One Derivation, Every Artifact

A customer completes an intake questionnaire once. The price, the delivery estimate, the justification for each line, the service agreement, the payment record, and the project's onboarding plan are all derived from that one set of answers.

Language models take part at several points in that sequence. None of them can set a rate. This document describes how that constraint is implemented, where it holds structurally, what the design costs to maintain, and the one path where the guarantee is procedural rather than structural.

99
KEYS IN THE PRICED VOCABULARY

27
RULES THAT RECORD THEIR OWN JUSTIFICATION

0
RATES A MODEL CAN SET

7
STAGES, INTAKE TO PROVISIONED PROJECT

Platform — Laravel · Inertia · React · MySQL · Reverb · **Resolution** — a deterministic pricing layer with no network calls and no model · **Narration** — gpt-4o, filtered before it can act · **Sale** — Stripe Payment Element, tablet point of sale, hashed sale evidence · **Delivery** — provisioning gated by the answers that priced it

One record, seven stages

The quote is not the deliverable. It is the first write in a sequence that ends with a provisioned project, and every stage after it reads the record that first write produced rather than composing its own version of the same facts.

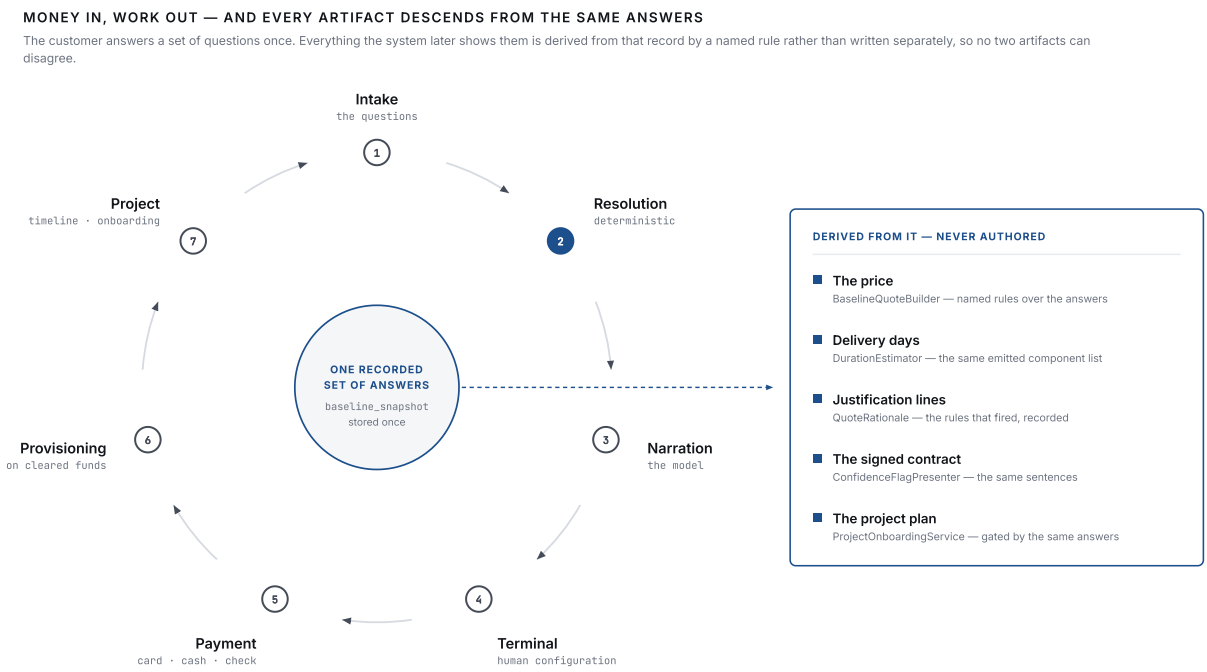

FIGURE 1 The seven stages, and the artifacts derived from the intake record. Models participate at stages three and four; neither stage owns a rate.

Five customer-visible artifacts descend from one set of `QuoteIntakeStep` rows. Each is produced by a named rule reading those rows, not by a separate authoring step that happens to agree with them.

`price`	**BaselineQuoteBuilder** over the recorded answers, persisted as `baseline_snapshot`.
`delivery estimate`	**DurationEstimator** over the same emitted modifier list that priced the work.
`justification lines`	**QuoteRationale** flags written by the rules that fired during pricing.
`service agreement`	**agreement_snapshot**, frozen at the moment payment clears.
`onboarding plan`	**ProjectOnboardingService** gates re-reading the same intake answers.

The reason to build it this way is narrow. A tool that writes its schedule in a second place will eventually quote one scope and schedule another, because nothing forces the two representations to stay in agreement. Deriving both from the same component list removes the second representation instead of adding a check that compares them. The same argument applies to the justification text, the contract, and the onboarding plan.

What follows is how each stage enforces that, and what the arrangement costs.

02 PRECEDENCE

Three sources, ranked, enforced downward

Three things can supply a value: what the customer explicitly selected, what the deterministic engine computed, and what the model drafted. They are ranked, and the ranking is applied as a single pass in one direction.

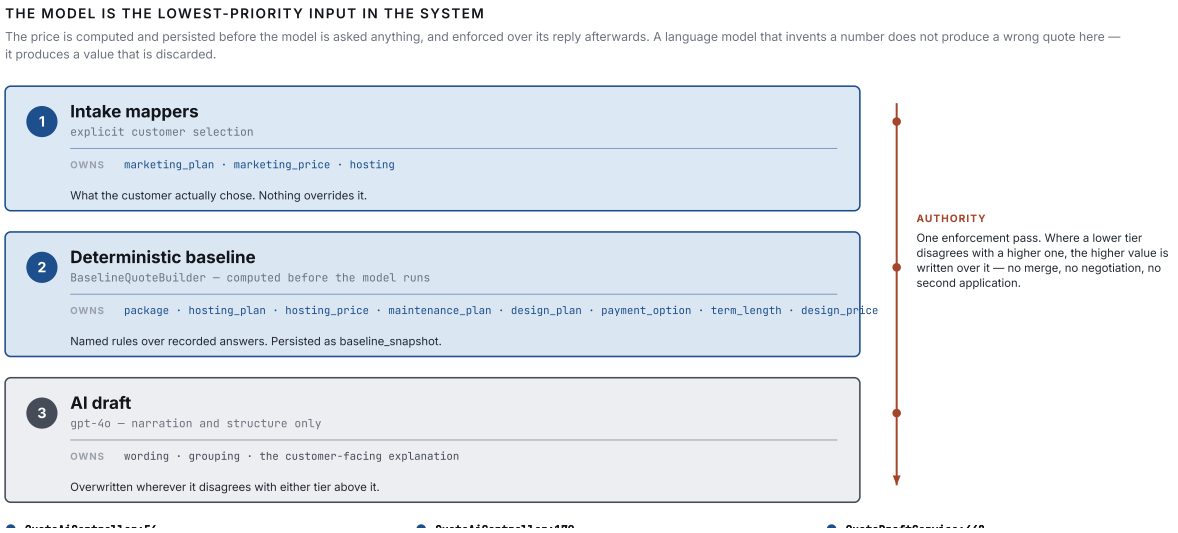

FIGURE 2 The three tiers and the fields each owns. Enforcement runs once, downward: explicit selection over computed baseline, computed baseline over model draft.

The ordering is visible in three call sites. The baseline is computed and persisted at `QuoteAiController:56`, before any request leaves the process. The model is not called until `:170`. Enforcement runs at `QuoteDraftService:448`, after the reply is parsed.

APP/SERVICES/QUOTEDRAFTSERVICE.PHP

```
/**
 * BASELINE ENFORCEMENT (ONE PASS, NO DUPLICATES)
 * Precedence:
 *  1) Intake mappers win (marketing + hosting)
 *  2) Baseline wins vs AI for remaining core selections/terms
 */
$baselineKeys = [
    'package',
    'hosting_plan',
    'hosting_price',      // plan + block volumes (hyperscaler mode)
    'maintenance_plan',
    'design_plan',
    // marketing_plan intentionally handled below
    'payment_option',
    'term_length',
    'design_price',
];
```

The single pass matters. Applying a precedence rule twice is how a percentage discount lands on an already-discounted line, which is the defect that produced the capitals in that comment.

The vocabulary is closed

Every key that can carry money is checked against configuration before it resolves: 12 application packages, 29 hosting plans, 5 maintenance tiers, 5 design plans, 8 marketing plans, 3 payment options, and 37 modifiers. Ninety-nine keys in total. A key outside that set does not price at zero and does not silently drop.

APP/SERVICES/QUOTE/PRICING/QUOTEKEYREGISTRY.PHP

```
public static function assertApplicationKey(string $key): void
{
    if (!array_key_exists($key, self::applications())) {
        throw new InvalidArgumentException("Invalid application key: {$key}");
    }
}
```

Intake itself is messier than the vocabulary. Customers enter `$6,000` where an integer is expected, answer a differently worded question, or describe the business in prose. That input is normalised by a parser whose docblock states its constraints as a contract:

APP/SERVICES/QUOTE/PRICING/HEURISTICNORMALIZER.PHP

```
/**
 * Deterministic, local-only parser to convert free-text / messy inputs into
 * structured "answers" your pricing layer already understands.
 *
 * NO AI CALLS. NO NETWORK. NO SIDE EFFECTS.
 */
```

The model proposes; the server decides

The ranking is easiest to describe on the drafting path, but it is tested hardest on the negotiation path, where the model is asked to act rather than to explain. When a customer pushes back on price, the model returns a structured change set: modifiers to add, modifiers to remove, plans to switch. `QuoteScenarioService` filters that set before any of it is applied.

APP/SERVICES/QUOTESCENARIOSERVICE.PHP

```php
// negotiation — only by changing the underlying requirement. The
// prompt tells the model not to offer it; this makes it impossible.
$blockedRemovals = [];
if (!empty($changes['remove_modifiers'])) {
    $blockedRemovals = array_values(array_filter(
        $changes['remove_modifiers'],
        fn ($k) => QuoteNegotiationService::isDerivedRequirementKey((string) $k)
    ));
```

Derived-requirement keys — anything prefixed `hipaa_`, `unit_`, `tenant_`, `phone_line_`, `phone_routing_`, or `large_dataset_`, plus six named fees — cannot be removed by negotiation at all. They are present because an intake answer requires them. Removing the charge without removing the requirement would quote a platform the customer has already stated they need.

A second filter handles capability overlap. Configuration under `quote.modifier_supersedes` names which services include which others. Adding a broader service consolidates the narrower one it covers rather than stacking on top of it; adding a narrower service that is already covered is dropped, and the covering service is named in the log. Only after both filters does the change set reach `calculatePricing`, which resolves every rate from configuration or from the persisted quote columns the baseline pass owns.

> **WHERE THE GUARANTEE IS PROCEDURAL, NOT STRUCTURAL**
>
> Baseline enforcement is guarded on a baseline that built successfully. If `BaselineQuoteBuilder` throws — an invalid configuration key, a malformed answer — both call sites log a warning and continue, and the enforcement block is skipped. The model's draft then survives unreviewed. Making this structural means failing the request instead, so the customer sees an error rather than an unchecked draft. That trade has not been made yet.

03 EVIDENCE

Answers the customer cannot be sure of

Intake asks people to quantify things they have never measured: catalogue size, monthly traffic, stored data volume. Trusting those answers produces an undersized platform. Ignoring them produces a quote the customer does not recognise.

Storage resolves against two independent floors. The declared floor is the band the customer selected. The estimated floor is computed from what the intake already proves: product count and images per product, migrated content volume, monthly traffic, subscriptions, customer accounts, the data profile follow-up, and unit count. The larger of the two wins, and a configured buffer is applied on top, because sizing capacity to exactly the volume of the data is not a capacity plan.

APP/SERVICES/QUOTE/PRICING/STORAGEESTIMATOR.PHP

```
// Layered resolution: the computed evidence and the declared band are
// both inputs; the larger wins. On top of the winner sits the
// free-space buffer — performance needs slack, so the requirement is
// never sized to 100% of the data.
$resolvedGb = max($declaredGb, $estimatedGb);
$bufferPct  = max(0.0, (float) ($cfg['free_space_buffer_pct'] ?? 0));
$bufferedGb = (int) ceil($resolvedGb * (1 + $bufferPct));
```

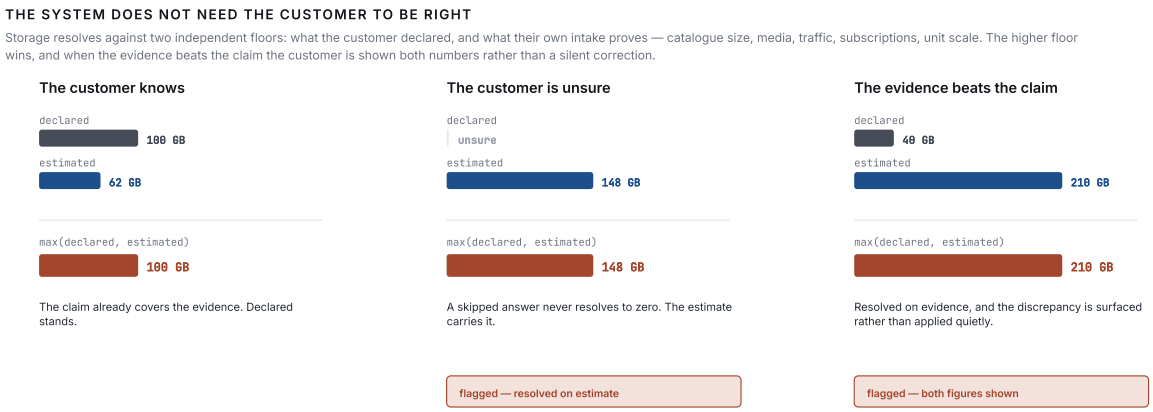

FIGURE 3 The declared band against the computed evidence across three intake outcomes. The resolution rule is the same in all three.

Three outcomes follow. When the declared band covers the evidence, the declaration stands. When the answer is "Unsure" or missing, the estimate carries the requirement and is flagged as an estimate to confirm, so a skipped question never resolves to zero. When the evidence exceeds the declaration, the evidence wins and the discrepancy is recorded with both figures.

The third case is the commercially significant one. The system raises the requirement and shows the customer both numbers rather than billing the difference quietly, which makes the follow-up conversation about the catalogue they described rather than about a line they did not expect.

When the customer and the evidence disagree

Hosting class resolves the same way, from usage drivers rather than from a small, medium, large picker. Strict per-unit isolation raises the class to dedicated; so does an organisation of sixteen or more units. Both are recorded as flags naming the answers that caused them.

The interesting case is an explicit collision: the customer has chosen VPS, and their other answers imply dedicated. The preference wins the class, the capacity floor rises within it, and the disagreement is written down with both verbatim answers.

APP/SERVICES/QUOTE/PRICING/HOSTINGPLANMAPPER.PHP

```
// Explicit-vs-explicit collision: the hosting preference wins the
// class, capacity floors rise within it, and the collision is
// flagged with both verbatim answers — the justification record.
if ($explicit === 'vps' && $strictIsolation) {
    $recommendedClass = 'dedicated';
    $r?->flag(
        'flag.hosting.collision',
        ['hosting_preference' => ..., 'unit_data_isolation' => ...],
        'Explicit VPS preference AND strict per-unit isolation both selected',
        'VPS preference kept; VPS capacity floor raised; dedicated recommended',
        'explicit_preference_wins_class_with_recommendation'
    );
}
```

The customer reads one sentence generated from that flag: *"You preferred VPS hosting — we kept your choice, raised its capacity floor for your scale, and noted dedicated as the recommended upgrade path."* This is the precedence rule from the previous section doing something visible. The explicit selection outranks the engine, and the engine records its dissent rather than discarding it.

04 JUSTIFICATION

What each rule leaves behind

The pass that computes the price records why it computed it. Each rule that fires writes the verbatim answers it read, the requirement it derived, the priced outcome, and its own name. That record persists with the quote.

APP/SERVICES/QUOTE/PRICING/QUOTERATIONALE.PHP

```
/**
 * Record a FLAG: the exact relationship between explicit user inputs and
 * a derived service requirement. Flags justify the quote line-by-line,
 * persist with the baseline snapshot, and protect against misalignment
 * claims — every priced (or deliberately suppressed) outcome names the
 * verbatim answers and the deterministic rule that produced it.
 *
 * @param array  $inputs      question id => verbatim selected answer(s)
 * @param string $requirement derived service requirement (human-readable)
 * @param string $resolution  priced outcome (modifier key/qty, package, or suppression)
 * @param string $rule        name of the deterministic rule that fired
 */
```

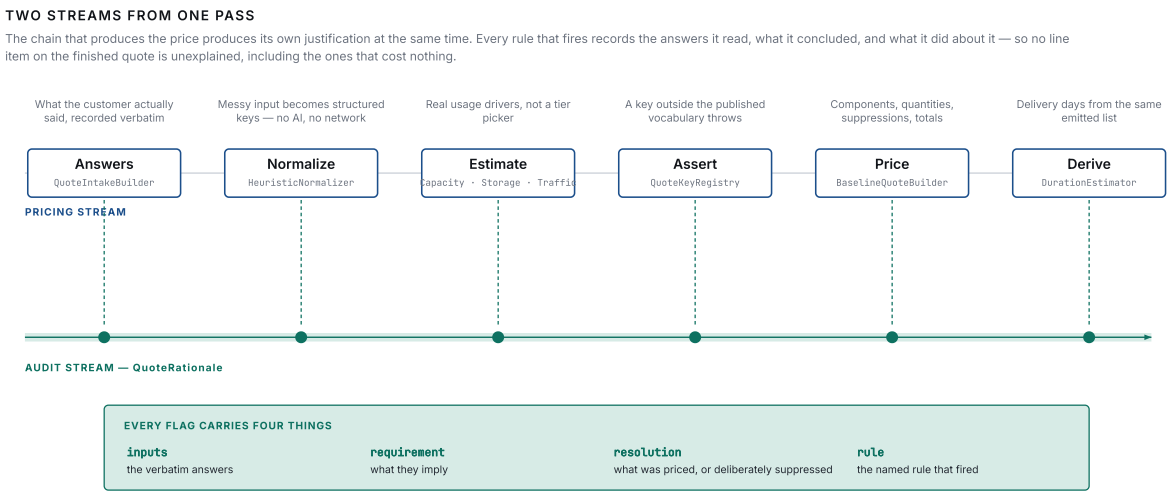

FIGURE 4 Price and justification are produced by the same pass. Each rule writes its inputs, its conclusion, and its name at the point it fires.

Twenty-seven flag codes are defined across the pricing services. Two of them are internal to delivery estimation and never shown. The remainder persist with the baseline snapshot and drive the customer-facing text.

The same sentence on the screen and in the contract

Flags become second-person lines through a presenter that matches on the flag code and returns wording. It performs no arithmetic and makes no decisions, and it is the only place that wording exists, so the review page and the signed PDF cannot diverge. An unrecognised code falls back to the recorded requirement text, so a rule added later renders as a plain sentence rather than as silence.

APP/SERVICES/QUOTE/PRICING/CONFIDENCEFLAGPRESENTER.PHP

```
/**
 * Deterministic verbal layer: each persisted flag (inputs → requirement →
 * resolution) becomes one second-person reassurance line — "you told us X,
 * we built Y." Suppression flags become not-double-charged trust lines.
 * No AI, no math, no judgment — wording only, sourced here once so the
 * review page and the signed PDF speak identical sentences.
 */
```

Suppression is a priced outcome

Both HIPAA packages include database encryption at rest, and HIPAA Complete includes audit logging. When a customer selects a package and also selects a capability the package covers, the resolver records the non-charge as an outcome — `db_encryption_fee NOT charged — included in HIPAA package` — with the same four fields as any priced flag. The presenter then renders it: *"Database encryption is already included in your HIPAA package — you were not charged separately for it."*

Recording suppression as an outcome rather than as an absence is what makes it renderable. A rule that simply declines to append a modifier leaves nothing for a later pass to find.

One derivation, two outputs

Delivery time is derived from the identical emitted modifier list that priced the work, with suppression and quantities already applied, plus the package and design plan. A quote and its schedule therefore describe the same amount of work by construction.

APP/SERVICES/QUOTE/PRICING/DURATIONESTIMATOR.PHP

```
/**
 * Derives working-day components from the SAME emitted modifier list that
 * priced the work (suppression and quantities already applied) plus the
 * package and design plan — one derivation, two outputs (dollars and days).
 *
 * The workload term reads the count of projects currently in build state;
 * with 0–1 active builds it is ×1.0, so during scarcity the estimate is pure
 * complexity and the factor self-activates only when builds overlap.
 */
```

The workload term reads live state, and that has a consequence worth stating: the same intake can produce different dates in different weeks. The estimate is therefore explainable but not reproducible from the answers alone. The active build count is written into the flag for exactly that reason, so a date quoted three months ago can still be accounted for. Any failure in that lookup returns ×1.0, which makes the fallback a pure complexity estimate rather than an error.

05 THE TERMINAL

Discretion with a defined shape

Automation that cannot be overridden does not survive contact with selling. The operator surface is a tablet point of sale: it loads a resolved quote, allows discretion within a defined shape, and closes the sale in person.

Discounts apply per component — application, design, hosting, maintenance, and each modifier individually — as a percentage, a flat amount, or both, floored at zero. Every discount input is persisted next to the line it modified, together with the original amount, the discounted amount, and the difference. An operator can move a price. The total remains reconstructible from what they entered, which is what makes the discount reviewable later rather than merely visible now.

Evidence is written before the card is charged

Acceptance of terms, confirmation of ownership, the signature image, device geolocation with its reported accuracy, IP, and user agent are written as a `SaleEvidence` row inside the same transaction that creates the payment intent. The row's id travels in the intent's own metadata, so the two records reference each other from both sides.

APP/SERVICES/LIVEQUOTESALESERVICE.PHP — CREATEPAYMENTINTENT

```php
return DB::transaction(function () use ($liveQuote, $sale) {
    $evidence = $this->writePendingEvidence($liveQuote, $sale);

    $intent = $stripe->paymentIntents->create([
        'amount'   => $amountCents,
        'metadata' => [
            'live_quote_id' => (string) $liveQuote->id,
            'evidence_id'   => (string) $evidence->id,
            'channel'       => 'admin_tablet_pos',
        ],
    ]);

    // Note: LiveQuote.status is NOT changed here. Stays 'generated' until
    // payment actually clears. Card decline = zero state corruption.
```

A declined card in a customer's kitchen leaves the evidence in place, which is correct: the customer did accept the terms, at that place and time. It creates no quote, no project, and no status change.

Each evidence row carries a SHA-256 hash over its own fields, recomputed and compared by `hashMatches()`. This is a checksum rather than a signature. It detects a row altered after capture; it does not defend against an actor who can rewrite the row and the hash together. The external anchor for that is Stripe's own immutable record, reachable through the evidence id in the payment metadata, which is why the linkage is written in both directions.

06 PROVISIONING

Cleared funds create the project

The webhook that confirms payment is the only thing that creates a project. Before it writes anything, it reconciles what Stripe reports against what the checkout recorded.

APP/HTTP/CONTROLLERS/API/STRIPEWEBHOOKCONTROLLER.PHP

```php
if (
    $stripeAmountCents !== (int) $checkout->amount_total_cents ||
    $stripeCurrency    !== (string) $checkout->currency
) {
    Log::error('[BILLING_MISMATCH] Stripe != QuoteCheckout expected', [...]);
    throw new \RuntimeException('Billing mismatch: refusing to finalize.');
}
```

Throwing here is deliberate. Stripe treats the delivery as failed and retries, which means a mismatch produces a retry and a logged error rather than a project provisioned against a total nobody agreed to.

`constructEvent`	**Authenticated.** An unsigned or mis-signed payload gets a 400 and never reaches a handler.
`amount reconciliation`	**Verified.** Stripe's total and currency must equal the recorded checkout.
`lockForUpdate`	**Serialised.** Concurrent deliveries of the same event contend on the row rather than racing.
`firstOrCreate(quote_id)`	**Idempotent.** One project per quote, regardless of how many times the event arrives.
`write-once merge`	**Append-only.** Agreement snapshot keys are filled when absent and never overwritten.

Two snapshots, deliberately separate

`agreement_snapshot` freezes what was sold: package, plans, the pricing totals, the baseline snapshot, the operator's adjustments, and the quote's `row_version`. `current_billable` carries what is billed now and is expected to change as the project does. Keeping them apart means a later upgrade changes the bill without rewriting the record of what the customer signed, which is the distinction that matters in a dispute.

The plan is gated by the answers that priced it

APP/SERVICES/PROJECTONBOARDINGSERVICE.PHP

```
// === ORGANIZATION & COMPLIANCE FOUNDATION (Day-1) ===
// Gated by the same explicit intake answers (or snapshot modifier keys
// for hand-built quotes) that priced the corresponding services.
```

Each gate fires on the exact intake answer that priced the corresponding service, or on the equivalent modifier key in the agreement snapshot when the quote was hand-built at the terminal and has no intake rows. A customer priced for per-unit data isolation is asked for a unit roster and receives isolation verification steps in the build phase. A customer who was not priced for it is not asked. The plan cannot request something nobody paid for, or omit something somebody did, because it reads the record that set the price rather than a checklist maintained alongside it.

Underneath sits ordinary multi-tenant scaffolding: organisations, businesses, teams, invitations, and six authorisation policies. A system that takes money and provisions work has to answer who is allowed to see a record before it does anything else with it.

07 INVARIANTS

The record reflects what happened

The same rule resolves four unrelated situations across two platforms, which is what makes it a position rather than a coincidence.

`Card declined`	**No status change.** The quote stays `generated`; the evidence of acceptance persists on its own.
`Webhook replayed`	**No second project.** The row lock and the unique quote key absorb the retry.
`Two callers, one slot`	**One booking.** The second caller receives a 409 and is offered what is genuinely open. *(companion system)*
`Failed tool call`	**No false confirmation.** A structurally valid result is synthesised that states the failure. *(companion system)*

In each case the convenient implementation writes the optimistic state and reconciles afterwards. Choosing otherwise costs a retry path and a reconciliation story in every subsystem that touches the boundary. What it buys is that an operator reading a record does not have to establish whether it is provisional before acting on it.

08 COST

What the architecture is expensive at

A description of what a design prevents is half a description. These are the places where this one is slow, fragile, or unfinished.

`Rule maintenance`	**Six edits per service.** Pricing occupies 3,937 lines across `app/Services/Quote/Pricing` over a 99-key configured vocabulary. Adding a billable service is a registry key, a resolver rule, a flag code, a presenter line, a duration weight, and an onboarding gate — in six files, in the right order. A pricing table would take one row.
`Combinatorial surface`	**Untested combinations exist.** Suppression, per-unit multiplicities, and the five-layer organisational escalator interact. The escalator fires at three or more active separation layers, and each layer carries its own foundation and provisioning keys. The reachable combinations outnumber the combinations that have been priced in production.
`Non-reproducible dates`	**Live state in a quoted number.** The workload multiplier reads current build load, so an identical intake yields different dates in different weeks. The count is recorded, which makes the estimate explainable after the fact but not derivable from the answers alone.
`A procedural seam`	**One guarantee is not structural.** Baseline enforcement is skipped when the baseline fails to build, and the failure is a logged warning rather than a rejected request. See §02.
`One reviewer`	**No second engineer.** Every judgment recorded here was made once, by one person, and tested by use rather than by review. The system has been corrected where selling exposed a misrepresentation; it has not been audited by anyone who did not write it.

None of these are arguments against the approach. They are the reason it is worth stating what the approach actually bought: the artifacts a customer sees cannot disagree with one another, because there is no second place where a disagreeing version could have been written.

09 CODA

Where the crawler came from

The headless-browser site analyser described in the companion case study — the acquisition pipeline that gives an autonomous receptionist its starting knowledge — was built here first, for a different reason.

It was a sales instrument. Read a prospect's website before quoting it, so the first conversation starts from evidence rather than from what someone remembered to mention. Its output still feeds this system's quote context.

Carried into the receptionist platform, the same component supplies a business's published site as its receptionist's knowledge on day one. The interface did not need to change, because it had been written to answer a question about a website rather than to serve a quoting tool.

The reuse was possible because the boundary held. That is usually the only evidence available that it was drawn in the right place.

Designed, built, and operated by one engineer.

Laravel · Inertia · React · MySQL · Reverb · Stripe · Playwright. A deterministic pricing layer of 3,937 lines over a 99-key vocabulary, 27 named justification rules, a tablet point of sale with hashed sale evidence, and project provisioning gated by the answers that set the price.

Every decision recorded here was reached the same way: build it, sell with it, find where it misrepresents something, and change the mechanism rather than restate the intention.

Code excerpts are from the running system. Comments are verbatim; parameter lists and log payloads are elided for width. Line references are to the current tree. Figures are drawn with D3 and printed as vector.