Table of Contents

[Dealing with assets, in general, in the various language repositories 1](#_Toc46830776)

[The solution to look at going forward 1](#_Toc46830777)

[A separate assets repository using sparse-checkout and partial clones to reduce clone times. 1](#_Toc46830778)

[Things that were tried that didn’t make the cut 3](#_Toc46830779)

[Git-LFS 3](#_Toc46830780)

[Azure Blob Storage 4](#_Toc46830781)

[Pulling a zipfile of a repository 5](#_Toc46830782)

# Dealing with assets, in general, in the various language repositories

There are growing concerns with the sizes of the various language repositories regarding items that are not source code. These items are primarily, but not limited to, test assets. In some cases, the size of the assets in the repository are far greater than the product’s code side. Probably the best example of this would be the azure-sdk-for-net repository. The repository’s size on disk is 998 MB with approximately 697 MB dedicated being recording files (Note: This size does not include the .git folder which adds another 1.48 GB. The history in the .git folder is going to effectively be an albatross unless the repositories are moved or recreated).

With these investigations we’re looking into is where to store assets (recording scripts, graphics or video required by tests etc.). The defining principals are:

* **Fetch time** – We need to be able to fetch the required assets (either in whole or in slices) in a timely manner. If the time to fetch the assets is longer than it would take to clone and pull the repository, as it exists today, then that’s considered a failing path.
* **Ease of use** – Whether we provide tooling, scripts, or a small set of commands with guidance, it must be easy and intuitive for anybody to use. Overly complex commands, configurations or crazy setups for an end user are a non-starter.
* **Visibility** – With the main repository being a public git repository it means that anything used to store assets must at least be public for fetching.

## The solution to look at going forward

### A separate assets repository using sparse-checkout and partial clones to reduce clone times.

Git 2.19.0 introduced the concept of [partial clones](https://git-scm.com/docs/partial-clone) and Git 2.25.0 introduced the concept of [sparse-checkout](https://git-scm.com/docs/git-sparse-checkout). In a nutshell partial clone allows Git to function without having a complete copy of the repository and sparse-checkout reduces the checkout to a set of paths given by a list of patterns. The separate assets repository would have to be in a well-known location for tests and infrastructure. *Note: While the minutiae of how exactly this is going to work is still being investigated, from my investigation it seems to make the most sense to partition things out by the area directories under the sdk directory (sdk/<area>).*

#### Why a separate repository for assets?

While we could certainly use partial clone and sparse-checkout with existing language repositories, a user/developer should be able to just pull a repository and be able to build the product. Interdependencies between libraries along with the various engineering systems pieces and infrastructure would have the biggest potential to cause problems if required pieces were missing. Further, the bloat or slower sync times isn’t because of the product code but because test assets and, in most cases, a missing test asset is far easier to identify and recover from.

#### Why not a submodule?

Submodules really can’t do part of a repository. The only way one could effectively trim a submodule would be to clone with depth which would leave things in a detached head state. The goal here is to minimize clone and sync times without negating the ability, or requiring the entire enlistment, to do updates.

#### At a high level what does do the partial clone and sparse-checkout commands look like?

Let’s take the azure-sdk-for-net repository.

git clone --filter=blob:none --no-checkout <https://github.com/Azure/azure-sdk-for-net>

The filter tells git that we’re going to pull down object we choose and not to pull down anything. The no-checkout option is required otherwise git will recognize that it doesn’t have any objects and try to pull them all down from the server. The enlistment directory will contain no files and a minimal .git directory with no objects. At this point we enable sparse checkout by executing the following command

git sparse-checkout init

This command enables sparse checkout (set core.sparseCheckout=true) and will set a default pattern that’ll match the files in the root directory and nothing else. Then, it’s just the matter of setting the directories that you want by executing the following

git sparse-checkout set <path1> <path2> or git sparse-checkout add <path1> <path2>

For example, if I just wanted the Azure.StorageBlob recordings I’d run

git sparse-checkout set git sparse-checkout set sdk/storage/Azure.Storage.Blobs/tests/SessionRecords

then, if I also wanted the records from from Microsoft.Azure.Management.RecoveryServices.Backup I would simply run

git sparse-checkout set git sparse-checkout add sdk/recoveryservices-backup/Microsoft.Azure.Management.RecoveryServices.Backup/tests/SessionRecords/IaasVmE2ETests

at which point I’d have both directories and everything beneath them.

The difference between set and add is that set overwrites the sparse checkout file with just the patterns given whereas add does exactly what you’d think, it appends the pattern to the sparse-checkout file. Git populates the files in the enlistment when a set or add is executed. The resulting enlistment is not in a headless state which means the files can be edited and a PR created though the normal creation process.

#### Pros:

1. Permissions aren’t a problem – It’s still git and whatever permissions we have for the main language repositories would be used for the secondary repository.
2. Syncing slices of the enlistment is faster than syncing the entire enlistment.
3. The resulting ‘slices’ of the enlistment can still have files added/edited or deleted and PRs created.
4. You’re not pulling down the entire repository history

#### Cons:

1. The Assets repositories would be a separate, secondary, repositories to their matching language repositories:
   1. When something is being released both repositories would need to be tagged.
   2. If work is being done in feature/release/hotfix branches of the language repository, we will have to deal with branches of the Assets as well.
   3. PRs can span multiple repositories, just the language repository or just the Assets repository. DevOps changes to deal with these could be problematic as pipeline definitions and triggers only target a single repository.

## Things that were tried that didn’t make the cut

### Git-LFS

This would have been the most ideal solution as all files would still exist in the same repository but the bottom line is that gitLFS just isn’t ready. Results were sent out at the end of March detailing git-LFS but here’s the summary.

1. **There’s** [**no transport level compression**](https://nam06.safelinks.protection.outlook.com/?url=https%3A%2F%2Fgithub.com%2Fgit-lfs%2Fgit-lfs%2Fissues%2F3683&data=02%7C01%7CJames.Suplizio%40microsoft.com%7C426b2983789c44c2596308d7d1ac7570%7C72f988bf86f141af91ab2d7cd011db47%7C1%7C0%7C637208410957976951&sdata=ZSUn8fbuqOh2Bn2Vjzx6Xueg6sau%2FKtFadnegEledbE%3D&reserved=0) - Obviously sending uncompressed files over the wire isn’t going to help checkout times.
2. **The physical drive the enlistment is on seems to matter more when using LFS** -  For example: two machines, both physically in Redmond  and on corpnet, one was a single spinning drive and the other was an array, the clone time difference was negligible (30 vs 22 seconds) but the checkout times were wildly variant (18 seconds vs 2 minutes and 58 seconds)
3. **Distance matters** – Fetching from Redmond produced very different numbers than fetching from Australia. On average clone times were double of what we’d see on corpnet and still around a minute, but the checkout times were horrendous. Non-LFS checkout had an average of 32 seconds where the LFS checkouts were taking an average of 8 minutes. *Note: This was on an SSD, not a spinning disk.*
4. **Git LFS files are pulled individually** - There’s no way to bulk pull everything. The files are pulled over https with a default concurrency of 3 and while this is something that could be tweaked in the lfs config it would potentially help checkout times on an SSD but make the times on a spinning disk worse.
5. **The size on disk size, for us, would initially get worse and then eventually taper off** – Because we wouldn’t be rewriting history we’re still going to have space in .git/objects from the previous versions of these files but with the way LFS works, you’re also going to have .git/lfs/objects. Unlike .git/objects, the .git/lfs/objects will only contain the version that’s been pulled local. If you checkout a version you don’t have, it’ll update the .git/lfs/object with that and the other version will simply be a file with pointer. In the case of the .net repo, which has 6029 recording json files taking up about 671MB, this means that the size on disk would grow by that amount. The reason for this bloat is that there’s [no compression on the LFS folder](https://nam06.safelinks.protection.outlook.com/?url=https%3A%2F%2Fgithub.com%2Fgit-lfs%2Fgit-lfs%2Fissues%2F260&data=02%7C01%7CJames.Suplizio%40microsoft.com%7C426b2983789c44c2596308d7d1ac7570%7C72f988bf86f141af91ab2d7cd011db47%7C1%7C0%7C637208410957986942&sdata=i0uGjbK1A4grAq33I10OA73toN8Q3Vkt95WxB%2FrVoGc%3D&reserved=0).

### Azure Blob Storage

All the recordings from the azure-sdk-for-net repository were moved into Azure Blob Storage in the Azure SDK Developer Playground with the following properties:

* Location – westus
* Account Kind – StorageV2
* SKU Tier- Standard
* Default Access Tier - Hot

*I’m going to go on record and say that, currently, the only effective way to pull large numbers of items out of blob storage is with azcopy. The AZ CLI’s performance was beyond abysmal being roughly 10x slower when pulling the same items. To be fair, the AZ CLI is using the track 1 version of the python libraries which do not have the benefits of the performance work that was done for the track 2 python libraries yet.*

Tests were done from a home connection in Seattle, corpnet and from a home connection in Australia.

#### *Pros*:

1. Size on disk only consists of the files that were pulled. There is no .git folder becoming more and more bloated as versions are added.
2. Pulling files piecemeal for an area or areas is relatively easy. The main benefit is how easy it would be to pull resources for a given area, like batch or keyvault. Most areas took under 5 seconds to pull but there were outliers. For example, Storage was the worst offender taking about 7 seconds to pull on corpnet, 15 seconds from Seattle and 50 seconds from Australia.

Cons:

1. Distance matters – Pulling everything from a machine on corpnet was relatively cheap but pulling the same files from Australia was 2-3x longer than pulling the files from corpnet.
2. Permissions would be a problem - Git on one side, corpnet on the other. For internal developers this is less problematic but external contributors I have no idea how’d we get around this.
3. A single azcopy call isn’t terrible, even if you’re pulling an entire Assets repository (in the US), however the more azcopy calls, the longer it would take to sync. For example, pulling all the assets on my home connection took 50 seconds but if I pulled them an sdk/<area> at a time it took 4 minutes and 13 seconds.
4. Versioning would be painful – We’d have to implement our own versioning scheme and a way to store it in the matching language repository which would have some additional issues to deal with:
   1. Anything other than versioning the full Assets repository would mean 1 call per. The more granular the versioning, the more azcopy calls, the worse the performance gets (see #2).
   2. The main language repository would have to store something that would indicate what the version/versions are to pull the correct versions. This becomes even worse when looking at something still in development: How would pipelines be able to access things that aren’t versioned yet?

The biggest benefit of using Azure Blob Storage is that, unlike git, you’re not dragging down the entire history but everything else, from versioning to fetching files, is decidedly worse.

### Pulling a zipfile of a repository

While it is possible to download a zipfile of a GIT repository it’s not very practical. Downloading the zip is only slightly faster than just syncing the repository but the unpacking of the zip that makes this a non-starter. For example: On my Surface Laptop 2, taking zipfile of azure-sdk-for-net and using powershell’s Expand-Archive took 8:07 to unpack. Using System.IO.Compression.ZipFile’s ExtractToDirectory took 5:06 to unpack. The decompression times alone were enough to end this investigation however even if that could be rectified the result would effectively be a read-only copy of the repository.