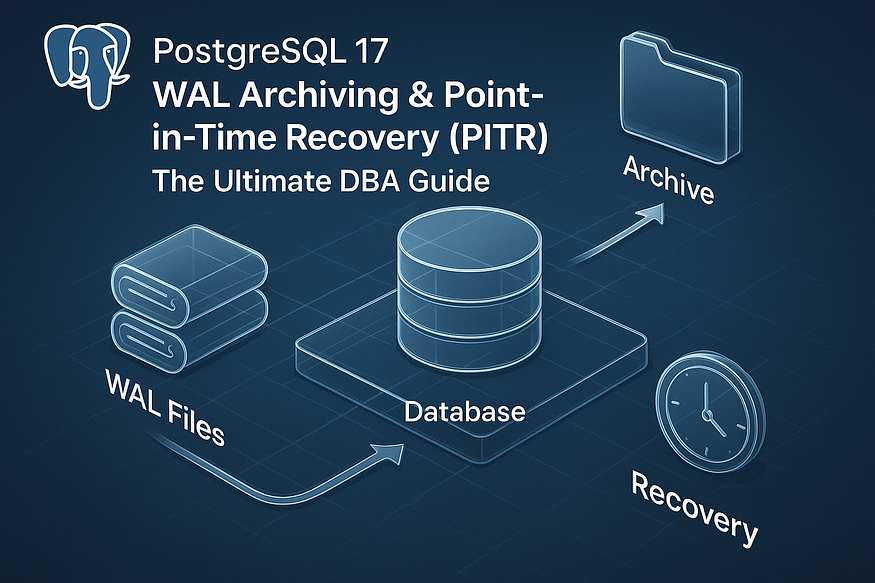
# **PostgreSQL 17 WAL Archiving & Point-in-Time Recovery (PITR): The Ultimate DBA Guide**



PostgreSQL 17 continues to offer rock-solid disaster recovery capabilities through ****Write-Ahead Logging (WAL)**** and ****Point-in-Time Recovery (PITR)****. If you’re a PostgreSQL DBA or architect managing critical production workloads, WAL archiving is your safety net.

In this guide, you’ll learn:

* ✅ What WAL is & how it works
* ✅ How to configure WAL archiving in PostgreSQL 17
* ✅ How to monitor, troubleshoot, and verify archiving
* ✅ Best practices for PITR readiness

Let’s go hands-on! 🔧

## **📖 What Is Write-Ahead Logging (WAL) in PostgreSQL?**

Write-Ahead Logging (WAL) is a foundational mechanism in PostgreSQL that ensures ****data integrity, durability, and crash recovery****. It’s a crucial component for any serious DBA, and here’s an in-depth look at its role and importance.

## **🧱 WAL: The Basis of Durability and Crash Recovery**

### **What WAL Does:**

* ****Before any changes are applied to data files****, PostgreSQL first writes these changes to a ****write-ahead log**** — a series of sequential files (typically 16 MB each).
* This log contains every modification (INSERT, UPDATE, DELETE, DDL operations) the database performs.

### **Why It Matters:**

* If a crash occurs (hardware failure, power loss, or OS crash), PostgreSQL uses the WAL to ****replay all committed changes**** that haven’t yet been flushed to disk.
* This ensures that your database is always restored to the ****most recent consistent state****, safeguarding against data loss.

## **🔄 WAL Makes PITR Possible**

Point-in-Time Recovery (PITR) leverages WAL to allow you to ****restore the database to any specific moment in time****, not just the moment of your last backup. Here’s how it works:

1. ****Base backup**** is taken at time T₀.
2. PostgreSQL continues generating WAL segments.
3. When needed, you restore the base backup.
4. Then you ****replay WAL files**** from T₀ up to your desired recovery point (e.g., a crash or a specific timestamp).

This capability is critical for recovery from:

* User errors (like a faulty UPDATE statement)
* Application bugs (mass data deletions)
* Recent crashes where you don’t want to lose minutes — or even seconds — of data

## **🧩 Summary: Why WAL Is Essential**

* 🆗 ****Guarantees durability****: Committed transactions aren’t lost — even if PostgreSQL hasn’t yet written to data files.
* 🔵 ****Enables crash recovery****: Automatically restores your database to a consistent state after unexpected failures.
* ⏳ ****Supports PITR****: Lets you rewind or fast-forward database state to any timestamp since your last base backup.

Understanding WAL is essential for anyone tasked with ****database reliability, performance, and recovery planning****. In our next section, we’ll dive into how you ****configure WAL archiving**** in PostgreSQL 17 — and how to monitor it for bulletproof disaster preparedness.

## **🔎 Where Are WAL Files Stored?**

At the heart of PostgreSQL’s reliability is the Write-Ahead Log (WAL). Every change your database makes — INSERTs, UPDATEs, DELETEs — is first written to WAL files before being applied to data files. This ensures data consistency and crash recovery.

## **📍 Default Location:**$PGDATA/pg\_wal

* ****$PGDATA**** refers to your database’s data directory. Common locations include /var/lib/postgresql/<version>/main on Linux or custom paths defined during initialization.
* Inside this directory, you’ll find ****pg\_wal/**** (in older versions, called pg\_xlog/).
* WAL segments are fixed-size files — ****16 MB**** by default — with names like:

00000001000000000000000A 00000001000000000000000B

* Each name encodes the log timeline and sequence. As transactions occur, PostgreSQL writes to the current segment. When it fills, it steps to a new one.

## **🔄 WAL Segment Lifecycle**

1. ****Filling****: The active segment fills up with transaction data.
2. ****Switching****: On full or checkpoint, PostgreSQL creates a new segment file.
3. ****Archiving or Recycling****: Depending on your configuration, the old segment is either:

* ****Archived**** (saved externally) ****or****
* ****Recycled**** (overwritten in place for future use)

Understanding this lifecycle is crucial for ensuring data durability and recoverability.

## **🗂️ The Archive Status Directory**

Inside pg\_wal, PostgreSQL uses a directory called archive\_status/ to track the archiving progress of each WAL segment. This metadata ensures reliable archiving and recovery.

## **🚩 Files and Their Meanings**

****.ready files****

* Appear automatically when a WAL segment is complete and ready for archiving.
* Example: 00000001000000000000000A.ready
* Signals your archiving process can (and should) copy the corresponding segment.

****.done files****

* Created when the archive\_command successfully copies the associated WAL segment.
* Example: 00000001000000000000000A.done
* Confirms the segment has been safely archived.

## **🛡️ Why It Matters**

* Ensures ****idempotency****: if your archiver falters and retries, .done prevents duplicate copies.
* Prevents ****coordinate chaos****: at any time, you can inspect archive\_status/ to discover what’s pending or completed.
* Failure to archive (e.g., missing permissions) can result in .ready files stacking up—the system warns you to act.

## **🚀 What Is WAL Archiving?**

## **🏗️ Definition in Action**

WAL archiving moves ****entire segments**** from pg\_wal to a secondary location. It’s not about streaming individual changes—rather, it’s segment-based, ensuring consistent units of transaction history.

## **📦 Common Destinations**

* Local disk directories such as /mnt/backup/wal/
* Mounted NFS shares
* Remote servers via SCP or rsync
* Cloud storage: AWS S3, Google Cloud Storage, Azure Blob
* Dedicated WAL archiving tools or services

The important part: the destination must be reliable, durable, and in sync with PostgreSQL’s archiving.

## **🧰 How to Configure WAL Archiving**

Let’s walk through an example postgresql.conf configuration:

# Increase WAL level to support replication and archiving  
wal\_level = replica

# Enable archiving  
archive\_mode = on# Command for copying WAL files  
archive\_command = 'test ! -f /mnt/backup/wal/%f && cp %p /mnt/backup/wal/%f'# Optionally adjust segment size (must be power of two, default 16MB)  
# wal\_segment\_size = 16MB

* ****wal\_level=replica**** ensures sufficient data is logged for standbys and archive.
* ****archive\_mode=on**** turns on the archiving system.
* ****archive\_command****—a shell invocation—tells PostgreSQL how to move segments, using:
* %p: full path to WAL segment
* %f: segment’s filename

## **🔄 Workflow Overview**

1. WAL fills → .ready file appears.
2. archive\_command runs (triggered by PostgreSQL or by an external launcher).
3. On success: WAL is copied → .done file created.
4. Segment may now be recycled.

## **🚨 Troubleshooting Tips**

* ****Permission issues****: ensure PostgreSQL can write to the archive directory.
* ****Failed commands****: logs in pg\_log/ show $archive\_command exit status.
* ****Stuck .ready files****: any .ready older than expected indicates pipeline issues.

## **🔒 Why WAL Archiving Is Essential**

## **1. Point-in-Time Recovery (PITR)**

PITR lets you restore your database to an exact moment:

* Take a full backup (e.g., nightly)
* Archive ongoing WAL segments
* On failure:
* Restore backup
* Apply archived WALs using recovery tools and settings like:

restore\_command = 'cp /mnt/backup/wal/%f %p' recovery\_target\_time = '2025-06-14 08:32:00'

* Replay logs up to a precise timestamp — ideal for recovering accidental deletes.

## **2. Avoiding WAL Recycling Pitfalls**

Without archiving, WAL segments are merely recycled — erasing all older transaction history:

* You can only restore from your ****last full backup****
* Any post-backup transactions become ****irrecoverable****

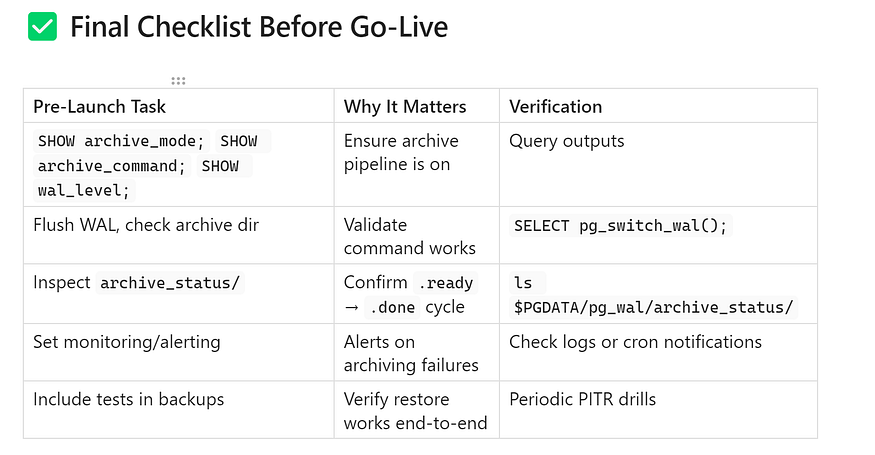
## **3. Support for Disaster Recovery & Replication**

* In replication setups, WAL files are shipped to standby systems.
* Even when not streaming, archived WALs can help rebuild replicas or audit transactions.

## **⚠️ Consequences of Skipping WAL Archiving**

* ****Lost historical logs****: critical changes might vanish during recycling.
* ****Recovery gap****: you may only restore up to your last backup snapshot.
* ****Risk of data loss****: all events between backup and failure are gone.
* ****Lost confidence****: in legal, financial, or compliance scenarios, this gap can be unacceptable.

Press enter or click to view image in full size



## **📝 Summary**

* WAL files live inside $PGDATA/pg\_wal in predictable 16 MB segments.
* Archiving status is managed automatically using .ready and .done markers.
* WAL archiving copies entire segments out to durable storage, enabling PITR and replication support.
* Without it, PostgreSQL recycles segments and you lose post-backup history — risking data gaps.
* Checking config, monitoring status markers, and validating commands before going live is vital.

## **⚙️ Step‑by‑Step: Configuring WAL Archiving in PostgreSQL 17**

Setting up WAL archiving ensures you can recover your database up to any specific point in time — even if you experience data loss. Here’s how to configure it properly in PostgreSQL 17.

## **✅ 1️⃣ Verify Current WAL Settings**

Before making changes, review your existing WAL configuration with:

SELECT name, setting, unit  
FROM pg\_settings  
WHERE name IN (  
 'wal\_level',  
 'wal\_keep\_size',  
 'archive\_mode',  
 'archive\_command',  
 'archive\_timeout'  
);

This query returns key parameters:

* ****wal\_level****: Controls how much detail is logged. For archiving and streaming replication, it must be set to replica or logical.
* ****wal\_keep\_size****: Determines how many megabytes of WAL files PostgreSQL keeps in pg\_wal to help replicas catch up.
* ****archive\_mode****: Indicates whether WAL archiving is currently turned on (on) or off (off).
* ****archive\_command****: Specifies the shell command PostgreSQL uses to copy WAL files to your archive destination.
* ****archive\_timeout****: Defines how often WAL files are forced to switch and get archived, even during low transaction volume.

****Why check before you change?****

* To confirm you’re not unintentionally overwriting a working setup.
* To understand your current disaster recovery posture.
* To spot potential misconfigurations or missing settings.

## **✅ 2️⃣ Create WAL Archive Directory**

You need a safe place where PostgreSQL can store archived WAL files:

mkdir -p $HOME/wal\_archive  
ls -ld $HOME/wal\_archive

Key points:

* The -p flag ensures creation of parent directories if they don’t exist.
* ls -ld confirms its permissions and ownership.
* Ensure the PostgreSQL service user (e.g., postgres) has ****write permission**** in this directory.

### **⚠️ Why this step matters:**

* A misconfigured or non-writable directory will cause archive\_command to fail.
* Archiving failures manifest as .ready files that never turn into .done, silently jeopardizing your disaster recovery capability.
* Setting up the directory correctly at the outset prevents later headaches.

## **✅ Summary: What You’ve Achieved**

Step Purpose Verified WAL settings via SQL Ensured preconditions for archiving are met and understood current configuration Created and verified archive directory Reserved a safe, writeable destination and avoided future permission errors

With these steps complete, you’re ready for configuring postgresql.conf, testing your archive pipeline, and setting up recovery routines in secure storage—laying a strong foundation for full Point-in-Time Recovery (PITR).

## **✅ 3️⃣ Update PostgreSQL Configuration**

With your archive directory ready, the next step is activating WAL archiving and fine‑tuning its behavior. You can make these adjustments in one of two ways:

### **🔧 A. Using**ALTER SYSTEM SET

This method applies changes immediately after the next restart without manually editing files:

ALTER SYSTEM SET archive\_mode = 'on';  
ALTER SYSTEM SET wal\_level = 'replica';  
ALTER SYSTEM SET archive\_timeout= '1h';  
ALTER SYSTEM SET wal\_keep\_size = '100MB';  
-- legacy alternative; modern setups prefer wal\_keep\_size  
ALTER SYSTEM SET wal\_keep\_segments = '10';  
ALTER SYSTEM SET archive\_command =  
 'test ! -f /var/lib/pgsql/wal\_archive/%f && cp %p /var/lib/pgsql/wal\_archive/%f';

****What each setting does:****

* ****archive\_mode = 'on'****  
  Enables the archive system to copy completed WAL segments.
* ****wal\_level = 'replica'****  
  Ensures logs include enough detail for replication and PITR.
* ****archive\_timeout = '1h'****  
  Forces a WAL file switch every hour, even during minimal activity, ensuring regular checkpointing.
* ****wal\_keep\_size = '100MB'****  
  Keeps at least 100 MB of WAL data in pg\_wal to support replicas catching up after brief delays.
* ****wal\_keep\_segments = '10'****  
  (Legacy) Retains ten 16 MB WAL segments as an alternative to wal\_keep\_size. Avoid mixing both parameters.
* ****archive\_command = '…'****  
  The shell command executed each time a WAL segment is ready to archive. PostgreSQL replaces:
* %p with the segment’s full file path
* %f with only the segment’s filename

The sample uses a shell predicate (test ! -f …) to avoid duplicate copies, followed by cp to move segments into your archive directory.

### **🔧 B. Editing**postgresql.conf**Directly**

If you prefer configuration files, open postgresql.conf and adjust or add:

archive\_mode = on  
wal\_level = replica  
archive\_timeout = 1h  
wal\_keep\_size = 100MB  
# wal\_keep\_segments = 10 # optional/legacy

archive\_command = 'test ! -f /var/lib/pgsql/wal\_archive/%f && cp %p /var/lib/pgsql/wal\_archive/%f'

After editing, save the file and proceed with a restart to apply changes.

## **✅ 4️⃣ Restart PostgreSQL to Apply Changes**

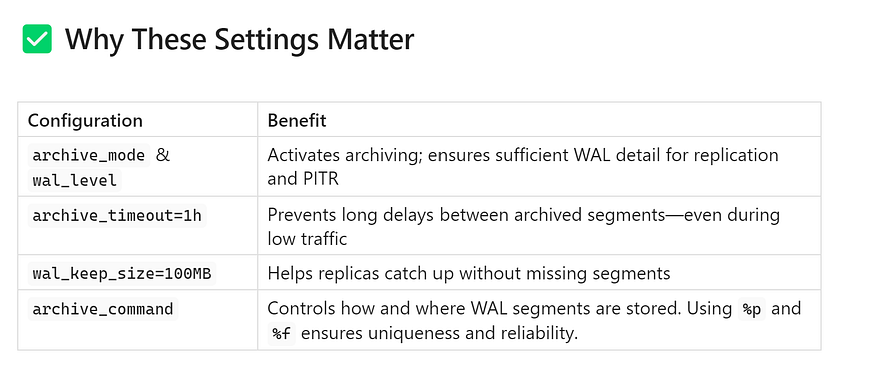
Whether you used SQL commands or edited the config, restart the service to activate archiving:

sudo systemctl restart postgresql-13

After restarting, PostgreSQL will:

* Load the updated archive settings
* Adhere to the specified timeout and keep-size rules
* Begin executing archive\_command whenever WAL segments cycle

Press enter or click to view image in full size



With these configurations in place and the service restarted, PostgreSQL is primed to archive each full WAL segment to your designated folder. The presence of .ready and .done markers in archive\_status/ will indicate progress, while your archive directory grows with time-stamped WAL segments—forming the foundation of a resilient Point-in-Time Recovery (PITR) and replication setup.

You’re now ready to test the pipeline, configure your recovery options, and build confidence in your disaster recovery strategy.

## **✅ 5️⃣ Validate Configuration**

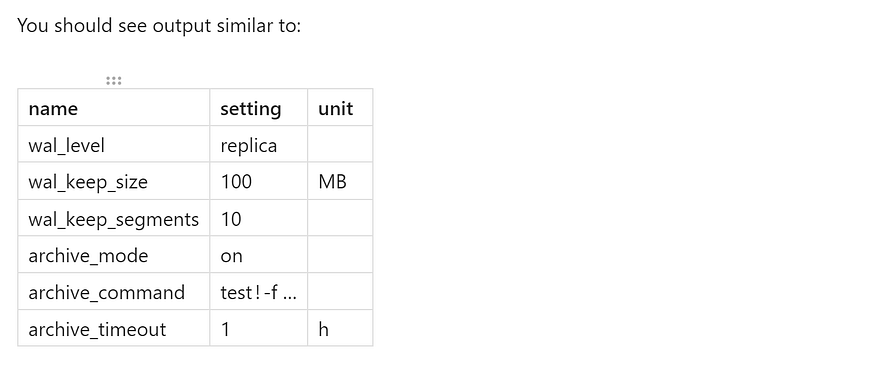
After updating your configuration and restarting PostgreSQL, it’s crucial to confirm that all settings are correctly applied and ready to work.

### **📝 Double-Check Active Settings**

Run this SQL query to verify your WAL-related parameters:

SELECT name, setting, unit  
FROM pg\_settings  
WHERE name IN (  
 'wal\_level',  
 'wal\_keep\_size',  
 'wal\_keep\_segments',  
 'archive\_mode',  
 'archive\_command',  
 'archive\_timeout'  
);

Press enter or click to view image in full size



* ****wal\_level = replica**** — Ensures transaction details are sufficient for archiving, replication, and PITR.
* ****wal\_keep\_size = 100 MB**** ****or**** ****wal\_keep\_segments = 10**** (legacy) — Defines how much WAL data is retained locally.
* ****archive\_mode = on**** — Enables WAL archival.
* ****archive\_command**** — Specifies how WAL segments are copied; verify its path and command syntax.
* ****archive\_timeout = 1 h**** — Forces segment rotation and archival hourly, even if there's low activity.

⚠️ Important: Avoid setting both wal\_keep\_size and wal\_keep\_segments at the same time—choose one to prevent conflicts.

## **🔄 Forcing a WAL Switch (Manual Testing)**

To test if your archive pipeline is working, manually trigger WAL segment rotation:

SELECT pg\_switch\_wal();

Each invocation does the following:

* Closes the current WAL segment.
* Opens a new WAL segment.
* Generates a new .ready status file in archive\_status/.
* Executes archive\_command to copy the segment to your archive directory.

### **🔍 What to do next:**

* Check $PGDATA/pg\_wal/archive\_status/ — You should see a new .ready file appear.
* Inspect your archive directory (/var/lib/pgsql/wal\_archive/) for the corresponding WAL file.
* Confirm a .done file appears in archive\_status/, indicating successful archival.

Run the pg\_switch\_wal() command multiple times to produce several WAL segments and confirm consistent archival behavior.

## **✔️ Summary: What You’ve Achieved**

By completing these steps:

1. ✅ Verified all key WAL and archive settings are correct and active.
2. ✅ Manually triggered WAL switches to validate your archive pipeline works reliably.
3. ✅ Learned to monitor for .ready → .done status transitions to detect any issues.

With this validation in place, you’re one step closer to a fully functioning WAL archive setup — crucial for robust disaster recovery, Point-in-Time Recovery (PITR), and fault-tolerant replication.

## **🛑 How to Stop Archiving (Optional)**

Sometimes you might need to ****temporarily disable WAL archiving**** — for maintenance, testing, or shifting to a different backup strategy. Here’s how to do it safely:

ALTER SYSTEM SET archive\_command = '/bin/false';  
SELECT pg\_reload\_conf();  
SELECT pg\_switch\_wal();

## **🔧 What This Does**

1. ****ALTER SYSTEM SET archive\_command = '/bin/false'****

* archive\_command now uses /bin/false, a command that always exits with failure.
* This means PostgreSQL will no longer archive WAL segments as it proceeds to invoke the command, but nothing will be copied.

2. ****SELECT pg\_reload\_conf();****

* Reloads the configuration without restarting the service, so the new archive\_command takes effect immediately.

3. ****SELECT pg\_switch\_wal();****

* Forces a WAL segment switch. The segment won’t get archived (because archive\_command is ineffective), but it will test the change in behavior.

🎯 This method allows you to ****suspend archiving cleanly****, without permanently disabling archive\_mode or making structural config changes. To resume, restore the original archive\_command via ALTER SYSTEM, reload, and WAL archiving resumes automatically.

## **📊 Monitoring WAL Archiving**

To track how well your WAL archiving is working, PostgreSQL includes a system view:

SELECT \*  
FROM pg\_stat\_archiver;

## **📌 Key Fields to Monitor**

* ****archived\_count**** – Number of WAL segments successfully archived.
* ****failed\_count**** – How many WAL segments failed to archive.
* ****last\_failed\_time**** – Timestamp of the most recent archival failure.

These metrics are crucial for spotting intermittent or persistent archiving issues — especially in production traffic.

## **🚩 Handling Archiving Failures**

When an archive attempt fails — perhaps due to permission issues, storage problems, or a transient network error — PostgreSQL will:

1. Leave the WAL segment’s ****.ready file**** in:

$PGDATA/pg\_wal/archive\_status/

2. Log the failure in log\_directory (commonly pg\_log/).

## **✅ Recommended Failure Response**

* ****Inspect the archives****:

ls $PGDATA/pg\_wal/archive\_status/\*.ready

* These indicate segments still pending archival.

****Investigate the failure****:

* Check postgresql.log or pg\_log/ for recent archive errors.
* Look for ERROR or FATAL messages related to archive\_command

****Fix the root cause****:

* Correct directory permissions, disk space, or network access.
* Ensure destination paths exist and are writable.

****Retry archiving****:

* Manually copy the failed WAL segments to your archive directory.
* Optionally rerun SELECT pg\_switch\_wal(); to simulate fresh segments.

****Verify recovery****:

* Confirm .done markers replace .ready files for each recovered segment.

## **✅ Summary**

* Disabling archiving is easy and reversible — just point archive\_command at a harmless command like /bin/false and reload.
* The view pg\_stat\_archiver provides live statistics on archival success and failure.
* Failed archival attempts leave .ready files and log entries—monitor both, fix issues, and manually handle old segments to maintain a clean archive.

By understanding how to safely disable archiving, monitor its performance, and handle failures, your PostgreSQL system stays resilient and reliable even under changing conditions.

## **⚠️ Common Causes of**.ready**Files Linger­ing**

When you see WAL segments stuck with .ready status in your archive\_status/ directory, it means PostgreSQL has flagged them for archiving—but something has prevented completion. Here's a detailed look into the most common reasons behind this:

## **1. 🛑 Archive Destination Full or Unavailable**

If your archive directory (e.g., /var/lib/pgsql/wal\_archive/) runs out of space or becomes unreachable, PostgreSQL cannot copy WAL segments. The .ready file remains because the segment hasn’t been archived.

****Signs to watch for:****

* cp commands failing due to "No space left on device"
* I/O errors such as “Read-only file system”

## **2. 🧩 Incorrect**archive\_command

A typo in the command, wrong path, or misused variables (%p vs %f) can break the entire pipeline. PostgreSQL will attempt the command and fail silently, leaving .ready files behind.

****Examples:****

* Path typos: /var/lib/psql/wal\_archive/ vs. /var/lib/pgsql/...
* Using absolute instead of relative paths, or vice versa

## **3. 🔐 Insufficient Permissions**

PostgreSQL’s background process must have sufficient rights to read WAL segments and write to the archive directory.

****Common mistakes:****

* Archive path owned by root, not postgres user
* Directory missing write permission (e.g., chmod 700 only allows access to owner)

## **4. 🌐 Network or Storage Failures**

In setups that archive to network storage (NFS, SMB), remote servers, or cloud buckets, temporary outages or latency issues can break archive\_command.

****Potential issues:****

* NFS mount dropped with “Stale file handle”
* SCP or aws s3 cp timing out or erroring out
* Firewall rules blocking remote connections

## **🧐 How to Find the Root Cause**

To pinpoint the issue, inspect PostgreSQL’s logs for detailed error messages:

less /var/lib/pgsql/data/log/postgresql.log

Look for entries around the timestamp when .ready files appeared. You might see lines like:

ERROR: archive command failed with exit status 1  
DETAIL: The failed command was: cp /var/lib/pgsql/data/pg\_wal/000000010000000000000017 /var/lib/pgsql/wal\_archive/000000010000000000000017

These logs often include helpful information:

* Exact failure message (e.g., “permission denied”, “no space left”, “network unreachable”)
* The exact shell command PostgreSQL tried

## **✅ What to Do Next**

1. ****Free up space or repair storage**** if capacity is the issue.
2. ****Fix typos or logic errors**** in your archive\_command.
3. ****Correct permissions**** — use chown postgres:postgres /path … and chmod 755.
4. ****Stabilize remote storage connections****, ensuring mounts are healthy and network is solid.
5. ****Clear stale .ready files**** and let PostgreSQL recreate them after fixing issues.
6. ****Test archiving**** with a manual WAL switch (SELECT pg\_switch\_wal();) and watch for .done markers.

Ensuring your archive pipeline is error-free and resilient is key to a successful disaster recovery strategy. By identifying and fixing issues early — before they escalate — you help safeguard against data loss and ensure your WAL archiving stays reliable.

## **🔄 Why WAL Archiving Is Critical for PITR (Point-in-Time Recovery)**

WAL (Write-Ahead Logging) is one of PostgreSQL’s most powerful features for ensuring data integrity. But its true strength is fully realized only when ****WAL archiving**** is enabled.

Many mistakenly associate WAL archiving only with streaming replication. In reality, ****WAL archiving is the foundation of Point-in-Time Recovery (PITR)**** — a critical capability that lets you restore your database to a specific moment in the past.

## **✅ With WAL Archiving Enabled, You Can:**

1. ****Restore from Any Full Backup****

* Begin recovery using a snapshot from pg\_basebackup or similar tools.

2. ****Replay WAL Up to a Specific Timestamp****

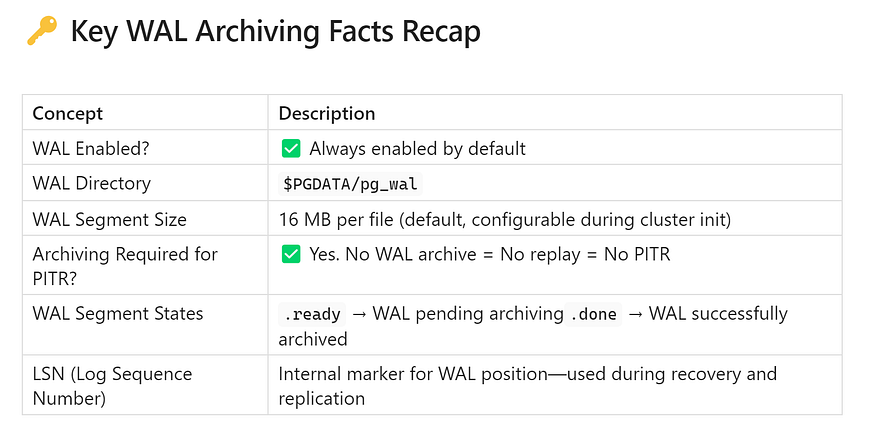
* Use recovery settings like recovery\_target\_time to stop at the exact moment before a failure occurred.

3. ****Recover from Human Errors or Software Bugs****

* Accidentally dropped a table or deleted rows? With PITR, you can roll the database back to a safer point — minutes or even seconds before the issue.

🔒 ****Without WAL archiving, PITR is not possible.****  
Once WAL segments are recycled (overwritten), you lose the ability to replay those changes. That means any user mistake or corruption after your last full backup is permanent and unrecoverable.

Press enter or click to view image in full size



Understanding these fundamentals ensures that you can effectively configure and monitor your archive strategy to support recovery goals.

## **🎯 Conclusion**

WAL archiving isn’t just a technical checkbox — it’s ****PostgreSQL’s secret weapon**** for true database resilience.

Here’s how to stay on top of it:

* ✅ ****Archive all WAL segments**** to a secure, versioned, and durable location.
* ✅ ****Monitor pg\_stat\_archiver**** to verify archive success rates and spot failures early.
* ✅ ****Review logs proactively**** for signs of stuck .ready files or command failures.
* ✅ ****Practice PITR recovery drills**** in staging environments — don’t wait for a real disaster to test your process.

🚀 With WAL archiving, you gain peace of mind, operational flexibility, and robust protection against even the most unexpected incidents. Treat it as an essential part of your PostgreSQL toolkit.