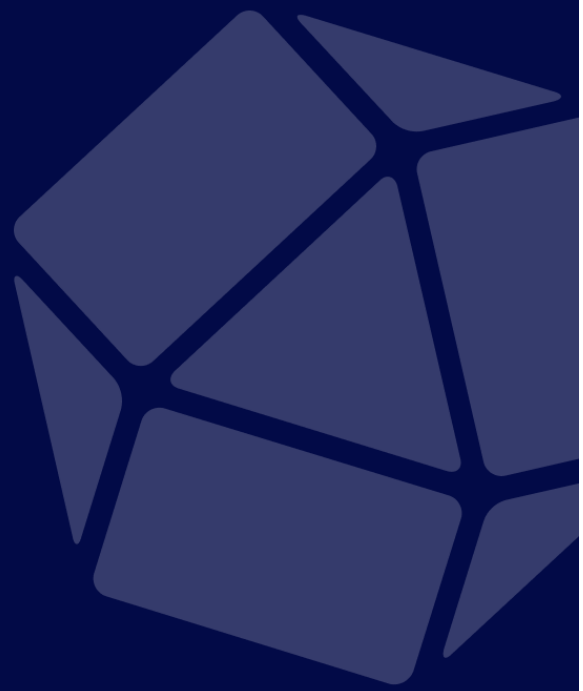
influxdata®

AN INFLUXDATA CASE STUDY

How Time Series Database Contributes to a Decentralized Cloud Object Storage Platform

John Gleeson
VP Operations, Storj
Ben Sirb



STORJ.IO

FEBRURY 2020 (REVISION 1)

Company in brief

Storj (pronounced storage), created by Storj Labs, is an open source decentralized cloud storage platform where anyone can sell their extra hard drive space for STORJ tokens. Unlike traditional cloud storage providers, Storj keeps data spread across a decentralized network, eliminating the problem of having a single point of failure. It also encrypts all data, making it impossible for anyone, including Storj, to gain access to users' files without possession of the corresponding private encryption key.

Case overview

Storj Labs uses blockchain technology and a distributed network to provide storage at lower costs than cloud providers. By equipping its customers with fast, secure and fully distributed storage, customers no longer need to manage their infrastructure. The Storj platform enables applications to store and share data across a distributed network with end-to-end encryption. Since most of Storj's metrics are time-stamped and since they wanted to track change over time, they chose InfluxDB as the time series component of Storj's Tardigrade service and workflows.

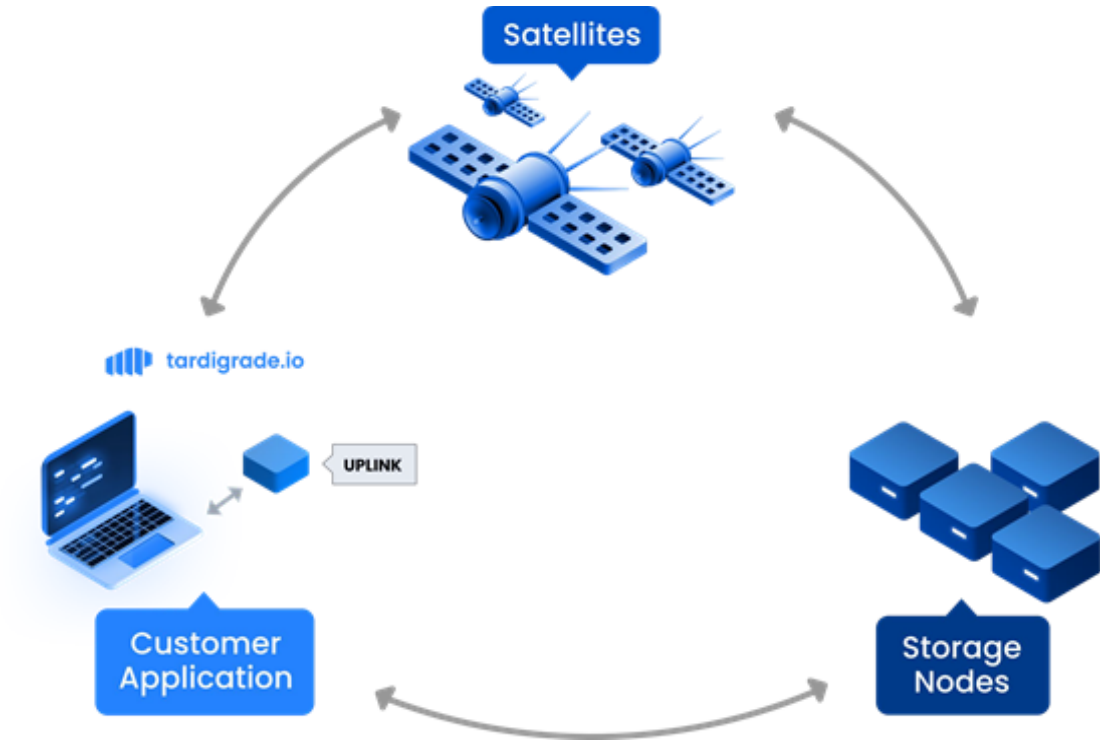
Storj uses InfluxDB to enable a file segment decay forecast and gain storage repair visibility. As an open source partner that is part of Storj Labs' Open Source Partner Program, InfluxDB helps Storj adjust the repair threshold to ensure file availability and monitor growth rate in order to maintain network supply-demand balance. This, in turn, helped the company achieve tradeoffs between file durability, retrievability, and repair while adhering to cost constraints, as well as providing storage performance comparable to leading cloud providers.

> *"If you can't measure it, you can't improve it. So we have a list of key metrics that directly impacted our roadmap progression. We wanted to make sure that the actual performance of the network, not based on theory but on data, was living up to the expectations we set for ourselves."*
>
> ***Ben Sirb**, Senior Data Scientist, Storj*

The business problem

With decentralized cloud storage, Storj has a large, distributed network comprised of thousands of independently owned and operated nodes across the globe that store data on Storj behalf. On the user's end, decentralized cloud storage operates exactly the same as traditional cloud storage options

like Amazon S3. But, instead of files being stored in a big data center that's vulnerable to outages and attacks, they are stored on thousands of distributed nodes all across the globe.



Storj: A decentralized cloud storage network

Overview of what Storj Labs does

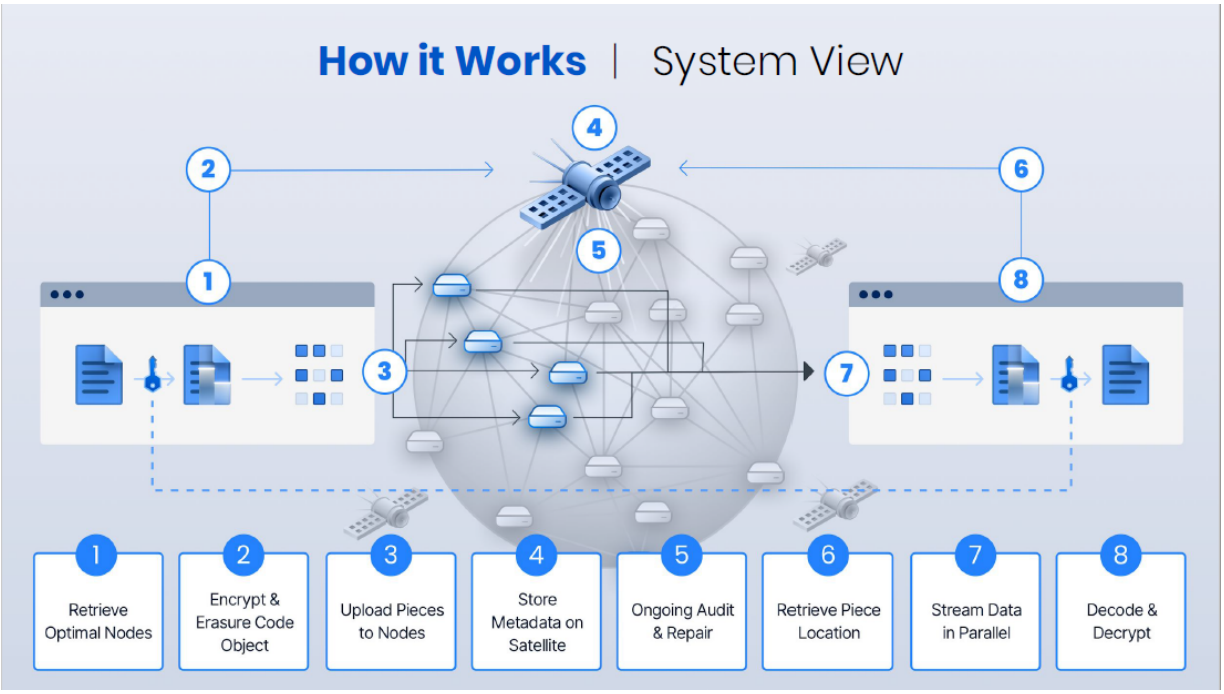
Storj's goal is to create the world's largest and most secure, resilient, performant, and economical cloud storage service, comparable to many of the other cloud storage providers from AWS to Google Cloud to Wasabi or Backblaze, and to do that without owning a data center.

Storj achieves that goal with the three pieces of software that they operate (all Storj software is open source):

- **Storage Nodes:** software that allows people who have excess hard drive capacity and excess bandwidth to share that with the network.
- **Satellites:** software that Storj runs, several of them under a Tardigrade brand, but anyone in their community can run a satellite. The satellite helps mediate between the people who have storage and bandwidth to share and the developers who want to store data on that excess storage capacity. The satellite handles developer accounts and is also responsible for data repair, billing and payments, and other tasks.
- **Uplink:** a customer application consisting of a developer toolset, including an S3 compatible gateway, command-line interface, and Go library with numerous language bindings for popular

development languages that allows developers to embed Storj software in their product and then store data on the platform.

The chart below shows a high-level view of how the Storj platform works.



When an Uplink client wants to upload a file to the network, the following occurs:

- First, that object is encrypted client-side — for security and privacy purposes — because the storage nodes are run by people in various countries.
- Next, the files are erasure-coded. Each file is broken up into 80 pieces which are then stored on 80 statistically uncorrelated storage nodes. If any one, two, or three storage nodes become available, Uplink only needs 29 of those file segments to recover and then decrypt the file.
- The satellite provides a list of storage nodes to Uplink as a selection for storing the nodes.
- Then, Uplink uploads directly to those storage nodes.
- In the background while that file is being stored, the satellite keeps track of all those storage nodes and monitors them to make sure that they are available, that they have high uptime, and also that they haven't failed or left the network by auditing the data stored on them.
- Over time, it is possible that storage nodes leave the network or that hard drives fail. The satellite is responsible for counting those storage nodes as they leave the network. If it reaches a certain threshold where file availability could be at risk, the satellite is able to download 29 pieces to itself, recreate the missing pieces, and upload them to healthy nodes, ensuring that the file remains available at all times.

For Storj, the above is an expensive process because it consumes bandwidth. A decentralized system like theirs is a highly bandwidth-constrained environment, and bandwidth is really a premium.

- There are over 50 million files being stored on the Storj network. Since each file is broken into a minimum of 80 pieces, this creates significant complexity in performing real-time system monitoring to track file segments, the storage nodes that they are stored on, and the files that they belong to. When an Uplink client wants those files, it should be able to retrieve them immediately because it has to be comparable to any other cloud provider.
- In terms of the number of storage nodes that come and go from the network, the satellite has to ensure that storage node churn does not impact file durability and availability.
- Storj also has a system of incentives that go along with that usage and tracking where they compensate storage nodes for storing and making their bandwidth available. Throughout all of this, there are a set of configuration settings they use to balance trade-offs between usability and performance on the client-side, and cost and complexity on the backend.

Creating a sustainable path to revenue for open source partners

As Storj is an open source project, it's key for Storj Labs to create synergies with other open-source projects since open source drives so much of cloud workloads today. Storj Labs offers an alternative to large cloud infrastructure providers. Those providers tend to use open-source as a loss leader for infrastructure sales (loss leadership is a strategy in which products are not priced for profit but rather to attract customers and stimulate other product sales).

If the dominant model for monetizing open source becomes "using open source as a loss leader for infrastructure," only a few large companies have the scale to operate centralized clouds and traditional infrastructure, and those same large companies also get a disproportionate amount of traffic, data, talent and economic returns, so how would Storj drive economic growth and innovation from new open source projects?

Storj's answer is decentralized infrastructure. This fundamentally different economic model for delivering infrastructure creates a new model for monetizing open source. For this purpose, Storj created an Open Source Partner Program, whose concept is to create a sustainable path to revenue for open source projects. (This program's positive impact on open source partners is described in more detail in the final section of this case study).

Meeting the demands of a two-sided market

Storj operates in a two-sided market — interfacing with both supply and demand:

- On the supply side, Storj is interested in bringing on reliable storage node operators because they're the ones bringing disk capacity and bandwidth capacity to the network.

- On the demand side, Storj analyzes the rate at which clients add data to the network and wants to ensure that the capacity they need is available.

As a function of files being stored on the network, Storj had to consider three key factors:

- **Durability: Function of storage node attrition over time**

  Durability is what they are using to describe the likelihood or probability that a client that adds data to the network can still access their data in the future. And we wanted to make sure that we would launch a product that can guarantee the same levels of durability as anyone can expect using a lead cloud storage provider.

- **Retrievability: Function of storage node operators' uptime at any point in time**

  Retrievability is a function of storage node uptime. If your data is stored on a storage node that is not currently online and you are unable to access your data, they call that an unreputable file. The data may still be there, and if the node comes back online, you might still be able to access it. Yet Storj also wanted to ensure that the data stored on the network is retrievable according to expected levels guaranteed by other leading cloud storage providers.

- **Repair: Expensive business function to ensure durability and retrievability**

  Repair is the process that the satellite undertakes to ensure that segments and files that are losing pieces due to churn and other factors are not actually lost and that the file remains durable and available for customers.

Balancing economics with developer experience

When Storj first set out to launch its product, they didn't have any data to base their conclusions on, so they had to start with some *a priori* assumptions. They did research to see what expressions they could use, what prior work had been done in this area, and arrived at some calculations that gave them confidence in the parameters that they were using to encode files on the network. The mathematics behind their functions is data-intense. More details on their formula are available in this whitepaper, but below is a snapshot of the functions involved:

- Some of the parameters, in the table to the left, appear in the expression on the right.
- The expression on the right is what Storj initially used to estimate the amount of repair expected to occur on the network.

Variable	Description
MTTF	Mean time to failure
α	1/*MTTF*
MRT	Mean reconstruction time
γ	1/*MRT*
D	Total bytes on the network
n	Total number of pieces per segment (RS encoding)
k	Pieces needed to rebuild a segment (RS encoding)
m	Repair threshold
LR	Loss rate
1-*LR*	Durability
E_D	Expansion factor
B_R	Ratio of data that is repair bandwidth
BW_R	Total repair bandwidth in the network

$$B\frac{\alpha(n-m+k)}{k\cdot\ln(n/m)}$$

- **B** is the amount of data stored on the network.
- **Alpha** is the expected level of churn.
- **M, N, and K** are the Reed-Solomon encoding parameters with N being the number of pieces that are created, K being the number of pieces that one needs to rebuild their file, and M being the repair threshold.

They used some calculations and simulations to generate a list of values that could give them the guaranteed durability that they want to promise as well as help produce the expected levels of repair.

Their goal at the onset was to balance the tradeoffs between usability, durability, and performance along with the economic constraints that they had to consider. Ultimately, their formula results in balancing economics with the developer experience. Once they had the theory, they wanted to make sure it measures up to actual real world usage.

The technical problem

To improve their network and estimate repair, they needed to extract insight from data:

- They have a list of key metrics that directly impacted their roadmap progression — Qualification Gates. They wanted to make sure that the network's actual performance, not based on theory but based on data, was living up to the expectations that they set for themselves.
- Exposing network behavior to the entire organization lets them act faster:
    - Everyone on the team needs access to quality data to do their job successfully.
    - Measuring this data helped teams across their organization move more quickly.
- The more insight they have into the network, the faster they can iterate, and the more they can improve.

They realized that they would succeed or fail based on the data that they were collecting. All the moving pieces described above generate a massive amount of time-stamped data that needs to be ingested.

The solution

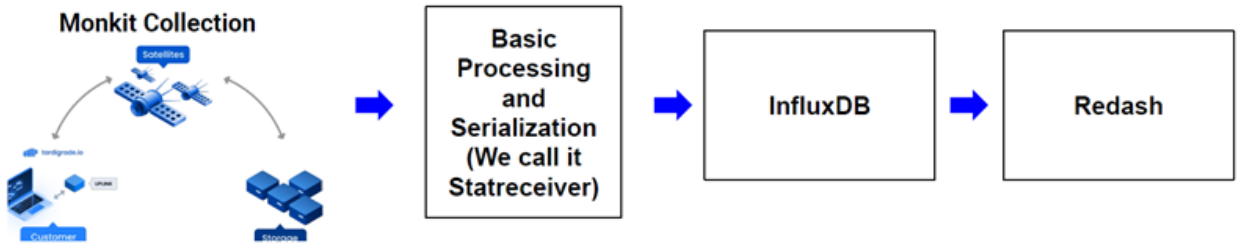
> *"We chose InfluxDB because the majority of our key metrics are very sensitive to change over time, and we needed to track these changes. It offers the scalability and features we need, giving us the greatest operational experience and confidence in the software with the least amount of work."*
>
> ***Ben Sirb**, Senior Data Scientist, Storj*

Why InfluxDB?

The decentralized nature of Storj's network drives their need for time series data, and therefore, for a time series database to collect and manage high-volume, real-time telemetry data from their distributed network. Storj chose InfluxDB because it is purpose-built for time series and therefore has the scalability and high ingest rate to support monitoring change over time.

Storj's data analytics approach

Using InfluxDB, Storj set up data ingestion with a simple ETL (extract, transform, load) process.



Monkit

The data collection process starts with a Golang package called Monkit, which Storj has sprinkled throughout their codebase in various aspects of the network: on Uplink, storage nodes, and satellites.

Monkit does basic monitoring and metric supporting as well as offering the capacity for custom measurement. Monkit outputs to what they call their Statreceiver binary (introduced below).

```
// IdentifyInjuredSegments checks for missing pieces off of the metainfo and overlay.
func (checker *Checker) IdentifyInjuredSegments(ctx context.Context) (err error) {
    defer mon.Task()(&ctx)(&err)

    observer := &checkerObserver{
        repairQueue:    checker.repairQueue,
        irrdb:          checker.irrdb,
        nodestate:      checker.nodestate,
        monStats:       durabilityStats{},
        overrideRepair: checker.repairOverride,
        log:            checker.logger,
    }
    err = checker.metaLoop.Join(ctx, observer)
    if err != nil {
        if !errs2.IsCanceled(err) {
            checker.logger.Error("IdentifyInjuredSegments error", zap.Error(err))
        }
        return err
    }

    mon.IntVal("remote_files_checked").Observe(observer.monStats.objectsChecked)                        //locked
    mon.IntVal("remote_segments_checked").Observe(observer.monStats.remoteSegmentsChecked)              //locked
    mon.IntVal("remote_segments_needing_repair").Observe(observer.monStats.remoteSegmentsNeedingRepair) //locked
    mon.IntVal("remote_segments_lost").Observe(observer.monStats.remoteSegmentsLost)                    //locked
    mon.IntVal("remote_files_lost").Observe(int64(len(observer.monStats.remoteSegmentInfo)))            //locked

    return err: nil
}
```

Monkit: metrics, monitoring, and more

Monkit captures event data (meters, rates, counts, etc.) right in the code (as the arrow above shows). Extra monitoring and telemetry data can be added right within the function call.

In addition to basic default monitoring, towards the top of the code is a `mon.Task` that reports percentiles, averages, success times, and counts by default. Monkit also allows the capacity to provide custom measurements:

- If a particular function in the code is reporting values that are particularly interesting to the operation of the network, you can call Monkit and have it report through the Storj data pipeline, resulting in a measurement that shows up in InfluxDB.
- This measurement is collected directly on the network (satellites, nodes, and Uplinks) and sent to a separate service for processing.

Statreceiver: Storj Labs' messaging service

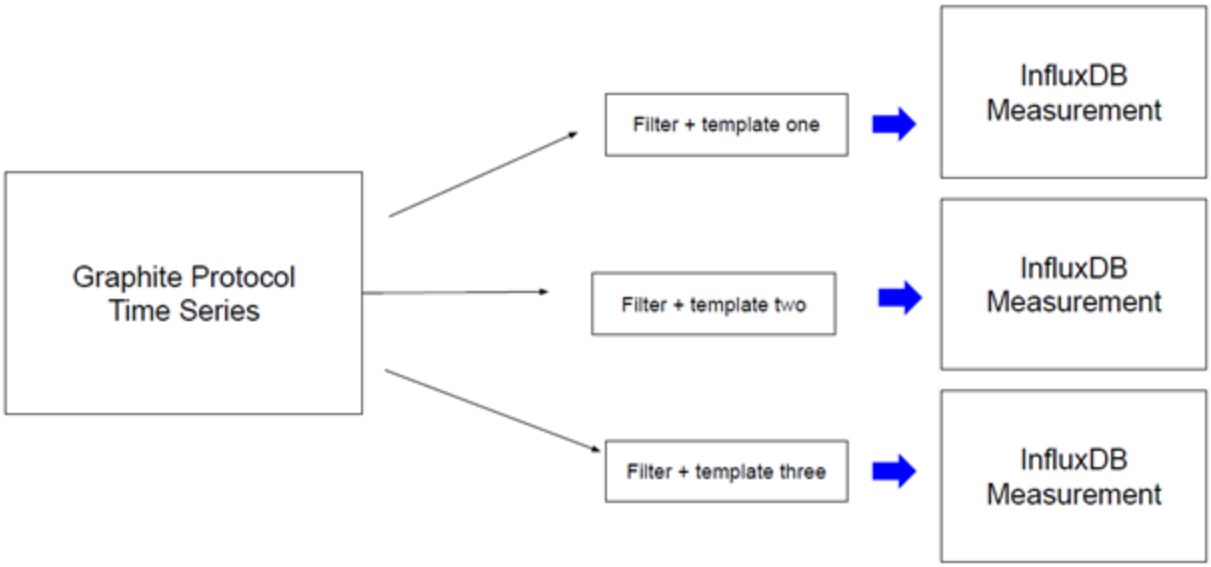
Data is ingested through Statreceiver, a custom binary. Using a simple configuration file, they apply basic templating and filtering to qualify the binary for use in their various data links. Statreceiver ingests the Monkit data and distributes it to various destinations within the organization. They have debugging endpoints, log outputs and various databases. One of these databases is InfluxDB. Statreceiver ingests UDP packets and sends time series data to InfluxDB via the Graphite protocol.

pipeline-example.lua

```
-- possible sources:
--  * udpin(address)
--  * filein(path)
-- multiple sources can be handled in the same run (including multiple sources
-- of the same type) by calling deliver more than once.
source = udpin(":9000")

-- multiple metric destination types
--  * graphite(address) goes to tcp with the graphite wire protocol
--  * print() goes to stdout
--  * db("sqlite3", path) goes to sqlite
--  * db("postgres", connstring) goes to postgres
graphite_out = graphite("localhost:5555")
db_out = mcopy(
  --print(),
  db("sqlite3", "db.db"))
  --db("postgres", "user=patrick password=abc123 dbname=patrick"))

metric_handlers = mcopy(
  -- send all satellite data to graphite
  appfilter("satellite-(dev|prod)",
    graphite_out),
  -- send specific storagenode data to the db
  appfilter("storagenode-(dev|prod)",
    keyfilter(
      "env\\.process\\." ..
        "|hw\\.disk\\..*Used" ..
        "|hw\\.disk\\..*Avail" ..
        "|hw\\.network\\.stats\\..*\\.(tx|rx)_bytes\\.(deriv|val)",
      db_out)))

-- create a metric parser.
metric_parser =
  parse(  -- parse takes one or two arguments. the first argument is
          -- a metric handler, the remaining one is a per-packet application or
          -- instance filter. each filter is a regex. all packets must
          -- match all packet filters.
    sanitize(metric_handlers), -- sanitize converts weird chars to underscores
    packetfilter("(storagenode|satellite|uplink)-(dev|prod)", ""))

-- pcopy forks data to multiple outputs
-- output types include parse, fileout, and udpout
destination = pcopy(
  --fileout("dump.out"),
  metric_parser,

  -- useful local debugging
  udpout("localhost:9001"),

  -- rothko
  udpout("localhost:9002"))

-- tie the source to the destination
deliver(source, destination)
```

InfluxDB, as it supports Graphite Protocol, allowed Storj to build useful additional structure:

- Monkit, by default, outputs completely unstructured data. The structure that Statreceiver assigns is limited because that receiver is sending the measurements out to various destinations.
- To assign more structure to the data, Storj used the Graphite filtering and templating schema. They were able to take certain measurements that they were interested in, apply a specific filter, and give it a specific measurement in InfluxDB, as shown below.



As a result, various internal teams were able to investigate the network's performance. Using the Monkit metrics monitoring output, they could investigate the performance of various function calls and examine key metrics to inform business decisions.

The chart above shows an example of a very important measurement that directly indicates the amount of repair they can expect. This measurement enables plotting different percentiles of pieces remaining for the number of segments on the network and thereby enables developing a storage repair forecast.

```
1
2  SELECT
3  percentile(checker_checker_segment_healthy_count_recent,10) as p_10
4  ,percentile(checker_checker_segment_healthy_count_recent,25) as p_25
5  ,percentile(checker_checker_segment_healthy_count_recent,50) as p_50
6  ,percentile(checker_checker_segment_healthy_count_recent,75) as p_75
7  ,percentile(checker_checker_segment_healthy_count_recent,90) as p_90
8  ,max(checker_checker_segment_healthy_count_recent) as max
9  ,min(checker_checker_segment_healthy_count_rmin) as min
10 ,min(checker_checker_segment_healthy_count_min) as process_lifetime_min
11 FROM satellite_repair
12 WHERE time >= '{{FROM}}'
13 AND instance = '{{Satellite}}'
14 GROUP BY instance, time({{Group by interval}}) fill(null)
15 order by time desc
```

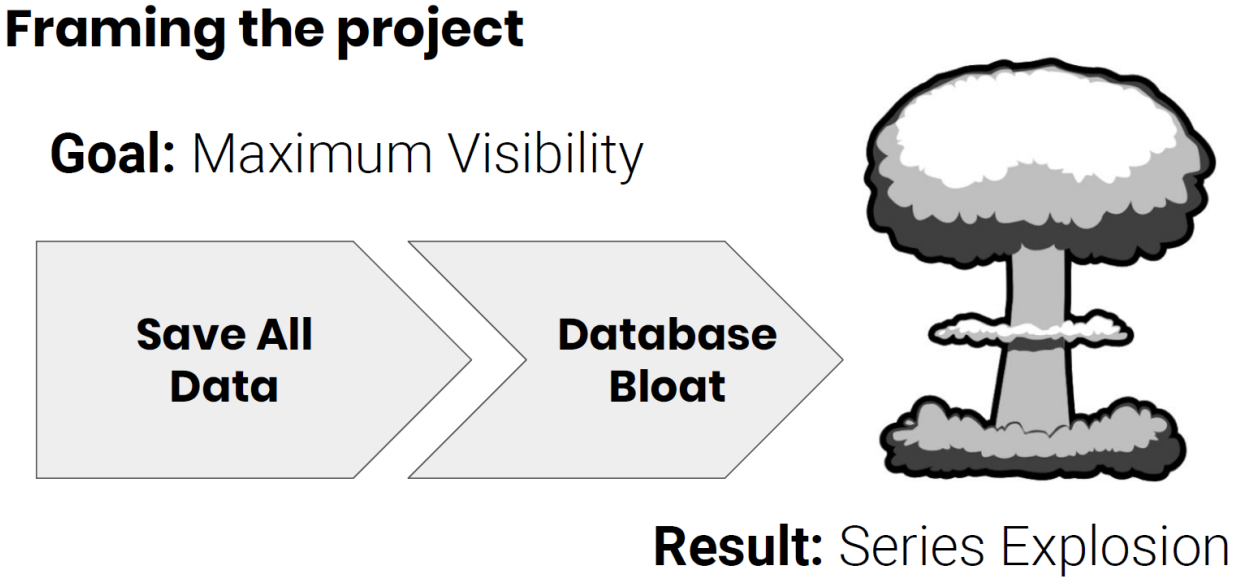
InfluxDB - Redash BI

Finally, Storj connected InfluxDB to their Redash BI investigative tool. Redash is their endpoint for the various databases that teams across their organization use to analyze network performance or answer business-related questions. As they generate massive amounts of data, they focused on optimizing the design, given the following considerations:

- The faster they start generating key data, the faster they improve.
- They don't have resources for a lot of maintenance overhead.
- The setup needs to be flexible as the network grows and as data needs change.
- They assume that non-Storj folks will use and run the solution at some point.
- The solution needs to scale with the network.

Overcoming the challenge of managing series cardinality

One challenge they had to overcome with regard to generating the above-mentioned InfluxDB measurements was managing series cardinality. As they had launched with the goal of maximum visibility into the network, this meant saving all the data and therefore led to massive database bloat and series explosion. The number of series grew at an unmanageably fast rate.



To solve this problem, they started filtering out certain measurements to prevent massive database size. They were still able to use other internal endpoints to debug and monitor performance but were no longer able to analyze the data coming in from storage nodes through InfluxDB. As they made improvements to their ETL process, the lesson learned was to not encode highly variable information as tags but rather as fields.

Results

> *"Now we succeed based on the data, and what InfluxDB lets us do is enable a very powerful segment decay forecast which is essential to knowing how much repair we can expect."*
>
> ***Ben Sirb**, Senior Data Scientist, Storj*

Storj's success today is data-driven and made possible through the powerful segment decay forecast that they generate through InfluxDB to estimate expected file repair threshold. Using the segment health data (see the query in the "Monkit" section above), they developed a predictive model by taking the historical data and bootstrapping a linear regression against the data to forecast when they can expect the first large chunk of repair to start.

This forecast was very impactful in the organization, as it led to them modifying their repair threshold, giving them a good idea of expected costs and the capability to test their repair mechanism by tweaking their repair threshold to start repair at an expected time. The forecast also enables them to closely monitor growth rate, which is essential to balance supply and demand within their network:

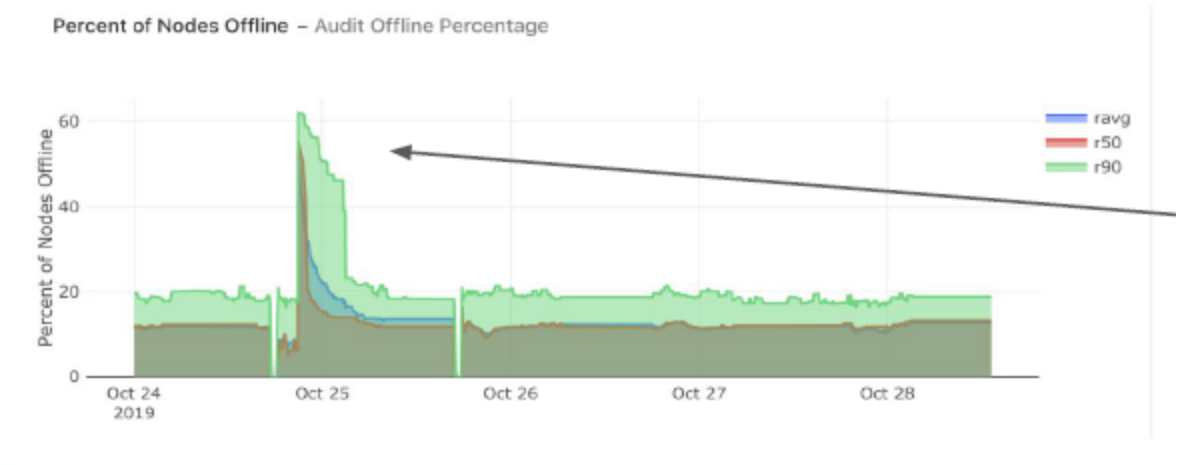
- If there's insufficient capacity for clients to upload data to the network, then it's hard to bring on new clients.
- On the other hand, if there's too much capacity and storage node operators aren't hosting data and aren't being paid for data they're not storing, that's no good either.

The forecast enabled by InfluxDB has helped them control complexity by estimating the number of objects on the network and allowed them to track historical bandwidth usage for various bandwidth operations (gets, puts, repair audits), which is critical in a bandwidth-constrained environment.

Finally, the forecast also allows them to track vetting success rate over time, an indication of health of network growth rate. Vetting is what they've implemented to help reduce the rate that storage nodes leave the network. Before they trust a storage node operator with data, they require the storage node to undergo a certain number of data audits to verify the integrity of data stored on their disk. Once they've successfully passed the number of data audits, then they consider those nodes vetted and fully trust them with more data on the network. They have implemented this vetting process as a way to reduce churn. The process allows them to make data-driven decisions and track historical rates of churn and segment decay.

They have created a number of internal dashboards for the benefit of the entire organization. This allows individuals and teams to dive in and do root cause analysis of any observed anomalies. As shown below, there is an uptick in the percent of nodes. They can compare this information with the amount of repair that they expected or the amount of pieces remaining for eroded segments. This enables them to review the current trajectory for key metrics.

The graphs below show how data-driven guidance now indicates the state of the Storj network.



A temporary churn uptick on October 24th due to a DNS issue on a US satellite

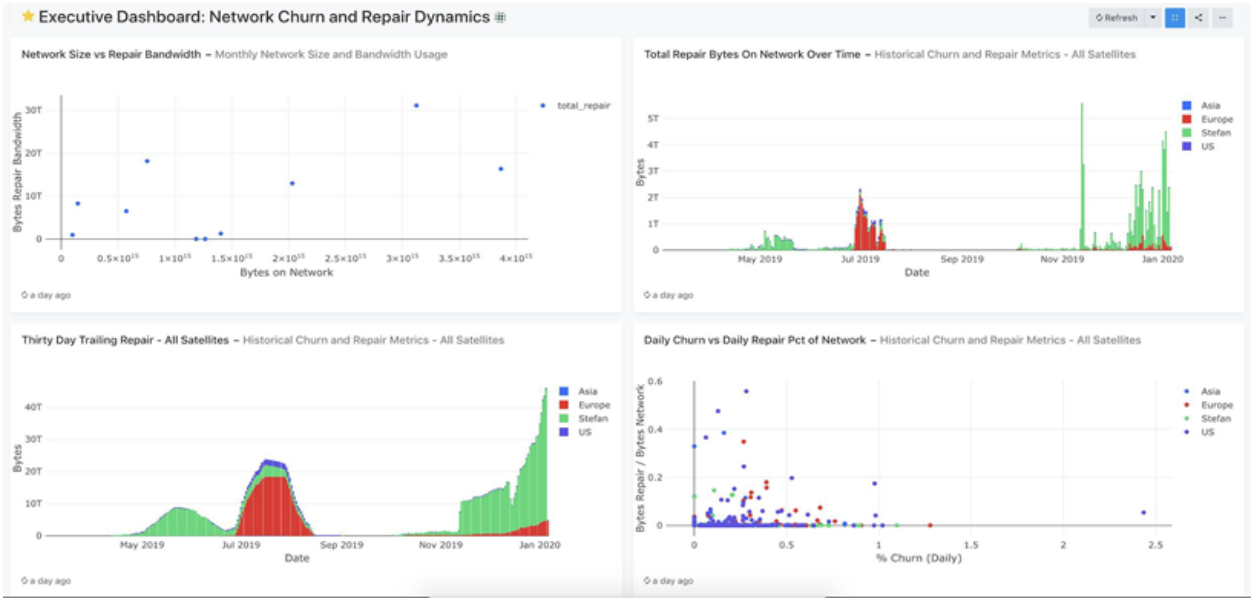
The above insights and dashboards are made possible by the time series nature of the data.

The chart below depicts a plot of the query, showing the percentiles of the number of pieces remaining for the segments on the network. This is something they rely on very heavily to decide whether files are in danger of being irrecoverable.

As they use a helpful expansion factor that allows for the rebuild to occur with only 29 pieces, none of the segments on the network ever get below 50, as shown below. Storj has a very generalized repair threshold to ensure that segments are never in danger of becoming irrecoverable.

Graph showing that repair is working as expected

Storj can also measure model parameters in real time, as shown below. They can compare key aspects of the network against one another. Storage node churn directly correlates to the amount of expected repair because the more that storage nodes can leave the network, the faster segments lose pieces, and this in turn results in a higher rate of repair flow for the segment.



Measuring model parameters in real time

Storj created a dashboard that highlights the churn-repair relationship. The dashboard visualizes how many bytes are on the network, how much repair is used, the amount of data used for repair, as well as comparing directly the rate of repair to that of churn.

While the initial implementation of their ETL resulted in measurements that they had to filter using Graphite, with the release of Monkit v3, they eliminated the need for the Graphite layer. This is because Monkit v3 can natively output measurements with appropriate tags and names. There was no longer any need to “lock” measurements (prior to Monkit v3, before “locking”, code changes resulted in broken measurements).

Monkit v3 “speaks InfluxDB” more natively. It outputs the default metrics that are now tagged more appropriately and has better default naming for the InfluxDB measurement, output, and browsing, and also allows very convenient measurement naming and customer measurement addition.

Today, Storj has visibility through a large set of dashboards enabling continuous network monitoring to ensure they are offering not only a commercially successful and economically viable product, but also a great end user experience for both demand and supply sides of their network.

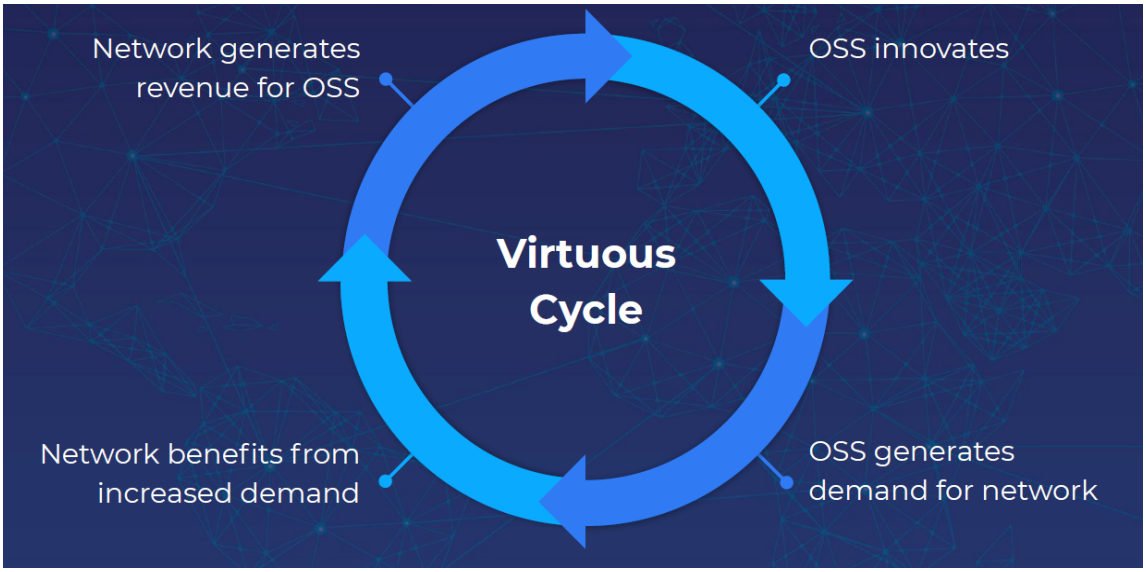
How InfluxData fits into Storj Labs’ Open Source Partner Program

Storj Labs’ primary incentive structure is to reward storage node operators (who contribute storage and bandwidth to the network) by fairly and transparently compensating them for the resources they provide so that they remain on the network long-term.

Storj Labs also has a program for partners who drive demand for storage to the network. Open source projects that generate a lot of data like database backups, for example, through InfluxDB for time series data are great candidates. Since open source is the biggest driver of cloud usage, Storj wanted to let open source projects get paid by the network for helping it grow.

Storj's network delivers significant benefits for open source projects and their users:

- **Customers** get secure, private, reliable cloud storage at half the price of the big cloud providers and drive revenue to open source projects they love.
- **Open source projects** get a meaningful share of subscription revenue when their end users store data on the Tardigrade platform.



The Storj business model generates a virtuous cycle:

- Open source innovation drives storage on the decentralized infrastructure, which in turn generates revenue for open source partners (Storj Labs shares a portion of its revenue with the open source project directly related to the use of their software).
- Then, that revenue allows open source partners to continue the cycle of innovation, which then drives more usage of the Storj platform.

What's next for Storj?

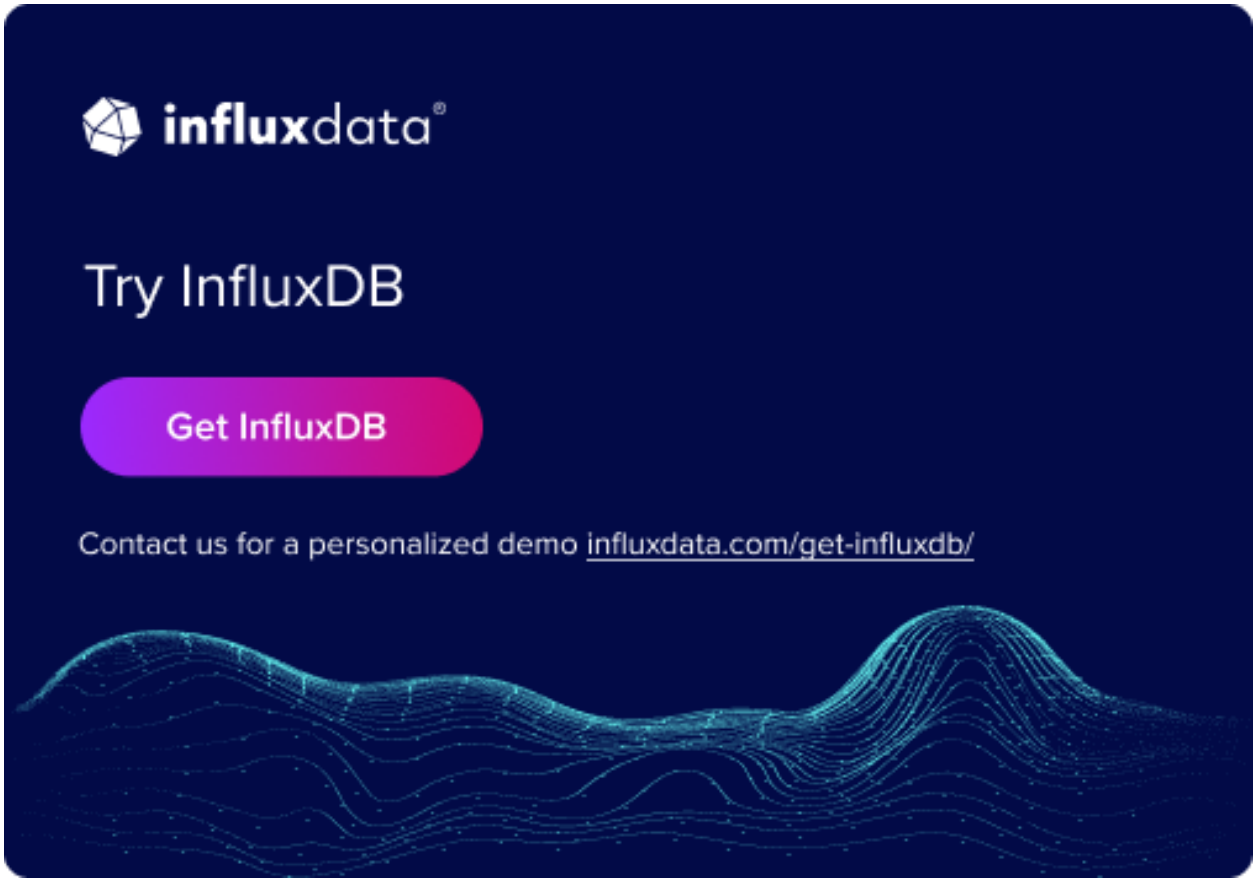
For its future roadmap, Storj is considering:

- **Using the complete InfluxDB platform:** Storj Labs has implemented an additional portion of the TICK Stack, Chronograf (the visualization engine native to InfluxDB 1.x), to browse the measurement that they are outputting and allow for various teams to investigate the data and start generating dashboards for internal dissemination.
- **Integrating statistical modeling directly into their architecture:** Storj has also recently rolled out statistical modeling directly into their architecture. They have implemented Apache Airflow (an automated batch processing paradigm) as well as a custom version of what they call Data Flow. Data Flow, a Golang version that allows for batch processing in near-real-time, is used to empower their teams to make better use of the data.
- **Decentralizing the network eventually needs decentralized analytics:** They want to avoid siloing the knowledge and power of their data in any one area because they believe that a decentralized network eventually needs decentralized analytics. Especially since they want to work very closely with their open-source partners, they need to ensure their tools are readily available for partners' use as well.

Storj also plans to look into using Flux, InfluxData's query and scripting language, as the InfluxDB version they were originally using hadn't yet incorporated Flux for cross-measurement analysis.

About InfluxData

InfluxData is the creator of InfluxDB, the leading time series platform. We empower developers and organizations, such as Cisco, IBM, Lego, Siemens, and Tesla, to build transformative IoT, analytics and monitoring applications. Our technology is purpose-built to handle the massive volumes of time-stamped data produced by sensors, applications and computer infrastructure. Easy to start and scale, InfluxDB gives developers time to focus on the features and functionalities that give their apps a competitive edge. InfluxData is headquartered in San Francisco, with a workforce distributed throughout the U.S. and across Europe. For more information, visit influxdata.com and follow us @InfluxDB.


influxdata
Try InfluxDB
Get InfluxDB
Contact us for a personalized demo influxdata.com/get-influxdb/