CONF42

DATABASE DEVOPS 2026

Schema Migrations in CI/CD

When AI Helps, When It Hurts, and How to Tell the Difference

A field guide to the riskiest part of software delivery — and where automation earns its keep.

Ahmed Tariq

Cloud DevOps Engineer · CI/CD automation, deployment reliability, observability & AI-assisted workflows

THE CORE PROBLEM

Why schema migrations aren't like app deployments

Stateless app deploy

- Ships immutable artifacts — containers, binaries
- Roll back by repointing to the previous image
- Blue/green & canary are well-understood
- Failure radius is contained to compute

Schema migration

- Mutates shared, persistent production data
- Rollback may be impossible once data changes
- Long DDL can lock tables or degrade live traffic
- Environment drift turns staging-green into prod-red

FIELD NOTES

Common failure modes in database releases

Locking DDL

ALTER on a hot table blocks writes; the app times out under load.

Irreversible rollback

Once a column is dropped or data is rewritten, 'just revert' is gone.

Environment drift

Staging and prod diverge — the migration that passed now fails.

Performance cliff

Backfills and index builds saturate I/O and starve live queries.

Big-bang change

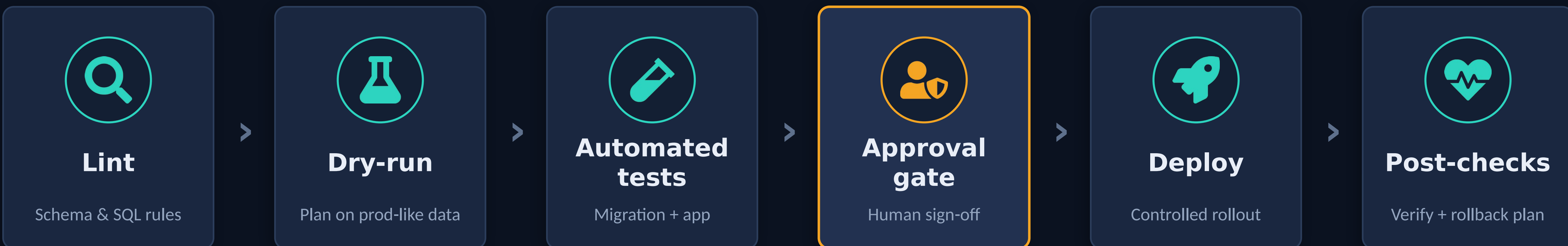
Code and schema must deploy in lockstep — no safe ordering.

Silent data loss

A wrong type cast or truncation corrupts rows without erroring.

THE PIPELINE

What good database CI/CD should include

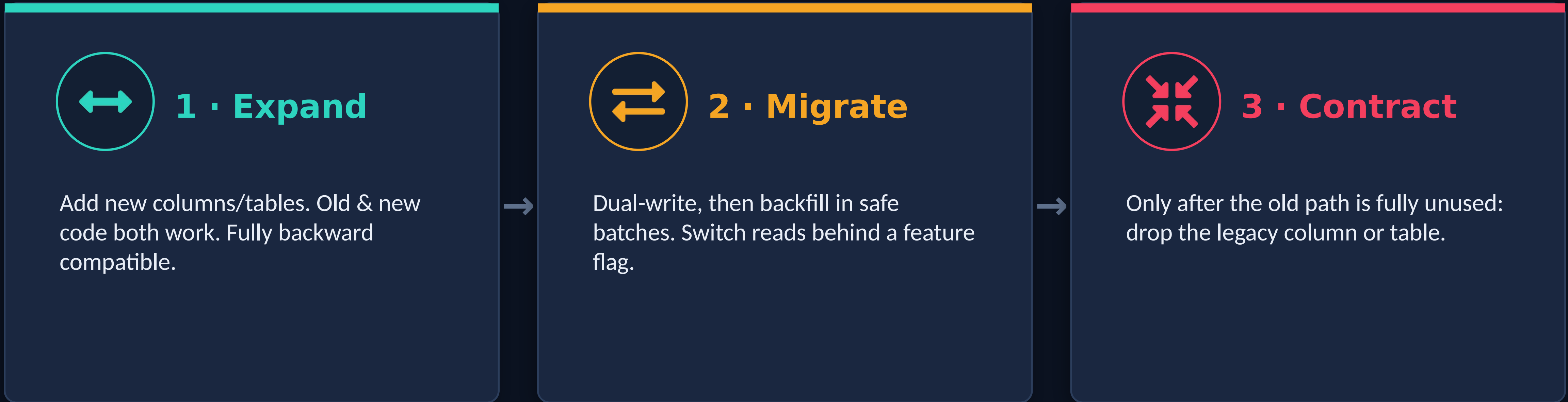


Principle: every stage is automated and repeatable — the human gate is for *judgement*, not for catching mechanical mistakes the pipeline should have caught.

In practice: Flyway · Liquibase · Alembic · Atlas for migrations · GitHub Actions · GitLab CI · Azure DevOps for orchestration

SAFE PATTERNS

Expand → migrate → contract, never big-bang



Why it works: each phase is independently deployable and reversible. Feature flags decouple the *schema change* from the *behaviour change* — so you roll back code, not data.

WORKED EXAMPLE

A safe customer_status migration, step by step

Scenario: add a `customer_status` column to a high-traffic `customers` table — hundreds of millions of rows, live writes.

1 Add nullable column

ALTER … ADD customer_status NULL — avoid defaults & full-table rewrites; verify lock behaviour for your engine.

2 Dual-write

Deploy code that writes both the old field and customer_status. Both paths valid.

3 Backfill in batches

Backfill historical rows in rate-limited chunks; watch lag and lock time.

4 Switch reads behind a flag

Move reads to the new column via a feature flag. Roll back = flip the flag.

5 Verify with observability

Confirm error rate, latency and the old path is provably unused.

6 Contract — later

Only now drop the old field. The risky step happens last, never first.

QUALITY GATES

Validation gates that run before production

GATE 1

Static analysis

Migration linter flags locking operations, missing indexes, unsafe type changes.

GATE 2

Dry-run on a clone

Apply against a production-sized snapshot; measure duration and lock time.

GATE 3

Automated test suite

App tests run against the new schema — old and new code paths both pass.

GATE 4

Rollback rehearsal

Down-migration (or forward-fix plan) is written, reviewed, and exercised.

GATE 5

Human approval

An engineer reviews blast radius and signs off — the gate machines can't own.

In practice: SQLFluff or a migration linter for static analysis · a production-sized clone for the dry-run

OBSERVABILITY

Watch the right signals — during and after

BEFORE · Baseline captured	**DURING** · Live migration window	**AFTER** · Bake & verify

Replication lag

Replicas falling behind during heavy writes.

Lock & wait time

Blocked sessions and lock queue depth.

Query latency

p95/p99 on the hottest paths, not averages.

Error rate

App-side errors tied to the new schema.

Backfill progress

Rows migrated / remaining + ETA.

Saturation

CPU, IOPS, connections vs. headroom.

In practice: OpenTelemetry to instrument · Prometheus + Grafana or Datadog to watch — tagged with the migration ID

Where AI genuinely helps

Review SQL & migration diffs — Surfaces missing indexes, risky type casts, and locking DDL for a human to confirm.

Summarize risk — Turns a dense diff into a plain-language 'what could go wrong' brief for reviewers.

Generate test cases — Proposes edge-case and data-integrity tests an engineer then curates and owns.

Draft runbooks — First-pass deploy + rollback steps that the team edits with real production context.

Support post-incident analysis — Correlates timelines and logs to accelerate — not replace — human root-cause work.

PROMPT PATTERN

“Review this migration for locking risk, data-loss risk, rollback complexity, and missing validation. ***Do not approve it.*** *List only risks and questions for a human reviewer.”*

Where AI can be dangerous

No production context — It can't see live data volume, traffic shape, or replica topology — so its 'safe' isn't your safe.

Misreads data dependencies — Hidden FKs, triggers, and downstream consumers are invisible in the diff it's given.

Hallucinated rollback advice — Confidently proposes a 'revert' that would itself destroy or corrupt data.

Misses business impact — Treats a billing-table change like any other; can't weigh revenue or compliance risk.

Unchecked decision-maker — Auto-approving or auto-applying removes the accountable human at the worst moment.

HARD RULE

AI must never auto-approve or auto-apply a production migration. A human owns the execute.

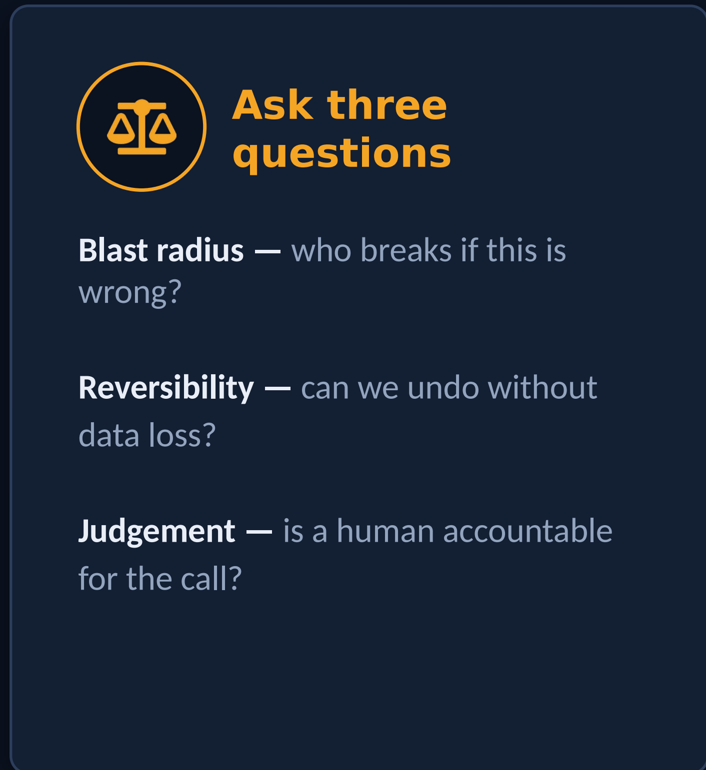
DECISION FRAMEWORK

Blast radius × reversibility × human judgement

Low blast · Hard to reverse AI may draft; engineer verifies & owns.	**High blast · Hard to reverse** Human-led. AI advisory only. Rehearse.
Low blast · Reversible Safe to automate with AI assist + checks.	**High blast · Reversible** Staged rollout; human approves the gate.

REVERSIBILITY: ◀ EASIER / HARDER ▶

◀ LOWER — BLAST RADIUS — HIGHER ▶



MAKE IT REAL

Practical implementation checklist

- Adopt expand-contract as the default migration shape
- Add a migration linter to CI — fail on locking DDL
- Dry-run every migration on a production-sized clone
- Require a written, reviewed rollback / forward-fix plan
- Gate production behind explicit human approval
- Keep environments in sync — detect drift automatically
- Instrument lag, locks, latency & errors per migration
- Use AI for review, summaries, tests, runbooks, RCA
- Never let AI auto-approve or auto-apply to production
- Make the accountable engineer explicit on every change

THE TAKEAWAY

AI assists.

Engineers own production.

Use it for review, documentation, risk detection, and incident analysis. Keep it out of the production decision. The pipeline makes migrations safe; the human makes them accountable.

- Expand-contract by default
- Automate the gates
- AI advises — never decides

Ahmed Tariq

Cloud DevOps Engineer · Conf42 Database DevOps 2026 · Thank you — questions welcome

CONF42