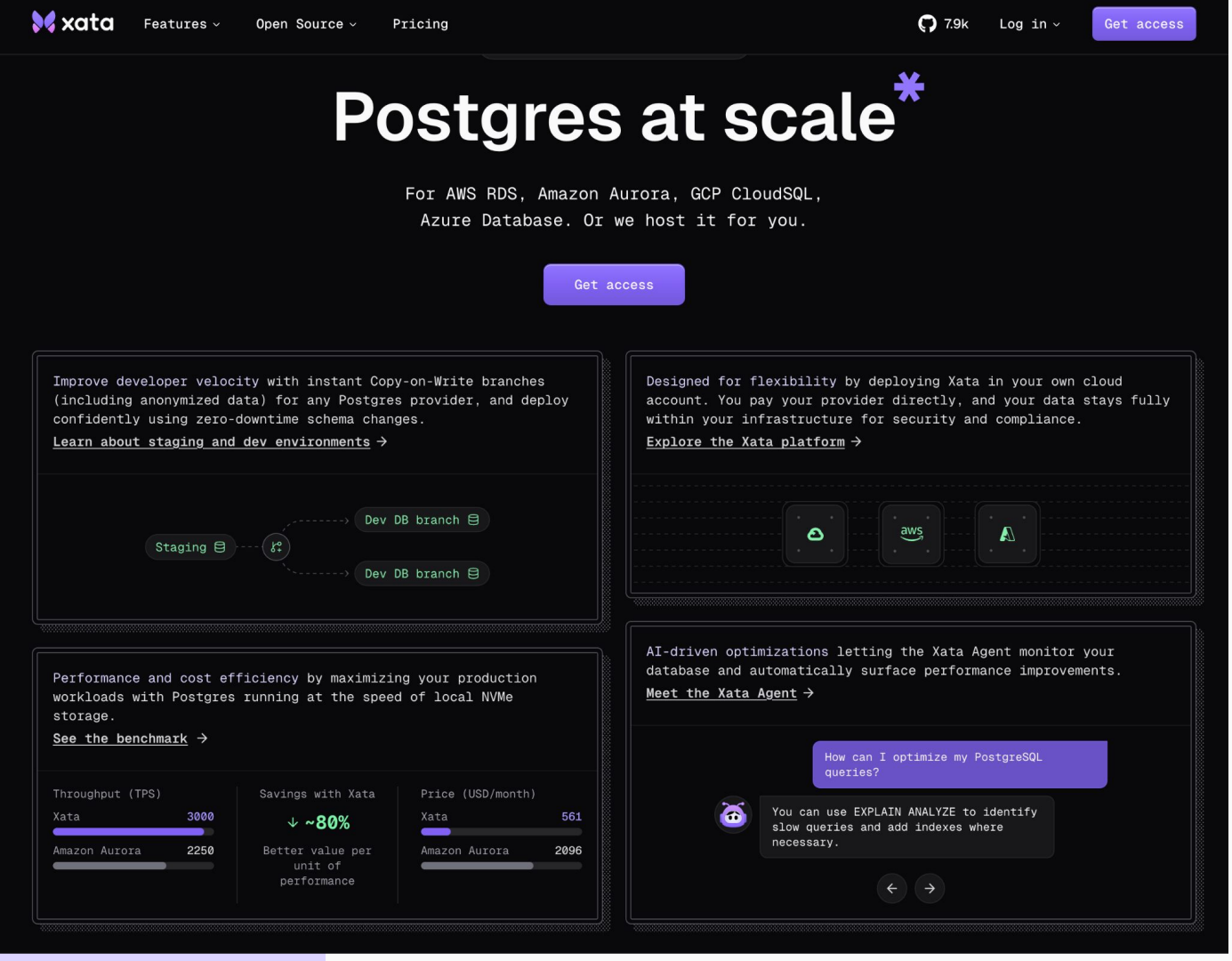
xata

Best Practices for Creating Ephemeral Environments with PII for Postgres

May 2025

Monica Sarbu
Founder & CEO @ Xata.ai

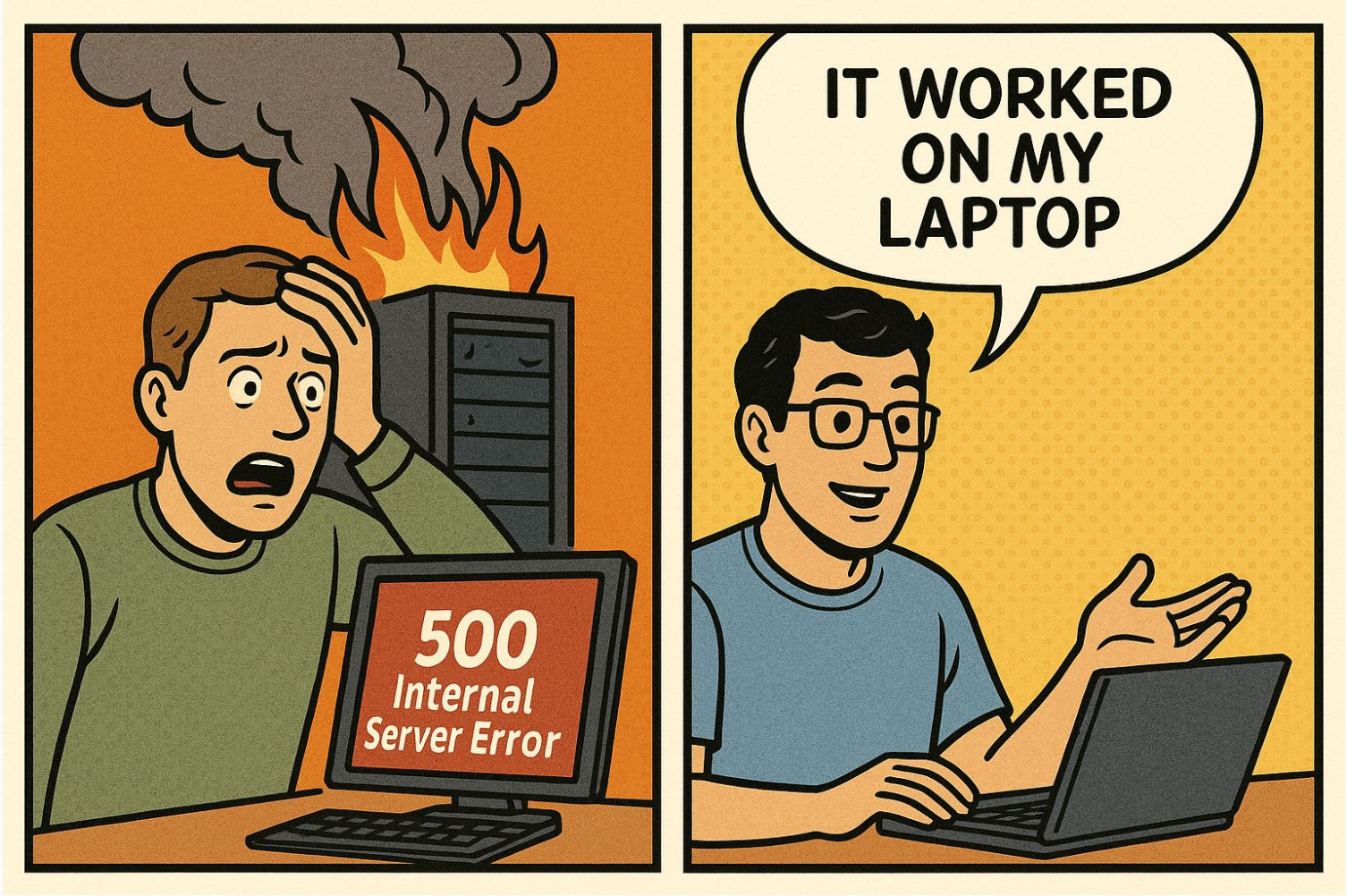
monicasarbu



xata

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

It's about scaling team productivity and operations.


500
Internal
Server Error
IT WORKED
ON MY
LAPTOP

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

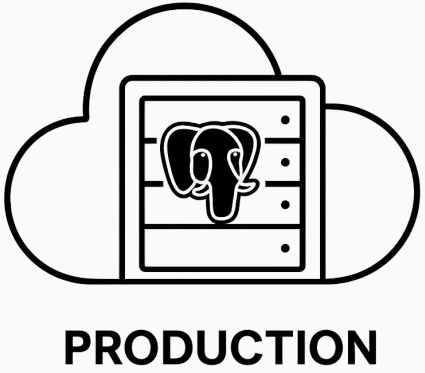
Pros:

- 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**



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

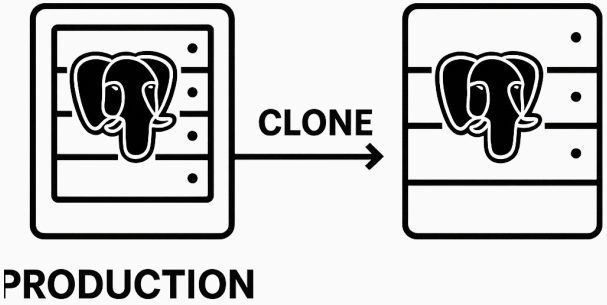
A full copy of the production database
Includes schema, indexes, and all the data

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



Common Practice: Shared staging environments

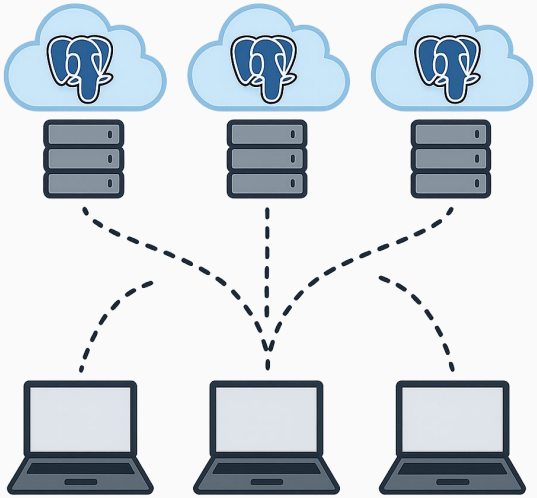
One or two shared Postgres instances are used by all developers and QA

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




Guess I'll test my code... tomorrow.
BUSY
BUSY
BUSY
Waiting for environment to be free...

xata

Rethinking database testing

Test your database like you test your code.

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

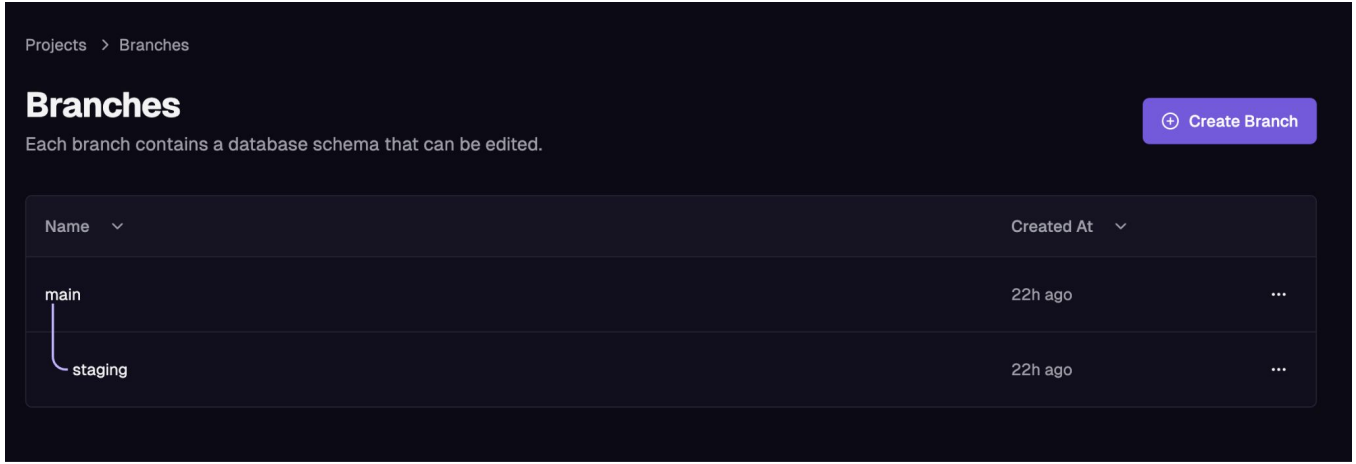
Goal

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



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)

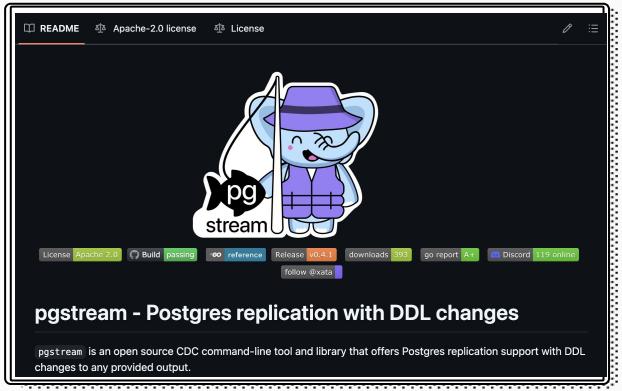
Introducing pgstream

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



https://github.com/xataio/pgstream

Using pgstream for data masking

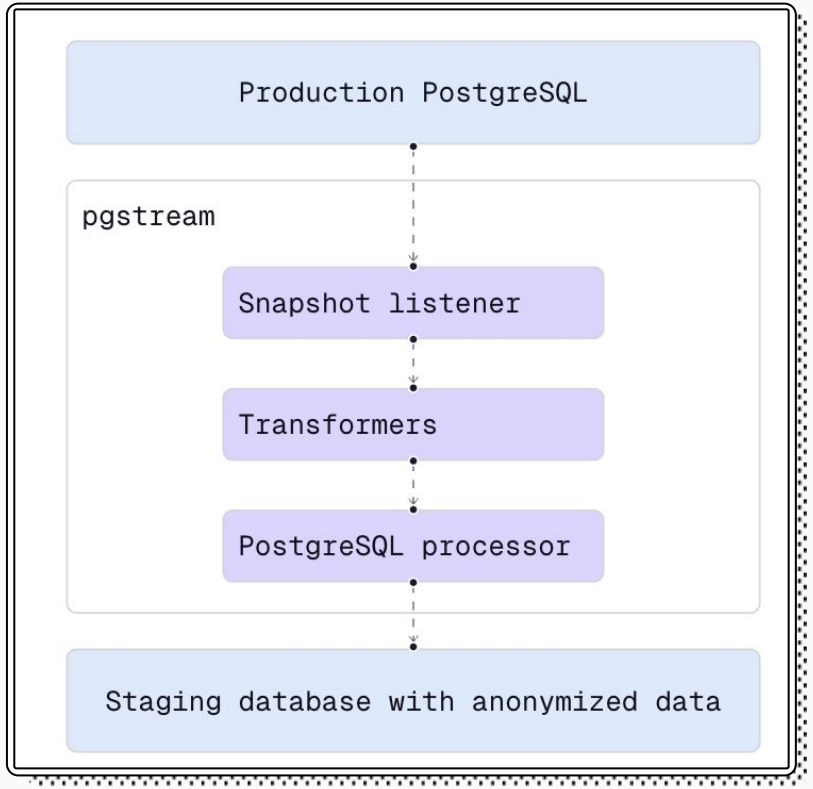
Supports parallel snapshotting for fast data copying

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

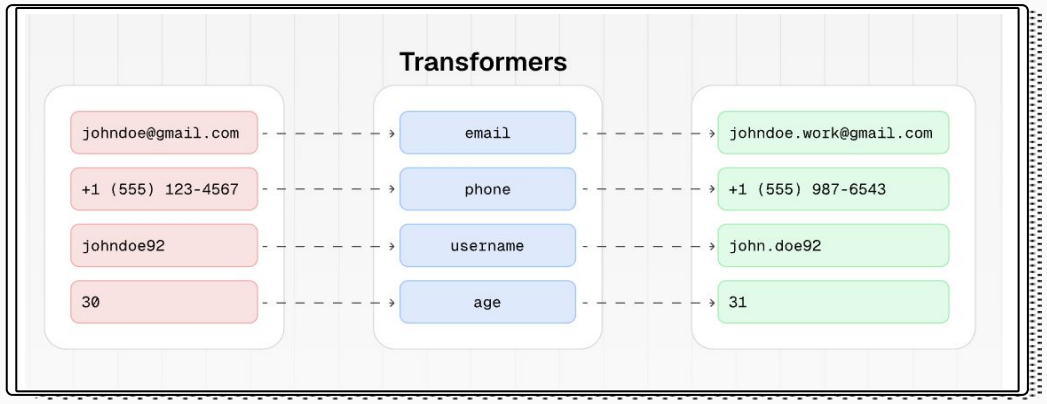


Deterministic & Realistic Transformers

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

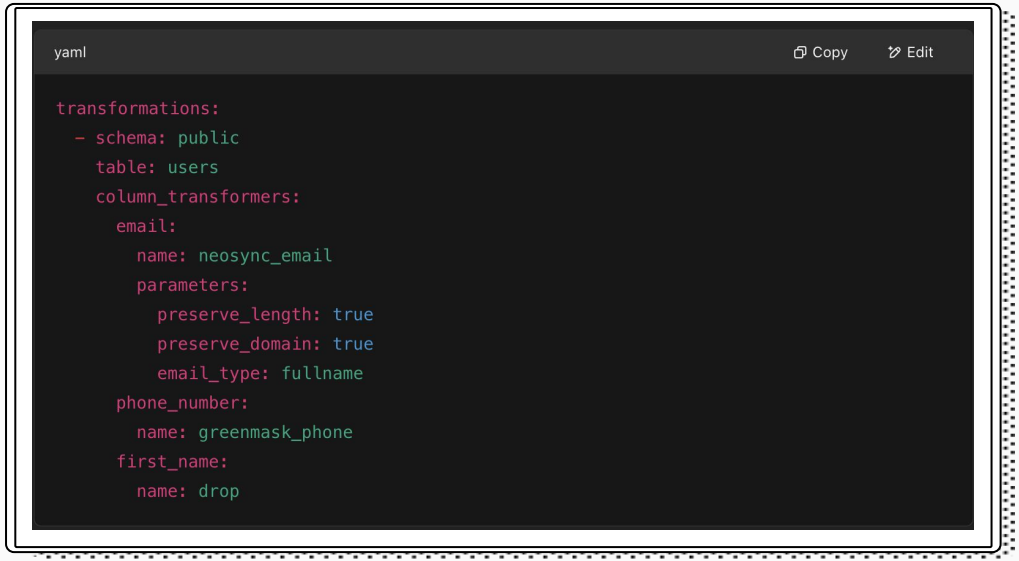


Built-in & custom transformers

pgstream integrates with:

- Greenmask
- NeoSync
- go-masker

Supports custom transformers in Golang for advanced masking needs

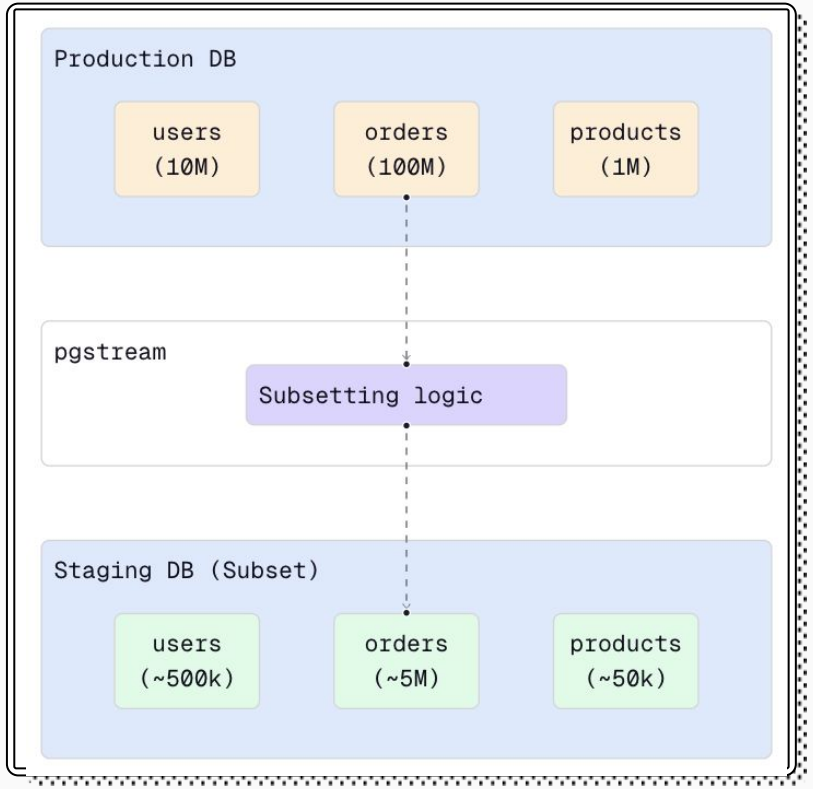


Smart Sampling

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

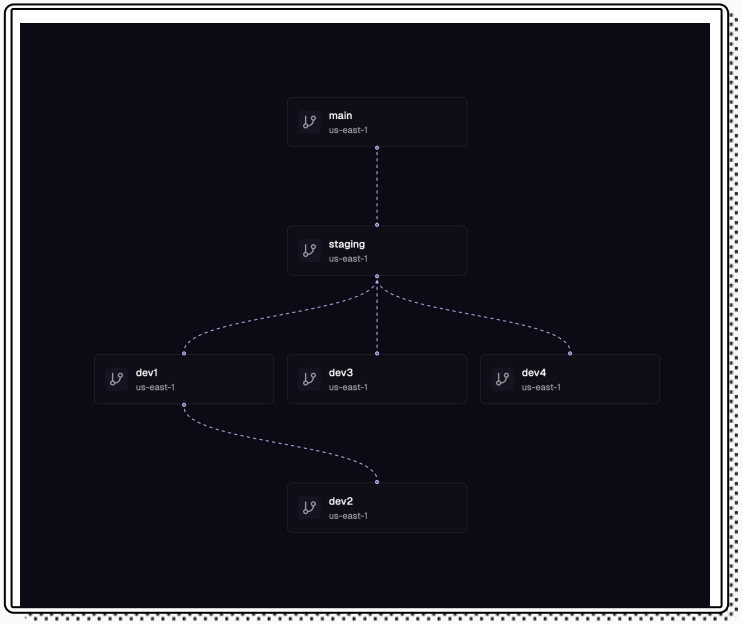
Data is not physically copied at the time of branch creation.

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.**

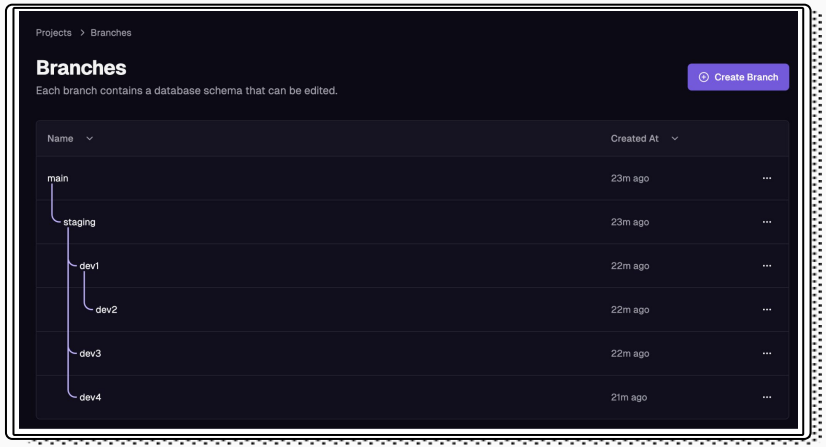


Copy-on-write Branching

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

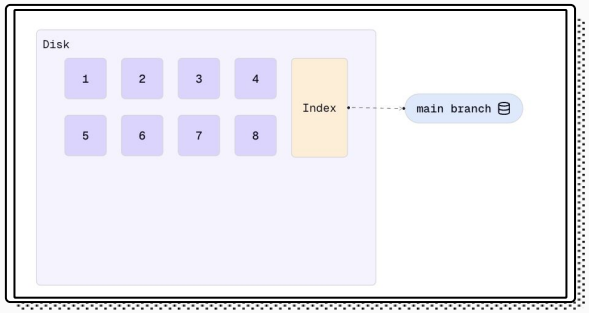
Uses **standard PostgreSQL** — no forks or custom storage extensions

Enables precise mirroring of production environments

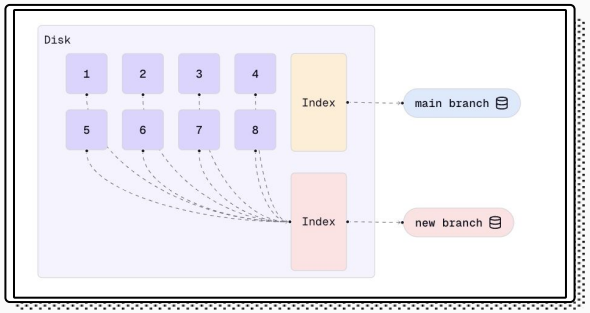


Copy-on-write branching explained

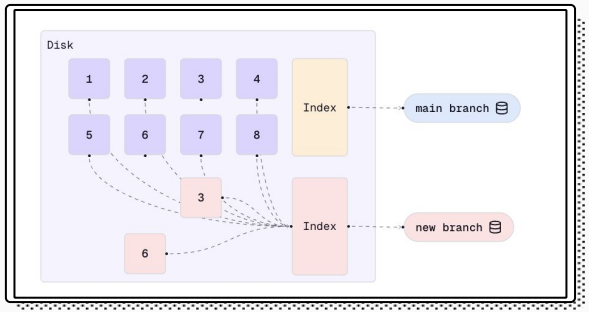
Step1: Initial state



Step2: Create a new branch



Step3: Copy only modified blocks

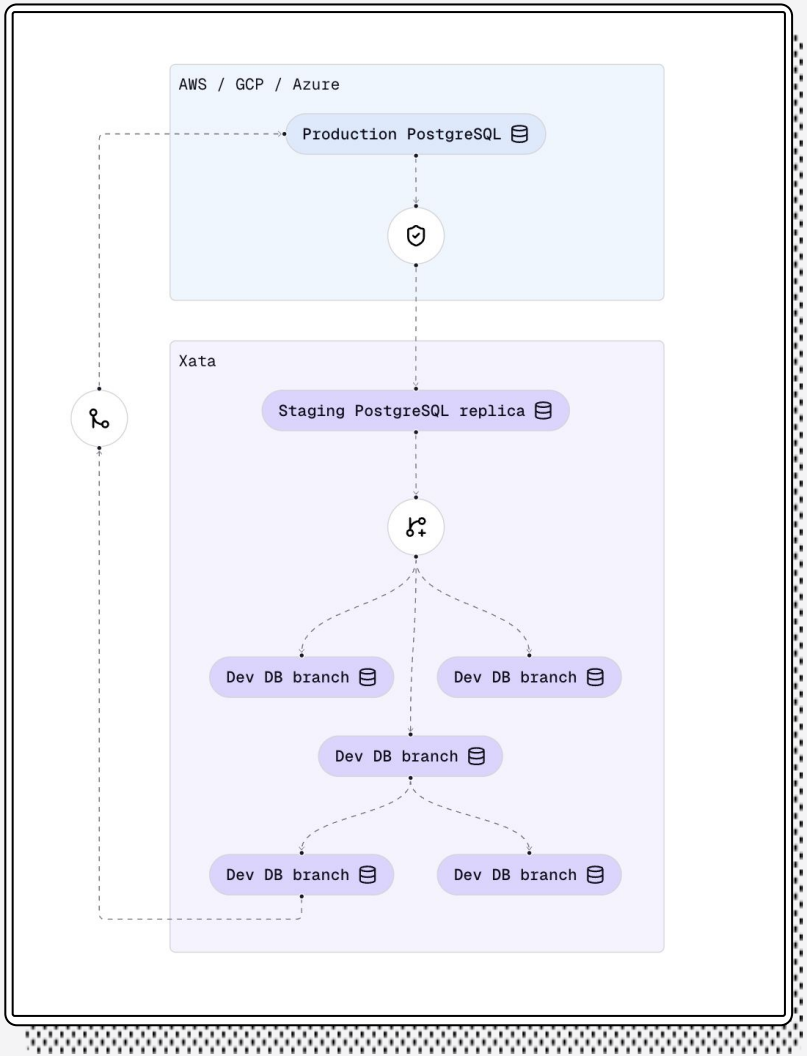


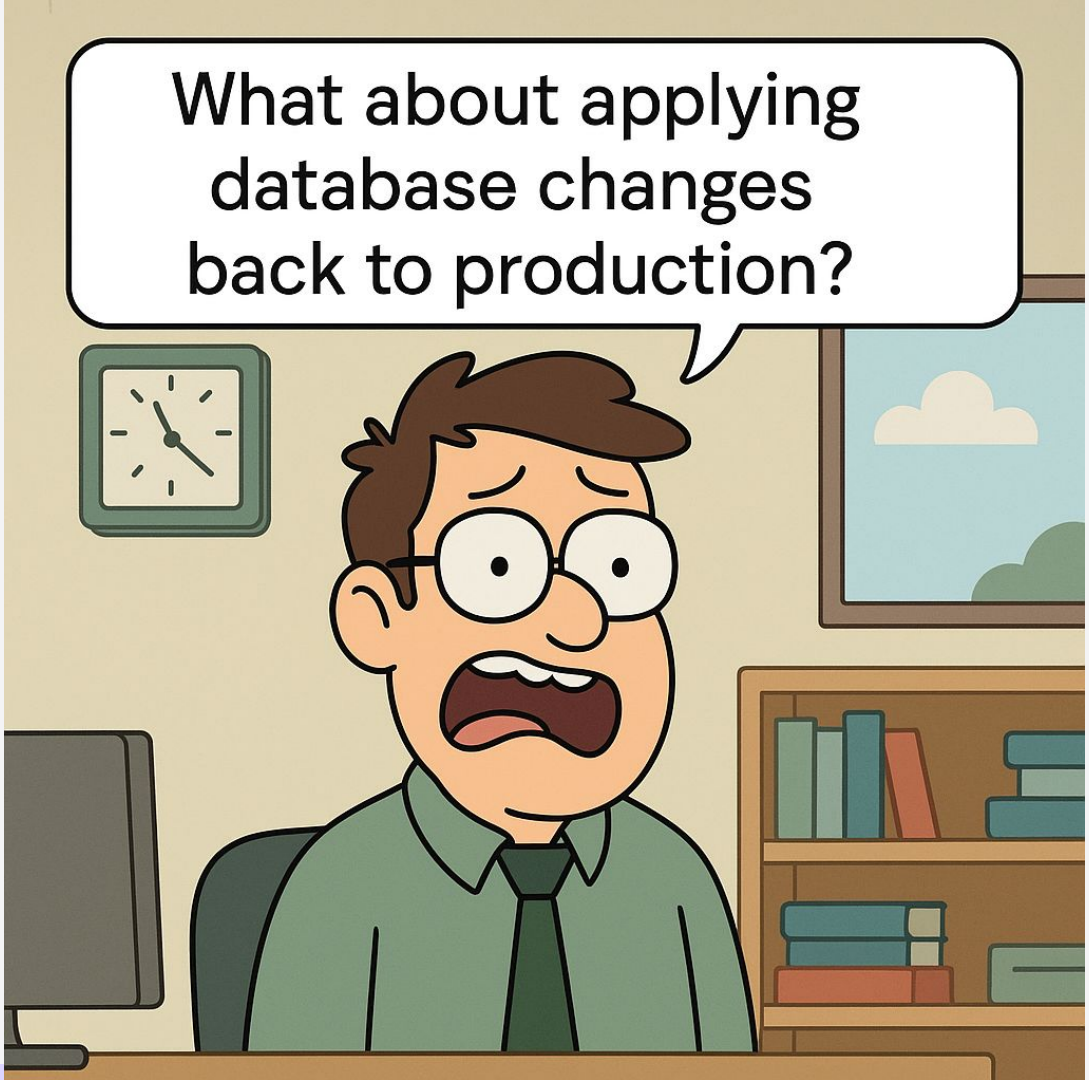
No full copy — new branches share existing data blocks

Instant creation — only metadata (index) is copied

Write isolation — modified blocks are copied only when changed

A Git-like workflow for your database

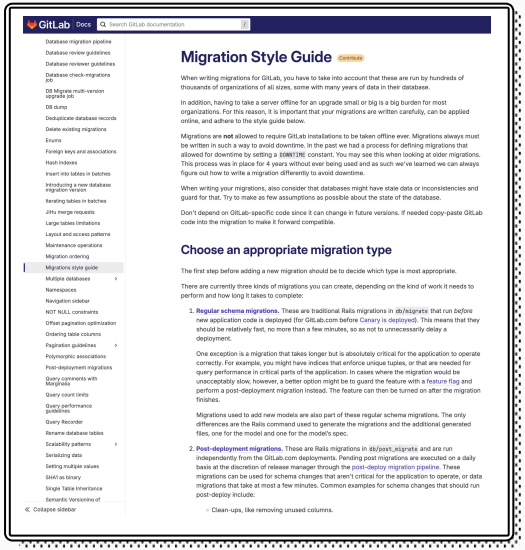



What about applying database changes back to production?

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

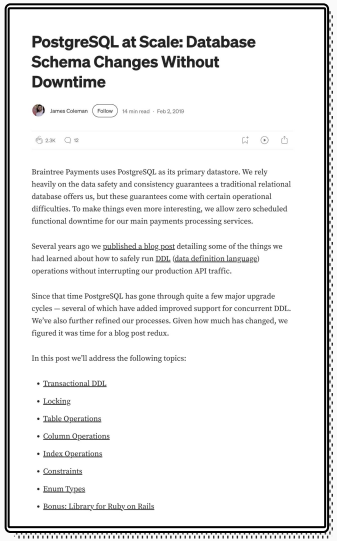
Playbooks for schema changes without downtime

GitLab



https://docs.gitlab.com/ee/development/migration_style_guide.html

PayPal



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

Custom tools for schema changes

Online Schema Change by Meta

https://github.com/facebookincubator/OnlineSchemaChange

Shift by Square

https://github.com/square/shift

Incidents or near-incidents caused by schema changes

GitHub

Occurs semaphore deadlock during schema changes of MySQL

Doctolib

Attempting to add NOT NULL constraint

GoCarless

Foreign key caused exclusive lock

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.

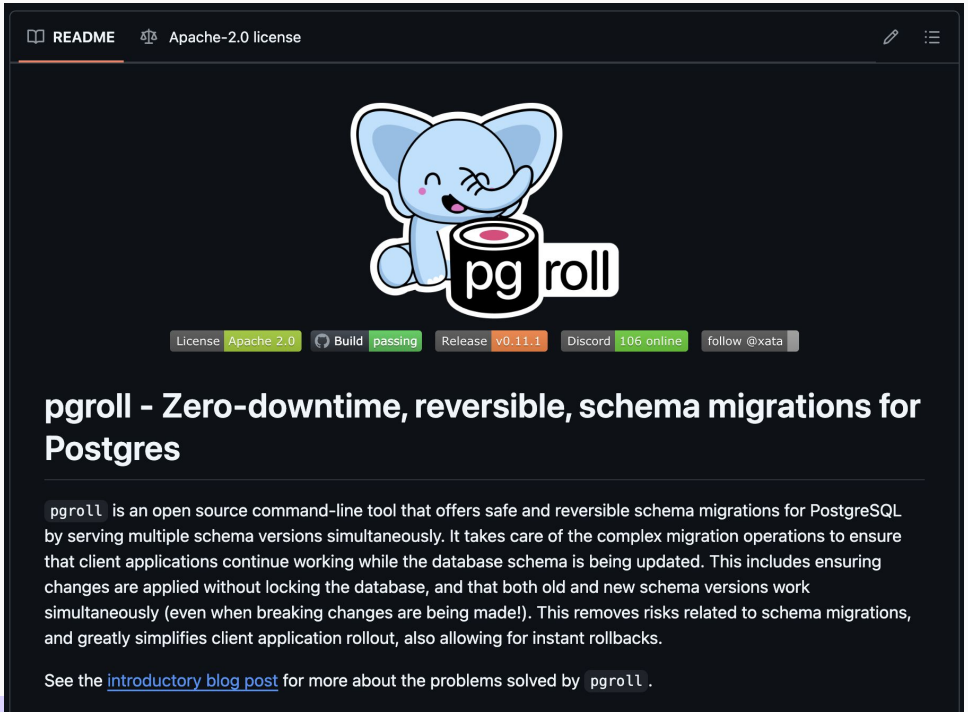
Rollback risk. Rolling back migrations is often left untested, and is error-prone.

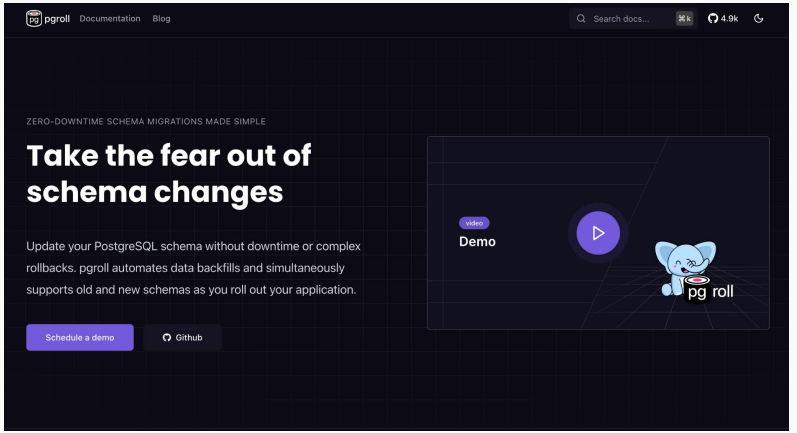
Complex development workflows. Performing schema changes safely requires multiple manual steps and multiple pull requests.

Introducing pgroll

https://github.com/xataio/pgroll

https://pgroll.com





pgroll - unique approach

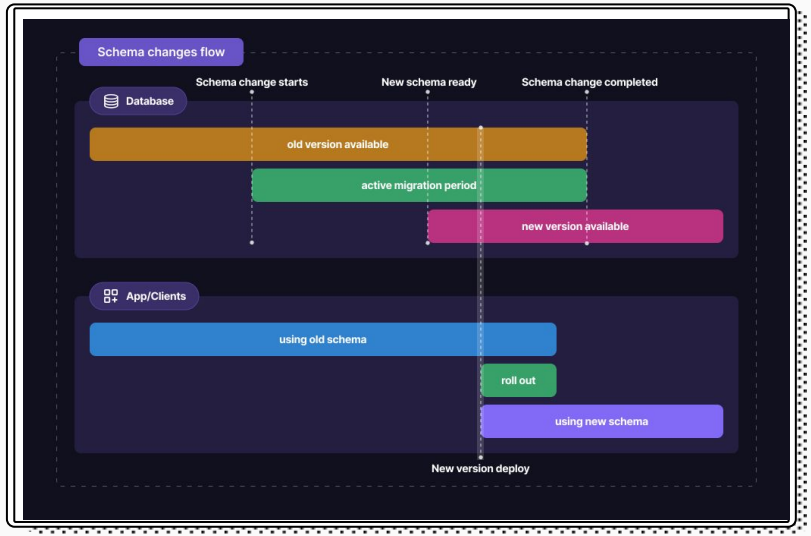
No Downtime – safe for prod

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.

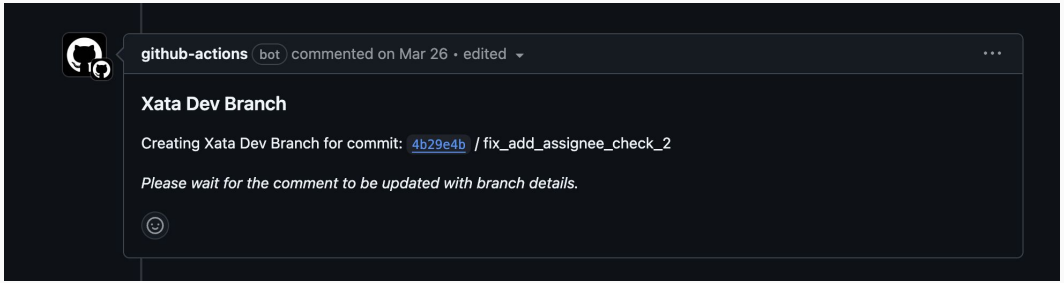
Online schema migrations. High level schema changes operations are performed online, without downtime.

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



Key CLI commands

```
$ xata branch create --name "${{ github.head_ref }}"
Name: feat_task_assignee
Parent Branch Id: ***************************%

$ 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**

xata

Postgres at scale

Thank you!

 xata.io/discord

 @xata

 xata.io

 github.com/xataio

 xataio