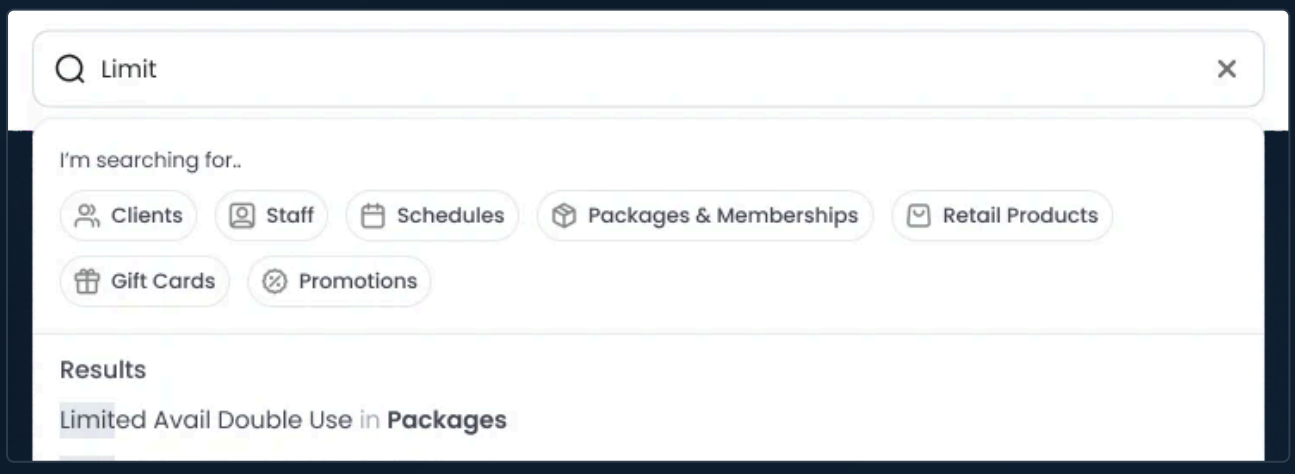
PLATFORM ENGINEERING

One search box, nineteen indices, fuzzy matching switched off

Search advice is written for people who do not know what they want. Front-desk staff are the opposite: they have a member's name on a sticky note or a transaction id out of a support email, and they want one record, first. So the first thing this search does is not a search — it works out what the user has already told it, then narrows as hard as it can.

19	6	1	10k
entity types behind one search box	query shapes classified before searching	index for a typed id, not fourteen	record import, one indexing job



CLIENT

Fitness studio management SaaS (anonymised)

ENGAGEMENT

Rebuild inside a long-running platform partnership

INDUSTRY

B2B SaaS — fitness & wellness

TIMELINE

Shipped thin July 2024, rebuilt February–May 2026

DELIVERED

19 searchable entities, 4 search objects, a Redis-deduplicated indexing pipeline, entity-keyed history

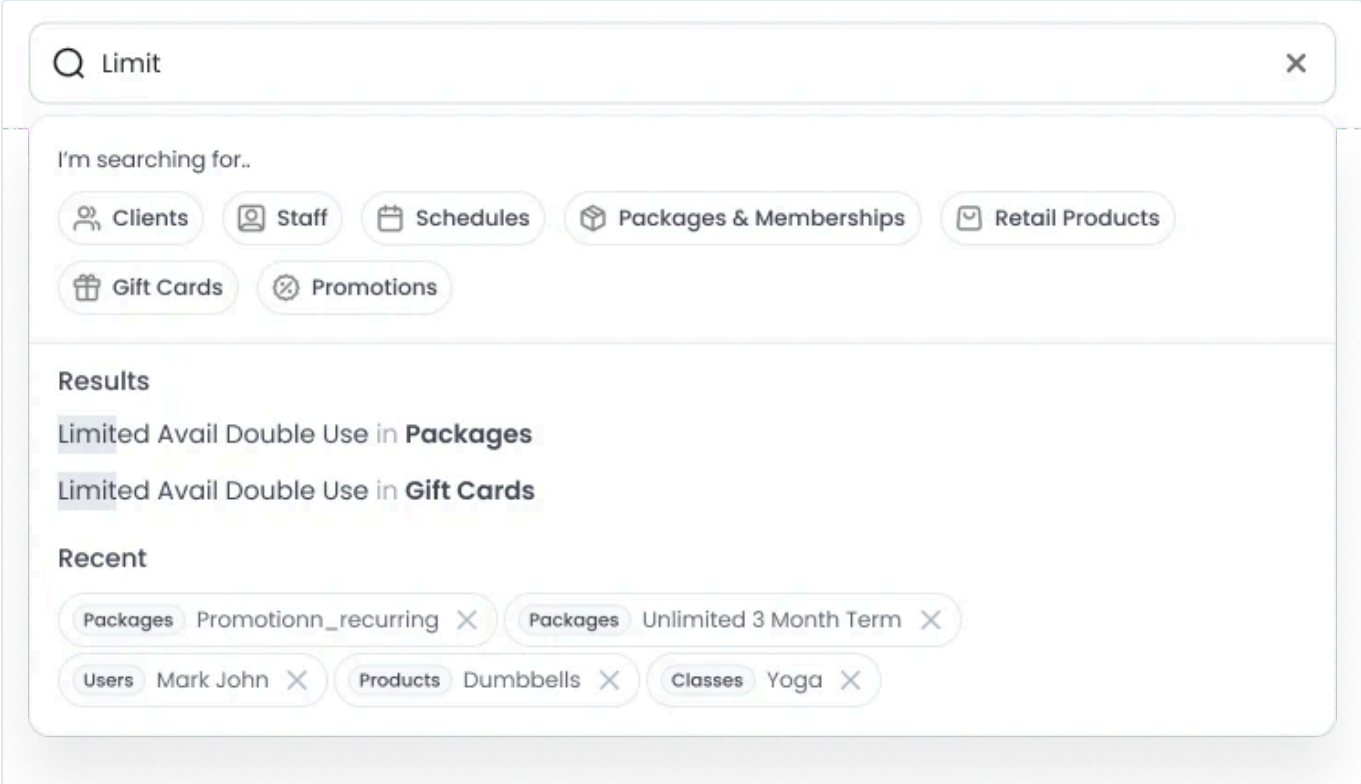
The problem: staff already know what they are looking for

Almost everything written about search is written for people who do not know what they want. Someone browsing a shop needs recall, forgiveness and suggestions — cast a wide net and let them wander through it.

An admin console is the opposite. A studio's front-desk staff know exactly what they are after. They have a member's name on a sticky note, or a transaction id copied out of a support email, or half a product name they are about to finish typing. They want one record, they want it first, and if it is not first they will type three more characters rather than scroll.

That inverts most of the defaults. Fuzzy matching stops being generosity and becomes noise. Ranking by popularity is meaningless when the target is one specific person. Meanwhile a problem appears that consumer search does not have: the thing being looked for could be any of **nineteen kinds of record** — a member, an instructor, a class, a package, an order, a transaction, a gift card, a penalty, a promotion, or one of the platform's own configuration objects like a notification template, a workflow or a landing page — and the person typing should not have to say which.

So the design question was never "how do we match more things." It was: **how do we work out what the user has already told us, and then narrow as hard as possible.**



Two things in one frame. The query is `Limit` and the highlight falls on **Limi** inside "Limited Avail Double Use" — that is prefix matching, and each hit says which index it came from. Underneath, "Recent" is not a list of things people typed. Each chip is a *record* with a type badge, which turns out to matter more than it looks.

This is the sixth write-up from this platform. It is the one with no commercial hook at all: nothing here converts better or bills more. It is entirely about judgement — which defaults to reject, where to spend simplicity, and which half of a search feature actually decides whether it scales.

Act one: classify the query before searching anything

The first thing this search does is not a search. It is a look at what was typed.

```
if @raw.empty?
  @kind = :blank
elsif (model = match_friendly_id(@raw))
  @kind = :friendly_id
elsif @raw.match?(UUID_PATTERN)
  @kind = :uuid
elsif @raw.match?(EMAIL_PATTERN)
  @kind = :email
elsif phone_shape?(@raw)
  @kind = :phone
else
  @kind = :text
end
```

Six kinds, in priority order, decided in one small value object before any index is touched. The kind then drives everything downstream: which searcher runs, and which exact-match rule applies to the results.

WHAT WAS TYPED	WHAT THE SEARCH DOES WITH IT
Friendly id — `UP1234`	One index. Relevance ranking skipped entirely, because either the id matches or it does not
UUID — `3f2a8c1e-…`	Direct id lookup, no text matching anywhere
Email — `ana@studio.com`	Means a person. An exact match is pinned to the top
Phone — `(415) 555-2929`	Reduced to digits, so formatting stops mattering
Text — `mark`	Prefix-matched across the indices, exact names pinned
Blank	Nothing is searched

The highest-value branch is the first one. Every record on this platform carries a human-readable id with a type prefix, and those prefixes are unambiguous:

```
FRIENDLY_ID_PATTERNS = {
  /\AUP\d*\z/ ⇒ UserProfile,
  /\AUT\d*\z/ ⇒ UserTransaction,
  /\AOR\d*\z/ ⇒ Order,
  /\AGI\d*\z/ ⇒ GiftCardInstance,
  # ... twelve in total
}.freeze
```

So `UP1234` is not a text query that happens to match a member. It is a statement that the user wants one specific member. A support agent pasting an id out of a ticket gets a **single-index lookup instead of a fourteen-index fan-out**, and the code knows it can skip ranking altogether, because either the id matches or it does not.

Normalisation is per-kind, and it is what makes the ranking work later. A phone number keeps only its digits, so `(415) 555-2929` and `4155552929` are the same query. Text is lowercased so name comparison is case-insensitive. The class carries worked examples in its own comment, which is how it should be — the behaviour is legible without running it.

Engineering deep dive

Two searchers behind one tuple shape, nineteen entities in fourteen round-trips, and the `types` parameter that makes the whole pipeline reusable.

Two searchers, one shape

Id-shaped queries go down a fast path; everything else goes to full text. Both return the same `[record, score, type]` tuples, which is why the orchestrating command has no branching after the dispatch:

```
scored_hits, total_count = fetch_results(query)
ranked  = ResultRanker.new(query: query).call(scored_hits)
results = build_payloads(paginate(ranked))
```

Five steps of work and four collaborators — `Query`, `IdSearcher`, `FullTextSearcher`, `ResultRanker`. It is worth noting *when* that structure appeared: exact-match ranking was written inline on 29 April 2026 and extracted into those objects the following day. The feature was built, then immediately given somewhere to live. A one-day gap between shipping a behaviour and giving it a home is a better signal about a codebase than any amount of architecture documentation.

Nineteen entities, fourteen round-trips

Text search fans out across the indices, but not naively. Six models share an identical field configuration — they are all found by id and friendly id rather than by name — so they are batched into a single multi-index query rather than six separate ones:

```
ID_FRIENDLY_ID_ONLY_MODELS = Set[
  ResourceOffering, PackageInstance, RecurringUserPackage,
  ResourceReservation, RetailInstance, UserPenalty
].freeze
```

Nineteen searchable entities, thirteen individual queries plus one batched multi-index query. Page one asks each index for a page's worth of documents rather than a fixed large number, so the fan-out cost scales with what is actually being displayed.

The trade is visible in the product and worth stating plainly: those six are findable by id, not by name. A class is found by its id today, not its title. That was a query-efficiency decision and it is on the roadmap below, now that the batching machinery exists to revisit it.

The same pipeline, narrowed

The pipeline takes a `types` parameter, and everything downstream respects it:

```
def models_to_search
  return ALL_SEARCHABLE_MODELS.dup if @types.blank?

  @types.map { |t| TYPE_TO_MODEL[t] }.compact.uniq
end
```

`types=User` runs the identical classification, boosting, reranking and payload building against a single index. The filter chips in the console are exactly this, and so is any caller that wants a scoped search rather than a global one.

Around it sits a forty-one entry alias map, so `users`, `user` and `userprofile` all resolve to the same thing, as do `gift_cards`, `gift_cards_instance` and `giftcardinstance`, with a fallback that tries `underscore` and case-insensitive matching before giving up. Callers written against older names keep working without anyone having to find them first. It is the same instinct as keeping old

> report URLs alive as real routes rather than redirects — on the same platform, by the same reasoning, a couple of months apart.

Act two: two arguments against the obvious setting

The relevance work in this subsystem is almost entirely subtractive. Two choices, both arguing *against* the default, and both with the reasoning written into the code rather than into a wiki page nobody will read.

Fuzzy matching is switched off

```
# NOTE: misspellings are disabled. Short query tokens like "mai" or "oi"
# otherwise fuzzy-match unrelated tokens via edit-distance 1, leaking
# irrelevant records (e.g., transactions whose notes contain "AI" or "of")
# into results. Typeahead UX does not need fuzziness; users self-correct
# as they type.
```

That is the most useful comment in the codebase, and the argument generalises well past this product:

> Fuzzy matching is a recall mechanism, and a typeahead has no recall problem. The user is mid-word and can fix their own typing.

On a three-character token, edit distance 1 matches an enormous amount of nothing. Type `mai` and you start pulling in every transaction whose note contains “AI” or “of”. The setting is applied on every query path rather than just the main one, and turning it off is what lets the ranking stay simple two sections from now, because the candidate set arriving from Elasticsearch is already tight.

An order cannot be found by the name of the person who placed it

The per-model field boosts are where the second argument lives, and the comment above them states it directly: person records search across identity fields, and non-person records *intentionally* exclude name fields.

```
'Product'         ⇒ ['id^30', 'friendly_id^20', 'name^15', 'product_type^12'],
'Order'           ⇒ ['id^30', 'friendly_id^20', 'order_note^10'],
'UserTransaction' ⇒ ['id^30', 'friendly_id^20', 'notes^10'],
```

An `Order` document *does* contain the buyer’s full name. It is indexed. `Order` simply does not search it. Same for transactions, which index the client’s name and never query it.

That is the difference between searching a member’s name and getting one person, and searching a member’s name and getting one person buried under four hundred of her transactions. Both are defensible index designs. Only one is usable in a box that shows twenty-five rows. The data is available

and the query deliberately declines to use it, which is the kind of decision that only looks obvious after somebody has made it.

Prefix matching that survives its own migrations

```
# Two-phase strategy:
#   - Try :word_start (prefix matching).
#   - If that returns no hits (typically because a field doesn't have
#     an autocomplete subfield in the current index mapping), fall back
#     to :word so the query still works during reindex/migrations.
def single_model_search(model)
  hits, total = run_query(model: model, match_type: :word_start)
  hits, total = run_query(model: model, match_type: :word) if hits.empty?
  [hits, total]
end
```

Prefix matching depends on autocomplete subfields existing in the index mapping, and during a mapping change they may not yet. A prefix query against a missing subfield returns nothing rather than failing — which is the worst available outcome, because search looks broken while every health check says the system is fine. The fallback trades relevance for availability for exactly as long as a reindex takes, and not a moment longer.

Act three: the reranker, which is smaller than the word suggests

Worth defining before describing, because “custom reranker” invites the wrong picture. There is no machine learning here. No learned weights, no recency, no popularity, no entity-type priority, no rescoring inside the search engine. It pins records that **exactly** match what the user typed to the top of the list, and leaves everything below in Elasticsearch’s own order. That is the entire mechanism, and its own comment explains why that is enough:

```
# Why exact-match-only:
#   With ES misspellings disabled (see FullTextSearcher), the candidate
#   set is already tightly relevant -- ES TF/IDF correctly orders prefix
#   matches. The only ranking signal ES can't produce on its own is
#   "this record EXACTLY matches what the user typed", which is the
#   single boost we add here.
```

The two acts before this one are what earn the simplicity. Because fuzziness is off and commerce records cannot match on names, what comes back is already relevant and TF/IDF orders it sensibly. The one thing Elasticsearch cannot know is that a document is not merely a good match but *the* thing the user asked for — somebody typing a complete email address is not browsing.

So exact matches are promoted to infinity and the list is sorted, with the record id as a stable tie-break:

```
scored_results.each do |entry|
  entry[1] = Float::INFINITY if exact_match?(entry)
end
```

The only magic number in the ranking layer is infinity. And it applies only where “exact” means something — the id paths are explicitly skipped, because there every hit is already an exact id match.

The behaviour is unit-tested in both directions. Exact email, phone and name matches are asserted to move to the front even when Elasticsearch scored them lower, and a prefix-only match is asserted **not** to be promoted. That negative test is what makes the positive ones mean anything.

Being precise about what this is not: the ranking rules are unit-tested, they are not measured. There is no evaluation set, no NDCG or MRR instrumentation and no A/B test behind any of this. What is proven is that the rules behave as described. Below the pinned rows, the ordering is Elasticsearch’s, assembled from independent per-index queries — deliberately controlled at the top, approximate underneath.

Act four: the half that actually decides whether it scales

The query side of a search is what gets demoed. The indexing side is what breaks.

Searchkick’s automatic callbacks are switched off on all nineteen models. Instead, a save enqueues an id and returns — and then two independent layers of deduplication do the real work.

1. **A save pushes an id into Redis and returns**

   An `after_commit` hook adds the record id to a pending set. `after_commit` rather than `after_save`, so nothing is ever queued for a record that did not durably exist. A record updated fifty times in ten seconds is one pending id, not fifty reindex operations.

2. **At most one job is scheduled per ten seconds**

   The callback tries to take a Redis lock with `SET NX EX`. Only the caller that wins it schedules a batch worker, ten seconds out. Ten thousand callbacks inside that window schedule zero additional jobs.

3. **The worker drains with an atomic pop**

   `SPOP` claims up to a hundred ids at a time, so two workers can never claim the same ids without any coordination between them, and the batch is reindexed in one bulk call.

4. **It reschedules itself while work remains**

   The queue drains at a bounded rate rather than in one unbounded job, which is what keeps a bulk import from becoming an incident.

The ten-second delay is the feature, not the cost. It is what turns a bulk import touching ten thousand records into **one** scheduled job and a set of ten thousand ids, instead of ten thousand jobs racing each other to the same index.

Step 2 is thirty seconds of Redis documentation doing an enormous amount of work:

```
# Only one job scheduled per SCHEDULE_INTERVAL from callbacks.
set = conn.set(lock_key, "1", nx: true, ex: SCHEDULE_INTERVAL)
SearchIndex::ReindexBatchWorker.perform_in(SCHEDULE_INTERVAL.seconds) if set
```

Engineering deep dive

The two-line ordering detail inside delete that stops a deleted record being resurrected in the index.

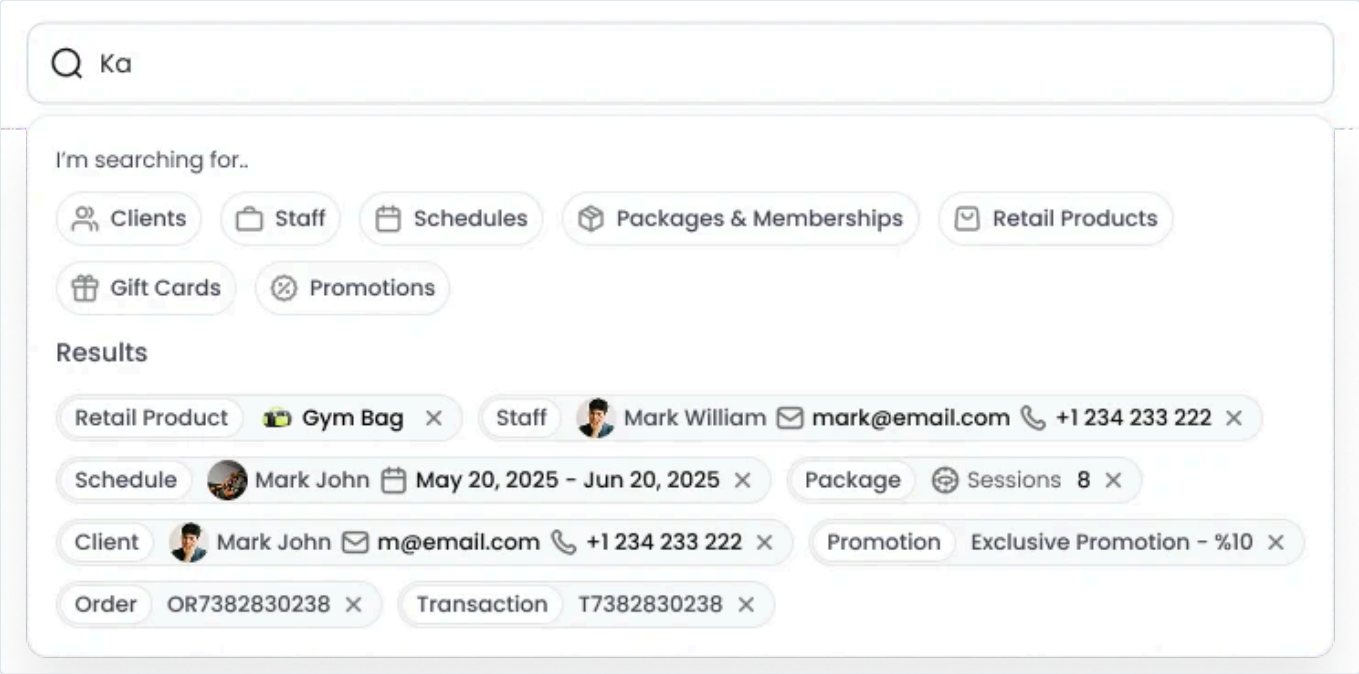
```
def enqueue_delete(model_name, id)
  return if model_name.blank? || id.blank?

  redis do |conn|
    conn.srem(upsert_key(model_name), id.to_s)
    conn.sadd(delete_key(model_name), id.to_s)
  end
  schedule_batch_job
end
```

The `srem` before the `sadd` is the part worth pausing on.

A record created and then destroyed inside the same ten-second window would otherwise sit in the pending-upsert set *and* the pending-delete set at once. The batch would then try to index a row that no longer exists, which either raises or writes a document for a deleted record depending on which set is drained first. Pulling the id out of the upsert set before adding it to the delete set makes “create then delete” collapse to “delete”, deterministically.

This is the class of bug that batching introduces and that nothing warns you about. The window that buys the deduplication is the same window in which a record’s whole lifecycle can happen. Two lines, in the right order, and the batch cannot resurrect a deleted record.

What a result is, and why history is keyed on records



Every row here carries different fields, and that is deliberate. Staff bring an email and a phone; a schedule brings its date range; a package brings a session count; an order and a transaction bring only their ids; a promotion brings its discount. A result knows what kind of thing it is and what a human needs to see to recognise it.

Result payloads are self-sufficient — name, image, amount, state, plus the ids and slug needed to deep-link — so selecting a result never triggers a second request to find out what it was. Associations are preloaded per entity type per page rather than per record, which is the difference between one query and twenty-five.

Then there is the detail from the hero image, which is the smallest decision on this page and possibly the most instructive. **Recent searches store the entity a staff member opened, not the string they typed**, with a unique index on the account, the staff member and the record.

That choice is what makes the feature useful rather than decorative. Visiting the same member twice refreshes one row instead of appending a duplicate, so the list reads as “the records you have been working on” rather than a log of half-typed words. A background job keeps the most recent hundred per person. And remembered records render through the *same* payload builder as live search results, so a member you looked at yesterday and a member you just found look identical, and there is one place to change what a result card shows.

Stated precisely

Five things that a search page usually leaves vague, and that we would rather put in writing.

- **It is not real-time, on purpose.** Indexing trails a write by roughly ten to twenty seconds: up to ten seconds of deliberate batching, plus the worker run. That delay is not latency waiting to be optimised away — it is what buys the deduplication above.

- **Fourteen Elasticsearch round-trips** for an unfiltered text search. Filtering by type cuts it to one per type, and a typed id cuts it to one full stop.
- **Deactivated *people* are excluded from the index; other records are not.** Member profiles get index-time filtering, including the nice touch of excluding profiles that are merge pointers to another profile. That is where it matters most and it is also the only model that does it. Extending it to the rest is one method per model, and it is on the list.
- **Six of the nineteen entities are id-only in text search.** Classes, reservations, package instances, retail items, recurring packages and penalties are found by id or friendly id, not by name.
- **The ranking rules are unit-tested; relevance is not measured.** No evaluation set, no A/B test. What is proven is that the rules behave as described.

Results

One search box over nineteen entity types, with a `business_account_id` filter enforced on every query path, rebuilt from a thin 2024 first cut into its current shape across three stages in 2026 without ever changing the endpoint contract.

WHAT THE DESIGN BOUGHT	HOW
A typed id resolves to one index	Pasting `UP1234` from a support ticket is a single-index lookup, not a fourteen-index fan-out, because the query told the system what it was
A name search returns people	Commerce records index the buyer name and deliberately do not query it, which is the difference between a usable result list and a flooded one
A bulk import is one indexing job	Two layers of Redis deduplication turn any volume of writes in a ten-second window into a single scheduled batch
Deleted records leave the index	And are never resurrected by a stale pending write, because of two lines in the right order
Live results and history cannot drift	One payload builder serves both, so there is one place to change what a result looks like
A scoped search is the same code	The `types` parameter narrows the identical pipeline, behind a forty-one entry alias map so older callers keep working

Test coverage runs to roughly 2,225 lines across 12 files, concentrated on the search orchestration and ranking rules, the payload builder, the batch worker and its scheduler, and recent searches — plus a real-Elasticsearch integration spec covering dependent reindexing and deletion. Search latency percentiles and per-staff usage figures are the client's to publish rather than ours to estimate.

What we would do next

Match the permission model the rest of the API already has. Every list endpoint on this platform gates on a capability — clients need `read_client`, sales need `read_sales`. Search authorises on "is a staff

member" and filters by tenant, which is a coarser rule than the endpoints it searches across. The plumbing is half built already: the command passes the acting user into the payload builder, which currently ignores it. Threading permissions through so a result set reflects what the viewer is allowed to see is the first thing we would build, and the single most valuable change on this list.

Extend index-time filtering past people. Member profiles get this right and are the only model that does. Deactivated staff, archived landing pages, deactivated workflows and cancelled penalties are all still findable. The mechanism exists; it is one method per model.

Point the mention picker at this pipeline. The @-mention autocomplete in the in-product assistant runs its own SQL `LIKE` query. The product therefore has two different definitions of "search for a person" — one with prefix matching, field weighting and exact-match promotion, one without. Consolidating onto `types=User` would delete code and make the two agree.

Surface partial failure instead of swallowing it. Per-model errors are caught and turned into zero results for that model, so an Elasticsearch problem can present as "no matches" rather than "something is wrong." Results are already assembled per model, so reporting which ones failed is a response-shape change more than a rewrite.

Give the six id-only entities searchable names. Classes and reservations are the two staff most plausibly want to find by name, and they are id-only because they were batched for query efficiency. That trade is worth revisiting now the batching exists.

Fix the disaster-recovery path. The bulk reindex task lists fifteen of the nineteen models. A rebuild from scratch would quietly leave four indices empty, and nothing would report it.

Upgrade Elasticsearch. 6.8 is end of life, and the 250-second client timeout should come down to something that fails a request rather than holding a thread for four minutes.

If this sounds like your system

The shape is common in any product with an admin console: a search box that returns too much of the wrong thing, a reindex that falls over during imports, and a "recent searches" list nobody uses because it remembers keystrokes.

The transferable decisions are these. **Read the query before you run it** — most applications have typed identifiers, and a prefix that identifies a record type turns a broad fan-out into a single lookup for exactly the queries where precision matters most. **Turn fuzzy matching off in a typeahead**, because the user is mid-word and can correct themselves, and the tighter candidate set lets the rest of your ranking stay small. **Decide which documents may match on a person's name**, and accept that indexing a field and querying it are separate decisions. **Put deduplication in front of your indexing, not concurrency limits behind it** — a Redis set plus a `SET NX EX` lock is a general answer to "my `after_commit` hook enqueues too many jobs," and it has nothing to do with search. And **key history on records rather than strings**, because people remember who they were working on, not what they typed.

Search is where the orders and transactions the revenue ledger reports on become findable by the people who answer the phone about them. If you have a search box that everyone in your company has

quietly stopped trusting, or an indexing pipeline that turns every bulk import into an afternoon, tell us about it — it is a problem with a known shape.

Tech stack

BACKEND

Ruby 3.3.9 · Rails 7.1 (API)

SEARCH

Elasticsearch 6.8 · Searchkick 4.6 · word_start prefix matching · Per-model field boosts

DATA

PostgreSQL 16 · Redis

ASYNC

Sidekiq · SET NX EX scheduling lock · SPOP batch draining

FRONTEND

Next.js admin console · TypeScript

TESTING

RSpec · Real-Elasticsearch integration spec

Frequently asked questions

Why classify the query instead of just searching everything?

Because the user has usually already told you what they want and it is cheap to listen. A friendly id like `UP1234` has an unambiguous type prefix, so it resolves to one index instead of fourteen and skips relevance ranking entirely. An email address means a person. Classification is a handful of regexes that turn a broad search into a narrow one for the queries where precision matters most.

Why disable fuzzy matching? Doesn't that hurt?

Not in a typeahead. Fuzzy matching exists to rescue queries the user cannot fix, and a user mid-word can fix their own. Edit distance 1 on a three-character token matches a huge amount of noise — the code's own example is that `mai` starts pulling in transactions whose notes contain "AI". Turning it off keeps the candidate set tight, which is what lets the ranking stay as simple as it is.

Is the "custom reranker" machine learning?

No, and it is worth being precise. It pins records whose email, phone or name exactly matches what was typed to the top of the list; everything below that is Elasticsearch's own TF/IDF ordering. No learned weights, no recency or popularity signals, no rescoring in the search engine. It is one signal Elasticsearch cannot produce on its own — "this is exactly what they asked for" — and the code says so in a comment.

How fresh are results after someone edits a record?

Roughly ten to twenty seconds. A save pushes an id into a Redis set and returns; a batch worker drains the sets every ten seconds. The delay is deliberate: it is what lets fifty updates to one record collapse into one reindex, and a bulk import of ten thousand records into a single scheduled job.

What stops one studio seeing another's data?

Every query on both search paths carries a `business_account_id` filter, and that field is on every indexed document. It is enforced at the two places that talk to Elasticsearch rather than in one shared wrapper, which is worth knowing if you add a third.

Can it search everything?

Nineteen entity types, covering people, classes, commerce, money and the platform's own configuration objects like notification templates, landing pages, workflows and widgets. Six of those are searchable by id and friendly id rather than by name, because they share a batched query for efficiency — so a class is found by its id today, not its title. That one is on the roadmap.

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`