POSTGRES CONFERENCE 2026

The Elephant in the Room

Tackling PostgreSQL Major Version Upgrades Head-On

A Complete Guide to Planning, Testing, and Executing PostgreSQL Upgrades

Shaily Porwal
Sr. Technical Account Manager
Amazon Web Services (AWS)

Vivek Singh
Principal Database Specialist – PostgreSQL
Amazon Web Services (AWS)

About the Speaker - Vivek Singh

20+ Years PostgreSQL Expertise

Principal Database Specialist at AWS

Trained 500+ AWS Professionals

Top 1.3% of 1,300+ DB Experts

AWS & Technical Publications

- Improve PostgreSQL performance using pgstattuple extension
- Optimize the cost of Amazon RDS snapshots
- Best practices for migrating PostgreSQL to Amazon RDS & Aurora
- Deep dive on Aurora & RDS for PostgreSQL architecture
- Best practices for cross-Region read replicas
- AWS Whitepaper: Optimizing PostgreSQL on EC2 with EBS
- DZone: MERGE Command in PostgreSQL 15
- IoT-enabled smart robotic baggage monitoring system

Author & Thought Leader

Everything shown today is based on real implementations and proven results.

About the Speaker - Shaily Porwal

15+ Years PostgreSQL Expertise

Senior Technical Account Manager at AWS

Trained 400+ AWS Professionals

About me

- 4x AWS Certified, including AWS Certified Solutions Architect - Professional
- Relational Database Service Core Subject Matter Expert and Relational Database Service PostgreSQL Subject Matter Expert
- Database Technical Field Community member with deep expertise in Relational Database Service and Aurora PostgreSQL, mentoring Technical Account Managers in database best practices
- Oracle Expert with extensive experience in Oracle-to-PostgreSQL migrations, having developed migration frameworks used across AWS internal teams
- Regular speaker and facilitator at enterprise customer workshops, webinars, and conference talks, including the DFW Relational Database Service/Aurora Operational Excellence Workshop with 82 participants and 56 enterprise customers
- Conducted multiple AWS internal enablement sessions on database services and AI/ML
- Passionate about bridging traditional database administration with modern AI-driven automation

Session Roadmap

01

Versioning & EOL Cycles

Understanding PostgreSQL release lifecycle

02

Real Risks of Staying Behind

Security, performance, and compliance impact

03

Upgrade Strategies

pg_upgrade, dump/restore, logical replication, Blue/Green

04

Testing Frameworks

Building confidence through systematic testing

05

Minimizing Downtime

Production upgrade execution and rollback strategies

06

Lessons Learned

Real-world case studies and common pitfalls

PostgreSQL Versioning Fundamentals

Version Format

16 MAJOR . **4** MINOR

Major Versions: Annual releases with new features, may break compatibility

Minor Versions: Bug fixes and security patches, always backward compatible

Support Lifecycle: 5 years from initial release, then End of Life

End of Life Timeline

PG 12	Nov 2024	EOL
PG 13	Nov 2025	EOL
PG 14	Nov 2026	URGENT
PG 15	Nov 2027	Active
PG 16	Nov 2028	Active
PG 17	Nov 2029	Current

What happens at EOL: No more security patches, bug fixes, or community support. Your database is exposed.

The Upgrade Challenge

Why Organizations Postpone Major Version Upgrades

Complexity Fear

Schema changes, extension compatibility, application rewrites – the unknowns paralyze action

Downtime Anxiety

Production systems can't afford extended outages; business continuity is non-negotiable

Risk Aversion

Data corruption, performance regression, rollback uncertainty – the stakes are too high

Resource Constraints

Limited DBA bandwidth, competing priorities, and lack of testing infrastructure

⚠ **PostgreSQL 14 End of Life: November 2026** – *The clock is ticking. Start planning now.*

Major vs Minor Upgrades

Know the Difference Before You Plan

Dimension	Major Upgrade	Minor Upgrade
Data Migration	Required – new data format	Not needed – binary compatible
Downtime	Minutes to hours (strategy-dependent)	Seconds to minutes (restart)
Compatibility	May break applications & extensions	Fully backward compatible
Protocol Changes	Possible wire protocol changes	No protocol changes
Extension Support	Must verify all extensions	Extensions remain compatible
System Resources	Requires 2x disk (temp)	Minimal additional resources
Planning Effort	Weeks of testing & validation	Standard maintenance window
Rollback	Complex – requires backup restore	Simple – reinstall old binaries

Key Takeaway: Major upgrades require careful planning; minor upgrades are routine maintenance operations.

The Real Risks of Not Upgrading

Performance Degradation

- Suboptimal query planning algorithms
- Resource utilization inefficiencies
- Scalability constraints on modern hardware
- Missing parallel query optimizations

Security Vulnerabilities

- Unpatched CVEs with known exploits
- Access control weaknesses
- Missing authentication improvements
- No TLS/SSL security enhancements

Compliance Violations

- GDPR data protection requirements
- HIPAA healthcare data standards
- PCI DSS payment card mandates
- SOX audit trail requirements

Technical Debt

- Increased maintenance overhead
- Loss of extension compatibility
- Growing skills gap for team
- Vendor support limitations

The Business Impact

When Technical Debt Becomes Business Risk

60%

of breaches exploit known unpatched CVEs

3-5x

cost multiplier for delayed upgrades

47%

of orgs fail compliance due to outdated software

Contractual Obligations

SLA violations and audit failures from unsupported software versions

Competitive Disadvantage

Missing new features: logical replication improvements, JSON enhancements, query performance

Team Impact

Recruiting challenges, skill atrophy, and engineer burnout from legacy maintenance

Compounding Cost

Each delayed version makes the next upgrade exponentially harder and riskier

Pre-Upgrade Planning Checklist

The Foundation for a Successful Upgrade

Version Compatibility Assessment

Review release notes, deprecated features, and breaking changes between current and target versions

Database Size & Complexity Evaluation

Catalog database size, number of tables, indexes, custom types, and partitioning schemes

Dependency Audit

Map all extensions (PostGIS, pg_stat_statements, etc.), client drivers, ORMs, and connection poolers

Performance Baseline Establishment

Capture query performance metrics, pg_stat_statements data, and I/O benchmarks pre-upgrade

Test Environment Setup

Mirror production: same OS version, PostgreSQL configuration, data volume, and workload patterns

Rollback Strategy Design

Define rollback triggers, test restore procedures, and ensure backup validity before proceeding

Pre-Upgrade Health Checks

Critical Validations Before You Pull the Trigger

Storage Capacity

Verify 2x disk space for pg_upgrade copy mode. Check tablespace locations. Avoid the "storage full" disaster during upgrade.

Configuration Review

Compare postgresql.conf between versions. Review changed/removed parameters. Migrate custom GUC settings.

Extension Compatibility

Validate all extensions against target version. Check PostGIS, pg_partman, timescaledb, pg_cron compatibility.

Replication Health

Ensure zero replication lag. Verify standby servers are current. Plan for replica recreation if needed.

Connection Management

Plan PgBouncer/connection pooler pause. Pre-configure max_connections for new version. Test failover paths.

Backup Validation

Take and verify fresh backup. Test point-in-time recovery. Ensure backup tool compatibility with new version.

Aurora/RDS PostgreSQL Pre-Upgrade Health Checks

- RDS/Aurora only runs pre-upgrade checks *during* the actual upgrade
- Failed upgrades = extended downtime, rollback complexity, wasted maintenance windows
- **Solution:** Aurora/RDS PostgreSQL Pre-Upgrade Check Tool — open-source bash tool to identify blockers *before* your upgrade window
- Generates an HTML report in 2–3 minutes with findings + recommended fixes + AWS doc references
- Supports: Aurora PostgreSQL, RDS PostgreSQL, IAM authentication, cross-region, Aurora Serverless v2, custom ports
- No customer data collected — queries system catalogs only, minimal performance impact

Aurora/RDS PostgreSQL Pre-Upgrade Health Checks

- **What It Checks & How to Run**
  - **20+ Checks Including:**
  - Target PostgreSQL version compatibility
  - Unsupported DB instance classes
  - Open prepared transactions
  - Unsupported reg* data types & unknown data types
  - Logical replication slots
  - Storage issues & incompatible parameters
  - Extension compatibility (PostGIS, pg_cron, TimescaleDB, pgvector, etc.)
  - Read replica upgrade concerns
  - Views dependency problems, GIST indexes, ICU Collations
  - Reserved keywords, Template1 settings, maintenance tasks in progress
- **How to Run:**
  - Requires: EC2 with AWS CLI + psql + network access to RDS/Aurora
  - chmod +x pg_upgrade_pre_check.sh → ./pg_upgrade_pre_check.sh
  - Provide: endpoint, DB name, port, username, target version
  - Output: HTML report with findings + fix recommendations

https://github.com/aws-samples/sample-AuroraRDSPostgresUpgradePrecheck

Aurora/RDS PostgreSQL Pre-Upgrade Health Checks

PostgreSQL Pre-upgrade Check Report For ISAU

Author: Vivek Singh, Principal Database Specialist - PostgreSQL, Amazon Web Servies | Version V01

Issue found. No issue found. Requires manual analysis.

Pre-upgrade check report for:
Postgres Endpoint URL: pg15.cjmh5xirlcl7.us-west-2.rds.amazonaws.com
Current Postgres version: 15.12
Target Postgres version: 17
Date of report: 10-07-2025

1. Check for target Postgres version:
17 is one of the target versions for current version of postgres 15.12. No issue found.

2. Check for unsupported DB instance classes:
postgres DB instance class db.m6g.12xlarge is supported for target postgres version 17. No issue found.

3. Check for open prepared transactions:
Prepared transactions that are open on the database might lead to upgrade failure. Be sure to commit or roll back all open prepared transactions before starting an upgrade. No uncommitted prepared transactions found.

4. Check for unsupported reg* data types:
The pg_upgrade utility doesn't support upgrading databases that include table columns using the reg* OID-referencing system data types. Remove all uses of reg* data types, except for regclass, regrole, and regtype, before attempt

5. Check for logical replication slots:
No logical replication slots found. An upgrade can't occur if your instance has any logical replication slots. Logical replication slots are typically used for AWS Database Migration Service (AMS DMS) migration.

6. Check for storage issues:
Total size of all databases in RDS instance pg15 instance is 250GB. Current FreeStorageSpace is 40GB. Make sure to have 15%-20% free storage to avoid upgrade failures.

7. Check for 'Incompatible Parameter' error:
Work_mem is set at default value 4MB. Higher value of work_mem can cause 'Incompatible Parameters' issue and might fail upgrade. No issues found.
Shared_buffers is 76 GB, 39% of total instance memory. The default value of Shared_buffers for RDS Postgres is set at ~24%. If the value is modified to higher value, please reset it to avoid upgrade failures.

8. Check for Unknown data types:
PostgreSQL versions 10 and later don't support unknown data types. UNKNOWN data type causes upgrade failure. No 'UNKNOWN' datatype found.

9. Check for Read Replica upgrade failure:
In RDS Postgres, all Read Replicas are upgraded followed up by Source instance, adding up outage. No Read Replica found for RDS instance pg15.

10. Check for Postgres extensions:
PostgreSQL engine upgrade doesn't upgrade most PostgreSQL extensions. To update a Postgres extension after a version upgrade, use the ALTER EXTENSION UPDATE command. 2 user extension found as below. Some extensions ma

name	version
pg_buffercache	1.3
pgstattuple	1.5

11. Check for user access:
This upgrade is being run by user pgone_user. Please make sure this user has access to all database objects. Below are the roles this user is member of. To make sure this user has access to all db objects, pelase grant all users to pg
upgrade doesn't have access to all tables, upgrade will fail.

rolname	member_of
pgone_user	{pgtle_admin,rds_superuser}

12. Check for sql_identifier data type:
Your installation doesn't contain the 'sql_identifier' data type in user tables and/or indexes. No issue found.

STRATEGY 1

pg_upgrade - In-Place Upgrade

How It Works

- Transfers data files directly between versions
- Catalogs are rebuilt, data files are linked or copied
- --link mode: Creates hard links (fastest, same FS)
- --copy mode: Full copy (safer, cross-filesystem)
- --jobs N: Parallel tablespace processing
- --check: Dry run before actual upgrade

Best For

Large databases (100GB+) where minimal downtime is critical. Most commonly used strategy.

Trade-offs

- Requires careful pre-checks and testing
- Rollback requires backup restoration
- Both old and new binaries must be installed

COMMAND

```
# Pre-check (dry run)
pg_upgrade --check --old-bindir=/usr/pgsql-14/bin --new-bindir=/usr/pgsql-17/bin \
  --old-datadir=/var/lib/pgsql/14/data --new-datadir=/var/lib/pgsql/17/data
# Execute with link mode + parallel jobs
pg_upgrade --link --jobs=4 --old-bindir=... --new-bindir=... --verbose
```

pg_upgrade Deep Dive

Link Mode vs Copy Mode – Choose Wisely

Aspect	--link Mode	--copy Mode
Speed	Extremely fast (seconds)	Slower (proportional to DB size)
Disk Space	Minimal additional space	Requires 2x disk space
Safety	Old cluster unusable after	Old cluster preserved
Filesystem	Must be same filesystem	Works cross-filesystem
Rollback	Requires backup restore	Can revert to old cluster
Best For	Large DBs, confident teams	First-time upgrades, caution

Essential Flags Reference

`--check`

Dry run - validates without upgrading

`--link`

Use hard links instead of copying files

`--jobs N`

Parallel processing for tablespaces

`--verbose`

Detailed output for troubleshooting

STRATEGY 2

Dump & Restore

The Clean Slate Approach for Smaller Databases

Workflow

1. pg_dumpall or pg_dump exports the database
2. Install new PostgreSQL version & create cluster
3. Restore dump into new cluster with psql/pg_restore
4. Run ANALYZE to rebuild optimizer statistics
5. Validate data integrity and application compatibility

Optimization Tips

- Use pg_dump -Fc for custom format (compressed)
- Parallel restore: pg_restore --jobs=N
- Exclude large tables, load separately

When to Use

- Need to skip multiple major versions
- Want a clean, defragmented database
- Cross-platform migration (Linux → different arch)

COMMANDS

```
pg_dumpall -U postgres > full_backup.sql
pg_dump -Fc -j 4 -d mydb -f mydb.dump
pg_restore -Fc -j 4 -d mydb_new mydb.dump && vacuumdb --analyze --all
```

STRATEGY 3

Logical Replication

Near-Zero Downtime Through Continuous Replication

Key Considerations

- Requires PostgreSQL 10+ (both publisher & subscriber)
- Configure wal_level = logical on source
- Create publication: CREATE PUBLICATION upgrade_pub FOR ALL TABLES
- Create subscription on target database
- Monitor replication lag before cutover
- Cutover when lag approaches zero

Trade-offs

✓ Near-zero downtime (seconds)

✓ Application can stay connected during sync

✓ Gradual rollover possible

✗ More complex initial setup

✗ DDL changes not replicated automatically

✗ Sequences must be synced manually

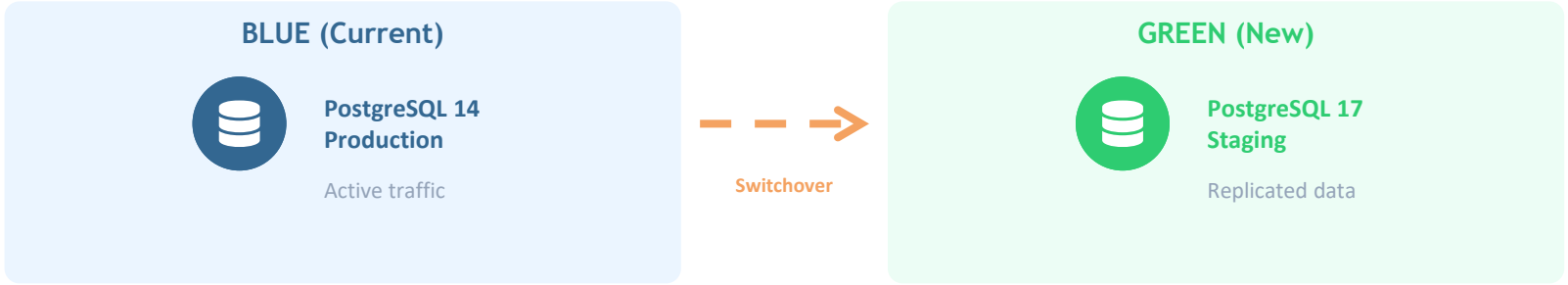
✗ Large objects (LOBs) need special handling

STRATEGY 4

Cloud-Native Upgrade Options

Amazon RDS & Aurora PostgreSQL Upgrade Capabilities

Blue/Green Deployment Architecture



In-Place Upgrade

Modify-db-instance API
Simple,
Best for non-critical workloads

Blue/Green Deploy

Near Zero-downtime switchover
Automatic replication setup
Best for production workloads

Out-of-Place

Snapshot + restore to new
Maximum safety & isolation
Best for cautious migrations

Minimizing Production Downtime

Strategy Downtime Comparison

Strategy	Typical Downtime	Complexity	Best For
pg_upgrade (link)	< 5 minutes	Medium	Large DBs, short window
pg_upgrade (copy)	10-60 minutes	Low-Medium	First-time upgraders
Dump & Restore	Hours	Low	Small DBs, clean migration
Logical Replication	Seconds	High	Mission-critical systems
Blue/Green (RDS)	< 1 minute	Low	AWS managed databases

Key Downtime Minimization Techniques

Pre-sync Data

Use logical replication to sync data before cutover window

Parallel Processing

pg_upgrade --jobs=N for multi-tablespace upgrades

Connection Draining

Gracefully drain connections via PgBouncer PAUSE

Staged Rollout

Upgrade read replicas first, then promote new primary

PITFALL 1

Storage & Configuration Pitfalls

Real-World Lessons from Production Upgrades

CASE STUDY

Storage Full During Upgrade

What Happened:
A 2TB database upgrade using pg_upgrade --copy failed at 85% when the disk filled up. The upgrade had to be aborted, and the partially copied data directory was corrupted.

Root Cause:
Team didn't account for temporary files, WAL segments, and TOAST tables in disk space calculations.

Prevention:
- Always calculate: DB size × 2.5 for copy mode
- Use --link mode when on same filesystem
- Monitor disk usage during upgrade with df -h
- Pre-clean: VACUUM FULL large tables first

CASE STUDY

Configuration Mismatch

What Happened:
After upgrading from PG 13 to PG 16, queries that previously took 200ms now took 5+ seconds. The application experienced cascading timeouts.

Root Cause:
postgresql.conf wasn't migrated. Default settings for shared_buffers, work_mem, and effective_cache_size were used.

Prevention:
- Diff old vs new postgresql.conf before upgrade
- Script config migration with parameter mapping
- Review removed/renamed GUC parameters
- Run ANALYZE on all databases post-upgrade

PITFALL 2

Application & Performance Pitfalls

CASE STUDY

Application Incompatibility in Production

Scenario: Application used deprecated implicit casts and PG-specific syntax that changed between versions. ORM-generated queries failed silently.

Fix: Replay production query logs against test cluster. Use pg_stat_statements to compare execution plans. Update application code before production cutover.

CASE STUDY

Post-Upgrade Performance Regression

Scenario: Query optimizer changed plans after upgrade. Hash joins replaced merge joins, causing 10x slowdown on specific report queries.

Fix: Run ANALYZE immediately post-upgrade. Compare EXPLAIN plans before/after. Use pg_hint_plan for critical queries while tuning optimizer settings.

Prevention Strategies

Query Log Replay

Capture and replay production queries against test cluster to find incompatibilities before they hit production

EXPLAIN Comparison

Systematically compare query execution plans between old and new versions using pg_stat_statements

Staging Parity

Mirror production data, configuration, extensions, and workload patterns in your test environment

Gradual Rollout

Route small percentage of traffic to upgraded cluster first; monitor metrics before full switchover

PITFALL 3

Extensions & Dependencies

The Hidden Upgrade Blockers

CASE STUDY

Extension Breaking Production

PostGIS 2.5 was incompatible with PG 15. Application relied heavily on spatial queries. The extension couldn't be loaded, blocking the entire database startup.

Solution: Pre-validate all extensions against target version. Upgrade extensions in staging first. Plan for extension upgrade path (PostGIS 2.5 → 3.4).

CASE STUDY

Don't Be Greedy - Multi-Version Jump

Team tried to jump from PG 11 to PG 16 in one step. Multiple deprecated features, changed behaviors, and extension incompatibilities combined to create an unmanageable upgrade.

Solution: Step-by-step version upgrades (11→12→13→...) or use logical replication for direct jumps with thorough testing.

Common Extension Compatibility Matrix

Extension	PG 14	PG 15	PG 16	PG 17
PostGIS	✓ 3.3	✓ 3.4	✓ 3.4+	✓ 3.5
pg_partman	✓ 4.x	✓ 4.x	✓ 5.x	✓ 5.x
TimescaleDB	✓ 2.x	✓ 2.x	✓ 2.x	⚠ Check
pgvector	✓ 0.4	✓ 0.5	✓ 0.6	✓ 0.7
pg_cron	✓ 1.5	✓ 1.6	✓ 1.6	✓ 1.6

Your Upgrade Action Plan

A 7-Step Framework for Successful PostgreSQL Upgrades

1 ASSESS

Inventory current state, identify risks & blockers

2 PLAN

Choose strategy, build timeline, allocate resources

3 TEST

Execute comprehensive testing framework

4 PREPARE

Train team, document procedures, set rollback

5 EXECUTE

Follow runbook, monitor closely, validate results

6 OPTIMIZE

Post-upgrade tuning, performance validation

7 DOCUMENT

Capture lessons learned, update procedures

PostgreSQL 14 EOL Timeline

Key Takeaways

Major version upgrades are manageable with a systematic approach – don't let fear paralyze action

Your testing framework is the #1 confidence builder – invest time in comprehensive test infrastructure

Choose your upgrade strategy based on constraints: downtime tolerance, complexity budget, and risk appetite

PostgreSQL 14 EOL is November 2026 – start planning NOW. Every month of delay increases risk

Cloud-native options (RDS/Aurora Blue/Green) dramatically simplify the upgrade process

Learn from others' mistakes – storage, config, extensions, and performance are the common pitfall categories

Resources & Next Steps

PostgreSQL Official Docs

- pg_upgrade documentation
- Logical replication guide
- Release notes (each major version)
- Migration & compatibility notes

AWS Resources

- RDS Blue/Green deployment guide
- Aurora PostgreSQL upgrade best practices
- AWS Database Migration Service
- AWS Database Blog – PostgreSQL articles

Community Resources

- PostgreSQL mailing lists (pgsql-general)
- PostgreSQL Wiki – upgrade guides
- Stack Overflow (postgresql tag)
- Postgres Conference presentations archive

Essential Tools

- pg_upgrade – in-place upgrade tool
- pgBackRest – backup & restore
- PgBouncer – connection pooling
- pg_stat_statements – query analysis

Thank You!

Questions & Discussion

Shaily Porwal

Sr. Technical Account Manager

Amazon Web Services (AWS)

- Enterprise customer success specialist
- Database migration & upgrade advisor
- AWS technical architecture guidance

Vivek Singh

Principal Database Specialist – PostgreSQL

Amazon Web Services (AWS)

- 17+ years in open-source database solutions
- Focuses on Aurora & RDS for PostgreSQL
- Enterprise performance optimization expert