

Bringing Database Infrastructure into GitHub

A practical onboarding guide for database teams

October, 2026 (Version 1.2)

Copyright ©2026, [Richard Koranteng](#)

Public

Table of Contents

Executive summary.....	3
Why database teams fall behind.....	3
What belongs in the repository.....	4
Database DevOps maturity model.....	5
Repository design.....	6
Pipelines with GitHub Actions.....	7
Security, compliance, and audit.....	9
Phased onboarding roadmap.....	10
People and process.....	11
Measuring success.....	12
Common pitfalls.....	12
Readiness checklist and next steps.....	13

Executive summary

Most database teams can move their schema, database code, and infrastructure into GitHub in about 90 days, without a big-bang cutover, by onboarding one database at a time.

Application teams have shipped through version control and automated pipelines for over a decade. Database changes at many organizations still move through emailed scripts, shared drives, and change tickets run by hand after hours. The database becomes the slowest and riskiest step in every release.

This paper gives database leaders a practical path to close that gap:

- **What to put in GitHub:** schema migrations, database code, reference data, infrastructure definitions, configuration, and runbooks.
- **How to structure it:** repository layout, branching, code owners, and review rules that respect a DBA team's need for control.
- **How to deploy it:** GitHub Actions pipelines that lint, test, and promote changes through environments with approvals.
- **How to keep auditors satisfied:** an audit trail that is stronger than the manual process it replaces.
- **How to roll it out:** a phased plan, the metrics to track, and the pitfalls that stall most teams.

The outcome is a database release process that is reviewed, repeatable, and fast, with DBAs moving from running scripts to setting the standards the pipeline enforces.

Why database teams fall behind

Databases are stateful, so they cannot be redeployed from scratch like application code. That one difference explains most of the gap, and it is solvable.

- **State makes rollback hard.** A dropped column cannot be undone by redeploying an older build. Teams respond by slowing every change down.
- **The source of truth is the production database.** When nobody trusts the scripts folder, the live schema becomes the only accurate record, and environments drift apart.
- **Knowledge sits with a few people.** Deployment steps live in one DBA's head or a wiki page that is out of date.
- **Change control is built around tickets, not code.** Approvals happen in a ticketing tool, separate from what was actually run.
- **Tooling was aimed at developers.** Many DBAs were never given time or training to adopt Git, pull requests, and pipelines.

The cost shows up as long release cycles, failed deployments caused by environment drift, after-hours work, and audit findings when no one can prove exactly what ran in production.

What belongs in the repository

If a change can affect a database, its definition belongs in Git. Data itself does not.

Asset	Examples	Typical tool
Schema migrations	Tables, indexes, constraints, partitions	Liquibase, Flyway, Sqitch
Database code	Stored procedures, packages, functions, views, triggers	Repeatable migrations in the same tool
Reference data	Lookup tables, status codes, configuration rows	Data migrations, CSV loads
Security model	Roles, grants, users for applications (never passwords)	Migrations or Terraform providers
Infrastructure	Cloud instances, parameter groups, replicas, backups, networking	Terraform, Bicep, CloudFormation
Server configuration	Init parameters, OS settings, agent setup	Ansible, scripts
Operations	Health checks, maintenance jobs, runbooks	SQL, Python, Markdown
Pipelines	Build, test, and deploy workflows	GitHub Actions YAML

Keep out of Git: production data, data dumps, passwords, keys, and connection strings with credentials. These belong in a secrets manager or GitHub's encrypted secrets.

Database DevOps maturity model

Most teams start at Level 0 or 1. Reaching Level 3 for your most-changed databases delivers most of the value; Level 4 is optional.

Level	Name	How changes reach production	Source of truth	Main risk
0	Manual	Scripts emailed or copied from a shared drive, run by hand	The production database	Drift, lost changes, no audit trail
1	Versioned	Scripts stored in Git, still run by hand	Git, partially	Git and the database disagree
2	Migration-managed	A migration tool applies changes and tracks what ran	Git plus the tool's change log	Untested changes reach production
3	Automated pipeline	Pull request, automated tests, approved promotion through environments	Git	Pipeline gaps for edge cases
4	Self-service	Teams ship their own database changes within guardrails the DBA team owns	Git, enforced by policy	Guardrails drift as teams grow

Use the table as a quick self-assessment. Score each database separately: a team often has one database at Level 3 and a legacy system still at Level 0.

Repository design

Start with one repository per application database, owned jointly by the application team and the DBA team. A shared platform repository holds infrastructure and reusable pipeline code.

Suggested layout for a database repository:

```
orders-db/  
  changelog/  
    db.changelog-master.yaml  
    releases/  
      2026.10/          # versioned migrations, one file per change  
code/  
  procedures/          # repeatable: re-applied when changed  
  views/  
data/  
  reference/           # lookup and seed data  
tests/                 # unit tests and data checks  
.github/  
  workflows/           # CI and deployment pipelines  
CODEOWNERS  
README.md              # how to add a change, how to deploy
```

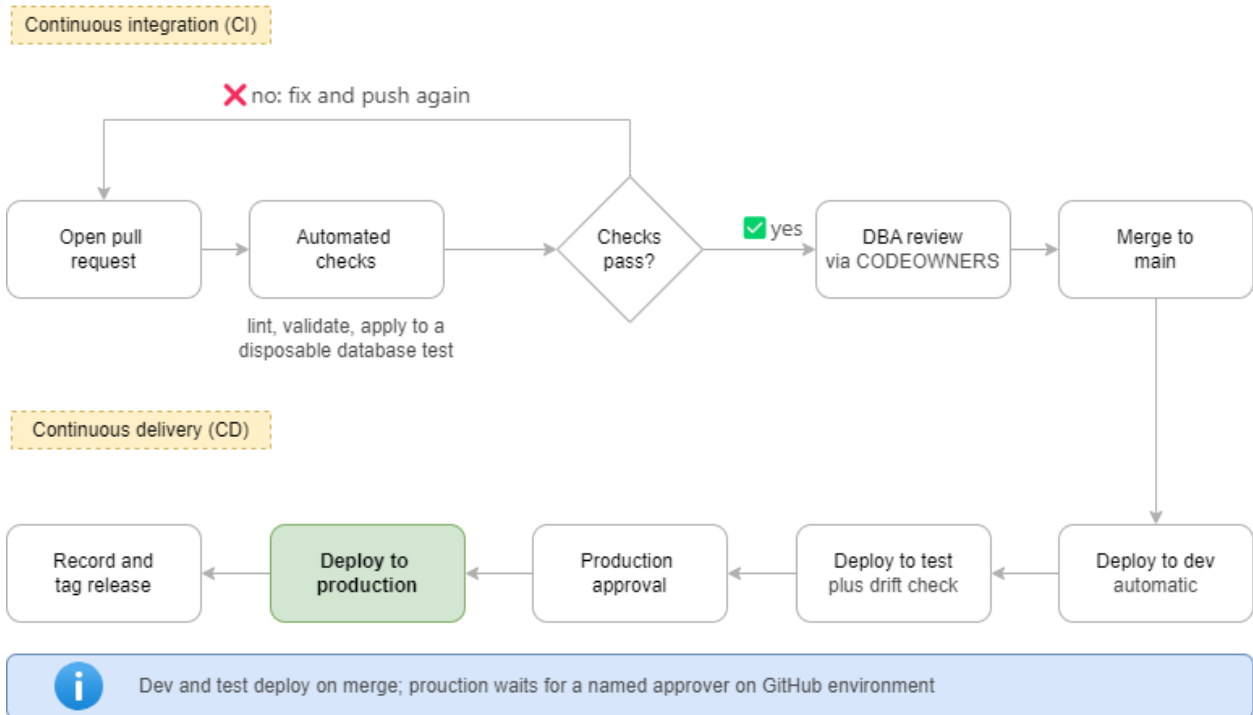
Conventions that prevent most problems:

1. **One logical change per migration file**, named with a sortable ID and a description, such as `20261011_1402_add_orders_status_index.sql`.
2. **Never edit a migration after it has run** in a shared environment. Fix forward with a new migration. Editing causes checksum errors and drift.
3. **Write a rollback for every change**, or mark it explicitly as forward-only so reviewers know.
4. **Use trunk-based development**. Short-lived feature branches merge into `main` through a pull request. Avoid long-lived environment branches; promote the same commit through environments instead.
5. **Protect `main`** with a ruleset: required pull request, at least one approval, passing checks, and no direct pushes.
6. **Use `CODEOWNERS`** so DBA review is required automatically for risky paths, for example `changelog/** @org/dba-team`. Developers can still own reference data or views where that makes sense.

Pipelines with GitHub Actions

One pipeline carries every change from pull request to production, and the same commit is promoted through each environment rather than rebuilt.

Every database change takes the same reviewed path to production



database change pipeline w/ pull request to production

A failed check sends the change back to its author before any person reviews it, so DBAs spend review time on design and risk, not syntax.

What each stage does:

- **Automated checks on every pull request.** Lint the SQL, validate the changelog, apply all migrations to a disposable database (a service container or a cloned test instance), run unit tests, and post the generated SQL to the pull request for reviewers.
- **DBA review.** CODEOWNERS routes changes to schema or security paths to the DBA team automatically.
- **Environment promotion.** Use GitHub environments named `dev`, `test`, and `production`. Each holds its own secrets and protection rules.
- **Production approval.** Add required reviewers and, if needed, a wait timer or deployment window on the production environment.
- **Drift detection.** A scheduled workflow compares each environment with the changelog and opens an issue when they differ.

Runners and credentials: use GitHub-hosted runners for checks against disposable databases. Use self-hosted runners inside your network for on-premises or private databases. Prefer OpenID Connect for short-lived cloud credentials over stored keys.

An illustrative skeleton (adapt the commands to your migration tool):

```
name: deploy-orders-db
on:
  push:
    branches: [main]
jobs:
  deploy-test:
    runs-on: [self-hosted, db-network]
    environment: test
    steps:
      - uses: actions/checkout@v4
      - run: liquibase update          # credentials come from environment secrets
  deploy-production:
    needs: deploy-test
    runs-on: [self-hosted, db-network]
    environment: production          # required reviewers approve here
    steps:
      - uses: actions/checkout@v4
      - run: liquibase update
```

Security, compliance, and audit

A well-built pipeline gives auditors more evidence than a manual process: every production change links to a reviewed pull request, a named approver, and a log of exactly what ran.

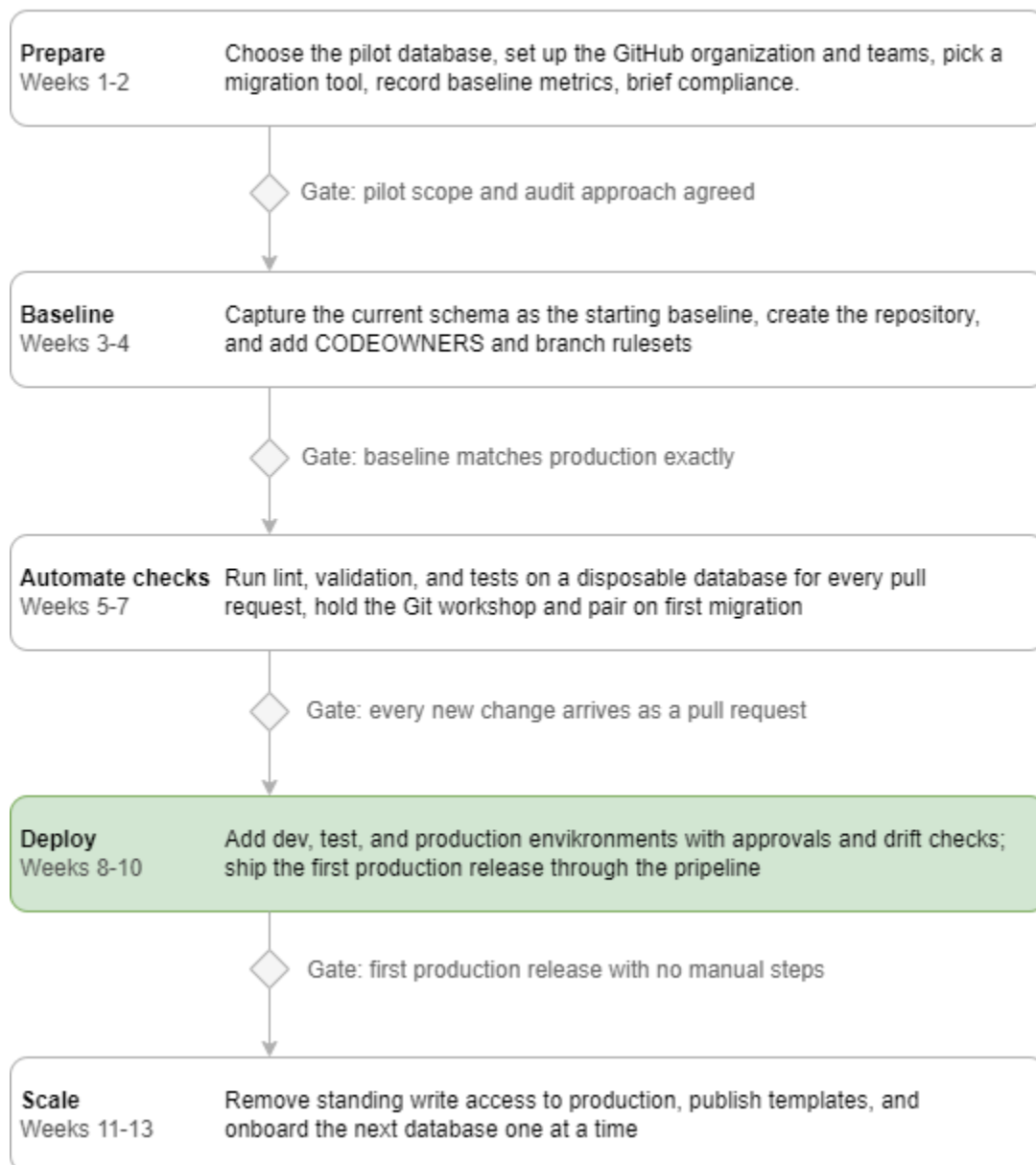
Control	How GitHub supports it
Separation of duties	Rulesets block self-approval; CODEOWNERS require DBA review; environment reviewers approve production deploys
Change approval	Pull request approvals plus a required reviewer on the production environment, recorded with names and timestamps
Traceability	Each deployment ties to a commit, a pull request, a workflow run, and the migration tool's change log in the database
Least privilege	The pipeline account holds deploy rights; individuals lose standing write access to production schemas
Credential protection	Environment secrets, short-lived cloud credentials through OpenID Connect, and secret scanning with push protection
Network isolation	Self-hosted runners inside your network reach on-premises or private databases without opening inbound access
Evidence retention	Workflow logs, pull request history, and the organization audit log (retention varies by plan; export them for long-term records)

Map these controls to the frameworks you already report against, such as SOX, SOC 2, or NIST 800-53, and share the mapping with your auditors before the first production deployment. Early agreement prevents a late veto.

Phased onboarding roadmap

A single pilot database can go from manual scripts to automated production releases in about 13 weeks. Each phase ends with a gate the team must pass before moving on.

One pilot database reaches automated production release in ~ 13 weeks



onboarding roadmap (5 phases, 4 gates)

Later databases move faster, often in three to four weeks each, because the repository template, pipeline, and training already exist. Legacy systems with little change can stay at Level 1 or 2 until there is a reason to invest.

People and process

The hardest part of onboarding is not the tooling. It is changing who does what, and DBAs need to see that their role grows rather than shrinks.

How roles change:

- **DBAs** move from running scripts to owning standards: reviewing risky changes, writing pipeline checks, defining guardrails, and handling the cases automation cannot.
- **Developers** write migrations alongside application code and get fast feedback from automated checks.
- **Release managers** approve deployments in GitHub instead of a separate ticketing step.

Training that works:

1. A half-day Git and pull request workshop using a copy of your real schema, not a toy example.
2. Pair the first five migrations: a DBA and a developer write and review them together.
3. A one-page "how to ship a database change" guide in each repository's README.

Common objections and answers:

Objection	Answer
"We'll lose control of production."	DBAs gain control: no change merges without passing their checks and, for risky paths, their review.
"Our auditors require tickets."	Link the ticket ID in the pull request title or a required check. The pull request becomes better evidence than the ticket alone.
"Our database is too complex for automation."	Start with new changes only. The existing schema becomes the baseline, and nothing old has to be rewritten.
"Rollbacks are too risky to automate."	Most teams fix forward. The pipeline makes small, frequent changes, which are easier to reverse than quarterly bundles.
"We don't have time."	The pilot targets one database and pays for itself through fewer failed and after-hours deployments.

Measuring success

Track the four DORA delivery metrics for database changes specifically, plus two database-only measures. Record a baseline before the pilot starts, or you will not be able to show improvement.

Metric	What it tells you	How to measure
Lead time for changes	Time from merged pull request to running in production	Pull request merge time vs. production deploy time
Deployment frequency	How often database changes reach production	Count of production workflow runs
Change failure rate	Share of deployments that cause an incident or need a fix	Failed or reverted deployments ÷ total
Time to restore	How fast you recover from a bad change	Incident open to resolved
Drift incidents	Schema differences found between environments	Scheduled drift-check failures
Manual production changes	Changes made outside the pipeline	Database audit logs compared to the migration change log

The last metric is the clearest signal of adoption. The goal is zero.

Common pitfalls

- **Trying to migrate everything at once.** Onboard one database, prove it, then repeat.
- **Reverse-engineering decades of history.** Take a baseline of the current schema instead of rebuilding every past change.
- **Leaving a back door open.** If people can still change production by hand, drift returns within weeks. Remove standing write access once the pipeline is trusted.
- **Long-lived environment branches.** Separate dev, test, and prod branches drift apart. Promote one commit instead.
- **Testing only in production-like environments.** Run fast checks on a disposable database for every pull request, and save full regression for staging.
- **Credentials in the repository.** Turn on secret scanning with push protection on day one.
- **No owner for the pipeline.** Pipelines decay without a named owner. Make it part of the DBA team's role.
- **Skipping the auditors.** Bring compliance in during the pilot, not after go-live.

Readiness checklist and next steps

Use this checklist to judge whether your team is ready to start the pilot.

- ☐ A pilot database is chosen: changes often, moderate risk, supportive team
- ☐ A GitHub organization with team accounts and single sign-on
- ☐ A migration tool selected (Liquibase, Flyway, or Sqitch)
- ☐ Network path from GitHub runners to each environment confirmed
- ☐ A secrets approach agreed (environment secrets, OIDC, or a vault)
- ☐ Baseline metrics recorded for lead time, failures, and manual changes
- ☐ Auditors or compliance contacts briefed on the new evidence trail
- ☐ A named owner for the pipeline on the DBA team
- ☐ Time set aside for the Git workshop and paired first migrations

Next step: run a two-week assessment of one database. Map how changes reach production today, score it against the maturity model, and leave with a concrete plan for the pilot.

About the author. Richard Koranteng is a database engineer who helps teams bring their databases into version control and automated CI/CD. To discuss your team's path to GitHub, schedule an intro call at rkkoranteng.com.