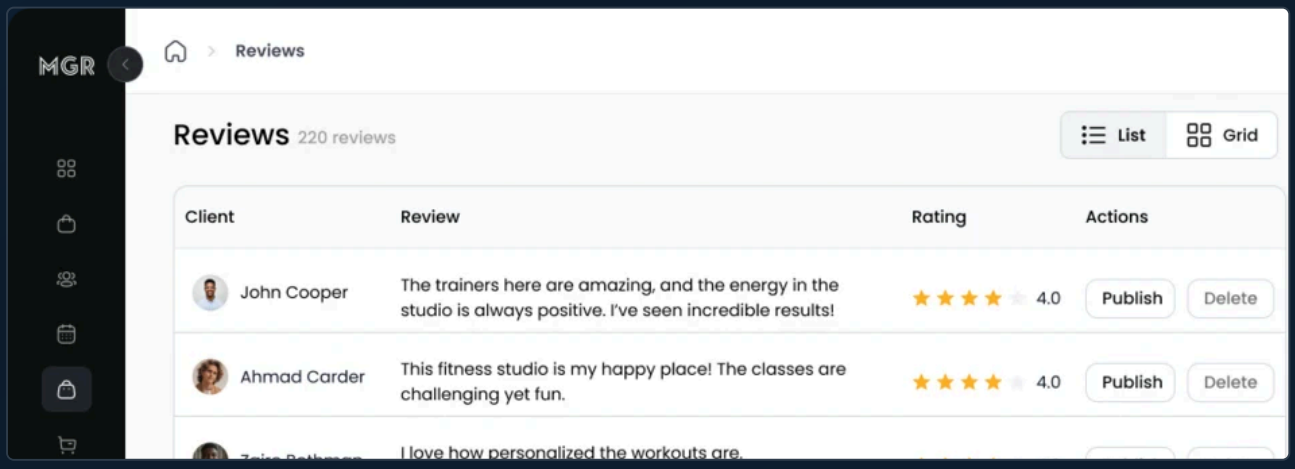
PLATFORM ENGINEERING

Two moderation gates, and the instructor who finds out last

Any product can add a star rating. The hard part is that the moment a studio can decide which reviews go public, the reviews stop meaning anything — unless the deciding is constrained, and constrained in the awkward case rather than the easy one. Approval and publication are separate decisions here, the people being reviewed cannot see unmoderated feedback about themselves, and only customers who actually turned up get asked.

2	6	100%	3
gates, each recording who decided	reviewable things, one visibility scope	made public by a named human	rebuilds in place, no data migration



CLIENT

Fitness studio management SaaS (anonymised)

ENGAGEMENT

Subsystem owned across a long-running platform partnership

INDUSTRY

B2B SaaS — fitness & wellness

TIMELINE

Mid-2023 to July 2026, rebuilt in place three times

DELIVERED

Two-gate moderation model, one visibility scope, per-class invitation scheduler, 6 send-time strategies, 15 tenant-editable templates

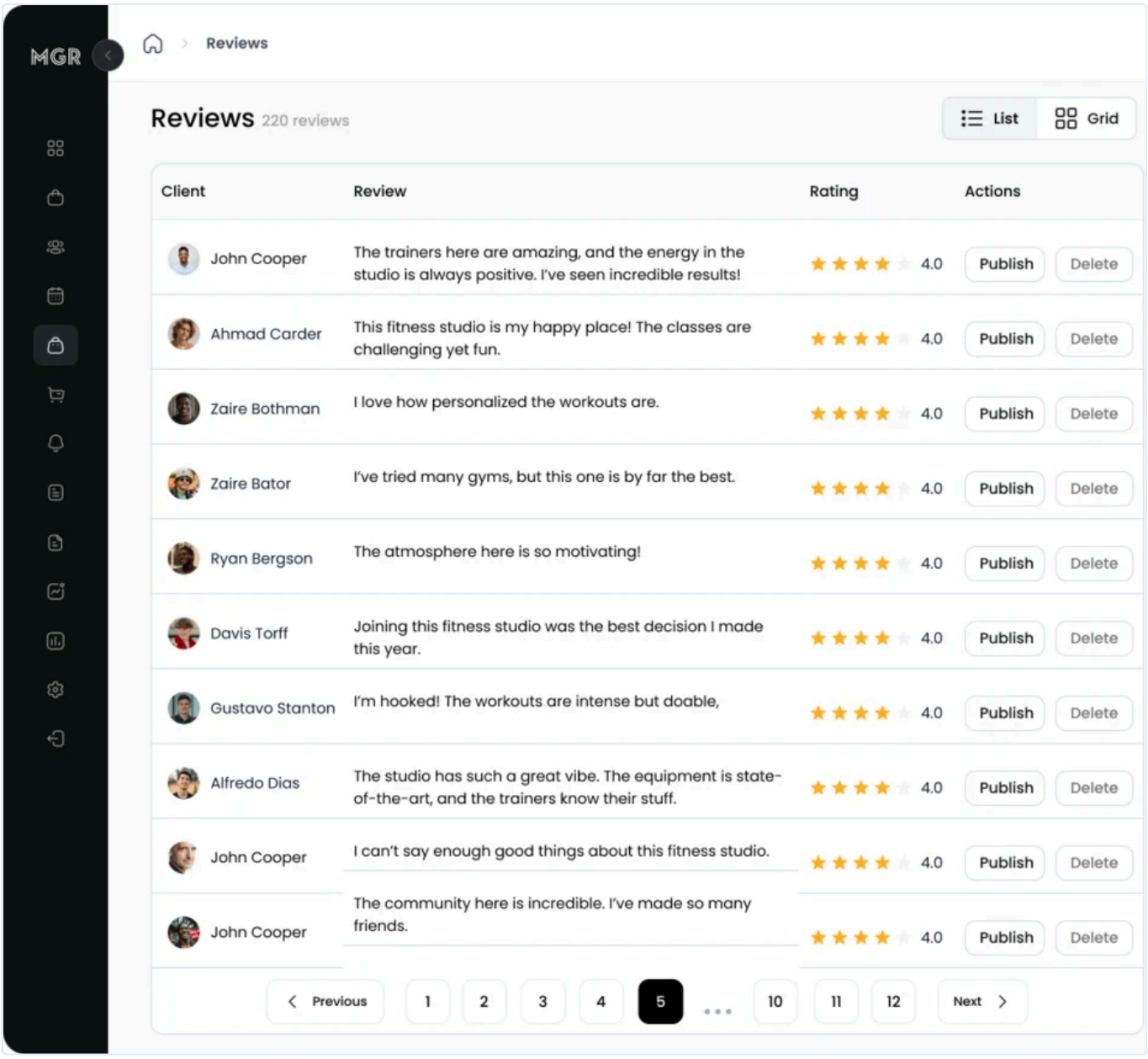
The problem: collecting feedback is easy, being trusted with it is not

Any product can add a star rating. The hard part is everything around it.

A studio wants reviews on its public pages, so it needs a way to stop abuse and spam appearing there. But the moment a studio can decide which reviews go public, the reviews stop meaning anything — unless the deciding is constrained. And the constraint has to hold in the awkward case, not the easy one: **the person with the strongest motive to bury a review is the instructor it is about**, and at a small studio that instructor may also be the person with the admin password.

There is a second, quieter problem. Asking for feedback is an act with a cost. Ask the wrong person and you get noise — somebody who booked and never showed up has nothing useful to say about the class. Ask the right person too often and you train them to ignore you. Ask at the wrong hour and you are a studio sending a push notification at eleven at night.

So the interesting parts of a reviews feature are not the reviews. They are: who gets asked, when, how often, who is allowed to see what came back, and who is allowed to make it public.



Nothing on this list is on a public page yet. Getting there requires somebody to press the button, and the platform records who pressed it — there is no auto-publish, no publish-if-rating-above-four, and no scheduled release.

This is the seventh write-up from this platform, and the oldest subsystem in the set. It first shipped in mid-2023 and has been revised more times than anything else we have written up here — which is the interesting part, because the moderation model it started with is the one it still has.

Act one: two decisions, not one

The obvious design is a single `approved` flag. A review is either live or it is not. We used two independent gates, each recording who made the call:

GATE	COLUMNS	MEANING
Approval	`approved_at`, `approved_by_id`	Cleared for internal visibility
Publication	`published_at`, `published_by_id`	Visible to the public

Splitting them separates two genuinely different judgements. *Is this legitimate feedback that our team should read and act on?* is a different question from *should this appear on our public page?* A blunt but fair review from a real member is usually yes to the first and, depending on the studio, no to the second. One flag forces those into a single decision and quietly loses the distinction.

Each gate records its own actor, so there is an answer to “who cleared this” and “who made it public” — two different people, potentially, and two different accountabilities.

A related default matters more than it looks. A studio can turn moderation off, in which case reviews are approved as they arrive. That does **not** make them public. Publication remains a deliberate human act either way. Turning off moderation buys a studio less work, not an unattended pipe to its public pages.

The visibility model is one scope, not scattered conditionals

Three audiences, three different views of the same table, expressed once:

```
scope :with_access, ->(entity) do
  if entity.is_a?(UserProfile)
    published.or(Review.where(user_profile_id: entity))
  elsif entity.is_a?(BusinessStaff)
    (entity.super_admin? || entity.manager?) ? all : approved
  end
end
```

Read it as a matrix:

VIEWER	UNAPPROVED	APPROVED, NOT PUBLISHED	PUBLISHED
The author	Sees own	Sees own	Yes
The public	No	No	Yes
Instructors, general staff	**No**	Yes	Yes
Owner, moderating	Yes	Yes	Yes

The load-bearing row is the third. **An instructor cannot see unmoderated feedback, including feedback about themselves.** Raw, unfiltered reviews are visible only to the studio’s most privileged role. Moderation is not a task the wider team helps with; it is a permission the wider team does not have.

The first row matters too, for a different reason. A customer can always see what they wrote, even while it is pending, because `published.or(user_profile_id: entity)` grants access to your own row regardless

of its state. Nothing is more corrosive to trust in a feedback form than submitting something and watching it vanish.

Everything reads through this one scope, and that is the actual engineering claim here — not that the rules are clever, but that there is exactly one place they live, so a new endpoint cannot accidentally invent its own answer.

Engineering deep dive

Why the notification fan-out has to agree with the visibility scope, and the parameters the moderation command deliberately refuses.

The notifications agree with the scopes

A design like this fails if the notifications leak what the scopes protect. Telling an instructor “you have a new review” the moment one lands would undo the whole thing — they would know it exists, and know it was withheld.

So the fan-out on submission is deliberately narrow. The author gets a confirmation. The studio’s owners get an alert. The instructor is not on that list:

```
def trigger_notifications
  Notifications::Reviews::UserReviewSubmittedNotificationWorker.perform_async(id)
  Notifications::Reviews::AdminReviewReceivedNotificationWorker.perform_async(id)
end
```

They appear only in the approval fan-out, gated on the review having actually been cleared:

```
def send_approved_notifications
  return unless approved_at.present?

  UserReviewApprovedNotificationWorker.perform_async(id)
  InstructorReviewReceivedNotificationWorker.perform_async(id) if reviewable_type ==
'BusinessStaff'
  AdminReviewApprovedNotificationWorker.perform_async(id)
end
```

The instructor is the last to know, by design. The scope and the notifications tell the same story, which is the property that makes the model real rather than decorative. Get this wrong in either direction and the other half stops being worth anything — a permission system whose notifications announce what it is hiding is a permission system with a side channel.

Staff can moderate a review, but never edit one

The moderation commands change moderation state and nothing else. The update command that staff can reach does not accept `content` or `rating` — those parameters are simply not permitted, and a test asserts that passing them changes nothing.

> A studio can decline to publish a review. It cannot rewrite one and then publish the edit. That distinction is worth more than it costs, because it means “approved” carries a specific promise: a human read this and let it through, unaltered.

Act two: asking the right person at the right moment

The other half of a reviews system is the invitation. Ours is scheduled per class rather than swept up in a nightly batch.

When a class is saved, a scheduler computes when to ask, enqueues a delayed job, and stores the job identifier on the class itself:

```
job_id = SendClassReviewRequestWorker.perform_at(job_time, resource_offering.id)
resource_offering.update_columns(review_request_job_id: job_id)
```

Keeping the job id on the record is what makes the rest possible, and it is the most portable idea on this page. **Classes move.** Schedules get rebuilt on a Sunday evening. A review request that was correct when it was scheduled is wrong the moment its class shifts by three hours, and a system with no handle on its own pending work cannot do anything about that.

Re-saving an unchanged class is a no-op, but if the end time changed, that guard falls through and the request is rescheduled against the new time:

```
def already_scheduled_without_changes?
  !resource_offering.end_time_previously_changed? &&
    !resource_offering.new_record? &&
    resource_offering.review_request_job_id.present?
end
```

Cancelled classes are skipped, and so is retroactive scheduling for classes already in the past. Every one of those decisions logs a reason — `already_scheduled`, `past_start_time`, `cancelled`. When somebody asks in six months why a particular class never generated a review request, the answer is in the log rather than in a re-reading of the code. The workflow engine on this same platform reaches for the same pattern when a journey has to park for days and then act on a schedule that may have moved underneath it.

Only people who turned up are asked

One line carries most of the credibility of the whole feature:

```
resource_offering.resource_reservations
  .where.not(signed_in_at: nil)
  .where(state: "reserved")
  .includes(:user_profile)
```

`signed_in_at` is the check-in timestamp. No check-in, no invitation. Somebody who booked and did not come is never asked what they thought of a class they did not attend, which keeps no-shows from quietly becoming one-star reviews. It is also the sentence that makes a published review mean something: every class review on this platform comes from a person the system can prove was in the room.

There is a small, human detail alongside it. A member who books for themselves and a guest holds two reservations for the same class, and the naive implementation asks them twice. The code says so out loud:

```
# NOTE: If user has more than one reservation (maybe guests)
#       - we only need one review request
review_request = reservation.user_profile.review_requests
  .find_or_create_by(review_requestable: reservation) do |rr|
    rr.request_type = "system"
  end
```

`find_or_create_by` also means the worker is safe to run twice, which matters because it is a background job with retries.

Engineering deep dive

Six ways to answer “when?”, the window arithmetic the naive version gets wrong, and why the timezone is resolved per location rather than per studio.

The send time comes from a small analyzer with six strategies: at the end of class, a fixed number of hours after, within a morning, afternoon or evening window, or at a specific time of day.

The window logic is the part worth showing, because the naive version is subtly wrong:

```
# Schedules a datetime within a [start_hour, end_hour) window
# on the same day.
#   - Before the window start → the window start.
#   - Inside the window       → now.
#   - At or after the end     → tomorrow's window start.
def schedule_in_window(reference_time, start_hour, end_hour)
  window_start = reference_time.change(hour: start_hour, min: 0, sec: 0)
  window_end   = reference_time.change(hour: end_hour,   min: 0, sec: 0)

  return window_start   if reference_time < window_start
  return reference_time if reference_time ≥ window_start &&
                           reference_time < window_end
```

A nine-p.m. class with an evening preference rolls to the following evening rather than firing immediately or being dropped. All three positions relative to the window are handled, and the comment says so, so the next person to read it does not have to derive the behaviour from the branches.

The timezone is resolved per location, not per studio

```
def offering_zone
  @offering_zone |⊨ ActiveSupport::TimeZone[
    resource_offering.location&.time_zone || Time.zone.name
  ]
end
```

A studio with sites in two timezones asks each set of members at the right local hour. “Evening” means evening where the class happened.

This is stricter than the platform’s reporting subsystem, which resolves timezone at the account level. Both are defensible for what they do — a report is read by the account holder, so the account’s zone is the right frame — but something that ends up buzzing in a member’s pocket needs the finer granularity, and it was worth the extra join to get it.

Malformed configuration degrades instead of raising: an unparseable custom time logs a warning and falls back to the default rather than taking down the job for the whole studio.

Act three: frequency, and configuration that is checked

A regular who trains five times a week must not be asked five times a week. The platform offers two ways to express that, both evaluated per customer. **By elapsed time** — do not ask if this customer has been asked within the window. **By classes attended** — count the classes this customer has attended since their last request, and ask only once that count is reached:

```
requests = user_profile.review_requests.by_type(review_type)
last_request_at = requests.maximum(:created_at)

scope = user_profile.resource_reservations
  .where(business_account: business_account)
  .where(state: "reserved")
  .where.not(signed_in_at: nil)
scope = scope.where("signed_in_at > ?", last_request_at) if last_request_at

next if scope.count < threshold
```

The second is the better model for a fitness business, because it scales with how much somebody actually uses the studio rather than with the calendar. A five-a-week regular and a fortnightly visitor both get asked every N visits, which is the behaviour a studio owner means when they say “do not spam my members.”

Reviews are not limited to classes either. The platform models six reviewable things, with a mapping from internal model names to the vocabulary the product actually uses:

```
REVIEWABLE_TYPES_MAPPING = {
  BusinessStaff:       'instructor',
  Resource:            'class',
  ResourceOffering:    'attended_class',
  Location:            'location',
  BusinessAccount:     'studio',
  ResourceReservation: 'reservation'
}.freeze
```

Instructor, location and studio feedback run on their own cadences, deliberately much slower than class feedback — you want a view of an instructor over a season, not after every session. One domain detail worth the sentence it costs: when a class was covered by a substitute, the review is attributed to whoever actually taught it, not to whoever was on the schedule.

The configuration is validated, not merely stored

Settings systems usually accept anything and fail later, in a worker, at three in the morning, for one tenant. These are validated when they are saved. Strategies are checked against an allowlist and the error names the legal values rather than saying “invalid”. A non-manual trigger mode must have a positive threshold. A custom time of day is regex-checked, with a message showing the expected format.

And one rule encodes a genuine domain constraint rather than a type check:

```
if mode == "after_x_classes" && type != :class
  errors.add("#{type}_review_trigger_mode",
             "is invalid for this review type")
  next
end
```

Counting classes attended is meaningful for class feedback. It is meaningless for “how do you feel about the studio”, which should be time-based. Rather than leave that as tribal knowledge in someone’s head, the incoherent combination simply cannot be saved.

Turning the system down is layered, from the whole studio to a single review type: off entirely, automation off but manual requests still allowed, automation off for one review type, or that type removed from the studio’s allowed list. Four independent gates, so a studio that wants instructor feedback but not location feedback does not need a developer — and one deliberate exception, because a human pressing “ask this member” should not be blocked by a switch that governs automation:

```
# Manual review requests should always be delivered
# even when config is "manual_only"
return false if rr&.manual?
return true  if mode == 'manual_only'
```

Channels

Email, SMS and push are each available per studio and each independently gated — by the studio’s own switch, by the customer’s notification preferences, by whether the template is enabled, and by whether the channel is even possible for that person (no phone number, no SMS). Fifteen request templates cover

five review types across the three channels, and every one is a tenant-editable template rather than a string in the codebase — the same system as the notification templates we later put an AI layer in front of.

Alongside those, pending requests are attached to the customer's profile payload, so the consumer web app can surface the prompt in-product on their next visit without any message being sent at all. That path reuses the same suppression rules as everything else, so a request that has been skipped, deferred, fulfilled or expired does not appear.

Skip, later, and never

There is no `state` column on a review request. State is four nullable timestamps and a link, and each one is a first-class scope:

```
scope :not_skipped,  → { where(skipped_at: nil) }
scope :not_expired,  → { where("expires_at > ?", Time.now) }
scope :not_reviewed, → { where(review_id: nil) }
scope :not_delayed,  → { where(next_trigger_at: nil)
                           .or(where(next_trigger_at: ..Time.now)) }

scope :actionable,   → { not_reviewed.not_expired.not_skipped.not_delayed }
```

`actionable` is the whole design in one line: four independent reasons not to put a prompt in front of somebody, composed into a single predicate that every read path uses.

The important distinction is that **"not now" and "never" are different things.** Skipping sets `skipped_at` and ends it. Deferring sets `next_trigger_at`, which hides the prompt until that moment passes. They are separate columns because they are separate intentions, and treating them the same is how products end up nagging people who already said no.

Deferring also has to protect the request from its own expiry, or the feature quietly lies to the customer:

```
if review_request.expires_at < review_request.next_trigger_at
  review_request.update(expires_at: 2.days.from_now)
end
```

Without those three lines, a customer who snoozes past the expiry window would find that "remind me tomorrow" had meant "never" — which is the kind of bug nobody reports and everybody notices.

Expiry itself is passive. The invitation simply stops being actionable: it disappears from the in-product prompt, the notification workers skip it, and a submission against an expired request is rejected. Nothing sweeps, and nothing needs to.

Stated precisely

Four things this system does not do, said plainly, because the alternative is a reader discovering them later.

- **It is not real time.** Review requests are scheduled work. A class ending at seven with a two-hour offset produces an invitation at nine, via a background queue, subject to its latency.
- **Review links require a login.** The link in an email or SMS points at a protected page. It is not a signed one-click token and it is not a magic link.
- **There are no reminders.** An invitation is sent once. A customer can defer it themselves, but nothing re-sends or nudges. Whether that is a virtue or a gap depends on the studio; it is a deliberate absence rather than an oversight.
- **Nothing is public without a person.** No auto-publish, no publish-if-rating-above-four, no scheduled release. Publication is always an act by a named member of staff.

Results

The subsystem has been in production since 2023 and rebuilt in place three times — moderation and visibility in 2023, automated invitations in 2024, the scheduling engine and per-customer frequency modes in 2026 — without a data migration or a change to the public contract, because the moderation model has held throughout. That is the outcome worth pointing at: the part that was hardest to get right is the part that never needed revisiting.

WHAT THE DESIGN BOUGHT	HOW
One place that answers “who can see this”	Six reviewable types, three audiences and two moderation gates, and adding an endpoint means calling `with_access` rather than reasoning about the matrix again
Two accountabilities instead of one flag	Who cleared it and who made it public are separate columns with separate actors, so the audit trail matches the decisions actually taken
Invitations that survive a schedule change	Pending work is addressable, so a class that moves takes its review request with it
Feedback tied to attendance	Every class review comes from somebody the system can prove was in the room
Configuration that cannot be set to something incoherent	Validated on write with allowlists, threshold checks and a regex, failing at the point of entry with a message naming the legal values
“Remind me later” that means later	Skip and defer are separate columns, and deferring extends the expiry so a snooze cannot become a silent refusal

Test coverage sits where the risk is: roughly 2,900 lines across 36 spec files, with the heaviest concentrations on review submission, the per-class worker, the scheduling engine, and both moderation gates. Review volume and response rates are the client’s to publish rather than ours to estimate.

What we would do next

Add a self-approval guard. The role boundary is solid — general staff can neither see nor clear unmoderated feedback. What is missing is the edge case at the other end: at a small studio, the instructor

being reviewed may be the same person who holds the owner account. The guard is a comparison between the acting staff member and the review's subject, and it turns a strong convention into an invariant. It is the first thing we would build here, and we would rather name it than let a reader assume it is already there.

Make the moderation queue a queue. Today reviews are moderated one at a time from a general list. A pending view with bulk approve, and a filter for approved-but-unpublished, would match how the work is actually done — and the scopes to build it already exist.

Expose the send-timing controls in the admin console. The engine supports six strategies with per-location timezones, and the validation is already there to keep studio owners out of trouble. The interface has not caught up with the engine, so choosing a strategy currently needs a developer. This is a UI gap in front of finished machinery, which is a good problem to have and a bad one to leave.

Escalate low ratings. A one-star review and a five-star review currently travel the same path. Routing very low ratings straight to an owner, before the normal moderation cycle, is a small change with an obvious operational payoff.

Give the business a right of reply. The most requested feature in any review system, and the one thing the schema has no room for. A published response, itself moderated, would let a studio answer criticism instead of only choosing whether to show it.

Sanitise content on the way out. Review text is written by customers and rendered in both consoles. It should be escaped or sanitised explicitly at the boundary rather than relying on the rendering layer to keep doing it.

If this sounds like your system

The shape recurs anywhere a product asks its users what they think and then shows some of the answers to other users: a moderation flag that means two things at once, staff who can see feedback about themselves before anyone has read it, and a prompt that keeps appearing for somebody who already declined it.

The transferable decisions are these. **Split a flag that is answering two questions**, and record an actor on each half, because "we should read this" and "the world should see this" have different consequences and deserve different accountabilities. **Put the whole visibility model in one place** — rules scattered across controllers will eventually disagree with each other, and the disagreement will be discovered by the person it hurts. **Make the notifications agree with the permissions**, or the permissions have a side channel. **Give your scheduled work a handle**, so a pending job can be moved when the thing it describes moves. And **keep "not now" and "never" in different columns**, because conflating them is how software becomes something people mute.

This is the half of product engineering that never shows up in a feature list — the same instinct that decides what an in-product assistant is structurally permitted to do rather than merely instructed not to. If you have a moderation queue nobody trusts, or a prompt your users have learned to dismiss, tell us about it — it is a problem with a known shape.

Tech stack

BACKEND

Ruby 3.3.9 · Rails 7.1 (API)

DATA

PostgreSQL 16 · Redis · Polymorphic reviewables · Timestamp-based state

ASYNC

Sidekiq · Per-class delayed jobs · Addressable job ids

AUTHORISATION

One ActiveRecord scope · Role-gated moderation · Two recorded actors

MESSAGING

SendGrid · Twilio · OneSignal · Handlebars templates

FRONTEND

Next.js admin console · TypeScript

TESTING

RSpec

Frequently asked questions

Why two gates instead of one `approved` flag?

Because “our team should read this” and “the public should see this” are different decisions with different consequences, and a single flag forces them together. Two gates also produce two audit trails: who cleared it, and who made it public. In practice most studios move a review through both, but the ones that care about the distinction can use it — and turning moderation off never turns publication off.

Can an instructor see bad reviews about themselves before anyone moderates them?

No. General staff, including instructors, see only approved reviews, and an instructor is not notified about a review until after it has been approved. Unmoderated feedback is visible only to the studio’s most privileged role. The remaining gap is the owner-operator case — a small studio where the instructor is also the account holder — which is why a self-approval guard is the next piece of work rather than a claim we make today.

Can a studio edit a review before publishing it?

No. The update command available to staff does not accept review content or rating — only the moderation state and the review’s target. A studio can decline to publish something; it cannot publish an edited version of it. A test asserts that passing content or a rating changes nothing.

How do you avoid pestering regulars?

Frequency is evaluated per customer, either by elapsed days or by classes attended since that customer’s last request. The second is the one to use for a busy studio, because it scales with attendance rather than the calendar. On top of that, a customer can skip an invitation permanently or defer it, and deferring extends the expiry so “remind me later” cannot silently become “never”.

What happens when a class gets rescheduled?

The pending review request is rescheduled with it. The scheduler keeps the job id on the class, so when the end time changes it recomputes the send time and replaces the scheduled work. Cancelled classes drop out. The worker also re-derives everything at run time and creates requests idempotently, so a stale job cannot produce a duplicate invitation.

Is this real time?

No, and it should not be. Review requests are deliberately delayed — the default asks a day after class, and a studio can choose an evening window or a specific hour instead. Delivery is a background job, so it trails by queue latency. The one thing that is resolved precisely is the timezone: “evening” means evening at the location where the class was taught.

Have a system that looks like this?

Bring us the part you have been putting off — the wide surface of manual configuration nobody finishes, the AI feature you are not sure how to make safe, the legacy job that keeps failing. We will tell you what we would build and what we would leave alone.

Book a 30-minute architecture review

`booleans.in` · `contact@booleans.in`