

GDC

March 21-25, 2022  
San Francisco, CA

# Efficiently Shuffling Loads of Data from Place to Place

Adrian Astley – Tools Engineer at Activision

#GDC22



# Agenda

- Problem statement
- History / prior approaches
- What we wanted to improve vs prior approaches
- Overview of Indy
- Lessons learned during development and use in production
- What's next / future plans
- Questions

This talk is a postmortem of a tool we created called Indy.

It's designed for bulk data transfer and build distribution.

Here's our agenda:

# Why we need a tool

## Single Studio Development

- One location
- Single fast LAN network

## Large Scale Distributed Development

- Multiple physical locations
- Multiple LAN networks connected by bandwidth-limited WAN

<click> In classic game dev, the game is developed at a single studio, with very fast LAN network

<click> With recent Activision games, this was becoming less and less true. We had one "lead" studio with 2-4 helper studios, and another 2-3 sites of QA testers. In addition, we were simultaneously developing 3 games in leap-frog. Each year, a different game was shipping, with a different "lead" studio.

# Continuous Testing - Compass

GDC 2018 - Automated Testing and Profiling  
for 'Call of Duty'

March 21-25, 2022 | San Francisco, CA #GDC22

GDC

Activision has a continuous testing tool called Compass. My coworker Jan van Valburg did a GDC 2018 talk about it that you can find in GDC Vault.

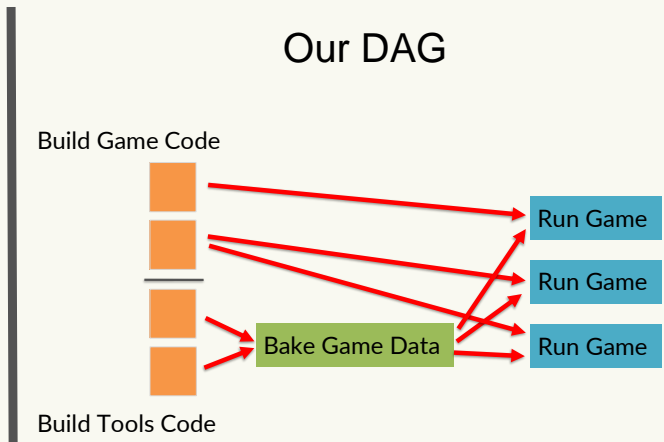
For each check in, Compass will compile the game, bake a test map, and then run the game

# Continuous Testing DAG

## Simple Test

1. Build All Code
2. Bake Game Data
3. Run Game

## Our DAG



In a simple implementation of this, you have a single task run on a single worker. It builds all the code, bakes the game data, and then runs the game.

This works, and it keeps all the artifact data local. However, it doesn't scale very well.

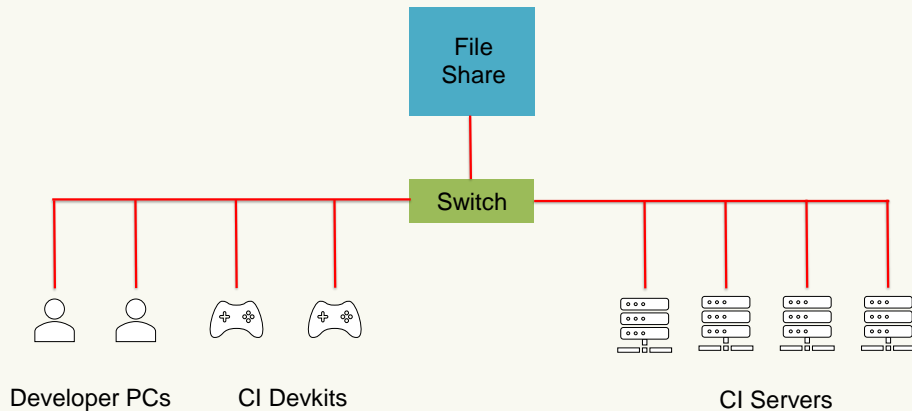
<click>

So we break up the test into multiple tasks that can run on multiple

workers in parallel. This allows us to better utilize our servers, and run tests faster.

However, there is no free lunch. By splitting up the test into multiple tasks, we now have to transfer all the artifacts between servers.

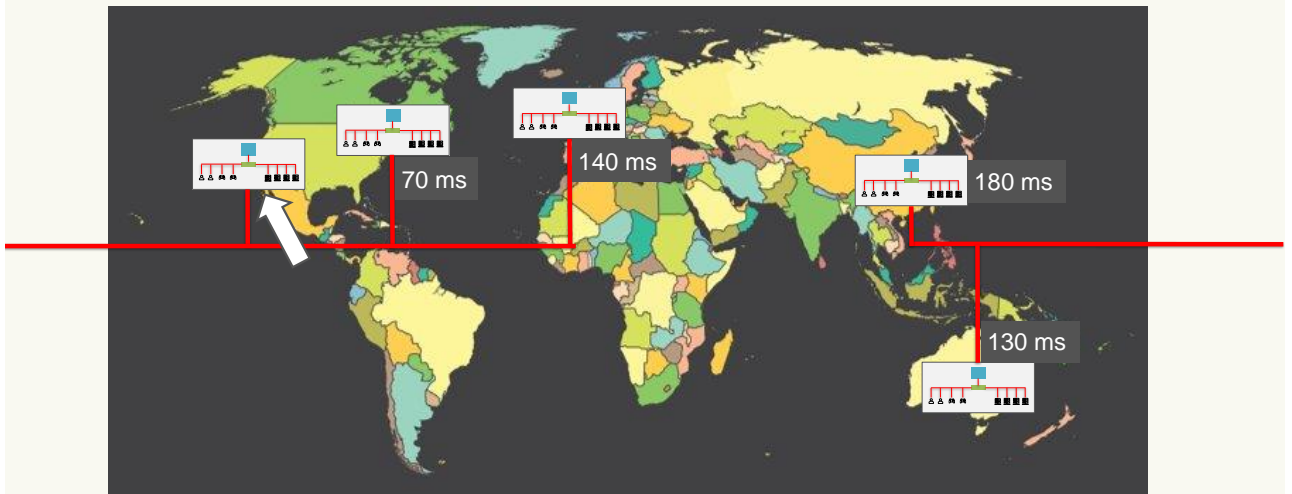
# Simple and Fast Storage



When there is only a single studio, this wasn't a big problem. Compass was originally designed to use a single file share server like SMB or NFS to store the artifacts from each task.

This is simple, and works relatively well when everything is on the same LAN network.

# Latency is Killer



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

However, as mentioned before, our projects are no longer being developed at a single studio. We have multiple studios and multiple QA sites.

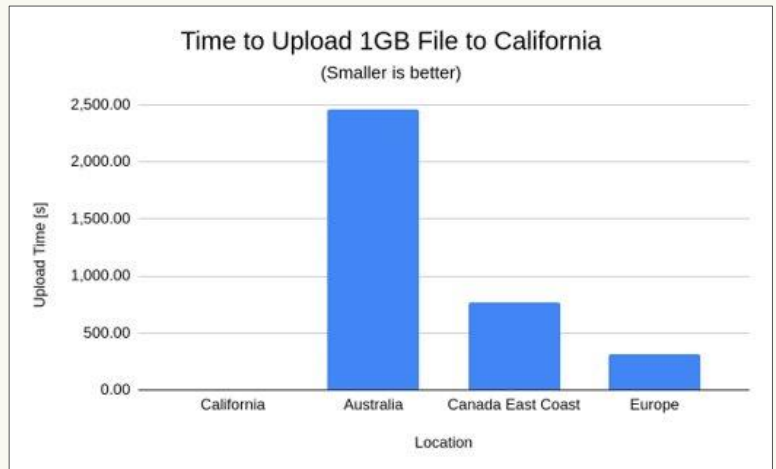
So where do we put the file share? We have studios spread out across the US, in Australia, Shanghai, and QA sites in Europe. If we put it in California <click>, the latencies to the other studios can end up being immense. <click> This isn't



slow network connections. It's just the speed of light. It's just physics

# File Share Performance with High Latency

Location	Upload Time [s]
California	2
Australia	2,463
Canada East Coast	772
Europe	312



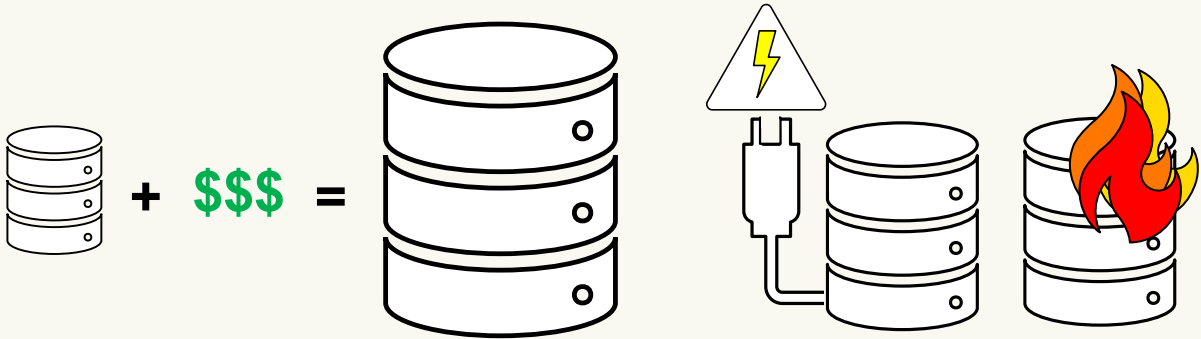
With a single file server, any workers not in the same physical area or network as the file share are guaranteed to have poor performance.

In this graph, I tested transferring a 1GB file containing random data to one of our file shares in California, from various locations. Transferring from within the same data center is only a few seconds. But as the latency increased, the

bandwidth drops drastically. Not just a few seconds more, but hundreds or thousands of seconds more.

So if the file share was in California, and a studio on the East Coast added workers to Compass. Awesome! Yay! More workers to churn through tasks! But actually, the workers ended up being not very useful, because any tasks run on those workers are much slower, for startup / finish just due to artifact transfer bandwidth being much much slower.

# Scaling and Fault Tolerance



The last two issues with file shares are scaling and fault tolerance.

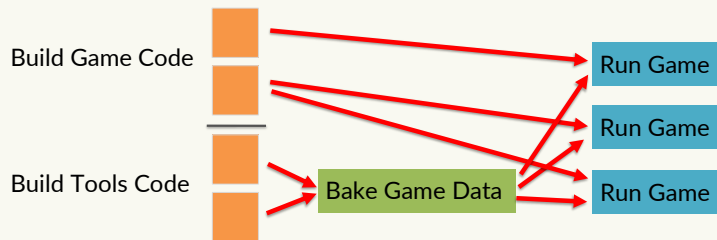
<click> If the current file share can't keep up with load, <click> your only options are to buy a bigger and more powerful server. Or perhaps figure out a way to shard your data across multiple file shares.

But the other issue is that there is no fault tolerance. <click> If the server dies, or needs maintenance, or

someone trips on the power cord, oops, now the entire CI system is having an outage.

NOTES: Maybe talk a bit more about the fact that maintenance happens a lot. Even with hotswapping. And failures happen a lot

# Deduplication

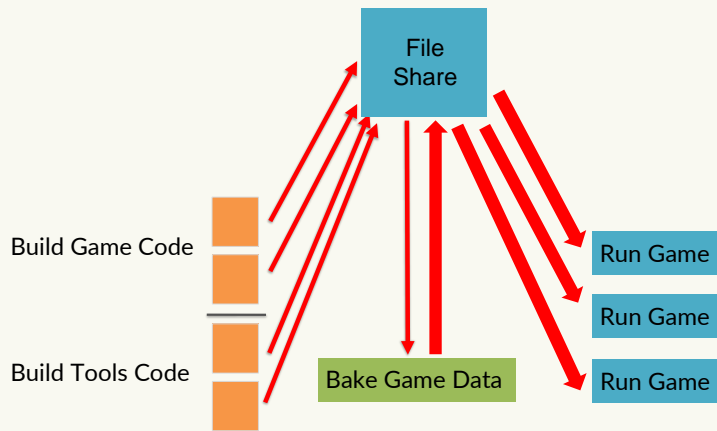


Lastly, let's talk about deduplication of data. <click> This is the task DAG that we talked about a few slides ago. The arrows show the dependencies of the tasks

Let's instead swap the arrows to show the artifact data flow

<click>

# Deduplication



The code build tasks will upload their compiled executables to the file share  
<click>

Then the Bake game task will download those, <click> and use it to bake the game maps. <click>

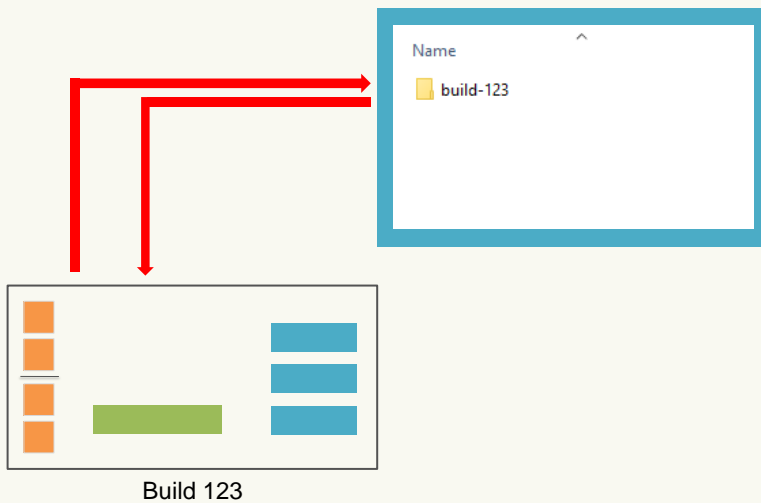
When finished it will upload the 70GB of map data to the file share. <click>

And finally each run game test will download the game executable

\*and\* the 70GB of data from the file share.



# Deduplication



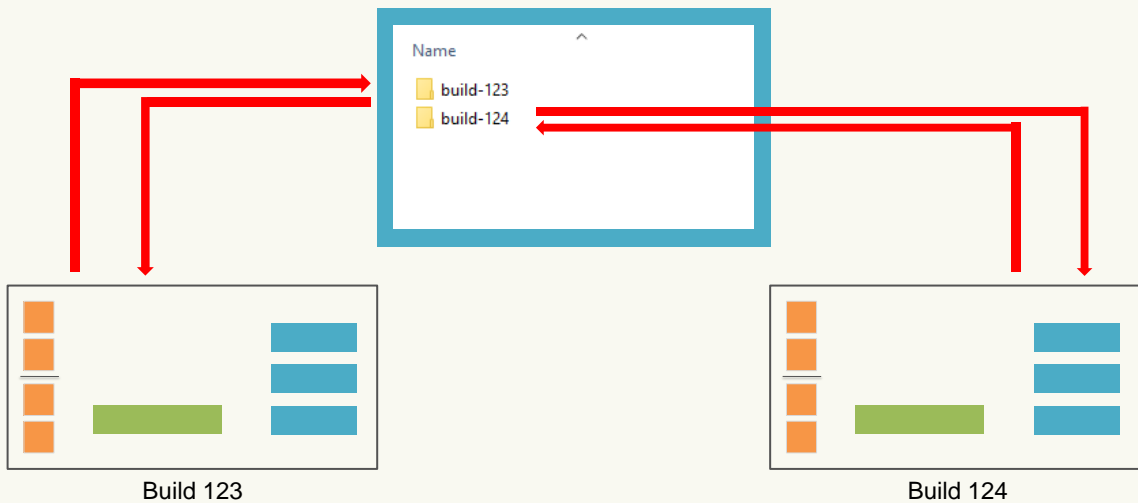
In the original Compass setup, each “build” stored its artifacts in a new folder on the file share.

<click>

So one build will bake all the maps, and store the 70GB of data in a folder.

<click>

# Deduplication



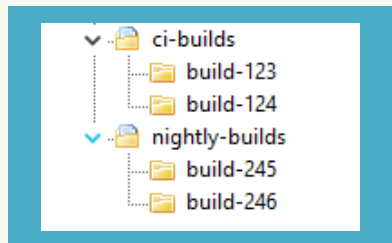
Then the next build will bake all the maps, and store *its* 70GB of data in a new folder.

Even if the only change between the two builds was a single texture, we had to transfer all 70GB *to* the file share for storing, and then transfer the whole 70GB *from* the file share when we ran the game test.

As an addendum, file shares *can* and *do* have block-level

deduplication on the server itself. So the 70GB of data *\*may\** be deduplicated somewhat in the server storage. But we still need to *\*transfer\** all the data to and from the server. Storage is cheap, but bandwidth is a precious commodity. So, it's bandwidth savings that we care about the most.

# Deduplication



One last thing is that in the previous slides, we were only looking at the artifacts from a single build type. We also have other builds, like a nightly build that builds ALL the maps, instead of just 1-3.

Again, there will be a huge amount of data that will be identical, not only between builds of the same type, but also builds of different types.

So it's clear that we need a solution that can generically deduplicate data,

no matter what the source of the data is.

# Deduplication

Our games are really big!

The 70GB number I used isn't just a random number. Our games and maps are really big.

A CI run might bake 1 to 3 maps. We don't do the hardcore compression like we do for ship, because that takes too long.

So the data output from the baking can be anywhere from 70-100GB, depending on the maps included.

That's a lot of data!

$$\begin{array}{ccccccc}
 \text{Commits per hour} & & \text{GB} & & \text{Platforms} & & \text{Per hour} & & \text{Per day} \\
 10 * 70 * 5 = 3.5 \text{ TB} = 84 \text{ TB}
 \end{array}$$

$$\begin{array}{ccccccccccc}
 \text{Commits per hour} & & \text{GB} & & \text{Platforms} & & \text{Maps} & & \text{Configs per Map} & & \text{Per hour} & & \text{Per day} \\
 10 * 70 * 5 * 3 * 2 = 21 \text{ TB} = 504 \text{ TB}
 \end{array}$$

Let's do some quick back of the envelope math

- Let's assume there are 10 commits per hour. <click>
- Each CI run builds 3 maps, totaling 70GB. <click>
- Let's assume we build for PC, XB3, XB4, PS4, and PS5. That's 5 platforms <click>

That means we're generating 3.5 TB of data per hour. <click> And 84 TB

of data per day. <click>

But that's only how much we generate.

We still need to use this data.

So let's do some more math for how much data is \*downloaded\* from the server.

- We again assume 10 commits per hour, 70GB of data, and 5 platforms <click>
- For every build we test 3 maps <click>
- And for each map, we run two tests, one for profiling, and one for logging, screenshots, etc. <click>

If we total this up <click>, for each build we're downloading 21TB of data per hour, and <click> 504 TB of data per day.



~ 8 Gbps upload to file share

~ 45 Gbps download from file share

That's a lot of data!

Looking from the file share's standpoint, that's about 8 Gbps sustained upload to the server. And about 45 Gbps sustained download from the server.

And this is only looking at the main CI. We also have presubmit builds, manual builds, package builds, and builds on a release branch. So in reality, the upload and download numbers would more than likely be 2

or 3 times higher.

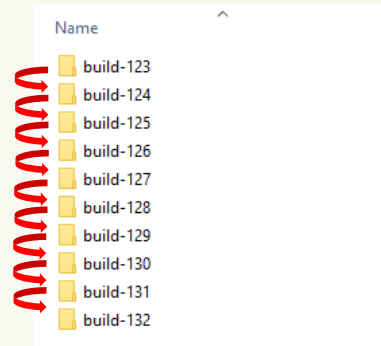
If this was all on the same LAN, it \*might\* be possible with a really fast networking setup. But given that we have workers spread across the world, supporting this amount of data bandwidth across the WAN is not feasible or cost effective.

NOTE: Maybe dedicated slide – Really hammer home that this is just CI. Real life is 3x or 4x higher.

# Clean Up

Don't

Script that walks the filesystem



The last problem we wanted to tackle was clean up. With traditional ad-hoc artifact systems, there are generally two approaches:

<click> First, just don't do any clean up all all.

You may laugh, but if you have semi-infinite storage or storage is cheap and the amount of data you're storing is small, this is actually a decent plan. It's very simple, and you can just nuke everything when the

project is done. However, as we saw in the previous slides, the amount of data we generate is not small. It would fill up any system in a week or less. Also, as an industry, we are quickly moving to a world where a project is never done; it continues on, with new content, and new patches.

<click> The second approach is usually some kind of script that walks the filesystem. This "works". But it has a number of drawbacks:

First, walking the filesystem itself could take hours. Artifact systems have the ability to generate a huge number of files. Therefore, it's very probable that the file system could have trillions of files for this script to walk through. While filesystems *\*can\** handle that number of files, walking the iNodes is not going to be fast, no matter what filesystem implementation that you use.

So, this clean up has the potential to death spiral. Where CI is adding files faster than the clean up script can index and delete them.

Lastly, as we found out in the previous slides, deduplication is very important. But if you have some kind of strategy for deduplication, how do you find out when a file is no longer referenced? Say you index all the files, and are able to do re-counting. How do you prevent the race condition where the indexing finds an unreferenced file, but in the time it took for the indexing to happen, a new reference was added, so that file \*shouldn't\* be deleted.

So, it's clear that any clean up system that we create must be an integral part of the artifact system as a whole, rather than a script tacked on at the end.

# History / Prior Tools and Approaches

March 21-25, 2022 | San Francisco, CA #GDC22

GDC

Hopefully now, you all can understand the problem we had. So, now we'll move on to what existing tools and approaches were available when we first started. And exploring their pros and cons.

# Packaging Tools

Installers

Compression Tools

Container Systems

First, we have packaging tools. And these can be broken down into a few categories.

First, installers. For example, the Windows installer (msi) and all the Linux various install formats (.deb, .rpm, etc.)

- These bundle a group of files with a small script for where to put the files when installing and what files to remove when uninstalling
- Some of the formats' install

scripts have optimizations around “don’t copy the a file, if the existing already exists”

- But there is no download deduplication. You have to download the entire .msi / .deb bundle

Next we have compression tools. Like zip, tar, and 7zip

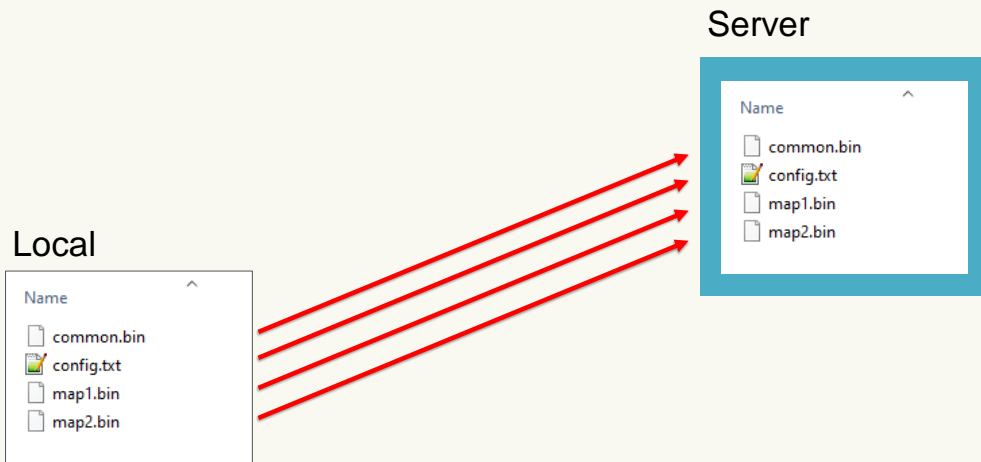
- These have the same arguments as installers, but now we don’t have *\*any\** deduplication

Finally, we have container systems, like Docker

- Docker is mostly designed around process isolation
- That said, it also has nice file isolation / deduplication properties
- Each line in the Dockerfile is a “layer”. All files changed or modified in that layer are grouped, hashed, and thus can be deduplicated between images
- So, when you “pull” an image, it only downloads the layers that aren’t already downloaded



# Data Transfer Tools



There are a bunch of open source data transfer tools. Or some built-in, like Microsoft's robocopy.

You give them a source directory <click>, and a destination file share path <click> and they will copy up files to make them identical. <click> They have really great algorithms for parallel transfer to make this really fast. <click>

However, they have a few

drawbacks.

First, as we know from the previous slides, the fastest transfer is the one you don't have to do.

The tools can attempt to figure which files are identical, the “cheap” way by comparing file sizes. But this isn't accurate. If a single texture is re-colored, the file size will be the same, but the content is completely different. Some tools can additionally attempt to use file modification timestamps as an additional piece of comparison. But these also aren't super accurate at the Operating System level.

The *\*best\** way to do this comparison is to compare hashes. However, these tools are all client-side. To calculate the hash, we would need to download the file from the server, which defeats the whole purpose.

Next, these tools are all optimized around “just” the transfer. Not the long term state after the transfer

That is, it’s up to you the user to organize / deduplicate any local “builds” that you have downloaded. Similarly, it's up to you to try an organize the files on the server so they can be deduplicated.

Lastly, these tools are all all reliant on a filesystem. That is, the destination has to be a file share of some kind, which as we saw before, doesn’t scale

# Data Transfer Tools

## Pros

- Good algorithms for parallel / fast data transfer

## Cons

- All client-side
  - Clean up is hard
  - Hard to have exact deduplication
- Deduplication is at the “whole file” level

So here's the final list of pros and cons.

Historically, at Activision we used a mix of things.

One project hashed their files, and robocopied them to a central file share that served them downstream over HTTP.

But they forgot to add cleanup, and the script they added on at the end

death spiraled. Oops.

Another project sharded their work across multiple file shares, so there was a smaller blast radius.

But they still had all the downsides of file share performance and scaling.

## Goals for the new tool

- Single binary CLI tool
- Data deduplication, with file chunking
- Automatic clean-up of old data
- The server components must be horizontally scalable and fault tolerant
- As much as makes sense, utilize the hard work done by the web community

So, now that we understand the problem statement and we've explored the pros and cons of existing tools, let's go over our goals for our new tool.

<click> First, we want the tool to be a single binary. As much as possible, it should be statically linked. So someone can download just the single file and be able to run it without any dependencies.

<click> Next, the tool needs to support data deduplication. The tool must be able to break files up into “chunks”, because deduplication at the whole file level will not work due to our large map file sizes.

<click> Next, the system must support automatic clean up of old data. Both locally and on the server.

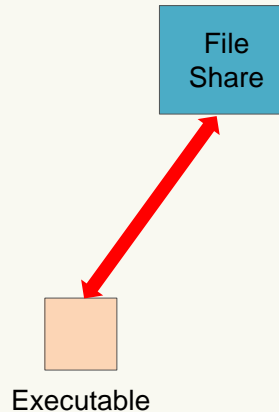
<click> The server components must be horizontally scalable and fault tolerant. Especially storage

<click> Finally, as a “bonus” goal, as much as makes sense, the system should utilize the hard work done by the web community. The technology of the web has exploded in the last decade. The people in that ecosystem are exceptionally smart and are solving very difficult problems similar to us. So rather than trying to re-invent what they've done, we should utilize their discoveries

and tools.



# Let's Design a New Tool



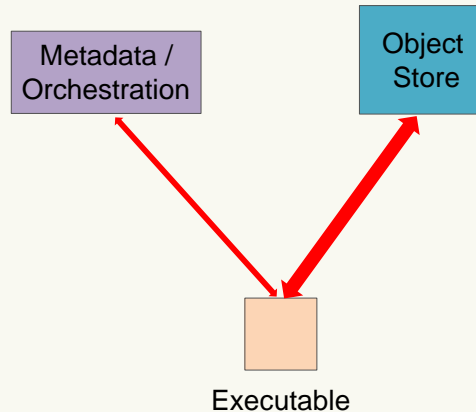
So given those goals, let's think about what we need.

First, the obvious one, we'll want a local command line program. <click> This will do all the "heavy lifting" for a user. It will have a nice clean set of commands, that can also be used by scripting.

The tool will upload to a remote server. <click> But we've learned that file shares aren't great. So let's

swap out the file server for an HTTP  
Object Store. <click>

# Let's Design a New Tool



Great. HTTP is highly scalable, works across all platforms, and in the last two decades, there has been a huge amount of tools and infrastructure work by the community to make HTTP fast and easy.

Lastly, we learned from file transfer tools like robocopy that we need some kind of server component <click>

This will orchestrate data deduplication and data cleanup.

And that's basically what we did!

# Introducing Indy

- Local binary - indy.exe
- Metadata server - Ark
- Object Storage Server - WebDAV

So with no further ado, let me introduce our tool called Indy.

The Indy ecosystem is made up of three components:

- 1.The indy binary itself
- 2.The metadata server called Ark
- 3.An Object Storage Server

We'll explore each of these in a series of examples

# Using Golang

- Familiar syntax to C
- Native cross-compiling
- Fantastic standard library
- For everything else, there's probably a library for that
  - Metrics / telemetry
  - Web server path routing
  - Terminal Progress bars
  - Etc

As a very small tangent, we chose to use golang to create our tool and server components.

This was for a few reasons:

<click> First, golang has a very familiar C-like syntax, which made it very easy for co-workers to learn

<click> Second, it has native cross-compiling for all the major operating systems

<click> Third, it has a fantastic

standard library. Http, sha1 hashing, string manipulation, all included.

<click> And for anything that *\*isn't\** included in the standard library, there's probably an open source library for it.

For example:

- Metrics and telemetry
- Fancier web path routing
- Terminal progress bars
- And more

# Overview of Indy

- Everything is structured an "image"
- An image is:
  - Bundle of files
  - Optional ENV variables
  - Optional commands to be run with files / ENV

Back to Indy.

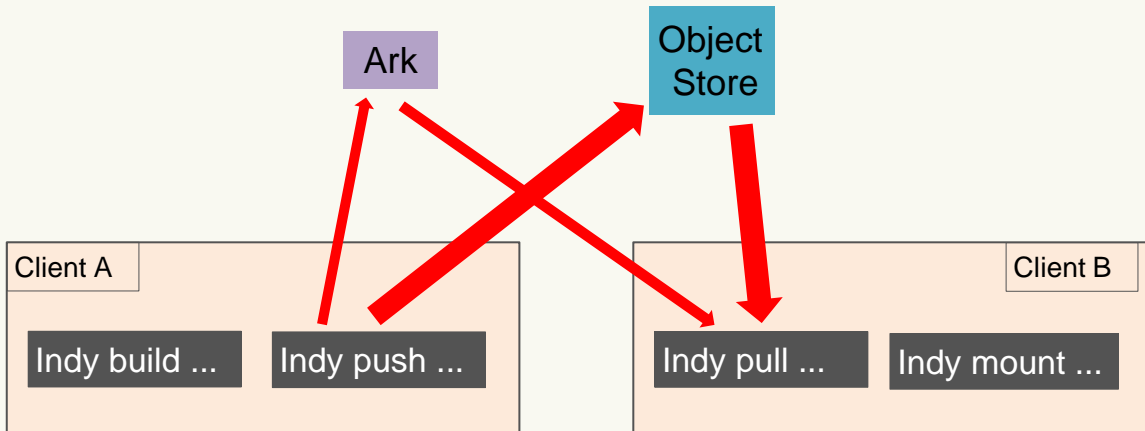
<click> In the Indy ecosystem, everything is structured around an "image".

<click> An image is a bundle of files with optional ENV variables and commands to be run with those files.



# The Image "Flow"

Ark - Metadata  
Object Store - Storage



Let's look at a high level diagram showing how images are created, moved around, and used.

We have two clients, A and B. And we have the Ark metadata server, and the Object Store.

<click> First, client A creates an image by using the `indy build` command. I'll go into more details in a second how this works. But for now, just think of build as bundling

up all the files into a black box. Client A wants to share the image with other machines. <click> To do that, client A uses the `indy push` command. This will contact Ark to figure out what data it needs to upload, and then will upload that in parallel to the Object Store.

So now Client B wants to download the image. <click> It does this by using the `indy pull` command. This will contact Ark to figure out which data it needs to download, and then download that in parallel from the Object Store.

Finally, Client B has an image. But an image is a bit of a black box. You can't use it directly. First you need to create an instance of the image. <click> You do this with `indy mount`.

At a high level, this doesn't seem much

different from the file transfer tools we're intending to be improving upon. So let's go back now, and look at each of the indy commands in more details and see how they work.

# A Very Simple Indyfile

```
ADD      map1.bin /maps/map1.bin
CHUNK 1M map2.bin /maps/map2.bin
CHUNK 1M map3.bin /maps/map3.bin
```

Let's start with image building. To do this, you first create an Indyfile. This is a custom declarative language that defines what you want in your image.

If you want to add a file you can use the `ADD` instruction. This will include the entire file as a single chunk.

Alternatively, you can use the `CHUNK` instruction, to tell Indy to split the file up into chunks. In this

example, it will split it into fixed-size 1MB chunks.

Now that we have the Indyfile, we can call ``indy build`` <click>

# Building an Indy Image

```
aastley@aastley-ubu:gdc $
```

```
<namespace>/<name>:<tag>
```

March 21-25, 2022 | San Francisco, CA #GDC22

GDC

Indy will read in the instructions in the Indyfile and use them to create an image.

The build command outputs a sha1 which uniquely identifies the image. However, it's highly inconvenient to have to reference images by sha1. So, we can create a label that is a human consumable, which acts as a pointer to the image sha1.

In the `indy build` command, you'll

notice we specified the label  
`aastley/gdc-test:v1`.

<click> Labels are of the form  
namespace, name, tag

# How Does Build Work?

```
ADD      map1.bin /maps/map1.bin  
CHUNK 1M map2.bin /maps/map2.bin  
CHUNK 1M map3.bin /maps/map3.bin
```

Let's talk a little bit more about what build does and what an indy image actually is.

When we call `indy build`, indy will use the Indyfile instructions to create an image manifest. It looks something like this <click>



```
{
  "Version": 0,
  "Size": 780000000,
  "Env": [],
  "Files": [
    {
      "Path": "/maps/map2.bin",
      "Sha1": "1bb7c068c4e162b0b82c3972c87119a49036d8f0",
      "Size": 248000000,
      "Chunks": [
        {
          "Sha1": "1bf0a17e0709a295efd8af3b33a5316bd65ddfcf",
          "Size": 1048576
        },
        {
          "Sha1": "e68641bba66f6b6dff18af14f1dlc607d8e757f1",
          "Size": 1048576
        },
        ...
      ]
    },
    ...
  ]
}
```

The manifest is a simple json file, which lists all the files in the image, and for each file, all the chunks that make it up.

For those reading this after the presentation, the manifest is highly truncated in order to fit in onto the slide. A "real" manifest has much more information in it.

Everything in the Indy ecosystem is

content addressable. That is, the name and identifier of everything is the sha1 hash of its content.

So here in the manifest, we list all of those hashes, plus the sizes of everything.

For example, the map2.bin file has the hash `1bb7`, and a size of 248 MB. It's made up of many many chunks. The first two are listed here, with sizes of 1MB each, and each have their own hashes.

# Local Indy Image Storage

```
> tree ~/.indy
.
├── images
│   └── 4d002477d3aa6bfab47d34f58aa98451fe96bb8b <----- json manifest file, filename=sha1
├── mounts
├── objects
│   ├── 03
│   │   └── 034a2bfcd34b70e279730dled4c055d0f93376b3 <----- binary blob, filename=sha1
│   ├── 1c
│   │   └── 1c755788fa01e5f4fe480e3f54e68e6597f7f0f6
│   ├── 1e
│   │   └── 1e558dfbcbe27afe1f1c9c6a50f423b812d9d456
│   ├── 1f
│   │   └── 1fa4c43eeab6a68997d80af4118592296650546f
│   └── ----- (abbreviated for slides)
│       └── ff
│           └── ff6b440094fe85ebfecdd493f5f9749fe9ab40663
└── repo.yaml <----- mapping of labels to image sha1
```

The indy build process walks through each file, and calculates the hashes mentioned in the last slide.

As it does so, it copies the chunks into the local Indy Image storage. Specifically into the "objects" folder.

All the files are named by their hash. This means that everything is immutable. A file in the Indy image storage can never change, since that would change the hash and thus the file name would also change. This is

really important and one of the key points of indy. We'll get more into this in a second.

The image manifests from the last slide are stored in the images folder, again named by the sha1 hash of the manifest itself

# Adding a new Label

```
aastley@aastley-ubu:gdc $ indy build --label aastley/gdc-test:v1
--- [=====] 100% 780 MB / 780 MB [3 files 727 chunks]
aastley/gdc-test:v1 [0bff14941e7c582e979fc740ea5a7fc1688eb524]
aastley@aastley-ubu:gdc $ indy images
  LABEL      | IMAGE      | CREATED      | TTL      | SIZE      | FILES      | CHUNKS      |
+-----+-----+-----+-----+-----+-----+-----+
aastley/gdc-test:v1 | 0bff149 | 2 seconds ago | infinite | 780 MB | 3 | 727
```

Let's dig more into labels. As mentioned before, labels are human-accessible pointers to an image. In this example, the label `aastley/gdc-test:v1` points at image `0bff1`. Since labels are just pointers, we are free to add as many as we like referencing the same image.

<click>

# Adding a new Label

```
aastley@aastley-ubu:gdc $ indy build --label aastley/gdc-test:v1
--- [=====] 100% 780 MB / 780 MB [3 files 727 chunks]
aastley/gdc-test:v1 [0bfff14941e7c582e979fc740ea5a7fc1688eb524]
aastley@aastley-ubu:gdc $ indy images
  LABEL      | IMAGE      | CREATED      | TTL      | SIZE      | FILES      | CHUNKS      |
+-----+-----+-----+-----+-----+-----+-----+
aastley/gdc-test:v1 | 0bfff149 | 2 seconds ago | infinite | 780 MB | 3 | 727
```

Here we add a new label  
`aastley/my-awesome-label:123`.

If we run `indy images` to list all the images we have locally, we can see the new label we created. Note that they're both pointing to the same image sha1.

# Image Time to Live

```
aastley@aastley-ubu:gdc $ indy build --label aastley/gdc-test:v1
--- [=====] 100% 780 MB / 780 MB [3 files 727 chunks]
aastley/gdc-test:v1 [0bfff14941e7c582e979fc740ea5a7fc1688eb524]
aastley@aastley-ubu:gdc $ indy images
  LABEL | IMAGE | CREATED | TTL | SIZE | FILES | CHUNKS |
+-----+-----+-----+-----+-----+-----+-----+
aastley/gdc-test:v1 | 0bfff149 | 2 seconds ago | infinite | 780 MB | 3 | 727 |
aastley@aastley-ubu:gdc $ indy label aastley/gdc-test:v1 aastley/my-awesome-label:123
INFO Successfully created label 'aastley/gdc-test:v1' -> 'aastley/my-awesome-label:123' with TTL: infinite
aastley@aastley-ubu:gdc $ indy images
  LABEL | IMAGE | CREATED | TTL | SIZE | FILES | CHUNKS |
+-----+-----+-----+-----+-----+-----+-----+
aastley/gdc-test:v1 | 0bfff149 | 21 seconds ago | infinite | 780 MB | 3 | 727 |
aastley/my-awesome-label:123 | 0bfff149 | 1 second ago | infinite | 780 MB | 3 | 727 |
aastley@aastley-ubu:gdc $
```

Labels have one more feature. When you create them, you can set a Time to Live. This represents a time in the future when the label will expire. In the labels we have created so far, the TTL is infinite.

Let's update the label we just created to have a TTL

# Image Time to Live

```
aastley@aastley-ubu:gdc $ indy build --label aastley/gdc-test:v1
--- [=====] 100% 780 MB / 780 MB [3 files 727 chunks]
aastley/gdc-test:v1 [0bfff14941e7c582e979fc740ea5a7fc1688eb524]
aastley@aastley-ubu:gdc $ indy images
  LABEL | IMAGE | CREATED | TTL | SIZE | FILES | CHUNKS |
+-----+-----+-----+-----+-----+-----+-----+
aastley/gdc-test:v1 | 0bfff149 | 2 seconds ago | infinite | 780 MB | 3 | 727 |
aastley@aastley-ubu:gdc $ indy label aastley/gdc-test:v1 aastley/my-awesome-label:123
INFO Successfully created label 'aastley/gdc-test:v1' -> 'aastley/my-awesome-label:123' with TTL: infinite
aastley@aastley-ubu:gdc $ indy images
  LABEL | IMAGE | CREATED | TTL | SIZE | FILES | CHUNKS |
+-----+-----+-----+-----+-----+-----+-----+
aastley/gdc-test:v1 | 0bfff149 | 21 seconds ago | infinite | 780 MB | 3 | 727 |
aastley/my-awesome-label:123 | 0bfff149 | 1 second ago | infinite | 780 MB | 3 | 727 |
aastley@aastley-ubu:gdc $
```

We'll set the TTL to 20 seconds. If we then run the `indy images` command, it shows that the label has 18 seconds left.

So now that we've set a TTL, what happens when a label expires?

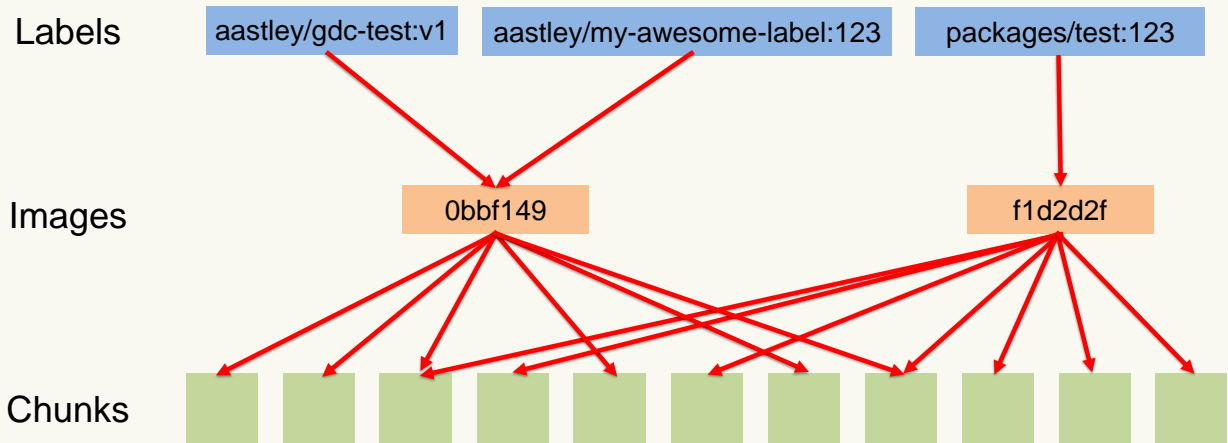


# Image Time to Live

```
aastley/gdc-test:v1 [0bfff14941e7c582e979fc740ea5a7fc1688eb524]
aastley@aastley-ubu:gdc $ indy images
+-----+-----+-----+-----+-----+-----+
| LABEL | IMAGE | CREATED | TTL | SIZE | FILES | CHUNKS |
+-----+-----+-----+-----+-----+-----+
| aastley/gdc-test:v1 | 0bfff149 | 2 seconds ago | infinite | 780 MB | 3 | 727 |
aastley@aastley-ubu:gdc $ indy label aastley/gdc-test:v1 aastley/my-awesome-label:123
INFO Successfully created label `aastley/gdc-test:v1` -> `aastley/my-awesome-label:123` with TTL: infinite
aastley@aastley-ubu:gdc $ indy images
+-----+-----+-----+-----+-----+-----+
| LABEL | IMAGE | CREATED | TTL | SIZE | FILES | CHUNKS |
+-----+-----+-----+-----+-----+-----+
| aastley/gdc-test:v1 | 0bfff149 | 21 seconds ago | infinite | 780 MB | 3 | 727 |
| aastley/my-awesome-label:123 | 0bfff149 | 1 second ago | infinite | 780 MB | 3 | 727 |
aastley@aastley-ubu:gdc $ indy label aastley/my-awesome-label:123 aastley/my-awesome-label:123 --force --ttl 20
INFO Successfully created label `aastley/my-awesome-label:123` -> `aastley/my-awesome-label:123` with TTL: 20 seconds
aastley@aastley-ubu:gdc $ indy images
+-----+-----+-----+-----+-----+-----+
| LABEL | IMAGE | CREATED | TTL | SIZE | FILES | CHUNKS |
+-----+-----+-----+-----+-----+-----+
| aastley/gdc-test:v1 | 0bfff149 | 41 seconds ago | infinite | 780 MB | 3 | 727 |
| aastley/my-awesome-label:123 | 0bfff149 | 1 second ago | 18 seconds | 780 MB | 3 | 727 |
aastley@aastley-ubu:gdc $
```

Nothing immediately. If we run `indy images`, now the image is marked as expired.  
However, this is where clean up comes in.

# Garbage Collection



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

Indy has a command ``gc``, aka garbage collection. To understand how it works, let's look at how indy tracks each component

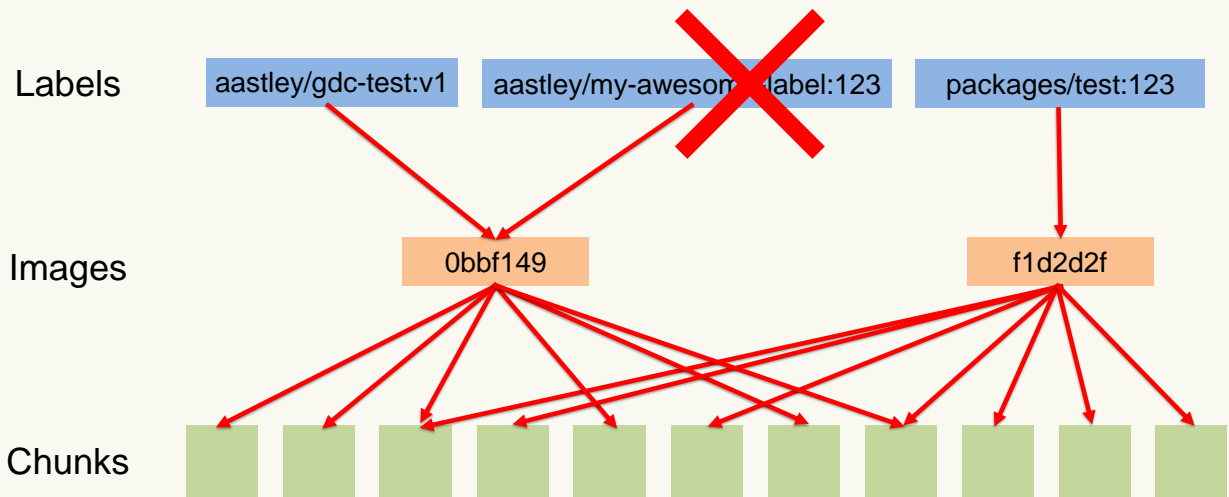
<click> The Indy system has a hierarchy of references. At the top there are the labels

<click> These reference specific image SHA1's

<click> The images, in turn, reference individual chunks. Note, that chunks can be referenced by

multiple images.

# Garbage Collection



March 21-25, 2022 | San Francisco, CA #GDC22

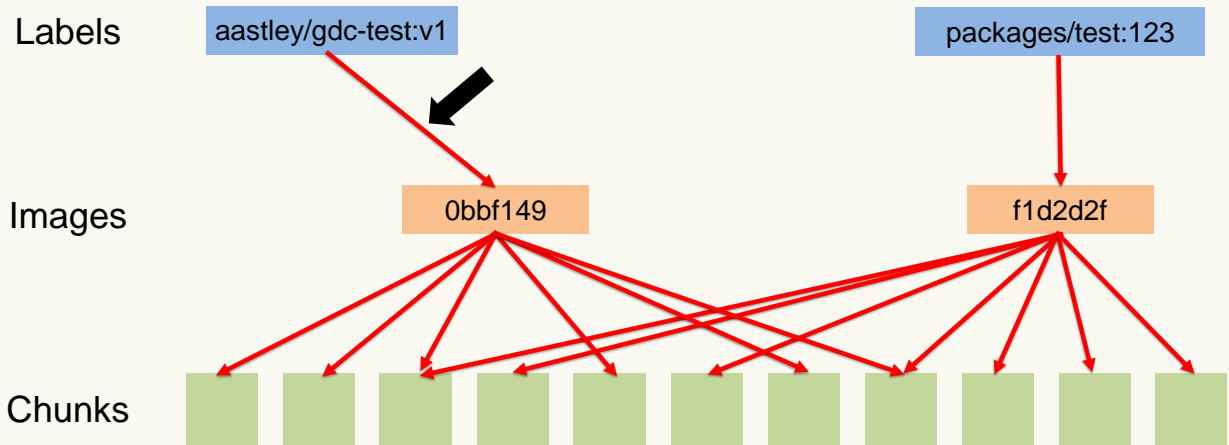
GDC

So back to Garbage Collection.

GC happens in 3 phases. First, we check if any labels have expired TTLs, and if so remove them.

<click> In the previous slides, we saw the `aastley/my-awesome-label:123` label was expired. So let's follow along with what GC would do and remove it.  
<click>

# Garbage Collection



March 21-25, 2022 | San Francisco, CA #GDC22

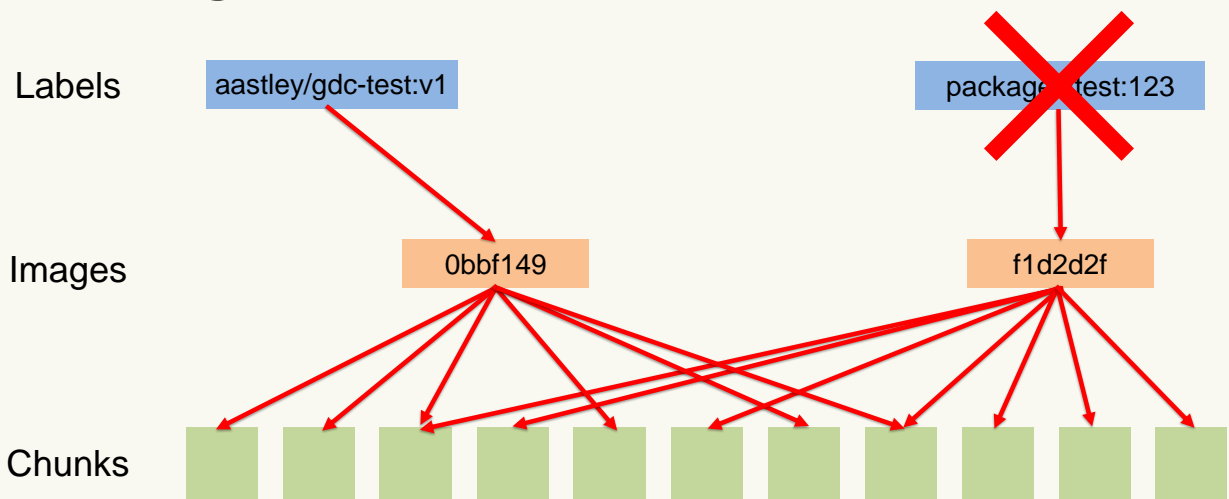
GDC

The next phase is to check all the images, and see if any are now unreferenced by any labels.

<click> In this case, the 0bbf149 is still referenced by the aastley/gdc-test:v1 label. So nothing happens.

Moving on the final phase. <click>

# Garbage Collection



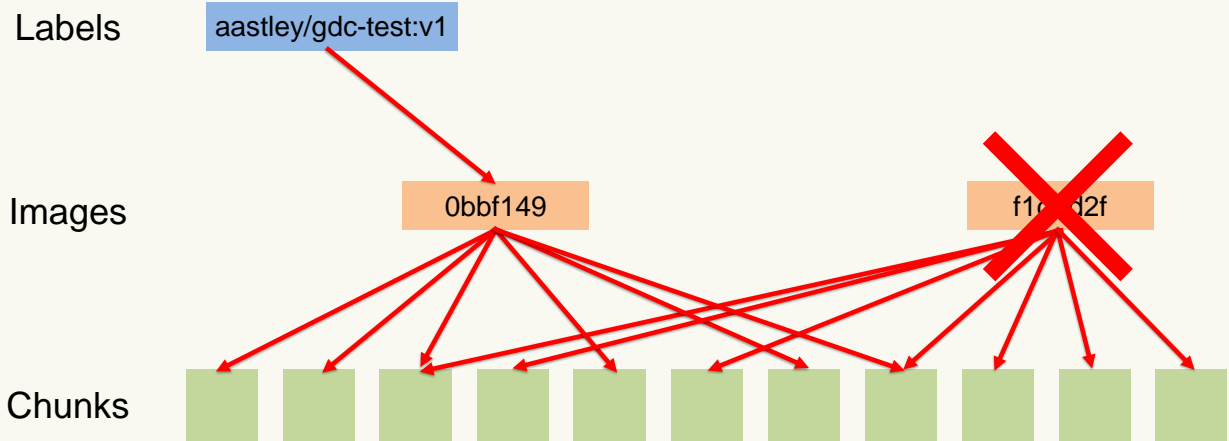
March 21-25, 2022 | San Francisco, CA #GDC22

GDC

In this phase, we reference count all the chunks, to see if any chunks are no longer referenced by any images. Again, in this example, we didn't remove any images. So all the chunks are still referenced.

So let's do one more example, so we *can* see the chunk deletion.  
<click> We'll simulate manually deleting the packages/test:123 label, and then running gc again.

# Garbage Collection



March 21-25, 2022 | San Francisco, CA #GDC22

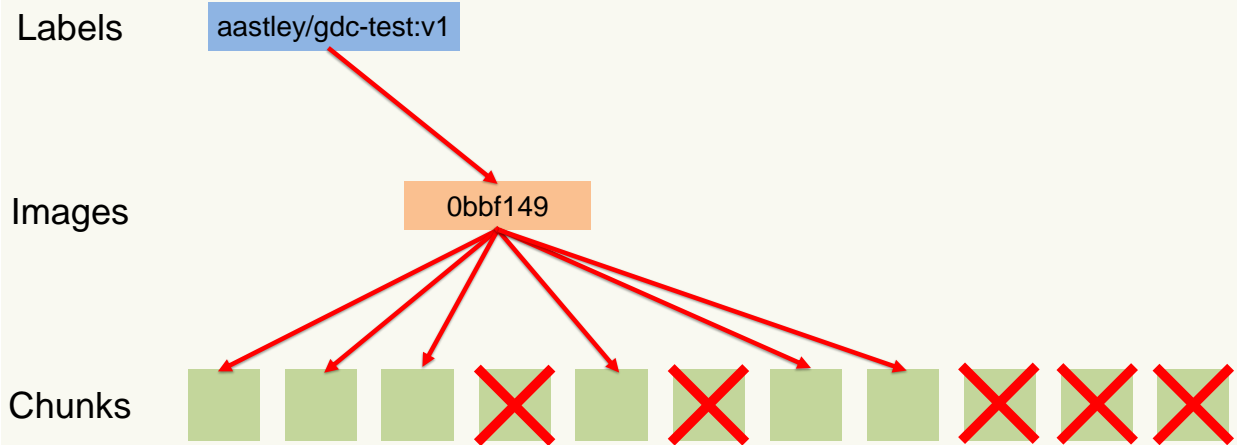
GDC

Phase 1: Remove all expired labels.  
There's nothing expired here, so we can move on.

Phase 2: Remove all unreferenced images.

<click> The `f1d2d` image is no longer referenced by any labels, so we remove it

# Garbage Collection



Phase 3: Remove all unreferenced chunks

<click> We find that these 5 chunks are no longer referenced and can be deleted. Note, there were a few chunks that were shared with the other image. These chunks aren't deleted, since they're still referenced by the 0bbf1 image



# Garbage Collection

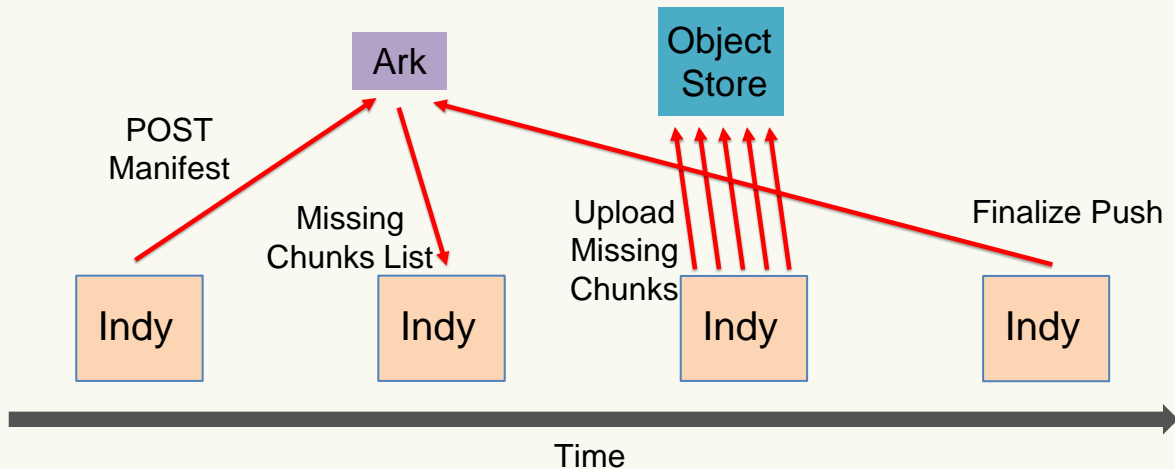
1. Delete all expired labels
2. Delete all unreferenced images
3. Delete all unreferenced chunks

So in summary, these are the steps we take

- \* Delete all expired labels
- \* Delete all unreferenced images
- \* And finally, delete all unreferenced chunks

# Pushing

Ark - Metadata  
Object Store - Storage



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

Next, let's talk about how  
`indy push` works, and how it  
deduplicates data.

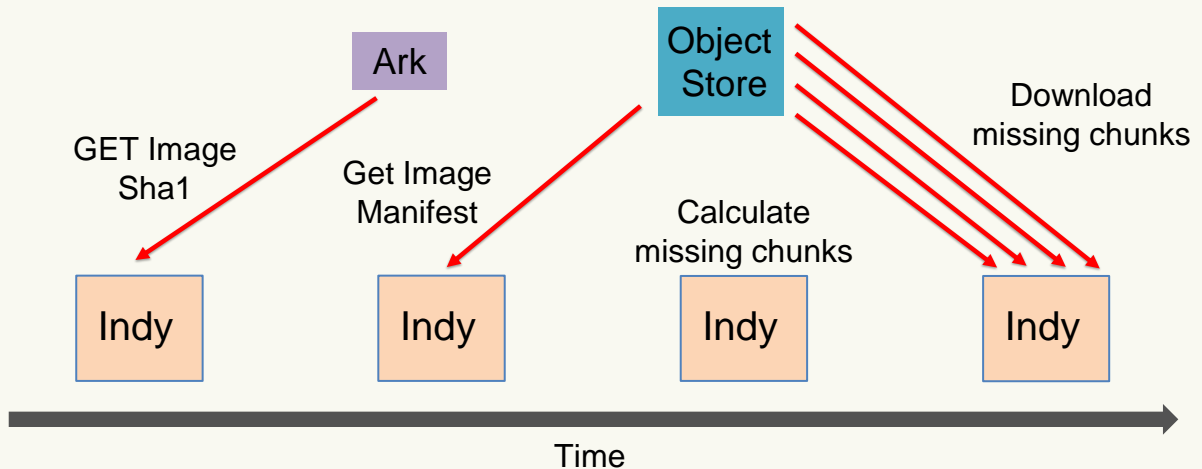
<click> First, indy will POST the  
image manifest to the Ark server.  
<click> Ark will query its internal  
database to figure out which chunks  
\*don't\* already exist on the server.  
And return this to `indy`  
<click> Next, in parallel, indy will  
upload the missing chunks to the  
object storage server.

<click> And finally send another POST to Ark, telling the server that it finished the upload

So, since Indy only uploads what is missing on the server, we can get excellent bandwidth savings, assuming the changes from image to image are small.

# Pulling

Ark - Metadata  
Object Store - Storage



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

Now we can look at pull

<click> First, indy will ask Ark for the image sha1, using the image label.

<click> Then, indy will use the image sha1 to download the manifest from the object store


<click> Indy will compare the chunks in the manifest against what it has locally already, so it can calculate the list of "missing" chunks

<click> Then indy will download the missing chunks in parallel

Again, since we only download the chunks that don't already exist on disk, we can get excellent bandwidth savings.

# Mounting an Indy Image

```
> tree ~/.indy
.indy
├── images
│   └── 4d002477d3aa6bfab47d34f58aa98451fe96bb8b
├── mounts
├── objects
│   ├── 03
│   │   └── 034a2bfcd34b70e279730d1ed4c055d0f93376b3
│   ├── 1c
│   │   └── 1c755788fa01e5f4fe480e3f54e68e6597f7f0f6
│   ├── 1e
│   │   └── 1e558dfbcbe27afe1f1c9c6a50f423b812d9d456
│   ├── 1f
│   │   └── 1fa4c43eeab6a68997d80af4118592296650546f
│   └── ----- (abbreviated for slides)
│       └── ff
│           └── ff6b440094fe85ebfec493f5f9749fe9ab40663
└── repo.yaml
```



Lastly, let's talk more about mounting.

<click> As discussed in previous slides, when we build or pull an image, the image chunks are stored in the "objects" folder on disk. <click> Specifically, they're stored by their content hash name. This is very important to clean up and data deduplication, but it's not useful to users.

Users want to see the files with the correct names and directory structure. This is where the `indy mount` command comes in.

<click>

# Mounting an Indy Image

```
> tree ~/.indy
.indy
├── images
│   └── 4d002477d3aa6bfab47d34f58aa98451fe96bb8b
├── mounts
│   ├── 6b46cd8a-91aa-4786-af51-fc206a2f6cb0
│   │   └── maps
│   │       ├── map1.bin -> hardlink to objects /1c/1c755788fa01e5f4fe480e3f54e68e6597f7f0f6
│   │       └── map2.bin -> hardlink to objects /03/034a2bfcd34b70e279730d1ed4c055d0f93376b3
│   └── 213c4192b81be6e3d1079c15d908c82c6447c4b3
├── objects
│   ├── 03
│   │   └── 034a2bfcd34b70e279730d1ed4c055d0f93376b3
│   ├── 1c
│   │   └── 1c755788fa01e5f4fe480e3f54e68e6597f7f0f6
│   └── 21
│       └── 213c4192b81be6e3d1079c15d908c82c6447c4b3
└── repo.yaml
```

----- (abbreviated for slides)  
<----- mapping of labels to image sha1

Mounts get created in the mounts folder.

We create the correct directory structure for the image, and then create hardlinks to the corresponding objects in the objects folder.

As a quick refresher, hard links are a filesystem concept where two filepaths actually correspond to the same file data. They're very fast to create, and don't cause any copying.



This means that creating mounts is semi-instantaneous, and we can create as many of them as we want with no additional disk space usage.

Let's move on to exploring some of the server concepts.

# Cache Nodes

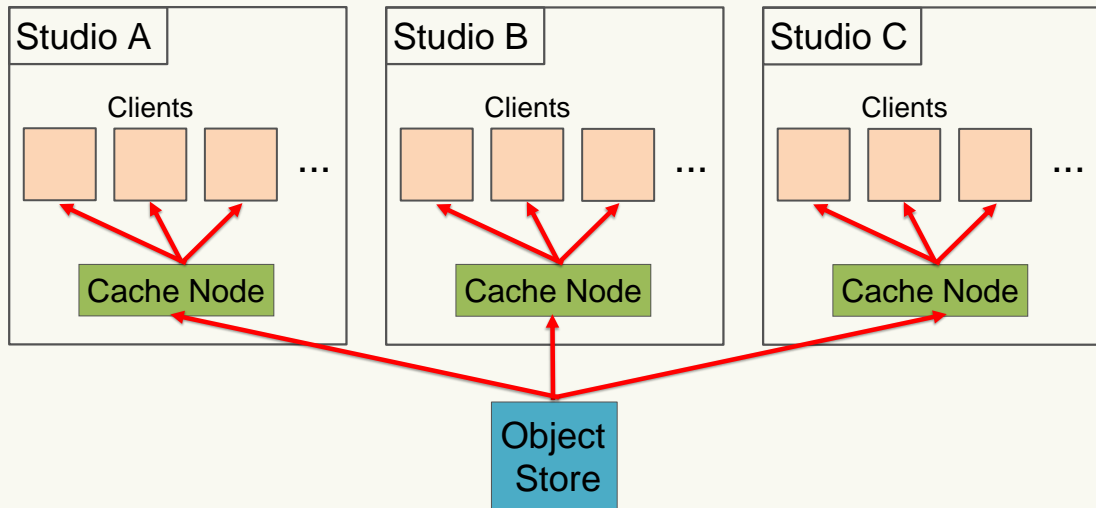
- All uploaded files are immutable
- Never have to worry about cache coherency

One of the great things about using content-addressable files for everything is that everything is immutable

This means that we can use cache nodes at each of our studio locations, because we never have to worry about cache coherency.

A file either exists, or it doesn't. The contents will never change.

# Cache Nodes



Our cache nodes are set up in a wheel and spoke model.

We have a central object store. And then each studio has a dedicated cache node.

All clients download and upload through the cache nodes. They never connect to the object store directly.

This has lots of benefits:

1. The cache nodes can cache files. So

if one client downloads an image, the next client can download the image chunks directly from the cache node. This reduces the WAN traffic, which is a much more precious resource. And it reduces load on the object store itself.

2. We control the cache nodes. So we can enable much more sophisticated and aggressive TCP tuning. For example, we enable the BBR congestion control algorithm. This guarantees that all WAN traffic is as optimized as possible
3. The cache nodes can be scaled to as many as we want. If a studio has a ton of clients, we can add multiple cache nodes, and load-balance between them. This also adds a layer of fault tolerance. We can have multiple cache nodes, so if we ever need to do maintenance on them, or if one of them crashes, a client can just switch to using a different cache node.

# Ark Garbage Collection

1. Delete all expired labels
2. Delete all unreferenced images
3. Ref-count all chunks
4. Delete all unreferenced chunks

When you push an image to Ark, you also specify a TTL for the label.

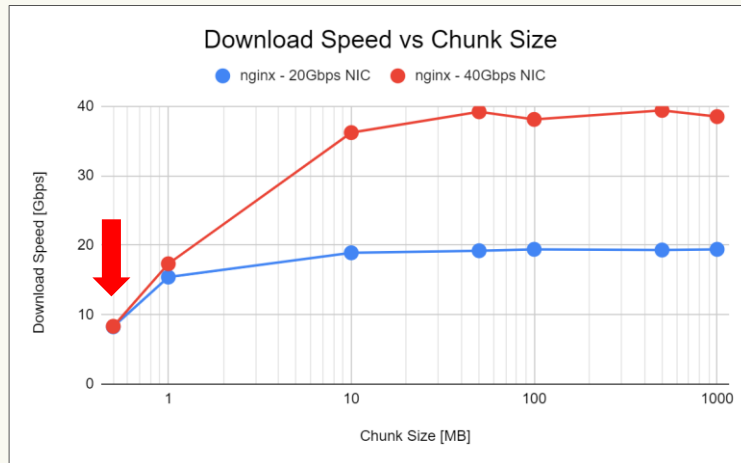
Similarly to local indy, Ark has a Garbage Collection process. This follows the same basic steps as local Garbage Collection, but utilizes a database to allow concurrency and atomicity at the server level.

Also, it runs automatically every 10 minutes, rather than being a manual command.

As a note, the details of \*how\* we execute these queries on the database in order to be atomic and race-free is a super interesting topic.

Unfortunately, it could cover an entire talk all to itself, so if you're curious, come talk to me after.

## Effect of Chunk Size



Lastly, let's cover a couple miscellaneous topics. Starting with exploring how chunk sizes affect things.

The first experiment that we did was to create files of different sizes and put them on two nearly identical nginx servers. One had a 20Gbps NIC, and the other had a 40 Gbps NIC.

Then we downloaded each file 1000 times, and measured the effective

bandwidth. <click> This chart is the result. Let's explore it a bit in detail.

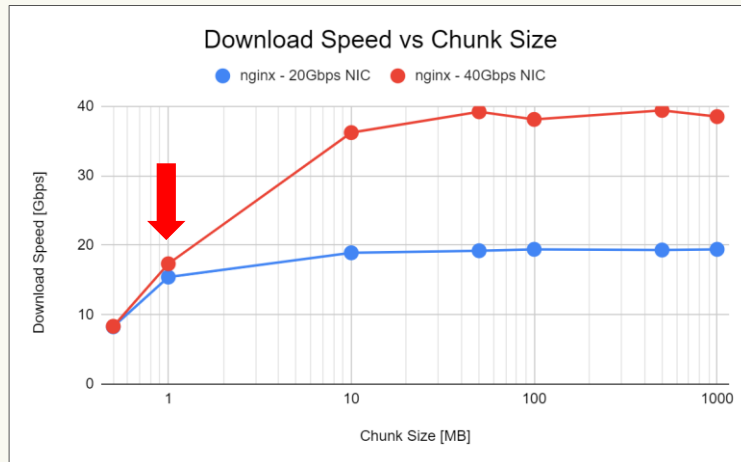
<click> When the chunk size is only 100kB the throughput is quite low. Even with a 40Gb NIC, we could only manage ~8Gbps of effective bandwidth.

This makes intuitive sense. TCP has overhead, and the HTTP framing itself has quite a bit of overhead. Plus, for every 100kB of data, you need to do a back and forth request / response with the server.

<click>



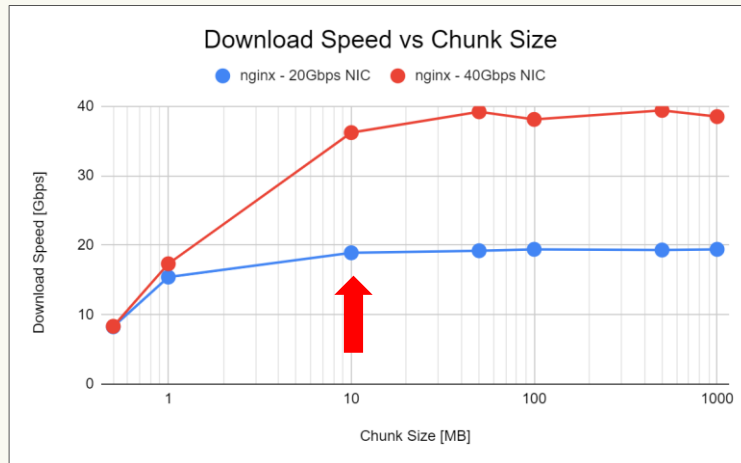
# Effect of Chunk Size



If we look at 1MB chunks, the performance is much better, <click> and by 10MB chunks, we're already hitting the NIC limit for the 20Gb server, and very close for the 40Gb server.

So the more data you can exchange per HTTP request, the higher your effective bandwidth will be. Though, after ~10MB, the bandwidth plateaus to the NIC limits.

## Effect of Chunk Size



If we look at 1MB chunks, the performance is much better, <click> and by 10MB chunks, we're already hitting the NIC limit for the 20Gb server, and very close for the 40Gb server.

So the more data you can exchange per HTTP request, the higher your effective bandwidth will be. Though, after ~10MB, the bandwidth plateaus to the NIC limits.

# Effect of Chunk Size

## Small Chunks

- Modified chunk is a small amount of data. (Higher deduplication)
- Retries are cheaper
- Transfer speed is slower

## Large Chunks

- Modified chunk is a large amount of data. (Lower deduplication)
- Retries are more expensive
- Transfer speed is faster

So what size do you choose? To answer that, we also have to look at the last two properties that chunk size affect:

- Deduplication
- HTTP retries

For small chunks, if a chunk is modified from one image to the next, it's not a bit deal. It's only a small amount of data to have to upload. Similarly, if the HTTP connection

breaks for whatever reason, it's not a big deal to have to re-transfer the entire chunk. On the flip side, the transfer speed can be quite slow.

For large chunks, if a single byte changes within a chunk, we still have to transfer the entire chunk. Similarly, if we're in the middle of uploading a 1GB chunk, and the TCP connection fails at the last 20 bytes, we have to re-upload the entire 1GB again. Though, in general, the transfer speed will be very fast.

# Effect of Chunk Size

~ 1 to 5 MB chunk size

So what size did *\*we\** choose? From our empirical testing, we found 1-5MB chunks to be a good balance between speed, deduplication, and retries.

# The Importance of \*Good\* Chunking

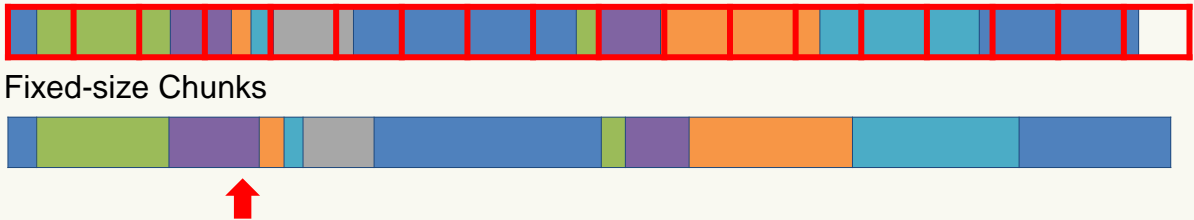


Now that we know how size affects performance, let's look at how we actually do the chunking with Indy.

This diagram represents a file. Each colored section represents a "logical" portion of the file. If this was a map file, one colored section might be a texture, and another model data. If we tell Indy to chunk every 1MB, this is what it would look like

<click>

# The Importance of \*Good\* Chunking



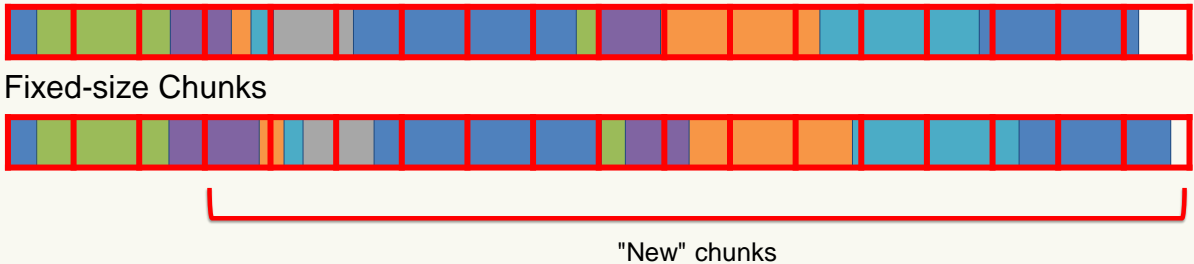
Everything inside each red square would be an Indy chunk. With the contents creating the sha1. Ok great. For a single file, in isolation, this works fine. However, our data isn't in isolation. We build similar data over and over and over again.

Let's say an artist modifies a texture for a tree and checked it in. <click> When we build the map, <click> that corresponds to this

purple section getting a little bit bigger.  
Ok, you say. What's wrong with that?



# The Importance of **\*Good\*** Chunking



Well, if we apply the same fixed-size chunking pattern to the file, we now have a problem. <click>

The addition to the purple section shifted the entire rest of the file down. So every single chunk after the modified section now has a different hash, and has to be uploaded to the server, because the contents changed.

This is obviously bad. It destroys all deduplication the file **\*should\*** have

gotten.

# The Importance of \*Good\* Chunking



Fixed-size Chunks



So what if we could tell Indy to chunk at the "natural" boundaries of the file, instead of at arbitrary fixed sizes? <click>

# The Importance of \*Good\* Chunking



Fixed-size Chunks



Custom-sized Chunks



That way, if a single section were to change <click>, the only chunk that would change would be that section.

Well there is! <click>

# Whole File

```
ADD my/local/path/file.txt /image/path/file.txt
```

## Pros

- Very simple

## Cons

- Single byte change = No deduplication

An Indyfile has 4 different ways to add a file. You've seen two of those ways so far.

<click> The first way is using the ADD command. It adds the file to the image using one giant chunk for the entire file

It's simple, but it isn't efficient, because a single byte change means the entire file needs to be re-transferred

# Fixed Chunk Size

```
CHUNK 1M      my/local/path/file.txt      /image/path/file.txt
```

## Pros

- Very simple

## Cons

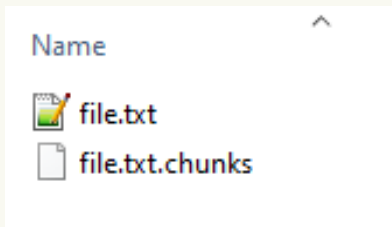
- Single byte change = All chunks after change in file, no deduplication

Next, you can tell Indy to use fixed-sized chunks. In this example, we tell it to make a chunk every 1MB

This is better than ADD, because you can change bytes without blowing the entire file. But, as we saw in the previous slides, if the file size changes at all, then we destroy the deduplication for the rest of the file.

# Custom Chunk Size

```
CHUNK .chunks my/local/path/file.txt /image/path/file.txt
```



```
0
23
54
120
436
```

Now we move to the new stuff.

<click> With this command you specify an extension. In this example, we used the ``.chunks`` extension.

What this does is instruct Indy to look for a file next the specified file, which has that extension <click>

The file should be a newline-separated list, <click> where each

entry gives the byte offset of each chunk.

So in this example, the first chunk would be from byte 0 to 22, the next from 23 to 53, etc.

<click>



# Custom Chunk Size

```
CHUNK .chunks my/local/path/file.txt /image/path/file.txt
```

## Pros

- Chunking matches "natural" file boundaries
- Change to a section of file -> only modifies that chunk

## Cons

- Requires knowing "natural" file boundaries
- Extra manual step
- "Natural" boundaries might create very tiny chunks or huge chunks

This allows user to specify exactly where Indy should split the file. This is great, because it means that if an artist changes a single texture, the only chunk that changes is that texture.

However, on the flip side, the user now has to know the exact file structure, they have to do an additional manual step to generate the .chunks file, and lastly, the natural boundaries might create

super tiny or super huge chunks. Which, from the previous slides, we know isn't good for performance.

<click>

# Content-Defined Chunking

March 21-25, 2022 | San Francisco, CA #GDC22

GDC

So what do you do if you don't know the file format? Or if the content inside the file is super tiny?

Is there a way that Indy can automatically find deterministic chunk locations? YES!

There is an algorithm called Content-Defined Chunking that uses hashing of the content to deterministically determine chunk boundaries

This is exactly what we need!

What's better, there are multiple open source implementations of the algorithm which we can use.

# Automatic Chunk Size

```
CHUNK ~1M my/local/path/file.txt /image/path/file.txt
```

CDC Setting	Chunk count	Match Size [GB]	Diff Size [GB]	Deduplication
~512K	245,516	119	22	85.31
~1M	127,955	114	27	81.70
~2M	69,353	108	34	77.18
~4M	40,177	101	41	71.96

We implemented this as follows  
<click>

You tell Indy the average chunk size that you want. Then it will pass that value to the Content-Defined Chunking algorithm to calculate the chunk offsets, and then it runs as usual.

As a quick and dirty test, I took the output of a package build from one day, and the output of another

package build from the next day.  
Then I ran `indy build` on each of the outputs, with 4 different CDC settings and compared their chunks.

<click> This chart shows the results.

To reiterate what this test did: For each setting, I would first build an image with the output from day 1. Then I would build another image with the output from day 2. And finally diff the two images.

As expected, when we specify smaller chunk sizes, the CDC algorithm is able to extract more deduplication from the files. However, the total number of chunks also goes up, and the chunk sizes are smaller. Which we know can lead to poor bandwidth performance.

Take these specific deduplication numbers with a grain of salt, but it's really interesting to see that the CDC algorithm is able to get such great

deduplication numbers, with no a-priori knowledge of the file structure.

# Automatic Chunk Size

```
CHUNK ~1M      my/local/path/file.txt      /image/path/file.txt
```

## Pros

- Very simple
- Excellent temporal deduplication

## Cons

- Two passes through file
  1. Figure out chunk boundaries
  2. "Normal" `indy build` portion

So, as a summary, automatic CDC chunking is very simple to use and doesn't require any a priori knowledge of the file structure. And on top of that, it performs very well.

That said, there is no free lunch. CDC does require you to do two passes through the file. First, to figure out the chunk boundaries, and then the second as the "normal" indy build. So builds are slightly slower.



# All the Ways

```
ADD my/local/path/file.txt /image/path/file.txt
```

```
CHUNK 1M my/local/path/file.txt /image/path/file.txt
```

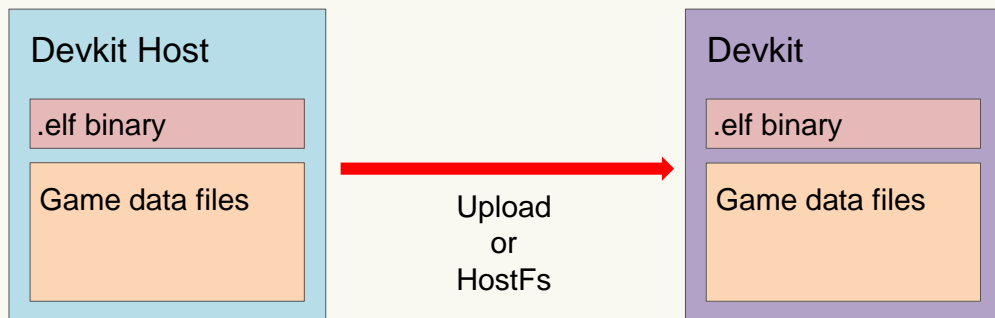
```
CHUNK .chunks my/local/path/file.txt /image/path/file.txt
```

```
CHUNK ~1M my/local/path/file.txt /image/path/file.txt
```

So here are all the Indyfile methods for adding a file.

- ADD the whole file
- Fixed-size chunks
- Custom-size chunks
- Automatic CDC chunks

# What about Devkits?



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

Ok great. We have this new tech that works for all our workstations and all our servers. But what do we do for the devkits?

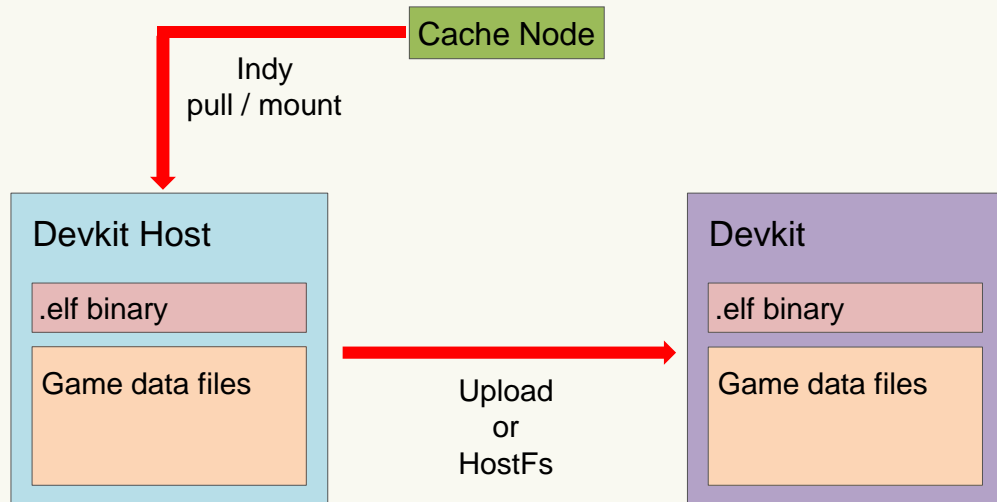
<click>

As a quick review, for console development, you have a host PC, which we call the devkit host. And the console devkit itself. To launch the game, you need to get the elf binary and the game data files

from the devkit host to the devkit.

<click> Xbox and Playstation support either uploading everything before you launch the game, or streaming the files from the host to the devkit just-in-time. We call this second approach HostFs.

# What about Devkits?



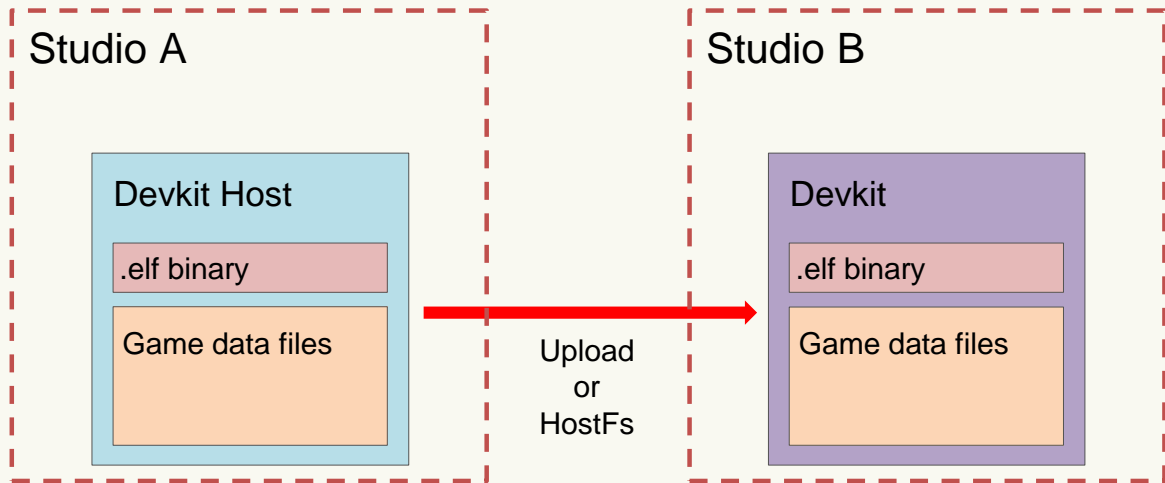
If we plug in Indy, it looks like this.

We use Indy to pull/mount the data on the devkit host.

Then transfer all the data to the devkit and finally launch the game.

This is *\*fine\** for local development. But it has a number of issues for a CI system.

# What about Devkits?



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

First, our devkits are a precious resource. We have a limited number of them for all the projects.

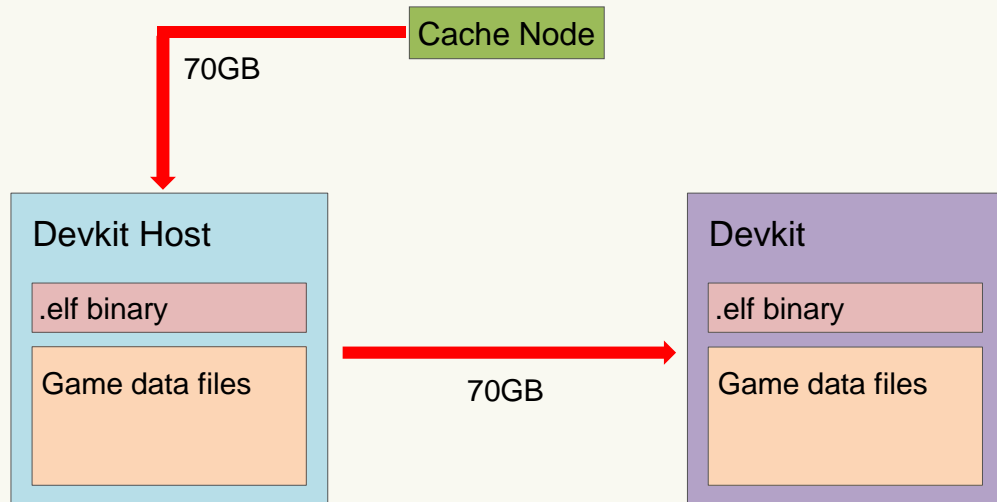
Ideally, we'd have devkit hosts at all the studios, but this doesn't always happen.

Especially since the hosts are kind of a "waste" of server resources. They basically sit idle during the game run and read the console logs.

<click>

So the host and devkit might not be in the same physical location. And we know that WAN bandwidth is both precious and super slow, because of latency.

## What about Devkits?



Next, if we look at the data path, we're downloading the elf and all the data to the host, and then immediately transferring all that data to the devkit. It's an extra hop.

In addition, even though the game build is 70GB, a run of a single map may only need to read 10 GB or so of data. But with this setup we still have to download and transfer all 70GB.

# IndyFs

```
FileSysHandle OpenFile( const char *filename, int flags );  
size_t Size( FileSysHandle handle );  
uint64_t Seek( FileSysHandle handle, const uint64_t pos );  
void Close( FileSysHandle handle );  
bool Exists( const char *filename );  
FileSysResult ReadBlocking( FileSysHandle handle, uint64_t offset, uint64_t size, void *dest,  
    uint64_t *numBytesRead );  
FileReadHandle StartRead( FileSysHandle handle, uint64_t offset, uint64_t size, void *dest,  
    FileReadCallback cbfn, void *cbContext );
```

So to solve these problems we created IndyFs

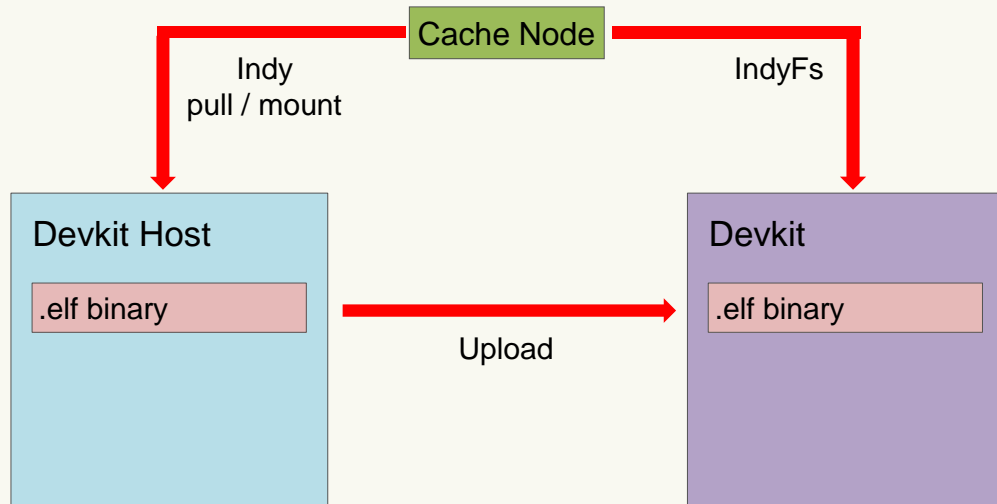
The game abstracts all filesystem access behind an interface that looks like this.

All your standard functions, open, close, read, etc.

So we created Indy File System, which is a virtual filesystem, using Indy behind the scenes.



# IndyFs



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

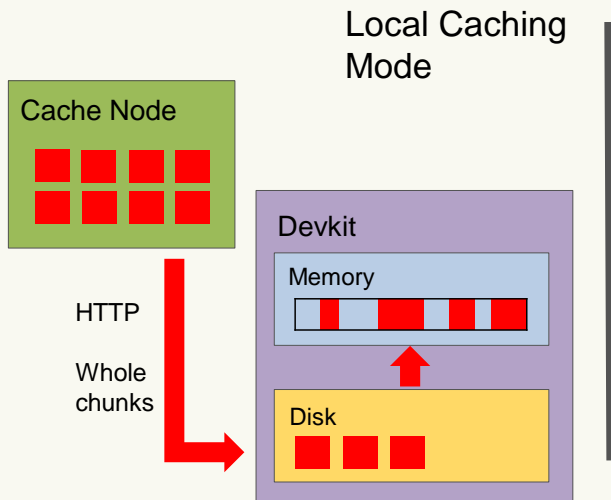
This is the flow.

<click> First, we download \*just\* the binary to the devkit host.

<click> Then we upload the binary to the devkit and launch the game

<click> During the game run, we download the game data on the fly using IndyFs

# IndyFs Modes



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

IndyFs can run in two different modes: Local Caching Mode, and Direct To Memory Mode

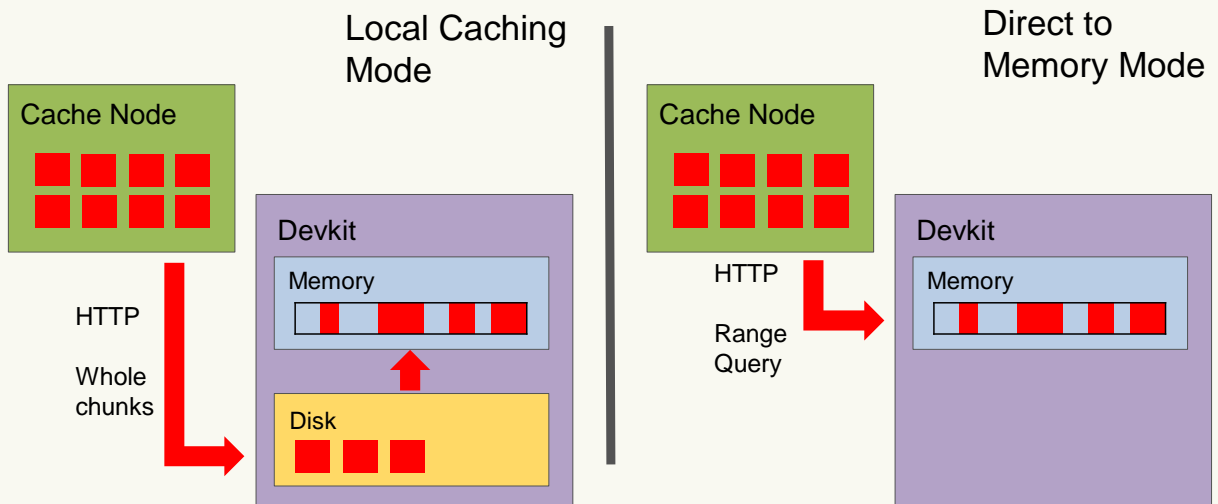
In Local Caching Mode, we download whole chunks to disk, and then use those to serve read requests to the game.

If the game asks for data from a chunk already downloaded, we don't have to download it again, since it's already on disk.

Similarly, from one game run to the

next, many of the chunks will be the same, so we don't have to re-download that data

# IndyFs Modes



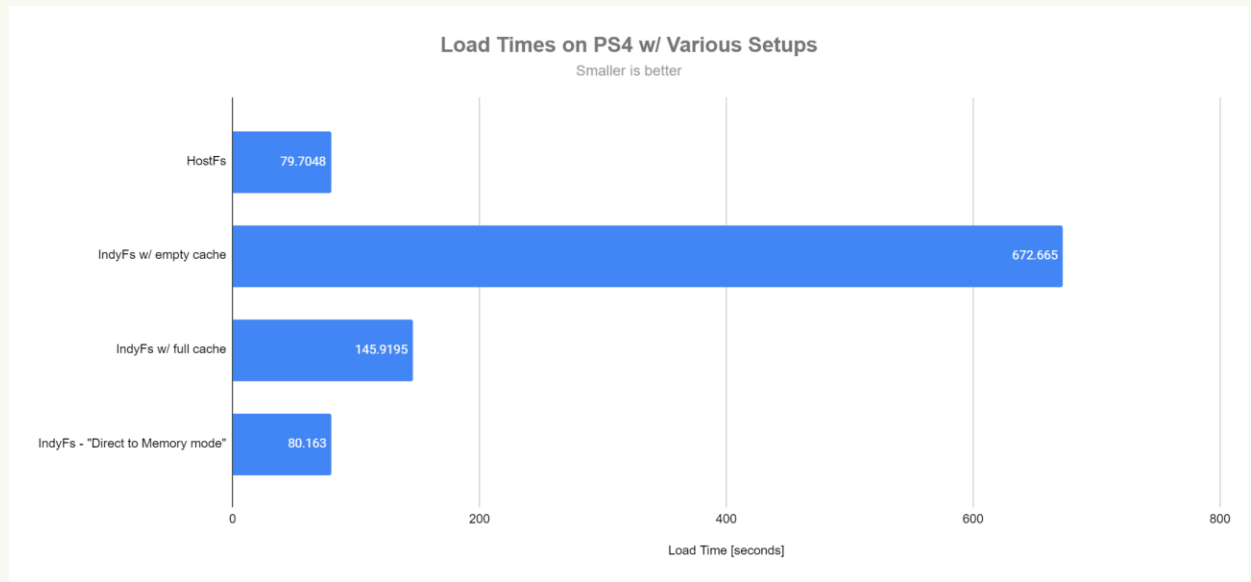
March 21-25, 2022 | San Francisco, CA #GDC22

GDC

In Direct to Memory mode, we don't touch disk at all. Instead, we use HTTP range requests to download chunk data Just-In-Time as the game requests it.

We lose all caching potential we could get vs Local Caching Mode, but we avoid the disk.

Why do we want to avoid the disk?



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

PS4 / XB3 HDD is specced for durability, not raw speed. Best-case scenario, the drive can do 120 MBps. But that's with fully sequential, large read.

Also, HDDs in general perform very poorly if you are doing lots of mixed read/writes/seek.

Local caching mode is the worst-case for this.

This graph shows a series of tests

that I did. I started the game in 4 different modes, and timed how long it took to boot into a map.

HostFs dynamically streams everything from the Host PC, and boots quite fast. IndyFs using Local Caching Mode, with an empty cache was horribly slow. An order of magnitude slower than HostFs. Even with the best case scenario, a full cache. Everything is read, no writes, it's still slow. This is just due to the specs of the drive, and having to do seeks to the different chunks.

IndyFs with Direct To Memory Mode, aka we download everything on the fly, we get performance on par with HostFS. IE the NIC vastly outperforms the disk.

That all said, if we switch to SSDs, the story is completely flipped. For PC, it's hugely beneficial to cache. I don't have a graph to show you, but IndyFs in caching

mode is vastly faster, because SSD speeds are on the order of GB/s and are much more resilient to seeks.

# How Well is it Working?

Very Well!

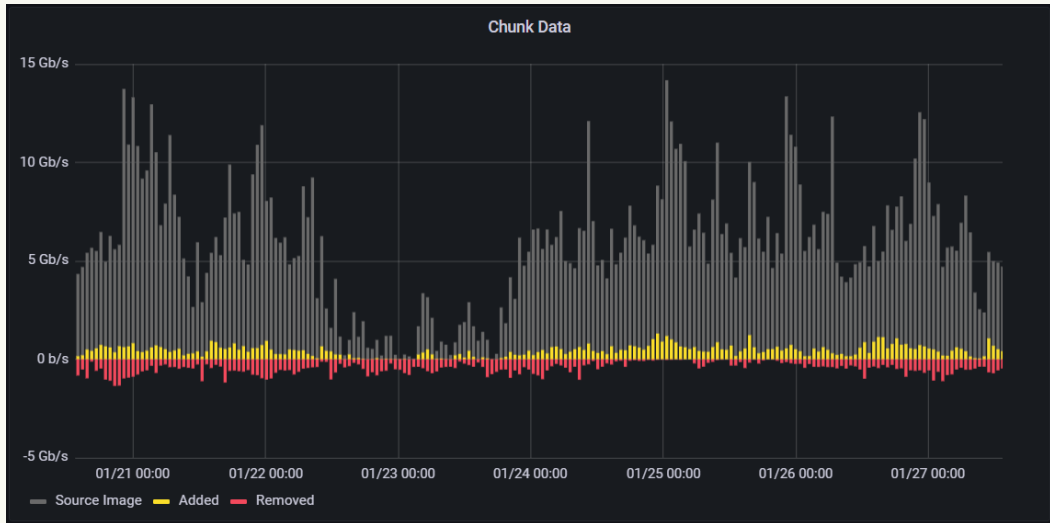
So, now we have this new ecosystem for transferring tons of data around. Just how well is it working?

<click>

Very well!



# Raw Data Deduplication



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

Let's look at the raw data deduplication we're getting.

This graph measures the data rate of images being created and removed for a \*single\* project

The gray bars represent how much data would be added if there was no deduplication from the server. That is, if every image push uploaded ALL chunks.

For those of you that can't read the

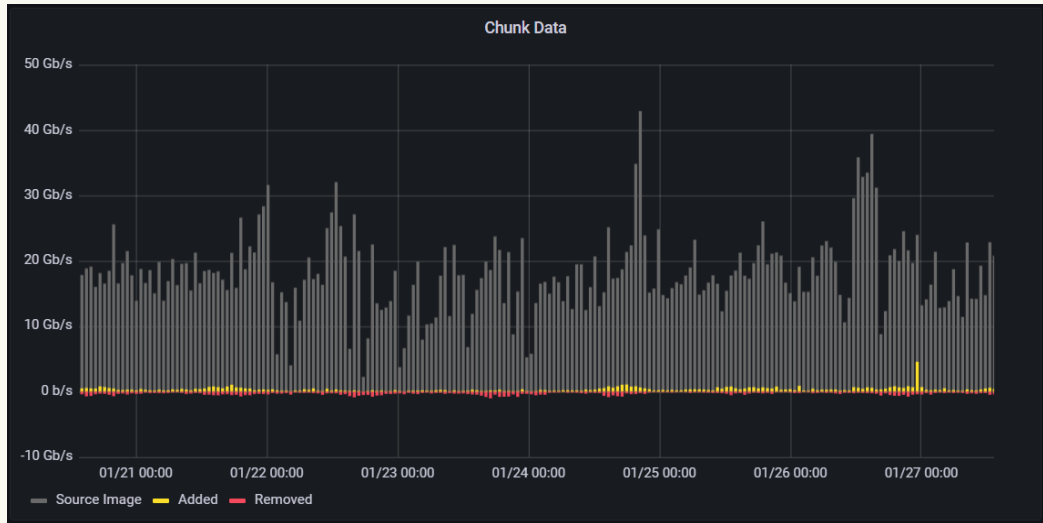
Y-axis numbers, the peaks here are about 5 to 10Gbps.

The yellow bars represent how much data was \*actually\* uploaded. It's about 1.2Gbps.

The red bars represent how much data is being deleted by Ark garbage collection.

NOTE: This only a single project. Here is another project <click>

# Raw Data Deduplication



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

This project has a very active package generation pipeline, which adds a ton of data. Pre-deduplication peaks are at 20 Gbps. However, they're getting excellent deduplication, so the actual data uploaded is just 1 Gbps.

## Cache Nodes Function like a CDN



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

Now let's look at the cache node performance. <click>

The Indy cache nodes end up functioning like a classic CDN.

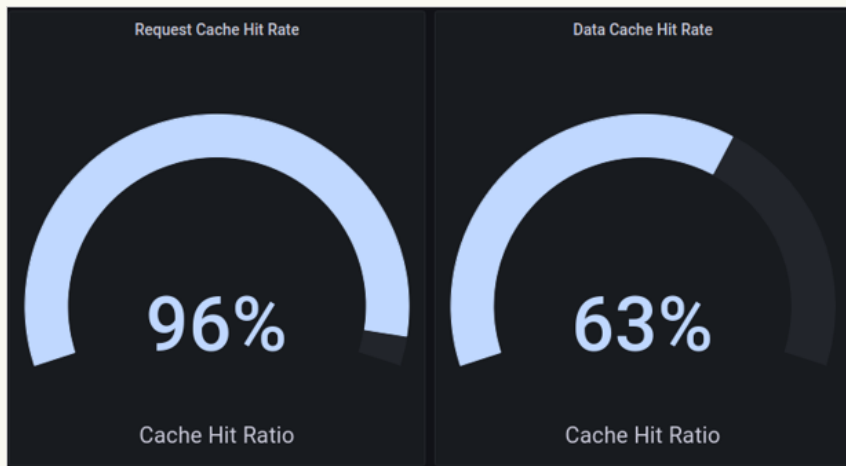
We build and push an image once, and then consume it multiple times. And the image is cached by the cache nodes.

We also get temporal caching. AKA, a build pulled from 10 minutes ago probably has lots of the same chunks as the build we're about to pull

This table shows those numbers.  
It shows that in 24 hours, we uploaded  
39.7 TB of data through the cache  
nodes.  
And downloaded 230 TB of data.

So we're downloading almost 6 times  
more data than we upload.

# Cache Hit Ratios



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

The cache nodes are also working exceedingly well as caches.

96% of all requests are cached already, which corresponds to 63% of all data is already cached. This is due to temporal deduplication, or clients pulling the same image.

This is really good, because it drastically reduces the load on the WAN and the object store.

# Lessons Learned Along the Way

March 21-25, 2022 | San Francisco, CA #GDC22

GDC

Now we can move on to some of the lessons we learned along the way while developing this tool and deploying it to production.

# Curse of Scale - Probability

- Bug: 1 in 1 billion chance
- We do operation 100 million times per day
- Bug will occur roughly every 10 days

First, the curse of scale. Indy is being used in production with most of our projects, and being used by thousands of CI workers each day. This is awesome, but it also shines a huge spotlight on any issues that we might have.

Let's imagine a bug that has a 1 in 1 billion chance of happening <click>. That seems pretty tiny. And it is. <click> But if we do that operation 100 million times a day <click>, then



the bug will occur roughly every 10 days...

Hence, the name, the curse of scale. Very small probabilities become very possible, or even semi-guaranteed, due to the huge number of operations we're doing.

# Curse of Scale - Hardware

Disks *\*will\** fail. Memory *\*will\** fail. BSoDs *\*will\** happen

Indy verifies data integrity at every step

`indy verify` to validate the contents on disk

<click> On that same note, at this scale of workers, it's semi-guaranteed that at least one of the workers will have a disk fail, or memory fail, or Blue Screen of Death. It's not a matter of *\*if\** it will fail, but *\*when\** it will fail.

<click> In order to combat this and prevent chunk corruption on the server object store, indy will do data verification at every single step of a pull or push. As it reads data from

disk or from the network, it will re-hash the data and verify it matches what we expect it to.

<click> Lastly, the data we store on disk must be durable. When we do a pull, we only download the chunks that we don't already have. So if a chunk is corrupt on disk, due to say, a disk error, this would corrupt all image files that use that chunk.

So to combat *\*this\** problem, we have the ``indy verify`` command which will verify the data integrity of all images and chunks on disk. We run this nightly on all our CI workers to help guarantee that we always have correct data for our CI testing.

# The Perfect DDoS Hammer

- Stampeding herd can cause further networking outage or DDoS
- Fix: Randomized exponential backoff

With Indy now being used to shuffle around so much data, we have to be very careful and considerate to the network as a whole.

<click> If the network is experiencing congestion or some other issue, Indy can very easily choke out other network traffic or cause the network issue to be worse, due to the stampeding herd problem. This is where many clients fail their requests and immediately retry,

causing a stampede of new requests, which can cripple the network.

<click> To solve this, we added randomized exponential backoff to all requests in Indy. This adds backpressure to the system, so network issues have a chance to be fixed without a herd of new requests.

# Firewalls

- Talk to your IT department
- Traffic Shaping
- Ask about firewall ratings
  - 10 Gbps rating != 10 Gbps bandwidth

Speaking of networking, let's talk about firewalls.

<click> If you're going to implement a system like this, make sure to talk to your IT department. They can get really confused and concerned when all of a sudden a few dozen cache nodes are now doing Gbps of network traffic.

<click> One way to possibly prevent Indy from stealing all the bandwidth

is to use traffic shaping on the firewall to limit the bandwidth for the cache nodes. This way, Indy is capped at how much of the straw it is allowed to suck through.

<click> Lastly, ask your IT department about the rating of the firewall.

Specifically, the ratings of firewall with all the settings enable. The reason being, a firewall rating can be a bit of a lie.

\*Usually\* the rating gives the bandwidth the firewall can handle with \*everything\* turned off. Which is makes for a pretty useless firewall. If you enable anything useful, like packet classification, the bandwidth the firewall can handle will be much much lower than the rating. If I remember correctly, one of our 10 Gbps rated firewalls could only handle 7 Gbps with packet classification turned on, and only 4 Gbps with everything turned on.

# Filesystems

Issues:

- Atomicity
  - Temp file, fsync(), rename
  - POSIX rename - ✓
  - NTFS rename - ✗

Let's move on to filesystems

Many of the local indy operations are creating files (using multiple threads, and potentially multiple processes of indy). And potentially the content of those files will end up being the same hash, so they will want to write to the same file name. <click>  
<click>

We do the classic "write to a temporary file, call fsync(), and then



rename to the correct name". To ensure that the file write is safe and complete.

However, we run into issues with the rename.

The POSIX spec guarantees a file rename will be atomic. It will either succeed or fail. There is no intermediate state.

NTFS doesn't have any similar guarantee. None of the official documentation mentions anything about atomicity. Worse yet, some of the "rename-like" functions even mention being implemented as a copy+delete, which is *\*definitely\** not atomic.

Windows used to have a "Transacted" API, but it's deprecated, and will be removed soon.

There is a great CppCon talk by Naill Douglass called "Racing the

Filesystem". <https://www.youtube.com/watch?v=uhRWMGBjIO8>

In the talk he goes through a magic incantation of Win32 calls that you can do, which *\*should\** be an atomic.

So, we do that. However, even with that, we were able to get a reproducible bug, where two threads trying to rename tmp files to the same dest file will both succeed, but it will end up with a corrupt truncated file.

So, we resorted to a global mutex on renames, to prevent collisions at the file system layer. It's an area that we'd love to investigate more and get a better solution for.

# fsync() is Very Important

- Filesystems only journal metadata
- Machines *\*will\** crash, especially human workstations
- Crash and OS hasn't flushed write buffers -> Corrupt file

I mentioned that our process was:

Write to a temp file, call fsync(), and then rename. Why do we need the fsync()?

<click>

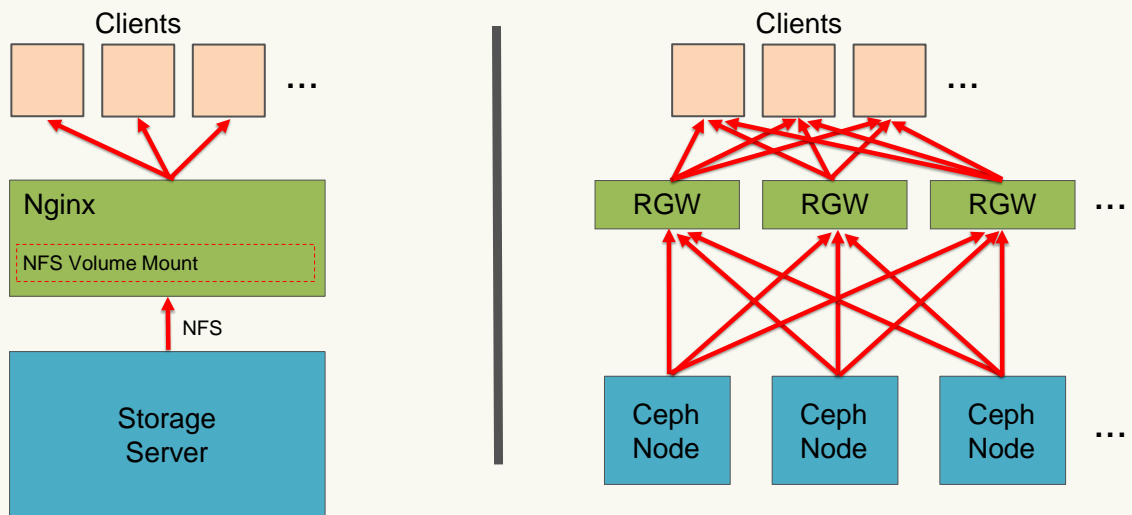
Most filesystems journal metadata. So if you do an iNode operation like a rename, the filesystem will not return to you until that operation is persisted in the journal.

However, almost no filesystems journal the data itself. This is purely for speed reasons.

So, if we write to a temp file, then rename without calling `fsync()`, the new file will exist, but the data for it is in an indeterminate state. The operating system will lazily flush its write buffers to disk, as it gets free CPU cycles. If our computer crashes before everything is flushed to disk, the new file will exist, because the metadata was journaled, but the data will be corrupt. Some filesystems will zero-fill the rest of the file and some will truncate.

We can't tolerate this corruption, so we call `fsync()` before calling `rename` to guarantee that the data exists on disk.

## Switch from NFS to RGW S3



Finally, let's talk about the object store itself.

In the first generation of Ark servers, we leveraged the existing storage servers we already had. <click>

We ran a single nginx container that mounted an NFS volume from one of our storage servers.

This worked relatively well, but had all the drawbacks that we discussed at the beginning of the talk. Namely:

- It was a single point of failure
- It couldn't scale
- And NFS implements a full posix file system. Which is nice, but Indy doesn't need that. Chunks are all immutable and content addressable. They either exist or they don't.
- Related to that, in order to be "safe" with the filesystem, nginx also does the standard, "write to a temp file, fsync(), and rename". Which is good for correctness, but it creates a huge number of IOPS for NFS to handle. Which can bring the storage server performance to a crawl

So, in order to address these concerns, we migrated to Ceph RGW. <click>

Ceph RGW is an implementation of the S3 object storage protocol on top of Ceph.

Ceph has lots of advantages:

1. The storage itself is sharded across a number of physical servers. Ceph uses redundant copies, so a single node failure doesn't cause any outage.
2. You can add additional ceph nodes basically to infinity. For increased storage capacity and bandwidth.
3. The RGW instances themselves are stateless. And we can scale out to any number.
4. The clients can connect to any RGW instance. The RGW instance will connect to the relevant ceph nodes and fetch the corresponding data.
5. Since the data is replicated across multiple nodes, RGW can fetch data from multiple nodes in parallel, for increased throughput, similar to RAID

# Improvements for the Future

- Fine-grained authorization
- Service discovery for cache nodes

To finish off today's presentation, we'll look at some of the improvements we're looking to do for the future of Indy.

<click> First, authorization. Currently, we only have service-level authorization implemented. That is, a user either has access to an Ark, or they don't. Ideally, we'd like to implement something more fine-grained. For example, access at the namespace level. So someone could



have read-only access to the namespace "ci-builds" and read/write access to their personal namespace.

<click> Next, we want to explore a better method for doing service discovery of the cache nodes. Let me explain <click>

# Service Discovery for Cache Nodes

`/storage`

<pre>[   {     "id": "0093c208-610b-498c-8830-eadd01c80844",     "site": "ctla",     "url": "http://example.com/foo/"   },   {     "id": "a87b2286-ce0b-45cc-80e2-80638b4156d6",     "site": "ctla",     "url": "http://example2.com/foo/"   },   {     "id": "4465e6bd-ba98-4420-9b63-6ef28cd2a6ca",     "site": "bnx",     "url": "http://bnx.example.com/foo/"   } ]</pre>	<p>← HEAD /health - 6ms</p> <p>← HEAD /health - 8ms</p> <p>← HEAD /health - 122ms</p>
---	---

Each Ark can have a number of cache nodes associated with it. When indy-cli needs to use a cache node, it first hits Ark's `/storage` endpoint

<click> This will return a JSON giving the list of all cache nodes.

<click> Then indy-cli will do simultaneous HEAD requests to all the cache nodes, and then pick the one with the fastest response time

This works well, and handles cache node outages / maintenance seamlessly. However, it's a lot of "manual" work for indy-cli to do every time.

Some of the approaches that we're exploring are:

1. GeoIP DNS resolution with dynamic BGP routing

This is the most ideal solution. Where you'd just give everyone the same DNS name, and it would dynamically resolve to the closest, alive node to them.

Unfortunately, it does require a ton of control at your network layer. So your IT team may just say no. This is the kind of thing that cloud provides.

2. Using a Consul service mesh to do the discovery for us

# Improvements for the Future

- Fine-grained authorization
- Service discovery for cache nodes
- Fine-grained locking for indy.exe

Next up, let's talk about locking in indy-cli.

Certain commands in indy have to be protected against other commands. For example, you don't want to have a gc start up in the middle of a pull. The gc would delete all the chunks you're in the process of pulling, because they're not referenced yet.

To protect against this, early in the design, we added a process file lock.

So only a single indy process could run at one time.

This guaranteed correctness, and protection. But with a cost; only one indy processs can be running at a time.

To be fair, in the 4 year of production, it hasn't been a huge problem.

That said, as we're moving to use indy in more user-facing tools, the lock can end up looking like a hang to a user.

So we're investigating how we can break up the lock into smaller pieces and adding exclusivity to the pieces.

For example, it's totally fine for multiple pulls to happen at the same time, since they're only adding files, and not deleting anything.

However, if we run a 'gc', we need get an exclusive lock on everything, so we can be sure that nothing is adding as we delete.

## Improvements for the Future

- Fine-grained authorization
- Service discovery for cache nodes
- Fine-grained locking for indy.exe
- UI to facilitate build management

Lastly, while indy-cli has a very intuitive set of commands, I don't expect an artist or QA person to have to break out the command line to get their daily builds.

So we're working on a GUI to facilitate build management and delivery. Behind the scenes it will use indy with all the benefits that come with it, but it will more user friendly for those that just want their data.

## Conclusion

- Indy has been a huge success
- In production use for 4 years
- Used by most of our projects
- Saves huge amounts of bandwidth and time
- Allows us to scale to workers spread across many physical locations

So, wrapping up: Indy has been a huge success. It's been used continuously for the past 4 years for most of our projects. It's saving us a huge amount of bandwidth and time, but more than that, it's allowed us to utilize workers spread out across many physical locations; allowing much closer collaboration between the studios working on a project.

# Special Thanks

Sean Houghton  
Will Brode  
Daniel Meirovitch

The studio tools teams

I want to give a big shout out to my team members that have helped build the tool:

Sean Houghton, Will Brode, and Daniel Meirovitch.

And another shout out to the studio tools teams that integrated indy into their pipelines and gave valuable feedback and improvements.



# We're Hiring!



<https://careers.activisionblizzard.com/>

And lastly, as you can see from the last part of the talk, we still have lots of unsolved problems and new tools to create.

Do you have ideas on how to solve these problems, or do you want to work on tools similar to this? Come work at Central Tech! We have lots of open positions and would love to hear from you

# Questions?

And with that, I want to thank you for coming and listening to my talk, and we can open up the floor to questions.

