WHITEPAPER

Perforce to GitLab: A Strategic Migration Blueprint for Modern DevOps Teams

VIVAOPS

Contents

Executive Summary

Perforce has long been the version control system of choice for managing complex, large-scale codebases in industries such as gaming, finance, and aerospace. However, as development teams become more distributed and the pace of software delivery increases, the limitations of centralized version control are increasingly evident.

This whitepaper presents a strategic blueprint for migrating from Perforce to GitLab, a modern DevOps platform that enables scalable, collaborative, and secure software delivery. At the core of this transition is Git, the distributed version control system that underpins GitLab's powerful capabilities. While Git provides the flexibility and speed developers need, GitLab brings everything together—from source code management and CI/CD to security and compliance—in a single, unified platform.

Whether you're looking to modernize your development toolchain, reduce legacy system costs, or adopt end-to-end DevOps practices, this guide outlines the rationale, process, and best practices for a successful migration. Designed for developers, DevOps engineers, and enterprise architects, this whitepaper helps ensure a seamless transition to GitLab, powered by the agility and efficiency of Git, to future-proof your software delivery.

Considering a move from Perforce? This whitepaper offers a strategic plan for migrating to GitLab, leveraging Git's power for scalable, collaborative, and secure software delivery, modernizing your DevOps practices.

1. Introduction - I

The Evolution of Version Control Systems

Version control systems (VCS) have always been the backbone of software development, enabling teams to manage code changes, collaborate efficiently, and maintain a historical record of their work. Over the past two decades, VCS technology has evolved significantly—from centralised systems like CVS and Subversion to more advanced solutions such as Perforce (Helix Core), which addressed the scalability challenges faced by large, enterprise-level development teams.

As software development practices matured and the industry embraced Agile, DevOps, and continuous delivery models, the limitations of centralised systems became increasingly apparent. The need for greater flexibility, distributed collaboration, and seamless integration with automation tools led to the rapid rise of distributed version control systems (DVCS) like Git.

1. Introduction - II

Perforce: Centralised Control for Complex Projects

Perforce (also known as Helix Core) was built to serve teams working on large-scale, high-performance projects. Its centralized architecture places a single server at the core of all version control activities. This model allows for:

- Precise access control down to the file level.
- Efficient handling of large binary files (which are often difficult to manage in other systems).
- Performance optimization for single, monolithic codebases.
- Integration tracking that records how and when files have been merged or branched.

Because of these features, Perforce has been widely adopted in industries such as gaming, finance, aerospace, and semiconductor design fields where projects involve vast amounts of data and strict control over every aspect of the codebase.

However, as development teams have become more distributed and software delivery practices have shifted toward Agile and DevOps models, many organisations are finding that the centralised nature of Perforce can become a limiting factor. Teams working across multiple locations often experience delays, and adapting Perforce to newer, more flexible workflows can require significant effort.

Git: Distributed Development for Modern Teams

Git was created to solve a different set of challenges. Instead of relying on a single central server, Git uses a distributed model where every developer has a complete copy of the entire repository, including its full history. This design offers:

- Local branching and merging, allowing developers to experiment and collaborate freely without affecting the main codebase.
- Improved speed and flexibility when working offline or across different time zones.
- Resilience and redundancy, as every copy of the repository acts as a backup.
- Seamless integration with continuous integration/continuous deployment (CI/CD) tools, making Git a natural fit for DevOps workflows.

Git's distributed nature aligns well with modern software development practices such as Agile and DevOps, where flexibility, collaboration, and rapid iteration are essential. It also provides a foundation for scaling development teams across different locations and time zones without compromising speed or reliability.

As a result, Git has become the preferred choice for most new projects and is widely adopted by both small teams and large enterprises. It supports the fast, collaborative, and automated development practices that drive modern software engineering.

2. Comparative Analysis: Perforce vs Git - I

Understanding the fundamental differences between Perforce and Git is essential for teams considering migration. Each system was designed with different goals and workflows in mind, and these differences affect how development teams operate daily.

Architecture & Design Philosophy

Perforce
Follows a centralised architecture. All version control operations depend on a single, authoritative server. This design provides tight control over files, access permissions, and integration history.

Git
Implements a distributed architecture. Every developer has a complete copy of the repository, including its entire history. This enables fast local operations and reduces reliance on a central server.

Branching and Merging

Perforce
Uses a Streams model. It's powerful and flexible, but can be complex to manage, especially for teams new to the system.

Git
Uses lightweight branches that developers can create, modify, and merge easily. This encourages flexible workflows like feature branching, Git Flow, and trunk-based development.

Performance and Scalability

Perforce
Performs exceptionally well for large binary files and monolithic repositories. It's optimised for projects with vast file sizes and strict version control requirements.

Git
Optimised for distributed teams and excels at managing large numbers of text-based files and frequent branching/merging. For large files, solutions like Git Large File Storage (Git LFS) can be used.

2. Comparative Analysis: Perforce vs Git - II

Collaboration Model

Perforce
Requires developers to connect to a central Perforce server for most operations. This can be limiting for distributed teams or those working offline.

Git
Enables decentralised collaboration. Developers can work offline, commit changes locally, and push updates when convenient.

Integration and Ecosystem

Perforce
Well-suited for industries like gaming, hardware development, and other fields requiring management of large repositories and binary assets. Integration with modern DevOps workflows may require additional tooling.

Git
Highly adaptable to DevOps practices. Offers strong integrations with CI/CD pipelines, microservices architectures, and modern security and testing tools.

Cost and Licensing

Perforce
Requires licensing fees that can increase with team size and feature requirements.

Git
The core system is open-source. Teams can use free options or opt for enterprise platforms like GitLab CE or GitHub Enterprise for additional features and support.

Learning Curve and Community Support

Perforce
It may have a steeper learning curve, especially for developers unfamiliar with its branching and permission models. Community resources are more limited compared to Git.

Git
Widely adopted, with a massive community and extensive documentation, tutorials, and support options available.

2. Comparative Analysis: Perforce vs Git - III

Summary Comparison Table

Feature	Perforce	Git
Version Control Model	Centralised	Distributed
Performance	Faster for large files & monorepos	Optimised for distributed teams
Branching	Streams (complex but powerful)	Lightweight branches
Collaboration	Requires P4 server connection	Distributed, allows offline work
Integration	Strong for gaming, hardware, and large repos	Strong for DevOps, CI/CD, microservices
Cost	Licensing required	Open-source (GitLab CE) & enterprise options

Perforce offers strong centralised control and is optimised for handling large files and monolithic repositories, making it ideal for specific industries. However, Git provides the flexibility, scalability, and modern tooling integrations required to support today's fast-paced, collaborative, and distributed development environments.

3. Strategic Considerations for Migration - I

Why Migrate from Perforce to Git?

The decision to migrate from Perforce to Git is often driven by a combination of business goals and technical needs. While Perforce offers powerful features for specific use cases, many organisations are finding that it no longer aligns with the demands of modern software development.

Migrating from Perforce to Git is a strategic decision influenced by several factors:

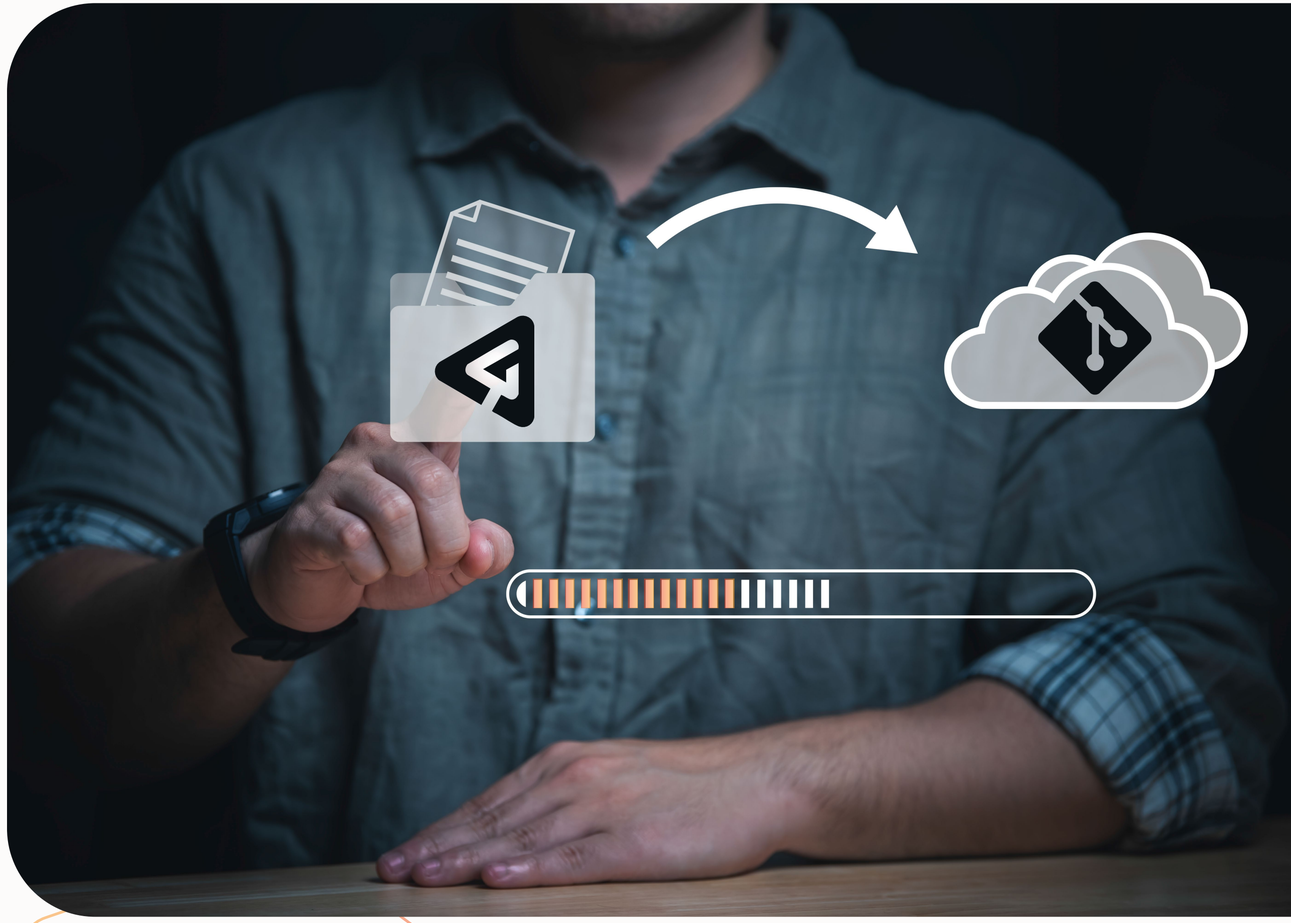
Licensing Costs: Git is an open-source tool under the GPL license, eliminating the licensing fees associated with proprietary systems like Perforce Helix.

Simplified Branching and Merging: Git's lightweight branching model allows for easier context switching and supports feature branching workflows more efficiently than Perforce's heavyweight branching system.

Enhanced Collaboration: Git enables developers to work independently on their local repositories, facilitating faster and more flexible collaboration without the need for constant communication with a central server.

Integration with Modern Tools: Git integrates seamlessly with contemporary DevOps tools, CI/CD pipelines, and issue tracking systems, providing a cohesive development environment.

Community and Support: The vast Git community offers extensive tutorials, documentation, and support, making it easier for teams to adopt and troubleshoot.



3. Strategic Considerations for Migration - II

Potential Challenges and Considerations

While the benefits are clear, migrating from Perforce to Git requires careful planning to avoid disruption and ensure long-term success. Some key challenges to consider include:

Preserving History and Data Integrity: Ensuring that the full history of the codebase—including commit histories, metadata, and permissions—is accurately transferred during the migration.

Workflow Adjustments: Development teams may need to adapt to Git's workflows and branching strategies, which differ significantly from Perforce's centralised model.

Tooling and Automation Changes: Existing automation scripts, integrations, and tooling built around Perforce may need to be reworked or replaced to function with Git.

Training and Change Management: Developers, testers, and operations teams may require training to become proficient with Git. Clear communication and support during the transition period are essential.

Large Files and Binary Data: While Git excels at managing text-based code, handling large binary files may require additional solutions like Git Large File Storage (Git LFS).

Assessing Organizational Readiness

Before initiating the migration, consider the following:

Team Proficiency: Evaluate your team's familiarity with Git to identify training needs.

Infrastructure Compatibility: Ensure that your current infrastructure can support Git and any associated tools.

Workflow Alignment: Analyze existing workflows to determine how they will translate to Git's model.

Data Migration Strategy: Plan for preserving commit history, metadata, and access controls during the transition.

Risk Management: Develop a rollback plan to mitigate potential issues during migration.

By carefully considering these factors, organizations can position themselves for a successful transition from Perforce to Git.

4. Migration Planning and Preparation - I

Migrating from Perforce to Git is a significant transition that requires detailed planning to ensure success. Before executing any migration tasks, organisations must assess their current environment, define their goals, and prepare both their teams and infrastructure for the change.

At VivaOps, the migration process involves transitioning code and version control data from Perforce Helix to Git, using Git P4 as the intermediary tool. Once the data is converted into Git repositories, the final step is to transfer them into GitLab, where ongoing development will continue.

Migration Overview

The migration will include:

- Converting repositories, commit history, branches, and labels (tags) from Perforce into Git using Git P4.
- Transferring the resulting Git repositories to GitLab as the new version control platform.
- Preserving historical data and workflows to maintain continuity for development teams.
- This approach ensures that essential project history and collaboration models are retained during the transition.

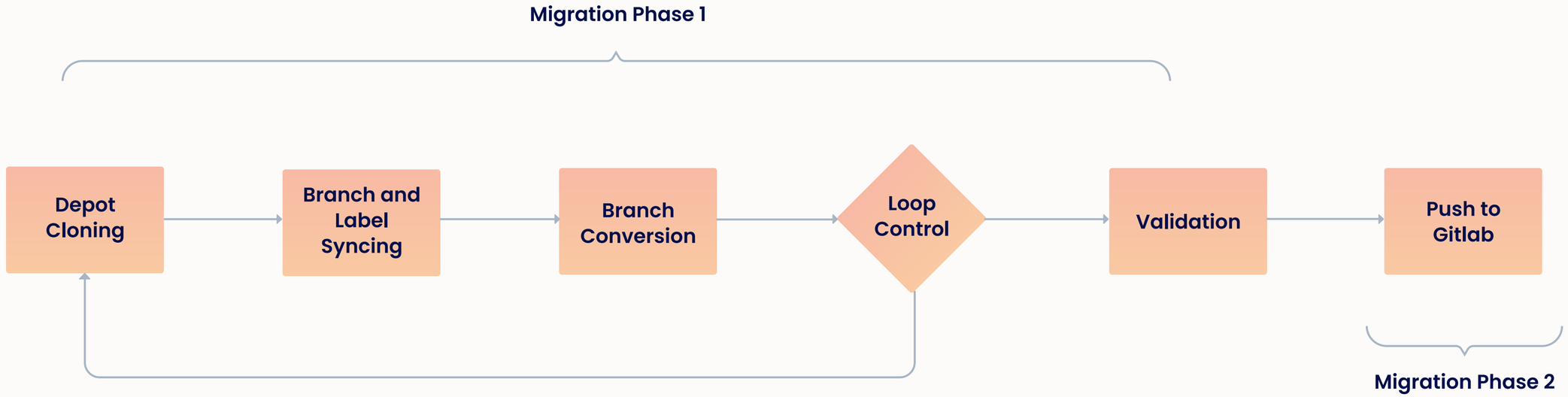
Key Migration Items

Item	Notes
Repositories	Commit history and branches preserved
Branches	Migrated as remote branches
Tags	Labels in Perforce converted to Git tags
Users	User mapping is configured through GitLab OAuth (Azure AD)

4. Migration Planning and Preparation - II

Process and Requirements

Migration Flow



Entry Criteria

Before starting the migration, the following must be defined and reviewed:

- Version of Perforce in use.
- Hosting details for the Perforce environment.
- Single Sign-On (SSO) and plugin configurations.
- Branching conventions.
- Number of depots.
- Size of binaries.
- Typical Perforce changelist sizes.

4. Migration Planning and Preparation - III

Edge Cases to Consider

- Case sensitivity: Git P4 is case sensitive.
- Depot migration limits: Git P4 can only migrate one depot at a time.
- Branch retrieval: Depot syncing is required after cloning to ensure all branches and labels are captured.

Prerequisites

- Git, Git-P4, and Perforce CLI tools installed on the staging server.
- Perforce credentials configured on the staging server.
- GitLab Personal Access Token (PAT) set up for pushing to GitLab.

Assumptions

- The project follows a conventional branching strategy (e.g., //depot/dev… maps to "dev").
- Large binary files are not stored in dummy depots during migration.

End State

Once the migration is complete:
Repositories, branches, commit history, and user access will be fully migrated to GitLab.

```
Updated branches: feature dev hotfix
labelDetails= {'code': 'stat', 'Label': 'test-label', 'Update': '2025/01/28 07:16:53', 'Access': '2025/02/05 15:47:45', 'Owner': 'sathish', 'Description
': 'Created by ajitesh.\n', 'Options': 'unlocked noautoreload', 'View0': '//depot/...'}
Processing branch: * master
Processing branch: remotes/p4/depot/dev
Creating and checking out new branch dev from remotes/p4/depot/dev
Switched to a new branch 'dev'
Processing branch: remotes/p4/depot/feature
Creating and checking out new branch feature from remotes/p4/depot/feature
Switched to a new branch 'feature'
Processing branch: remotes/p4/depot/hotfix
Creating and checking out new branch hotfix from remotes/p4/depot/hotfix
Switched to a new branch 'hotfix'
Processing branch: remotes/p4/master
Creating and checking out new branch master from remotes/p4/master
fatal: a branch named 'master' already exists
Cloned and processed: depot into depot-repo
```

Teams will need to complete post-migration configuration, including:

- Setting up CI/CD pipelines.
- Defining branch policies.
- Configuring merge request (MR) approval workflows.
- Implementing segregation of duties (SOD) policies.

This preparation ensures that both historical data and operational workflows are preserved, enabling teams to transition smoothly to Git.

5. Step-by-Step Migration Process - I

The migration from Perforce to Git, using Git P4 as an intermediary, follows a structured sequence of steps. This approach ensures a smooth transition, maintains historical data integrity, and addresses known limitations during the process.

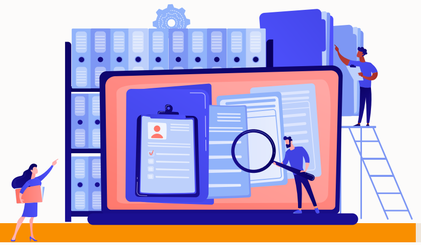
Step 1

Environment Preparation

- Install required tools on the staging server:
  - Git
  - Git-P4
  - Perforce client CLI
- Configure Perforce credentials.
- Set up GitLab Personal Access Token (PAT) for repository push access.
- Review and confirm:
  - Perforce version and hosting setup.
  - Branching conventions.
  - Number of depots and binary sizes.

Step 2

Repository Assessment



- Identify the repositories to migrate.
- Map out corresponding branches, labels (tags), and user access control requirements.
- Validate that no unsupported configurations or irregular branching conventions exist.

Step 3

Initial Migration with Git P4

Use Git P4 to:

- Clone the Perforce depot.
- Convert commit history, branches, and labels into Git format.

- Note: Git-P4 will create remote branches (remotes/p4/<branch>).

```
vivaops@vivaops-ThinkPad-L14-Gen-4 ~/projects/new-depot/depot (master)$ git branch -a
* master
  remotes/p4/HEAD -> p4/master
  remotes/p4/depot/dev
  remotes/p4/depot/feature
  remotes/p4/depot/hotfix
  remotes/p4/master
```

Edge consideration:

Since Git-P4 can only migrate one depot at a time and is case sensitive, careful tracking and validation are required at this stage.

5. Step-by-Step Migration Process - II

The migration from Perforce to Git, using Git P4 as an intermediary, follows a structured sequence of steps. This approach ensures a smooth transition, maintains historical data integrity, and addresses known limitations during the process.

Step 4

Branch Structure Adjustment

- **Automation scripts** are applied to:
  - Convert **remote branches** into standard Git branches.
  - Iterate over all depots for complete migration (since Git-P4 does not natively support migrating all depots at once).

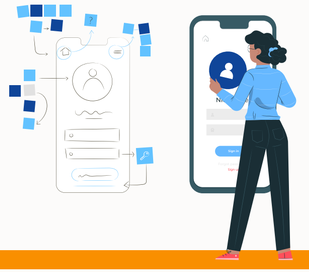
Step 5

Large Files Handling

- Review for large binary files.
- Migrate these using **Git Large File Storage (Git LFS) or S3 integration,** as Git alone does not manage large binaries efficiently.

Step 6

User Mapping



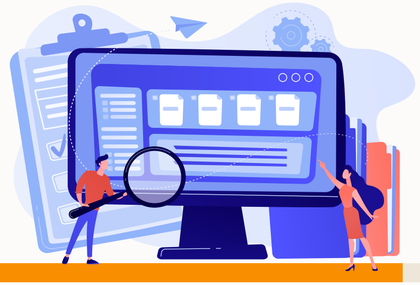
- Update GitLab configuration (gitlab.rb) to support **OAuth with Azure AD**.
- Align user accounts between Perforce and GitLab to maintain access control consistency.

5. Step-by-Step Migration Process - III

The migration from Perforce to Git, using Git P4 as an intermediary, follows a structured sequence of steps. This approach ensures a smooth transition, maintains historical data integrity, and addresses known limitations during the process.

Step 7

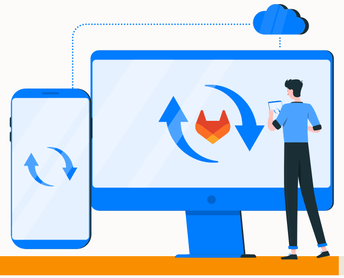
Validation and Testing



- Validate the migrated repositories:
  - Check commit history, branches, and tags for completeness.
  - Verify user access and permissions.
  - Ensure all branches and labels have been captured correctly.
- Perform **sync tests** between Perforce and Git to confirm data consistency.

Step 8

Final Transfer to GitLab



- Push the fully converted and validated Git repositories to GitLab.
- Confirm successful repository creation and access in GitLab.

Step 9

Post-Migration Configuration

- Set up:
  - **CI/CD pipelines.**
  - **Branch protection policies.**
  - **Merge Request (MR) approval workflows.**
  - **Segregation of Duties (SOD) policies.**
-
- Notify development teams of the completed migration and provide documentation/training as needed.

6. Post-Migration Optimisation - I

Completing the migration from Perforce to Git is only the first step. To fully realise the benefits of the new version control system and minimise disruption, teams must optimise workflows, configurations, and team practices in the new GitLab environment.

A. Performance Monitoring and Validation

Monitor repository performance to identify and resolve any latency or access issues.

Validate branch structures, commit histories, and tags to ensure data integrity has been preserved.

Periodically verify that large files stored via Git LFS or external storage (e.g., S3) are accessible and performing as expected.

B. Workflow Enhancements

Branching policies:
Establish or refine branching strategies suitable for Git, such as feature branching, Git Flow, or trunk-based development.

CI/CD pipelines:
Build or migrate continuous integration and deployment pipelines within GitLab to support automated testing, security scanning, and deployment.

Merge request (MR) workflows:
Define MR approval processes, including peer reviews and automated checks to maintain code quality.

C. Security and Access Controls

Access permissions:
Review and adjust user roles and permissions in GitLab to ensure appropriate access levels.

Single Sign-On (SSO) integration:
Confirm that SSO (such as OAuth with Azure AD) is fully functional and covers all migrated users.

Segregation of Duties (SOD):
Implement or update SOD policies to maintain compliance and prevent unauthorised changes to critical codebases.

6. Post-Migration Optimisation - II

Completing the migration from Perforce to Git is only the first step. To fully realise the benefits of the new version control system and minimise disruption, teams must optimise workflows, configurations, and team practices in the new GitLab environment.

D. Developer Training and Support

Provide training on Git workflows, GitLab features, and best practices to help developers adapt quickly.

Establish **support channels** where team members can get help with new tools or processes.

Encourage sharing of **lessons learned** and continuous feedback to refine workflows.

E. Automation and Tooling Enhancements

Evaluate and adopt **automation tools** that integrate with GitLab for code quality checks, security scanning, and deployment monitoring.

Explore additional **DevOps integrations** that can further streamline the development process, such as issue tracking, monitoring, and container orchestration tools.

F. Continuous Improvement

Regularly review the new environment's performance, developer satisfaction, and operational efficiency.

Collect feedback from team members and iterate on processes to ensure the Git/GitLab setup continues to meet evolving team needs.

Post-migration optimisation is not a one-time task but an ongoing effort. By monitoring performance, refining workflows, strengthening security, and fostering a culture of continuous improvement, organisations can ensure their new Git environment delivers long-term value and supports agile, collaborative development.

7. Conclusion

A version control migration is never just about moving data—it's about enabling what comes next.

By completing the transition from Perforce to Git, teams don't just adopt a new tool; they open the door to better collaboration models, smoother DevOps workflows, and a foundation that can evolve as technology and business needs change.

While this migration requires careful planning and execution, it also creates an opportunity to revisit workflows, streamline processes, and empower developers to work more efficiently.

For teams ready to future-proof their development practices, this migration is not the end of a project, it's the beginning of new possibilities.

8. How VivaOps Can Help

At VivaOps, we understand that Version control and toolchain migrations are not a simple lift-and-shift exercise. It's a complex process that affects code history, team workflows, automation, and compliance. As a strategic GitLab Select Partner, VivaOps helps engineering leaders go beyond migration to modernise their entire DevSecOps ecosystem.

What we deliver:

Version control and toolchain migrations
Migrate from Perforce or other legacy systems to Git and GitLab, preserving commit history, branch structures, and user access.

CI/CD optimisation and pipeline modernisation
Reduce build times, eliminate flakiness, and accelerate deployment velocity.

Security and compliance integration
Shift-left security with GitLab's SAST, DAST, and dependency scanning integrated into development workflows.

Cloud native transformation and CloudOps
Containerise workloads, adopt GitLab SaaS, and implement Kubernetes at scale.

SRE and TestOps excellence
Embed reliability engineering and automated testing to improve release quality and resilience.

Enablement and training
Equip teams with best practices, live coaching, and standardised GitLab blueprints.

VIVAOPS

About vivaops

VivaOps is a leading provider of AI-powered DevSecOps solutions. We offer solutions designed to modernize and improve your development tasks, ensuring your software is built with utmost flexibility, security, and scalability.

Contact us at info@vivaops.com