Considus

TECHNICAL WHITE PAPER

Catchlight for iPhone

Sync with no server and no account

How two devices agree with no server and no account, what the design gives up to get there, and the measured provider constraint it imposes.

Applies to Catchlight 1.0

Introduction

Nearly every encrypted notes app asks you to make an account, and then syncs your writing through a server it runs. That is not laziness. It is the only design most teams have ever seen, and it solves real problems: the server knows which devices exist, it can order changes, it can hold a queue while a phone is in a tunnel.

It also creates a company that holds something of yours, which is the thing the rest of this programme spends its time arguing about.

Catchlight has no account and no server. Your encrypted files go into a folder you already own, on a cloud provider you already pay, and two devices agree with each other by leaving files for one another in that folder. There is no infrastructure in the middle because there is no middle.

This paper is how that works, what it cost to build, and the three places it is genuinely worse than a server. The third one is the interesting part, and it is the reason a whole category of cloud providers does not work with this app.

What this covers, and what it doesn't

This covers the file-based sync design: how the store is laid out, how two processes avoid corrupting each other's writes, how deletion works without a server to remember it, and how a second device gets the key without either device speaking to us. Section 6 covers the provider constraint that this design imposes and the architectural choice that would lift it.

Some things are not here. The cryptography is in The Catchlight encryption architecture and this paper assumes it rather than repeating it. Nothing here is about how a cloud provider stores your files, which is that provider's design and not ours.

The largest limit is section 6 and it is not a footnote. Version 1.0 works with iCloud Drive and Dropbox, the only two I could get working on real hardware, and that is a consequence of the design in this paper rather than a gap in it.

1 What is actually in the folder

Not a database. A folder of ordinary files, which is the whole trick.

FILE	WHAT IT IS
One `.clk` blob per Take	The sealed content, encrypted under that Take's own key
A manifest	The index: what exists, what was deleted (kept 180 days, then pruned), what changed when
Handshake files	Written only during device pairing, and deleted afterwards

The manifest is encrypted too, which took a schema change to get right and is worth stating precisely because it is easy to overclaim. The manifest no longer *states* how many Takes you hold, when you last changed one, or what you deleted. It does not follow that the folder reveals nothing: anyone who can read the folder still sees how many files there are, how large each one is, and the provider's own timestamps on them, and a count of files approximates a count of Takes.

So the sealed manifest removes the explicit record. It does not remove the inference. Who might read your notes covers what a provider can still work out, and that is the accurate short version.

Two details in the manifest's design are worth pulling out, because both are decisions rather than accidents.

The version number and the integrity tag sit outside the encrypted envelope. That looks wrong at first glance, since they are the parts you might most want hidden. The reason is that a reader has to be able to check whether a manifest is intact and whether it is a version this app understands *before* it has the key material to

open it. Put those inside and a corrupted or future-version manifest fails in a way the app cannot diagnose or refuse cleanly. They are outside so that the fail-closed check stays reachable.

Filenames are the Take's UUID. Opaque, rotating filenames were considered and rejected. A random identifier that never changes is exactly as linkable across snapshots as a UUID, so rotation would have bought nothing real while making every file operation harder to reason about. The UUID being visible is stated in the architecture paper's limits section rather than glossed.

2 Two devices, one folder, no referee

A server's most useful property is that it is a single place where things happen in order. Take it away and the ordering problem is yours.

The answer is not clever, and that is deliberate.

File coordination. Apple provides `NSFileCoordinator`, which exists precisely so that two processes touching the same file through a file provider do not tread on each other. Using the platform's own mechanism rather than inventing one is the same instinct that runs through the cryptography: the well-tested path and the fast path should be the same path.

Lock-file serialisation. Writes that must not interleave take a lock file first. It is an old technique and it is old because it works on any filesystem, including one that is really a cloud provider pretending to be a filesystem.

A watermark for timing. Each device records how far through the folder's history it has read. A sync is then "show me what changed after this point" rather than "compare everything", which matters when the comparison is happening over somebody's mobile connection.

Tombstones for deletion. This is the one that surprises people. Deleting a file is not enough, because the other device cannot tell the difference between a file you deleted and a file it has not seen yet. Both look like absence. So a delete writes a

small record saying *this Take was deleted*, and the record is what travels. The other device reads the tombstone and removes its copy.

That means a deleted Take leaves a marker behind. It is content-free, it exists so deletion actually propagates, and the alternative is a design where deleting a note on one phone silently restores it from the other a week later.

3 Getting the key to the second device

If there is no server, there is no place to store a key for a new device to collect. And the whole architecture rests on the studio never holding key material, so we cannot be the courier.

The design passes wrapped key material through the user's own folder, and the sequence is in the architecture paper as Figure 3. In short: the new device writes a request, the existing device sees it and asks you to approve, the existing device wraps the master key under a secret only those two devices can derive, and the new device unwraps it. Three controls bound the risk that a provider retains an old version of a transfer file, all of them detailed there.

This is built, tested and not shipped in version 1.0. The cryptography exists in the app. The interface that would let you drive it does not, and the reason is that concurrent-edit validation between two real phones cannot be done without two real phones. Live pairing is Planned for v1.1.

What ships in 1.0 is the other route, and it is the more important one anyway: install the app on a new device, point it at your folder, type your twelve words, and everything decrypts. No server is involved, nothing needs Considus to exist, and it works if the studio disappears.

The Privacy phrase does not travel between devices. It is held under a Keychain attribute that keeps it on the device that made it and out of any backup. So the recovery route above requires the words you wrote down, and there is no other copy anywhere. That is the cost of the design and it is stated on the setup screen for the same reason it is stated here.

4 Why not just run a server

The rejected alternative deserves a fair hearing, because it is what nearly everyone does and they are not fools.

A server gives you push notification of changes rather than polling, ordering without lock files, a queue for offline devices, device management, and the ability to fix a customer's sync problem by looking at it. Every one of those is a genuine benefit and this design gives up all of them.

What it buys is a single property: **there is no infrastructure holding your data, so there is nothing to compel, breach, or shut down.** The company cannot be asked for your notes because it does not have them, and a company that folds does not take your writing with it.

That trade is only worth it for a product whose entire proposition is the property being bought. For most software it would be a bad deal. For this one it is the deal.

There is a second, quieter benefit. **No account means no user database.** No email addresses, no password hashes, no reset flow, and no breach notification to write. The most common way notes leak is not cryptographic and never has been.

5 What this design costs

Read this with section 4, because section 4 alone is an advertisement.

Sync is slower to notice changes. A server pushes, a folder has to be looked at. Apple's file provider does notify, but a change made on another device is seen when the app next looks rather than the instant it happens.

Conflicts are more likely and are handed to you. Two devices editing offline will collide, and with no referee the app cannot decide for you. What Catchlight does is surface both versions and ask, which is correct and is also work you would not have to do with a server arbitrating. Silently keeping one is the alternative, and it is worse.

Diagnosis is harder. If your sync misbehaves I cannot look at your data, because I cannot see it. That is the design working as intended and it is genuinely worse for support.

It depends on a provider we do not control. If your provider has an outage, changes its file-provider behaviour, or breaks its Files app integration, sync stops and there is nothing I can do about it. This has already happened to one provider.

And it restricts which providers work at all, which needs its own section.

6 The provider constraint, and how we found it

This is the part I would most want a reviewer to read, because it is where the design cost something real and where the documentation was confidently wrong.

The original plan, written from Apple's and the providers' own documentation, assumed that any cloud service with a file-provider extension would appear in the folder picker and be selectable. Apple's material supports that reading. It is false on device.

Tested on real hardware in June 2026, against every provider installed:

PROVIDER	SELECTABLE AS A FOLDER	WORKS IN THE FILES APP
iCloud Drive	Yes	Yes
Dropbox	Yes	Yes
Google Drive	**No, greyed out**	Yes
OneDrive	**No, greyed out**	Yes
Box	**No, greyed out**	Yes
pCloud	**No, greyed out**	Yes
MEGA	**No, greyed out**	Yes
Proton Drive	**No**, integration broken provider-side	No

The greyed-out providers work perfectly well in the Files app. They grey out *only* when an app asks for persistent in-place access to a whole folder.

The cause is specific. Handing an app a folder as a persistent workspace requires the provider to have implemented the modern replicated file-provider extension fully. As far as I can tell from testing, only iCloud Drive and Dropbox have. The others probably implement the mainstream case, which is editing a single document in place, and that is a different capability. **The presence of an extension does not imply folder access**, and no setting or code change on our side un-greys them.

So version 1.0 supports iCloud Drive and Dropbox. That is a platform constraint rather than a defect, and it is stated in onboarding rather than discovered.

What that list is, and what it is not. It is what I could test on real hardware and watch work. It is not a judgement about the providers that are not on it, and it is not a decision anyone here made about which ones deserve supporting. No provider publishes whether it has implemented the replicated extension fully, so there is no list to consult and no announcement to wait for. One of them could ship it tomorrow and I would find out the same way you would, by trying. A provider missing from the table above might already work, one on it could change, and **if you find another that works I would genuinely like to know**, because trying them one at a time is the only way that table ever grows.

The architectural trade. It is the multi-file layout in section 1 that requires folder access. A single encrypted document, edited in place, would very likely work with every provider in that table, because single-document editing is the capability they all support. It would cost whole-store reads and writes and a coarser conflict model in exchange. That is a real decision for a future rewrite and it is recorded so the rewrite does not re-inherit the assumption that started this section.

Two workarounds were considered and rejected. Picking a file and using its parent folder fails, because the access granted is bound to that file and not its directory, so the manifest and blobs cannot be written alongside it. Per-provider native SDKs

work and reintroduce OAuth keys and per-provider code, which is the complexity this design exists to avoid.

7 What this doesn't protect you against

An unlocked phone. Everything here concerns data at rest and in transit.

Provider metadata. Section 1 says what remains visible. Encrypting the manifest narrowed it and did not close it.

Losing the phrase. No server means no recovery. Section 3 is explicit and so is the app.

A provider retaining old versions. Three controls bound the pairing case specifically, and they are in the architecture paper. Outside that flow, a provider that keeps old file versions keeps old ciphertext, which stays encrypted under the same keys.

Concurrent editing across two devices in v1.0. Single-device sync is what ships. The conflict machinery exists and is used, but the two-device path is Planned for v1.1 and has not been validated on two real phones.

8 Don't take my word for any of it

Open your sync folder and look. In the Files app or on a computer. You will see files. Open one, there is not a readable word in it. Count them, and notice that the count itself is visible, which is the point section 1 makes.

Turn off the network for a day. Everything works, because none of it was going through us. That is test 3 in How to tell whether a notes app is actually private.

Try to make an account. There isn't one. There is no reset flow because there is nothing to reset.

Seeing your provider in the Files app proves nothing. Every provider in section 6's table works in the Files app, including every one Catchlight cannot use. Browsing a folder and handing an app a folder are different questions, and the only way to settle the second is to open Catchlight and try to add it. It takes a few seconds and it either appears or it is greyed out.

Read the code. Catchlight is open source under Apache 2.0. The sync engine, the manifest schema and the handshake are all in there.

References

Platform documentation

Apple, NSFileCoordinator, the coordination mechanism section 2 relies on

Apple, NSFileProviderReplicatedExtension, the interface a provider must implement fully for section 6's folder access

Apple Developer Forums, thread 691738, third-party providers in the document picker

Companion documents

The Catchlight encryption architecture, for the key hierarchy, the ciphers and the handshake sequence this paper assumes

Who might read your notes, for what a cloud provider can still observe

How to tell whether a notes app is actually private, for the tests in section 8 applied to any app

Version history

VERSION	DATE	CHANGE
1.2	5 September 2026	Founder-voice pass. Three prose semicolons replaced, one honesty-signalling phrase cut, and "the architectural lever" renamed to "the architectural trade" because lever is banned as a metaphor for an intervention. No technical change
1.1	2 September 2026	Section 6 now reports what testing showed rather than asserting what other providers are capable of. The June table is unchanged
1.0	1 September 2026	First release. Describes the v1.0 single-device path, live pairing is Planned v1.1