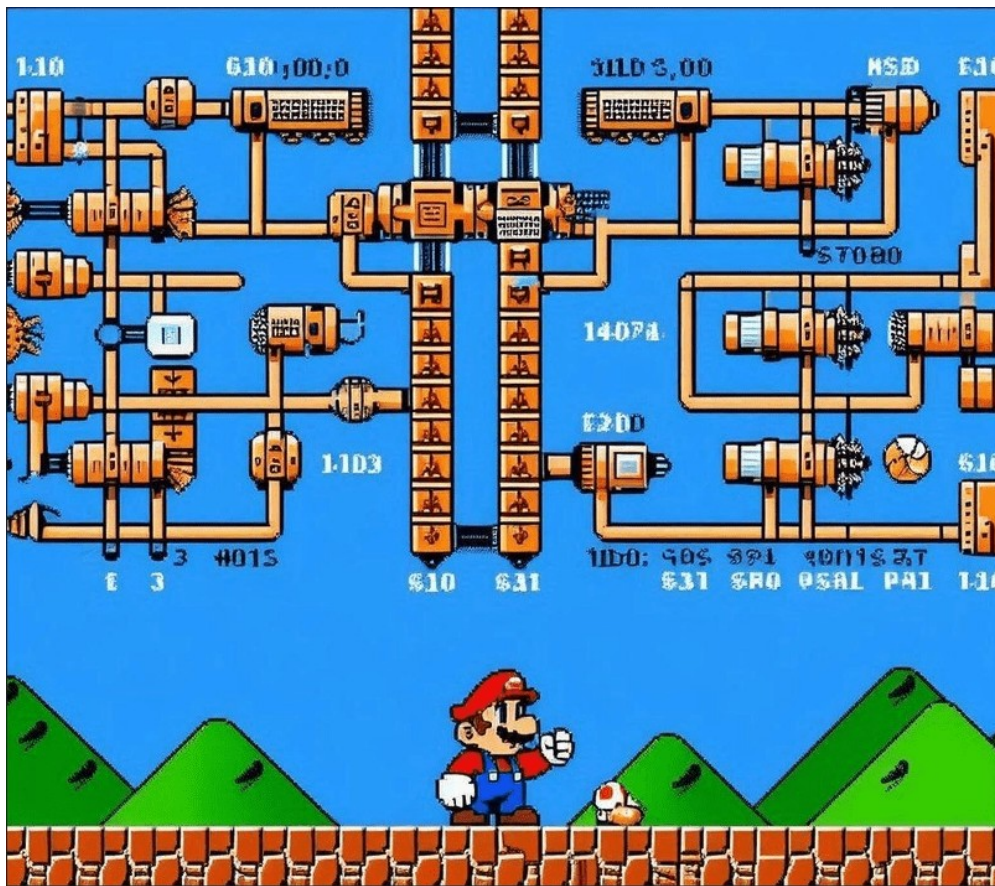


Thanos TSDB: How Default Configurations Can Lead to Silent Data Loss

Julien Laurenceau | Published 2025-12-06 | Updated 2026-10-01

<https://pepitedata.com/thanos-tsdb-default-config-silent-data-loss/>

PepiteData | LinkedIn: Julien Laurenceau | X: @Julien_Lau



Thanos is a widely adopted open-source project that extends Prometheus' capabilities, offering long-term storage, global querying, and downsampling. It's a powerful tool for monitoring and observability, but like any complex system, it has its quirks. Thanos is cloud native and use s3 as its main storage backend. It can have infinite retention contrary to prometheus limitations (15d).

One of the most critical risks in Thanos lies in its compactor component, which, under certain conditions, can silently lead to irreversible data loss. This issue is more than theoretical: it is rooted in real-world scenarios, as highlighted in [GitHub Issue #813](#) and [GitHub Issue #7908](#). If you're using Thanos, understanding these risks is essential to protecting your historical data.

TL;DR : retention-1h = retention-5m = retention-raw >= 10d Worst case scenario is overall retention is retention-raw.

The Core Problem: Downsampling and Retention Cleaning

Thanos runs three jobs in one compactor: compaction, downsampling and retention cleaning. The danger is the ordering between them. Retention deletes raw blocks on schedule, while downsampling only succeeds intermittently, so any gap between a deletion and its downsampled replacement is permanent data loss.

<https://thanos.io/tip/components/compact.md>

The Thanos compactor is responsible for managing data lifecycle tasks, including:

- Compaction: Merging smaller blocks into larger ones.
- Downsampling: Reducing data resolution for older blocks.
- Retention and Cleaning: Applying retention policies and deleting expired data.

While these tasks are critical for efficient storage and querying, the way they are implemented can lead to silent data loss in two key scenarios.

1. Downsampling Failures Halt the Compactor

If the compactor encounters an error during the downsampling phase, it halts the current iteration entirely. This means that retention cleaning is never executed for that iteration. Over time, this can result in raw data blocks accumulating beyond their intended retention period, while downsampled data remains incomplete or missing. In the same way if the compaction step fails, downsampling and cleaning never starts.

This behavior is particularly problematic because the compactor does not retry or recover from these errors automatically. If you're not actively monitoring the compactor's logs, you might not even realize that retention cleaning has been skipped.

This can be partially prevented by using the compactor option : `--no-debug.halt-on-error`

1. Downsampling "Succeeds" with Warnings but Outputs No Data

The second, more subtle issue arises when downsampling completes with warnings rather than outright errors. For example, the compactor may log a warning like "empty chunks" but still consider the downsampling task "successful." In such cases, the compactor proceeds to the retention cleaning phase, assuming that downsampled blocks have been created.

Here's the catch: if no downsampled data was actually generated (despite the task "succeeding"), the compactor will still apply retention policies to all blocks. This can result in the deletion of raw data before a downsampled version has been created, leading to irreversible gaps in your historical data.

Example : `caller=streamed_block_writer.go:116 level=warn msg="empty chunks happened, skip series"`

The Key Takeaway: Retention Durations Must Be Aligned

Thanos' documentation recommends setting the same retention duration for raw data, 5-minute downsampling, and 1-hour downsampling but does not highlight the importance of this setting. In my view, this is a requirement to ensure data integrity, not merely a recommendation.

If your retention durations are misaligned (e.g., raw data is retained for 1 year, but 5m downsampling is retained for 6 months), the compactor may delete raw data blocks before a downsampled version has been successfully created. This is especially dangerous in scenarios where downsampling "succeeds" with warnings but produces no output.

How to Protect Your Data

The defensive rules are short and mostly about removing silent defaults. Set every retention duration to the same value so raw blocks cannot expire before their downsampled replacements exist, alert on compactor warnings instead of errors alone, and rehearse compaction and downsampling failures in staging before production.

- **Align Retention Durations:** Always configure `retention_raw = retention_5m = retention_1h`. Treat this as a hard requirement, not a recommendation. Misaligned retention policies are a direct path to data loss.
- **Monitor the Compactor Closely:** Set up alerts for compactor errors and warnings. Pay special attention to logs that mention "empty chunks" or other anomalies during downsampling and use flag `--no-debug.halt-on-error`.
- **Understand the Risks:** Familiarize yourself with the behavior of the compactor and its assumptions. The issues outlined in [GitHub #813](#) and [GitHub #7908](#) provide valuable insights into these risks.
- **Test Your Configuration:** Simulate failures in a staging environment to understand how your setup behaves under edge cases. This can help you identify potential gaps before they impact production.

Final Thoughts

Thanos is a powerful tool that can transform your monitoring stack, but its default configurations and design assumptions can lead to silent data loss if not properly understood. The compactor's behavior, particularly its handling of downsampling errors and warnings, requires careful attention.

By ensuring that your retention durations are aligned and monitoring the compactor closely, you can mitigate these risks and safeguard your historical data. However, these issues also highlight the need for greater clarity in Thanos' documentation and potential improvements in its design to prevent such pitfalls.

It's a young project (v0.39.0) evolving quickly in particular regarding the feature linked to HA and multi-site. Anyhow, this fully open source project has 191 pull requests pending.

Originally published on [dev.to](#)

[book a call](#) !

Update 2026-09-29. This post is now part of [Time-Series and Specialized Data Stores: Choosing the Right Engine](#), the field guide to choosing the right data store.

Related posts

- [Time-Series and Specialized Data Stores: Choosing the Right Engine](#)
- [Why file count matters as much as file size on object storage](#)
- [TigerBeetle DB and financial database performance](#)
- [Which message broker for IoT and real-time control](#)

PDF snapshot of the public article. The online version is authoritative; links remain clickable.