BitKeeper
Scalable Version Control

The Advantages and Disadvantages of Monolithic, Multiple, and Hybrid Repositories

By Oscar Bonilla
BitKeeper, Inc.

Abstract

Large organizations that produce a lot of code face an important choice in how to structure their source code. They can choose to create a single monolithic repository that holds everything or they can split their source code into sub-repositories and manage them independently.

Traditionally, choosing between these two approaches involves significant tradeoffs. Particularly for companies that desire a distributed workflow, these tradeoffs can feel like a false dichotomy, but few viable alternatives have been available.

In this article, we explore the advantages and disadvantages involved in monolithic and multiple repository structures as well as examining BitKeeper’s hybrid approach, which provides the benefits of both options while minimizing complexity and preserving distributed workflow even for very large source bases.

The Cardinal Rule of SCM: History Matters!

One of the guiding principles of source control is that history matters. The source base is more valuable if its history is preserved and easily accessed. This avoids repeating mistakes from the past, can elucidate why code ended up being written in a certain way, and is generally useful.

Any solution that makes history harder to follow or impossible to access should automatically be considered suboptimal.

Monolithic Repositories

Advantages of Monolithic Repositories

There are many significant advantages to having all of your code in the same repository. Perhaps the most compelling one is **the ability to tightly couple interdependent changes**. That is, the ability to atomically update both an interface and all the users of that interface.

Some examples:

- An API of a library evolves and the clients of the library need to be modified to use the new API.
- A server component is updated with a new protocol and the client is updated to talk to the server using it.
- A kernel interface is added and the user library used by applications needs to be updated.
- All of the above when an API, protocol, or interface is deprecated.
- Documentation of a system needs to be kept up to date with changes.

Wasting hours on a problem that's due to synchronization issues between two different components is an extremely frustrating (and common) debugging experience. Atomicity of commits eliminates this problem in most modern source control systems when all the source code resides in a single repository.

In summary, keeping all of your code in a single monolithic repository **provides the advantage of simplicity**.

Additional advantages of having the source base in a single repository include:

Simplified Repository Administration

When managing a single large repository, some common maintenance tasks are easier to deal with than with multiple repositories. For example, backups are easier to plan because there is only a single repository that needs to be backed up. Also, triggers are simple to write because they only need to be maintained in one place.

Easy Branching and Tagging

Monolithic branching strategies are less complex than the same strategies applied to multiple repositories because they don't need to be maintained across multiple repositories. Tags also mark a single reproducible point in time for all of the source base.

Straightforward Code Management

All of the tools that deal with code and its requirements work better if everything is in one place. For instance, searching across files, moving files around, writing scripts; all of these actions benefit from a single global repository.

Better Utilization of SCM tools

Most modern version control management systems have a series of tools designed to make searching within the source code better. The bisect command, for example, that performs a binary search over the history looking for when a bug (or feature) got introduced doesn't work across multiple repositories.

Easier Developer Training

Having all of the source base in a single repository allows developers to leverage their existing knowledge of SCM commands. They don't need to learn anything new and can use tools like the internet for finding answers about how to do SCM operations.

Disadvantages of a Monolithic Repository

Of course, there's a downside to having everything in one place.

Decreased Performance

This really only impacts large distributed repositories. While distributed workflow is very desirable, the nature of having everything everywhere means that at a certain size[1] a traditional distributed architecture will grind to a halt.

Security and Access Control Problems

Securely restricting access to sections of the source becomes very difficult or impossible in traditional monolithic structures. This is particularly true with distributed architectures, which by their very nature give everyone a copy of everything. However, it is also an issue for centralized systems where access control is forced to use a permissions-based system subject to many well-known foibles and hacks. There's no way to air-gap sensitive code in a monolithic architecture.

In organizations or open source projects where every engineer (or the entire world) is welcome to access the entire source base, this is not a problem. However, for organizations and projects in which the security of intellectual property is an important consideration, this can be extremely problematic. It also limits the ability of these organizations to outsource or scale up non-core functions like support, documentation, and peripheral components.

Loss of Productivity and Local Meaning

As repositories grow, the time it takes to do repository-level operations increases accordingly. But more importantly, the signal-to-noise ratio decreases as well. For example, searching for the callers of a particular interface in the repository will find *all callers*, even in unrelated packages that just happen to have a similarly named interface. At a certain size, this can become extremely unwieldy.

For tags and commit comments in particular, having to come up with meaning that applies globally might be impossible. For example, it might be convenient to have a component that is released independently as well as used as part of a larger product (e.g., a library). Tags and commit comments that make sense in the context of the product might not make any sense in the context of the component.

1. This generally happens at around 4 GB total repo size, but it depends also on the number of files, the depth of the history, the exact setup, and the SCM system being used. Even in extremely optimized environments, at around 4M changesets and 1M files (with about 9.5 GB of source code) git becomes totally unusable and takes days for some operations.

Multiple Repositories

Advantages of Multiple Repositories

There are significant advantages to splitting your source into different repositories, especially with distributed SCMs, as this is the only way that they can scale beyond 4GB or so of source code.

Improved Performance

Most modern distributed version control systems make it impossible to clone a subset of the repository. If all of your code is in a single huge repository, obtaining it might be unacceptably slow. Having separate repositories means you can effectively clone just a subset of your source base and increase the performance and scalability of large projects by chopping up your source base into however many repositories are required from a logical organization and performance perspective. This is traditionally the only way to scale distributed architectures.

Security and Access Control

For distributed system access control at least becomes *theoretically* possible with multiple repositories. Sensitive code can be controlled in multiple ways (including air-gapping) for both distributed and centralized systems.

Increased Agility and Scalability

If you have multiple repositories, outsourcing is less complicated from a security standpoint; just hand out the repository for the component to the third party and you're done. There's no complicated access control layer or forcing contractors to work outside the SCM system.

This additional flexibility from a security point of view can become essential for adding additional resources and providing agility and scalability to an organization's otherwise linear development efforts.

Increased Productivity and Local Meaning

This is the inverse of the downside of productivity from large monolithic repositories. Signal-to-noise ratios go up, as anyone searching a smaller, more focused code base is more likely to find what they are looking for and references are more easily understood.

Flexibility

Having one repository per component allows you to separate policy for the different components. For example, different triggers can be used for each component.

Preserved Workflow

Multiple repositories allow very large projects to scale and maintain a distributed workflow, without grinding to a halt or hogging unreasonable amounts of compute, storage, or network resources.

Disadvantages of Having Multiple Repositories

There are, of course, significant downsides to multiple repositories. The biggest is the **increase of complexity** and all the pitfalls that this brings to any system. The more complicated a system, the more potential points of failure. Such as:

Loss of Atomicity of Commits

Changes that should logically belong in one commit are artificially split in multiple commits—one per component—with no binding between them. This makes debugging and understanding of the history of the code harder.

Workflow Limitations

Having many disconnected repositories makes it hard for engineers to use certain workflows. For example, to refactor a product and work on an API, multiple disconnected commits must be made and reviewed. This increases the probability of introducing bugs in the process.

Early Hybrid Solutions

In this section we'll discuss some solutions that have been attempted to gain the advantages of both monolithic and multiple repositories without the resulting problems. As we shall see, none of these solutions is satisfactory from either a theoretical point of view or—as a quick search on Google will confirm—from a practical one.

Our focus here is on distributed systems both because the distributed workflow is generally seen as superior to centralized (a topic for another paper) and because distributed systems have been more obviously limited to a certain size before being *forced* to adopt a multiple repository structure due to performance issues and therefore have shown more innovation in this area.

Checking in URLs and Keys

One of the simplest ways you can combine several repositories into a large one is to just check in a file with the mapping between directories in the file system and a tuple (URL, key) that describes where to get the component from (URL) and what revision to fetch (key). In order to make the common SCM operations seamless, the SCM tool can be extended to understand this file or a wrapper can be written that does the necessary recursive calls to the SCM tool.

While this approach might seem to work in simplistic workflows, it is both cumbersome to use and error prone.

For example, one of the most useful features of distributed source control is the ability to do sideways pulls — that is, to collaborate with peers without necessarily publishing your changes to a centralized server. In a multi-repository configuration, however, the traditional approach for implementing synchronization operations — like pull — is to iterate over the different components and run the operation in each of them. However, this has the following drawbacks:

- The atomicity of SCM operations is broken such that an error in one of the components can leave a repository that is out of sync. What is worse, depending on the amount of error checking performed by the component iterator, it would be possible to half-push your changes and not notice.
- With a large enough number of components, doing one synchronization operation per repository can become unacceptably slow. As an example of one of the worst cases, if there are no changes between the repositories being synchronized, it would still take one synchronization operation per component to realize there are no differences.

As a real world example, let's look at Git submodules, which use a mechanism very similar to checking in a (URL, key) tuple.

Git Submodules

The file `.gitmodules` in the root of the repository consists of a map of URLs for components and the directories in the main repository where they are supposed to go.

Whenever you commit in git, the key of the top commit in the submodule is checked into the repository as if it was a file (using a special `mode`). Git submodules are not unlike just having the build system handle the population of the multiple components.

For example, `git pull` does not automatically descend into submodules to execute the pull; it needs to be given an explicit option. What's worse, if there are conflicts during the pull, the merge is not in any way handled in a standard way. The submodules end up in a detached head state, where it's easy to get confused and commit the wrong head to the main repository.

Another problem is branching. When you switch branches in a git repository that has submodules, the submodules are not necessarily switched to the new branch. Unless the developer carefully iterates over each submodule, or uses a script that has knowledge of the structure of the repository, it's easy to get confused and work with a collection of submodules that are not in the same branch.

Git Subtrees

Git provides an alternative to git submodules: git subtrees. The idea behind subtrees is to have a single repository in which all of the different components reside, and provide an on-demand way to split off components into their own git repositories.

The most obvious drawback of this approach is that the main repository is still a single entity. No savings in space or number of files are achieved in the main repository.

A less obvious problem is that the history of the components is now intermingled with the history of the product. That is, whenever a commit is made in the main repository, git has no knowledge about component boundaries. This means that developers need to be very careful about how they commit their work. But even careful developers will run into problems.

In the man page for git subtree it is recommended that:

> *In order to keep your commit messages clean, we recommend that people split their commits between the subtrees and the main project as much as possible. That is, if you make a change that affects both the library and the main application, commit it in two pieces. That way, when you split the library commits out later, their descriptions will still make sense. But if this isn't important to you, it's not necessary. git subtree will simply leave out the non-library-related parts of the commit when it splits it out into the subproject later.*
>
> — *https://github.com/git/git/blob/master/contrib/subtree/git-subtree.txt*

Following the above advice and making two commits for what is effectively a logical change breaks the atomicity of the commits and introduces a failure point where the repository can be cloned to a commit in which it's broken. Not following the advice and making a single commit forces the developer to pollute the commit comments with information that won't make sense once the individual components are split up. This is not only confusing, but it could be a leak of IP if one of the components is going to be outsourced.

BitKeeper's Nested Collections

BitKeeper invented and popularized distributed version control in 1998. Therefore, we've been struggling with the limitations imposed by the effective performance limit of 4GB for distributed systems longer than anyone else. For several years, we've had a way to split up large repositories without breaking the atomicity of commits.

With BitKeeper 7.0, we've officially rolled BitKeeper Nested Collections (aka BK/Nested), our answer to the dilemma.

Monolithic or Multiple Repositories?

The answer is a hybrid.

Of course, BitKeeper is not the only organization to offer up a hybrid between centralized and distributed systems. Centralized systems have been trying to graft distributed workflows onto their products for awhile now, and distributed systems have repeatedly tried to solve the scalability problem by borrowing elements of centralized repositories — for example, git submodules is also a hybrid solution.

However, we believe that our solution is by far the most evolved and the only one that captures all the fundamental advantages of both monolithic repositories and multiple repositories without any of their fatal flaws — all while supporting a distributed workflow, preserving history, and allowing the smooth rollback of *any* given commit.

Of course we would say that, so let's dig in and examine this hybrid solution in more detail.

What Is It?

A nested repository in BitKeeper is comprised of a collection of repositories bound together by a single timeline. This provides the atomicity of commits familiar to developers while at the same time providing the flexibility of treating each of the components as a standalone repository.

The key insight is that these components are still each individual BitKeeper repositories. In this sense they are similar to git submodules. However, and unlike git submodules, the components are more tightly bound to the product than just a (URL, key) pair.

Architecture

BitKeeper was the first version control system to popularize the concept of a ChangeSet. Earlier systems, like CVS, tracked revisions of individual files and didn't have any support for binding all of the files' changes together. BitKeeper introduced a manifest that records in an efficient manner exactly what revision of each file the entire repository is supposed to be at, and gives the entire repository a revision as well. In this manner, it becomes possible to talk about a certain revision of the repository and have that repository represent a consistent view of all the files.

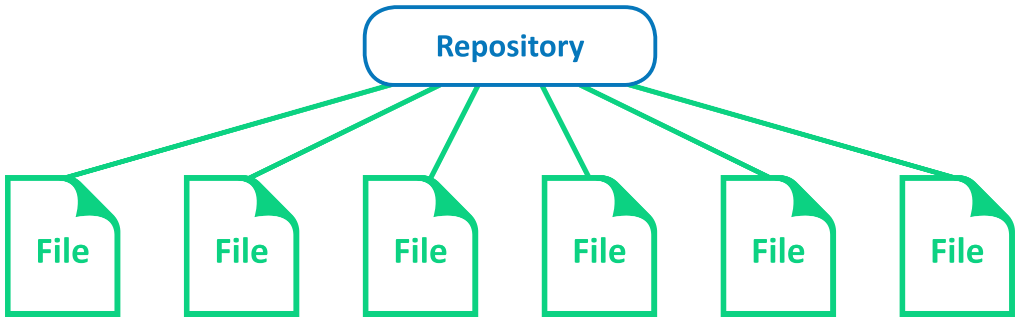


Figure 1. Traditional BitKeeper Repository

The way BitKeeper Nested stitches multiple repositories together is by creating an enclosing repository called the product. In this product, you can attach other repositories called components. The relationship between the product and its components is like the relationship between a repository and the files it contains.

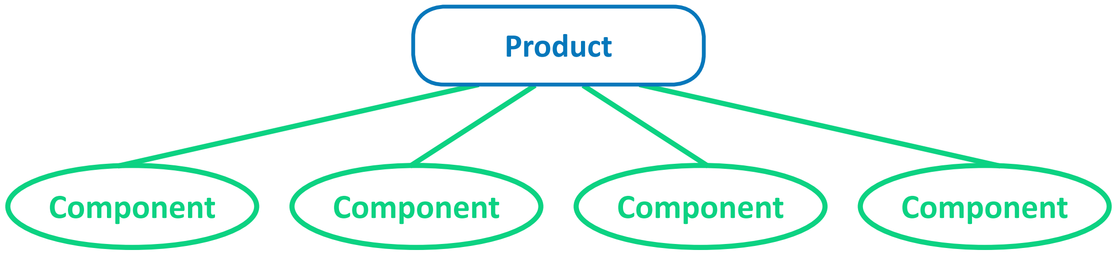


Figure 2. Nested BitKeeper Repository

However, there is one significant difference. In a nested repository the individual components can be treated in two different ways: 1) as part of the larger collection, and 2) individually as standalone repositories. The product enclosure provides a single timeline in which all the components are bound.

Unlike other solutions, all of the management of components is handled transparently by BitKeeper. That is, there are no new commands or special steps needed to work in a nested repository. The set of verbs familiar to users of traditional repositories — clone, pull, push, commit, etc. — remain unaffected.

There are, however, a few new concepts when operating in a nested repository that let developers make full use of them. We'll explore those in the next few sections.

Definitions

This section is provided as a quick reminder of some of the terminology used in describing BitKeeper Nested Collections (aka BK/Nested).

Product

A BitKeeper repository that has other BitKeeper repositories attached to it. In some sense, these components behave as if they were files in the enclosing product.

Component

A BitKeeper repository that is attached to a product. Components can only be attached to one product and cannot have other components attached to them (that is, a repository can't be both a product and a component).

Alias

A symbolic name stored in the product that refers to a collection of components. This name is version controlled such that the collection it references can vary without losing the history of its previous values.

For example, if in order to work on the documentation it's necessary to have the tools and docs components. They can be bound by an alias called `DOCS`.

Gate

A product that is considered safe. That is, a product that is on a server and backed up. The central idea is that if a change has made it to a gate, it is assumed to be in a safe location that won't be rolled back.

Portal

A fully populated product. That is, a product in which all of the components are present (populated).

Operation

BitKeeper's Nested repositories behave like traditional BitKeeper repositories in most ways, with one notable exception. Components can be absent from any given clone, but their presence is still recorded in the Product's manifest file. This effectively allows for sparse repositories where not all of the components are populated, but in which the product still knows about them and the namespace they occupy.

Furthermore, components can be detached from a product and either attached to a different product or be used standalone like traditional repositories. This provides an easy and natural way to collaborate by sharing components.

Note that the fact that the components are recorded in the product's manifest file is really important. For example, this can help prevent many conflicts like trying to create a file in the same path where a component should be. It can also help BitKeeper detect whether a pull in a sparsely populated product would result in an unresolved merge for a given component. BitKeeper can then either disallow the pull and inform the user of the problem or automatically populate the components necessary for completing the pull.

Aliases

Aliases are a powerful way to bundle subsets of components into a logical name. At their essence, aliases are version-controlled names for collections of components.

A few special aliases are provided:

ALL

This alias refers to all the components in the product, whether present or not. For example, if we want a new clone that is fully populated, we can run the following command:

```
bk clone -sALL bk://server/source
```

and it will bring in all the components of the product from source. If there are components which are not present in source, they will be cloned from the default gate.

HERE

This alias refers to the components currently populated in the repository in which the command is run.

THERE

This alias refers to the components populated in the remote URL that is specified in the command line. The following example should make its meaning clear. If, for example, we want to clone whatever subset of components is in the URL `bk://server/product`, we could run the command:

```
bk clone -sTHERE bk://server/product
```

and it will expand the alias **HERE** in the URL `bk://server/product` and clone that set of components.

PRODUCT

This special alias refers to just the product.

Advantages

Let’s take a look at the various advantages previously discussed for both monolithic and multiple repositories and see how BK/Nested measures up.

Simplified Repository Administration

BitKeeper’s Nested repositories can easily be managed like traditional repositories by having them be fully populated. Although the effort to split up a large existing monolithic repository is not trivial, the administration overhead imposed by BK/Nested is extremely modest.

Performance

BK/Nested allows for any repository to be divided into an arbitrary number of logical components. Aliases are really useful for this because it’s possible to bundle subsets of components under an easy to remember name. For example, the **`docs`** alias can bring in just the parts of the product needed to work on documentation, while the **`src`** alias can bring in all the components needed to work with the source code.

As the numbers of the repositories managed is arbitrary, BK/Nested can easily scale to extremely large repositories without performance issues.

Increased Productivity and Local Meaning

Just like with multiple repositories, BK/Nested allows developers to quickly focus on what’s important to them, without being bothered by irrelevant projects. The **HERE** alias is especially useful for this as it limits operations to that specific repository.

Easy Branching and Tagging

Since tags are applied at the product level, but also interpreted at the component level, the whole collection behaves as expected. Detached components can have their own tags — which won’t interfere with tags done in multiple products.

Straightforward Code Management

Since the behavior of Nested repositories is in most cases identical to traditional repositories, most tools work the same. The only caveat is for components that are unpopulated. Most BitKeeper commands will print a warning about them, but tools that don’t know about this might silently ignore them.

Better Utilization of SCM Tools and Easier Developer Training

The entire BitKeeper system was completely architected to be BK/Nested aware so any command that works in a monolithic repository will also work seamlessly across repositories as desired. This simplifies the administration of Nested repositories and makes it easier for developers to use them as their muscle memory of commands still applies.

Security and Access Control

Access to different parts of the source base can be restricted by either triggers, or by detaching individual components and reattaching them to a different product that does not include the components that should not be shared. This can allow for effectively air-gapping truly critical components without introducing build, merge, and sync nightmares down the road.

Increased Agility and Scalability

BK/Nested can be really useful for outsourcing because the external work can be managed efficiently under the umbrella of source control, without having security issues or problems when changes head back to integration with the main product.

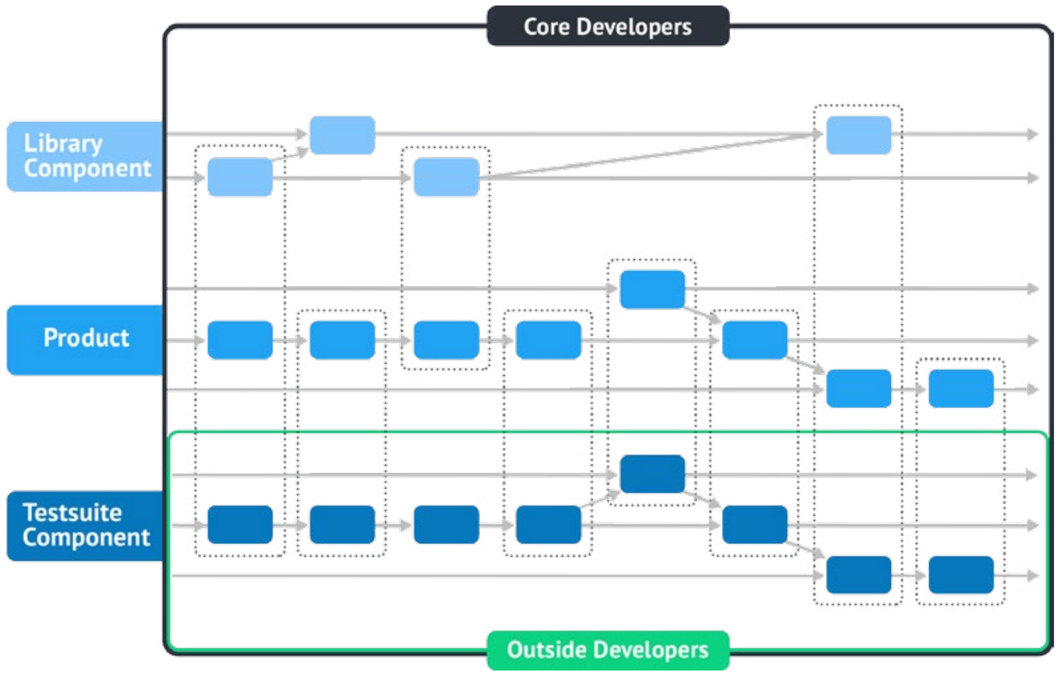
This allows organizations to quickly and easily expand and contract their potential resource base for sprints related to product releases and bug fixes as well as outsourcing less central activities like documentation, UI, peripheral components, etc.

Flexibility

Components are full BitKeeper repositories, which allow developers to separate policy for different components. For example, different triggers can be used for each component.

Preserved Workflow

BK/Nested allows very large projects to scale and maintain a distributed workflow, without grinding to a halt or hogging unreasonable amounts of compute, storage, or network resources.



Disadvantages

BK/Nested provides the advantages of both monolithic and multiple repositories, but does it also avoid the pitfalls?

Increased complexity

Although there is indeed a more complex structure underlying BK/Nested than handling a standard monolithic repository, for developers and administrators, there is very little additional complexity. BitKeeper seamlessly keeps everything up to date and in sync.

Loss of Atomicity of Commits

BitKeeper always keeps all components and libraries in sync across timelines *and* across collections of repositories. BitKeeper has a very mature feature set to ensure that interdependent changes are always tightly coupled by default. We call it TimeSync™.

Workflow Limitations

BK/Nested keeps repositories connected and aware of each other's changes. For example, if you refactor a component and change an API, both sets of changes will be bound by a product ChangeSet.

Decreased Performance

As previously discussed, by dividing repositories as needed, BK/Nested overcomes performance issues even for very large repositories, providing extremely scalable distributed workflows

Security and Access Control

BK/Nested avoids the problems of monolithic systems for both the all-or-nothing approach of traditional distributed SCM tools, which give everyone everything, and the centralized systems, which rely on a permissions-based system.

Loss of Productivity and Local Meaning

The ability to break repositories up into logical collections means that even very large collections retain local context and therefore keep developers from getting "lost" in irrelevant code.

Conclusion

In this article we have described the advantages and disadvantages of putting all of your source code in a big repository versus splitting it in multiple repositories. We have explored different solutions to both approaches and have shown how they fail to be satisfactory. We have also described BitKeeper's Nested repositories and shown how the approach taken by BitKeeper represents a good tradeoff between the simplicity of a single repository and the advantages of splitting the source base in multiple repositories.

The Sales Pitch

Wow. Look at that. We managed to check ***every** benefit box without a **single** disadvantage **in our own whitepaper***. That's pretty impressive. But we understand if you think it's potentially biased.

Cruise over to BitKeeper.com and try out a 90-day free trial and see for yourself.

With BitKeeper Nested Collections you really can have your cake and eat it too. Heck, if you're sitting at a big organization with a giant unwieldy source base that you'd just love to chop up and implement distributed workflows, without any of those pesky disadvantages we mentioned above, just give us a call at +1-408-370-9911. We'd like nothing better than to consult with you about how (and if) BK/Nested could help.

About the Author:

Oscar Bonilla is a Senior Software Engineer at BitKeeper, where he is one of the core developers of the source control engine. His curiosity has led him to work in many different areas including compilers, databases, performance, and document processing systems. He has more than 15 years experience with software development and has helped Fortune 500 companies with very long histories optimize their development workflow.

About BitKeeper:

BitKeeper launched the first viable distributed version control system in 1998 while also providing the first version control system for the Linux kernel.

BitKeeper, now on its 7th major release, provides scalable version control that is designed to be deployed with a minimum of fuss across the enterprise. It is used by organizations large and small including Cisco, Intel, EMC, Hitachi, HP, and Sony.

For more information:

+1-408-370-9911
+1-888-401-8808

www.bitkeeper.com

BitKeeper, Inc.
15466 Los Gatos Blvd. #109-285
Los Gatos, CA 95032

Copyright 2015 by BitKeeper, Inc. All rights reserved.