

Chapter 26 - Data Integrity: What You Read Is What You Wrote



1. [Table of Contents](#)
2. [Foreword](#)
3. [Preface](#)
4. [Part I - Introduction](#)
5. [1. Introduction](#)
6. [2. The Production Environment at Google, from the Viewpoint of an SRE](#)
7. [Part II - Principles](#)
8. [3. Embracing Risk](#)
9. [4. Service Level Objectives](#)
10. [5. Eliminating Toil](#)
11. [6. Monitoring Distributed Systems](#)
12. [7. The Evolution of Automation at Google](#)
13. [8. Release Engineering](#)
14. [9. Simplicity](#)
15. [Part III - Practices](#)
16. [10. Practical Alerting](#)
17. [11. Being On-Call](#)
18. [12. Effective Troubleshooting](#)
19. [13. Emergency Response](#)
20. [14. Managing Incidents](#)
21. [15. Postmortem Culture: Learning from Failure](#)
22. [16. Tracking Outages](#)
23. [17. Testing for Reliability](#)
24. [18. Software Engineering in SRE](#)
25. [19. Load Balancing at the Frontend](#)
26. [20. Load Balancing in the Datacenter](#)
27. [21. Handling Overload](#)
28. [22. Addressing Cascading Failures](#)
29. [23. Managing Critical State: Distributed Consensus for Reliability](#)
30. [24. Distributed Periodic Scheduling with Cron](#)
31. [25. Data Processing Pipelines](#)
32. [26. Data Integrity: What You Read Is What You Wrote](#)
33. [27. Reliable Product Launches at Scale](#)
34. [Part IV - Management](#)
35. [28. Accelerating SREs to On-Call and Beyond](#)
36. [29. Dealing with Interrupts](#)
37. [30. Embedding an SRE to Recover from Operational Overload](#)
38. [31. Communication and Collaboration in SRE](#)
39. [32. The Evolving SRE Engagement Model](#)
40. [Part V - Conclusions](#)
41. [33. Lessons Learned from Other Industries](#)
42. [34. Conclusion](#)
43. [Appendix A. Availability Table](#)
44. [Appendix B. A Collection of Best Practices for Production Services](#)
45. [Appendix C. Example Incident State Document](#)
46. [Appendix D. Example Postmortem](#)
47. [Appendix E. Launch Coordination Checklist](#)
48. [Appendix F. Example Production Meeting Minutes](#)
49. [Bibliography](#)

Data Integrity: What You Read Is What You Wrote

Written by Raymond Blum and Rhandeev Singh

Edited by Betsy Beyer

What is "data integrity"? When users come first, data integrity is whatever users think it is.

We might say *data integrity is a measure of the accessibility and accuracy of the datastores needed to provide users with an adequate level of service*. But this definition is insufficient.

For instance, if a user interface bug in Gmail displays an empty mailbox for too long, users might believe data has been lost. Thus, even if no data was *actually* lost, the world would question Google's ability to act as a responsible steward of data, and the viability of cloud computing would be threatened. Were Gmail to display an error or maintenance message for too long while "only a bit of metadata" is repaired, the trust of Google's users would similarly erode.

How long is "too long" for data to be unavailable? As demonstrated by an actual Gmail incident in 2011 [\[Hic11\]](#), four days is a long time—perhaps "too long." Subsequently, we believe 24 hours is a good starting point for establishing the threshold of "too long" for Google Apps.

Similar reasoning applies to applications like Google Photos, Drive, Cloud Storage, and Cloud Datastore, because users don't necessarily draw a distinction between these discrete products (reasoning, "this product is still Google" or "Google, Amazon, whatever; this product is still part of the cloud"). Data loss, data corruption, and extended unavailability are typically indistinguishable to users. Therefore, data integrity applies to all types of data across all services. When considering data integrity, what matters is that *services in the cloud remain accessible to users. User access to data is especially important*.

Data Integrity's Strict Requirements

When considering the reliability needs of a given system, it may seem that uptime (service availability) needs are stricter than those of data integrity. For example, users may find an hour of email downtime unacceptable, whereas they may live grumpily with a four-day time window to recover a mailbox. However, there's a more appropriate way to consider the demands of uptime versus data integrity.

An SLO of 99.99% uptime leaves room for only an hour of downtime in a whole year. This SLO sets a rather high bar, which likely exceeds the expectations of most Internet and Enterprise users.

In contrast, an SLO of 99.99% good bytes in a 2 GB artifact would render documents, executables, and databases corrupt (up to 200 KB garbled). This amount of corruption is *catastrophic* in the majority of cases—resulting in executables with random opcodes and completely unloadable databases.

From the user perspective, then, every service has independent uptime and data integrity requirements, even if these requirements are implicit. The worst time to disagree with users about these requirements is after the demise of their data!



Figure 26-1.

To revise our earlier definition of data integrity, we might say that *data integrity means that services in the cloud remain accessible to users. User access to data is especially important, so this access should remain in perfect shape*.

Now, suppose an artifact were corrupted or lost exactly once a year. If the loss were unrecoverable, uptime of the affected artifact is *lost* for that year. The most likely means to avoid any such loss is through

proactive detection, coupled with rapid repair.

In an alternate universe, suppose the corruption were immediately detected before users were affected and that the artifact was removed, fixed, and returned to service within half an hour. Ignoring any other downtime during that 30 minutes, such an object would be 99.99% available that year.

Astonishingly, at least from the user perspective, in this scenario, data integrity is still 100% (or close to 100%) during the accessible lifetime of the object. As demonstrated by this example, *the secret to superior data integrity is proactive detection and rapid repair and recovery.*

Choosing a Strategy for Superior Data Integrity

There are many possible strategies for rapid detection, repair, and recovery of lost data. All of these strategies trade uptime against data integrity with respect to affected users. Some strategies work better than others, and some strategies require more complex engineering investment than others. With so many options available, which strategies should you utilize? The answer depends on your computing paradigm.

Most cloud computing applications seek to optimize for some combination of uptime, latency, scale, velocity, and privacy. To provide a working definition for each of these terms:

Uptime

Also referred to as *availability*, the proportion of time a service is usable by its users.

Latency

How responsive a service appears to its users.

Scale

A service's volume of users and the mixture of workloads the service can handle before latency suffers or the service falls apart.

Velocity

How fast a service can innovate to provide users with superior value at reasonable cost.

Privacy

This concept imposes complex requirements. As a simplification, this chapter limits its scope in discussing privacy to data deletion: data must be destroyed within a reasonable time after users delete it.

Many cloud applications continually evolve atop a mixture of ACID^{[122](#)} and BASE^{[123](#)} APIs to meet the demands of these five components.^{[124](#)} BASE allows for higher availability than ACID, in exchange for a softer distributed consistency guarantee. Specifically, BASE only guarantees that once a piece of data is no longer updated, its value will *eventually* become consistent across (potentially distributed) storage locations.

The following scenario provides an example of how trade-offs between uptime, latency, scale, velocity, and privacy might play out.

When velocity trumps other requirements, the resulting applications rely on an arbitrary collection of APIs that are most familiar to the particular developers working on the application.

For example, an application may take advantage of an efficient BLOB¹²⁵ storage API, such as Blobstore, that neglects distributed consistency in favor of scaling to heavy workloads with high uptime, low latency, and at low cost. To compensate:

- The same application may entrust small amounts of authoritative metadata pertaining to its blobs to a higher latency, less available, more costly Paxos-based service such as Megastore [\[Bak11\]](#), [\[Lam98\]](#).
- Certain clients of the application may cache some of that metadata locally and access blobs directly, shaving latency still further from the vantage point of users.
- Another application may keep metadata in Bigtable, sacrificing strong distributed consistency because its developers happened to be familiar with Bigtable.

Such cloud applications face a variety of data integrity challenges at runtime, such as referential integrity between datastores (in the preceding example, Blobstore, Megastore, and client-side caches). The vagaries of high velocity dictate that schema changes, data migrations, the piling of new features atop old features, rewrites, and evolving integration points with other applications collude to produce an environment riddled with complex relationships between various pieces of data that no single engineer fully groks.

To prevent such an application's data from degrading before its users' eyes, a system of out-of-band checks and balances is needed within and between its datastores. [Third Layer: Early Detection](#) discusses such a system.

In addition, if such an application relies on independent, uncoordinated backups of several datastores (in the preceding example, Blobstore and Megastore), then its ability to make effective use of restored data during a data recovery effort is complicated by the variety of relationships between restored and live data. Our example application would have to sort through and distinguish between restored blobs versus live Megastore, restored Megastore versus live blobs, restored blobs versus restored Megastore, and interactions with client-side caches.

In consideration of these dependencies and complications, how many resources should be invested in data integrity efforts, and where?

Backups Versus Archives

Traditionally, companies "protect" data against loss by investing in backup strategies. However, the real focus of such backup efforts should be data recovery, which distinguishes *real* backups from archives. As is sometimes observed: No one really *wants* to make backups; what people *really* want are *restores*.

Is your "backup" really an archive, rather than appropriate for use in disaster recovery?



Figure 26-2.

The most important difference between backups and archives is that backups *can* be loaded back into an application, while archives *cannot*. Therefore, backups and archives have quite differing use cases.

Archives safekeep data for long periods of time to meet auditing, discovery, and compliance needs. Data recovery for such purposes generally doesn't need to complete within uptime requirements of a service. For example, you might need to retain financial transaction data for seven years. To achieve this goal, you could move accumulated audit logs to long-term archival storage at an offsite location once a month. Retrieving and recovering the logs during a month-long financial audit may take a week, and this weeklong time window for recovery may be acceptable for an archive.

On the other hand, when disaster strikes, data must be recovered from *real backups* quickly, preferably

well within the uptime needs of a service. Otherwise, affected users are left without useful access to the application from the onset of the data integrity issue until the completion of the recovery effort.

It's also important to consider that because the most recent data is at risk until safely backed up, it may be optimal to schedule real backups (as opposed to archives) to occur daily, hourly, or more frequently, using full and incremental or continuous (streaming) approaches.

Therefore, when formulating a backup strategy, consider how quickly you need to be able to recover from a problem, and how much recent data you can afford to lose.

Requirements of the Cloud Environment in Perspective

Cloud environments introduce a unique combination of technical challenges:

- If the environment uses a mixture of transactional and nontransactional backup and restore solutions, recovered data won't necessarily be correct.
- If services must evolve without going down for maintenance, different versions of business logic may act on data in parallel.
- If interacting services are versioned independently, incompatible versions of different services may interact momentarily, further increasing the chance of accidental data corruption or data loss.

In addition, in order to maintain economy of scale, service providers must provide only a limited number of APIs. These APIs must be simple and easy to use for the vast majority of applications, or few customers will use them. At the same time, the APIs must be robust enough to understand the following:

- Data locality and caching
- Local and global data distribution
- Strong and/or eventual consistency
- Data durability, backup, and recovery

Otherwise, sophisticated customers can't migrate applications to the cloud, and simple applications that grow complex and large will need complete rewrites in order to use different, more complex APIs.

Problems arise when the preceding API features are used in certain combinations. If the service provider doesn't solve these problems, then the applications that run into these challenges must identify and solve them independently.

Google SRE Objectives in Maintaining Data Integrity and Availability

While SRE's goal of "maintaining integrity of persistent data" is a good vision, we thrive on concrete objectives with measurable indicators. SRE defines key metrics that we use to set expectations for the capabilities of our systems and processes through tests and to track their performance during an actual event.

Data Integrity Is the Means; Data Availability Is the Goal

Data integrity refers to the accuracy and consistency of data throughout its lifetime. Users need to know that information will be correct and won't change in some unexpected way from the time it's first recorded to the last time it's observed. But is such assurance enough?

Consider the case of an email provider who suffered a weeklong data outage [\[Kinc09\]](#). Over the space of 10 days, users had to find other, temporary methods of conducting their business with the expectation that they'd soon return to their established email accounts, identities, and accumulated histories.

Then, the worst possible news arrived: the provider announced that despite earlier expectations, the trove of past email and contacts was in fact gone—evaporated and never to be seen again. It seemed that a series of mishaps in managing data integrity had conspired to leave the service provider with no usable backups. Furious users either stuck with their interim identities or established new identities, abandoning their troubled former email provider.

But wait! Several days after the declaration of absolute loss, the provider announced that the users' personal information *could* be recovered. There was no data loss; this was only an outage. All was well!

Except, *all was not well*. User data had been preserved, but the data was not accessible by the people who needed it for too long.

The moral of this example: From the user's point of view, data integrity without expected and regular data availability is effectively the same as having no data at all.

Delivering a Recovery System, Rather Than a Backup System

Making backups is a classically neglected, delegated, and deferred task of system administration. Backups aren't a high priority for anyone—they're an ongoing drain on time and resources, and yield no immediate visible benefit. For this reason, a lack of diligence in implementing a backup strategy is typically met with a sympathetic eye roll. One might argue that, like most measures of protection against low-risk dangers, such an attitude is pragmatic. The fundamental problem with this lackadaisical strategy is that the dangers it entails may be low risk, but they are also high impact. When your service's data is unavailable, your response can make or break your service, product, and even your company.

Instead of focusing on the thankless job of taking a backup, it's much more useful, not to mention easier, to motivate participation in taking backups by concentrating on a task with a visible payoff: the *restore*! *Backups are a tax*, one paid on an ongoing basis for the municipal service of guaranteed data availability. Instead of emphasizing the tax, draw attention to the service the tax funds: data availability. We don't make teams "practice" their backups, instead:

- Teams define service level objectives (SLOs) for data availability in a variety of failure modes.
- A team practices and demonstrates their ability to meet those SLOs.

Types of Failures That Lead to Data Loss

As illustrated by [Figure 26-3](#), at a very high level, there are 24 distinct types of failures when the 3 factors can occur in any combination. You should consider each of these potential failures when designing a data integrity program. The factors of data integrity failure modes are as follows:

An unrecoverable loss of data may be caused by a number of factors: user action, operator error, application bugs, defects in infrastructure, faulty hardware, or site catastrophes.

Scope

Some losses are widespread, affecting many entities. Some losses are narrow and directed, deleting or corrupting data specific to a small subset of users.

Rate

Some data losses are a big bang event (for example, 1 million rows are replaced by only 10 rows in a single minute), whereas some data losses are creeping (for example, 10 rows of data are deleted every minute over the course of weeks).




Figure 26-3. The factors of data integrity failure modes

An effective restore plan must account for any of these failure modes occurring in any conceivable combination. What may be a perfectly effective strategy for guarding against a data loss caused by a creeping application bug may be of no help whatsoever when your colocation datacenter catches fire.

A study of 19 data recovery efforts at Google found that the most common user-visible data loss scenarios involved data deletion or loss of referential integrity caused by software bugs. The most challenging variants involved low-grade corruption or deletion that was discovered weeks to months after the bugs were first released into the production environment. Therefore, the safeguards Google employs should be well suited to prevent or recover from these types of loss.

To recover from such scenarios, a large and successful application needs to retrieve data for perhaps millions of users spread across days, weeks, or months. The application may also need to recover each affected artifact to a unique point in time. This data recovery scenario is called "point-in-time recovery" outside Google, and "time-travel" inside Google.

A backup and recovery solution that provides point-in-time recovery for an application across its ACID and BASE datastores while meeting strict uptime, latency, scalability, velocity, and cost goals is a chimera today!

Solving this problem with your own engineers entails sacrificing velocity. Many projects compromise by adopting a tiered backup strategy without point-in-time recovery. For instance, the APIs beneath your application may support a variety of data recovery mechanisms. Expensive local "snapshots" may provide limited protection from application bugs and offer quick restoration functionality, so you might retain a few days of such local "snapshots," taken several hours apart. Cost-effective full and incremental copies every two days may be retained longer. Point-in-time recovery is a very nice feature to have if one or more of these strategies support it.

Consider the data recovery options provided by the cloud APIs you are about to use. Trade point-in-time recovery against a tiered strategy if necessary, but don't resort to not using either! If you can have both features, use both features. Each of these features (or both) will be valuable at some point.

Challenges of Maintaining Data Integrity Deep and Wide

In designing a data integrity program, it's important to recognize that *replication and redundancy are not recoverability*.

Scaling issues: Fulls, incrementals, and the competing forces of backups and restores

A classic but flawed response to the question "Do you have a backup?" is "We have something even better than a backup—replication!" Replication provides many benefits, including locality of data and protection from a site-specific disaster, but it can't protect you from many sources of data loss. Datastores that automatically sync multiple replicas guarantee that a corrupt database row or errant delete are pushed to all of your copies, likely before you can isolate the problem.

To address this concern, you might make nonserving copies of your data in some other format, such as frequent database exports to a native file. This additional measure adds protection from the types of errors replication doesn't protect against—user errors and application-layer bugs—but does nothing to guard against losses introduced at a lower layer. This measure also introduces a risk of bugs during data conversion (in both directions) and during storage of the native file, in addition to possible mismatches in semantics between the two formats. Imagine a zero-day attack^{[126](#)} at some low level of your stack, such as the filesystem or device driver. Any copies that rely on the compromised software component, including

the database exports that were written to the same filesystem that backs your database, are vulnerable.

Thus, we see that diversity is key: protecting against a failure at layer X requires storing data on diverse components at that layer. Media isolation protects against media flaws: a bug or attack in a disk device driver is unlikely to affect tape drives. If we could, we'd make backup copies of our valuable data on clay tablets.^{[127](#)}

The forces of data freshness and restore completion compete against comprehensive protection. The further down the stack you push a snapshot of your data, the longer it takes to make a copy, which means that the frequency of copies decreases. At the database level, a transaction may take on the order of seconds to replicate. Exporting a database snapshot to the filesystem underneath may take 40 minutes. A full backup of the underlying filesystem may take hours.

In this scenario, you may lose up to 40 minutes of the most recent data when you restore the latest snapshot. A restore from the filesystem backup might incur hours of missing transactions. Additionally, restoring probably takes as long as backing up, so actually loading the data might take hours. You'd obviously like to have the freshest data back as quickly as possible, but depending on the type of failure, that freshest and most immediately available copy might not be an option.

Retention

Retention—how long you keep copies of your data around—is yet another factor to consider in your data recovery plans.

While it's likely that you or your customers will quickly notice the sudden emptying of an entire database, it might take days for a more gradual loss of data to attract the right person's attention. Restoring the lost data in the latter scenario requires snapshots taken further back in time. When reaching back this far, you'll likely want to merge the restored data with the current state. Doing so significantly complicates the restore process.

How Google SRE Faces the Challenges of Data Integrity

Similar to our assumption that Google's underlying systems are prone to failure, we assume that any of our protection mechanisms are also subject to the same forces and can fail in the same ways and at the most inconvenient of times. Maintaining a guarantee of data integrity at large scale, a challenge that is further complicated by the high rate of change of the involved software systems, requires a number of complementary but uncoupled practices, each chosen to offer a high degree of protection on its own.

The 24 Combinations of Data Integrity Failure Modes

Given the many ways data can be lost (as described previously), there is no silver bullet that guards against the many combinations of failure modes. Instead, you need defense in depth. Defense in depth comprises multiple layers, with each successive layer of defense conferring protection from progressively less common data loss scenarios. [Figure 26-4](#) illustrates an object's journey from soft deletion to destruction, and the data recovery strategies that should be employed along this journey to ensure defense in depth.

The first layer is *soft deletion* (or "lazy deletion" in the case of developer API offerings), which has proven to be an effective defense against inadvertent data deletion scenarios. The second line of defense is *backups and their related recovery methods*. The third and final layer is *regular data validation*, covered in [Third Layer: Early Detection](#). Across all these layers, the presence of *replication* is occasionally useful for data recovery in specific scenarios (although data recovery plans should not rely upon replication).




Figure 26-4. An object's journey from soft deletion to destruction

First Layer: Soft Deletion

When velocity is high and privacy matters, bugs in applications account for the vast majority of data loss and corruption events. In fact, data deletion bugs may become so common that the ability to undelete data for a limited time becomes the primary line of defense against the majority of otherwise permanent, inadvertent data loss.

Any product that upholds the privacy of its users must allow the users to delete selected subsets and/or all of their data. Such products incur a support burden due to accidental deletion. Giving users the ability to undelete their data (for example, via a trash folder) reduces but cannot completely eliminate this support burden, particularly if your service also supports third-party add-ons that can also delete data.

Soft deletion can dramatically reduce or even completely eliminate this support burden. Soft deletion means that deleted data is immediately marked as such, rendering it unusable by all but the application's administrative code paths. Administrative code paths may include legal discovery, hijacked account recovery, enterprise administration, user support, and problem troubleshooting and its related features. Conduct soft deletion when a user empties his or her trash, and provide a user support tool that enables authorized administrators to undelete any items accidentally deleted by users. Google implements this strategy for our most popular productivity applications; otherwise, the user support engineering burden would be untenable.

You can extend the soft deletion strategy even further by offering users the option to recover deleted data. For example, the Gmail trash bin allows users to access messages that were deleted fewer than 30 days ago.

Another common source of unwanted data deletion occurs as a result of account hijacking. In account hijacking scenarios, a hijacker commonly deletes the original user's data before using the account for spamming and other unlawful purposes. When you combine the commonality of accidental user deletion with the risk of data deletion by hijackers, the case for a programmatic soft deletion and undeletion interface within and/or beneath your application becomes clear.

Soft deletion implies that once data is marked as such, it is destroyed after a reasonable delay. The length of the delay depends upon an organization's policies and applicable laws, available storage resources and cost, and product pricing and market positioning, especially in cases involving much short-lived data. Common choices of soft deletion delays are 15, 30, 45, or 60 days. In Google's experience, the majority of account hijacking and data integrity issues are reported or detected within 60 days. Therefore, the case for soft deleting data for longer than 60 days may not be strong.

Google has also found that the most devastating acute data deletion cases are caused by application developers unfamiliar with existing code but working on deletion-related code, especially batch processing pipelines (e.g., an offline MapReduce or Hadoop pipeline). It's advantageous to design your interfaces to hinder developers unfamiliar with your code from circumventing soft deletion features with new code. One effective way of achieving this is to implement cloud computing offerings that include built-in soft deletion and undeletion APIs, making sure to *enable said feature*.¹²⁸ Even the best armor is useless if you don't put it on.

Soft deletion strategies cover data deletion features in consumer products like Gmail or Google Drive, but what if you support a cloud computing offering instead? Assuming your cloud computing offering already supports a programmatic soft deletion and undeletion feature with reasonable defaults, the remaining accidental data deletion scenarios will originate in mistakes made by your own internal developers or your developer customers.

In such cases, it can be useful to introduce an additional layer of soft deletion, which we will refer to as "lazy deletion." You can think of lazy deletion as behind the scenes purging, controlled by the storage system (whereas soft deletion is controlled by and expressed to the client application or service). In a lazy deletion scenario, data that is deleted by a cloud application becomes immediately inaccessible to the application, but is preserved by the cloud service provider for up to a few weeks before destruction. Lazy deletion isn't advisable in all defense in depth strategies: a long lazy deletion period is costly in systems with much short-lived data, and impractical in systems that must guarantee destruction of deleted data within a reasonable time frame (i.e., those that offer privacy guarantees).

To sum up the first layer of defense in depth:

- A trash folder that allows users to undelete data is the primary defense against user error.
- Soft deletion is the primary defense against developer error and the secondary defense against user error.
- In developer offerings, lazy deletion is the primary defense against internal developer error and the secondary defense against external developer error.

What about *revision history*? Some products provide the ability to revert items to previous states. When such a feature is available to users, it is a form of trash. When available to developers, it may or may not substitute for soft deletion, depending on its implementation.

At Google, revision history has proven useful in recovering from certain data corruption scenarios, but not in recovering from most data loss scenarios involving accidental deletion, programmatic or otherwise. This is because some revision history implementations treat deletion as a special case in which previous states must be removed, as opposed to mutating an item whose history may be retained for a certain time period. To provide adequate protection against unwanted deletion, apply the lazy and/or soft deletion principles to revision history also.

Second Layer: Backups and Their Related Recovery Methods

Backups and data recovery are the second line of defense after soft deletion. The most important principle in this layer is that backups don't matter; what matters is recovery. The factors supporting successful recovery should drive your backup decisions, not the other way around.

In other words, the scenarios in which you want your backups to help you recover should dictate the following:

- Which backup and recovery methods to use
- How frequently you establish restore points by taking full or incremental backups of your data
- Where you store backups
- How long you retain backups

How much recent data can you afford to lose during a recovery effort? The less data you can afford to lose, the more serious you should be about an incremental backup strategy. In one of Google's most extreme cases, we used a near-real-time streaming backup strategy for an older version of Gmail.

Even if money isn't a limitation, frequent full backups are expensive in other ways. Most notably, they impose a compute burden on the live datastores of your service while it's serving users, driving your service closer to its scalability and performance limits. To ease this burden, you can take full backups during off-peak hours, and then a series of incremental backups when your service is busier.

How quickly do you need to recover? The faster your users need to be rescued, the more local your backups should be. Often Google retains costly but quick-to-restore snapshots¹²⁹ for very short periods of time within the storage instance, and stores less recent backups on random access distributed storage

within the same (or nearby) datacenter for a slightly longer time. Such a strategy alone would not protect from site-level failures, so those backups are often transferred to nearline or offline locations for a longer time period before they're expired in favor of newer backups.

How far back should your backups reach? Your backup strategy becomes more costly the further back you reach, while the scenarios from which you can hope to recover increase (although this increase is subject to diminishing returns).

In Google's experience, low-grade data mutation or deletion bugs within application code demand the furthest reaches back in time, as some of those bugs were noticed months after the first data loss began. Such cases suggest that you'd like the ability to reach back in time as far as possible.

On the flip side, in a high-velocity development environment, changes to code and schema may render older backups expensive or impossible to use. Furthermore, it is challenging to recover different subsets of data to different restore points, because doing so would involve multiple backups. Yet, that is exactly the sort of recovery effort demanded by low-grade data corruption or deletion scenarios.

The strategies described in [Third Layer: Early Detection](#) are meant to speed detection of low-grade data mutation or deletion bugs within application code, at least partly warding off the need for this type of complex recovery effort. Still, how do you confer reasonable protection before you know what kinds of issues to detect? Google chose to draw the line between 30 and 90 days of backups for many services. Where a service falls within this window depends on its tolerance for data loss and its relative investments in early detection.

To sum up our advice for guarding against the 24 combinations of data integrity failure modes: addressing a broad range of scenarios at reasonable cost demands a tiered backup strategy. The first tier comprises many frequent and quickly restored backups stored closest to the live datastores, perhaps using the same or similar storage technologies as the data sources. Doing so confers protection from the majority of scenarios involving software bugs and developer error. Due to relative expense, backups are retained in this tier for anywhere from hours to single-digit days, and may take minutes to restore.

The second tier comprises fewer backups retained for single-digit or low double-digit days on random access distributed filesystems local to the site. These backups may take hours to restore and confer additional protection from mishaps affecting particular storage technologies in your serving stack, but not the technologies used to contain the backups. This tier also protects against bugs in your application that are detected too late to rely upon the first tier of your backup strategy. If you are introducing new versions of your code to production twice a week, it may make sense to retain these backups for at least a week or two before deleting them.

Subsequent tiers take advantage of nearline storage such as dedicated tape libraries and offsite storage of the backup media (e.g., tapes or disk drives). Backups in these tiers confer protection against site-level issues, such as a datacenter power outage or distributed filesystem corruption due to a bug.

It is expensive to move large amounts of data to and from tiers. On the other hand, storage capacity at the later tiers does not contend with growth of the live production storage instances of your service. As a result, backups in these tiers tend to be taken less frequently but retained longer.

Overarching Layer: Replication

In an ideal world, every storage instance, including the instances containing your backups, would be replicated. During a data recovery effort, the last thing you want is to discover is that your backups themselves lost the needed data or that the datacenter containing the most useful backup is under maintenance.

As the volume of data increases, replication of every storage instance isn't always feasible. In such cases, it makes sense to stagger successive backups across different sites, each of which may fail independently, and to write your backups using a redundancy method such as RAID, Reed-Solomon erasure codes, or GFS-style replication.^{[130](#)}

When choosing a system of redundancy, don't rely upon an infrequently used scheme whose only "tests" of efficacy are your own infrequent data recovery attempts. Instead, choose a popular scheme that's in common and continual use by many of its users.

1T Versus 1E: Not "Just" a Bigger Backup

Processes and practices applied to volumes of data measured in T (terabytes) don't scale well to data measured in E (exabytes). Validating, copying, and performing round-trip tests on a few gigabytes of structured data is an interesting problem. However, assuming that you have sufficient knowledge of your schema and transaction model, this exercise doesn't present any special challenges. You typically just need to procure the machine resources to iterate over your data, perform some validation logic, and delegate enough storage to hold a few copies of your data.

Now let's up the ante: instead of a few gigabytes, let's try securing and validating 700 petabytes of structured data. Assuming ideal SATA 2.0 performance of 300 MB/s, a single task that iterates over all of your data and performs even the most basic of validation checks will take 8 decades. Making a few full backups, assuming you have the media, is going to take at least as long. Restore time, with some post-processing, will take even longer. We're now looking at almost a full century to restore a backup that was up to 80 years old when you started the restore. Obviously, such a strategy needs to be rethought.

The most common and largely effective technique used to back up massive amounts of data is to establish "trust points" in your data—portions of your stored data that are verified after being rendered immutable, usually by the passage of time. Once we know that a given user profile or transaction is fixed and won't be subject to further change, we can verify its internal state and make suitable copies for recovery purposes. You can then make incremental backups that only include data that has been modified or added since your last backup. This technique brings your backup time in line with your "mainline" processing time, meaning that frequent incremental backups can save you from the 80-year monolithic verify and copy job.

However, remember that we care about *restores*, not backups. Let's say that we took a full backup three years ago and have been making daily incremental backups since. A full restore of our data will serially process a chain of over 1,000 highly interdependent backups. Each independent backup incurs additional risk of failure, not to mention the logistical burden of scheduling and the runtime cost of those jobs.

Another way we can reduce the wall time of our copying and verification jobs is to distribute the load. If we shard our data well, it's possible to run N tasks in parallel, with each task responsible for copying and verifying $1/N$ th of our data. Doing so requires some forethought and planning in the schema design and the physical deployment of our data in order to:

- Balance the data correctly
- Ensure the independence of each shard
- Avoid contention among the concurrent sibling tasks

Between distributing the load horizontally and restricting the work to vertical slices of the data demarcated by time, we can reduce those eight decades of wall time by several orders of magnitude, rendering our restores relevant.

Third Layer: Early Detection

“Bad” data doesn’t sit idly by, it propagates. References to missing or corrupt data are copied, links fan out, and with every update the overall quality of your datastore goes down. Subsequent dependent transactions and potential data format changes make restoring from a given backup more difficult as the clock ticks. The sooner you know about a data loss, the easier and more complete your recovery can be.

Challenges faced by cloud developers

In high-velocity environments, cloud application and infrastructure services face many data integrity challenges at runtime, such as:

- Referential integrity between datastores
- Schema changes
- Aging code
- Zero-downtime data migrations
- Evolving integration points with other services

Without conscious engineering effort to track emerging relationships in its data, the data quality of a successful and growing service degrades over time.

Often, the novice cloud developer who chooses a distributed consistent storage API (such as Megastore) delegates the integrity of the application’s data to the distributed consistent algorithm implemented beneath the API (such as Paxos; see [Managing Critical State: Distributed Consensus for Reliability](#)). The developer reasons that the selected API alone will keep the application’s data in good shape. As a result, they unify all application data into a single storage solution that guarantees distributed consistency, avoiding referential integrity problems in exchange for reduced performance and/or scale.

While such algorithms are infallible in theory, their implementations are often riddled with hacks, optimizations, bugs, and educated guesses. For example: in theory, Paxos ignores failed compute nodes and can make progress as long as a quorum of functioning nodes is maintained. In practice, however, ignoring a failed node may correspond to timeouts, retries, and other failure-handling approaches beneath the particular Paxos implementation [\[Cha07\]](#). How long should Paxos try to contact an unresponsive node before timing it out? When a particular machine fails (perhaps intermittently) in a certain way, with a certain timing, and at a particular datacenter, unpredictable behavior results. The larger the scale of an application, the more frequently the application is affected, unbeknownst, by such inconsistencies. If this logic holds true even when applied to Paxos implementations (as has been true for Google), then it must be more true for eventually consistent implementations such as Bigtable (which has also shown to be true). Affected applications have no way to know that 100% of their data is good until they check: trust storage systems, but verify!

To complicate this problem, in order to recover from low-grade data corruption or deletion scenarios, we must recover different subsets of data to different restore points using different backups, while changes to code and schema may render older backups ineffective in high-velocity environments.

Out-of-band data validation

To prevent data quality from degrading before users’ eyes, and to detect low-grade data corruption or data loss scenarios before they become unrecoverable, a system of out-of-band checks and balances is needed both within and between an application’s datastores.

Most often, these data validation pipelines are implemented as collections of map-reductions or Hadoop jobs. Frequently, such pipelines are added as an afterthought to services that are already popular and successful. Sometimes, such pipelines are first attempted when services reach scalability limits and are rebuilt from the ground up. Google has built validators in response to each of these situations.

Shunting some developers to work on a data validation pipeline can slow engineering velocity in the short term. However, devoting engineering resources to data validation endows other developers with the courage to move faster in the long run, because the engineers know that data corruption bugs are less likely to sneak into production unnoticed. Similar to the effects enjoyed when unit tests are introduced early in the project lifecycle, a data validation pipeline results in an overall acceleration of software development projects.

To cite a specific example: Gmail sports a number of data validators, each of which has detected actual data integrity problems in production. Gmail developers derive comfort from the knowledge that bugs introducing inconsistencies in production data are detected within 24 hours, and shudder at the thought of running their data validators less often than daily. These validators, along with a culture of unit and regression testing and other best practices, have given Gmail developers the courage to introduce code changes to Gmail's production storage implementation more frequently than once a week.

Out-of-band data validation is tricky to implement correctly. When too strict, even simple, appropriate changes cause validation to fail. As a result, engineers abandon data validation altogether. If the data validation isn't strict enough, user experience—affecting data corruption can slip through undetected. To find the right balance, only validate invariants that cause devastation to users.

For example, Google Drive periodically validates that file contents align with listings in Drive folders. If these two elements don't align, some files would be missing data—a disastrous outcome. Drive infrastructure developers were so invested in data integrity that they also enhanced their validators to automatically fix such inconsistencies. This safeguard turned a potential emergency "all-hands-on-deck-omigosh-files-are-disappearing!" data loss situation in 2013 into a business as usual, "let's go home and fix the root cause on Monday," situation. By transforming emergencies into business as usual, validators improve engineering morale, quality of life, and predictability.

Out-of-band validators can be expensive at scale. A significant portion of Gmail's compute resource footprint supports a collection of daily validators. To compound this expense, these validators also lower server-side cache hit rates, reducing server-side responsiveness experienced by users. To mitigate this hit to responsiveness, Gmail provides a variety of knobs for rate-limiting its validators and periodically refactors the validators to reduce disk contention. In one such refactoring effort, we cut the contention for disk spindles by 60% without significantly reducing the scope of the invariants they covered. While the majority of Gmail's validators run daily, the workload of the largest validator is divided into 10–14 shards, with one shard validated per day for reasons of scale.

Google Compute Storage is another example of the challenges scale entails to data validation. When its out-of-band validators could no longer finish within a day, Compute Storage engineers had to devise a more efficient way to verify its metadata than use of brute force alone. Similar to its application in data recovery, a tiered strategy can also be useful in out-of-band data validation. As a service scales, sacrifice rigor in daily validators. Make sure that daily validators continue to catch the most disastrous scenarios within 24 hours, but continue with more rigorous validation at reduced frequency to contain costs and latency.

Troubleshooting failed validations can take significant effort. Causes of an intermittent failed validation could vanish within minutes, hours, or days. Therefore, the ability to rapidly drill down into validation audit logs is essential. Mature Google services provide on-call engineers with comprehensive documentation and tools to troubleshoot. For example, on-call engineers for Gmail are provided with:

- A suite of playbook entries describing how to respond to a validation failure alert
- A BigQuery-like investigation tool
- A data validation dashboard

Effective out-of-band data validation demands all of the following:

- Validation job management
- Monitoring, alerts, and dashboards
- Rate-limiting features
- Troubleshooting tools
- Production playbooks
- Data validation APIs that make validators easy to add and refactor

The majority of small engineering teams operating at high velocity can't afford to design, build, and maintain all of these systems. If they are pressured to do so, the result is often fragile, limited, and wasteful one-offs that fall quickly into disrepair. Therefore, structure your engineering teams such that a central infrastructure team provides a data validation framework for multiple product engineering teams. The central infrastructure team maintains the out-of-band data validation framework, while the product engineering teams maintain the custom business logic at the heart of the validator to keep pace with their evolving products.

Knowing That Data Recovery Will Work

When does a light bulb break? When flicking the switch fails to turn on the light? Not always—often the bulb had already failed, and you simply notice the failure at the unresponsive flick of the switch. By then, the room is dark and you've stubbed your toe.

Likewise, your recovery dependencies (meaning mostly, but not only, your backup), may be in a latent broken state, which you aren't aware of until you attempt to recover data.

If you discover that your restore process is broken before you need to rely upon it, you can address the vulnerability before you fall victim to it: you can take another backup, provision additional resources, and change your SLO. But to take these actions proactively, you first have to know they're needed. To detect these vulnerabilities:

- Continuously test the recovery process as part of your normal operations
- Set up alerts that fire when a recovery process fails to provide a heartbeat indication of its success

What can go wrong with your recovery process? Anything and everything—which is why the only test that should let you sleep at night is a full end-to-end test. Let the proof be in the pudding. Even if you recently ran a successful recovery, parts of your recovery process can still break. If you take away just one lesson from this chapter, remember that *you only know that you can recover your recent state if you actually do so*.

If recovery tests are a manual, staged event, testing becomes an unwelcome bit of drudgery that isn't performed either deeply or frequently enough to deserve your confidence. Therefore, automate these tests whenever possible and then run them continuously.

The aspects of your recovery plan you should confirm are myriad:

- Are your backups valid and complete, or are they empty?
- Do you have sufficient machine resources to run all of the setup, restore, and post-processing tasks that comprise your recovery?
- Does the recovery process complete in reasonable wall time?
- Are you able to monitor the state of your recovery process as it progresses?
- Are you free of critical dependencies on resources outside of your control, such as access to an offsite media storage vault that isn't available 24/7?

Our testing has discovered the aforementioned failures, as well as failures of many other components of a successful data recovery. If we hadn't discovered these failures in regular tests—that is, if we came across

the failures only when we needed to recover user data in real emergencies—it's quite possible that some of Google's most successful products today may not have stood the test of time.

Failures are inevitable. If you wait to discover them when you're under the gun, facing a real data loss, you're playing with fire. If testing forces the failures to happen before actual catastrophe strikes, you can fix problems before any harm comes to fruition.

Case Studies

Life imitates art (or in this case, science), and as we predicted, real life has given us unfortunate and inevitable opportunities to put our data recovery systems and processes to the test, under real-world pressure. Two of the more notable and interesting of these opportunities are discussed here.

Gmail—February, 2011: Restore from GTape

The first recovery case study we'll examine was unique in a couple of ways: the number of failures that coincided to bring about the data loss, and the fact that it was the largest use of our last line of defense, the GTape offline backup system.

Sunday, February 27, 2011, late in the evening

The Gmail backup system pager is triggered, displaying a phone number to join a conference call. The event we had long feared—indeed, the reason for the backup system's existence—has come to pass: Gmail lost a significant amount of user data. Despite the system's many safeguards and internal checks and redundancies, the data disappeared from Gmail.

This was the first large-scale use of GTape, a global backup system for Gmail, to restore live customer data. Fortunately, it was not the first such restore, as similar situations had been previously simulated many times. Therefore, we were able to:

- Deliver an estimate of how long it would take to restore the majority of the affected user accounts
- Restore all of the accounts within several hours of our initial estimate
- Recover 99%+ of the data before the estimated completion time

Was the ability to formulate such an estimate luck? No—our success was the fruit of planning, adherence to best practices, hard work, and cooperation, and we were glad to see our investment in each of these elements pay off as well as it did. Google was able to restore the lost data in a timely manner by executing a plan designed according to the best practices of *Defense in Depth* and *Emergency Preparedness*.

When Google publicly revealed that we recovered this data from our previously undisclosed tape backup system [\[Slo11\]](#), public reaction was a mix of surprise and amusement. Tape? Doesn't Google have lots of disks and a fast network to replicate data this important? Of course Google has such resources, but the principle of Defense in Depth dictates providing multiple layers of protection to guard against the breakdown or compromise of any single protection mechanism. Backing up online systems such as Gmail provides defense in depth at two layers:

- A failure of the internal Gmail redundancy and backup subsystems
- A wide failure or zero-day vulnerability in a device driver or filesystem affecting the underlying storage medium (disk)

This particular failure resulted from the first scenario—while Gmail had internal means of recovering lost data, this loss went beyond what internal means could recover.

One of the most internally celebrated aspects of the Gmail data recovery was the degree of cooperation

and smooth coordination that comprised the recovery. Many teams, some completely unrelated to Gmail or data recovery, pitched in to help. The recovery couldn't have succeeded so smoothly without a central plan to choreograph such a widely distributed Herculean effort; this plan was the product of regular dress rehearsals and dry runs. Google's devotion to emergency preparedness leads us to view such failures as inevitable. Accepting this inevitability, we don't hope or bet to avoid such disasters, but anticipate that they will occur. Thus, we need a plan for dealing not only with the foreseeable failures, but for some amount of random undifferentiated breakage, as well.

In short, we always *knew* that adherence to best practices is important, and it was good to see that maxim proven true.

Google Music—March 2012: Runaway Deletion Detection

The second failure we'll examine entails challenges in logistics that are unique to the scale of the datastore being recovered: where do you store over 5,000 tapes, and how do you efficiently (or even feasibly) read that much data from offline media in a reasonable amount of time?

Tuesday, March 6th, 2012, mid-afternoon

Discovering the problem

A Google Music user reports that previously unproblematic tracks are being skipped. The team responsible for interfacing with Google Music's users notifies Google Music engineers. The problem is investigated as a possible media streaming issue.

On March 7th, the investigating engineer discovers that the unplayable track's metadata is missing a reference that should point to the actual audio data. He is surprised. The obvious fix is to locate the audio data and reinstate the reference to the data. However, Google engineering prides itself for a culture of fixing issues at the root, so the engineer digs deeper.

When he finds the cause of the data integrity lapse, he almost has a heart attack: the audio reference was removed by a privacy-protecting data deletion pipeline. This part of Google Music was designed to delete very large numbers of audio tracks in record time.

Assessing the damage

Google's privacy policy protects a user's personal data. As applied to Google Music specifically, our privacy policy means that music files and relevant metadata are removed within reasonable time after users delete them. As the popularity of Google Music soared, the amount of data grew rapidly, so the original deletion implementation needed to be redesigned in 2012 to be more efficient. On February 6th, the updated data deletion pipeline enjoyed its maiden run, to remove relevant metadata. Nothing seemed amiss at the time, so a second stage of the pipeline was allowed to remove the associated audio data too.

Could the engineer's worst nightmare be true? He immediately sounded the alarm, raising the priority of the support case to Google's most urgent classification and reporting the issue to engineering management and Site Reliability Engineering. A small team of Google Music developers and SREs assembled to tackle the issue, and the offending pipeline was temporarily disabled to stem the tide of external user casualties.

Next, manually checking the metadata for millions to billions of files organized across multiple datacenters would be unthinkable. So the team whipped up a hasty MapReduce job to assess the damage and waited desperately for the job to complete. They froze as its results came in on March 8th: the refactored data deletion pipeline had removed approximately 600,000 audio references that shouldn't have been removed, affecting audio files for 21,000 users. Since the hasty diagnosis pipeline made a few

simplifications, the true extent of the damage could be worse.

It had been over a month since the buggy data deletion pipeline first ran, and that maiden run itself removed hundreds of thousands of audio tracks that should not have been removed. Was there any hope of getting the data back? If the tracks weren't recovered, or weren't recovered fast enough, Google would have to face the music from its users. How could we not have noticed this glitch?

Resolving the issue

Parallel bug identification and recovery efforts

The first step in resolving the issue was to identify the actual bug, and determine how and why the bug happened. As long as the root cause wasn't identified and fixed, any recovery efforts would be in vain. We would be under pressure to re-enable the pipeline to respect the requests of users who deleted audio tracks, but doing so would hurt innocent users who would continue to lose store-bought music, or worse, their own painstakingly recorded audio files. The only way to escape the Catch-22¹³¹ was to fix the issue at its root, and fix it quickly.

Yet there was no time to waste before mounting the recovery effort. The audio tracks themselves were backed up to tape, but unlike our Gmail case study, the encrypted backup tapes for Google Music were trucked to offsite storage locations, because that option offered more space for voluminous backups of users' audio data. To restore the experience of affected users quickly, the team decided to troubleshoot the root cause while retrieving the offsite backup tapes (a rather time-intensive restore option) in parallel.

The engineers split into two groups. The most experienced SREs worked on the recovery effort, while the developers analyzed the data deletion code and attempted to fix the data loss bug at its root. Due to incomplete knowledge of the root problem, the recovery would have to be staged in multiple passes. The first batch of nearly half a million audio tracks was identified, and the team that maintained the tape backup system was notified of the emergency recovery effort at 4:34 p.m. Pacific Time on March 8th.

The recovery team had one factor working in their favor: this recovery effort occurred just weeks after the company's annual disaster recovery testing exercise (see [Kri12]). The tape backup team already knew the capabilities and limitations of their subsystems that had been the subjects of DiRT tests and began dusting off a new tool they'd tested during a DiRT exercise. Using the new tool, the combined recovery team began the painstaking effort of mapping hundreds of thousands of audio files to backups registered in the tape backup system, and then mapping the files from backups to actual tapes.

In this way, the team determined that the initial recovery effort would involve the recall of over 5,000 backup tapes by truck. Afterwards, datacenter technicians would have to clear out space for the tapes at tape libraries. A long, complex process of registering the tapes and extracting the data from the tapes would follow, involving workarounds and mitigations in the event of bad tapes, bad drives, and unexpected system interactions.

Unfortunately, only 436,223 of the approximately 600,000 lost audio tracks were found on tape backups, which meant that about 161,000 other audio tracks were eaten before they could be backed up. The recovery team decided to figure out how to recover the 161,000 missing tracks after they initiated the recovery process for the tracks with tape backups.

Meanwhile, the root cause team had pursued and abandoned a red herring: they initially thought that a storage service on which Google Music depended had provided buggy data that misled the data deletion pipelines to remove the wrong audio data. Upon closer investigation, that theory was proven false. The root cause team scratched their heads and continued their search for the elusive bug.

First wave of recovery

Once the recovery team had identified the backup tapes, the first recovery wave kicked off on March 8th. Requesting 1.5 petabytes of data distributed among thousands of tapes from offsite storage was one matter, but extracting the data from the tapes was quite another. The custom-built tape backup software stack wasn't designed to handle a single restore operation of such a large size, so the initial recovery was split into 5,475 restore jobs. It would take a human operator typing in one restore command a minute more than three days to request that many restores, and any human operator would no doubt make many mistakes. Just requesting the restore from the tape backup system needed SRE to develop a programmatic solution.¹³²

By midnight on March 9th, Music SRE finished requesting all 5,475 restores. The tape backup system began working its magic. Four hours later, it spat out a list of 5,337 backup tapes to be recalled from offsite locations. In another eight hours, the tapes arrived at a datacenter in a series of truck deliveries.

While the trucks were en route, datacenter technicians took several tape libraries down for maintenance and removed thousands of tapes to make way for the massive data recovery operation. Then the technicians began painstakingly loading the tapes by hand as thousands of tapes arrived in the wee hours of the morning. In past DiRT exercises, this manual process proved hundreds of times faster for massive restores than the robot-based methods provided by the tape library vendors. Within three hours, the libraries were back up scanning the tapes and performing thousands of restore jobs onto distributed compute storage.

Despite the team's DiRT experience, the massive 1.5 petabyte recovery took longer than the two days estimated. By the morning of March 10th, only 74% of the 436,223 audio files had been successfully transferred from 3,475 recalled backup tapes to distributed filesystem storage at a nearby compute cluster. The other 1,862 backup tapes had been omitted from the tape recall process by a vendor. In addition, the recovery process had been held up by 17 bad tapes. In anticipation of a failure due to bad tapes, a redundant encoding had been used to write the backup files. Additional truck deliveries were set off to recall the redundancy tapes, along with the other 1,862 tapes that had been omitted by the first offsite recall.

By the morning of March 11th, over 99.95% of the restore operation had completed, and the recall of additional redundancy tapes for the remaining files was in progress. Although the data was safely on distributed filesystems, additional data recovery steps were necessary in order to make them accessible to users. The Google Music Team began exercising these final steps of the data recovery process in parallel on a small sample of recovered audio files to make sure the process still worked as expected.

At that moment, Google Music production pagers sounded due to an unrelated but critical user-affecting production failure—a failure that fully engaged the Google Music team for two days. The data recovery effort resumed on March 13th, when all 436,223 audio tracks were once again made accessible to their users. In just short of 7 days, 1.5 petabytes of audio data had been reinstated to users with the help of offsite tape backups; 5 of the 7 days comprised the actual data recovery effort.

Second wave of recovery

With the first wave of the recovery process behind them, the team shifted its focus to the other 161,000 missing audio files that had been deleted by the bug before they were backed up. The majority of these files were store-bought and promotional tracks, and the original store copies were unaffected by the bug. Such tracks were quickly reinstated so that the affected users could enjoy their music again.

However, a small portion of the 161,000 audio files had been uploaded by the users themselves. The Google Music Team prompted their servers to request that the Google Music clients of affected users re-upload files dating from March 14th onward. This process lasted more than a week. Thus concluded the complete recovery effort for the incident.

Addressing the root cause

Eventually, the Google Music Team identified the flaw in their refactored data deletion pipeline. To understand this flaw, you first need context about how offline data processing systems evolve on a large scale.

For a large and complex service comprising several subsystems and storage services, even a task as simple as removing deleted data needs to be performed in stages, each involving different datastores.

For data processing to finish quickly, the processing is parallelized to run across tens of thousands of machines that exert a large load on various subsystems. This distribution can slow the service for users, or cause the service to crash under the heavy load.

To avoid these undesirable scenarios, cloud computing engineers often make a short-lived copy of data on secondary storage, where the data processing is then performed. Unless the relative age of the secondary copies of data is carefully coordinated, this practice introduces race conditions.

For instance, two stages of a pipeline may be designed to run in strict succession, three hours apart, so that the second stage can make a simplifying assumption about the correctness of its inputs. Without this simplifying assumption, the logic of the second stage may be hard to parallelize. But the stages may take longer to complete as the volume of data grows. Eventually, the original design assumptions may no longer hold for certain pieces of data needed by the second stage.

At first, this race condition may occur for a tiny fraction of data. But as the volume of data increases, a larger and larger fraction of the data is at risk for triggering a race condition. Such a scenario is probabilistic—the pipeline works correctly for the vast majority of data and for most of the time. When such race conditions occur in a data deletion pipeline, the wrong data can be deleted nondeterministically.

Google Music's data deletion pipeline was designed with coordination and large margins for error in place. But when upstream stages of the pipeline began to require increased time as the service grew, performance optimizations were put in place so Google Music could continue to meet privacy requirements. As a result, the probability of an inadvertent data-deleting race condition in this pipeline began to increase. When the pipeline was refactored, this probability again significantly increased, up to a point at which the race conditions occurred more regularly.

In the wake of the recovery effort, Google Music redesigned its data deletion pipeline to eliminate this type of race condition. In addition, we enhanced production monitoring and alerting systems to detect similar large-scale runaway deletion bugs with the aim of detecting and fixing such issues before users notice any problems. [133](#)

General Principles of SRE as Applied to Data Integrity

General principles of SRE can be applied to the specifics of data integrity and cloud computing as described in this section.

Beginner's Mind

Large-scale, complex services have inherent bugs that can't be fully grokked. Never think you understand enough of a complex system to say it won't fail in a certain way. Trust but verify, and apply defense in depth. (Note: "Beginner's mind" does *not* suggest putting a new hire in charge of that data deletion pipeline!)

Trust but Verify

Any API upon which you depend won't work perfectly *all* of the time. It's a given that regardless of your engineering quality or rigor of testing, the API will have defects. Check the correctness of the most critical elements of your data using out-of-band data validators, even if API semantics suggest that you need not do so. Perfect algorithms may not have perfect implementations.

Hope Is Not a Strategy

System components that aren't continually exercised fail when you need them most. Prove that data recovery works with regular exercise, or data recovery won't work. Humans lack discipline to continually exercise system components, so automation is your friend. However, when you staff such automation efforts with engineers who have competing priorities, you may end up with temporary stopgaps.

Defense in Depth

Even the most bulletproof system is susceptible to bugs and operator error. In order for data integrity issues to be fixable, services must detect such issues quickly. Every strategy eventually fails in changing environments. The best data integrity strategies are multitiered—multiple strategies that fall back to one another and address a broad swath of scenarios together at reasonable cost.

Revisit and Reexamine

The fact that your data “was safe yesterday” isn't going to help you tomorrow, or even today. Systems and infrastructure change, and you've got to prove that your assumptions and processes remain relevant in the face of progress. Consider the following.

The Shakespeare service has received quite a bit of positive press, and its user base is steadily increasing. No real attention was paid to data integrity as the service was being built. Of course, we don't want to serve *bad* bits, but if the index Bigtable is lost, we can easily re-create it from the original Shakespeare texts and a MapReduce. Doing so would take very little time, so we never made backups of the index.

Now a new feature allows users to make text annotations. Suddenly, our dataset can no longer be easily re-created, while the user data is increasingly valuable to our users. Therefore, we need to revisit our replication options—we're not just replicating for latency and bandwidth, but for data integrity, as well. Therefore, we need to create and test a backup and restore procedure. This procedure is also periodically tested by a DiRT exercise to ensure that we can restore users' annotations from backups within the time set by the SLO.

Conclusion

Data availability must be a foremost concern of any data-centric system. Rather than focusing on the means to the end, Google SRE finds it useful to borrow a page from test-driven development by proving that our systems can maintain data availability with a predicted maximum down time. The means and mechanisms that we use to achieve this end goal are necessary evils. By keeping our eyes on the goal, we avoid falling into the trap in which “The operation was a success, but the system died.”

Recognizing that not just *anything* can go wrong, but that *everything* will go wrong is a significant step toward preparation for any real emergency. A matrix of all possible combinations of disasters with plans to address each of these disasters permits you to sleep soundly for at least one night; keeping your recovery plans current and exercised permits you to sleep the other 364 nights of the year.

As you get better at recovering from any breakage in reasonable time N , find ways to whittle down that time through more rapid and finer-grained loss detection, with the goal of approaching $N = 0$. You can then switch from planning recovery to planning prevention, with the aim of achieving the holy grail of *all*

the data, all the time. Achieve this goal, and you can sleep on the beach on that well-deserved vacation.

¹²²Atomicity, Consistency, Isolation, Durability; see <https://en.wikipedia.org/wiki/ACID>. SQL databases such as MySQL and PostgreSQL strive to achieve these properties.

¹²³Basically Available, Soft state, Eventual consistency; see https://en.wikipedia.org/wiki/Eventual_consistency. BASE systems, like Bigtable and Megastore, are often also described as "NoSQL."

¹²⁴For further reading on ACID and BASE APIs, see [Gol14] and [Bai13].

¹²⁵Binary Large Object; see https://en.wikipedia.org/wiki/Binary_large_object.

¹²⁶See [https://en.wikipedia.org/wiki/Zero-day_\(computing\)](https://en.wikipedia.org/wiki/Zero-day_(computing)).

¹²⁷Clay tablets are the oldest known examples of writing. For a broader discussion of preserving data for the long haul, see [Con96].

¹²⁸Upon reading this advice, one might ask: since you have to offer an API on top of the datastore to implement soft deletion, why stop at soft deletion, when you could offer many other features that protect against accidental data deletion by users? To take a specific example from Google's experience, consider Blobstore: rather than allow customers to delete Blob data and metadata directly, the Blob APIs implement many safety features, including default backup policies (offline replicas), end-to-end checksums, and default tombstone lifetimes (soft deletion). It turns out that on multiple occasions, soft deletion saved Blobstore's clients from data loss that could have been much, much worse. There are certainly many deletion protection features worth calling out, but for companies with required data deletion deadlines, soft deletion was the most pertinent protection against bugs and accidental deletion in the case of Blobstore's clients.

¹²⁹"Snapshot" here refers to a read-only, static view of a storage instance, such as snapshots of SQL databases. Snapshots are often implemented using copy-on-write semantics for storage efficiency. They can be expensive for two reasons: first, they contend for the same storage capacity as the live datastores, and second, the faster your data mutates, the less efficiency is gained from copying-on-write.

¹³⁰For more information on GFS-style replication, see [Ghe03]. For more information on Reed-Solomon erasure codes, see https://en.wikipedia.org/wiki/Reed-Solomon_error_correction.

¹³¹See [http://en.wikipedia.org/wiki/Catch-22_\(logic\)](http://en.wikipedia.org/wiki/Catch-22_(logic)).

¹³²In practice, coming up with a programmatic solution was not a hurdle because the majority of SREs are experienced software engineers, as was the case here. The expectation of such experience makes SREs notoriously hard to find and hire, and from this case study and other data points, you can begin to appreciate why SRE hires practicing software engineers; see [Jon15].

¹³³In our experience, cloud computing engineers are often reluctant to set up production alerts on data deletion rates due to natural variation of per-user data deletion rates with time. However, since the intent of such an alert is to detect global rather than local deletion rate anomalies, it would be more useful to alert when the global data deletion rate, aggregated across all users, crosses an extreme threshold (such as 10x the observed 95th percentile), as opposed to less useful per-user deletion rate alerts.

[previous](#)

[Chapter 25 - Data Processing Pipelines](#)

[next](#)

[Chapter 27 - Reliable Product Launches at Scale](#)

Copyright © 2017 Google, Inc. Published by O'Reilly Media, Inc. Licensed under [CC BY-NC-ND 4.0](#)