RHEBA RESEARCH

The Case Against Unix Time

Why modern scheduling software still models human time wrong.

5 MIN READ AARON SHOEMAKER, RHEBA, INC.

> *Unix time was a machine answer to a machine problem. Human coordination needs something else — a representation that preserves intent instead of constantly translating away from it.*

Most calendar and scheduling software still rests on an assumption that made sense for machines and became a liability for people.

That assumption is Unix time.

Unix time stores time as a count of seconds since January 1, 1970. For computers trying to synchronize events across systems, that was a clean solution. For human coordination, it has always been a compromise.

Humans do not think in epoch seconds.

They think in intent.

When someone says “2 PM,” they mean 2 PM. They do not mean a UTC-relative numeric value that should be constantly converted, interpreted, adjusted, and recomputed depending on travel, device settings, daylight saving transitions, or the assumptions of whatever software happens to be in the loop.

The farther software gets from that human intent, the more friction it creates.

That friction is so common that most people have stopped treating it like a design failure. They treat it like weather.

> *Of course recurring events drift. Of course traveling across time zones makes calendars feel unstable. Of course coordinating people in different locations becomes a mess of conversions, screenshots, and follow-up messages.*

But those are not isolated bugs. They are symptoms of a deeper architectural problem.

Unix time was designed to help machines agree on sequence. It was not designed to help humans agree on meaning.

That distinction matters more every year.

Today, coordination is not just “what time is my meeting?” It is:

- who is available
- where they can be
- whether travel makes that timing realistic
- how schedules align across groups, roles, and obligations
- how recurring patterns should behave over time

Once you move from simple timestamp storage into real coordination, the cracks in the model become structural.

Most systems try to solve that by layering more logic on top of the same underlying assumption. More conversions. More timezone handling. More edge-case cleanup. More UX smoothing around an abstraction that was never built for the job.

That is why so much scheduling software feels brittle.

The problem is not only the interface. The problem is the underlying representation.

At Rheba, we took a different approach.

We started from a simpler question:

> **What if time in coordination systems should be stored in a way that preserves human intent instead of constantly translating away from it?**

That leads to a very different design philosophy.

Instead of treating time as a number line first and a human experience second, you begin by preserving what the person actually means. Then you build coordination on top of that. That shift changes more than display logic. It changes what becomes possible in the system.

It changes how recurring patterns behave.

It changes how shared availability can be computed.

It changes how much software has to fight its own abstraction every time real life gets involved.

What replaces Unix time is not one more conversion layer.

It is a different storage model.

In the Rheba time system, the core primitives are separated on purpose:

- RDate stores a calendar date without collapsing it into an epoch value
- RTime stores a local time of day as a first-class value
- RNQH combines the two into a coordination-ready date-and-time value

That means the system stores the meaning of “April 25 at 2 PM” directly instead of translating it into a UTC-relative second count and hoping that later conversions preserve what the person intended.

> *Store meaning first. Convert second.*

Time zones still matter. Synchronization still matters. Interop still matters. But they belong at the edges of the system, not as the primary shape of the data.

In other words, conversion becomes an explicit operation instead of the hidden default underneath everything.

That changes the behavior of recurrence, travel-aware planning, shared availability, and calendar arithmetic because the system is no longer constantly reconstructing human meaning from a machine representation that discarded it.

Unix time is still useful when machines need to serialize, transmit, or interoperate with other systems. It is just the wrong primitive to put at the center of human coordination.

And it matters even more once availability becomes a network problem instead of a single-user calendar problem.

Traditional scheduling tools are mostly designed to help one person manage one calendar or expose a small slice of availability to another person. That is not the same thing as building infrastructure that can answer a broader coordination question:

Who is available, where, and when?

At that point, time storage is no longer a background implementation detail. It becomes part of the category definition.

That is why we argue that modern scheduling software is still built on the wrong abstraction.

Unix time was a machine answer to a machine problem. Human coordination needs something else.

If the job is helping people align real life across time, place, and commitments, then the system should start by honoring human time as people actually mean it.

That is not a UX tweak. It is an architectural correction.