Laravel After Deploy

Architecture, Performance, and Operations at Scale

Nuruzzaman Milon

Laravel After Deploy

Architecture, Performance, and Operations at Scale

Nuruzzaman Milon

First published: August 2026

Written with	Markdown, dictated in part with Hex, an open-source, on-device voice-transcription tool, and typed the rest.
Produced with	Ibis Next, using mPDF for PDF generation and CommonMark for HTML rendering.
Trim size	7.44in x 9.69in (Crown Quarto)
Typefaces	Linux Libertine (body text), Times (headings), and 0xProto (code, SIL Open Font License 1.1).
Code width	66 characters per line, so a listing never wraps awkwardly on the page.

To my family, for making every effort worthwhile.

Preface

Shipping a Laravel application is the easy part. The hard part starts after deploy: when a flash sale or a viral post turns usual traffic into a spike, when bots hammer your login endpoint, when a dependency you don't control has a bad afternoon, when the same webhook arrives twice, when the on-call phone rings at 3 a.m. and the answer isn't in the framework docs. This book is about that layer - the decisions that keep a real system fast, correct, and operable once it's already in production.

Who this book is for

This book is written for a mid-to-senior Laravel engineer who already knows how to build a Laravel application and is now being asked to keep one alive - the person a growing team hands the architecture decisions, the performance incident, and the migration that has to happen without a maintenance window. It assumes you already know how routing, Eloquent, queues, and testing work in Laravel; it doesn't stop to explain the framework. Every page is instead about the layer of decisions that sits on top of it: how you structure a codebase as more than one team starts touching it, how you keep it fast once traffic outgrows what fits comfortably on one database, how you keep it correct when the same request can arrive twice, how you keep it available when a dependency you don't control goes down, and how you run all of that at 3 a.m. on a Tuesday without it becoming a story you're still telling a year later.

If you're looking for a Laravel tutorial - how to define a route, how to write a migration, how Eloquent relationships work - this isn't that book, and there are excellent ones already. If you're the engineer who's outgrown those books and is now staring at a monorepo, a queue backlog, or a balance that doesn't add up, keep reading.

How to read it

The book is organized into eight parts, and the parts are ordered by dependency rather than importance, so reading front to back works, but it isn't the only sane way through it. Part 1 (Foundations) and Part 2 (Performance Engineering) build vocabulary that later parts assume - "shared kernel," "N+1," "backfill," and so on. Part 5 (Observability & Monitoring) is worth reading earlier than its page number suggests, because "correlation ID," "production dashboard," "SLI/SLO/SLA," and "burn rate" get reused heavily in Parts 6 and 7. Beyond that, Parts 3 through 7 are mostly skippable for a live fire - with the vocabulary caveats above - so if you picked this book up because of a live incident with your queues, your database failover, or your CI pipeline, it's reasonable to jump straight to the relevant chapter. Chapter 46, the capstone, assumes you've read - or at least skimmed - everything before it: it's a single worked incident that touches nearly every decision in the book, and it's meant to be read last.

Each chapter follows the same shape. It opens with a concrete production failure - drawn from more than fifteen years of real incidents, anonymized - then works through the pattern that would have prevented or fixed it, and closes with a short "Chapter recap" you can scan later without rereading the whole chapter. Not every one of those incidents happened in a Laravel project; over those years I've worked across many languages and frameworks, and where the original system wasn't Laravel, I've translated the failure into a Laravel setting so it can be demonstrated in the stack this book is about.

About the examples

Every incident described in this book actually happened in real production systems. The failure modes, the trade-offs, and the fixes come from systems I helped build and then had to keep running, not from thought experiments. What isn't real is any identifying detail: company names, product names, team names, exact numbers, and anything else that could point back to a specific employer or client has been changed, generalized, or recombined. The named fictional businesses that recur throughout the book are stand-ins, not case studies of any single company:

Name	Domain	Typical chapters
Adda	Social platform; abuse and rate limiting	16
Chanda	Subscription billing, queues, IaC	13, 37
Dokan	Multi-tenant SaaS, release checklist	10, 36
Drishti	Livestreaming and short-video; real-time, cost/capacity	11, 15, 25, 38–39, 44
HaatBazaar	Marketplace (search, inventory); capstone synthesis	18, 21, 46
Hishab	Wallets, ledger, payouts, observability	19–20, 22–23, 28, 31–32, 40, 42

Name	Domain	Typical chapters
Jatri	Ride-dispatch; runbooks and on-call	43
Karbar	Order-processing platform	1–3, 8–9, 17, 26, 29, 34–35, 41

Those chapter lists are the main appearances, not an exhaustive index - several of these names show up elsewhere when the same failure mode needs a familiar face.

Nothing here discloses a trade secret, a proprietary architecture, or confidential information belonging to any organization I've worked with; the anonymization exists specifically to protect that while keeping the underlying lesson intact. The architecture decisions and opinions in these pages are my own.

The code examples are written in Laravel, because that's the stack this book is built around - but most of the ideas aren't Laravel-specific. Idempotency, backpressure, observability, failure isolation, and the rest of what these chapters cover are production concerns that show up in any language and any framework. If you aren't on Laravel, the patterns still apply, even when the exact code examples don't.

Reading paths for live incidents

If you're here because something is on fire, these paths get you to the relevant chapters without reading front to back. Skim the glossary first if a term is unfamiliar.

- **Money, balance, or payout looks wrong** → 19 (idempotency, state machines) → 20 (webhooks) → 22 (ledger and fund holds). Add 23 only if you need rejected-attempt history and audit reconstruction, not just successful movements. Overselling / inventory races → 21.
- **Deploy went green but nothing changed** → 35 (containers) → 36 (release checklist: workers, scheduler, `/ready`) → 37 (IaC roles).
- **Multi-app monorepo or auth pain across services** → 3 (monorepo) → 34 (CI path filters) → 41 (Sanctum vs JWT).
- **On-call can't see what's happening** → 31 (logs/metrics/traces) → 32 (production dashboard) → 33 (SLOs and alerting) → 43 (runbooks).

A note on unfamiliar terms

Abbreviations are spelled out the first time they're used in a chapter, but if you're dropping into a chapter out of order, Appendix A (Glossary) collects the recurring ones - SLO, SLA, SLI, MTTR, RPO/RTO, ADR, and the rest - in one place, each with a pointer back to the chapter that covers it properly.

Chapter 9 - Safe Schema Evolution at Scale

The migration that only failed in production

At Karbar, 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 8, 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," "validate it separately," and only then "make the column formally `NOT NULL`" 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 - records the constraint without checking
existing rows.
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;

-- Step 3: after the validated check proves there are no nulls,
PostgreSQL
-- can set the column attribute without repeating the full-table
scan.
ALTER TABLE orders
    ALTER COLUMN customer_email SET NOT NULL;
```

Run as separate deploys, the first one is safe to ship immediately; validation and the final `SET NOT NULL` can run once the backfill from the next section has finished, with no code depending on their 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

Phase	Schema state	Application state	Rollback if something's wrong
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 7 - 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) {
        // after() is MySQL-only; omit it on PostgreSQL.
        $table->string(
            'customer_email'
        )->nullable()->after('buyer_email');
    });
}
```

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 - with the caveat that "every write path" means every write path that goes through a model instance. `Order::query()->update( ... )` and anything built on `DB::table('orders')` skip model events entirely, so grep for those before trusting this hook, because phase 4's correctness rests on 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) {
        Order::query()
            ->whereIn('id', $orders->pluck('id'))
            ->update(
                ['customer_email' => DB::raw('buyer_email')]
            );

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

Running this as a resumable Artisan command that processes one bounded batch (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} {--batch=1000}';

    public function handle(): int
    {
        $lastId = $this->option(
            'fresh'
        ) ? 0 : (int) DB::table('backfill_progress')
            ->where('name', 'customer_email')
            ->value('last_id');

        $dryRun = $this->option('dry-run');
        $orders = Order::query()
            ->whereNull('customer_email')
            ->where('id', '>', $lastId)
            ->orderBy('id')
            ->limit((int) $this->option('batch'))
            ->get(['id']);

        if ($orders->isEmpty()) {
            $this->components->info(
                'Backfill complete: no null emails left.'
            );

            return self::SUCCESS;
        }

        $nextLastId = $orders->last()->id;

        if (! $dryRun) {
            /*
             * whereNull again, so a concurrent write that
already
             * populated one of these rows isn't overwritten by a
             * stale read.
             */
```

```
                Order::query()
                    ->whereIn('id', $orders->pluck('id'))
                    ->whereNull('customer_email')
                    ->update(
                        ['customer_email' => DB::raw('buyer_email')]
                    );

                DB::table('backfill_progress')->updateOrInsert(
                    ['name' => 'customer_email'],
                    ['last_id' => $nextLastId],
                );
            }

            $this->components->info(
                ($dryRun ? '[dry run] ' : '')
                ."Processed {$orders->count()} rows"
                ." (last ID: {$nextLastId})."
            );

            return self::SUCCESS;
        }
    }
```

Dispatched via `Schedule::command('orders:backfill-customer-email')->everyMinute()->withoutOverlapping()`, this makes progress in small, bounded increments and survives being interrupted by a deploy at any point. The cursor lives in a table rather than in the cache for a boring reason that bites people anyway: a cache is allowed to lose your key. An eviction under `maxmemory`, a `cache:clear` during a release, a Redis restart without persistence - any of those silently resets the backfill to ID 0, and the `whereNull` guard means it will quietly grind back through two hundred million already-completed rows without ever reporting that anything went wrong.

`--dry-run` is worth adding to every backfill command before it ever touches a large table: it walks the exact same query and batch-selection logic, reports how many rows and which ID range the next batch would affect, 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, 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 Karbar'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 a consumer's `SELECT` list can go on naming `buyer_email` long after the table stopped having that column. They do still have to repoint the `FROM` clause at the view, which is a one-line change on their side rather than a rewrite - and it has to happen while the old column still exists, not on the day it disappears. 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,
            // Wire format is frozen; the column behind it is
not.
            '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 the other way round from what you'd expect: it maps the new column onto the response field the API already promised, instead of the API dictating what the column is allowed to be called.

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 - shipped during the deprecation window rather than alongside the drop, since a view that appears on the same day the column vanishes has nobody pointed at it yet:

```
-- Ships during the deprecation window, well before phase 6's
drop, so
-- consumers have time to repoint FROM orders at
orders_legacy_shape and
-- keep buyer_email in their SELECT list afterwards.
CREATE VIEW orders_legacy_shape AS
    SELECT id, customer_id, customer_email AS buyer_email,
status, total_cents
    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 Resource (or `league/fractal`) makes the response shape an explicit contract instead of a mirror of column names - the safest fix is the one that prevents the exposure from existing at all.
- Bridge downstream consumers with a compatibility view or an announced deprecation window; let the slowest consumer's timeline, not your migration's convenience, decide when the old shape can actually be dropped.

Chapter 16 - Rate Limiting & Abuse Protection

2 a.m., and the login endpoint won't stop ringing

The graphs look like an attack, because it is one. Login attempts on Adda, a fictional social platform, have gone from a baseline of a few hundred per minute to over forty thousand, spread across tens of thousands of IP addresses, each one trying a handful of username/password combinations before moving on. It's credential stuffing: a botnet replaying a leaked password list from an unrelated breach, hoping a slice of users reused the same password here.

The naive fix - block anyone who fails login more than five times - makes things worse. The attack is already distributed across so many source IPs that per-IP blocking barely slows it down, while a support queue fills with real users who mistyped a password twice and are now locked out for an hour. The team that survives this incident well isn't the one with the cleverest CAPTCHA; it's the one that had already decided, in calmer times, what a rate limit protects, at what granularity, and what happens to a legitimate request caught in the blast radius.

That's the actual subject of this chapter: rate limiting is not one mechanism, it's a layered set of decisions about *who* you're limiting, *what unit of time* you're limiting them over, and *what you do* when the limit is hit - decisions that are much cheaper to make before an attack than during one.

Two limiter algorithms, and why the difference matters

Laravel ships with one of the two algorithms below, and it's worth knowing which one before you assume an endpoint is protected: the `RateLimiter` facade and the `throttle` middleware both count requests in cache-based fixed windows.

Fixed window counts requests in discrete buckets (e.g. "requests this calendar minute"). It's cheap - one counter, one TTL - but it has a boundary problem: a client can send its full limit at 11:59:59 and its full limit again at 12:00:01, getting two limits' worth of traffic in two seconds.

Token bucket (and its close cousin, the sliding-window log) models a bucket that refills at a steady rate and is drained by each request. It smooths out the boundary problem because there's no reset instant - the bucket's state is continuous. It costs slightly more to compute (you need the last-refill timestamp, not just a count) but it's a much closer match to "how many requests can this client sustainably send," which is usually the thing you actually care about.

	Fixed window	Token bucket / sliding window
Storage cost	One counter + TTL per key	Timestamp + count per key
Boundary burst	Up to 2x limit across a window edge	Bounded by bucket size, not window edge

	Fixed window	Token bucket / sliding window
Good for	Coarse, cheap limits (e.g. "1000 req/hour per API key")	Limits where burst behavior matters (login, payment attempts)
Laravel primitive	`RateLimiter::attempt()` with default cache-based windows	Custom atomic bucket in Redis or an edge-provider limiter

For most application-level throttles, the fixed window's imprecision doesn't matter - nobody cares if a reporting endpoint allows a small burst across a minute boundary. For anything security-sensitive (login, password reset, payment attempts, OTP verification), the boundary burst is exactly the gap an attacker will find, so it's worth the extra complexity of a token bucket - which you will be building yourself, in Redis or at the edge, because it isn't a mode you can switch the stock limiter into.

Choosing the key: per-user, per-IP, per-action, or a combination

A rate limit is only as good as the key it's keyed on, and this is where most naive implementations fail.

- **Per-IP** limits stop a single machine from hammering an endpoint, but are close to useless against a distributed botnet, and actively harmful behind carrier-grade NAT or corporate proxies, where thousands of

real users share one visible IP.

- **Per-user (or per-account)** limits are precise but only apply once you know who the user is - they don't help on an unauthenticated login attempt where the "user" is exactly what's being guessed.
- **Per-action** limits (e.g. "5 password reset emails per hour, regardless of who's asking") protect a shared downstream resource (an email provider, an SMS gateway) from being exhausted by any single actor.
- **Composite keys** - IP *and* the attempted username, or device fingerprint *and* account - are usually where the real protection lives, because they let you set a tight limit on the narrow combination ("this IP trying this specific username") without punishing an entire shared IP range for one bad actor.

The credential-stuffing scenario above is precisely why single-axis keys fail: per-IP limiting is defeated by the botnet's IP diversity, and per-username limiting alone would let the botnet grind through the account list one attempt per username, staying under any reasonable per-account threshold. The fix is layered limits on different keys simultaneously, each cheap on its own, expensive to defeat all at once.

```
/*
 * A layered login throttle: IP-wide, then IP+username, then
 * account-wide.
 */
public function attempt(Request $request): RedirectResponse
{
    $ip = $request->ip();
    $username = (string) $request->input('email');

    $limits = [
        // this IP, any account, per minute
        "login-ip:{$ip}" => 30,
```

```
        // this IP against this account
        "login-ip-user:{$ip}:{$username}" => 5,
        // this account, from anywhere
        "login-user:{$username}" => 10,
    ];

    foreach ($limits as $key => $maxAttempts) {
        if (RateLimiter::tooManyAttempts($key, $maxAttempts)) {
            $seconds = RateLimiter::availableIn($key);

            throw ValidationException::withMessages([
                'email' =>
                    "Too many attempts. Retry in {$seconds}s.",
            ])->status(429);
        }
    }

    if (! Auth::attempt($request->only('email', 'password'))) {
        foreach (array_keys($limits) as $key) {
            /*
             * The first failed attempt must create the fixed
             * window's TTL.
             */
            RateLimiter::hit($key, decaySeconds: 60);
        }

        throw ValidationException::withMessages([
            'email' =>
                'These credentials do not match our records.',
        ]);
    }

    /*
     * Successful logins do not increment any of the failure
     * counters.
```

```
        */
        return redirect()->intended('/dashboard');
    }
```

Hitting the limiter only on *failed* attempts, rather than on every request, is what keeps this from punishing legitimate traffic. A limiter that counts successes too will eventually lock out the user whose only crime is logging in a lot.

Bot and risk scoring: don't let a vendor outage become your outage

Many teams layer a third-party risk-scoring or bot-detection service (device fingerprinting, behavioral scoring, a managed CAPTCHA) on top of application-level rate limits. These are valuable, but they introduce a dependency that can fail independently of your own infrastructure - and a naive integration turns "the risk vendor is slow" into "nobody can log in."

The rule that matters here: **decide, explicitly, what happens when the risk check itself fails or times out**, rather than letting an exception bubble up and 500 the request. It's the same fail-open-vs-fail-closed call Chapter 26 builds into a general framework for any dependency; applied here to a risk-scoring vendor specifically, there are two defensible postures, and the right one depends on what the endpoint protects:

Failure mode	Fail closed (block)	Fail open (allow)
Behavior on vendor timeout	Treat as high-risk, block or challenge	Treat as unscored, let the request through
Right for	High-value, low-volume actions (large withdrawals, account recovery)	High-volume, low-marginal-risk actions (viewing a page, browsing a catalog)
Failure mode if wrong	Vendor blip becomes a full outage for real users	A vendor outage becomes a temporary abuse window
Mitigation	Short timeout (100-300ms) + circuit breaker so a slow vendor degrades to "unscored" quickly	Alert loudly on sustained fail-open so someone investigates rather than it becoming permanent

For Adda's login endpoint, the pragmatic default is: apply the application-level rate limits (which are self-hosted and always available) unconditionally, and treat the risk score as an *additional* signal that can tighten the response (e.g. require a CAPTCHA) but should not be a hard dependency for the base throttle to function. That way a risk-vendor outage degrades the system to "slightly less sophisticated bot detection," not "login is down."

In code, that means a tight timeout on the vendor call and an explicit fail-open branch, rather than letting the exception propagate:

```
public function scoreLogin(string $ip, string $email): ?RiskScore
{
    try {
        $response = $this->http
            // 250ms: fail fast, don't hold up the login request
            ->timeout(0.25)
            ->post('https://risk-vendor.example.com/v1/score', [
                'ip' => $ip,
                'email' => $email,
            ]);

        if ($response->failed()) {
            /*
             * A 4xx/5xx arrives here, not in the catch block
             * below.
             */
            Log::critical(
                'Risk vendor returned an unsuccessful response',
                [
                'status' => $response->status(),
            ]);

            return null;
        }

        return RiskScore::fromArray($response->json());
    } catch (ConnectionException $e) {
        /*
         * Fail open: an unscored login still passes the
         * application-level throttle above, it only skips the
         * extra CAPTCHA/challenge step.
         */
        Log::critical(
            'Risk vendor unavailable, failing open',
```

```
                ['exception' => $e->getMessage()]
            );

            return null; // caller treats null as "unscored"
        }
    }
```

The loud `Log::critical` call is what keeps a fail-open decision visible instead of silently permanent.

DDoS protection lives above Laravel

The credential-stuffing attack that opened this chapter is abuse: many small, well-formed requests that look enough like real logins to reach your application. A distributed denial-of-service (DDoS) attack is a different shape - enough traffic, or enough half-open connections, that the goal is to exhaust bandwidth, file descriptors, or worker capacity before your code ever gets a vote. Laravel's `RateLimiter` never fires, because the request never makes it to a PHP worker. Treating those two problems as the same "rate limit" problem is how teams spend a week tuning `throttle:60,1` while the load balancer is already dropping SYN packets.

The layer that can absorb a volumetric flood is the one in front of your origin: a CDN or DDoS-aware edge (Cloudflare, AWS Shield / WAF, Fastly, and their peers), with connection limits and SYN cookies at the load balancer as a second line. That edge is where you put IP reputation, geographic blocks you actually intend to enforce, managed bot challenges, and traffic shaping that can drop junk without waking a Laravel container. Application rate limits still matter for the abuse cases this chapter covers - they are just not the DDoS plan.

Layer	What it can stop	What it cannot
Edge / CDN / WAF	Volumetric floods, obvious botnets, TLS exhaustion, cheap challenge-before-origin	Business-logic abuse that looks like a real user (credential stuffing with residential IPs)
Load balancer / reverse proxy	Connection storms, slowloris-style holds, per-IP connection caps	Anything that already looks like a healthy HTTP request
Laravel `RateLimiter` / `throttle`	Per-key abuse once a request reaches the app	Floods that saturate the link or the worker pool before PHP runs

For Adda, the practical split after the stuffing incident was: keep the layered login throttles in the app (they still catch the "looks like a user" case), and put volumetric protection at the edge so a packet flood never becomes a Horizon backlog. Two other habits make the origin cheaper to defend when something does leak through:

- **Fail expensive work late.** Don't run bcrypt, Eloquent, or a risk-vendor call before the cheapest checks (routing, auth middleware, a cache hit on a known-bad IP). Every millisecond of origin work during a flood is capacity an attacker can buy more cheaply than you can.
- **Keep health and static paths cheap and separate.** A `/up` or health probe that still boots the full application, or a homepage that always misses cache and hits the database, turns "protect the login endpoint"

into "the whole fleet is busy answering probes and HTML." Edge caching for anonymous GETs, and a health check that doesn't touch Redis or MySQL, shrink the surface an attack can usefully burn.

What you should not do is invent an in-app "DDoS mode" that flips every limit to zero when CPU is high. That couples abuse policy to capacity noise, locks out real users during a legitimate spike (the flash-sale case Chapter 17 plans for), and still does nothing if the attack never reaches PHP. Edge absorption first; application throttles for the requests that deserve application judgment.

Backpressure: what a limit does after it triggers

A rate limit isn't just "reject the request." What you return, and what you do internally when a limit is being approached, is its own design surface:

- **Return `429 Too Many Requests`**, not a generic `403` or `500` - well-behaved clients (and API consumers) know to back off on a 429 specifically, and a `Retry-After` header lets them do it without guessing.

Concretely, that's `RateLimiter::availableIn()` feeding straight into the response:

```
public function login(Request $request): JsonResponse
{
    $key = "login-ip:{$request->ip()}";

    if (RateLimiter::tooManyAttempts($key, 30)) {
        $retryAfter = RateLimiter::availableIn($key);

        return response()->json(
            [
                'message' =>
                    "Too many attempts. Retry in {$retryAfter}s."
            ],
            429,
        )->header('Retry-After', $retryAfter);
    }

    // ...
}
```

- **Shed load progressively, not as a cliff.** If an upstream dependency (a payment gateway, a third-party API) is itself struggling, consider tightening your own limits proactively rather than waiting for it to start timing out request-by-request - this is the same backpressure idea covered for queues in Chapter 13, applied at the edge instead of the worker layer.
- **Distinguish abuse throttling from capacity throttling.** A login rate limit exists to stop abuse and should feel deliberately strict. A general API rate limit protecting your own database from being overwhelmed is a capacity control, and Chapter 17's load-testing work is what tells you where that number should actually sit - guessing it produces either a limit so loose it doesn't protect anything, or so tight it throttles normal peak traffic.

Chapter recap

- Rate limiting is a layered decision, not a single mechanism: pick an algorithm (fixed window for cheap coarse limits, token bucket for anything where burst behavior matters), a key (IP, user, action, or a composite), and a response (reject, challenge, or degrade).
- Composite keys (IP + username, device + account) catch attacks that defeat any single-axis limit, including distributed credential stuffing that no per-IP limit alone can stop.
- Increment attempt counters on failure, not on every request, or you'll rate-limit your legitimate power users.
- Third-party risk/bot signals are an enhancement, not a foundation - decide explicitly whether each check fails open or closed, and never let a vendor timeout become your outage.
- DDoS is an edge problem, not a `RateLimiter` problem: absorb volumetric floods at the CDN/WAF/load balancer, keep origin work cheap for what leaks through, and reserve application throttles for abuse that looks like a real request.
- Return `429` with `Retry-After`, and treat abuse throttling (strict, security-motivated) and capacity throttling (informed by load testing, see Chapter 17) as two different tuning problems with two different failure costs.

Chapter 25 - Working with Datetime and Timezones

The streak that broke at midnight

Drishti, a fictional livestreaming and short-video app, runs a seasonal campaign with a daily streak: watch or post on consecutive local days and unlock a reward on Monday. The product copy says "day" the way a user means it - midnight to midnight in Pacific time, where the company and most of its creators sit. The implementation means something else.

`APP_TIMEZONE` is `UTC`, which is correct for storage. Streak logic calls `now()->startOfDay()` and stores the result as the active date. At 11:30 p.m. Pacific on a Tuesday, that's already Wednesday UTC. A creator who posts once on Tuesday evening and once on Wednesday morning - two distinct Pacific calendar days - gets a single streak day, because both actions fall on the same UTC date. Support tickets pile up the week the campaign goes viral. Engineering "fixes" it by flipping the app timezone to `America/Vancouver`. Overnight, every `created_at` comparison, every TTL, and every cron that assumed UTC silently shifts by seven or eight hours depending on daylight saving, and Monday's payout job either double-fires or doesn't fire at all.

Timezone bugs are correctness bugs. They don't show up as 500s; they show up as the wrong calendar day, the wrong week boundary, a schedule that runs at the wrong wall-clock time, or an admin panel that displays yesterday as today. This chapter is about keeping two clocks straight - **UTC for instants**, a named **business timezone for calendar math** - and the edge cases that appear the moment a product says "day," "week," or "Monday at

midnight."

Instants and calendar days are different types

An *instant* is a point on the timeline: "this ticket was created at this exact moment," stored once, comparable everywhere. A *calendar day* is a civil label that depends on a timezone: "Tuesday, March 10" in Vancouver is not the same set of instants as "Tuesday, March 10" in UTC. Mixing them is the root failure mode.

Question	Use
When did this row get written?	UTC instant (`timestampTz`, `now()` in UTC)
Is this user still inside a 24-hour redeem window?	Duration from a UTC instant (`subHours(24)`)
Did they act on consecutive *local* days?	Convert to business TZ, then `toDateString()`
Does "this week" mean Mon-Sun for the campaign?	Week boundaries in the business TZ
When should the Monday payout cron run?	Scheduler `->timezone( ... )` matching that calendar

Laravel's `config('app.timezone')` should almost always be `UTC`. That makes `now()`, Eloquent timestamps, and queue `available_at` values mean the same thing on every server and in every region. Business calendar

semantics belong in application code as an explicit constant - not in `APP_TIMEZONE`, and not as a per-row column unless you genuinely need per-user timezones (wallets with local spending days, Chapter 23's `spendingDay`, are that case; a single-company campaign usually is not).

```
final class CampaignConfig
{
    /*
     * Calendar days, weeks, and payout schedules for this
     * product. DB timestamps stay UTC instants; convert to
     * this zone only when asking civil-date questions.
     */
    public const BUSINESS_TIMEZONE = 'America/Vancouver';
}
```

One constant beats a `timezone` column you will forget to set on half the rows, and beats three competing literals (`America/Vancouver`, `America/Los_Angeles`, `PST`) scattered across jobs.

Convert at the boundary, not in the middle

When an event arrives - a watch, a purchase, a ticket grant - keep the instant in UTC, and derive the civil day only where the product rule needs it:

```
public function recordActiveDay(
    int $userId,
    CarbonImmutable $activeAt,
): void {
    $activeDate = $activeAt
        ->timezone(CampaignConfig::BUSINESS_TIMEZONE)
        ->startOfDay();
    $today = $activeDate->toDateString();

    $progress = StreakProgress::query()
        ->lockForUpdate()
        ->firstOrCreate(
            [
                'user_id' => $userId,
                'active_date' => $today,
            ],
        );

    /*
     * ... streak increment using $today / yesterday
     * in the same business timezone
     */
}
```

The important move is `->timezone(BUSINESS_TIMEZONE)` *before* `startOfDay()` or `toDateString()`. Calling `startOfDay()` in UTC and then "displaying" Pacific is how you mint the wrong day. The inverse shows up in queries: when you need "everything since one week ago at Pacific midnight," compute that bound in the business zone and convert back to UTC for the SQL comparison:

```
$weekAgoPacificMidnightUtc = CarbonImmutable::now(
    CampaignConfig::BUSINESS_TIMEZONE,
)
    ->subWeek()
    ->startOfDay()
    ->utc();

Campaign::query()
    ->where('ends_at', '>=', $weekAgoPacificMidnightUtc)
    ->get();
```

That pattern - civil math in the business zone, storage and filters in UTC - is the same idea Chapter 23 uses when it freezes `spendingDay` at decision time so replay never re-asks "what is today?"

Schedulers must declare the timezone they mean

Laravel's scheduler inherits `app.timezone`. If that is UTC, then `->dailyAt('00:05')` means 00:05 UTC, not "just after midnight for the Pacific campaign." The failure mode from the opening incident is usually quieter than a wrong app timezone: one job uses `->timezone('America/Vancouver')`, a sibling job forgets and runs in UTC, and both claim to be "the Monday payout."

```
$schedule->job(new DisburseStreakRewards)
    ->weekly()
    ->mondays()
    ->at('00:05')
    ->timezone(CampaignConfig::BUSINESS_TIMEZONE)
    ->withoutOverlapping()
    ->onOneServer();

$schedule->command('campaign:sync-all-items')
    ->dailyAt('06:00')
    ->timezone('UTC')
    ->withoutOverlapping()
    ->onOneServer();
```

Be explicit even when the answer is UTC. A codebase that mixes Vancouver payouts, New York compliance jobs, and Los Angeles digests is fine - as long as each entry names its zone next to the clock time, and that zone matches the product rule the job implements. A schedule entry with no `->timezone()` is not "timezone-agnostic"; it is "whatever `app.timezone` is today," which is the wrong kind of default for wall-clock work.

Uniqueness keys must use the same clock as the schedule

`ShouldBeUnique` and cache locks are timezone-sensitive when the key embeds a date or hour. A streak reward job that must run once per Pacific calendar day needs a Pacific day in `uniqueId()`. A missing-ticket generator that may safely run once per UTC hour needs a UTC hour. Using the wrong one either blocks a legitimate second run or allows a duplicate:

```
public function uniqueId(): string
{
    /*
     * Pacific calendar day - aligned with the Monday
     * Vancouver schedule, not with UTC midnight.
     */
    return $this->campaign->slug.'::'.now(
        CampaignConfig::BUSINESS_TIMEZONE,
    )->format('Y-m-d');
}
```

```
public function uniqueId(): string
{
    // Hourly reconciliation: UTC hour is the right grain.
    return $this->campaign->slug.'::'.now()
        ->format('Y-m-d-H');
}
```

If the unique key and the schedule disagree about which midnight matters, you will get exactly the incident the unique constraint was supposed to prevent - either skipped work or double payouts - and it will reproduce only near timezone boundaries.

"Day" sometimes means 24 hours

Not every product "day" is a calendar day. A redeem cap of "once per day" often means "no more than once in any rolling 24-hour window," which is a duration from a UTC instant:

```
$recent = Redeem::query()
    ->where('user_id', $userId)
    ->where('created_at', '>=', now()->subHours(24))
    ->exists();
```

A weekly purchase limit that uses `startOfWeek()` in UTC while marketing promises "this Pacific week" is a quieter cousin of the streak bug. Decide deliberately:

- **Rolling duration** (`subHours(24)`, `subDays(7)`) when the rule is elapsed time.
- **Civil calendar** (business-TZ `startOfDay` / `startOfWeek`) when the rule is a named day or week the user would recognize on a wall calendar.

Write the choice down next to the constant. The incident is almost always someone using the other interpretation without noticing.

Don't write time as a magic number

Durations have the same readability problem as timezones: a bare `300` in a cache call does not say whether the author meant five minutes, and after Laravel started treating integer TTLs as seconds, half the codebase's "I thought that was minutes" assumptions became silent bugs. Prefer a helper that names the unit.

```
use function Illuminate\Support\{
    minutes,
    seconds,
    hours,
    days,
};

/*
 * Readable at the call site - five minutes, not "300 of
 * something." These return a CarbonInterval the cache
 * layer accepts directly.
 */
Cache::put('trending', $feed, minutes(5));
Cache::remember(
    'exchange-rate',
    seconds(30),
    fn () => $this->fetchRate(),
);
Cache::put('sitemap', $xml, hours(6));
Cache::put('annual-report', $pdf, days(1));
```

`now()->addMinutes(5)` (or `now()->plus(minutes: 5)`) is the same idea when you need an absolute expiry instead of an interval - Chapter 11's cache examples use that form. What to avoid is the unadorned integer and the arithmetic that pretends to document itself:

```
// Opaque: seconds? minutes? a typo for 30?
Cache::put('trending', $feed, 300);

// Still a magic number, just wearing a disguise.
Cache::put('trending', $feed, 5 * 60);
```

If a TTL is a product decision reused in more than one place, lift it to a named constant (`private const TRENDING_TTL = ...` backed by `minutes(5)`), the same way Chapter 19 names `TTL_MINUTES` next to the partner idempotency cache. The goal is that a reviewer can read the call site and know the unit without doing mental arithmetic or checking the framework docs for which epoch the integer means this year.

Admin panels lie in the default timezone

Operators debug production in the admin UI. If admin panel renders `created_at` in UTC while everyone on the team thinks in Pacific, someone will approve the wrong day's payout, reopen the wrong dispute, or swear a job "didn't run Monday" because the timestamp looks like Sunday evening.

Set a display default in the business timezone, and make sure date columns actually go through the datetime formatter - a bare text column ignores the panel timezone entirely:

```
/*
 * Panel default: America/Vancouver. Columns still need
 * ->dateTime() (or equivalent) so the timezone applies;
 * a plain TextColumn shows the raw UTC string.
 */
Table::configureUsing(function (Table $table): void {
    $table->timezone(
        CampaignConfig::BUSINESS_TIMEZONE,
    );
});
```

Label ambiguous fields in the UI ("Starts at (Pacific)") when compliance or finance will read them. For user-facing legal timestamps where UTC is the contract, label UTC explicitly instead of "helpfully" converting.

DST, ambiguous hours, and tests that freeze the wrong clock

`America/Vancouver` (and any IANA zone with daylight saving) has days that are 23 or 25 hours long. Local times between 2:00 and 3:00 may not exist, or may exist twice, on transition nights. Prefer:

- IANA names (`America/Vancouver`), never fixed offsets (`UTC-8`) or abbreviations (`PST` / `PDT`) as the source of truth. A fixed offset is wrong half the year wherever daylight saving applies - `UTC-8` is Pacific *Standard* Time, so every civil midnight you compute with it during PDT is an hour off. Abbreviations are worse: they name a single offset (`PST` vs `PDT`), they collide across regions (`CST` is Central in North America and China Standard Time elsewhere), and they force every call site to know which seasonal label is "current" instead of letting the

zone database apply the transition rules.
- Civil boundaries via `startOfDay()` / `startOfWeek()` in that zone, not `setTime(0, 0)` on a UTC instant you hope is midnight.
- Storing the *result* of calendar decisions when they must survive replay (Chapter 23's frozen `spendingDay`), instead of recomputing "today" later.

Tests are where these bugs either get caught or get permanently skipped. Pin time in the zone the rule uses:

```
public function test_streak_counts_pacific_days(): void
{
    CarbonImmutable::setTestNow(
        CarbonImmutable::parse(
            '2026-03-10 23:30:00',
            CampaignConfig::BUSINESS_TIMEZONE,
        ),
    );

    /*
     * act in Pacific evening...
     * assert active_date === '2026-03-10', not 03-11
     */
}
```

Also test the UTC side of the same instant (`06:30` the next day UTC during PDT) so a future refactor that drops the `->timezone( ... )` call fails loudly. Boundary cases worth a dedicated test: one second before and after Pacific midnight, the spring-forward gap, the fall-back overlap, and the first/last partial week of a campaign that does not start on Monday.

A short checklist before shipping calendar logic

1. `app.timezone` is UTC; business calendar TZ is a named constant.
2. Every `startOfDay` / `startOfWeek` / `toDateString` for product rules runs after converting to that constant.
3. Every scheduler entry that cares about wall-clock time calls `->timezone( ... )` explicitly.
4. Unique locks and idempotency keys embed dates or hours in the *same* zone as the schedule or product rule they protect.
5. Rolling windows use durations; civil windows use the business zone - never both for the same rule.
6. Cache TTLs and other durations use `minutes()` / `seconds()` / `hours()` / `days()` (or `now()->addMinutes( ... )`), not bare integers like `300`.
7. Admin date columns format through the panel timezone; labels say which zone the human is looking at.
8. Tests freeze time in the business zone and assert the civil date, not only the UTC timestamp.

Chapter recap

- Timezone bugs are silent correctness failures: wrong streak days, wrong week boundaries, wrong cron walls, wrong admin timestamps - usually without an exception.
- Keep `app.timezone` at UTC for instants; put civil-calendar semantics in an explicit business-timezone constant and convert only at the boundary where the product asks a calendar question.
- Schedulers and uniqueness keys must name the same timezone as the product rule they implement; "default app timezone" is not a safe stand-in for "Monday midnight Pacific."

- Decide whether "day" means a rolling duration or a civil calendar day, and encode that choice once - mixing the two is how redeem caps and streak counters drift apart.
- Write durations with unit-named helpers (`minutes(5)`, `seconds(30)`, `now()->addMinutes(5)`), not magic numbers - bare `300` hides the unit and has already bitten teams when cache TTLs changed from minutes to seconds.
- Admin UIs need the business timezone *and* actual datetime formatting; otherwise operators debug the wrong clock.
- Prefer IANA zones, freeze civil decisions that must survive replay, and write tests around Pacific midnight and DST transitions - those are the hours production will find if you don't.

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