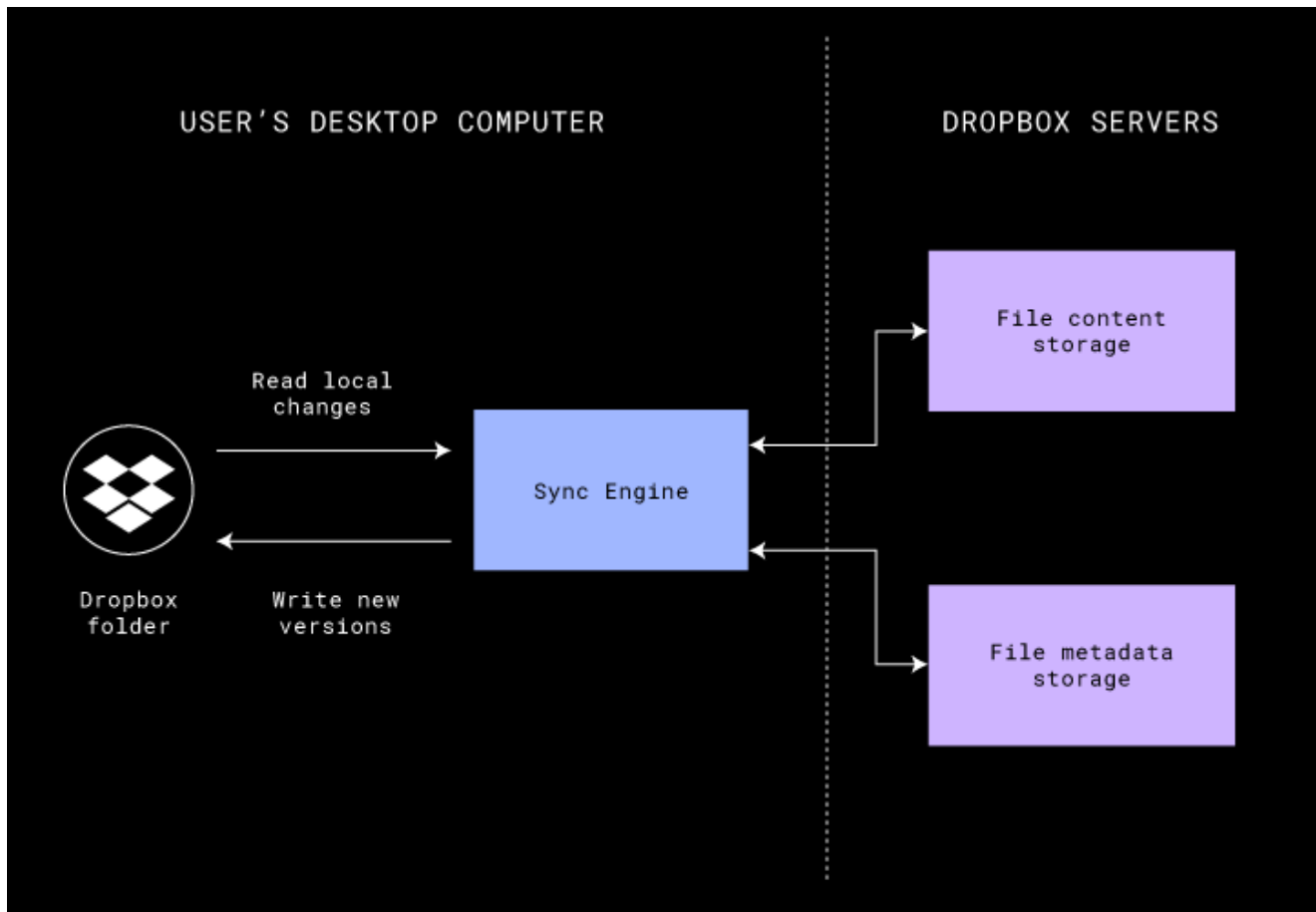


 dropbox.tech/infrastructure/rewriting-the-heart-of-our-sync-engine

1. Sync at scale is hard
2. Why rewrite?
3. A rewrite checklist
4. So, what did we build?

Rewriting the sync engine was really hard, and we don't want to blindly celebrate it, because in many environments it would have been a terrible idea. It turned out that this was an excellent idea for Dropbox but only because we were very thoughtful about how we went about this process. In particular, we're going to share reflections on how to think about a major software rewrite and highlight the key initiatives that made this project a success, like having a very clean data model.

Motion Offense Exhibit 2022
1/11



What's a sync engine? A sync engine lives on your computer and coordinates uploading and downloading your files to a remote filesystem.

Since then we've taken file sync pretty far. We started with consumers syncing what fit on their devices for their own personal use. Now, our users bring Dropbox to their jobs, where they have access to millions of files organized in company sharing hierarchies. The content of these files often far exceeds their computer's local disk space, and they can now use Dropbox Smart Sync to download files only when they need them. Dropbox has hundreds of billions of files, trillions of file revisions, and exabytes of customer data. Users access their files across hundreds of millions of devices, all networked in an enormous distributed system.

Sync at scale is hard

Syncing files becomes much harder at scale, and understanding why is important for understanding why we decided to rewrite. Our first sync engine, which we call "Sync Engine Classic," had fundamental issues with its data model that only showed up at scale, and these issues made incremental improvements impossible.

Distributed systems are hard

The scale of Dropbox alone is a hard systems engineering challenge. But putting raw scale aside, file synchronization is a unique distributed systems problem, because clients are allowed to go offline for long periods of time and reconcile their changes when they return. Network partitions are anomalous conditions for many distributed systems algorithms, yet they are standard operation for us.

Getting this right is important: Users trust Dropbox with their most precious content, and keeping it safe is non-negotiable. Bidirectional sync has many corner cases, and durability is harder than just making sure we don't delete or corrupt data on the server. For example, Sync Engine Classic represents moves as pairs of deletes at the old location and adds at the new location. Consider the case where, due to a transient network hiccup, a delete goes through but its corresponding add does not. Then, the user would see the file missing on the server and their other devices, even though they only moved it locally.

Durability everywhere is hard

Dropbox also aims to “just work” on users’ computers, no matter their configuration. We support Windows, macOS, and Linux, and each of these platforms has a variety of filesystems, all with slightly different behavior. Under the operating system, there's enormous variation in hardware, and users also install different kernel extensions or drivers that change the behavior *within* the operating system. And above Dropbox, applications all use the filesystem in different ways and rely on behavior that may not actually be part of its specification.

Guaranteeing durability in a particular environment requires understanding its implementation, mitigating its bugs, and sometimes even reverse-engineering it when debugging production issues. These issues often only show up in large populations, since a rare filesystem bug may only affect a very small fraction of users. So at scale, “just working” across many environments and providing strong durability guarantees are fundamentally opposed.

Testing file sync is hard

With a sufficiently large user base, just about anything that's theoretically possible will happen in production. Debugging issues in production is much more expensive than finding them in development, especially for software that runs on users’ devices. So, catching regressions with automated testing before they hit production is critical at scale.

However, testing sync engines well is difficult since the number of possible combinations of file states and user actions is astronomical. A shared folder may have thousands of members, each with a sync engine with varied connectivity and a differently out-of-date view

of the Dropbox filesystem. Every user may have different local changes that are pending upload, and they may have different partial progress for downloading files from the server. Therefore, there are many possible “snapshots” of the system, all of which we must test.

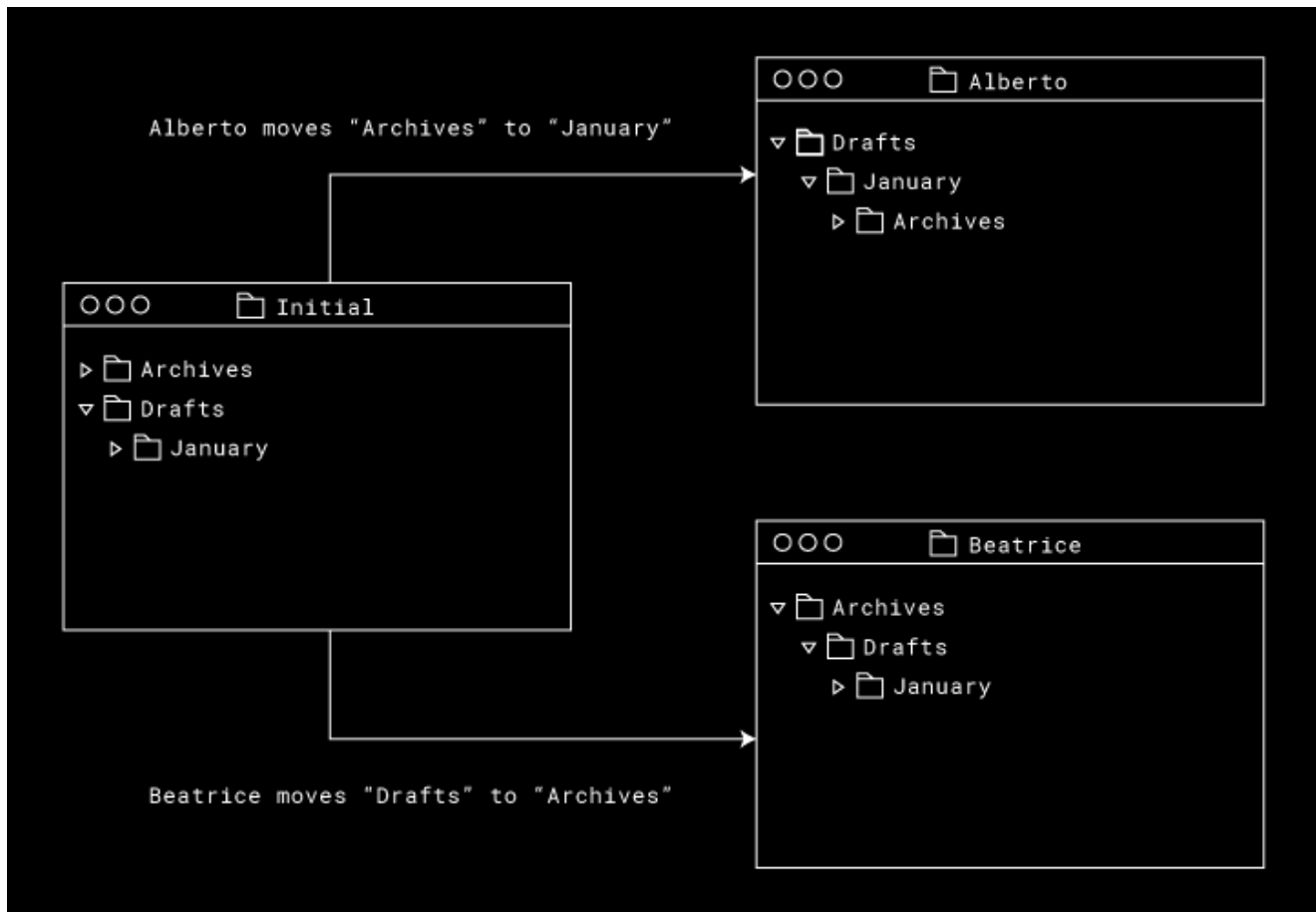
The number of valid actions to take from a system state is also tremendously large. Syncing files is a heavily concurrent process, where the user may be simultaneously uploading and downloading many files at the same time. Syncing an individual file may involve transferring content chunks in parallel, writing contents to disk, or reading from the local filesystem. Comprehensive testing requires trying different sequences of these actions to ensure our system is free of concurrency bugs.

Specifying sync behavior is hard

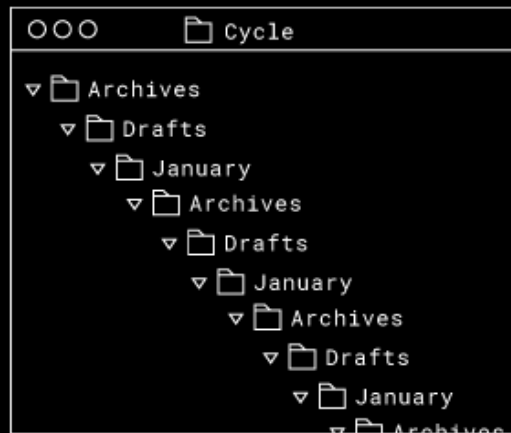
Finally, if the large state space wasn’t bad enough, it’s often hard to precisely define correct behavior for a sync engine. For example, consider the case where we have three folders with one nested inside another.



Then, let’s say we have two users—Alberto and Beatrice—who are working within this folder offline. Alberto moves “Archives” into “January,” and Beatrice moves “Drafts” into “Archives.”



What should happen when they both come back online? If we apply these moves directly, we'll have a cycle in our filesystem graph: "Archives" is the parent of "Drafts," "Drafts" is the parent of "January," and "January" is the parent of "Archives."



What's the correct final system state in this situation? Sync Engine Classic duplicates each directory, merging Alberto's and Beatrice's directory trees. With Nucleus we keep the original directories, and the final order depends on which sync engine uploads their move first.



In this simple situation with three folders and two moves, Nucleus has a satisfying final state. But how do we specify sync behavior in general without drowning in a list of corner cases?

Why rewrite?

Okay, so syncing files at scale is hard. Back in 2016, it looked like we had solved that problem pretty well. We had hundreds of millions of users, new product features like Smart Sync on the way, and a strong team of sync experts. Sync Engine Classic had years of production hardening, and we had spent time hunting down and fixing even the rarest bugs.

Joel Spolsky called rewriting code from scratch the "single worst strategic mistake that any software company can make." Successfully pulling off a rewrite often requires slowing feature development, since progress made on the old system needs to be ported over to the new one. And, of course, there were plenty of user-facing projects our sync engineers could have worked on.

But despite its success, Sync Engine Classic was deeply unhealthy. In the course of building Smart Sync, we'd made many incremental improvements to the system, cleaning up ugly code, refactoring interfaces, and even adding Python type annotations. We'd added copious telemetry and built processes for ensuring maintenance was safe and easy. However, these incremental improvements weren't enough.

Shipping any change to sync behavior required an arduous rollout, and we'd still find complex inconsistencies in production. The team would have to drop everything, diagnose the issue, fix it, and then spend time getting their apps back into a good state. Even though we had a strong team of experts, onboarding new engineers to the system took years. Finally, we poured time into incremental performance wins but failed to appreciably scale the total number of files the sync engine could manage.

There were a few root causes for these issues, but the most important one was Sync Engine Classic's *data model*. The data model was designed for a simpler world without sharing, and files lacked a stable identifier that would be preserved across moves. There were few consistency guarantees, and we'd spend hours debugging issues where something theoretically possible but "extremely unlikely" would show up in production. Changing the foundational nouns of a system is often impossible to do in small pieces, and we quickly ran out of effective incremental improvements.

Next, the system was not designed for testability. We relied on slow rollouts and debugging issues in the field rather than automated pre-release testing. Sync Engine Classic's permissive data model meant we couldn't check much in stress tests, since there were large sets of undesirable yet still legal outcomes we couldn't assert against. Having a strong data model with tight invariants is immensely valuable for testing, since it's always easy to check if your system is in a valid state.

We discussed above how sync is a very concurrent problem, and testing and debugging concurrent code is notoriously difficult. Sync Engine Classic's threading-based architecture did not help at all, handing all our scheduling decisions to the OS and making integration tests non-reproducible. In practice, we ended up using very coarse-grained locks held for long periods of time. This architecture sacrificed the benefits of parallelism in order to make the system easier to reason about.

A rewrite checklist

Let's distill the reasons for our decision to rewrite into a "rewrite checklist" that can help navigate this kind of decision for other systems.

Have you exhausted incremental improvements?

☐ Have you tried refactoring code into better modules?

Poor code quality alone isn't a great reason to rewrite a system. Renaming variables and untangling intertwined modules can all be done incrementally, and we spent a lot of time doing this with Sync Engine Classic. Python's dynamism can make this difficult, so we added MyPy annotations as we went to gradually catch more bugs at compile time. But the core primitives of the system remained the same, as refactoring alone cannot change the fundamental data model.

□ **Have you tried improving performance by optimizing hotspots?**

Software often spends most of its time in very little of the code. Many performance issues are not fundamental, and optimizing hotspots identified by a profiler is a great way to incrementally improve performance. We had a team working on performance and scale for months, and they had great results for improving file content transfer performance. But improvements to our memory footprint, like increasing the number of files the system could manage, remained elusive.

□ **Can you deliver incremental value?**

Even if you decide to do a rewrite, can you reduce its risk by delivering intermediate value? Doing so can validate early technical decisions, help the project keep momentum, and lessen the pain of slowed feature development.

Can you pull off a rewrite?

□ **Do you deeply understand and respect the current system?**

It's much easier to write new code than fully understand existing code. So, before embarking on a rewrite, you must deeply understand and respect the "Classic" system. It's the whole reason your team and business is here, and it has years of accumulated wisdom through running in production. Get in there and do archaeology to dig into why everything is the way it is.

□ **Do you have the engineering-hours?**

Rewriting a system from scratch is hard work, and getting to full feature completeness will require a lot of time. Do you have these resources? Do you have the domain experts who understand the current system? Is your organization healthy enough to sustain a project of this magnitude?

□ **Can you accept a slower rate of feature development?**

We didn't completely pause feature development on Sync Engine Classic, but every change on the old system pushed the finish line for the new one further out. We decided on shipping a few projects, and we had to be very intentional about allocating resources for guiding their rollouts without slowing down the rewriting team. We also heavily invested in telemetry for Sync Engine Classic to keep its steady-state maintenance cost to a minimum.

Do you know what you're going towards?

□ **Why will it be better the second time?**

If you've gotten to here, you already understand the old system thoroughly and its lessons to be learned. But a rewrite should also be motivated by changing requirements or business needs. We described earlier how file syncing had changed, but our decision to rewrite was also forward looking. Dropbox understands the growing needs of collaborative users at work, and building new features for these users requires a flexible, robust sync engine.

□ What are your principles for the new system?

Starting from scratch is a great opportunity to reset technical culture for a team. Given our experience operating Sync Engine Classic, we heavily emphasized testing, correctness, and debuggability from the beginning. Encode all of these principles in your data model. We wrote out these principles early in the project's lifespan, and they paid for themselves over and over.

So, what did we build?

Here's a summary of what we achieved with Nucleus. For more details on each one, stay tuned for future blog posts.

- We wrote Nucleus in Rust! Rust has been a force multiplier for our team, and betting on Rust was one of the best decisions we made. More than performance, its ergonomics and focus on correctness has helped us tame sync's complexity. We can encode complex invariants about our system in the type system and have the compiler check them for us.
- Almost all of our code runs on a single thread (the "Control thread") and uses Rust's [futures library](#) for scheduling many concurrent actions on this single thread. We offload work to other threads only as needed: network IO to an event loop, computationally expensive work like hashing to a thread pool, and filesystem IO to a dedicated thread. This drastically reduces the scope and complexity developers must consider when adding new features.
- The Control thread is designed to be entirely deterministic when its inputs and scheduling decisions are fixed. We use this property to fuzz it with pseudorandom simulation testing. With a seed for our random number generator, we can generate random initial filesystem state, schedules, and system perturbations and let the engine run to completion. Then, if we fail any of our sync correctness checks, we can always reproduce the bug from the original seed. We run millions of scenarios every day in our testing infrastructure.
- We redesigned the client-server protocol to have strong consistency. The protocol guarantees the server and client have the same view of the remote filesystem before considering a mutation. Shared folders and files have globally unique identifiers, and clients never observe them in transiently duplicated or missing states. Finally, folders and files support  atomic moves independent of their subtree size. We now have strong consistency checks between the client's and server's view of the remote filesystem, and any discrepancy is a bug.

If you're interested in working on hard systems problems in Rust, [we're hiring](#) for our core sync team.

Thanks to Ben Blum, Anthony Kosner, James Cowling, Josh Warner, Iulia Tamas, and the Sync team for comments and review. And thanks to all current and past members of the Sync team who've contributed to building Nucleus.

// Copy link

Link copied