

# Best Practices for Creating Ephemeral Environments with PII for Postgres

May 2025

Monica Sarbu Founder & CEO @ Xata.ai



Get access

#### 💓 xata

# Postgres at scale\*

For AWS RDS, Amazon Aurora, GCP CloudSQL, Azure Database. Or we host it for you.

#### Get access

Improve developer velocity with instant Copy-on-Write branches (including anonymized data) for any Postgres provider, and deploy confidently using zero-downtime schema changes.

<u>Learn about staging and dev environments</u>  $\rightarrow$ 

Staging ()					
Staging 🗐 ···· 🦚 Dev DB branch 🗐		Dev	DB	branch	
Dev DB branch 😂	Staging 🖯 🖨				
		Dev	DB	branch	

Performance and cost efficiency by maximizing your production workloads with Postgres running at the speed of local NVMe storage.

 $\underline{\texttt{See the benchmark}} \, \rightarrow \,$ 



Designed for flexibility by deploying Xata in your own cloud account. You pay your provider directly, and your data stays fully within your infrastructure for security and compliance. Explore the Xata platform  $\rightarrow$ 

							•	۵			aw	s,										

AI-driven optimizations letting the Xata Agent monitor your database and automatically surface performance improvements. Meet the Xata Agent  $\rightarrow$ 



"Postgres at scale" for Xata means more than just handling large data or compute.

It's about scaling team productivity and operations.



From conversations with technical leaders across the industry

# Velocity is the #1 bottleneck holding engineering teams back.

Why?

## Shipping is easy. Not breaking things is hard.

The hardest part of testing your PR? Testing against your database

Most bugs don't come from your code. They come from how your code interacts with your database.

Common practices

# How are teams testing their pull requests against the database?

## **Common Practice: Local Postgres**

Developers run PostgreSQL locally

#### **Pros:**

- Fully isolated
- Fast feedback loop

#### Cons:

- Seeded with fake or minimal test data
- Local config might differ from staging or production.
- No collaboration



## **Common Practice: Test against production**

Especially common in early-stage startups

#### **Pros:**

- Real data & real configuration
- No need for mock/staging setup
- Fast feedback loop

#### Cons:

- High risk of data corruption
- May trigger real emails/notifications
- Migrations or load tests can cause outages
- Harder to isolate/test edge cases safely

#### Convenient but dangerous



PRODUCTION

#### 🗙 xata

## Common Practice: Read replica (RDS/Aurora read replica)

AWS native, automatically synced read-only copy of production

#### **Pros:**

- No schema or data drift
- Great for testing read performance, benchmarking queries

#### Cons:

- Read-only, cannot test writes or schema changes
- Not useful for full end-to-end testing or integration testing



## **Common Practice: Production clone**

A full copy of the production database Includes schema, indexes, and all the data

#### **Pros:**

- Mirrors production exactly
- Useful for performance testing, regression tests, and debugging
- No need to mock or synthesize test data **Cons:** 
  - PII exposure risk: may contain sensitive user data
  - High storage costs: large datasets increase cloud or infra bills
  - Slow cloning/syncing: operational overhead and delays



PRODUCTION

## **Common Practice: Shared staging environments**

One or two shared Postgres instances are used by all developers and QA

#### **Pros:**

- Mirrors production infrastructure and configuration
- Safe, isolated environment for testing
- Great for end-to-end testing and QA validation

#### Cons:

- Shared Postgres instances lead to test data gets overwritten, state becomes unpredictable, and debugging gets harder.
- Staging tends to drift from production over time
- High setup and maintenance overhead





Rethinking database testing

## Test your database like you test your code.

Use branches, run tests, merge with confidence — all in a Git-like workflow.

Goal

## Move fast without breaking prod



## Creating staging environment with one click

Connect to your production database wherever it is Create staging replica with production like data

Projects > Branches Branches Each branch contains a database schema that can be edited.		Create Branch
Name 🗸	Created At 🗸 🗸	
main	22h ago	
staging	22h ago	

## **Keeps Private Data Safe**

Avoids using raw production data in test environments

Mask sensitive fields (e.g., names, emails, addresses)

Keep structure & statistical relevance without exposing real data

Reduces risk of data leaks and ensures regulatory compliance (GDPR, HIPAA)



### Introducing pgstream

Open-source tool for capturing data and schema changes from Postgres.

Supports replication to Postgres, Elasticsearch, Kafka, webhooks.

Key Features:

- DDL change detection (CREATE, ALTER, DROP)
- Snapshot + real-time streaming
- Data transformation and masking during sync



## Using pgstream for data masking

Supports parallel snapshotting for fast data copying

Automated masking built into the snapshot process

Writes anonymized snapshots to staging

Runs on a schedule or in CI/CD pipelines (ex. Run nightly)

Can operate from a read replica to protect production





#### **Deterministic & Realistic Transformers**

**Column-level** anonymisation for fine-grained control

**Deterministic** to always produce the same output for the same input

**Realistic** allows you to generate output that looks and behaves like real data



## **Built-in & custom transformers**

pgstream integrates with:

- Greenmask
- NeoSync
- go-masker

Supports custom transformers in Golang for advanced masking needs



## **Smart Sampling**

If production data is in terabytes, use a smaller, representative subset to test performance

Subsetting is complex because you need to preserve relationships across tables to keep the data consistent.

The 'orders' table is subsetted to 5% of the total size.



## From Staging to Branching

Creating **copy-on-write branch** for each pull request

Data is not physically copied at the time of branch creation.

Instant branches with schema and data

Branches let devs test schema changes, migrations, and queries **in isolation**.

Accelerate development, testing, and collaboration.

This makes branching fast and storage-efficient.



## **Copy-on-write Branching**

Built on a multi-tenant block storage system that separates storage from compute.

Uses **standard PostgreSQL** – no forks or custom storage extensions

Enables precise mirroring of production environments

ojects > Branches		
Branches ach branch contains a database schema that can be edited.	0	Create Branch
Name 🗸	Created At 🗸 🗸	
main		
staging	23m ago	
- devi		
dev2		
- dev3		
dev4	21m ago	

## Copy-on-write branching explained

#### Step1: Initial state



Step3: Copy only modified blocks



#### Step2: Create a new branch



**No full copy** - new branches share existing data blocks

Instant creation - only metadata (index)
is copied

```
Write isolation – modified blocks are copied only when changed
```

# A Git-like workflow for your database





# Schema changes is one of people's least favorite parts of working with databases

### Playbooks for schema changes without downtime

#### GitLab

tutatase mgration pipeline	
Database review guidelines	Migration Style Guide
Database reviewer guidelines	ingration office outdo
Database check-migrations job	When writing migrations for GitLab, you have to take into account that these are run by hundreds of thousands of organizations of all sizes, some with many years of data in their database.
upgrade job	
Dil dump	in addition, having to take a server offline for an upgrade small or big is a big burden for most organizations. For this reason, it is important that your migrations are written carefully, can be applied
Delate solution and and and	online, and adhere to the style guide below.
Change of the state of the stat	Migrations are not allowed to require GitLab installations to be taken offline ever. Migrations always mus
Enans	be written in such a way to avoid downtime. In the past we had a process for defining migrations that
Foreign keys and associations	allowed for downtime by setting a towntime constant. You may see this when looking at older migrations
Hash indexes	This process was in place for 4 years without ever being used and as such we've learned we can always
Insert into tables in batches	ngure out now to write a migration omeramity to avoid downlime.
Introducing a new database migration version	When writing your migrations, also consider that databases might have stale data or inconsistencies and quard for that. Try to make as few assumptions as possible about the state of the database.
Renarry secons in perches	
ana merge requests	Don't depend on GitLab-specific code since it can change in future versions. If needed copy-paste GitLa
Large tables limitations	code into the migration to make a forward competible.
Layout and access patterns	
Maintenance operations	Choose an appropriate migration type
Migration ordering	
Migrations style guide	The first step before adding a new migration should be to decide which type is most appropriate.
Multiple detabases >	These are consulty these blacks of sciencifiers on one could descenden as the black of could it would be
Namespaces	perform and how loop it takes to complete:
Navigation sidebar	harden and a state of a
NOT NULL constraints	1. Regular schema migrations. These are traditional Rails migrations in ob/esgrate that run before
Offset pagination optimization	new apparation code is deputyed (for whilab.com before Canary is deployed). This means that they should be relatively fast no more than a few minutes, so so not to unnecessarily delay a
Ordering table columns	declarment.
Pagination guidelines >	
Polymorphic associations	One exception is a migration that takes longer but is absolutely critical for the application to operat
Post-deployment migrations	correctly, non-exempte, you might have indices that enforce unique tuples, or that are needed for might performance in critical parts of the application. In cases where the micration would be
Query comments with Morginalia	unacceptably slow, however, a better option might be to guard the feature with a feature flag and
Query count timits	finishes
Query performance guidelines	Migrations used to add new models are also part of these regular schema migrations. The only
Query Recorder	differences are the Rails command used to generate the migrations and the additional generated
Rename database tables	files, one for the model and one for the model's spec.
Scalability patterns >	2. Post-deployment migrations. These are Rails migrations in db/post migrate and are run
Serializing data	independently from the GitLab.com deployments. Pending post migrations are executed on a daily
Setting multiple values	basis at the discretion of release manager through the post-deploy migration pipeline. These
SHA1 as binary	migrations can be used for schema changes that aren't critical for the application to operate, or da
Single Table Inheritance	migrations that take at most a few minutes. Common examples for schema changes that should ru
Semantic Versioning of	post-depixy include:
ilapse sidebar	<ul> <li>Cleanurs, Bio removing unused columns.</li> </ul>

https://docs.gitlab.com/ee/development/migration\_style\_guide.html

#### **PayPal**



https://medium.com/paypal-tech/postgresql-at-scale-database-sch ema-changes-without-downtime-20d3749ed680



#### Custom tools for schema changes



🔀 xata

### Incidents or near-incidents caused by schema changes





### Why schema changes are hard



**Locking tables during migrations.** Large tables can be locked for long periods of time, causing downtime.



**Limited deployment windows**. Changes often need to wait for off-peak hours, slowing down iteration.



**Rollback risk.** Rolling back migrations is often left untested, and is error-prone.



**Complex development workflows.** Performing schema changes safely requires multiple manual steps and multiple pull requests.

## Introducing pgroll

#### https://github.com/xataio/pgroll



#### pgroll - Zero-downtime, reversible, schema migrations for Postgres

pgroll is an open source command-line tool that offers safe and reversible schema migrations for PostgreSQL by serving multiple schema versions simultaneously. It takes care of the complex migration operations to ensure that client applications continue working while the database schema is being updated. This includes ensuring changes are applied without locking the database, and that both old and new schema versions work simultaneously (even when breaking changes are being made!). This removes risks related to schema migrations, and greatly simplifies client application rollout, also allowing for instant rollbacks.

#### https://pgroll.com



See the introductory blog post for more about the problems solved by pgroll.

#### pgroll - unique approach

No Downtime - safe for prod

Multi-Version Schema — dual views for smooth rollout

Expand/Contract Workflow - add first, remove later

Auto Backfills - no manual scripts

Hidden Columns – seamless background changes



## After using pgroll



**Safe by default** All schema changes are safe and guaranteed to not lock tables for a long time.



**Online schema migrations.** High level schema changes operations are performed online, without downtime.



**Instant and safe rollback.** Rolling back migrations means simply dropping the views and temporary objects.



**Simple workflow via multi-version views.** The old and the new schema are available at the same time, which greatly simplifies the schema change workflow.



## GitHub actions integration makes it powerful



Trigger on PR (excluding main)

Install and auth Xata CLI

Create a dev branch based on the PR

Checkout and apply pgroll migrations



#### 🔀 xata

## **Key CLI commands**

```
$ xata branch checkout "${{ github.head_ref }}"
Switched to branch feat_task_assignee
```

```
# connect to the branch
$ psql `xata branch url`
```

\$ xata roll migrate

## **Recommended talk:**

# Anatomy of Table-Level Locks in PostgreSQL

Date: 2025-05-09 Time: 12:40-13:25 Room: **Berlin 2+3** Level: Intermediat Speaker: **Gülçin Yıldırım Jelinek** 



### **Recommended talk:**

The AI DBA agent: Would you trust it to tune your PostgreSQL?

Date: 2025-05-09 Time: 11:10-11:55 Room: Madrid Level: Intermediate Speaker: **Tudor Golubenco** 





Postgres at scale

# Thank you!

🔯 xata.io/discord





