05 Continuous Archiving and PITR

01 Why WAL Archiving Exists (Foundation of PITR)

In simple words

- WAL archiving exists so PostgreSQL can **go back in time**.
- A normal backup gives you one fixed restore point.
- WAL archiving gives you **every change after that point**.
- This is what enables **Point-In-Time Recovery (PITR)**.

The problem without WAL archiving

Imagine this:

- Full backup taken at 01:00 AM
- Accident happens at 11:37 AM

Without WAL archiving:

- you can restore only till 01:00 AM
- you lose ~10 hours of data

For many businesses, this data loss is unacceptable.

What WAL already does internally

PostgreSQL always writes changes in this order:

- change is written to WAL
- WAL is flushed to disk
- data pages are written later

So WAL already contains a complete **change history**.

WAL archiving simply **preserves this history instead of deleting it**.

What WAL archiving means

WAL archiving means:

- completed WAL files are copied
- copied to a safe external location
- before PostgreSQL removes them

This creates a continuous timeline of changes.

Base backup + WAL = full recovery chain

Think in two parts:

1. Base backup

- gives starting point
- file-level snapshot of database

2. Archived WAL files

- describe every change after backup

Together, they allow recovery to **any moment after the base backup**.

What PITR really allows

With WAL archiving, I can:

- recover to a specific timestamp
- recover before a bad transaction
- recover to last known good state

This is impossible with backups alone.

Why WAL archiving is mandatory in production

In real systems:

- human mistakes happen
- scripts fail
- bugs delete data

WAL archiving:

- minimizes data loss
- gives DBAs confidence
- reduces panic during incidents

Senior DBAs treat it as mandatory.

Common misunderstanding

Myth:

> "I have daily backups, that's enough"

Reality:

- backups define recovery *points*
- WAL defines recovery *continuity*

Both are needed for real protection.

Storage requirements

WAL archiving requires:

- reliable storage
- enough space
- cleanup/retention policy

If archive storage fails, PITR fails.

What WAL archiving does NOT replace

WAL archiving:

- does NOT replace base backups
- does NOT replace logical backups
- does NOT store configuration files

It complements backups, it doesn't replace them.

Final mental model

- Base backup = starting line
- WAL files = change timeline
- PITR = choose your restore moment
- Archiving = safety guarantee

One-line explanation

WAL archiving preserves PostgreSQL change history so databases can be restored to any point in time after a base backup.

02 WAL Level, archive_mode, and archive_command (How WAL Archiving Actually Works)

In simple words

To make WAL archiving work, PostgreSQL needs three settings to cooperate:

- `wal_level`
- `archive_mode`
- `archive_command`

If even one is wrong, WAL archiving silently fails.

Why configuration matters

PostgreSQL never guesses your intention.

Unless these settings are explicitly correct:

- WAL files are recycled
- change history is lost
- PITR becomes impossible

That's why WAL archiving failures are usually **configuration failures**.

`wal_level` (how much information WAL stores)

What `wal_level` means

`wal_level` defines **how much detail** PostgreSQL writes into WAL.

Common values:

- `minimal` (*just survive crashes*)
- `replica` (*backups + replicas*)
- `logical` (*stream changes outside PGSQL*)
- `wal_level` decides how much information PGSQL writes into WAL.
- More information = more features, but also more WAL size.
- **`minimal`**: This writes only the bare minimum WAL needed for crash recovery. It's lightweight and fast, but you cannot use replication or PITR. This is fine for simple, standalone databases where you only care about basic safety.
- **`replica`**: This writes enough WAL data to support physical replication and PITR. It's the most common setting in production systems. You get crash recovery, replicas, and backups, with acceptable overhead.
- **`logical`**: This writes the most detailed WAL. In addition to everything in replica, it includes information needed for logical decoding. This allows logical replication, change data capture, and streaming changes to external systems. It produces more WAL but enables advanced architectures.

Which `wal_level` is required

For WAL archiving:

```
wal_level = replica
```

Why:

- `minimal` does not generate enough WAL
- `replica` guarantees crash recovery and PITR

Logical replication requires `logical`, but PITR does not.

archive_mode (turns archiving on)

What archive_mode does

`archive_mode` tells PostgreSQL:

> "Do not delete WAL until it is archived somewhere safe."

Enable it with:

```
archive_mode = on
```

Without this, PGSQL reuses WAL files automatically.

`archive_command` (how WAL is archived)

What `archive_command` is

`archive_command` is a **shell command** executed every time a WAL segment is completed.

PostgreSQL runs it like:

```
archive_command %p %f
```

Where:

- `%p` = full path to WAL file
- `%f` = WAL file name

A simple `archive_command` example

```
archive_command = 'cp %p /backup/wal_archive/%f'
```

Meaning:

- copy WAL file to archive directory
- only mark success if command exits with 0

If command fails, PostgreSQL retries.

Why `archive_command` failures are dangerous

If `archive_command`:

- returns non-zero
- hangs
- writes to full disk

Then:

- WAL is not archived
- WAL cannot be recycled
- `pg_wal` directory grows
- database may stop

This is a classic production outage.

Testing WAL archiving

Always verify:

```
SELECT * FROM pg_stat_archiver;
```

Check:

- `archived_count` increases
- `failed_count` stays zero

Never trust configuration blindly.

Reload vs restart

Changes to these parameters require:

- `wal_level` → restart
- `archive_mode` → restart
- `archive_command` → reload

Restart planning is required.

Security and permissions

`archive_command` runs as:

- PostgreSQL OS user

Ensure:

- archive directory permissions are correct
- command cannot be exploited

Security mistakes here leak data.

Common DBA mistakes

- forgetting restart after `wal_level` change
- wrong archive path
- no monitoring of archive failures
- assuming archive = backup

WAL archiving is only as good as its validation.

Final mental model

- `wal_level` = how much history
- `archive_mode` = keep history
- `archive_command` = where history goes

All three must work together.

One-line explanation

PostgreSQL WAL archiving depends on `wal_level` for data detail, `archive_mode` to enable archiving, and `archive_command` to store WAL files safely.

03 Base Backup Using `pg_basebackup` (Foundation of PITR)

In simple words

- A base backup is **a full physical copy of the entire PostgreSQL cluster** taken at a specific point in time.
- The `pg_basebackup` tool is used to create this backup safely while the database is still running.
- WAL files by themselves cannot restore anything unless there is a base backup to start from.

Why base backup is required

- A base backup is required because WAL files only contain the changes made to the database, not the original data.
- To restore a database, PGSQL first needs a base backup as the starting point and then replays WAL files on top of it.
- Without the base backup, WAL files have nothing to apply to, so recovery is impossible.

What `pg_basebackup` actually does

`pg_basebackup`:

- connects to PGSQL as a replication client
- copies the entire data directory
- ensures consistency using WAL
- optionally streams WAL during backup

It is WAL-aware by design.

Role and permission requirements

`pg_basebackup` requires:

- superuser, or
- role with REPLICATION and BACKUP privileges

A normal database user cannot take a base backup.

Basic `pg_basebackup` command

```
pg_basebackup -D /backup/base -Fp -X stream -P
```

Meaning:

- `-D` → destination directory
- `-Fp` → plain file format
- `-X stream` → stream WAL during backup
- `-P` → show progress

This creates a consistent physical backup.

WAL handling during base backup

Two common options:

Stream WAL (recommended)

```
-X stream
```

- WAL is streamed live
- safest option
- avoids missing WAL segments

Fetch WAL after backup

```
-X fetch
```

- WAL is copied after data files
- riskier if WAL is recycled too fast

Streaming is preferred in production.

Compression and performance

`pg_basebackup` supports compression:

```
pg_basebackup -D /backup/base -Fp -X stream -Z 9
```

Higher compression:

- reduces disk usage
- increases CPU load

Balance based on system capacity.

Using tar format

```
pg_basebackup -D /backup/base -Ft -X stream
```

Tar format:

- creates archive files
- easier to move
- slower to extract during restore

Impact on running database

During `pg_basebackup`:

- read I/O increases
- WAL generation increases
- archive pressure rises

This must be monitored on production systems.

Restoring from a base backup

Restore steps:

- stop PostgreSQL
- clean or replace PGDATA
- copy base backup into place
- configure recovery settings
- start PostgreSQL

WAL replay completes the restore.

Common pg_basebackup mistakes

- running without WAL streaming
- insufficient disk space
- wrong permissions on destination
- forgetting tablespaces

Base backups must be tested.

When I use pg_basebackup

I use it when:

- PITR is required
- physical backups are primary recovery
- downtime must be minimal

It is the backbone of serious recovery setups.

Final mental model

- Base backup = starting snapshot
- pg_basebackup = safe physical copier
- WAL streaming = consistency guarantee
- Restore = base + WAL replay

One-line explanation

pg_basebackup takes a consistent physical snapshot of a PostgreSQL cluster, forming the base for WAL-based recovery and PITR.

04 WAL Archiving Flow and Internals (How PostgreSQL Moves WAL)

In simple words

WAL archiving is not magic.

PGSQL follows a strict lifecycle for every WAL segment:

- create
- write
- close
- archive
- recycle

Understanding this flow is the key to debugging PITR issues.

WAL lifecycle at a high level

Each WAL segment goes through these stages:

1. active in `pg_wal`
2. completed and ready to archive
3. archived successfully
4. recycled or removed

Archiving decides when a WAL is safe to delete.

- At a high level, each WAL segment has a simple life cycle. It is first actively **written inside `pg_wal`**, then it gets completed and **becomes ready for archiving**. Once it is successfully archived, PGSQL knows it is safe, and only after that the **segment is recycled or removed**. Archiving is what decides when a WAL file can be safely deleted.

How a WAL segment is created

- PGSQL writes changes continuously
- WAL files are written sequentially
- default WAL segment size is **16MB**

While a WAL file is active:

- it cannot be archived
- PostgreSQL keeps writing to it
- PGSQL writes all changes continuously into WAL, and it always writes WAL files in a sequential order. Each WAL segment has a fixed size, which by default is 16 MB.
- While a WAL file is active and still being written to, it cannot be archived, so PGSQL keeps appending changes to it until it becomes full and is closed.

When a WAL becomes archive-ready

A WAL segment becomes archive-ready when:

- it is completely filled, or
- PGSQL switches to a new WAL file

At this point:

- PGSQL calls `archive_command`
- the WAL file is copied to archive storage

`archive_command` execution flow

For each completed WAL file:

- PostgreSQL runs `archive_command`
- passes `%p` (path) and `%f` (filename)
- waits for success

If the command fails:

- WAL is retried
- WAL is not removed

This guarantees no WAL is lost silently.

What happens on archive failure

If archiving fails repeatedly:

- WAL files accumulate in `pg_wal`
- disk usage grows
- database may stop accepting writes

This is one of the most common PITR-related outages.

How PGSQL knows WAL is archived

PGSQL tracks:

- successful archive operations
- failed archive attempts

You can inspect this using:

```
SELECT * FROM pg_stat_archiver;
```

This view is your first stop during debugging.

WAL recycling vs archiving

Without archiving:

- WAL files are reused when safe

With archiving enabled:

- WAL files are kept until archived
- recycling waits for archive success

Archiving changes WAL cleanup behavior.

Timelines (basic concept)

Each recovery creates a new **timeline**.

Timelines allow:

- divergence from old history
- safe recovery without overwriting past states

WAL files belong to specific timelines.

Why timelines matter

During PITR:

- PostgreSQL selects correct WAL timeline
- old timelines are preserved

Mixing WAL from different timelines causes restore failure.

Where DBAs get confused

Common confusion points:

- WAL files not disappearing
- pg_wal growing endlessly
- archive directory filling up

These are symptoms, not bugs.

DBA debugging checklist

When WAL archiving issues occur, I check:

- `pg_stat_archiver`
- archive_command exit behavior
- disk space on archive destination
- permissions

Most issues are operational, not PostgreSQL bugs.

Final mental model

- WAL flows forward
- Archiving freezes history
- Recycling waits for safety
- Timelines protect recovery paths

One-line explanation

PostgreSQL archives WAL segments only after they are completed, using `archive_command`, and manages retention through strict lifecycle and timeline rules.

05 Point-in-Time Recovery (PITR) in PostgreSQL

In simple words

Point-in-Time Recovery (PITR) allows PGSQL to **rewind the database to an exact moment in time**.

Instead of restoring only to the time of the last backup, PITR lets me restore to:

- a specific timestamp
- just before a bad transaction
- the last known good state

This is the **real power** of WAL archiving.

Why PITR exists

Backups alone restore databases to fixed points.

Real incidents happen between backups:

- accidental DELETE
- bad deployment
- faulty script

PITR exists so I can recover data **without losing hours of work**.

What PITR actually needs

PITR requires two things:

1. a base backup
2. continuous WAL archive

If either is missing, PITR is impossible.

How PITR works (high-level flow)

1. Restore base backup
2. PostgreSQL starts in recovery mode
3. WAL files are replayed sequentially
4. Replay stops at chosen recovery point
5. Database becomes usable

Recovery is deterministic and repeatable.

Choosing a recovery target

PostgreSQL allows recovery based on:

- timestamp
- transaction ID
- named restore point

Most commonly, timestamp-based recovery is used.

Timestamp-based recovery example

If accident happened at `2025-02-10 11:37:00`, I recover to:

```
recovery_target_time = '2025-02-10 11:36:59'
```

This restores the database to just before the mistake.

What happens during WAL replay

During recovery:

- WAL changes are applied
- committed transactions are replayed
- uncommitted ones are skipped

PostgreSQL ensures consistency automatically.

Recovery targets and safety

Recovery stops when:

- target time is reached
- or required WAL is missing

If WAL is missing:

- recovery fails
- data loss occurs

This is why WAL retention is critical.

After recovery completes

Once recovery stops:

- PostgreSQL creates a new timeline
- old WAL history is preserved
- database starts accepting writes

Recovery cannot continue past this point unless re-restored.

Common PITR mistakes

- missing WAL files
- wrong recovery target
- restoring into dirty PGDATA
- forgetting to switch to new timeline

Most PITR failures are procedural errors.

Testing PITR

A PITR setup must be tested:

- simulate bad deletes
- recover to a time before error
- verify data correctness

Untested PITR is false confidence.

Final mental model

- Base backup = starting point
- WAL archive = full history
- PITR = controlled rewind
- Testing = real safety

One-line explanation

Point-in-Time Recovery allows PostgreSQL to restore a database to an exact moment using a base backup and archived WAL files.

06 recovery.signal and restore_command (How PostgreSQL Enters Recovery)

In simple words

PGSQL does **not guess** when to perform recovery.

It enters recovery mode only when I explicitly tell it to.

That signal comes from two things:

- `recovery.signal`
- `restore_command`

If either is missing or wrong, recovery does not work.

How PostgreSQL decides to start recovery

When PostgreSQL starts, it checks the data directory.

If it finds:

- `recovery.signal` → start recovery mode

If it does not find it:

- PostgreSQL starts normally
- WAL replay stops

This small file controls everything.

What is `recovery.signal`

`recovery.signal` is an **empty file** placed in `PGDATA`.

Its presence means:

- "This cluster must recover using archived WAL."

It replaces older `recovery.conf` (pre-PostgreSQL 12).

Why `recovery.signal` exists

Before PostgreSQL 12:

- recovery was controlled by `recovery.conf`

Now:

- recovery settings live in `postgresql.conf`
- `recovery.signal` only triggers recovery mode

This simplifies startup logic.

What is restore_command

`restore_command` tells PostgreSQL:

- "Where and how to fetch archived WAL files."

It is a shell command executed during recovery.

Example:

```
restore_command = 'cp /backup/wal_archive/%f %p'
```

Meaning:

- `%f` = WAL file name
- `%p` = path where PostgreSQL expects WAL

`restore_command` execution flow

During recovery:

- PostgreSQL requests next WAL file
- runs `restore_command`
- expects the file to be placed at `%p`

If command succeeds:

- WAL is replayed

If command fails:

- recovery stops

Common `restore_command` mistakes

- wrong archive path
- incorrect permissions
- command returns non-zero
- missing WAL file

Any one of these breaks recovery.

Full recovery setup example

Steps:

1. Restore base backup into PGDATA
2. Place `recovery.signal` file
3. Configure restore_command
4. (Optional) set recovery_target_time
5. Start PostgreSQL

PostgreSQL handles the rest.

What happens after recovery completes

Once recovery reaches its target:

- PostgreSQL removes `recovery.signal`
- creates a new timeline
- starts accepting writes

Recovery mode ends automatically.

When recovery does NOT stop

Recovery keeps waiting when:

- target WAL is not available
- `restore_command` cannot fetch WAL

PostgreSQL will keep retrying until WAL appears.

DBA verification during recovery

I monitor:

- PostgreSQL logs
- recovery progress messages
- WAL fetch activity

Logs tell exactly what is missing.

Final mental model

- `recovery.signal` = enter recovery
- `restore_command` = fetch WAL
- WAL replay = rebuild state
- new timeline = safe future

One-line explanation

PostgreSQL enters recovery mode when recovery.signal is present and uses restore_command to fetch archived WAL files during PITR.

07 Recovery Target Time and Timeline (Deep PITR Behavior)

In simple words

During Point-in-Time Recovery (PITR), PostgreSQL needs two decisions:

1. **Where to stop recovery**
2. **Which history (timeline) to follow**

These are controlled by recovery targets and timelines.

Understanding this avoids accidental data loss during repeated restores.

What is a recovery target

A recovery target tells PostgreSQL:

- "Stop replaying WAL at this exact point."

Without a recovery target:

- PostgreSQL replays WAL until the latest available WAL
- database recovers to the most recent state

With a recovery target:

- recovery stops earlier, by choice

Types of recovery targets

PostgreSQL supports multiple recovery targets:

recovery_target_time

Recover to a specific timestamp.

```
recovery_target_time = '2025-02-10 11:36:59'
```

Most commonly used option.

recovery_target_xid

Recover up to a specific transaction ID.

```
recovery_target_xid = '1234567'
```

Used when the exact failing transaction is known.

recovery_target_name

Recover to a named restore point.

```
SELECT pg_create_restore_point('before_deploy');
```

```
recovery_target_name = 'before_deploy'
```

Useful during planned risky operations.

What happens during recovery replay

During recovery:

- WAL is replayed sequentially
- PostgreSQL checks each record
- stops when the recovery target is reached

Committed transactions before target are applied.

Committed transactions after target are ignored.

What is a timeline

A timeline represents a **history branch** of the database.

Every PostgreSQL cluster starts on timeline 1.

Whenever recovery completes:

- PostgreSQL creates a new timeline
- future WAL belongs to the new timeline

Why timelines are necessary

Timelines prevent accidental overwriting of history.

Example:

- timeline 1 → original database
- timeline 2 → recovered version

Both histories coexist safely.

Timeline files in WAL archive

Timeline information is stored in:

- `.history` files

PostgreSQL reads these to:

- choose correct WAL path
- avoid mixing histories

Missing history files cause recovery failure.

Restoring multiple times (important behavior)

If you restore again from the same base backup:

- PostgreSQL creates another new timeline
- old timeline remains unchanged

This is expected behavior, not a bug.

Common mistakes with recovery targets

- choosing wrong timestamp
- forgetting timezone differences
- missing WAL beyond target
- assuming restore can continue forward later

Recovery stops permanently at the target.

DBA best practices

- always note incident time accurately
- prefer `recovery_target_time`
- keep full WAL history
- understand timeline branching

Final mental model

- Recovery target = stopping point
- WAL replay = rewind mechanism
- Timeline = history branch
- New writes = new future

One-line explanation

Recovery targets define where PITR stops, and timelines ensure PostgreSQL safely branches database history after recovery.

08 PITR – Real-Life Disaster Scenario (How DBAs Actually Use It)

In simple words

PITR sounds theoretical until **something really bad happens**.

This section walks through a **real production-style disaster** and shows how PITR saves data step by step — exactly how a DBA thinks and acts.

The real situation (very common)

- Production PostgreSQL database
- WAL archiving enabled
- Nightly base backups running

At 11:42 AM:

- A developer runs a wrong DELETE query
- Critical data is deleted
- Transaction is committed

This is **not a crash**.

This is **human error**.

Immediate reality check

What we know:

- Database is still running
- Data is already committed
- Normal rollback is impossible

What we fear:

- Waiting longer will overwrite more WAL
- Panic actions may make recovery harder

This is where PITR matters.

First rule: stop the damage

Before recovery planning:

- stop application access
- prevent further writes

Why:

- every new write creates WAL
- more WAL makes recovery slower and riskier

Freezing the system is critical.

Identify the recovery point

We need to answer one question:

- "To what exact moment should I restore?"

Inputs used:

- application logs
- developer statement
- PostgreSQL logs

We decide:

- bad DELETE happened at **11:42:10 AM**
- safe recovery time = **11:42:09 AM**

One second matters.

Choose recovery strategy

Options:

- logical restore → too slow
- manual data repair → unreliable
- PITR → safest and fastest

Decision:

- Use PITR and rewind the database

High-level recovery plan

The plan is clear:

1. Restore last base backup
2. Replay WAL up to 11:42:09
3. Start database on new timeline

Everything else is noise.

Step 1: Restore base backup

Actions:

- stop PostgreSQL
- clean PGDATA
- restore last base backup files

This brings the database back to **backup time**, not to the final state.

Step 2: Configure PITR

Key settings:

```
restore_command = 'cp /backup/wal_archive/%f %p'
recovery_target_time = '2025-02-15 11:42:09'
```

And place:

```
recovery.signal
```

This tells PostgreSQL:

- “Replay WAL, but stop before the damage.”

Step 3: Start PostgreSQL

Now PostgreSQL:

- enters recovery mode
- fetches WAL sequentially
- replays changes
- stops exactly at target time

Logs confirm recovery stop.

Step 4: New timeline is created

After recovery:

- PostgreSQL creates a new timeline
- old history is preserved
- database starts accepting writes

This prevents accidental replay of bad WAL again.

Step 5: Validate data

Before opening to users:

- verify row counts
- validate critical tables
- confirm deleted data is back

Never trust recovery blindly.

Outcome

- Data loss avoided
- Downtime limited
- No manual fixes needed
- Audit trail preserved

This is exactly why PITR exists.

What would happen without PITR

Without PITR:

- restore last night backup
- lose hours of data
- manual data recreation
- business impact

PITR converts disasters into incidents.

Lessons every DBA must learn

- WAL archiving is non-negotiable
- knowing *when* to stop recovery matters
- calm, structured steps win over panic

Experience is built here.

Final mental model

- Disaster = committed mistake
- PITR = rewind button
- WAL = time machine
- DBA = decision maker

One-line explanation

In a real PITR disaster, a DBA restores a base backup and replays WAL up to just before the damaging transaction to recover lost data safely.

09 Common PITR Failures and Debugging (PostgreSQL)

In simple words

Most PITR failures are not PostgreSQL bugs.

They are **missing files, wrong configs, or bad assumptions**.

This file lists the failures DBAs actually hit and how to debug them calmly.

Failure 1: Recovery does not start at all

Symptom

- PostgreSQL starts normally
- No WAL replay
- Database opens immediately

Root cause

- `recovery.signal` file missing

Fix

- Place an empty `recovery.signal` file in PGDATA
- Restart PostgreSQL

PostgreSQL does not guess recovery intent.

Failure 2: Recovery starts but stops immediately

Symptom

- Recovery messages appear
- Recovery exits very fast

Root cause

- No recovery target set
- Or WAL archive empty

Fix

- Verify WAL files exist
- Set proper `recovery_target_*`

Failure 3: Recovery waits forever

Symptom

- PostgreSQL stays in recovery
- Repeats WAL fetch attempts

Root cause

- Missing required WAL file
- restore_command cannot fetch WAL

Debug

- Check PostgreSQL logs
- Validate archive path
- Test restore_command manually

Failure 4: restore_command fails silently

Symptom

- `pg_wal` requests WAL
- archive directory has files
- recovery still fails

Root cause

- restore_command returns non-zero
- permission or path issue

Debug

```
# test manually
cp /backup/wal_archive/WALFILE /tmp/test
```

Fix permissions or command syntax.

Failure 5: Wrong recovery target time

Symptom

- Data still missing
- Or bad data still present

Root cause

- Wrong timestamp
- Timezone confusion

Fix

- Check `timezone` setting
- Confirm application log time

One-minute mistake = wrong recovery.

Failure 6: Missing timeline history file

Symptom

- Recovery aborts with timeline error

Root cause

- `.history` file missing in archive

Fix

- Ensure timeline history files are archived
- Never delete `.history` files

Failure 7: WAL archive filled disk

Symptom

- Archiving stops
- Database slows or stops

Root cause

- No retention policy
- Archive destination full

Fix

- Clean old WAL safely
- Implement retention automation

Failure 8: Restoring into dirty PGDATA

Symptom

- Random startup errors
- Inconsistent state

Root cause

- Old files mixed with restored files

Fix

- Always restore into empty PGDATA
- Never overlay restore

Failure 9: PITR works once, fails later

Symptom

- First restore works
- Second restore fails

Root cause

- Wrong timeline selected
- Old WAL reused incorrectly

Fix

- Restore from base backup again
- Respect timeline branching

DBA debugging checklist

When PITR fails, I check:

- PostgreSQL logs (first)
- `pg_stat_archiver`
- WAL archive content
- recovery settings
- timestamps and timeline

Logs always tell the truth.

Final mental model

- PITR failures are procedural
- WAL and timelines are fragile
- Logs are your guide
- Calm debugging wins

One-line explanation

Most PITR failures occur due to missing WAL files, incorrect recovery configuration, or timeline mismatches rather than PostgreSQL bugs.