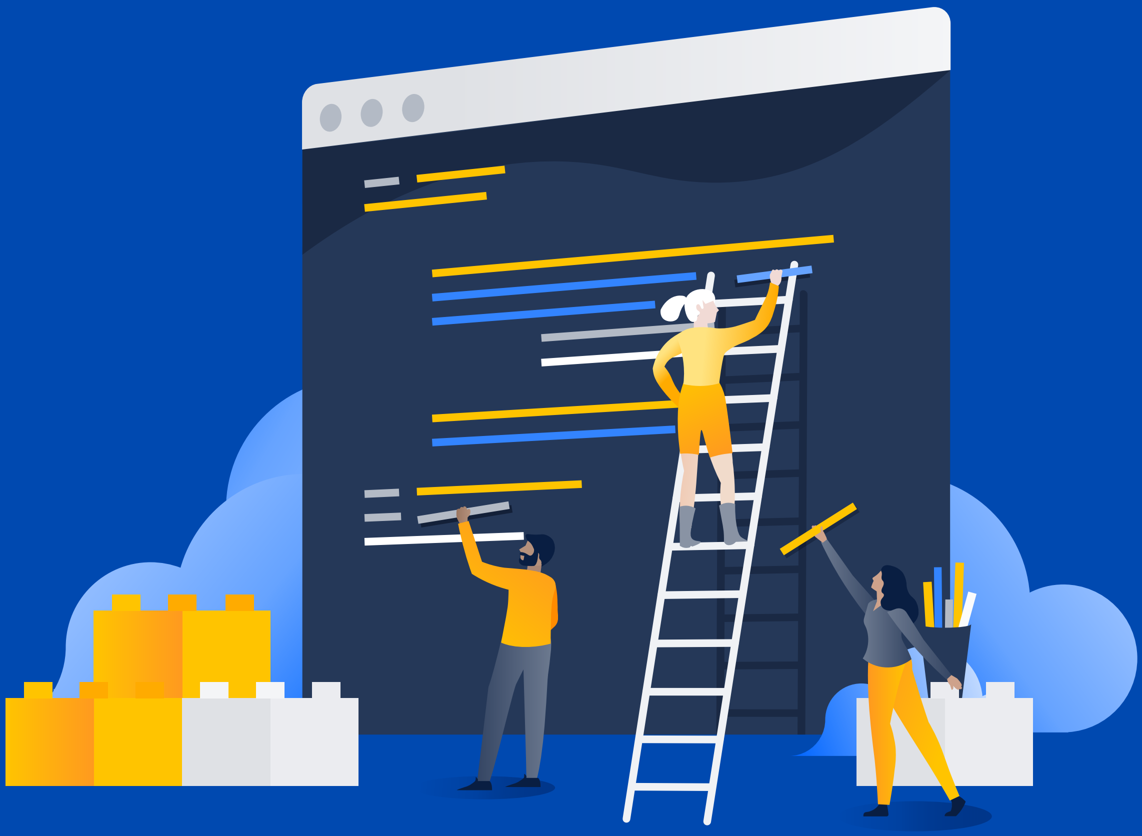
ATLASSIAN



Scale your organization's repositories with Bitbucket Mesh

Table of contents

Executive Summary

As you and your developers continue to write great software, your repositories continue to grow. We want to ensure that Bitbucket can support even the largest customers with the most complex repositories.

In this whitepaper, you'll learn about Bitbucket Mesh, the next generation of repository storage that can improve the performance and reliability of your repositories, even as you scale.

Introduction

Most developers don't put much thought into *how* their code is stored; they just learn what they need to know in order to write, edit, and push their code along. However, for engineering leads and managers who manage their organization's repositories, the storage and management of their team's code may keep them up at night.

Since clustering was introduced in Bitbucket Data Center, Git repositories have been hosted on a shared network file system or **NFS**.

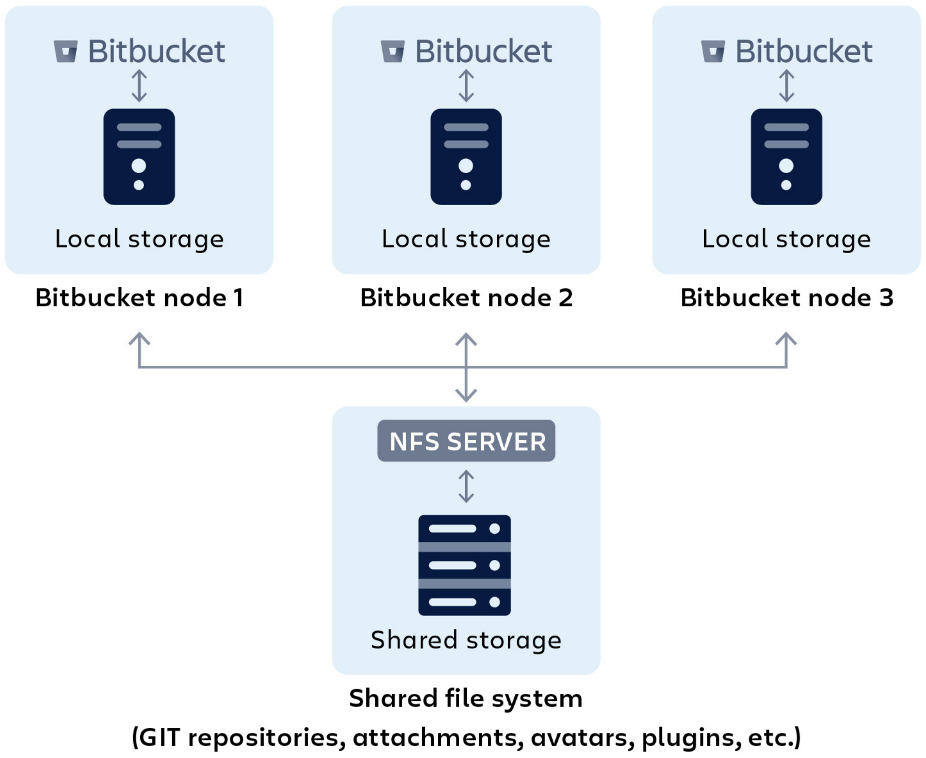


Fig. 1 – Bitbucket cluster for an NFS (pre version 8.0)

What is a network file system (NFS)?

A NFS is a framework designed to allow a user on a device to access remote files over a network. It defines the way files are stored and retrieved from storage devices across networks. It is commonly used where file sharing and storing happens across multiple machines or operating systems.

Bitbucket Server, Data Center, and Cloud have been using NFSv3. The NFSv3 specification was published in 1995, which most notably added support for 64-bit files and removed the 4.2 GB file size limit associated with early versions of NFS.

Network file systems have several advantages: centralized source of data, remote access, support for multiple operating systems, and reliability to name a few. However, due to the central configuration of an NFS server, any disruption to the server can cause a massive disruption as your developers would be unable to access or edit any files, making it a single point of failure. Additionally, as your repositories grow you will notice slower performance when accessing files.

To help increase the performance, reliability, and scalability of Bitbucket (especially as many of you look to migrate or are already operating in the cloud), Atlassian developed **Bitbucket Mesh**.

Mesh Overview

Bitbucket Mesh is a distributed, replicated, and horizontally scalable Git repository storage subsystem designed for high performance, scalability, and resilience. It's a new approach to storing Git data and handling Git requests which now operate as separate Java processes outside of Bitbucket. Repositories (and copies of repositories) are stored on the Mesh nodes, while the NFS server continues to host non-Git data, including projects, user avatars, attachments that may be associated with pull requests, comments, plugins, and Git Large File Storage (LFS) objects.

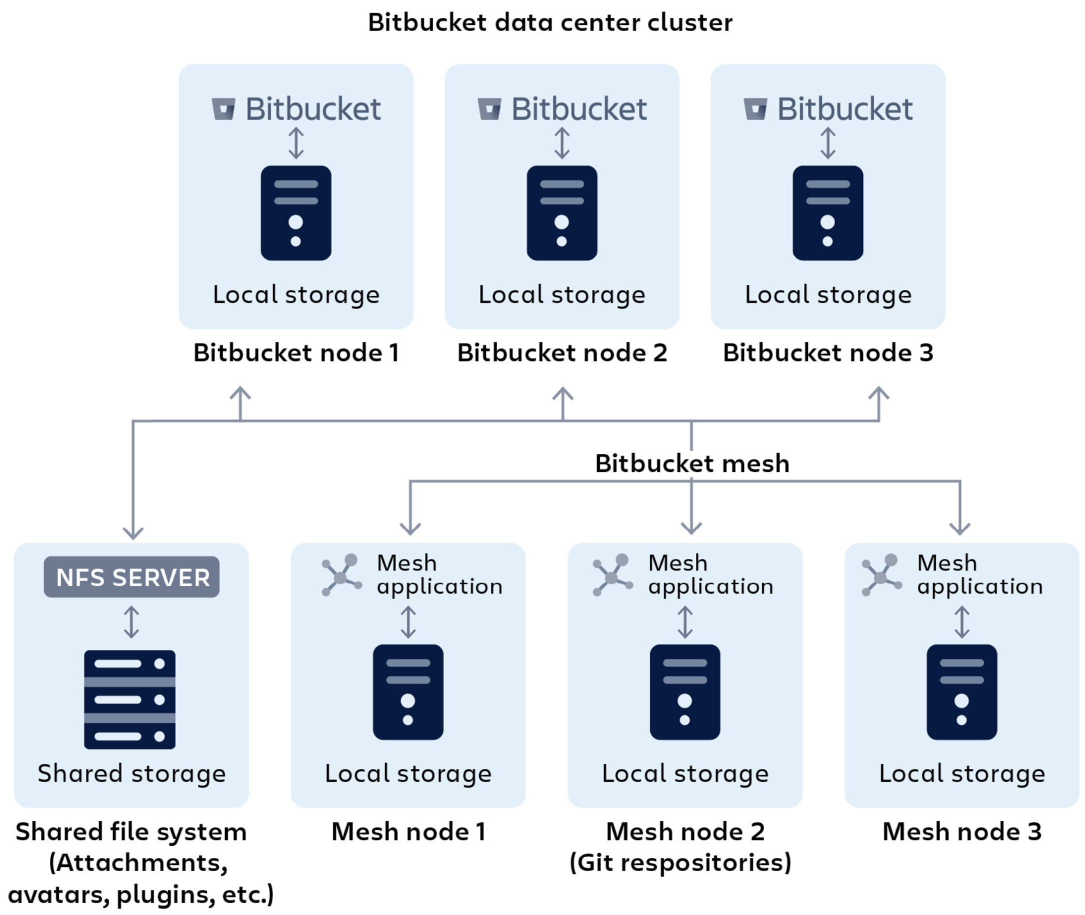


Fig. 2 – Bitbucket Mesh configuration with three nodes

Mesh upgrade and migration

It's possible to upgrade to Bitbucket 8.0+ and not adopt Mesh. Repositories would continue to reside on the NFS-based repository store. When you are ready to leverage Mesh, three or more Mesh nodes are deployed and Bitbucket is configured to use them. However, once the nodes are added to the system, they remain unused until existing repositories are migrated to Mesh or until new repositories are created there.

By default, new repositories aren't created on Mesh. This can be changed by enabling **Create new repositories on Mesh** in Bitbucket Administration. Forking existing repositories doesn't result in the fork being created on Mesh. If a repository resides on the NFS store, so do all forks, the existing and new ones.

You can also migrate existing repositories to Mesh. The UI provides a tool that allows repositories to be migrated individually or for all repositories to be migrated at once. Repository migration doesn't require downtime and can be carried out even while the repositories being migrated are still in use.

Not all repositories need to be migrated to Mesh simultaneously, permitting a gradual migration that may be phased and stretched out over many days, weeks, or more. This is often useful to de-risk a Mesh migration in case you are unsure if your Mesh deployment is appropriately sized or ready for production load. It is possible to move repositories gradually while monitoring load and resource usage.

Key concepts and terminology

The following information will help you understand the entirety of the Mesh system.

Mesh nodes: Instances of mesh java applications and their respective home directories.

Mesh app: The mesh java application instance.

Sidecar: A bundled mesh app – responsible for handling Git operations locally on the Bitbucket node. Imitates a pre-mesh environment.

Control plane: A subsystem in the core Bitbucket application that is responsible for managing the Mesh nodes. It is responsible for distributing configuration information, allocating repository replicas to nodes, routing requests, and the management and distribution of replica state, either consistent or inconsistent.

Partition ID: A collection of Repository Hierarchies that are to be distributed across mesh nodes. A set of mesh nodes gets assigned a Partition (default is three mesh nodes) to become the destination for these repository hierarchies.

Replication: Process of mesh nodes communicating with each other to either distribute git data from an incoming push or to repair any nodes that have detected inconsistencies.

Write change: Any git operation that will apply changes to refs. For example, a push.

Vote Majority / Quorum: In order for a write change to be committed, follower mesh nodes need to have majority “yes” votes to successful updates. Otherwise, the write change is rejected.

“Yes” vote: A yes vote instructs the leader that the git operations completed successfully and that changes can be applied to the mesh node.

Just-In-Time Fetch: The fetch that a mesh node will do to another mesh node if it fails to update refs. After successful fetching, a ref update is attempted again.

Topology: Data structure for mesh nodes informing it of which partitions it holds and which other mesh nodes hold other replicas of the partitions (and their RPC URL).

Sideband Channel: A constant RPC channel on which mesh nodes can communicate metadata and state back to Bitbucket.

gRPC: A modern, open-source, high-performance Remote Procedure Call (RPC) framework that can run in any environment. Efficiently connects services in and across data centers with pluggable support for load balancing, tracing, health checking and authentication. Applicable in the last mile of distributed computing to connect devices, mobile applications and browsers to backend services.

JWT: JSON Web Token – an open internet standard for creating or transmitting information between parties as a JSON (Javascript Object Notation). JSON objects can be transmitted quickly and contain all the required information about an entity to avoid querying a database more than once. The recipient of a JWT also does not need to call a server to validate a token.

Mesh Properties and Benefits

Replication and resilience

The key to Bitbucket Mesh's resilience is the concept of replication. In the NFS-based repository system, there's only one copy of the repository. In Mesh, multiple copies of any given repository exist on different mesh nodes, with the specific number controlled by the "replication factor", which is adjustable at the global level.

When Bitbucket introduced clustering, it improved the availability of the NFS. Fig. 3 shows a typical Bitbucket cluster in a high-availability deployment. This type of deployment can sustain the loss of a cluster node either due to scheduled maintenance or a failure.

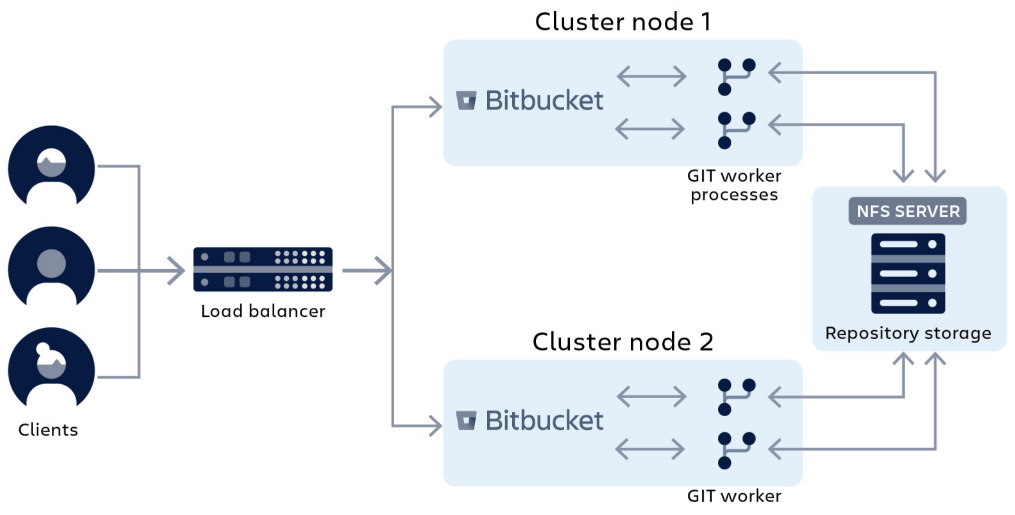


Fig. 3 – Bitbucket cluster with NFS-based repository store in a typical high-availability deployment

While this system may be built with redundant disks, power supplies, and network interfaces, it's still a single node and subject to failures. Besides, it can't be restarted for maintenance (such as operating system patching) without a Bitbucket system outage.

Various commercial NFS "appliances" take the concept of redundancy further, including redundant system boards. These boards mean that updates can be carried out without interrupting operations, and often most components can be replaced without an outage. However, these appliances are expensive and still deployed in a single physical location, so they aren't truly redundant.

Unfortunately, fully redundant network filesystems are highly unsuited to Bitbucket's needs, or more specifically Git's needs, as the synchronization and coordination overheads result in very high I/O latency for filesystem operations. As a consequence, Bitbucket's performance suffers. This problem also applies to cloud-based NFS services such as Amazon's Elastic Filesystem and other cloud offerings, making highly available cloud deployments unobtainable.

Bitbucket Mesh solves this problem by spreading and replicating the repositories, consisting of multiple redundant nodes. When repositories are migrated to Mesh, they're replicated to multiple Mesh nodes, ensuring that the **loss of any single node has no impact on the availability of the repositories it hosted** because each still has replicas available on other nodes. When Mesh nodes are brought back online, they automatically repair their replicas and are returned to service.

Furthermore, it's possible to host Mesh nodes in different physical locations. This permits different Mesh nodes to reside in different data centers, with separate power supplies, network infrastructure, cooling systems, and other factors that can greatly increase resilience. Using cloud terminology enables a multi-availability zone repository store.

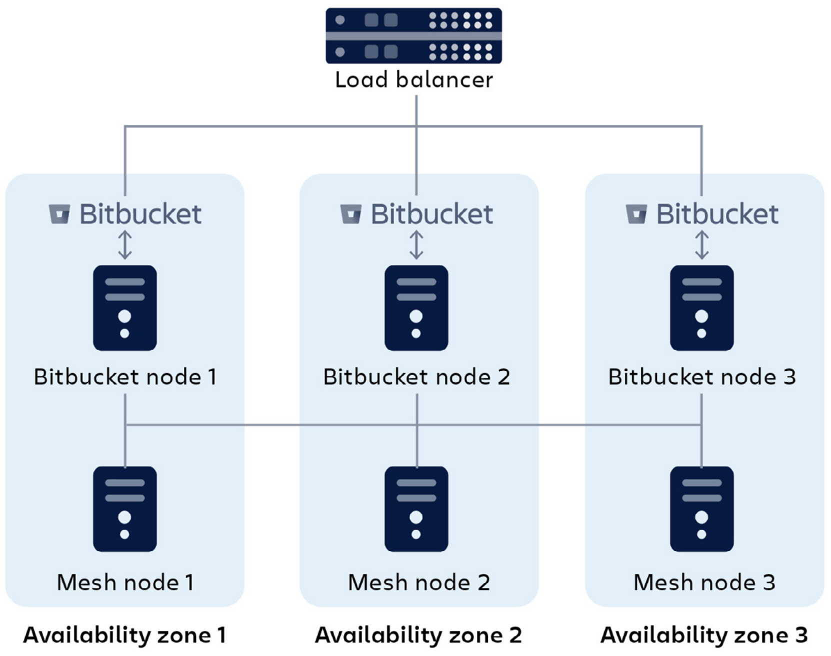


Fig. 4 – Multi-availability zone deployment of Bitbucket with Mesh

Fault tolerance

Fault tolerance enables a system to continue operating in the event of a fault with any of the components. Replication provides increased fault tolerance over the NFS-based repository store. The standard supported NFS deployment is a Linux-based NFS server. While this system may be built with redundant disks, power supplies, and network interfaces, it's still a single node and is subject to failures. Besides, it can't be restarted for maintenance (such as operating system patching) without a Bitbucket system outage.

Mesh replicas are located on three or more completely separate Mesh nodes. The nodes can leverage independent hardware and don't share the same power source, cooling, network, or even physical location. This means that any physical disturbance to one node doesn't result in a total shutdown.

The minimum configurable replication factor is three. This permits the loss of one replica while still supporting writes. The writes succeed on a quorum of replicas, where "n" is the replication factor a quorum of (n/2 + 1) replicas must be available for a write to succeed. The result of the division should be rounded down. Read operations aren't subject to the same quorum logic and only require one available and consistent replica.

Example

For example, with a replication factor of three, a minimum of two replicas must be present for a write to succeed. For a replica to participate in a write operation, it must be consistent. A replica may be inconsistent because the node missed one or more writes while it was offline and hasn't been repaired yet. So, a node hosting a replica may be online but may still be inconsistent and thus, ineligible for participating in a write transaction. As a result, it won't count towards the quorum.

Rebalancing

Bitbucket Mesh implements partition migration to allow repository replicas (actually, partition replicas) to be migrated between nodes. This process helps keep the nodes balanced in terms of storage and performance.

Partition migration exists to support rebalancing repositories across Mesh nodes in support of the following two use cases:

- Adding a new Mesh node: When a new Mesh node is added to the system, it should start servicing requests for existing repositories. When a new Mesh node is added, a rebalancing operation takes place, migrating one or more partition replicas from existing Mesh nodes to the new Mesh node.
- Removing a Mesh node: When a Mesh node is removed, it becomes unavailable to host replicas. It's important to understand the difference between an offline node and a removed Mesh node. If a Mesh node is simply shut down or is drained and disabled, this node still hosts replicas. They are unavailable temporarily.

Removing a Mesh node is a configuration change that means the Mesh node is no longer known to the control plane and no longer hosts replicas. For the system to maintain the same availability guarantees, the replicas must be hosted by that node to be migrated to another node before removal, and specifically, to another node that doesn't already host replicas for the given partition.

Rebalancing doesn't take available disk space or load into account. It implements an algorithm that tries to uniformly distribute replicas amongst available Mesh nodes. By keeping your nodes balanced, it ensures that any one node features a drop in performance compared to the others.

Repository repair

Bitbucket Mesh features a built-in repository repair function to help maintain resilience and consistency. A repository replica needs to be repaired when the replica has fallen behind either because the node missed one or more writes while it was offline, or because the node failed to replicate the write. The repair is also used to initialize a repository replica from scratch.

Repository repair happens when:

- A new replica is created for a partition.
- A repository is migrated from NFS to Mesh. The migration does an upload to a 'primary' migration target and then, uses the repair to sync up the other replicas.
- Migrating partitions from one node to another. This happens during rebalancing, after a new Mesh node has been added or before the deletion of a Mesh node.

Scalability

Scalability ensures that Bitbucket can keep up as your repositories continue to grow in size and complexity. There are two types of scalability, horizontal and vertical. The primary difference between horizontal scaling and vertical scaling is that horizontal scaling involves adding more machines or nodes to a system, while vertical scaling involves adding more power (CPU, RAM, storage, etc.) to an existing machine. Individual NFS-based repository systems can not scale horizontally, as all files are stored centrally.

For Bitbucket Mesh, we focused on three core scalability factors:

- Disk I/O (input/output) bandwidth and IOPS (input/output operations per second) capacity
- CPU available to Git worker processes
- Memory available to Git worker processes

With NFS, scaling disk I/O bandwidth is restricted to vertical scaling. You can add additional NFS filesystems to provide some horizontal scaling, but any given repository can only ever exist on one filesystem at a time. However, CPU and memory can be added to the system by adding *application nodes*. Since each application node has a shared view of mounted NFS filesystems, it can service a request for **any repository** hosted by the system. This is true regardless of whether the instance has two or 20 application nodes.

The NFS-based Bitbucket cluster permits horizontal scalability but with some limitations as to how the NFS-based repository store can be scaled.

Specifically:

- Adding new cluster nodes increases CPU and memory available to Git worker processes as well as increases network bandwidth.
- Adding additional NFS data stores increases repository storage I/O bandwidth.
- Adding additional NFS data stores only provides the ability to scale the I/O bandwidth available to existing repositories. Only new repositories are created on the additional NFS data stores while existing repositories don't benefit from the additional data stores. Additional data stores provide scaling where the load is mostly uniformly distributed over all repositories.

If your development teams are using a monorepo – a single large repository that hosts multiple projects, potentially used by all developers or a large fraction of the development staff – additional data stores do not offer any scalability.

Mesh has a slightly different characteristic when scaling horizontally since a given Mesh node can only service requests for repositories for which it hosts a replica. This can become a bottleneck where some "hot" repositories exist, repositories that are large, busy, and have a disproportionally large fraction of usage. But Mesh provides true horizontal scaling of both processing and storage capacity. Repositories are replicated to multiple Mesh nodes, and each replica is capable of actively serving both read and write traffic. Each replica adds capacity, and this capacity can be incrementally added or removed, permitting flexible scaling both up and down.

Example

If your Mesh system has a replication factor of three and a deployment of 20 mesh nodes, only three mesh nodes can service requests for a given “hot” repository, with the other 17 nodes remaining idle or only servicing requests for other repositories.

This problem could be resolved by increasing the replication factor to 20, resulting in all 20 mesh nodes hosting a replica of each repository, and thus being able to service requests for any repository. However, this comes at the cost of increased storage space required: in this case, a 20x increase over the storage requirement of the NFS-based deployment.

In reality, for most systems, there will be a happy middle ground that balances the need for scaling with the desire to minimize the cost of storage. When migrating an existing system, this middle ground can be obtained, at least approximately, by analyzing the distribution of requests using the access logs.

Performance

Bitbucktet Mesh aims to improve the performance across your repositories, especially as you have more code and more users. When moving from a single node to a multiple-node (clustered) deployment, the system has increased scalability due to the additional CPU and memory available to service user requests. A clustered system can sustain more concurrent users successfully. However, individual requests can become slower. This occurs as a side-effect of moving the repository storage from a local filesystem to a network-attached filesystem, specifically NFS. In effect, this moves the storage further away from the processing, increasing filesystem input and output (I/O) operation latency.

In a single-node Bitbucket deployment, the repository storage is hosted on a local filesystem (see Fig. 5). In such a system, I/O latency for obtaining the size of a file, reading a block of data, and other operations is fast, often taking a few microseconds where data is cached, or on the order of 100 microseconds for a disk read.

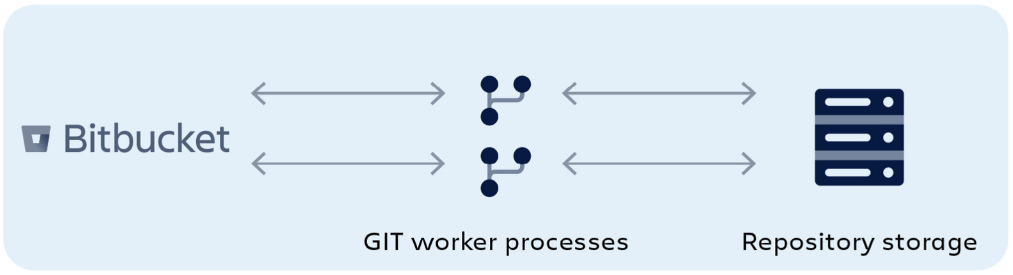


Fig. 5 – Bitbucket with local repository store

In a multi-node Bitbucket cluster deployment, the repository storage is hosted on a remote NFS (see Fig. 6). In such a system, I/O latency for similar operations is 10 to 1000 times slower due to the necessity of requests transiting the network, and due to the shared nature of the filesystem, many things can't be cached on the NFS client (that is the cluster node) but can only be cached on the NFS server, thus still incurring network latency overheads.

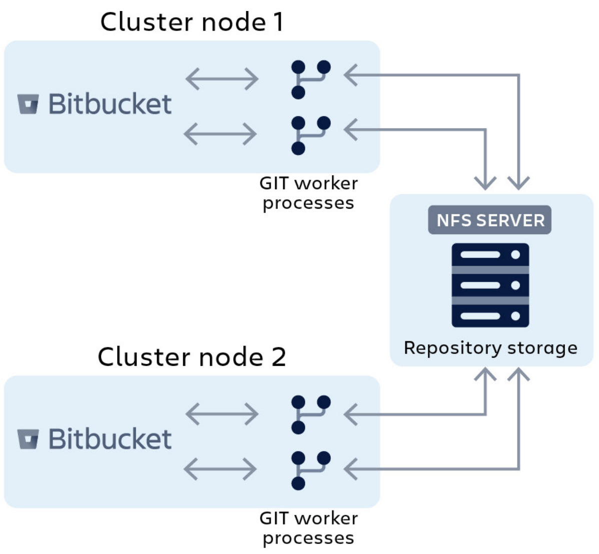


Fig. 6 – Bitbucket cluster with NFS-based repository store

This increase in I/O operation latency is particularly harmful to Git as it relies on low-latency filesystems for high performance.

Example

To illustrate this, we can take a user request to list all branches or tags in a repository. This would result in Bitbucket forking a **git-for-each-ref** process to obtain the list from the repository on disk. For a repository with many branches, particularly if **git-pack-refs** hasn't run recently, such a request may require, for example, 5000 individual I/O operations. On a local filesystem where operation latency is 10 µs, this request would take 50 ms to complete, appearing almost instantaneous to a user.

The same request on an NFS-based repository store, where operation latency is often in the range of 0.5-2 ms, could instead take between 2.5 and 10 s, which is an unacceptably long time for an interactive user interface.

Bitbucket Mesh solves the above problem by **moving the processing to the storage**, eliminating the additional I/O operation latency that exists in the NFS-based system (see Fig. 7). If we take the above example and apply it to a Mesh-based system, the cluster node makes a single remote procedure call (RPC) to a Mesh node. Then, the forked Git process makes its 5000 I/O requests to the local storage, taking 50 ms to complete (that is 5000 x 10 µs). Then, factoring in the RPC round trip overhead of, for example, 1 ms, the entire request would take a total of 51 ms to complete – again appearing almost instantaneous to a user, improving the overall performance experience of each user.

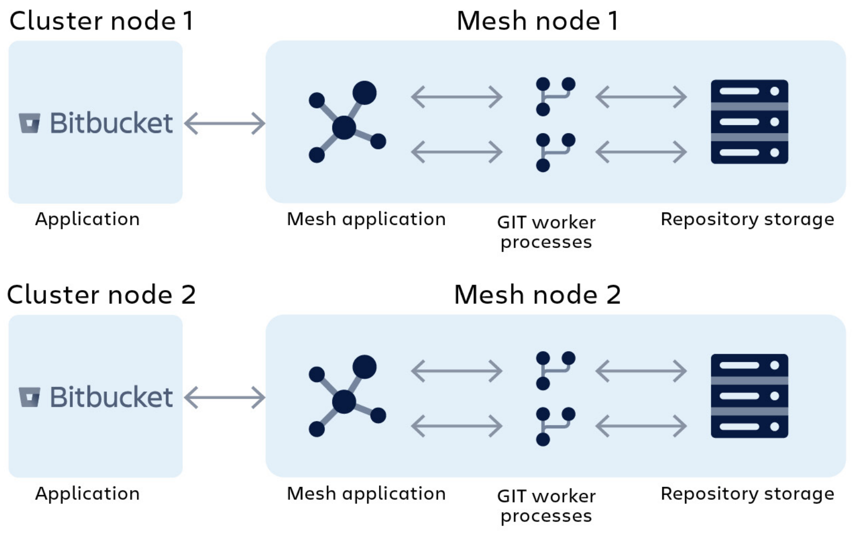


Fig. 7 – Bitbucket with Mesh-based repository store

Request routing

A request for a given repository can be routed to potentially any Mesh node that hosts a replica of that repository. When a client, either a web UI client or a Git client connecting via SSH or HTTP, makes a request, they're connected to one of the nodes of the primary cluster that runs the core Bitbucket application.

These connections are initially handled by the load balancer, which then proxies those connections through to one of the cluster nodes. Web UI connections generally require session stickiness so subsequent requests for the same session are routed through to the same node, although the initial connection is typically randomly assigned to a node. So, given a large number of users, the load from web users will be roughly uniformly distributed. However, connections from Git clients don't require stickiness, and a user performing multiple clones can see each request connected to a different cluster node.

Example

For example, the user may be asking for a list of all branches, viewing the contents of a file, or comparing the diff between two branches. These needs are fulfilled by the Mesh subsystem, with the application running on the cluster, making gRPC remote procedure calls on the Mesh nodes.

While processing a request, the cluster node handling the request may need to query the database for information, and it may need to read or write to the Git repository. This need is obvious for Git operations such as clone, fetch, or push. However, even the web UI connections often require information from the Git repository.

Before making an RPC, a Mesh node must be selected to fulfill the request. This node:

- Must host a replica of the repository that is the target of the request.
- Must be online and not draining. Draining means the system is trying to quiesce the node so it can be taken offline, perhaps for maintenance.
- The replica must be consistent. A replica may be inconsistent either because the node missed one or more writes while it was offline or because the node failed to replicate a write.

Given the set of Mesh nodes that match the above criteria, the request will be assigned to a Mesh node randomly, with the set of all requests expected to be uniformly distributed across eligible Mesh nodes, which should help with load balancing and performance.

Mesh Deployment Considerations

A traditional (pre-Mesh) clustered Bitbucket deployment is comprised of the following components:

- One or more Bitbucket Application nodes
- Load balancer
- Relational database management system (RDBMS)
- OpenSearch instance
- Network filesystem (NFS)

Enhancing this deployment to include Mesh requires the addition of a minimum of three Mesh nodes. A minimal clustered Bitbucket deployment with Mesh can be seen in Fig. 8.

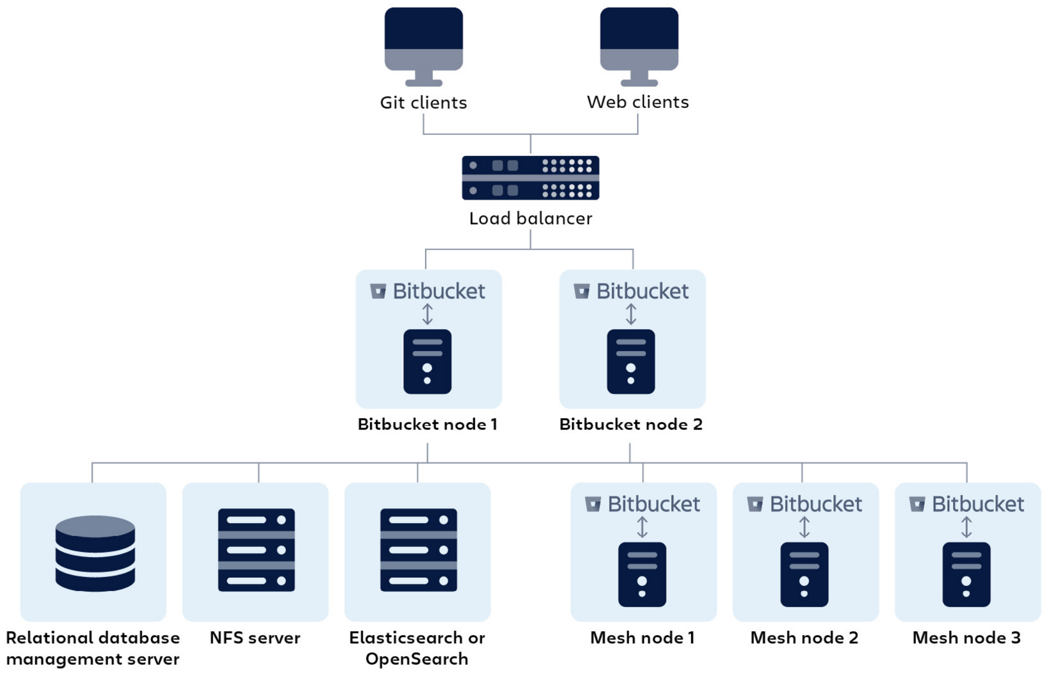


Fig. 8 – Minimal clustered Bitbucket Data Center deployment with Mesh

A minimum of three Mesh nodes must be deployed because this is the minimum supported replication factor required for the system to sustain a failure of one node and still form a write quorum. Any number of Mesh nodes three or greater can still be deployed, as this may be needed to scale processing, storage, or networking capacity.

It should be noted at this point that the application running on the Mesh nodes isn't the normal Bitbucket Java application, which we'll call the core Bitbucket application from here onwards. Rather, a new application is installed on the Mesh nodes – we'll call it the Mesh application. This new application is a gRPC server that provides remote procedure calls (RPCs) to read, write, and manage the Git repositories managed by the Mesh application.

On the surface, it might appear that once the repository data is migrated to the Mesh nodes, the NFS server could potentially be removed. The NFS server is still necessary and continues to host non-Git data. However, once all repositories have been migrated to Mesh, many of the strict performance requirements Bitbucket sets for the shared filesystem are no longer present.

This means:

- The requirement to use NFSv3 can be relaxed to permit NFSv4 usage. Historically, NFSv4 wasn't supported as it requires more round trips for the same operation when compared to NFSv3, which resulted in inferior performance.
- Cloud-managed NFS filesystems such as AWS Elastic Filesystem (EFS) can be utilized.
- Potentially, in the future, non-NFS shared filesystems may be available for utilization. This is subject to further testing to ensure the basic requirements are still met, including POSIX compatibility, delete on the last close, locking, etc.

Sidecar

The Git source code management (SCM) logic, which was part of the core Bitbucket application before Bitbucket 8.0, has been extracted to the Mesh application.

Specifically, when upgrading Bitbucket to 8.0+, even repositories hosted on the NFS repository use a sliver of the Mesh code path. Before Bitbucket 8.0, the Git SCM logic existed in the core Bitbucket application. It was responsible for forking Git worker processes (see Fig. 1). In Bitbucket 8.0, this Git SCM logic is factored out of the core Bitbucket process into a separate process that we call the Sidecar (see Fig. 9.)

This sidecar is the same application as Bitbucket Mesh, but only a small subset of the functionality is used in this role. Think of it as a Mesh-lite process. It's used for repository access but doesn't leverage concepts such as replication or partitions.

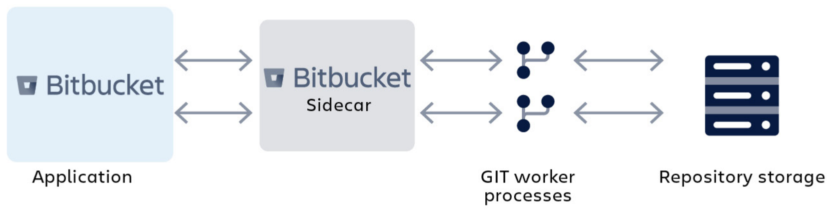


Fig. 9 – Bitbucket with sidecar process

Where previously the Bitbucket application made Java method calls to access process-local SCM code, now it makes a gRPC call to the sidecar process to do the same. The primary areas where the administrator needs to be aware of the existence of the sidecar are monitoring and troubleshooting.

The existence of the sidecar process doesn't constitute "using Mesh." The sidecar process isn't listed in Mesh nodes in the administration UI.

Authentication

The core Bitbucket application communicates with the Mesh process via gRPC (general remote procedure call to connect services). The Mesh application acts as the gRPC server and the core Bitbucket application acts as the gRPC client. Mesh application processes also communicate amongst each other via gRPC, primarily for tasks such as write replication and repairs.

These RPCs are authenticated using JWT. Each request has a JWT auth token with claims signed by the caller and each response has a token signed by the responder. A 2048-bit RSA signing key pair exists for each Mesh node, and one exists for the control plane, that is for the core Bitbucket application. The key exchange happens when a Mesh node is first added to the system.

Complexity

A typical Bitbucket Server instance is relatively simple. It consists of the Bitbucket Java application plus a database, filesystem, and an OpenSearch instance. This is made slightly more complex when an instance is deployed in a cluster since there are multiple instances of the Bitbucket Java application running.

Mesh complicates this, with a second Java application type needing deployment (the Mesh application) and the additional core state existing on multiple Mesh nodes. The following activities become more complex:

- Deployment
- Bitbucket version upgrades
- Monitoring
- Backup and restore
- Troubleshooting

For small Bitbucket instances that wouldn't benefit from any of the performance, resilience, or scalability benefits that come with Mesh, migrating to a Mesh-based deployment may not be desirable. Instead, the NFS-based Git repository storage subsystem may better suit such instances.

Additional storage capacity requirement

With the traditional NFS-based repository store, there's exactly one copy of each repository on disk. Filesystems may also be configured as RAID1, RAID5, or similar, which incurs some additional storage space.

Bitbucket Mesh provides increased scalability, performance, and resilience, but at the cost of additional disk storage requirements. Bitbucket Mesh uses replication of repositories to achieve these goals, and with a minimum replication factor of three, the minimum disk storage requirement is also increased by a factor of three.

Example

If you have 500 GB of repository data on NFS, deploying three Mesh nodes with a replication factor of three, each Mesh node will require 500 GB of storage for these repositories. This is a total of 1500 GB of storage.

It's quite challenging to determine the exact amount of disk space required. Take the same 500 GB of repository data on NFS, with a replication factor of three, and five Mesh nodes. In this case, the total storage requirement is also 1500 GB but distributed over five Mesh nodes. Assuming a large number of repositories of the same size, 1500 GB could be divided by five, indicating a storage requirement of 300 GB per Mesh node.

In reality, not all repositories are of equal size. In the above example, if the size of one repository was 400 GB, three of the five Mesh nodes would require at least 400 GB of storage.

This doesn't necessarily translate to a linear (for example, three-fold) increase in storage pricing. Most Bitbucket deployments that are built for high availability rely on expensive "NFS appliances" for highly scalable and reliable storage. Bitbucket Mesh permits building out horizontally using usually the most cost-effective internal storage, direct attached storage (DAS), or storage area network (SAN) based storage.

Multiple availability zone deployments

The concept of an availability zone is a common cloud term used to describe a data center where all resources share a physical location and often cooling, power, and other core subsystems. A multiple availability zone deployment leverages two or more of these availability zones to provide additional redundancy. The system is then more resilient in the face of power, cooling, and other hardware failures, as well as to events such as fires and floods.

As described previously, Bitbucket Mesh supports the concept of multi-availability zone deployment. This wasn't possible with the NFS-based repository store since the NFS server could only exist in a single location and thus, be a single point of failure. Consequently, deploying the Bitbucket application nodes in multiple availability zones didn't increase resilience. Furthermore, the low filesystem I/O latency Bitbucket and Git require means that even if a multi-availability zone NFS service was available, the performance of this deployment would be unacceptable.

A successful multi-availability zone deployment of Bitbucket Mesh requires the following:

- The ability to ensure the additional latency incurred for the RPCs is acceptable.
- Replicas are distributed across Mesh nodes so that they exist in a sufficient number of availability zones to permit a single availability zone failure, while still having enough replicas to form a write quorum.

The first requirement can be met by ensuring the round trip latency between Mesh nodes is under five milliseconds (ms), and similarly, the round trip latency between the Bitbucket application nodes and the Mesh nodes is under five ms. This can be measured with an Internet Control Message Protocol (ICMP) ping between nodes. This figure of five ms implies that these availability zones must be relatively close geographically, generally within the same city. In cloud terminology, this also means that while multi-availability zone deployments are viable, multi-region deployments aren't. The typical latency between regions is often tens of milliseconds and often over one hundred milliseconds.

The second requirement can be met when nodes are distributed between availability zones so that the loss of one availability zone leaves a sufficient number of replicas for a write to succeed. The writes succeed on a quorum of replicas, where "n" is the replication factor a quorum of (n/2 + 1) replicas must be available. Note that the result of the division should be rounded down. For example, with a replication factor of three, a minimum of two replicas must be present for a write to succeed.

Example

In a simple scenario with a replication factor of three and three Mesh nodes each in separate availability zones, it's easy to check how an outage in one availability zone would still result in two replicas being available. See Fig. 10.

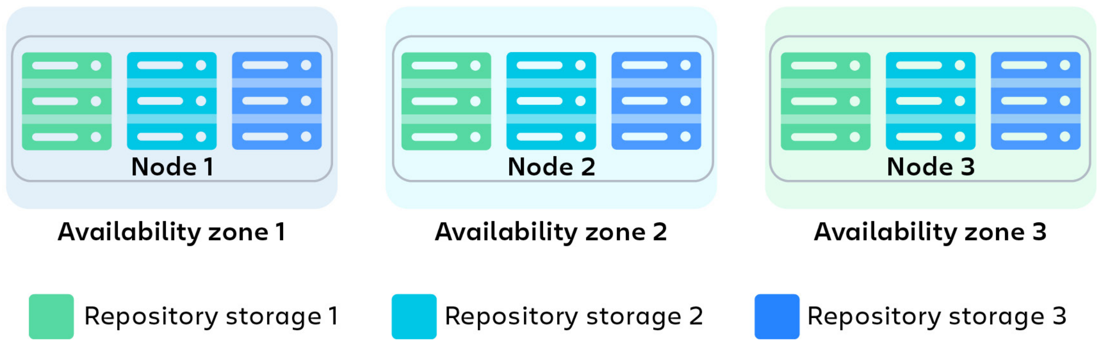


Fig. 10 – Redundant multi-availability zone deployment

Example

However, a scenario with a replication factor of three and only two availability zones results in a non-redundant deployment. See Fig. 11, where a failure of availability zone 1 would mean repository 1 only has one remaining replica, and thus a quorum can't be achieved and writes would be rejected. Reads would be successful via node 3.

However, having a redundant deployment isn't sufficient in many cases. Bitbucket must be aware of availability zones and the replica placement aware of availability zones must be implemented.

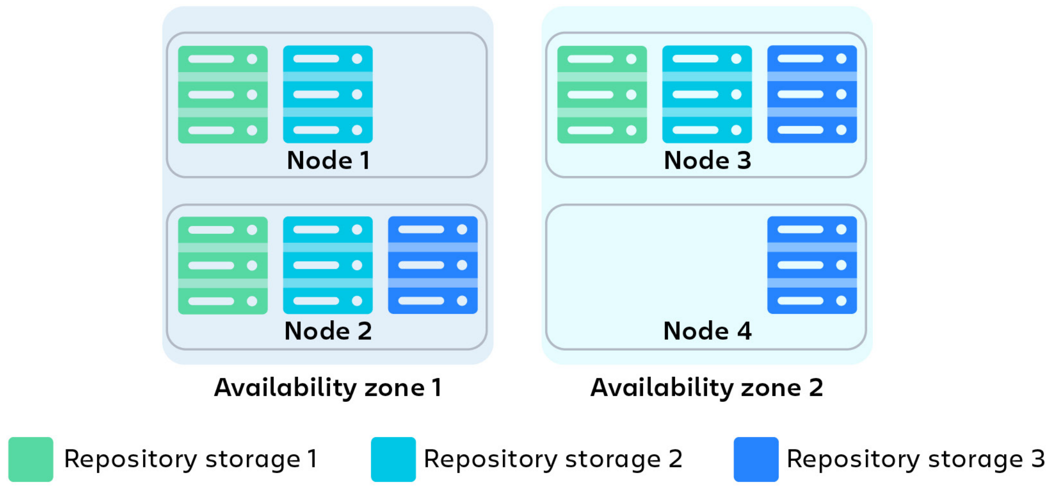


Fig. 11 – Non-redundant multi-availability zone deployment

Example

Take the scenario in Fig. 12. For a replication factor of three, each replica can be placed in a separate availability zone. However, as illustrated, this hasn't happened. An outage in any availability zone will result in one of the three repositories not being able to form a quorum for writing.

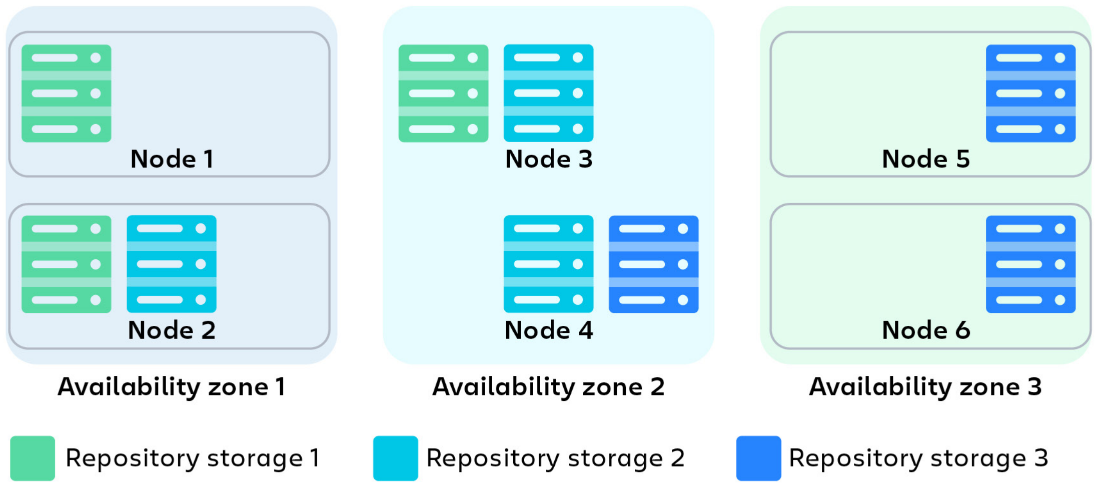


Fig. 12 – Multi availability zone deployment with non-redundant replica placement

Achieving a redundant multi-availability zone deployment must be implemented manually. Even with the most pessimistic replica placement, the loss of a single availability zone would still permit a write quorum to be formed. A simple fix is to increase the replication factor.

Example

The case in Fig. 12 can be made resilient by increasing the replication factor from the default three to at least five.

The simplest approach may be to ensure that each Mesh node resides in its own availability zone, with no other Mesh nodes residing in the same availability zone.

Auto-scaling

Bitbucket Mesh can scale up by adding more nodes and scale down by removing nodes. This can be a desirable characteristic for a Bitbucket deployment since the load is often very spiky. These spikes occur due to the load from build systems that can execute hundreds of build jobs in response to a change being pushed to Bitbucket. These build jobs can result in hundreds of Git clones or fetch requests almost simultaneously.

In many cloud environments, it's desirable to implement automatic scaling or auto-scaling. This is a method of scaling up and down automatically, based on continuous monitoring of the load combined with some logic that decides when to add or remove nodes.

Auto-scaling works well for mostly stateless applications. However, Mesh is very stateful by definition. Adding a Mesh node so that it can service requests involves rebalancing, as mentioned above. This process migrates some replicas from existing Mesh nodes to new Mesh nodes. Likewise, deleting a Mesh node also involves rebalancing, where replicas are evacuated to the remaining Mesh nodes. So, the configured replication factor is maintained after the node is deleted. These processes can take several minutes or even hours for larger systems. Such timeframes are somewhat incompatible with the demand for auto-scaling because the bursts of traffic that auto-scaling aims to handle have a duration of about five to 20 minutes typically. So, by the time a Mesh node is available to service requests, the spike may have subsided.

Furthermore, when adding a new node, populating it with repository replicas places a load on the existing nodes, as these are the source of the data being copied. So, right at the moment, the system is trying to better handle a load spike while replication taxes the system, reducing its capacity to handle user-driven requests. Consequently, it's unlikely that fine-grained auto-scaling would be beneficial, so it wasn't a design goal for Bitbucket Mesh.

Conclusion

With Bitbucket Mesh, we look to continue to support development teams as their repositories grow or as they require more reliability and performance.

If you think Bitbucket Mesh is right for your organization, upgrade to Bitbucket 8.0+ and check out the following resources in Atlassian Support:

Bitbucket Mesh

Set up and configure Mesh nodes

Migrate repositories to Bitbucket Mesh

ATLASSIAN