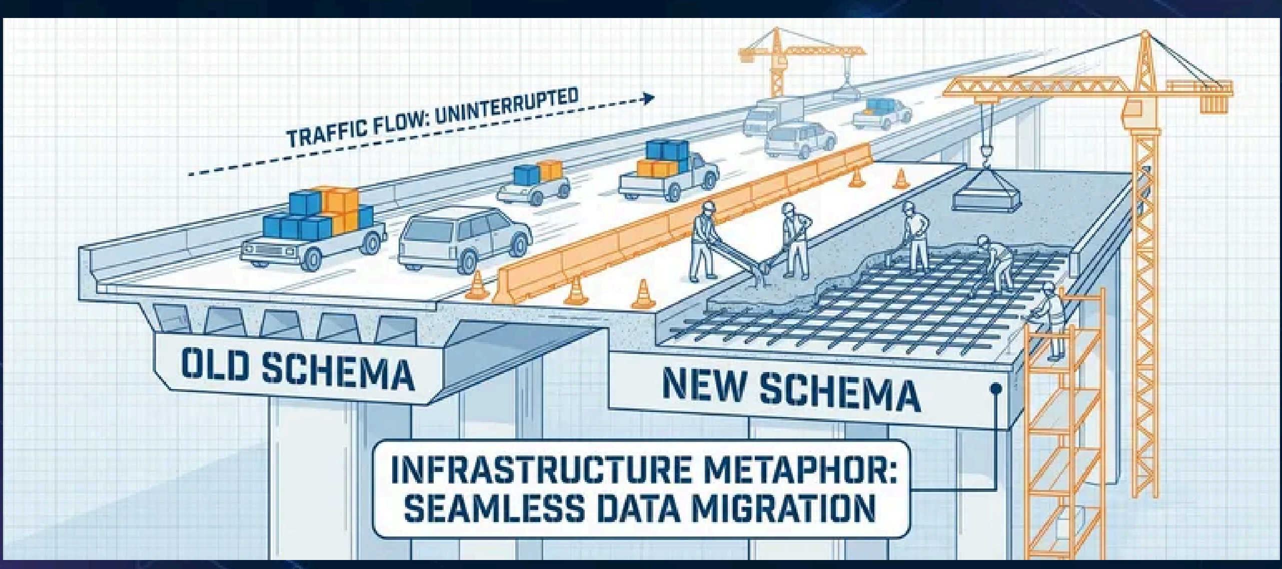
Database Migrations in CD Pipelines



Published on September 4, 2022

Table of Contents

Why Migrations Break Deployments ... 3
- The Lock Problem ... 3
- The Compatibility Window Problem ... 5
- Common Migration Failures ... 6

The Expand-Contract Pattern ... 7
- Pattern Overview ... 7
- Implementation Example ... 8

Safe Migration Patterns ... 11
- Adding Columns ... 11
- Removing Columns ... 12
- Renaming Columns ... 13
- Changing Column Types ... 13
- Adding Indexes ... 14
- Adding Constraints ... 15

Live Schema Change Tools ... 16
- PostgreSQL Tools ... 16
- MySQL Tools ... 17
- Using gh-ost ... 18

CI/CD Pipeline Integration ... 19
- Migration Pipeline Design ... 19
- Separating Migration and Application Deploys ... 20
- Migration Safety Checks ... 21
- Rollback Strategy ... 23

Migration Framework Configuration ... 24
- Two Paradigms ... 24
- Greenfield Projects ... 26

Monitoring Migrations ... 26
- What to Watch During Migrations ... 26
- Pre-Migration Checklist ... 27

Conclusion ... 28

Application code can roll back in seconds. Schema changes can take hours to reverse — if they're reversible at all. That asymmetry makes database migrations the hardest part of continuous deployment.

The core tension: CD assumes stateless, independent deployments. Databases are stateful and shared. Your application runs across multiple pods that can be replaced atomically, but every instance talks to the same database. A migration that takes 200ms on your development dataset can lock tables for four minutes against 50 million production rows. During that window, writes queue up, connection pools exhaust, timeouts cascade, and users see errors.

I've watched teams debate mid-incident: do we wait out the lock, or roll back and make things worse? Neither option is good once you're in that situation. The answer is to never get there — to treat migrations as a distinct deployment concern with their own safety patterns, rollback strategies, and monitoring.

> **WARNING**
>
> The biggest migration failures happen when teams treat schema changes like application code. A code deploy that fails just rolls back to the previous version. A migration that fails might leave your database in a state that neither the old code nor the new code can handle.

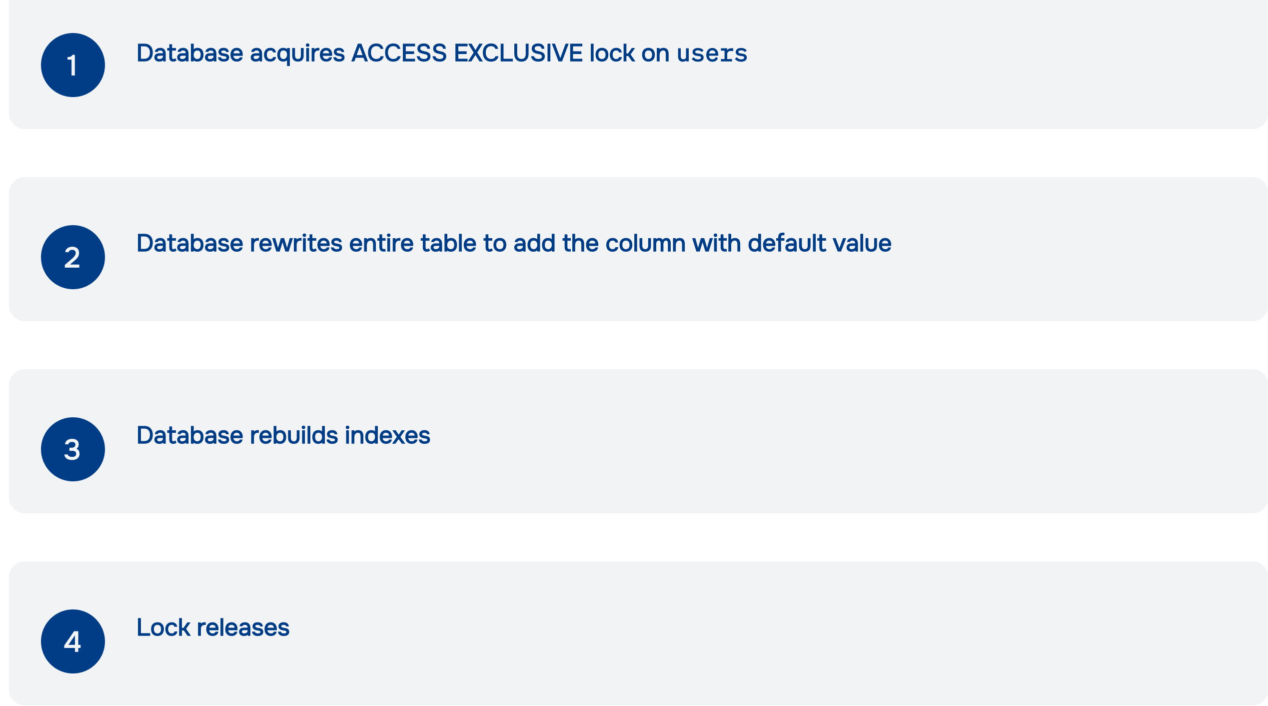
This article covers why migrations fail, how to decompose risky changes into safe ones, and how to build pipelines that make zero-downtime the default.

Why Migrations Break Deployments

The Lock Problem

Most DDL (Data Definition Language) operations acquire exclusive locks. While you hold that lock, every other query on that table waits — reads, writes, everything. On a small table, nobody notices. On a table with tens of millions of rows, the lock can persist for minutes while the database rewrites data or rebuilds indexes.

Here's what happens when you run `ALTER TABLE users ADD COLUMN status VARCHAR(20) DEFAULT 'active'` on older PostgreSQL (pre-11):



During steps 1-3, every application thread trying to touch that table blocks. Connection pools fill up with waiting queries. Those queries eventually timeout. The timeouts cascade through your application as dependent operations fail. Users see errors.

Modern PostgreSQL (v11+) made some operations instant — adding a nullable column or a column with a default no longer rewrites the table. But plenty of operations still require rewrites: changing column types, adding NOT NULL constraints to existing columns, certain index operations. MySQL has similar behavior without online schema change tools.

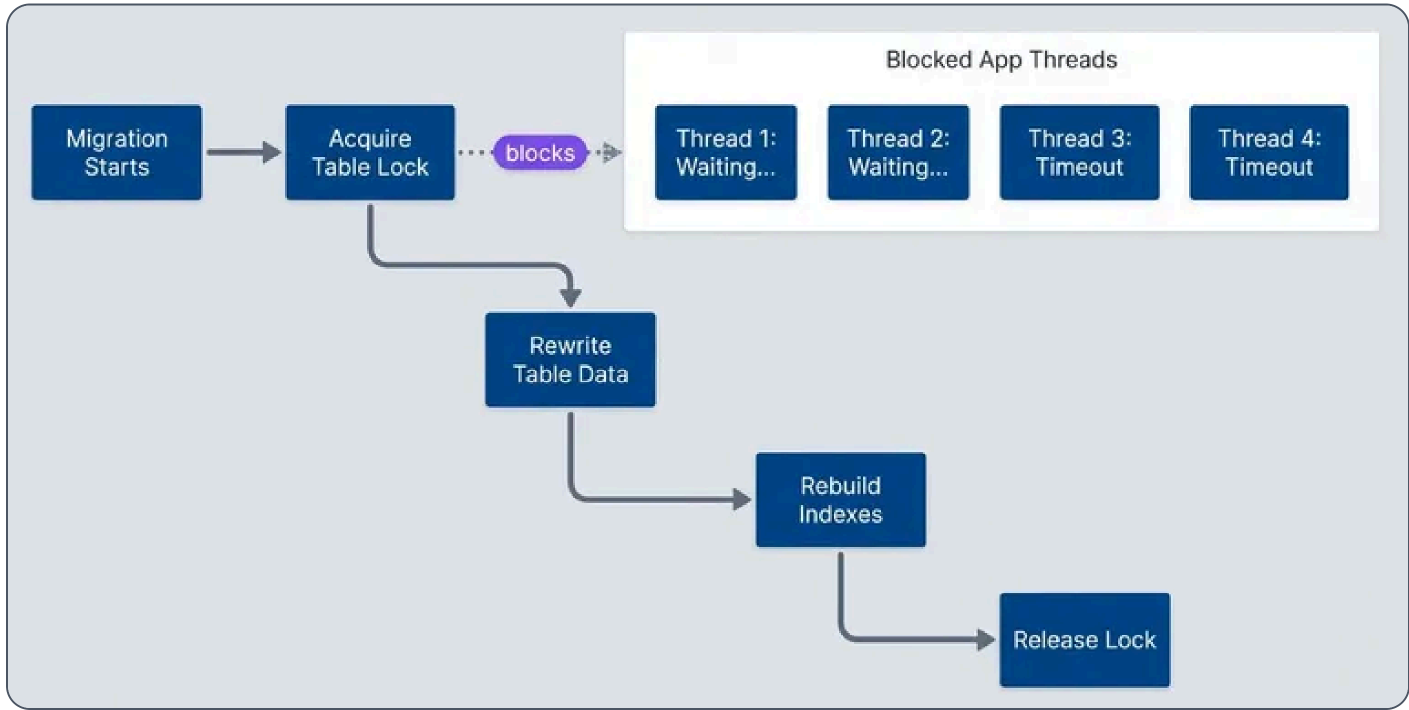


Table lock during migration blocks all application threads.

The Compatibility Window Problem

Rolling deployments mean old and new code run simultaneously. During a typical Kubernetes deployment, you might have 15 minutes where half your pods run v1 and half run v2. Your schema needs to work with both versions throughout that window.

Database | Old Code (v1) | New Code (v2)

Rolling deployment starts

SELECT email FROM users

Both versions running

SELECT email, phone FROM users

If phone column doesn't exist yet...

ERROR: column 'phone' does not exist

If phone column added but old code still running...

INSERT INTO users (email) VALUES (...)

phone is NULL - is that OK?

Database | Old Code (v1) | New Code (v2)

Compatibility window during rolling deployment.

The timing creates a catch-22:

- Add column after deploying new code: new code errors because column doesn't exist
- Add column_before_old code terminates: old code might fail if it doesn't expect the column
- Remove column while old code still runs: old code errors immediately

The solution is that every schema change must be backward-compatible. The schema after your migration must work with both the old application version and the new one. This constraint drives most of the patterns in this article.

> **INFO**
>
> The compatibility window can last from seconds (blue-green deployments with instant cutover) to hours (canary deployments with gradual traffic shift). Design your migrations for the longest window you might encounter.

Common Migration Failures

These are the failure modes I see repeatedly:

#	Failure Mode	Cause	Symptom	Prevention
1	Table lock timeout	Long-running ALTER	Application errors spike	Non-locking DDL tools (gh-ost, pt-osc)
2	Column not found	Code deployed before migration	500 errors on new pods	Deploy migration first
3	NOT NULL violation	Old code doesn't provide new column	Insert failures	Add column as nullable first
4	Foreign key violation	Data exists that violates new constraint	Migration fails	Clean data before constraint
5	Index creation timeout	Creating index on large table	Lock timeout	CREATE INDEX CONCURRENTLY
6	Rollback impossible	Destructive migration ran	Can't undo	Always write reversible migrations

Common migration failures and their prevention.

The theme across all of these: migrations fail when they assume atomic, instantaneous changes in a system that's actually gradual and concurrent. The fix is always some form of phased rollout — expand first, then contract.

The Expand-Contract Pattern

Pattern Overview

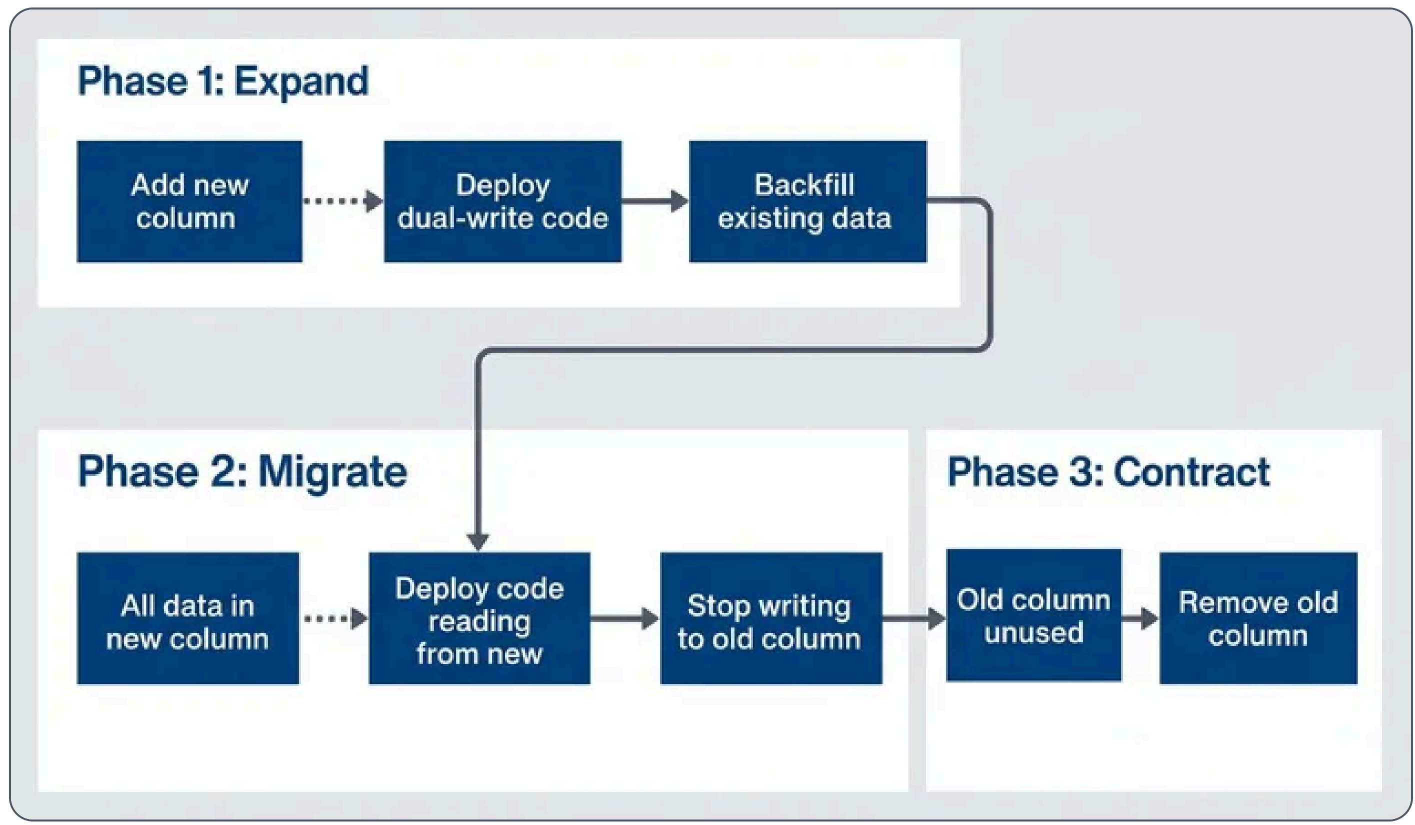
The expand-contract pattern (sometimes called parallel change) is the fundamental technique for zero-downtime schema changes. Instead of making one risky change, you decompose it into multiple safe changes that maintain backward compatibility at every step.

The pattern has three phases:

1. **Expand:**
Add the new structure alongside the old. Both exist simultaneously. Deploy application code that writes to both structures but still reads from the old one.

2. **Migrate:**
Backfill existing data into the new structure. Once complete, deploy code that reads from the new structure (still writing to both for safety).

3. **Contract:**
Remove the old structure. Deploy code that only uses the new structure, then drop the old column/table.



Expand-contract pattern phases.

Let's trace through a concrete example: renaming a column from `email` to `email_address`. A direct `ALTER TABLE RENAME COLUMN` would break every query referencing the old name instantly. Here's the expand-contract approach:

Week 1 (Expand):

Run a migration to add the `emailaddress` column. Deploy code that writes to both columns on every insert and update, but still reads from `email`. At this point, new rows have both columns populated; old rows only have `email`.

Week 2 (Migrate):

Run a backfill job to copy `email` values to `email_address` for existing rows. Once complete, deploy code that reads from `email_address` instead of `email`. Keep writing to both columns — old pods might still be running.

Week 3 (Contract):

After all old pods are gone and you've verified nothing reads `email`, deploy code that only uses `email_address`. Then run a migration to drop the `email` column.

Total time: 2-3 weeks for a “simple” rename. This is why you think carefully before renaming columns in production.

Implementation Example

Here’s the concrete code for each phase. First, the expand migration:

expand-phase-migration.sql

```sql
-- Migration 001: Expand - Add new column
-- Safe: Adding nullable column is instant in PostgreSQL 11+

ALTER TABLE users ADD COLUMN email_address VARCHAR(255);

-- Create index concurrently (doesn't lock table)
CREATE INDEX CONCURRENTLY idx_users_email_address ON users(email_address);
```

Phase 1 migration adding the new column.

expand-phase-code.ts

```typescript
// Application code during expand phase
// Writes to BOTH columns, reads from OLD column

class UserRepository {
  async createUser(data: CreateUserInput): Promise<User> {
    const result = await db.query(`
      INSERT INTO users (name, email, email_address)
      VALUES ($1, $2, $2)
      RETURNING *
    `, [data.name, data.email]);

    return this.mapToUser(result.rows[0]);
  }

  async updateEmail(userId: string, newEmail: string): Promise<void> {
```

```
16      // Write to both columns
17      await db.query(`
18        UPDATE users
19        SET email = $1, email_address = $1
20        WHERE id = $2
21      `, [newEmail, userId]);
22    }
23
24    async getUser(userId: string): Promise<User> {
25      // Still read from old column during expand
26      const result = await db.query(`
27        SELECT id, name, email FROM users WHERE id = $1
28      `, [userId]);
29
30      return this.mapToUser(result.rows[0]);
31    }
32  }
```

Application code during expand phase — dual writes.

migrate-phase-backfill.sql

```
1   -- Migration 002: Backfill existing data
2   -- Run as background job, not blocking migration
3
4   -- For small tables:
5   UPDATE users SET email_address = email WHERE email_address IS NULL;
6
7   -- For large tables (millions of rows), batch it:
8   DO $$
9   DECLARE
10    batch_size INT := 10000;
11    rows_updated INT;
12  BEGIN
13    LOOP
14      UPDATE users
15      SET email_address = email
16      WHERE id IN (
17        SELECT id FROM users
18        WHERE email_address IS NULL
19        LIMIT batch_size
```


Webstack Builders
1032 E. Brandon Boulevard, Suite 5230
Brandon, FL 33511

support@webstackbuilders.com
Toll Free (888) 987-1881
Local (302) 608-6864

```
20        );
21
22        GET DIAGNOSTICS rows_updated = ROW_COUNT;
23        EXIT WHEN rows_updated = 0;
24
25        -- Small sleep to reduce load
26        PERFORM pg_sleep(0.1);
27
28        RAISE NOTICE 'Updated % rows', rows_updated;
29      END LOOP;
30    END $$;
```

Backfill migration for large tables using batches.

contract-phase-migration.sql

```sql
-- Migration 003: Contract - Remove old column
-- Only run AFTER all application code uses email_address

-- First, verify no code uses old column (monitoring/logs)
-- Then drop:
ALTER TABLE users DROP COLUMN email;
```

Phase 3 migration removing the old column.

Some teams use database views as an additional compatibility layer during migrations. A view can expose the old column name while the underlying table uses the new name, letting legacy code work without modification. Feature flags at the application layer can also control which schema path code takes, though this adds complexity and should be cleaned up once the migration completes.

> ✓ **SUCCESS**
>
> The expand - contract pattern is slower than a direct rename, but it guarantees zero downtime. Each phase can be deployed independently and rolled back without data loss.

Safe Migration Patterns

Now that we've covered the fundamental pattern, here's a quick reference for common schema changes. Each pattern follows expand - contract principles, but some operations have database-specific shortcuts that are safe to use directly.

Adding Columns

Adding columns is the most common migration, and modern databases have made the safe path easy:

adding-columns-safely.sql

```sql
-- ✅ SAFE: Adding nullable column (PostgreSQL 11+)
ALTER TABLE users ADD COLUMN phone VARCHAR(20);
-- Instant, no table rewrite

-- ✅ SAFE: Adding column with default (PostgreSQL 11+)
ALTER TABLE users ADD COLUMN status VARCHAR(20) DEFAULT 'active';
-- Instant, default stored in catalog not data

-- ❌ DANGEROUS: Adding NOT NULL without default
ALTER TABLE users ADD COLUMN phone VARCHAR(20) NOT NULL;
-- Fails if table has data, or requires table rewrite

-- ✅ SAFE: Add nullable first, then add constraint
ALTER TABLE users ADD COLUMN phone VARCHAR(20);
UPDATE users SET phone = 'unknown' WHERE phone IS NULL;
ALTER TABLE users ALTER COLUMN phone SET NOT NULL;
```

Safe patterns for adding columns.

The key insight: PostgreSQL 11+ and recent MySQL versions store defaults in the catalog rather than rewriting every row. Adding a nullable column or a column with a default is now instant regardless of table size. But adding NOT NULL without a default still requires a table scan.

Removing Columns

Removal is where teams most often skip the safety steps. The column seems unused, so why not just drop it?

removing-columns-safely.sql

```sql
-- ❌ DANGEROUS: Direct drop while code might use it
ALTER TABLE users DROP COLUMN legacy_field;

-- ✅ SAFE: Expand-contract removal

-- Step 1: Stop code from using the column (code deploy)
-- Step 2: Wait for all old pods to terminate
-- Step 3: Verify column is unused (query logs)
-- Step 4: Drop column
ALTER TABLE users DROP COLUMN legacy_field;
```

Safe pattern for removing columns.

The “verify column is unused” step is critical. I’ve seen teams drop columns that were still referenced by scheduled jobs, reporting queries, or that one microservice nobody remembered existed. Query your database logs or enable query sampling before the drop.

Renaming Columns

There’s no safe way to rename a column in one step. The moment the name changes, every query using the old name fails.

renaming-columns-safely.sql

```sql
-- ❌ DANGEROUS: Direct rename
ALTER TABLE users RENAME COLUMN email TO email_address;
-- Breaks all code instantly

-- ✅ SAFE: Expand-contract rename (3 deployments)

-- Deploy 1: Add new column
ALTER TABLE users ADD COLUMN email_address VARCHAR(255);

-- Deploy 2: Application writes to both, reads from old
-- Backfill: UPDATE users SET email_address = email WHERE email_address IS NULL;

-- Deploy 3: Application reads from new
-- After verification: ALTER TABLE users DROP COLUMN email;
```

Safe pattern for renaming columns.

Yes, this means a “simple” rename takes three separate deployments spread over days or weeks. That’s the cost of zero-downtime schema changes. Factor this into your planning when naming columns — getting it right the first time saves significant effort later.

Changing Column Types

Type changes combine the risks of renames (code compatibility) with the risks of rewrites (locking). They’re the highest-risk common migration.

changing-types-safely.sql

```sql
-- ❌ DANGEROUS: Direct type change
ALTER TABLE orders ALTER COLUMN amount TYPE DECIMAL(10,2);
-- May require table rewrite, blocks writes

-- ✅ SAFE: Expand-contract type change

-- Step 1: Add new column with new type
ALTER TABLE orders ADD COLUMN amount_decimal DECIMAL(10,2);

-- Step 2: Dual-write in application
-- Step 3: Backfill with conversion
UPDATE orders SET amount_decimal = amount::DECIMAL(10,2) WHERE amount_decimal IS NULL;

-- Step 4: Switch reads to new column
-- Step 5: Drop old column
ALTER TABLE orders DROP COLUMN amount;
ALTER TABLE orders RENAME COLUMN amount_decimal TO amount;
```

Safe pattern for changing column types.

> ⚠ **WARNING**
>
> The backfill step can fail if data doesn’t convert cleanly. Test the conversion query against production data (or a recent copy) before running the migration. A failed conversion mid-backfill leaves you with partially migrated data.

Adding Indexes

Index creation on large tables can lock writes for minutes or hours. Every database has a way to avoid this:

adding-indexes-safely.sql

```sql
-- ❌ DANGEROUS: Standard index creation
CREATE INDEX idx_users_email ON users(email);
-- Locks table for writes during creation

-- ✅ SAFE: Concurrent index creation (PostgreSQL)
CREATE INDEX CONCURRENTLY idx_users_email ON users(email);
-- Doesn't lock table, but takes longer
-- Note: Can't run inside a transaction

-- MySQL: Use pt-online-schema-change or gh-ost
-- pt-online-schema-change --alter "ADD INDEX idx_email (email)" D=mydb,t=users

-- Check index creation progress (PostgreSQL)
SELECT
  phase,
  round(100.0 * blocks_done / nullif(blocks_total, 0), 1) AS "% done",
  round(tuples_done / 1000000.0, 1) AS "millions of tuples done"
FROM pg_stat_progress_create_index;
```

Safe index creation patterns.

The tradeoff with concurrent index creation: it takes longer and uses more resources because it has to handle concurrent writes during the build. On a busy table, expect it to take 2-3x longer than a locking index creation.

Adding Constraints

Foreign key constraints are particularly tricky because they lock **both** tables and require scanning all existing data. The NOT VALID approach separates these concerns:

adding-constraints-safely.sql

```sql
-- ❌ DANGEROUS: Adding constraint that might fail
ALTER TABLE orders ADD CONSTRAINT fk_user
```

```
3     FOREIGN KEY (user_id) REFERENCES users(id);
4   -- Fails if orphan data exists, locks both tables
5
6   -- ✅ SAFE: Validate data first, add constraint with NOT VALID
7
8   -- Step 1: Find violations
9   SELECT o.id, o.user_id
10  FROM orders o
11  LEFT JOIN users u ON o.user_id = u.id
12  WHERE u.id IS NULL;
13
14  -- Step 2: Fix or delete violations
15  DELETE FROM orders WHERE user_id NOT IN (SELECT id FROM users);
16
17  -- Step 3: Add constraint without validation (instant)
18  ALTER TABLE orders ADD CONSTRAINT fk_user
19    FOREIGN KEY (user_id) REFERENCES users(id) NOT VALID;
20
21  -- Step 4: Validate constraint separately (can be slow but doesn't block)
22  ALTER TABLE orders VALIDATE CONSTRAINT fk_user;
```

Safe pattern for adding foreign key constraints.

The NOT VALID constraint enforces referential integrity for new rows immediately while allowing you to validate existing data as a separate, non-blocking operation.

Live Schema Change Tools

When native database features aren't enough — particularly for MySQL or complex PostgreSQL alterations — dedicated schema migration tools let you modify large tables without locking. These tools work by creating a shadow copy of your table, applying the schema change to the copy, then atomically swapping the tables once the copy catches up with production writes.

PostgreSQL Tools

PostgreSQL has made significant progress with native non-blocking DDL (Data Definition Language) support. Most common operations no longer require external tools:

Operation	PostgreSQL v11+ Behavior	External Tool Needed?
Add `nullable` column	Instant (metadata only)	No
Add column with default	Instant (default in catalog)	No
Create index	`CONCURRENTLY` option available	No
Add foreign key	`NOT VALID` then `VALIDATE`	No
Change column type	Requires table rewrite	Sometimes
Add `NOT NULL` to existing column	Requires scan	Sometimes

Native PostgreSQL DDL operations and external tool requirements

For operations that still require ACCESS EXCLUSIVE locks, **pgroll** from Xata automates the expand-contract pattern. It creates triggers to dual-write during migrations and provides CLI commands for progressive rollout. The tradeoff: it's a newer tool with a smaller community than the MySQL alternatives.

MySQL Tools

MySQL's native DDL (Data Definition Language) support is more limited than PostgreSQL's, making external tools essential for large tables. Two tools dominate:

Tool	Vendor	Approach	Key Advantage	Key Limitation
pt-online-schema-change	Percona	Shadow table + triggers	Battle-tested, works with replication	Triggers add write overhead
gh-ost	GitHub	Shadow table + binlog streaming	No triggers, pausable/resumable	Requires binlog access

Comparison of external MySQL schema migration tools

Both tools follow the same general strategy:

- Create an empty shadow table with the new schema
- Copy existing rows in chunks (throttled to avoid overloading the database)
- Capture ongoing writes (via triggers or binlog) and replay them to the shadow table
- Once caught up, atomically rename tables to swap old and new

The difference is in step 3: pt-online-schema-change uses database triggers to capture writes, while gh-ost streams from the MySQL binlog. The binlog approach has less overhead but requires more infrastructure access.

Using gh-ost

GitHub developed gh-ost (GitHub Online Schema Transmogrifier) after running into limitations with pt-online-schema-change at scale. The tool has become the de facto standard for MySQL schema changes at companies running large databases — it's battle-tested against tables with billions of rows.

What makes gh-ost particularly useful in production: you can pause and resume migrations, throttle based on replica lag or server load, and even test the migration without actually cutting over. The `--dry-run` and `--test-on-replica` modes let you validate that a migration will complete successfully before touching your primary.

Here's a typical gh-ost invocation for adding a column:

gh-ost-example.sh

```
#!/bin/bash
# Add column to large MySQL table without locking

gh-ost \
  --host=mysql-primary.example.com \
  --database=myapp \
  --table=users \
  --alter="ADD COLUMN phone VARCHAR(20)" \
  --allow-on-master \
  --chunk-size=1000 \
  --max-load=Threads_running=25 \
  --critical-load=Threads_running=100 \
  --execute

```

```
15   # Options explained:
16   # --chunk-size: Rows to copy per iteration
17   # --max-load: Pause if this load threshold exceeded
18   # --critical-load: Abort if this load threshold exceeded
19   # --execute: Actually run (vs --dry-run)
20
21   # Monitor progress:
22   # gh-ost creates a socket file for control
23   echo "status" | nc -U /tmp/gh-ost.myapp.users.sock
```

gh-ost example for adding a column to a large MySQL table.

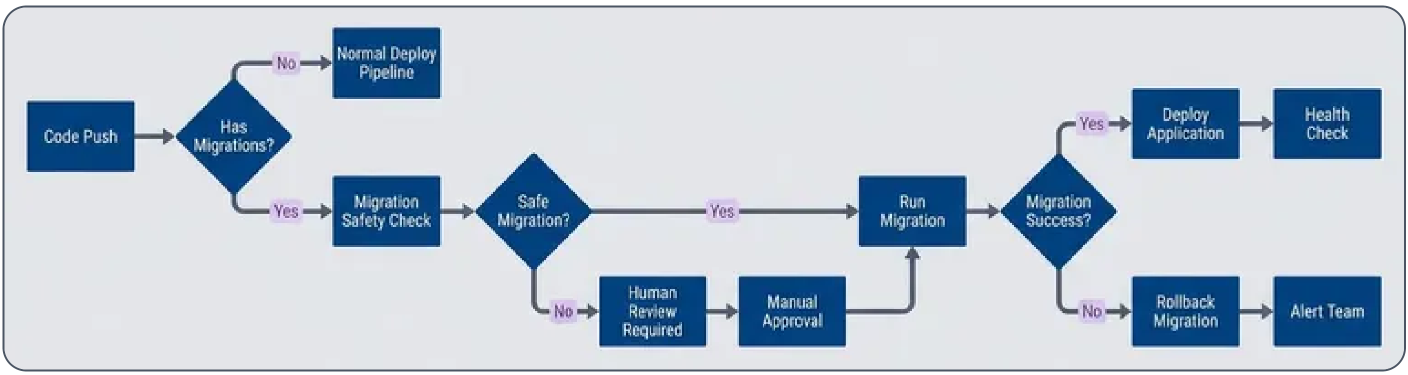
The `--max-load` and `--critical-load` flags are where gh-ost shines. If your database starts struggling (threads piling up, replica lag increasing), gh-ost automatically backs off. If things get critical, it aborts entirely rather than making the situation worse. This self-throttling behavior is what makes it safe to run during business hours on production databases.

CI/CD Pipeline Integration

The patterns we've discussed work manually, but the real value comes from encoding them into your deployment pipeline. A well-designed pipeline makes safe migrations the default path and forces dangerous ones through review gates.

Migration Pipeline Design

The key insight is that migrations and application deploys are **different operations** with different risk profiles and rollback characteristics. Your pipeline should treat them that way.



Migration-aware deployment pipeline.

The flow handles three scenarios: code-only changes skip migration steps entirely, safe migrations run automatically before application deployment, and potentially dangerous migrations pause for human review. This balances velocity (most deploys are code-only) with safety (risky changes get scrutiny).

Separating Migration and Application Deploys

In GitHub Actions, you can detect whether a PR includes migrations and route accordingly. The `dorny/paths-filter` action watches for changes in your migrations directory and sets output variables that control which jobs run.

.github/workflows/deploy.yaml

```yaml
name: Deploy

on:
  push:
    branches: [main]

jobs:
  detect-changes:
    runs-on: ubuntu-latest
    outputs:
      has_migrations: ${{ steps.changes.outputs.migrations }}
      has_code: ${{ steps.changes.outputs.code }}
    steps:
      - uses: actions/checkout@v4
      - uses: dorny/paths-filter@v2
        id: changes
        with:
          filters: |
            migrations:
              - 'migrations/**'
            code:
              - 'src/**'
              - 'package.json'

  run-migrations:
    needs: detect-changes
```


Webstack Builders
1032 E. Brandon Boulevard, Suite 5230
Brandon, FL 33511

support@webstackbuilders.com
Toll Free (888) 987-1881
Local (302) 608-6864

```
27        if: needs.detect-changes.outputs.has_migrations == 'true'
28        runs-on: ubuntu-latest
29        steps:
30          - uses: actions/checkout@v4
31          - name: Safety check migrations
32            run: python ./scripts/check-migration-safety.py
33          - name: Run migrations
34            run: npm run migrate
35            env:
36              DATABASE_URL: ${{ secrets.DATABASE_URL }}
37          - name: Verify migration
38            run: npm run migrate:verify
39
40      deploy-application:
41        needs: [detect-changes, run-migrations]
42        if: |
43          always() &&
44          needs.detect-changes.outputs.has_code == 'true' &&
45          (needs.run-migrations.result == 'success' || needs.run-migrations.result ==
    'skipped')
46        runs-on: ubuntu-latest
47        steps:
48          - name: Deploy to Kubernetes
49            run: kubectl apply -f k8s/
```

GitHub Actions workflow that separates migration and application deploys.

The `deploy-application` job's conditional is the interesting part: it runs if there's code to deploy **and** either migrations succeeded or there weren't any migrations to run. This ensures we never deploy application code if a migration failed.

Migration Safety Checks

Automated safety checks can catch the most common mistakes before they reach production. This script scans migration files for patterns that typically require the expand-contract pattern:

check-migration-safety.py

```
1    #!/usr/bin/env python3
```

```python
# Use this in CI to gate migrations that need human review.
from __future__ import annotations

from pathlib import Path
import re
import sys

MIGRATION_DIR = Path("./migrations")
UNSAFE_PATTERNS = [
  r"DROP\s+COLUMN",
  r"DROP\s+TABLE",
  r"RENAME\s+COLUMN",
  r"ALTER\s+.*TYPE",
  r"NOT\s+NULL",
  r"DROP\s+INDEX",
  r"TRUNCATE",
]

def find_sql_files(directory: Path) -> list[Path]:
  return [path for path in directory.rglob("*.sql") if path.is_file()]

def file_has_pattern(path: Path, pattern: re.Pattern[str]) -> bool:
  try:
    content = path.read_text(encoding="utf-8", errors="ignore")
  except OSError:
    return False
  return bool(pattern.search(content))

def main() -> int:
  sql_files = find_sql_files(MIGRATION_DIR)
  if not sql_files:
    print("✅ No migration files found")
    return 0

  needs_review = False
  for raw_pattern in UNSAFE_PATTERNS:
    pattern = re.compile(raw_pattern, re.IGNORECASE)
    for sql_file in sql_files:
      if file_has_pattern(sql_file, pattern):
        print(f"⚠️  Found potentially unsafe pattern: {raw_pattern} in {sql_file}")
        needs_review = True
        break

```


Webstack Builders
1032 E. Brandon Boulevard, Suite 5230
Brandon, FL 33511

support@webstackbuilders.com
Toll Free (888) 987-1881
Local (302) 608-6864

```
45      if needs_review:
46        print("❌ Migration requires human review before deployment")
47        return 1
48
49      print("✅ Migration passes automated safety checks")
50      return 0
51
52  if __name__ == "__main__":
53      sys.exit(main())
```

Python script to flag potentially dangerous migration patterns.

The script isn't trying to be smart — it just flags keywords that warrant a second look. A `DROP COLUMN` might be the final step of a well-executed expand-contract migration, completely safe and intentional. Or it might be someone who didn't realize the column was still in use. The script can't tell the difference, but it can force the conversation.

WARNING

Automated checks catch obvious issues, but they can't understand context. Human review remains essential for destructive operations — the script's job is to ensure that review happens.

Rollback Strategy

Not all migrations can be rolled back, and knowing the difference ahead of time determines your incident response options.

Reversible migrations

Have straightforward inverses: `ADD COLUMN` reverses with `DROP COLUMN`, `CREATE INDEX` reverses with `DROP INDEX`, and `ADD CONSTRAINT` reverses with `DROP CONSTRAINT`. Most migration frameworks generate these down migrations automatically. When something goes wrong, you run a rollback command with the project's CLI tool and you're back to the previous state.

Webstack Builders
1032 E. Brandon Boulevard, Suite 5230
Brandon, FL 33511

support@webstackbuilders.com
Toll Free (888) 987-1881
Local (302) 608-6864

Irreversible migrations

The dangerous ones. Once you `DROP COLUMN`, that data is gone. Once you `TRUNCATE`, there's no undo. For these operations, your safety net is backups — take one immediately before the migration, verify it's restorable, and document the restoration steps. If you're running a backfill that transforms data, keep the original data until you've verified the migration succeeded.

Forward fixes

Your last resort when rollback isn't an option. Maybe the migration already ran in production. Maybe rolling back would lose data you can't afford to lose. Maybe the rollback would take longer than fixing the problem. In these cases, you deploy code that handles both the old and new schema states, then run a corrective migration to get to a clean state. It's messy, but sometimes it's the only path forward.

Migration Framework Configuration

Most full-stack frameworks and ORMs include migration tooling out of the box. They all solve the same fundamental problem — tracking which schema changes have been applied to which environments — but they approach it differently.

Two Paradigms

Schema-diff tools (Prisma, Entity Framework, Django)

These tools compare your model definitions against the current database state and generate migrations automatically. This is convenient for rapid development but requires careful review: the generated SQL might not follow safe migration patterns. A model change that looks like a rename might generate a destructive `DROP COLUMN` followed by `ADD COLUMN` instead of the expand-contract approach.

SQL-first tools (Flyway, Knex, raw migration files)

These tools put you in control of the exact SQL that runs. You write the migration manually, which means more work but also more predictability. There's no magic diff algorithm making decisions about your production database.

Most tools share a common CLI pattern: a command to generate a new migration file, a command to apply pending migrations, a command to check status, and optionally a command to rollback. The table below shows how this looks across popular frameworks.

Framework	Generate	Apply	Rollback	Notes
Rails	`rails generate migration AddPhone`	`rails db:migrate`	`rails db:rollback`	Wraps each migration in transaction, auto-rollback
Django	`python manage.py makemigrations`	`python manage.py migrate`	`python manage.py migrate app 0001`	Atomic by default, good dependency tracking
Laravel	`php artisan make:migration add_phone`	`php artisan migrate`	`php artisan migrate:rollback`	Transaction per migration, clean rollback
.NET EF Core	`PS> Add-Migration AddPhone`	`PS> Update-Database`	`PS> Update-Database PreviousMigration`	No transaction by default, manual control
RedwoodJS	`yarn rw prisma migrate dev`	`yarn rw prisma migrate deploy`	Manual	Uses Prisma; no auto-rollback generation
Knex.js	`npx knex migrate:make add_phone`	`npx knex migrate:latest`	`npx knex migrate:rollback`	Transaction optional, SQL-first control

Common migration CLI patterns across frameworks.

When evaluating migration tooling, look for **idempotency** (can you safely run the same migration twice?), **dry-run support** (can you preview what SQL will execute?), **transaction wrapping** (does a failed migration leave your database in a broken state?), and **rollback generation** (does the tool create down migrations automatically, or do you write them manually?).

> **INFO**
>
> Schema-diff tools are convenient but can generate unsafe migrations. Always review generated SQL before running against production, especially for operations that look like renames or type changes.

Greenfield Projects

If you're building an application without a full-stack framework — maybe a microservice, a CLI tool with persistent state, or an API built on a minimal framework — adding migration tooling early pays dividends. It's tempting to just run `ALTER TABLE` manually during early development, but that approach doesn't survive the first production deployment.

The investment is small: Flyway, Liquibase, or a language-specific tool like Knex takes an hour to set up. What you get back is a repeatable deployment process, a history of every schema change, and the ability to spin up fresh environments that match production. The alternative — a wiki page of SQL commands to run in order, or worse, tribal knowledge about what the schema "should" look like — becomes technical debt that compounds with every team member and every environment.

Monitoring Migrations

Migrations that work perfectly in staging can still cause problems in production. The difference is scale: a backfill that takes 2 seconds against 10,000 rows might take 20 minutes against 10 million. Monitoring tells you when a migration is taking longer than expected, when it's impacting application performance, or when you need to abort.

What to Watch During Migrations

Migration duration is your primary metric. Establish baselines in staging and alert if production takes significantly longer. A migration that usually completes in 30 seconds but is still running after 5 minutes deserves investigation.

Lock wait time reveals contention. In PostgreSQL, you can query `pg_stat_activity` to see queries waiting on locks:

check-lock-waits.sql

```sql
SELECT pid, now() - query_start AS duration, query
FROM pg_stat_activity
WHERE wait_event_type = 'Lock';
```

PostgreSQL query to identify queries waiting on locks.

If you see application queries piling up with multi-second waits, your migration is blocking production traffic. This is exactly the situation you're trying to avoid with the patterns in this article.

Application error rate should be part of your deployment dashboard regardless, but pay special attention during migrations. A spike in 500 errors or database connection timeouts immediately after a migration starts is a strong signal to abort.

Replication lag matters if you're running read replicas. Large migrations can cause replicas to fall behind, which means your application might read stale data or — worse — data that doesn't match the schema your code expects.

Pre-Migration Checklist

Before running any migration in production, verify the basics:

- **Staging verification:**
  The migration ran successfully in staging with production-like data volumes. The application works with the new schema. You've tested the rollback path.
- **Database health:**
  No long-running transactions that might conflict with your migration. Replication lag is minimal. Sufficient disk space exists (shadow-table tools like gh-ost temporarily double table size). A recent backup exists and you've verified it's restorable.
- **Timing and staffing:**
  You're not deploying during peak traffic. Team members are available to monitor the migration and respond to problems. No other deployments are in progress that might confuse incident response.
- **Rollback readiness:**
  The rollback script is tested. You know how long rollback will take. You've defined clear criteria for when to rollback versus wait it out.

Webstack Builders
1032 E. Brandon Boulevard, Suite 5230
Brandon, FL 33511

support@webstackbuilders.com
Toll Free (888) 987-1881
Local (302) 608-6864

WARNING

The checklist feels bureaucratic until it saves you. Migrations fail most often when someone skips a step because they're "just adding a column" or "it's a small table." The checklist exists precisely for those moments.

Conclusion

Database migrations don't have to be the scariest part of your deployment pipeline. The patterns here — expand-contract changes, live schema change tools, separated migration jobs, automated safety checks — transform migrations from high-risk events into routine operations.

✓ SUCCESS

Zero-downtime migrations aren't about clever tricks — they're about patience. The expand-contract pattern works because it decomposes one risky change into multiple safe changes. Accept that a "simple rename" takes three deployments and you'll never have a migration-induced outage.

The core principles: treat migrations as a distinct deployment concern with their own safety gates. Maintain backward compatibility so old and new code can coexist during rolling deployments. Use non-blocking DDL (Data Definition Language) operations wherever possible. Always have a rollback plan, even if that plan is "restore from backup." And test migrations against production-like data volumes before they reach production — a migration that works on 10,000 rows can behave very differently against 10 million.

Copyright © 2022 Webstack Builders, Inc.

The text, diagrams, and images in this work are licensed under CC BY-NC 4.0

All code samples in this article are licensed under the MIT License. Feel free to use, modify, and distribute them in any project.

https://www.webstackbuilders.com/articles/database-schema-migrations-continuous-deployment-zero-downtime

Webstack Builders
1032 E. Brandon Boulevard, Suite 5230
Brandon, FL 33511

support@webstackbuilders.com
Toll Free (888) 987-1881
Local (302) 608-6864