# **Mastering PostgreSQL 17: Continuous Archiving & Point-In-Time Recovery (PITR) — A Complete Hands-On Guide**

[](https://medium.com/@jramcloud1?source=post_page---byline--c880a9d50a90---------------------------------------)

[Jeyaram Ayyalusamy](https://medium.com/@jramcloud1?source=post_page---byline--c880a9d50a90---------------------------------------)

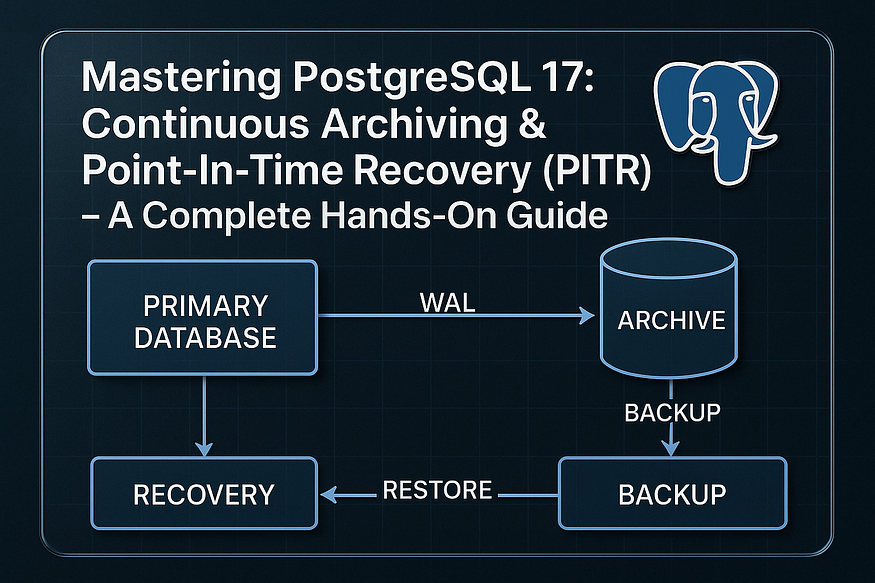
Following

13 min read

·

Jun 21, 2025

Press enter or click to view image in full size



PostgreSQL is widely admired not just for its power and performance but also for its advanced backup and recovery capabilities. Among these, ****Continuous Archiving**** and ****Point-In-Time Recovery (PITR)**** stand out as lifesavers for DBAs, allowing recovery from data loss, corruption, or accidental changes.

## **🔍 Understanding the Foundation: WAL in PostgreSQL**

At the core of PostgreSQL’s resilience — whether facing crashes, human error, or system corruption — is ****Write-Ahead Logging (WAL)****. It’s the bedrock upon which reliable backups, replication, and advanced recovery mechanisms stand. Let’s dig into how WAL works and why it’s indispensable.

## **🛡️ What Is WAL?**

Write-Ahead Logging is a mechanism designed to ensure ****atomicity****, ****consistency****, ****durability****, and ****recoverability**** within PostgreSQL:

* ****✅ All database changes are first written to WAL before being applied to data files.****  
  Whether you’re inserting a row, updating an index, or deleting records, PostgreSQL logs the intended change to a WAL record *before* writing to the main data files. This ordering is critical — it means even incomplete transactions can be safely recovered after a crash.
* ****✅ Guarantees data integrity and allows crash recovery.****  
  In the event of an unexpected shutdown, PostgreSQL uses WAL on startup to replay any logged but uncommitted changes, bringing the database back to a consistent state without data corruption.

## **📁 Where Are WAL Files Stored?**

* ****WAL files live in the pg\_wal/ directory**** inside your database cluster’s root.
* ****Each WAL file is fixed at 16 MB by default****, though their number scales to match your database activity.
* Filenames encode the timeline and sequence, such as 0000000100000000000000A1, ensuring precise tracking of changes over time.

## **🚀 Beyond Crash Recovery**

While crash recovery is the original purpose, WAL empowers a suite of advanced PostgreSQL capabilities:

1. ****🔄 Point-in-Time Recovery (PITR)****  
   By archiving WAL segments, you can restore a full backup and then ****replay transactions up to any target timestamp**** — down to the second. This enables recovery from accidental deletions or data anomalies.
2. ****🌐 Streaming Replication****  
   WAL segments or logs aren’t just local — they can be shipped to standby servers in real-time, keeping replicas closely synchronized with the primary.
3. ****💾 Continuous Archiving****  
   WAL files are written in high-frequency chunks that can be periodically offloaded to durable storage. This supports long-term backup retention, legal compliance, and geo-redundancy.

## **🧠 Why WAL Is Fundamental to PostgreSQL**

Benefit Description ****Crash recovery**** Restores data after an unexpected shutdown. ****Transactional consistency**** Ensures every committed change is preserved in the logs. ****PITR capability**** Enables accurate restoration to any chosen moment. ****Replication support**** Facilitates real-time sync between primary and replicas. ****Archive and audit readiness**** Volunteers WAL files for long-term storage and forensic analysis.

## **🔧 Step 1: Setting Up PostgreSQL 17 on AWS EC2 (RHEL 8)**

Before diving into advanced PostgreSQL features like WAL archiving, you need a solid foundation. Here’s how to spin up and prepare two EC2 instances running RHEL 8:

## **🖥 1. Launch Two EC2 Instances**

1. In the AWS Console, go to ****EC2 → Instances → Launch instances****.
2. Choose a ****Red Hat Enterprise Linux 8 (RHEL 8)**** AMI.
3. Select instance types (e.g., t3.medium for moderate workloads).
4. Review key security thresholds:

* Ensure ****SSH (port 22)**** is open for your IP.
* Open ****PostgreSQL (port 5432)**** to allow traffic between your two nodes.

This setup ensures both your primary and standby servers can communicate securely.

## **🗝️ 2. Convert PEM to PPK (Windows Only)**

If you’re using ****PuTTY on Windows****, you’ll need to convert the key file:

1. Download PuTTYgen.
2. Load your your-key.pem.
3. Save the private key as your-key.ppk.

You can now log in with PuTTY rather than directly using the PEM file.

## **🔐 3. Connect via SSH or PuTTY**

Use the terminal (Linux/macOS) or PuTTY (Windows) to access your instance:

ssh -i "your-key.pem" ec2-user@<public-ip>

Once connected, switch to root privileges:

sudo -i

You’re now ready to configure the system.

## **🔒 4. Disable SELinux (Optional for Labs)**

SELinux is ****enforcing**** by default on RHEL and can restrict PostgreSQL behavior during setup. For lab environments, it’s simpler to disable or switch to permissive mode:

### **A. Disable Completely:**

Edit /etc/selinux/config, change:

SELINUX=enforcing

to:

SELINUX=disabled

Then reboot:

sudo reboot

On RHEL 8, this stops SELinux enforcement after reboot ([devopscube.com](https://devopscube.com/install-configure-postgresql-amazon-linux/?utm_source=chatgpt.com" \t "https://medium.com/@jramcloud1/_blank), [proventa.de](https://proventa.de/en/securing-our-postgresql-database-with-selinux/?utm_source=chatgpt.com" \t "https://medium.com/@jramcloud1/_blank), [kafana.dev](https://kafana.dev/docs/1.2.Linux/How2s/dis-selinxu-rehel8/?utm_source=chatgpt.com" \t "https://medium.com/@jramcloud1/_blank)).

### **B. Or Switch to Permissive Mode:**

To retain logging but avoid blockages, set:

SELINUX=permissive

And reboot. This logs denied actions without preventing them — ideal for troubleshooting ([severalnines.com](https://severalnines.com/blog/how-configure-selinux-postgresql-and-timescaledb/?utm_source=chatgpt.com" \t "https://medium.com/@jramcloud1/_blank), [docs.aws.amazon.com](https://docs.aws.amazon.com/linux/al2023/ug/disable-option-selinux.html?utm_source=chatgpt.com" \t "https://medium.com/@jramcloud1/_blank)).

## **✅ Why This Setup Is Important**

* ****Dual instances**** prepare you for replication and failover testing.
* ****Security groups**** ensure essential network flows are permitted.
* ****Disabling or softening SELinux**** removes early roadblocks during initial setup.
* ****SSH access**** provides robust administrative control from day one.

With your EC2 instances ready, SSH working, and SELinux in permissive mode, you’re set for the next stages — installing PostgreSQL 17, configuring replication, and enabling WAL archiving. Let me know if you’d like help with installations, firewall tuning, or database initialization next!

## **🛠 Step 2: Installing and Configuring PostgreSQL 17 on RHEL 8**

Get your PostgreSQL environment ready by setting up version 17 from the official PostgreSQL Yum repository. This ensures you’re working with the latest supported features and patches.

## **1️⃣ Add the PostgreSQL Yum Repository**

Fetch and enable the PostgreSQL repository package:

sudo dnf install -y \  
 https://download.postgresql.org/pub/repos/yum/reporpms/EL-8-x86\_64/pgdg-redhat-repo-latest.noarch.rpm

This installs the pgdg-redhat-repo package, unlocking Postgres 17 and other versions for installation ([jbitc.de](https://www.jbitc.de/2025/01/20/postgresql-and-pgadmin-installation/?utm_source=chatgpt.com" \t "https://medium.com/@jramcloud1/_blank)).

## **2️⃣ Disable the Default PostgreSQL Module**

RHEL ships its own PostgreSQL versions via modules, which might conflict. Disable the default:

sudo dnf -qy module disable postgresql

This ensures a clean, conflict-free install of Postgres 17.

## **3️⃣ Install PostgreSQL 17 Server Packages**

Install the PostgreSQL 17 server and client packages:

sudo dnf install -y postgresql17-server

This includes the core server, client tools, and essential libraries ([postgresql.org](https://www.postgresql.org/download/linux/redhat/?utm_source=chatgpt.com" \t "https://medium.com/@jramcloud1/_blank)).

## **4️⃣ Initialize the Database Cluster**

Set up the database cluster with initdb:

sudo /usr/pgsql-17/bin/postgresql-17-setup initdb

This initializes the data directory and default configs.

## **5️⃣ Enable and Start PostgreSQL Service**

Make PostgreSQL start on boot and launch it now:

sudo systemctl enable --now postgresql-17

This registers and starts the postgresql-17 service automatically at boot ([jbitc.de](https://www.jbitc.de/2025/01/20/postgresql-and-pgadmin-installation/?utm_source=chatgpt.com" \t "https://medium.com/@jramcloud1/_blank)).

## **🌐 Enable Remote Access**

To allow connections from other hosts, update your configuration:

### **A. Update**postgresql.conf

Enable PostgreSQL to listen on all network interfaces:

sudo sed -i "s/^#listen\_addresses = 'localhost'/listen\_addresses = '\*'/g" /var/lib/pgsql/17/data/postgresql.conf

### **B. Adjust**pg\_hba.conf

Allow password authentication from any IPv4 address:

cat <<EOF | sudo tee -a /var/lib/pgsql/17/data/pg\_hba.conf  
# Allow connections from all IPv4 users  
host all all 0.0.0.0/0 scram-sha-256  
EOF

### **C. Reload Configuration**

Apply changes without downtime:

sudo -u postgres pg\_ctl reload

## **🔑 Set a Strong Password for the**postgres**User**

Secure the default superuser account:

sudo -u postgres psql -c "ALTER USER postgres PASSWORD 'your-password';"

## **🔁 Restart PostgreSQL to Apply All Changes**

Finish up with a clean restart:

sudo systemctl restart postgresql-17

This ensures that listen settings and authentication rules take effect.

## **✅ What You’ve Achieved**

* Installed PostgreSQL 17 via the official Postgres Yum repo.
* Initialized your database cluster.
* Configured the service to start at boot.
* Opened the server for remote connections using secure, password-based authentication.

## **📦 Step 2: Configuring Continuous Archiving**

Now that PostgreSQL 13 is up and running, let’s enable ****continuous WAL archiving****. This process ensures that every completed WAL segment is copied to a safe archive location — paving the way for full Point-in-Time Recovery (PITR).

## **🗄 1️⃣ Create Your Archive Directory**

Pick a durable, accessible location for archived WAL segments:

sudo mkdir -p /var/lib/pgsql/17/data/pg17\_archive  
sudo chown postgres:postgres /var/lib/pgsql/17/data/pg17\_archive  
sudo chmod 700 /var/lib/pgsql/17/data/pg17\_archive

* The -p flag ensures all parent folders are created.
* Ownership and permissions are set so that only the postgres user can access it securely.
* Prevents unauthorized access and reduces risk of tampering.

## **📝 2️⃣ Modify**postgresql.conf

Open your primary configuration file (/var/lib/pgsql/17/data/postgresql.conf) and update these settings:

archive\_mode = on  
wal\_level = replica  
archive\_timeout = 300  
archive\_command = 'test ! -f /var/lib/pgsql/17/data/pg17\_archive/%f && cp %p /var/lib/pgsql/17/data/pg17\_archive/%f'

* ****archive\_mode = on****  
  Enables WAL segment archiving.
* ****wal\_level = replica****  
  Ensures WAL contains enough data for archiving and potential replication.
* ****archive\_timeout = 300****  
  Forces segment switch every 5 minutes, ensuring WAL is archived even during quiet periods.
* ****archive\_command = 'test … && cp …'****
* %p: full file path of the WAL segment being archived.
* %f: filename only.
* test ! -f: skips copying if the file already exists, avoiding duplicates.
* cp: copies the file into your designated archive directory.

This setup ensures that every WAL segment is backed up consistently and efficiently.

## **🔄 3️⃣ Restart PostgreSQL**

Apply the new settings by restarting the service:

sudo systemctl restart postgresql-17

This reloads your postgresql.conf and activates continuous archiving.

## **✅ 4️⃣ Verify Your Configuration**

To confirm archiving is properly configured, run the following within psql:

SHOW archive\_mode; -- should output 'on'  
SHOW archive\_command; -- should show your configured command  
SHOW wal\_level; -- should output 'replica'

Seeing these expected settings reassures you that PostgreSQL is set up for archiving.

## **🧪 5️⃣ Force a WAL Switch for Testing**

Trigger an immediate WAL segment boundary:

SELECT pg\_switch\_wal();

This closes the current WAL file, spawns .ready and .done flags in pg\_wal/archive\_status/, and invokes archive\_command to copy the completed segment to your archive directory.

## **🔎 6️⃣ Verify the Archived Files**

On the server, check that the WAL files are now archived:

ls /var/lib/pgsql/17/data/pg17\_archive

You should see one or more 16 MB WAL segment files — named like 0000000100000000000000A1—indicating successful archival.

## **🛡️ Why It Matters**

* Establishes a ****reliable backup pipeline**** for every transaction since your last base backup.
* Ensures WAL segments won’t be recycled before being archived — critical for recovery.
* Provides a concrete way to test and confirm that your archiving framework is functioning in real time.

## **🔐 Step 3: Take a Physical Backup for PITR (Point-in-Time Recovery)**

With WAL archiving in place, the next essential step toward full database resilience is taking a ****physical backup****. This backup, combined with archived WAL files, enables recovery to any point in time — ensuring minimal data loss.

## **📍 1️⃣ Confirm Your Data Directory**

Before running a backup, you need to know exactly where PostgreSQL stores the active data files.

Connect to your server using psql and check:

SHOW data\_directory;

This query returns the full path to your active data directory — usually something like /var/lib/pgsql/17/data. Knowing this location is crucial, as it identifies the baseline you’ll back up.

## **📁 2️⃣ Create a Backup Destination Directory**

You’ll want to store your base backup in a safe and separate location. Use:

sudo mkdir -p /var/lib/pgsql/17/data/pg17\_backup  
sudo chown postgres:postgres /var/lib/pgsql/17/data/pg17\_backup  
sudo chmod 700 /var/lib/pgsql/17/data/pg17\_backup

* The -p flag ensures the parent directories are created.
* Ownership is set to postgres to grant proper permissions during the backup.
* Permission 700 ensures only the postgres user can access or modify the backup.

## **💾 3️⃣ Run the Physical Backup with**pg\_basebackup

The easiest and most reliable method to create a base backup in PostgreSQL is using pg\_basebackup. It works seamlessly with your ongoing WAL archiving setup:

Switch to the postgres user and initiate the backup:

sudo -u postgres pg\_basebackup -D /var/lib/pgsql/17/data/pg17\_backup

By default, pg\_basebackup:

* Creates a consistent snapshot of your entire data directory.
* Includes a copy of the control files needed for recovery.
* Captures essential WAL information to enable seamless PITR alongside your archived WAL segments.

## **🛡️ Why This Matters**

* ****✔️ Completes the PITR foundation****: A physical backup plus archived WAL logs allows you to recover to any point between the backup’s creation and the present.
* ****✔️ Keeps your data consistent****: pg\_basebackup ensures an online, crash-consistent backup, even while your database remains operational.
* ****✔️ Prepares for disaster scenarios****: With this snapshot, you’re protected from full-server failures, data corruption, and accidental deletions — armed with the ability to restore up to seconds before the event.

## **✅ Next Steps**

Once your base backup finishes successfully:

1. ****Store the backup**** — ideally in a secure, off-site location or connected object storage.
2. ****Archive the directory**** — tar and compress if needed before moving to long-term storage.
3. ****Test a restore scenario**** — combine this backup with your WAL archive to validate real-world recovery.
4. ****Schedule regular backups**** — using cron or backup tools to automate full-BACKUP + WAL syncing.

## **💣 Step 4: Simulate Failure & Perform PITR (Point-in-Time Recovery)**

With your base backup and WAL archive ready, it’s time to simulate a real-world worst-case scenario and put your recovery plan to the test. This step verifies that you can restore your database to a specific point using PITR.

## **🛑 1️⃣ Stop PostgreSQL (Simulate Service or Disk Loss)**

First, shut down the PostgreSQL server to mimic a crash or catastrophic system failure:

sudo systemctl stop postgresql-17

This halts all database activity, ensuring we start recovery from a clean — but “failed” — state.

## **📂 2️⃣ Rename the Old Data Directory**

To prevent any accidental conflict or reuse, rename the current data directory:

sudo mv /var/lib/pgsql/17/data /var/lib/pgsql/17/data\_old

This preserves the pre-failure database files for reference while we reconstruct a fresh environment.

## **🆕 3️⃣ Restore from the Base Backup**

Create a clean data directory and copy the contents of your base backup into it:

sudo mkdir -p /var/lib/pgsql/17/data  
sudo cp -a /var/lib/pgsql/17/data\_old/pg17\_backup/. /var/lib/pgsql/17/data  
sudo chown -R postgres:postgres /var/lib/pgsql/17/data  
sudo chmod 700 /var/lib/pgsql/17/data

* ****cp -a**** preserves file permissions, ownership, and timestamps.
* Resetting ****ownership**** and ****permissions**** prevents access issues during startup.

## **🗑 4️⃣ Clean Existing WAL (Prevent Conflicts)**

Clear the WAL directory from your new data environment to avoid mixing segments from different timelines:

sudo rm -rf /var/lib/pgsql/17/data/pg\_wal/\*

This ensures a fresh start using only your archived WAL segments for recovery.

## **🔁 5️⃣ Restore Archived WAL Files**

To replay changes beyond the base backup, move the archived WAL segments into your new WAL directory:

sudo cp -a /var/lib/pgsql/17/data\_old/pg\_wal/\* /var/lib/pgsql/17/data/pg\_wal/  
sudo cp -a /var/lib/pgsql/17/data\_old/pg17\_archive/\* /var/lib/pgsql/17/data/pg\_wal/

These WAL files contain the transaction history needed to bring your backup up to the failure point.

## **🛠 What Happens Next (Not in Original Content)**

Once WAL segments are in place, PostgreSQL can replay them on startup, using a configured recovery.conf or postgresql.auto.conf recovery settings. You’d typically include:

restore\_command = 'cp /var/lib/pgsql/17/data/pg\_wal/%f %p'  
recovery\_target\_time = '<desired-timestamp>'

This tells PostgreSQL ****how to find the WAL files**** and ****when to stop replaying****, enabling a precise PITR.

## **✅ In Summary**

By performing this simulated recovery procedure, you validate:

* You can ****reconstruct your data directory**** from a base backup.
* Your ****WAL archive holds the complete transaction log**** required for PITR.
* PostgreSQL can ****replay those logs and recover to your target point in time****.

Successfully completing this simulation confirms that, in a production failure, you’ll be able to restore your database with confidence.

## **⚙ Step 5: Configure PITR Settings (Point-in-Time Recovery)**

Now that you’ve restored your base backup and positioned your archived WAL files, it’s time to configure PostgreSQL to actually ****perform the recovery****. This involves setting up the instructions PostgreSQL needs to replay your archived WAL log files and optionally stop at a specific recovery target.

## **📝 1️⃣ Add**restore\_command**in**postgresql.conf

Open your ****postgresql.conf**** file in the newly restored data directory (for example, /var/lib/pgsql/17/data/postgresql.conf) and add:

restore\_command = 'cp /var/lib/pgsql/17/data/pg13\_archive/%f %p'

* ****cp**** copies archived WAL segments into the active WAL directory during recovery.
* ****%f**** is the WAL segment filename.
* ****%p**** is the path PostgreSQL wants to restore it to.

This command tells PostgreSQL ****where and how to fetch archived WAL files**** during recovery, enabling it to replay the transaction history.

## **🏁 2️⃣ Create a Recovery Signal File**

To trigger recovery mode on startup, PostgreSQL in versions 12+ relies on a special signal file:

sudo touch /var/lib/pgsql/17/data/recovery.signal

* The presence of ****recovery.signal**** tells PostgreSQL to enter ****recovery mode**** instead of normal operation.
* It then automatically activates the restore\_command process, reading archived WAL segments and replaying them in sequence.

## **🔥 3️⃣ (Optional) Define a Recovery Target Time**

If your goal is to restore to a specific past moment — say, just before an accidental deletion — you can add a ****target timestamp****:

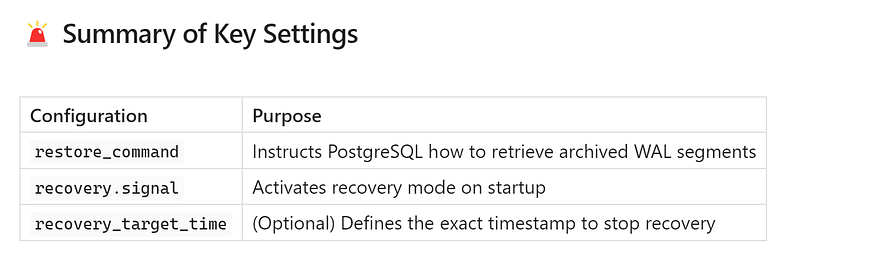
recovery\_target\_time = '2025-06-14 15:30:00'

PostgreSQL then:

1. Restores the base backup.
2. Replays WAL segments up to the specified timestamp.
3. Stops exactly at that moment, applying only committed transactions up to your target.

This precise stopping ability is the core of ****Point-in-Time Recovery****, giving you granular control over restoration.

Press enter or click to view image in full size



## **🎯 Final Steps**

1. Double-check that the restore\_command path and filenames are correct.
2. Ensure recovery.signal is placed in the ****active data directory****.
3. Optionally set a recovery target time if you’re tuning the restore point.
4. Restart or start PostgreSQL (sudo systemctl start postgresql-17) and monitor logs (tail -f pg\_wal/postgresql.log) to watch recovery progress.

With these settings in place, PostgreSQL will seamlessly replay archived WAL segments and restore your database to the exact state you need — either up to the latest available log or to a precise historical moment.

## **🚀 Step 6: Start PostgreSQL in Recovery Mode**

Now that you’ve configured the recovery settings — including restore\_command, recovery.signal, and optionally a recovery target timestamp—it's time to ****start PostgreSQL in recovery mode**** and verify that everything works as expected.

## **▶️ 1️⃣ Start the PostgreSQL Service**

Use systemd to launch PostgreSQL, which will immediately detect the recovery.signal file and enter recovery mode:

sudo systemctl start postgresql-17

Upon startup, PostgreSQL processes the archived WAL segments according to your restore\_command and begins replaying transactions, either up to the latest or stopping at your specified recovery\_target\_time.

## **🔍 2️⃣ Verify Recovery Status**

To confirm that PostgreSQL is running in recovery mode without errors, check the service status:

sudo systemctl status postgresql-17

Look for key hints in the output:

* Indicators such as ****“in recovery”****, ****“recovering”****, or ****“recovery in progress”****
* Absence of error messages related to missing WAL files or recovery failures

If recovery encounters a WAL segment gap or fails to locate the correct logs, it will log errors and may refuse to start until those issues are addressed.

## **📊 3️⃣ Validate the Recovered Data**

Once the service indicates recovery is ongoing or complete, verify that your database content is present and accurate:

psql -U postgres

Then run:

\l

This will list all databases and show that the cluster is online and accessible.

Next, inspect application tables:

SELECT \* FROM your\_tables;

If you specified a recovery\_target\_time, confirm that the data reflects the expected state at that moment. For example:

* Rows added after the target time should be absent.
* Rows present before should remain intact.

## **🧩 Why These Steps Matter**

1. ****Service Start with recovery.signal**** – Signals PostgreSQL to recover not initialize.
2. ****Status Check**** — Ensures there are no hidden errors that could compromise data integrity.
3. ****Data Validation**** — Confirms your recovery target was hit and that your data matches expectations.

## **✅ What to Do Next**

* ****Monitor logs****: Continue monitoring postgresql-13 logs (journalctl -u postgresql-13) for full replay activity and potential errors.
* ****Complete recovery****: Once all WAL segments are replayed (or recovery target reached), PostgreSQL will transition to normal operation.
* ****Remove recovery.signal****: After successful recovery and verification, rename or delete the recovery.signal file if it remains—though PostgreSQL may do so automatically, depending on your version.
* ****Conduct testing****: Run query tests or application-level checks to verify that all functionality is restored.

By following these steps, you ensure your system is truly recoverable — and that your database reflects the precise moment you intended. This completes your hands-on demonstration of a full WAL-based recovery process.