CYBERTEC

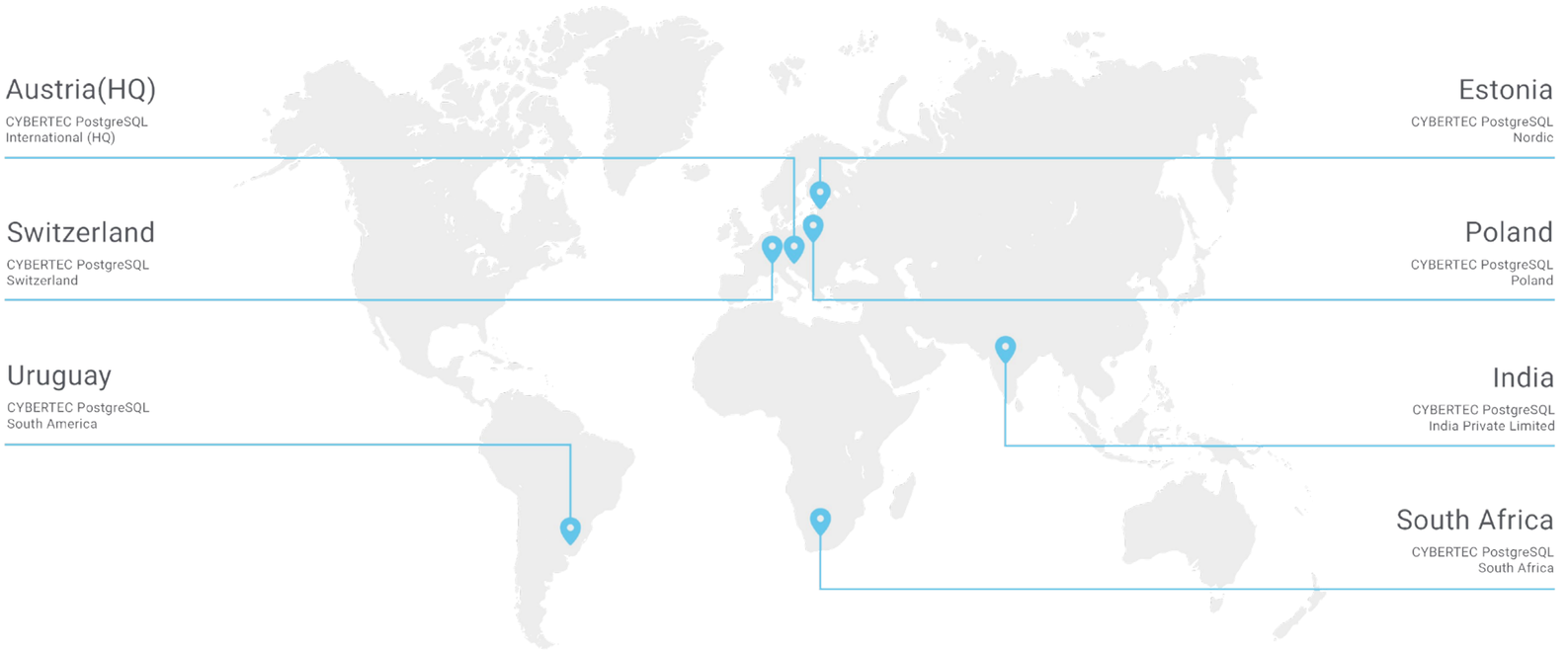
pg_hardstorage

Storage internals of a PostgreSQL backup engine

Hans-Jürgen Schönig — CYBERTEC · Valencia, 21 October 2026

www.pghardstorage.org

CYBERTEC PostgreSQL International



What we will cover

A 45-minute tour, weighted towards storage internals

01 **What pg_hardstorage is**
The short version: scope, and what it deliberately does not do.

02 **How we store data**
Content-defined chunking, content addressing, and the on-disk layout.

03 **The replication protocol**
Why streaming beats reading files, and what it takes to connect.

04 **Patroni integration**
Following a leader across failover without losing WAL silently.

05 **Compatibility**
Replacing pgBackRest, Barman or WAL-G, and how that migration actually goes.

06 **Everything else**
Signing, drills, time travel, and where the LLM helper is going.

Three things it does

Everything else in this talk is a consequence of these



Backup and replay

Streams base backups and WAL, and replays them to any point in time.



Integrates with Patroni

Follows the leader across failover, keeping slots and timelines intact.



Drop-in replacement

Replaces pgBackRest, Barman and WAL-G; your config is translated for you.

What it is, and what it is not

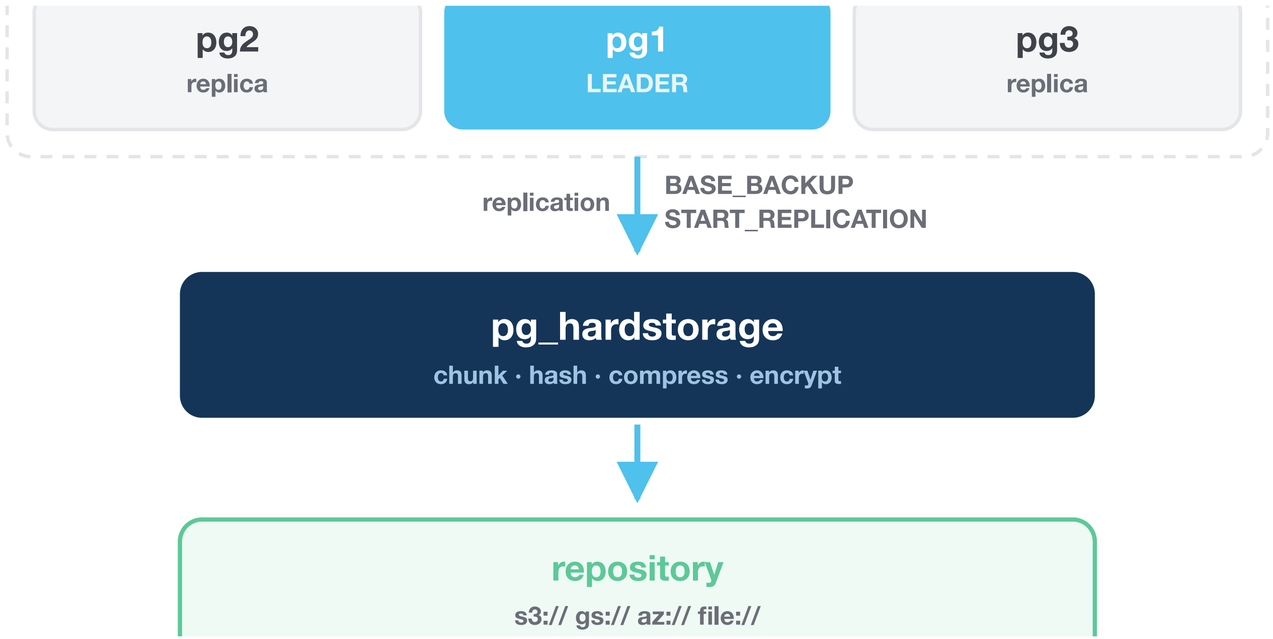
CYBERTEC

Scope, stated before anyone has to ask

- Does **not** work for RDS or other proprietary managed services. They do not expose a replication connection, so there is nothing for this design to attach to.
- Works with every recent PostgreSQL release, 15 through 18. No patched server, no extension, no fork — only the interfaces PostgreSQL already gives you.
- Works over the wire. Nothing has to run on the database host: no agent on the box, no sidecar, no filesystem access to the data directory.

Where pg_hardstorage sits

One replication connection, and a repository



Over the wire, from the leader

It opens a replication connection to whichever node Patroni calls the leader — the same interface `pg_basebackup` uses. Base backup first, then WAL streamed continuously. When the leader changes it re-points and carries on. Nothing is installed on the database hosts.

From nothing to a verified backup

CYBERTEC

day 0

```
# one-time setup: probes PG, creates the repo, generates the signing
# key, writes pg_hardstorage.yaml, takes the first backup
pg_hardstorage init --yes \
    --pg-connection 'host=/var/run/postgresql dbname=postgres' \
    --repo s3://backups/prod

# the config it wrote also schedules the recurring work
#     backup   every 6h         (--schedule-backup)
#     rotate   daily_at 04:00   (--schedule-rotate)

# from then on
pg_hardstorage backup db1        --repo s3://backups/prod
pg_hardstorage verify db1 latest --repo s3://backups/prod
pg_hardstorage list   db1        --repo s3://backups/prod
```

How we store data

Chunks, addresses, envelopes and manifests

Everything in a repository is one of three things

CYBERTEC

Data, the index over it, and the WAL record

Chunks

Variable-length pieces of file data, deduplicated across every backup.

Manifests

One signed record per backup: files, chunk lists, LSNs, timeline.

WAL segment manifests

One record per 16 MB WAL segment, keyed by timeline.

The write path, end to end

backup pipeline

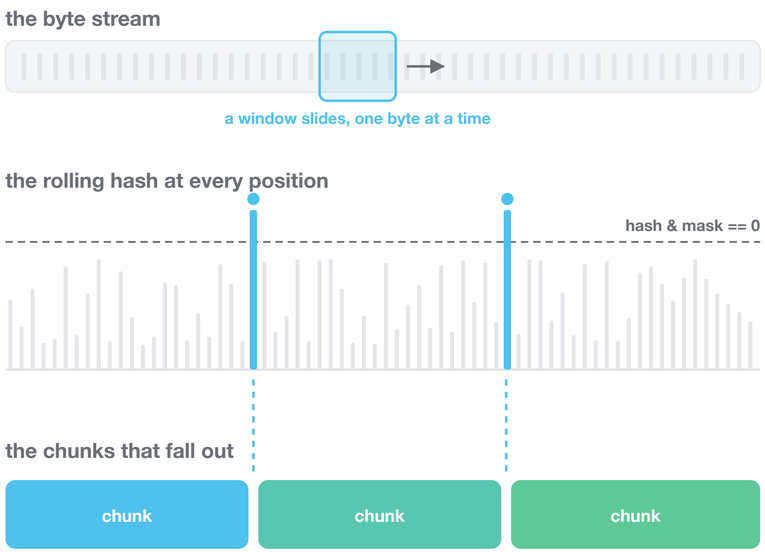
```
postgres ──BASE_BACKUP──► tar stream
                ▼
    [ FastCDC ]  content-defined split ──► chunk chunk chunk
                ▼
    [ SHA-256 ]  content address
                ▼
     already stored? ──yes──► skip the write
                ▼ no
    [ zstd ] ─► [ AES-256-GCM ]
                ▼
    chunks/sha256/aa/bb/<hash>.chk

content address: the chunk's name IS the SHA-256 of its bytes -- so
identical data always lands on one name, and corruption cannot hide.
```

Why we cut where the data says, not every 8 KiB

Fixed-size blocks lose to insertion

- A backup differs from the last one mostly by *insertion*, not overwrite. Page splits, VACUUM and reorganisation all shift bytes along.
- Fixed-size blocks re-align after an inserted byte, so every later block hashes differently and nothing deduplicates.
- FastCDC hashes a sliding window and cuts where the hash matches a mask — a boundary the *data* picks, so it survives insertion.



4 KiB minimum · 64 KiB average · 256 KiB maximum

The boundary is chosen by the content, never by the offset.

One inserted byte, two outcomes

CYBERTEC

boundary alignment

```
Fixed 8 KiB blocks — insert one byte at the front:

  before  |AAAAAAAA|BBBBBBBB|CCCCCCCC|DDDDDDDD|
  after   |xAAAAAAA|ABBBBBBB|BCCCCCCC|CDDDDDDD|
           ^^^^^^^^ ^^^^^^^^ ^^^^^^^^ ^^^^^^^^
           every block changed -> nothing dedupes

Content-defined boundaries — cut where the data says:

  before   |AAAAA|BBBBBBB|CCCC|DDDDDDDDD|
  after   |xAAAAA|BBBBBBB|CCCC|DDDDDDDDD|
           ^^^^^^  reused  reused  reused
           one chunk changed, the rest are free
```

The chunker, in numbers

CYBERTEC

`FastCDC parameters`

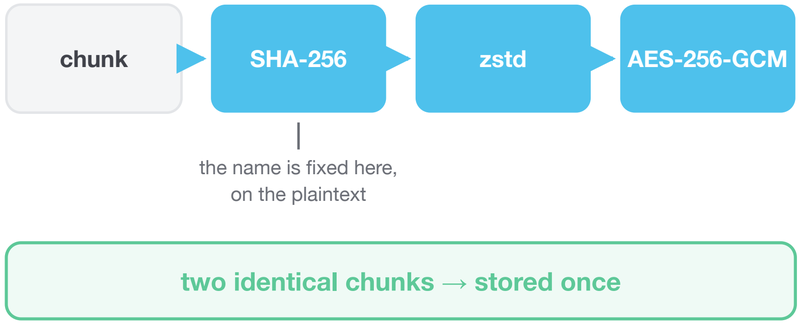
Parameter	Value	Why this value
Minimum chunk	4 KiB	Below this, per-chunk metadata starts to dominate
Target average	64 KiB	Balances deduplication ratio against index size
Maximum chunk	256 KiB	Bounds worst-case memory and re-upload cost
Hash seed	0xC0FFEE_BEAD_42	Fixed: identical data must always cut identically
Digest	SHA-256	Content address and integrity check in one value

The seed fixes where the rolling hash finds a boundary. Change it and identical data cuts at different offsets, hashes differently, and deduplicates against nothing — every backup would upload in full. A format constant, not a tuning knob.

Hash first, then compress, then encrypt

Reverse any two and deduplication dies

- The address is the SHA-256 of the *plaintext*, so it never depends on the codec or the compression level.
- Change the level and the same data keeps its address — new chunks still deduplicate against old ones.
- Encryption uses a fresh nonce per chunk. Encrypt first and identical data gets a different name every time.



**Compression sits in the middle:
it changes the bytes on disk,
never the name.**

Every chunk describes itself

chunk envelope

```
bytes        0         1         2          3 – 14      15 onwards
        +---------+---------+----------+--------------+------------+
        | version |   algo  | enc_algo |    nonce     |  payload   |
        +---------+---------+----------+--------------+------------+
          1 byte    1 byte     1 byte      12 bytes      the rest
        |<------------- 15-byte header -------------->|

  version   format revision, so old chunks stay readable
  algo      compression codec, or none
  enc_algo  encryption codec, or none
  nonce     per chunk, never reused; reserved even when unencrypted
  payload   compressed, then encrypted, plus a 16-byte GCM tag --
            AES-GCM's checksum: flip a bit and decryption fails loudly
```

From digest to path

CYBERTEC

content-addressed layout

```
hash = sha256(plaintext chunk)
     = aabbcc7f19 ... e04        (64 hex characters)

  chunks/sha256/aa/bb/aabbcc7f19...e04.chk
                |  |  |
                |  |  └ the full digest
                |  └ characters 3-4  ─┐  two levels
                └ characters 1-2  ────┘  of fan-out

  2 levels x 256 = 65 536 leaf directories

Why bother: object stores and filesystems both degrade
with a million entries under one prefix. Two levels keep
every directory small at any repository size.
```

The manifest is the backup

CYBERTEC

Chunks are just bytes; the manifest is what makes them a cluster

- One document per backup: every file, its mode and size, and the ordered list of chunk hashes that reconstruct it.
- Serialised deterministically — sorted keys, fixed number formatting — so the same backup always produces byte-identical JSON to sign.
- Signed with Ed25519. Verification needs nothing but the repository: the public key travels with the manifests.
- Written to a random `.tmp` name, then renamed only if the target does not already exist. Two writers racing cannot corrupt a manifest.
- A second copy lands under `_replicas/`. Losing one object does not lose the backup's index.

The repository on disk

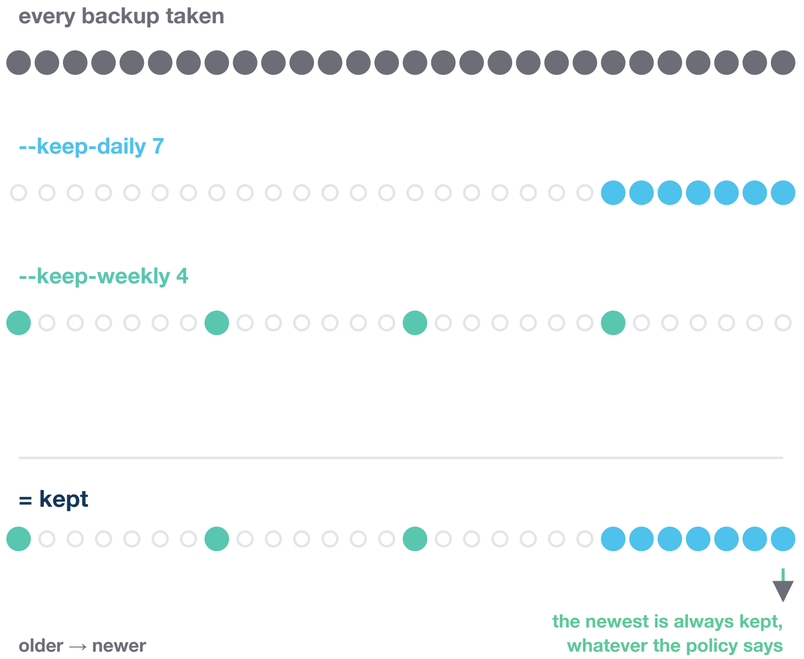
repository layout

```
repo/
├── chunks/sha256/aa/bb/<hash>.chk     deduplicated data
│
├── manifests/<deployment>/<id>.json   signed backup record
│   └── _replicas/                     second copy of each
│
├── wal/<deployment>/<tli>/<seg>.json  WAL segment manifest
│                       └ 8-digit HEX
│
├── timelines/<tli>.history            copied from postgres
│              └ DECIMAL, as postgres names it
│
├── gaps/<tli>-<unixnanos>.json        recorded WAL gaps
└── leases/<deployment>/backup.json    writer mutex
```

Retention: which backups survive

Each rule protects a slice; you keep the union

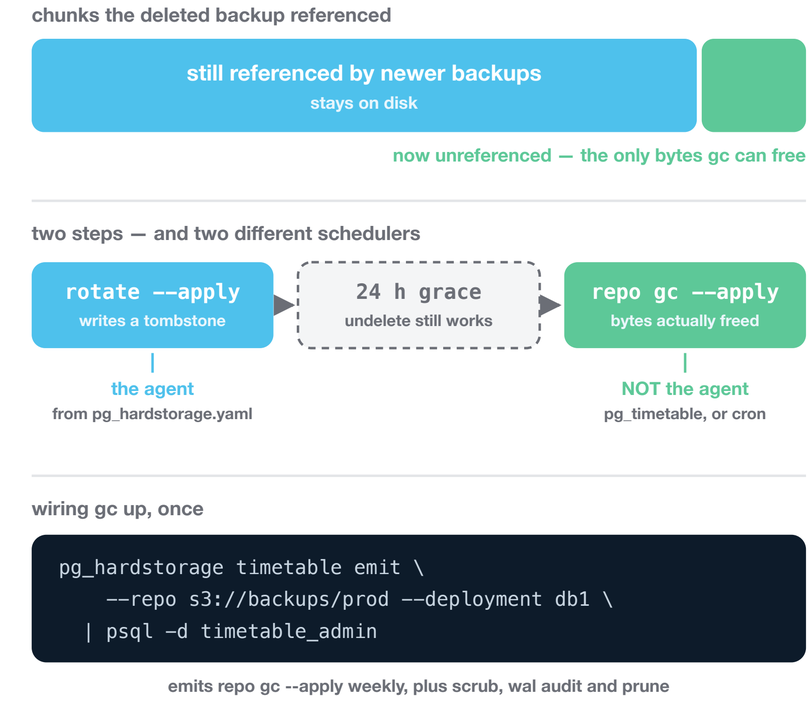
- Grandfather-father-son is the default: one backup per day, week, month and year, for the counts you give. `simple` keeps everything younger than a duration; `count` keeps the last N fulls.
- A backup protected by *any* rule is kept. The rules do not compete — the survivors are the union, which is why the daily and weekly rows overlap.
- The policy comes from your `--keep-*` flags, else the deployment's `retention:` block, else the built-in default — and `rotate` prints which of the three it used.



Out-phasing: what deleting actually frees

In a deduplicated repository, far less than you think

- Dropping a backup from the policy does not delete its data. Most of its chunks are shared with the backups you are keeping, and those stay exactly where they are.
- `repo gc` walks every live manifest, builds the set of chunk hashes still referenced, and sweeps only what is left over. That remainder is usually a thin slice.
- And the two halves are scheduled by different systems. The agent runs `rotate` from `pg_hardstorage.yaml`; it has no `gc` task at all, so the bytes are reclaimed by `pg_timetable` or cron — or never.
- The three schedules compose: `backup` bounds replay time and advances the frontier that lets old WAL be pruned, `rotate` drops what the policy no longer wants, `gc` reclaims the bytes.



One number, three spellings

PostgreSQL itself is not consistent here: WAL file names carry the timeline in hex, .history files use decimal. We follow PostgreSQL in each place rather than inventing a third convention — an operator should be able to guess the path.

Timeline across repository artifacts

Artifact	Timeline written as	Example
WAL segment directory	8-digit hex	wal/db1/00000002/
Timeline history	decimal	timelines/2.history
Gap record	decimal + unix nanos	gaps/2-1815...json
Manifest field	integer	"timeline": 2

The replication protocol

What actually crosses the wire

Why stream instead of reading files

The same interface every physical replica already uses

PostgreSQL does the sending

It hands over a consistent copy and then a WAL stream. No reading data files behind its back.

Run it anywhere

A backup host, a container, another data centre. Nothing is installed next to the database.

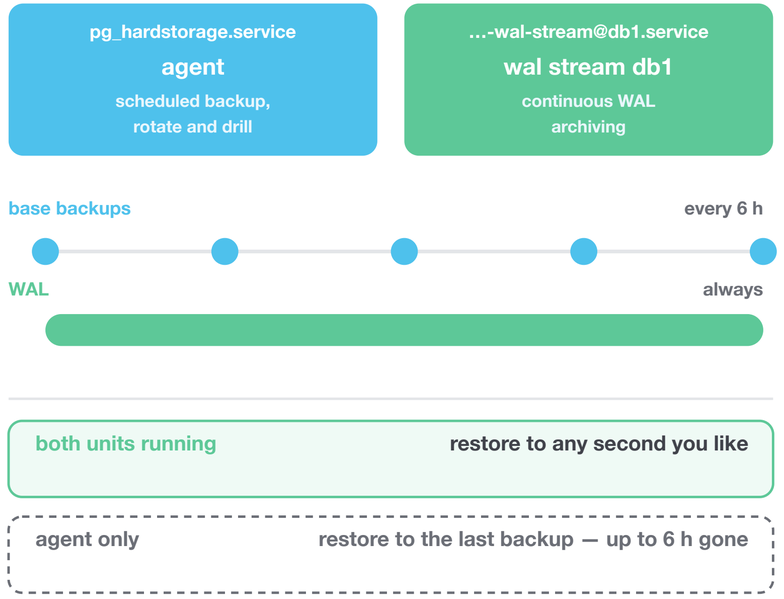
Nothing exotic

The same protocol pg_basebackup and every streaming replica speak. No extension, no patch.

Two processes, not one

The agent never opens a WAL stream

- `pg_hardstorage.service` runs the agent: scheduled backup, rotate and drill. That is all it does.
- `pg_hardstorage-wal-stream@db1.service` runs `wal stream` — the always-on half. It ships in the packages from v1.5; before that the docs pointed at the agent and continuous archiving silently never ran.
- Enable both, per deployment. With only the agent you can restore to a backup; the minutes since the last one are not in the repository at all.



The agent never opens a WAL stream. That is the other unit.

Connecting is just PostgreSQL

CYBERTEC

An ordinary client connection, nothing bespoke

- A standard connection string — the same URL or keyword/value DSN you already use. Host, port, user, `sslmode`, all of it.
- Authentication is PostgreSQL's own: scram-sha-256, md5, and TLS client certificates. Kerberos/GSSAPI is the one common method not wired up today.
- The role needs `REPLICATION`, and a `pg_hba.conf` line permitting replication connections. That is the whole server-side setup.
- Encrypt the link with TLS the way you already do. pg_hardstorage adds no transport, no daemon and no port of its own.

Restoring to a point in time

CYBERTEC

restore

```
# what can I restore to?
pg_hardstorage recovery windows db1 --repo s3://backups/prod

pg_hardstorage restore db1 latest --repo s3://backups/prod \
    --target /var/lib/pg/17 \
    --to '2026-10-20 14:30' --to-action promote

#  db1           the deployment, as named in your config
#  latest        which backup to start from, or a backup id
#  --repo        where the chunks and manifests live
#  --target      an empty directory; the data dir is built here
#  --to          the recovery target -- see the next slide
#  --to-action   pause, promote or shutdown once it arrives
#  --preview     plan it all and write absolutely nothing
```

Choosing the recovery target

CYBERTEC

Pick one. With no target at all a restore boots using only the WAL bundled inside the backup and silently ignores everything archived since — which is why --to-latest is the ordinary disaster-recovery shape.

`restore targets`

Flag	What it does
--to <time>	Stop at a time — RFC3339, or natural language
--to-lsn <lsn>	Stop at a log position, e.g. 0/3000028
--to-name <name>	Stop at a named restore point
--to-latest	No target: replay every segment the repository holds
--to-exclusive	Stop just before the target instead of just after
--to-action	On arrival: pause (default), promote, or shutdown

Patroni, compatibility, and the rest

Operating this in a cluster that fails over

Following the leader

patroni follower

```
every 5s:   GET http://patroni:8008/cluster

  { "members": [
      { "name": "pg1", "role": "leader",  "timeline": 3 },
      { "name": "pg2", "role": "replica", "timeline": 3 } ] }

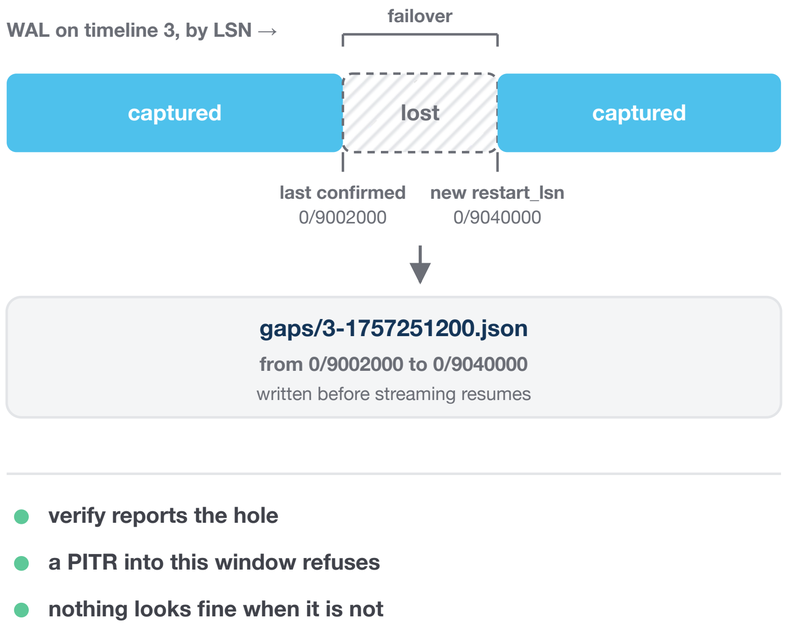
  leader unchanged  ──────────────────►  keep streaming

  leader changed:
    1. ensure our replication slot exists on the new leader
    2. fetch TIMELINE_HISTORY, backfill every missing TLI
    3. if WAL was lost, write gaps/<tli>-<nanos>.json
    4. resume streaming from the new leader
```

When a failover eats WAL

A slot is a promise the new leader may not have made

- A replication slot is PostgreSQL's promise not to recycle WAL we have not confirmed yet. After a failover, the new leader may not carry that slot.
- A fresh slot reserves WAL from *now*, not from where we stopped. Whatever fell between the two positions is already gone — that byte range is the gap.
- So we write it down before resuming. Verification reads the record, and a restore into that window refuses rather than producing a cluster that only looks fine.



After a failover, what you actually run

CYBERTEC

failover diagnostics

```
# Patroni's own REST API (default port 8008) -- /cluster returns
# the whole topology: who leads, who follows, which timeline
pg_hardstorage patroni status --url http://pg1:8008

# did the failover cost us WAL?  the second is cron-friendly
# and exits non-zero on findings
pg_hardstorage wal gaps  db1 --repo s3://backups/prod
pg_hardstorage wal audit db1 --repo s3://backups/prod

# the slot did not survive the promotion
pg_hardstorage wal repair db1 --repo s3://backups/prod

# what can we still recover to?
pg_hardstorage recovery readiness db1 --repo s3://backups/prod
```

You do not have to switch in one step

CYBERTEC

migration

```
# translate the config you already have
pg_hardstorage compat translate --from pgbackrest \
    /etc/pgbackrest/pgbackrest.conf --output pg_hardstorage.yaml

#  --from pgbackrest | barman | walg
#  settings with no clean equivalent come out as YAML
#  comments, with a summary on stderr to review

# PostgreSQL's integration points keep their usual shape
archive_command =
  'pg_hardstorage wal push db1 %p --repo <url>'
restore_command =
  'pg_hardstorage wal fetch db1 %f %p --repo <url>'
```

The rest of the surface

Briefly, so you know it exists

Signed manifests

Ed25519 over a deterministic serialisation; the public key travels with the repository.

Time travel

An ephemeral read-only PostgreSQL pinned to a historical point, with a TTL so a forgotten session gets reaped.

Recovery drills

Restore a real backup to a temporary directory, run pg_verifybackup, report the actual RTO, tear it down.

Six storage backends

The same repository layout on every one of them.

Prometheus metrics

A documented catalogue, with tests that fail when the docs drift from the code.

LLM helper, in the making

Explain a command, or let --on-error-llm drop you into the matching skill when one fails. Still taking shape.



Hans-Jürgen Schönig

CEO & Founder, CYBERTEC

Email

hs@cybertec-postgresql.com

Phone

+43 2622 930 22-666

- www.pghardstorage.org
- www.cybertec-postgresql.com