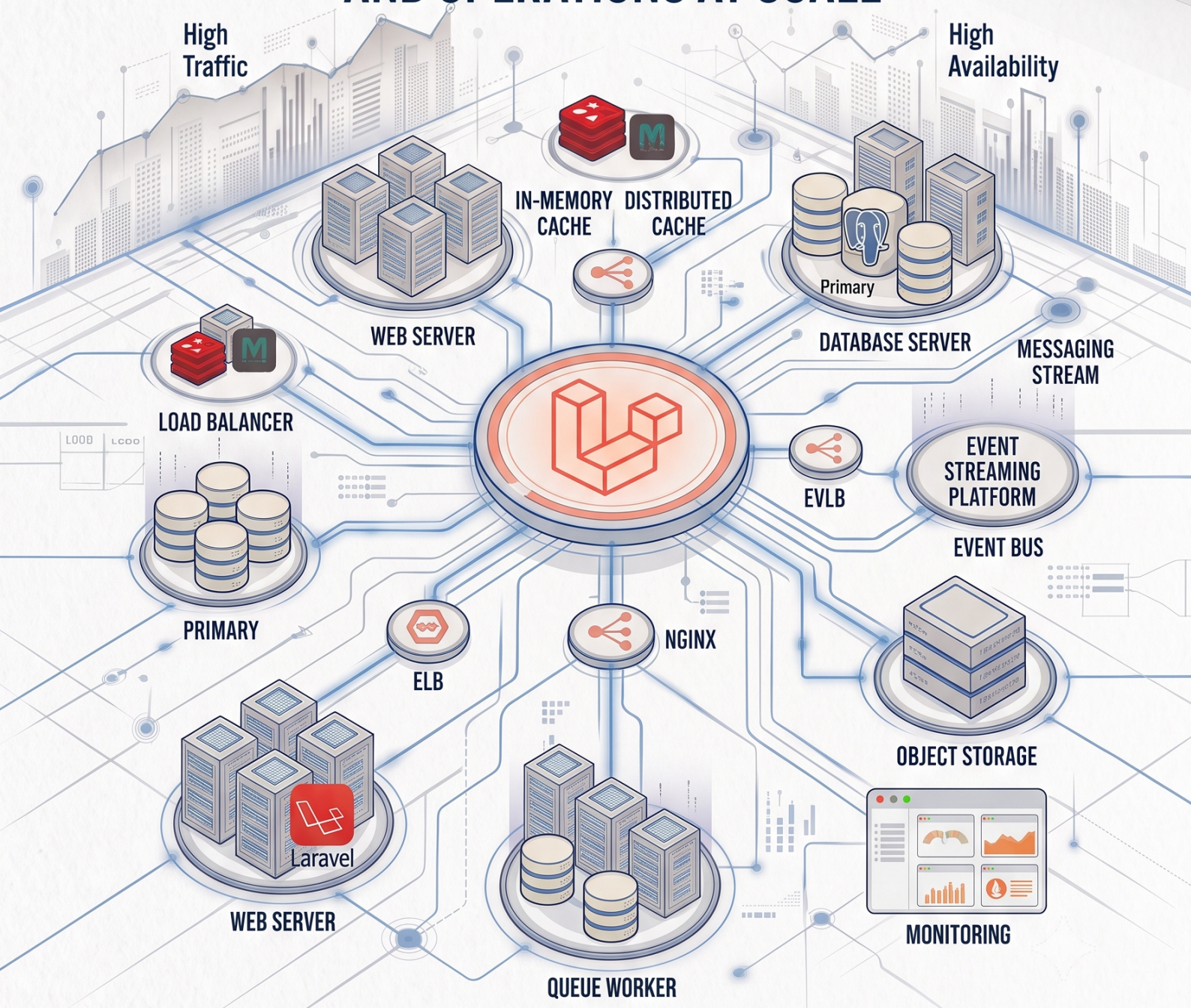
LARAVEL IN PRODUCTION

ARCHITECTURE, PERFORMANCE, AND OPERATIONS AT SCALE



NURUZZAMAN MILON

Laravel in Production

Architecture, Performance, and Operations at Scale

Nuruzzaman Milon

First published: July 2026
Cover art generated with Google Gemini.

Chapter 8 - Safe Schema Evolution at Scale

The migration that only failed in production

On Northfield, a fictional order-processing platform, a routine cleanup task renames a column: `orders.buyer_email` becomes `orders.customer_email`, to match naming used everywhere else in the domain. The migration is small, obviously correct, and reviewed without objection. In staging, against a ten-thousand-row table, it runs in forty milliseconds. In production, against a table with two hundred million rows, the `ALTER TABLE` takes an exclusive lock for twenty-five minutes. Every write to `orders` blocks behind it. The API starts returning timeouts, the on-call engineer gets paged, and the migration that "obviously" worked has to be killed mid-run, leaving the schema in an ambiguous state.

Nothing about the migration was wrong, syntactically. It was tested at the wrong scale, the same failure mode as Chapter 7, applied to schema changes instead of queries. This chapter covers the patterns that make schema changes safe regardless of table size: non-blocking DDL, a backward-compatible rename pattern, batched backfills, and - the part teams forget most often - tracking who else reads your schema before you assume a "safe" change is actually safe for everyone.

Why "fast in staging" doesn't transfer

The cost of a schema change is a function of data volume, not query complexity, and that cost varies by database engine and by the specific operation:

- Adding a nullable column with no default is metadata-only and near-instant on both MySQL (InnoDB, modern versions) and PostgreSQL, regardless of table size.
- Adding a column with a default value, or adding a `NOT NULL` constraint to an existing column, historically required rewriting every row to populate or validate the new value - a cost proportional to table size. Recent versions of both engines have narrowed this (PostgreSQL avoids a full rewrite for constant defaults since version 11; MySQL's behavior depends on the specific `ALGORITHM` used), but the safe assumption is: verify the specific operation against the specific engine version in use, on a copy of production-sized data, before trusting it in staging.

- Renaming a column, dropping a column, or changing a column's type can each range from instant metadata changes to full table rewrites, again depending on engine and version.

The rule that survives all of that engine-specific detail: never trust a migration's timing from a small staging database. Run it against a production-sized copy (a recent snapshot, restored somewhere disposable) before it ships, and know in advance whether the specific operation takes a blocking lock.

Concretely, adding a `NOT NULL` constraint to `orders.customer_email` on PostgreSQL as a single step forces the database to scan and validate every one of two hundred million rows while holding a lock that blocks writes for the duration:

```
// Locks orders for the full validation scan - the whole table, all at
once.
Schema::table('orders', function (Blueprint $table) {
    $table->string('customer_email')->nullable(false)->change();
});
```

Splitting it into "add the constraint without enforcing it yet" and "validate it separately" turns one long exclusive lock into a near-instant metadata change plus a scan that only takes a brief lock at the very end:

```
-- Step 1: instant - takes the constraint's existence for granted, doesn't
scan.
ALTER TABLE orders ADD CONSTRAINT customer_email_not_null
    CHECK (customer_email IS NOT NULL) NOT VALID;

-- Step 2: scans the table, but only takes a brief lock to flip the
-- constraint's state once the scan finds no violations - not for the
-- duration of the scan itself.
ALTER TABLE orders VALIDATE CONSTRAINT customer_email_not_null;
```

Run as two separate deploys, the first one is safe to ship immediately; the second can run whenever the backfill from the next section has finished, with no code depending on either step's timing.

Concurrent and non-blocking index creation

Adding an index to a large table is one of the most common migrations to get wrong, because a plain `CREATE INDEX` takes a lock that blocks writes for as long as the index build takes - which, on a two-hundred-million-row table, can be tens of minutes.

PostgreSQL's answer is `CREATE INDEX CONCURRENTLY`, which builds the index without holding a write lock, at the cost of a slower two-pass build and a real failure mode: if it's interrupted, it can leave behind an invalid index that has to be dropped and retried, not resumed.

```
CREATE INDEX CONCURRENTLY idx_orders_customer_created
    ON orders (customer_id, created_at);
```

MySQL's InnoDB engine supports online DDL for many index operations via `ALGORITHM=INPLACE, LOCK=NONE`, which allows concurrent reads and writes during the build - but not every DDL operation supports every algorithm, and the safe habit is to check the specific operation against the running engine version rather than assume.

```
ALTER TABLE orders
    ADD INDEX idx_orders_customer_created (customer_id, created_at),
    ALGORITHM=INPLACE, LOCK=NONE;
```

Laravel's migration files don't manage this distinction automatically - a `Schema::table()` migration using `$table->index()` issues a plain `CREATE INDEX`/`ADD INDEX` unless the raw SQL or a database-specific driver option is used to request the non-blocking variant. On a large table, that's a decision to make explicitly, not a default to trust:

```
public function up(): void
{
    // Schema::table()->index() takes a blocking lock for the build - fine
    // on a small table, a production incident on a two-hundred-million-
row one.
    DB::statement(
        'CREATE INDEX CONCURRENTLY idx_orders_customer_created ON orders
(customer_id, created_at)'
    );
}
```

`CREATE INDEX CONCURRENTLY` can't run inside the transaction Laravel wraps each migration in by default, so the migration class also needs `public $withinTransaction = false;` - forgetting that is the most common way this pattern fails silently in a migration file, even though it works fine when run by hand. And because an interrupted concurrent build leaves an unusable index behind rather than rolling back, always check for one before retrying:

```
-- Finds indexes left in an invalid state by an interrupted CONCURRENTLY
build.
SELECT indexrelid::regclass AS index_name
FROM pg_index
WHERE indisvalid = false;

-- Safe to drop and retry - an invalid index is never used by the planner.
DROP INDEX CONCURRENTLY idx_orders_customer_created;
```

The backward-compatible column rename pattern

Renaming a column directly - `ALTER TABLE orders RENAME COLUMN buyer_email TO customer_email` - is fast as a pure metadata operation on most engines. The real danger isn't the `ALTER` itself; it's that the moment it commits, every piece of code still referencing the old column name breaks atomically, everywhere, at once - including code you don't control (see the next section). The fix is to never do a rename as a single step. Expand it into phases that can each be shipped, verified, and rolled back independently:

Phase	Schema state	Application state	Rollback if something's wrong
1. Add	New column exists, nullable	Not yet used	Drop the new column - no impact
2. Dual-write	Both columns exist	App writes to both columns on every write path; reads still from the old column	Stop dual-writing - old column is still authoritative
3. Backfill	Both columns exist	Batch job populates the new column for existing rows	Backfill is idempotent - safe to re-run or pause
4. Migrate reads	Both columns exist	App reads from the new column; old column no longer read	Revert the read change - old column is still fully populated
5. Stop dual-write	Both columns exist	App writes only to the new column	Revert to dual-writing if any consumer still expects the old column
6. Drop	Old column removed	Fully migrated	Not reversible past this point - this phase only ships once nothing reads the old column

Each phase is a separate, independently deployable change. If phase 4 reveals a bug (the new column has a subtly different meaning than assumed), the rollback is "revert one deploy," not "recover from a completed rename." This is the same expand-and-contract shape used for the shared-package upgrades in Chapter 6 - the pattern generalizes to any change where "old" and "new" need to coexist for a transition window.

Phase 1 is the only phase that touches the database schema directly - it's a plain, additive migration:

```
public function up(): void
{
    Schema::table('orders', function (Blueprint $table) {
$table->string('customer_email')->nullable()->after('buyer_email');     //
BTW after() doesn't work in postgres
    });
}
```

Phase 2's dual-write lives in the model, not in a migration, and is the one line of application code every write path now depends on until phase 5 removes it:

```
protected static function booted(): void
{
    static::saving(function (Order $order) {
        if ($order->isDirty('buyer_email')) {
            $order->customer_email = $order->buyer_email;
        }
    });
}
```

Phase 6's drop is deliberately its own migration, shipped only after the checklist at the end of this chapter is fully checked off:

```
public function up(): void
{
    Schema::table('orders', function (Blueprint $table) {
        $table->dropColumn('buyer_email');
    });
}
```

Batching large backfills

Phase 3 above - populating the new column for two hundred million existing rows - is itself a hazard if done as one statement:

```
// Don't do this on a large table.
DB::table('orders')->update(['customer_email' => DB::raw('buyer_email')]);
```

A single `UPDATE` touching every row holds row locks (and, depending on engine and isolation level, can hold them for the duration of one very long transaction), competes with live traffic for the same rows, and - on replicated setups - can create significant replication lag, since a replica applies the same enormous write as one unit. The safe alternative is chunking: touch a bounded number of rows per statement, with a small pause between batches to let replicas catch up and to leave room for live traffic.

```
Order::query()
    ->whereNull('customer_email')
    ->orderBy('id')
    ->chunkById(1000, function ($orders) {
        foreach ($orders as $order) {
            $order->update(['customer_email' => $order->buyer_email]);
        }

        usleep(100_000); // 100ms - bound replication lag and lock
contention
    });
```

Running this as a queued, resumable Artisan command (tracking the last processed ID) rather than a single migration step means it can be paused, monitored, and restarted without redoing completed work - important for a backfill that might run for hours against a genuinely large table:

```
final class BackfillCustomerEmail extends Command
{
    protected $signature = 'orders:backfill-customer-email {--fresh} {--
dry-run}';

    public function handle(): int
    {
        $lastId = $this->option('fresh') ? 0 : (int)
Cache::get('backfill:customer_email:last_id', 0);
```

```
        $dryRun = $this->option('dry-run');
        $processed = 0;

        Order::query()
            ->whereNull('customer_email')
            ->where('id', '>', $lastId)
            ->orderBy('id')
            ->chunkById(1000, function (Collection $orders) use ($dryRun,
&$processed) {
                if (! $dryRun) {
                    foreach ($orders as $order) {
                        $order->update(['customer_email' =>
$order->buyer_email]);
                    }

                    // Persisted, not just held in memory - a restart
after a
                    // deploy or an `artisan:down` picks up where this
left off
                    // instead of re-scanning rows already backfilled.
                    Cache::forever('backfill:customer_email:last_id',
$orders->last()->id);
                }

                $processed += $orders->count();

                // Without this, a backfill that runs for hours looks
                // identical to a stuck one - there's nothing to
distinguish
                // "still working" from "silently hung" on the terminal.
                $this->components->info(
                    ($dryRun ? '[dry run] ' : '')."Backfilled {$processed}
rows so far (last ID: {$orders->last()->id})."
                );

                usleep(100_000);
            });

        $this->components->info($dryRun
            ? "Dry run complete: {$processed} rows would have been
backfilled."
```

```
            : 'Backfill complete: no rows remain with a null
customer_email.');

        return self::SUCCESS;
    }
}
```

Dispatched via `Schedule::command('orders:backfill-customer-email')->everyMinute()->withoutOverlapping()` (Chapter 10 covers scheduling and overlap guards in depth), this makes progress in small, bounded increments and survives being interrupted by a deploy at any point.

`--dry-run` is worth adding to every backfill command before it ever touches a large table: it walks the exact same query and chunking logic, reports how many rows and which ID range would be affected, and writes nothing - so a mistake in the `WHERE` clause shows up as a suspicious row count in dry-run output, not as two hundred million incorrectly updated rows. And it's always worth emitting a progress line per batch, not just a single message at the end - a backfill that runs for hours with no output is indistinguishable from one that's silently hung, and the per-batch line (row count so far, last ID reached) is what turns "is this still running?" into a question the terminal already answers.

Downstream consumers: your schema is a public API

The phased rename above protects the application. It does nothing for anything else that reads the same database directly and isn't part of the deploy you control: a nightly analytics export, another internal service with direct read access to the same tables (an anti-pattern, but a common inherited one), or a BI tool querying a read replica. From their perspective, dropping `buyer_email` in phase 6 is a breaking change shipped with zero notice, because nothing about the application's own migration process is aware they exist.

The fix starts with an inventory: before any schema change on a table of nontrivial age, identify who reads it besides the application - export jobs, other services, reporting queries, scheduled dashboards. For Northfield's rename, that inventory turned up an internal reporting export that ran nightly against `orders.buyer_email` directly, owned by a different team on a different release cycle.

Two patterns handle this without forcing every downstream consumer onto the application's timeline:

- **A compatibility view.** Keep a database view (or, during the dual-write window, the old column itself) that presents the old shape, so existing consumer queries keep working unmodified while the underlying table has already moved on. The view gets dropped only once every known consumer has confirmed it's no longer needed - not on the application's schedule, but on the slowest consumer's.
- **A deprecation window with an actual notification.** Whoever owns the schema change communicates the timeline to known downstream owners the same way a public API deprecates an endpoint: an announced window, not a silent drop. This only works if the inventory step actually happened - a consumer nobody knew about can't be notified.

Both patterns above are damage control for consumers that already read columns directly. The more durable fix, when the downstream consumer is an HTTP client rather than a direct database reader, is to never let that exposure exist in the first place: don't return `Model::toJson()` or a bare `$order->toArray()` from a controller, and don't let a resource's field names default to mirroring column names. A Laravel API Resource (or a `league/fractal` transformer, on a project that predates or doesn't use Resources) sits between the two and makes the wire format an explicit, independently versioned contract:

```
final class OrderResource extends JsonResource
{
    public function toArray(Request $request): array
    {
        return [
            'id' => $this->id,
            // The response field stays `buyer_email` - the contract every
            // existing API client already depends on - even though the
            // column behind it is `customer_email` after phase 6 ships.
            // The rename happens entirely on one side of this line.
            'buyer_email' => $this->customer_email,
            'status' => $this->status,
            'total_cents' => $this->total_cents,
        ];
    }
}
```

With that boundary in place, the entire six-phase rename in this chapter can happen without a single HTTP API consumer noticing - the resource is the only place that has to change, and it changes in the opposite direction: mapping the new column to the response field the API already promised, rather than the API being what dictates the column name.

For the rename, the old `buyer_email` column stayed in place, populated by the dual-write step, until the reporting team confirmed their nightly export had been updated to the new column name - on their schedule, not the application's. Only then did phase 6 ship. Had the export needed more time than the dual-write window allowed, a compatibility view would have covered the gap:

```
-- Ships in the same deploy as phase 6's drop, so the reporting export's
-- `SELECT buyer_email FROM orders` keeps working unmodified, reading
-- from the new column under an old name it still recognizes.
CREATE VIEW orders_legacy_shape AS
    SELECT orders.*, customer_email AS buyer_email
    FROM orders;
```

A schema-change checklist

```
- [ ] Tested the specific operation (not just "a migration") against a
production-sized copy of the table
- [ ] Confirmed whether the operation takes a blocking lock on the current
engine/version
- [ ] Used a non-blocking index build (CONCURRENTLY / ALGORITHM=INPLACE)
for large tables
- [ ] Renames/type changes are split into add -> dual-write -> backfill ->
migrate reads -> drop phases
- [ ] Backfills run in bounded batches with a pause, not as one statement
- [ ] Inventoried downstream consumers (exports, other services, BI tools)
that read this table directly
- [ ] HTTP API responses go through a Resource/transformer, not a bare
model - column renames don't reach API clients
- [ ] Downstream consumers notified or bridged with a compatibility view
before the old shape is dropped
- [ ] The "drop" phase is the last, separately reviewed step - never
bundled with the rest
```

Chapter recap

- A migration's cost scales with table size and the specific engine/operation, not with how simple the change looks in code - always test DDL against production-sized data.
- Use non-blocking index builds (`CREATE INDEX CONCURRENTLY` on PostgreSQL, `ALGORITHM=INPLACE, LOCK=NONE` on MySQL) on any table large enough that a lock would be noticed.
- Never rename or retype a column in one step - expand it into add, dual-write, backfill, migrate reads, stop dual-write, and drop, each independently deployable and reversible until the last phase.
- Backfill large tables in bounded, pausable batches (`chunkById` plus a small delay), not a single sweeping `UPDATE`, to protect replication and live traffic.
- Your schema is effectively a public API the moment anything outside your own deploy - an export, another service, a BI tool - reads it directly. Inventory those consumers before assuming a change is safe.
- For consumers that go through HTTP rather than the database directly, a Laravel API

This is a sample from "Laravel in Production: Architecture, Performance, and Operations at Scale".