go. Manual processes are required to perform and verify each merge.

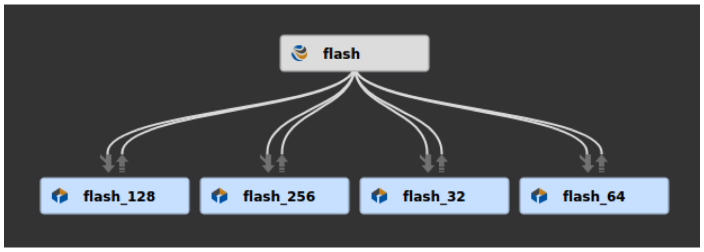
For example, a new feature may need to be pushed to several different hardware options. The base model contains 90% of your code. Each variant also has some code specific to the hardware.

If the wrong variant is installed in a piece of hardware, it can cause massive losses. There could be inconsistencies across the variants. Maybe one piece of hardware works as expected, but the others have problems. If this piece of hardware ends up in an embedded or automotive device, the need to trace changes becomes critical. It is a tremendous challenge to keep up with. Manual processes increase this risk of costly errors.

Multiple Variants + Streams

Using the same hardware example, the common stream contains the code that applies to all different variants. But then there are child streams for each individual variant. If a change is made to the shared parent stream, it is going to light up and tell each of the individual variants there are available changes. Then these changes can be merged down and tested.

If there is an issue with a variant, for example flash_128, then it can be handled in isolation from the other variants. The Stream Graph is built based on how you set up and define relationships between each variant. Developers can understand how code flows, giving you consistency across products.



Stream Graph to management multiple variants.

No matter your environment, Streams can help you:

- **AUTOMATE** any development process
- Implement **FLEXIBLE** workflows
- Maintain **CONTROL**
- **SIMPLIFY** complex environments

How to Migrate to Perforce Streams

Ready to migrate? For most, Perforce Streams will be easy and straightforward to use. However, Streams is a departure from classic branching strategies.

Start by bringing your team together — development leads, developers, architects, QA people, and build engineers. Discuss what your process would look like if there were no restrictions or challenges.

- What could you build?
- How could you handle these different processes?

Once you outlined your flow chart, open up the Stream Graph and build it. **You can build anything**. Customize to your optimal environment and see how easy it is to get everybody to collaborate.

If you want to change it — click, drop, and drag another stream. It is that easy to refine your process. No scripts. As teams add branches and merge, the graph automatically updates. You no longer need to rely on one person to manually update a static deliverable outlining your development process. Streams is flexible enough to adapt to how teams actually work.

MIGRATING YOUR MONOLITH

If you're looking to breakup your monolith, Streams allows you to slowly transition. Helix Core can handle your entire SVN or ClearCase monolith "as-is" without requiring major refactoring first. Because Helix Core is built to scale, it can support all your legacy code, while improving your overall performance and quality.

Over time, you can start moving projects into components. This allows you to break up your monolith structure on your timeline. Because Helix Core integrates with a variety of plugins and CI/CD tools, you can get immediate benefits as you transition.

STREAMS BEST PRACTICES

Here are some things to keep in mind as you are setting up your project.

1. Consider which types of streams to use.
2. Review the flow of change, which is suggested by the stream type based on best practices.
   - Most development streams allow bidirectional flow of change. Release streams usually do not accept changes from the parent (mainline).
3. Choose stream names carefully.
   - Stream metadata captures the most important information about a stream and organizes the structure. Formalize a naming convention and communicate it to teams.
4. Determine the implications of parent-child relationships between Streams.
   - Consider how you perform code and configuration merges between streams.
5. Capture any relevant information about stream composition in the stream view.
   - This information can help determine how changes are inherited.

Try Helix Core + Streams Free

You can get started for free for up to five users and 0 workspaces.

TRY TODAY

perforce.com/products/helix-core/free-version-control

START USING STREAMS

What you need to set up streams.

GET STARTED

perforce.com/products/helix-core/set-up-streams

WANT HELP?

We're here to support your teams.

CONTACT US

perforce.com/contact-us

About Perforce

Perforce powers innovation at unrivaled scale. With a portfolio of scalable DevOps solutions, we help modern enterprises overcome complex product development challenges by improving productivity, visibility, and security throughout the product lifecycle. Our portfolio includes solutions for Agile planning & ALM, API management, automated mobile & web testing, embeddable analytics, open source support, repository management, static code analysis, version control, IP lifecycle management, and more. With over 20,000 customers, Perforce is trusted by the world's leading brands to drive their business-critical technology development. For more information, visit www.perforce.com.