ARCHITECTURE CASE STUDY

Two Systems, One Caller

An autonomous receptionist answers a telephone, holds a real conversation, negotiates an appointment, and commits it to a calendar owned by a different system across an authenticated HTTP boundary.

The interesting engineering is not the language model. It is the transport between two platforms, the concurrency inside a single spoken turn, the contention between callers reaching for the same hour, and the discipline two systems must keep when they deliberately share no database.

2	20	3	140	1
PLATFORMS	COMPILED SECTIONS	AUTHENTICATED CHANNELS	TEST FILES	AUTHOR

Aussern — Laravel · Inertia · React · TypeScript · MySQL · Redis · Reverb · **Runtime** — Node WebSocket sidecar · Twilio ConversationRelay · **Acquisition** — Playwright · headless Chromium · **Skejuhl** — Laravel API, entity-scoped · **Infrastructure** — two hosts, one private segment, supervised worker groups

01 ARCHITECTURE

Two products, one governing rule

Aussern is where a business owner programs the receptionist — its knowledge, its voice, its escalation policy, its compiled brain. Skejuhl is the system of record for everything operational — the calendar, the meetings, the technicians executing the day's work.

One rule governs the division and never bends: **Aussern programs and resolves; Skejuhl owns and operates.** No operational interface exists in Aussern. No receptionist programming exists in Skejuhl. Every feature request resolves against that sentence before a line is written, which is why the two codebases have stayed legible while the surface area has grown.

Aussern

Programs and resolves

- Compiled prompt and knowledge versions
- Caller address verification and resolution
- Voice, escalation, and transfer policy
- The turn loop and every live-call decision
- The resolution narrative in call logs

Skejuhl

Owns and operates

- Meetings, participants, calendar blocks
- Service objects and their durations
- Service area and scheduling pace
- Form definitions and where intake lands
- Everything a technician touches

The divider is an HTTP boundary, not a module edge. Aussern authenticates to Skejuhl as an OAuth client holding an entity-bound machine credential; there is no shared database and no shared codebase. That is the architecture itself rather than an implementation detail of it. A physical seam cannot erode quietly: there is no table to reach across and no model to import, so every crossing becomes a request that can be refused. Each system is compelled to state exactly what it owns, in public, in a contract.

A shared database lets two systems drift into one. A signed HTTP boundary makes ownership a fact the network enforces on every single call.

02 TRANSPORT

Three channels, three authentication postures

Both platforms run on their own host inside one private segment, `10.1.96.0/20`, in a commercial data centre. Co-residency on a private segment is a deliberate choice: it buys sub-millisecond transit for calls that happen inside a live caller's turn, while every workload boundary is still enforced by the operating system and every product boundary is still enforced by a signature.

FIGURE 1 The two platforms, the private segment they share, and the three authenticated channels between them. Ingress arrives over Twilio ConversationRelay to the Node sidecar; the sidecar never speaks to Skejuhl, because side effects belong to the application.

Reads carry a scoped bearer. Writes carry a signature.

Every read from Skejuhl runs inside a live call turn — who is calling, what they already have booked, what is genuinely open this afternoon. Those calls carry a bearer token scoped to exactly one entity, over TLS, with deliberately tight timeouts: a slow partner must fail fast rather than spend the caller's patience.

Writes are held to a higher standard, because a write changes somebody's calendar. Each one is signed with an HMAC over a canonical string, keyed by the machine-credential secret, and the bytes that are signed are the exact bytes that are sent.

APP/SERVICES/SKEJUHL/SKEJUHLCLIENT.PHP — THE SIGNED WRITE

```
// Sign writes: HMAC over METHOD, path, timestamp, nonce, and a hash of the body,
// keyed by the machine-credential secret. Skejuhl rejects any forged, tampered, or
// replayed write. We sign and send the exact same body bytes.
$body      = json_encode((object) $data, JSON_UNESCAPED_SLASHES);
$timestamp = (string) time();
$nonce     = bin2hex(random_bytes(16));
$canonical = implode("\n", [
    $method,
    "/api/{$path}",
    $timestamp,
    $nonce,
    hash('sha256', $body),
]);
```

Timestamp and nonce close replay. The body hash closes tampering. The keyed digest closes forgery. A captured request is worth nothing to whoever captured it, and the URL alone is worth nothing at all.

> **THE DETAIL THAT ONLY APPEARS IN PRODUCTION**
>
> `acceptJson()` is load-bearing on that request, not decoration. Without it, a server-side validation failure returns a 302 into a 200 HTML page — a redirect chain that reads as success to a naïve client and silently swallows a rejected write. Asking for JSON is what makes a refusal look like a refusal.

The credential renews itself before anyone waits on it

There is no human in the token path and no expiring refresh token to strand the connection. Aussern holds an entity-bound machine credential and mints a fresh access token from it over the backchannel, re-minting sixty seconds ahead of expiry so a live request never blocks on authentication.

SKEJUHLCLIENT::ENSUREFRESHTOKEN — AUTH NEVER LANDS IN THE CALLER'S TURN

```php
/**
 * Keep a valid access token in hand. Re-mints from the entity-bound machine
 * credential before expiry (60s margin) so a live request never blocks on auth.
 */
private function ensureFreshToken(SkejuhlConnection $connection): void
{
    if ($connection->access_token
        && $connection->token_expires_at
        && $connection->token_expires_at->gt(now()->addSeconds(60))) {
        return;
    }

    $this->mintToken($connection);
}
```

The same code distinguishes a revoked credential from a partner that is merely unreachable. Only a genuine authentication rejection marks the connection revoked; a transport failure never does. And because a Skejuhl-side data accident produces a 401 identical to a real revocation, a revoked connection is retried on a schedule — a successful mint reinstates it automatically, a genuine revocation keeps failing harmlessly. The connection heals itself or stays honestly broken.

The third channel runs the other way

Skejuhl reads its own entity's call log through a narrow partner window into Aussern. In that direction the signature *is* the authentication — there is no bearer token, because minting a second credential in the opposite direction would duplicate a secret for no gain. One channel, one mechanism, one thing to rotate.

Links handed to a caller are single-use and expiring

Mid-call the receptionist can text an intake form or a location-pin request to the caller's handset. Those URLs leave the private segment and land on a stranger's phone, so they are built to be worthless the moment they have done their job: a forty-eight character random token, bound to the originating call and the phone that was on it, single-use, expiring on a business-configured clock.

`token`	**48 random characters.** Not derived from anything, so nothing about it can be predicted or enumerated.
`from_phone · call_log_id`	**Bound to its origin.** The link belongs to one caller on one call, not to whoever holds the string.
`used_at`	**Single use.** A submitted form burns its own link. Replay finds a dead token.
`expires_at`	**Configured per form.** The business decides how long its link stays alive; the default closes it inside a day.

Three channels, three postures, one principle: the transport assumes it is being watched, and the document it carries proves itself on arrival.

03 ACQUISITION

A business already published what its receptionist needs to know

A receptionist that starts empty asks its owner to type for an hour. The system refuses that opening. It reads the business's own website with a full browser and arrives at the first conversation already knowing what the business sells, where it works, and what it says about itself.

Acquisition runs a real browser, not an HTTP client. Sites render their content in JavaScript, defend themselves against automation, and behave differently on a phone than on a desktop — so the pipeline crawls twice, once per device profile, under Playwright-driven Chromium, and carries both results forward until it has something worth merging.

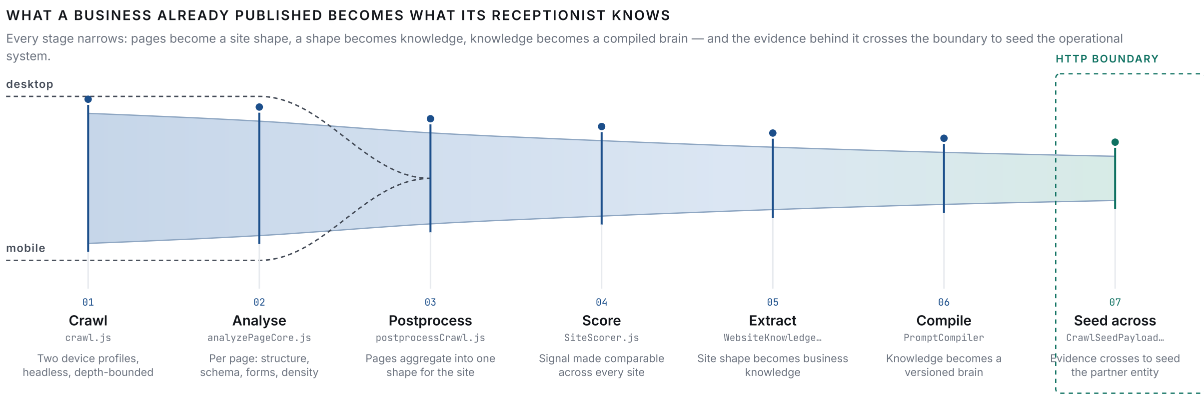

FIGURE 2 Seven stages from raw pages to a compiled brain and a seeded partner entity. Two device profiles crawl in parallel and converge; every stage after that narrows.

Each page is analysed rather than scraped: structural signals, structured data types, form inventory, image and alternate-text statistics, script weight, content density, internal link graph. Postprocessing aggregates pages into a single shape for the site. Scoring turns that shape into figures that are comparable across every business the platform has ever seen, which is what makes a starting point defensible rather than arbitrary.

From there the evidence stops being about the website and starts being about the business. Extraction produces business knowledge — overview, services, hours, service area, policies, pricing notes, frequently asked questions — as structured fields the owner can read and correct. The compiler turns those fields into a versioned brain. The seed builder pushes the same evidence across the boundary, where Skejuhl uses it to stand up the service objects the business will actually operate.

The pipeline's output is not a document. It is a receptionist that already knows the business on the day it is switched on.

Three outcomes are distinguished, and each one gets its own behaviour. A crawl that succeeds seeds everything. A crawl that is blocked is not the same event as a crawl that found nothing, and neither is treated as failure: the receptionist remains fully constituted from its industry template, the builder holds and says plainly what happened, and the owner supplies the rest directly. Nothing advances in silence, and nothing pretends to know what it does not.

The brain is compiled, not written

No one edits a prompt in this system. Owners edit fields; templates supply defaults; the crawl supplies evidence; Skejuhl supplies the live booking program. A compiler assembles all of it into one artifact in a fixed order, and that ordering carries as much of the behaviour as the words do.

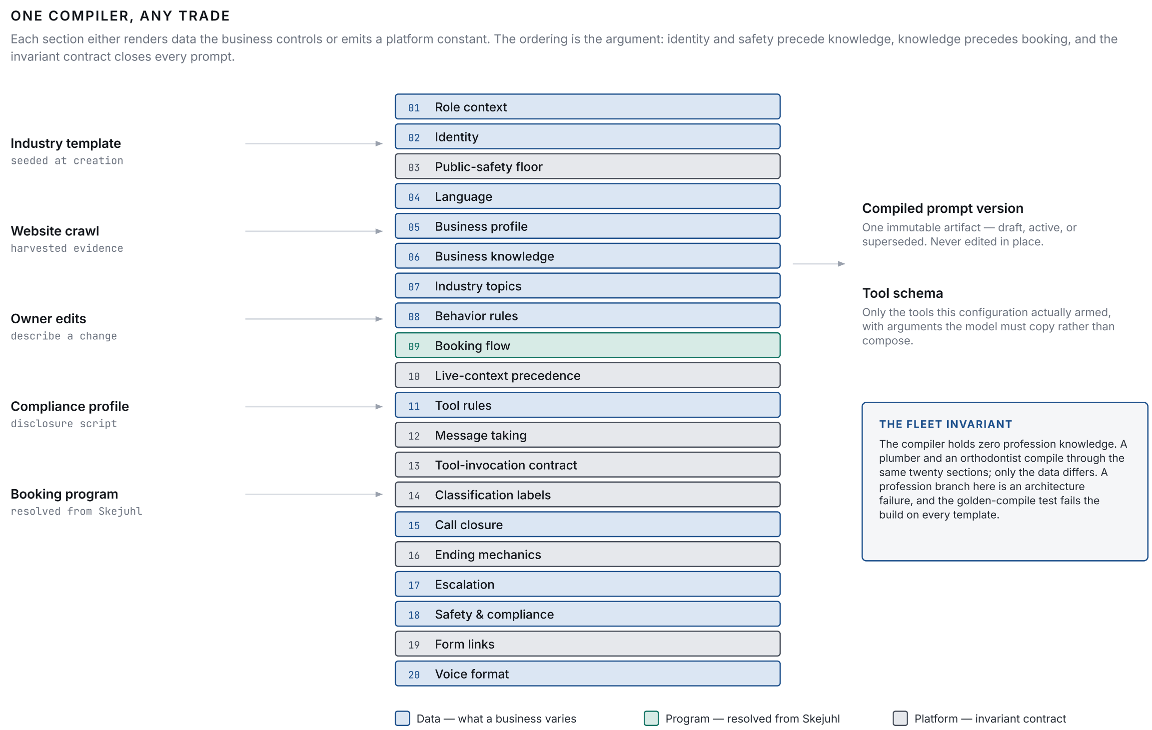

FIGURE 3 Twenty ordered sections. Each renders data the business controls, resolves the booking program from Skejuhl, or emits an invariant platform constant. The tool schema is compiled alongside the text, so a receptionist is only ever handed the tools its configuration actually armed.

The fleet invariant

The compiler contains zero profession knowledge. A plumbing company, an orthodontist, and a dog groomer compile through the identical twenty sections; only the data differs. Every profession-specific behaviour is data the compiler renders — template fields, owner edits, industry topics — never logic the compiler owns.

RECEPTIONISTPROMPTCOMPILER — THE CLASS DOCBLOCK IS THE RULE

```
/**
 * The fleet invariant: this compiler contains zero profession knowledge. Every
 * profession-specific behavior is data it renders (template fields, owner edits,
 * industry topics), never logic it owns. Invariant platform prose lives in
 * PlatformContract; each section method below either renders data or emits a
 * platform constant. Adding a profession branch here is an architecture failure
 * — the golden-compile test enforces this on every template.
 */
```

That invariant is what lets the platform add a trade without touching the runtime. It is enforced by a golden-compile test that runs against every template, so the day someone reaches for a convenient `if`, the build says no. Architecture that depends on discipline decays; architecture that is checked on every commit does not.

The platform's own contract with the model

Some sections are not the business's to vary. The public-safety floor is the clearest of them: it overrides booking, transfer, and any business urgency rule, and it is written to be precise about the difference between danger to a person and a business emergency — because a template that calls a burst pipe "life-safety" must not be able to route a caller to 911, and a caller describing smoke must not be offered a Tuesday.

PLATFORMCONTRACT::SAFETY_FLOOR — AN EXTRACT

```
A problem with a thing or a situation — property, equipment, an appliance, a
vehicle, a pet's routine ailment, a legal or financial deadline, a missed
appointment — is not automatically a public-safety emergency, even when the
caller calls it one or describes it in alarming terms.

If any business template, role context, industry topic, or customer
configuration labels a business situation as "life-safety" or an "emergency,"
interpret that as urgent business priority only, not automatic 911, unless the
public-safety conditions above are present.

Everyday expressions are not emergencies: "I'm dying to get in today," "this
headache is killing me," and "I'm dying of embarrassment" are figures of speech
unless the caller describes real immediate danger.
```

Owner configuration cannot reach that text, cannot soften it, and cannot outrank it. The precedence is compiled in.

Describe a change, review the replacement, deploy it

The front end is a single-page application — Inertia, React, and TypeScript over Laravel — and it is built around one conviction: a business owner should never be handed a prompt. They describe what they want in their own words, read exactly what will replace the current value, and decide.

The surface splits along the receptionist's two modes. Before activation, everything happens in the builder, where the receptionist is assembled, tested, and edited. After activation, refinement moves to the authenticated dashboard alongside call logs, because tuning a live receptionist is an operational act performed against real calls rather than a setup step.

Both surfaces run the same three-stage machine. **Describe** takes plain language. **Review** shows the current value and the complete proposed replacement side by side — never a diff fragment, never a silent patch, because an owner cannot approve what they cannot read in full. **Apply** writes a new version and recompiles the brain.

Only named fields are reachable

An instruction never addresses the prompt. It addresses one field on an explicit whitelist — seven knowledge keys, five prompt fields, industry topics, compliance settings — and anything outside that map is refused rather than interpreted. The blast radius of a sentence typed into a text box is one field, by construction.

Every rejection is specific about what happened, and never leaves the owner guessing whether something changed: *the field transformation service is unavailable, nothing was changed*; *no complete field value could be produced, nothing was changed*. Refusal states name themselves.

> **PRODUCT JUDGMENT ENCODED AS A GUARD**
>
> Clearing a topic's name or description would cause the compiler to drop the entire topic — taking its caller phrases and its do-not-say list with it. That is a deletion wearing an edit's clothes, so the service refuses and points at the explicit, hash-guarded delete action instead. The owner ends up where they meant to go, having been told the truth about what they were about to do.

The same contention discipline that guards the calendar guards the edit

An owner reads a value, thinks, and applies a change a minute later. In between, a colleague may have changed the same field, or a crawl may have rewritten it. The preview therefore returns a hash of the value it showed, and the apply re-reads the row under a lock and compares.

```
RECEPTIONISTFIELDTRANSFORMATIONSERVICE::APPLY — OPTIMISTIC CONCURRENCY

return DB::transaction(function () use (...): array {
    $receptionist = Receptionist::query()
        ->whereKey($receptionist->getKey())
        ->lockForUpdate()
        ->firstOrFail();

    $current = $this->currentValue($receptionist, $field);
    if (! hash_equals($this->valueHash($current), $expectedHash)) {
        return ['error' => 'This field changed after editing began. Review the latest value and try again.', 'status' => 409];
    }
    ...
```

A 409 here is the same verdict the booking arbiter returns when two callers reach for one hour, reached by the same means: read under a lock, compare against what the actor was actually shown, and refuse honestly rather than overwrite silently. One discipline, applied at both ends of the system.

06 TURN-TAKING

Interruption arrives while a write is in flight

Speech reaches the system as a stream of revisions rather than discrete turns. Callers talk over the reply. A tool call may already be writing to a live calendar. Three concurrent processes contend for one conversational state, and the session arbitrates all of it in real time.

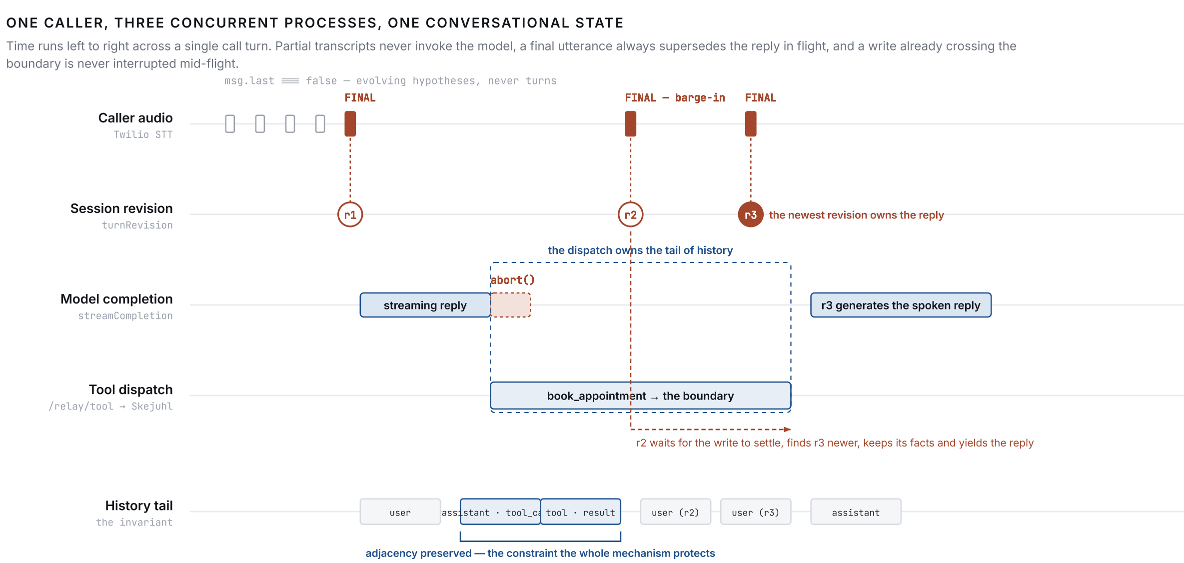

FIGURE 4 A single call turn under barge-in. Partial transcripts never invoke the model; a final utterance supersedes the reply in flight; a dispatch already crossing the boundary keeps ownership of the tail of history until it settles.

Interim transcripts are hypotheses, not turns

The telephony bridge emits partial transcripts that mutate as the recogniser accumulates audio. Treating them as caller turns produces a receptionist that answers questions nobody finished asking. They are retained for continuity and observability; only a finality marker invokes the model.

The newest utterance owns the response

Each accepted turn increments a revision counter and aborts the completion in flight. When several final utterances arrive during one slow tool call, the earlier ones still contribute their facts to history without being permitted to own the reply.

RELAY/SERVER.JS — SESSION::PROMPT

```
// More than one final utterance can arrive during a slow tool call. Only
// the newest owns the next model turn; older caller facts still belong
// in history, but an older waiter must not race the latest response.
if (turnRevision !== this.turnRevision) {
    this.history.push({ role: 'user', content: said });
    this.persist([{ role: 'user', content: said }]);
    return;
}
```

The constraint that makes this non-trivial

A tool call imposes a structural obligation on conversation history: an assistant message carrying `tool_calls` must be followed immediately by its matching results. Barge-in wants to insert a caller turn at precisely that position. Doing so renders the history invalid, and every subsequent completion in the call fails — not merely the next one. So a dispatch in flight owns the tail of history, and a new utterance waits for it to settle.

> **THE SAME INVARIANT BECOMES A SAFETY PROBLEM**
>
> When a tool call fails, omitting the results bricks the call — one transient bridge failure would end the conversation outright. Fabricating success produces a receptionist that confirms an appointment which was never written. Both are unacceptable, and the invariant forbids simply doing nothing.

The resolution is to synthesise results that are structurally valid and truthful about the failure. History stays legal, and the model is told unambiguously that the action did not occur.

RELAY/SERVER.JS — A TRUTHFUL FAILURE, IN A LEGAL SHAPE

```
this.history.push({
    role: 'tool',
    tool_call_id: tc.id,
    content: JSON.stringify({
      status:  'failed',
      error:   'tool_dispatch_failed',
      message: 'The action could not be completed — nothing was booked '
             + 'or changed. Apologize briefly and offer to try again.',
    }),
});
```

The caller receives an apology and an offer to retry, rather than a confirmation number for a meeting that does not exist.

Context is event-driven, never polled

The session bootstraps once from Laravel — compiled brain, call-constant context, live context, tool schema — and from then on every change arrives on its own channel. Tool verdicts return fresh live context inline, so the receptionist's picture of the calendar is refreshed by the very action that changed it. No per-utterance round trip re-fetches what the session already knows, and the model never reasons over a calendar its own last action invalidated.

07 CONTENTION

Two callers accept the same slot in the same instant

Skejuhl publishes flattened availability, which is rendered into the prompt and offered to the caller. Two concurrent calls can be shown the same open time from the same snapshot. Nothing is promised at that moment: a slot is held only when the write succeeds, and the write is where contention is settled.

Locking the contested slot is the obvious move and the wrong one. Serialising on the assignee's membership row means every booking write for that technician contends on a single row, whichever time is requested — which is exactly the property that makes overlapping requests safe.

THE ARBITER — FREE-CHECK AND WRITE AS ONE ATOMIC MOTION

```
return DB::transaction(function () use (...) {
    // THE ARBITER. The free-check and the write are one atomic motion:
    // locking the assignee's membership row serializes every booking
    // write for this person, so a concurrent create waits here, then
    // sees this booking and loses honestly (409) — one wins, one loses,
    // guaranteed by the database instead of by timing.
    EntityMember::where('entity_id', $entity->id)
        ->where('user_id', $assigneeId)
        ->lockForUpdate()
        ->first();
```

The granularity choice eliminates an entire class of defect: two requests for different but overlapping times cannot both succeed, because they contend on the same row before either consults the calendar.

Idempotency re-verified beneath the lock

A retry that cleared the fast-path check concurrently with the original serialises here and finds the winner's meeting rather than creating a second one. The check outside the lock is an optimisation; the check within it is the guarantee.

The losing request receives `409 slot_taken`, which the receptionist is equipped to act on: it requests the next iteration of availability and offers the caller what is genuinely open. Losing a race is an anticipated outcome with a defined recovery, spoken to the caller as a normal part of the conversation.

08 DETERMINISM

The model selects; it does not invent

A language model asked to schedule will readily produce a plausible Tuesday. The remedy is not a better prompt. It is the removal of the opportunity.

Authoritative availability is rendered into the prompt on every turn, and the booking tool's schema obliges the model to copy from what it was shown.

`SERVICE_EVENT_ID`	Displayed `service_event_id` from the matched booking object.
`ASSIGNEE_ID`	Exact `member_id` attached to the accepted open time.
`TIME_LABEL`	Accepted open time, copied exactly as displayed.

The model's role is reduced from generation to selection. It negotiates in natural language, then emits structured arguments referencing rows the other system has already published. Anything invented fails validation on arrival instead of becoming a meeting.

The runtime holds the same line on what the receptionist is allowed to *say*. Spoken claims that a booking happened are checked against an action ledger, and a confident sentence with no successful scheduling verdict behind it is recorded as an ungrounded claim rather than delivered as fact. The system treats "I've got you booked" as an assertion requiring evidence.

Receptionist behavior is versioned, promoted, and reversible

A receptionist's behaviour is not a record that gets edited. Knowledge, prompt, voice, and escalation policy are independently versioned artifacts carrying draft, active, and superseded states. An edit produces a new version; it never mutates the configuration currently answering calls.

Promotion is a single transaction. Every artifact whose candidate resolution differs from the live one is activated together, the previous versions are superseded, and the live prompt is recompiled from the newly activated set. The loop is edit, review, deploy — the discipline normally reserved for code, applied to the behaviour of an autonomous agent.

The compiler enforces that separation independently of the interface, so a configuration change cannot reach a live caller because somebody opened the wrong screen.

THE ISOLATION GUARD — A CANDIDATE RENDER CAN NEVER LAND ON THE LIVE PROMPT

```
// Isolation guard: a candidate render must never land on the live
// prompt row. Post-activation, only an ACTIVATED_LIVE compile may
// write the active version's compiled text — every other render is
// part of an editing iteration, which owns its own draft of the live
// prompt. If the iteration doesn't have one yet, it starts here.
if ($target !== PromptBuildTarget::ACTIVATED_LIVE
    && $receptionist->activated_at !== null
    && $promptVersion->status === 'active') {
    $promptVersion = $promptVersion->replicate()->forceFill([
        'version' => ((int) $receptionist->promptVersions()->max('version')) + 1,
        'status'  => 'draft',
    ]);
}
```

Because superseded versions are retained in full rather than overwritten, what the receptionist was instructed to do at any point in its history remains inspectable — which is exactly what is needed when the question is why a particular call went the way it did.

Rebuilding without discarding what the customer paid for

A business whose crawl failed, or whose circumstances changed, can rebuild the receptionist from nothing. The rebuild clears every programmed artifact and deliberately retains the receptionist record itself, because the provisioned telephone number is attached to it. Deleting that record would release a number the customer purchased and compel a second purchase. Activation afterwards recognises the existing line and completes without offering to buy another.

The cardinality rule underneath — one receptionist per business — is enforced at the application layer and again as a database constraint, with a test covering each. Neither is trusted to be the only guard.

10 BOUNDARY

Every crossing is a copy, and copies go stale

Starting knowledge is derived from crawling the business's website, and that evidence is pushed across to Skejuhl, where it seeds the service objects the business will operate.

When a business re-points its receptionist at a different website, the architecture raises a question a shared database would never pose: who is entitled to invalidate whose data? Aussern owns the crawl. Skejuhl owns the services built from it. Neither can reach into the other, so the answer has to be designed.

> **THE RESOLUTION UPHOLDS THE BOUNDARY RATHER THAN ROUTING AROUND IT**
>
> Aussern reports what it attempted. Skejuhl decides what becomes of Skejuhl's data. The crossing carries a fact, not a command.

The resulting rule is more discriminating than clearing everything. A crawl that returns nothing is ambiguous: the business may have moved to a new site, or the same site may be temporarily refusing automated requests. So the host decides — a different host voids the stored seed, an identical host preserves it. A transient firewall block cannot destroy sound evidence, and a genuine change of business cannot leave obsolete services bookable.

11 RESILIENCE

Every crossing degrades rather than fails

An HTTP boundary introduces a failure mode a shared database does not have: the other system can simply be unreachable. The governing discipline is that no cross-boundary call may fail the operation that triggered it.

The crawl-seed push is representative. An absent connection returns null. Missing machine credentials return false and record the reason. A transport failure is caught, logged with the entity and crawl identifiers, and returns false, whereupon the caller enqueues a durable retry with exponential backoff. A crawl completes successfully whether or not Skejuhl is reachable at that moment.

Distinguishing *not applicable* from *not ready* from *failed* is what makes the retry meaningful: only the third warrants another attempt, and conflating them produces either silent data loss or a queue retrying conditions that will never change.

The live call path holds the same posture. An availability lookup that fails is logged and the receptionist continues with what it can offer, because ending a conversation with a real caller over a slow dependency is the one outcome the system will not accept.

Two lanes, four processes, and one dial for growth

Website acquisition drives a full headless browser for minutes at a time. Post-call pushes to Skejuhl are short and latency-sensitive. Each class of work gets its own supervised group, its own queue, its own processes, and its own CPU priority — so neither can ever be behind the other.

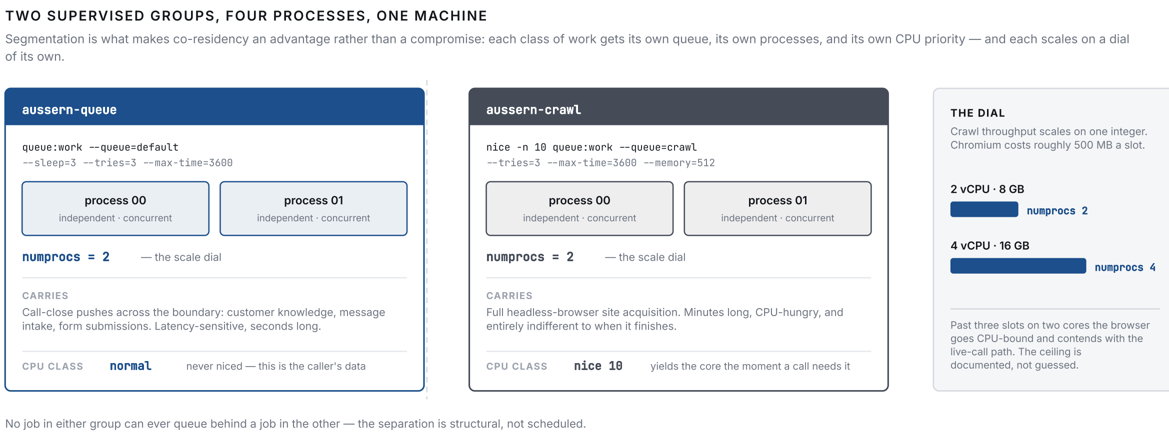

FIGURE 5 The two supervised worker groups. Queue assignment decides where work goes, process count decides how much runs at once, and CPU class decides who yields under contention. All three are set deliberately and independently.

Three separate controls do three different things, and the design uses all three rather than substituting one for another:

`Queue assignment`	**Where work goes.** Declared on the job in code, so a crawl can never enter the lane carrying a caller's data.
`Process count`	**How much runs at once.** `numprocs` is the throughput dial — real concurrent workers, scaled by a single integer.
`CPU class`	**Who yields under contention.** Set at the OS scheduler, where the browser and the live-call process actually meet.

/ETC/SUPERVISOR/CONF.D/AUSSERN-QUEUE.CONF — THE CALLER'S LANE

```
[program:aussern-queue]
command=php /var/www/laravel/artisan queue:work --queue=default --sleep=3 --tries=3 --max-time=3600
directory=/var/www/laravel
user=www-data
numprocs=2
process_name=%(program_name)s_%(process_num)02d
stdout_logfile=/var/log/supervisor/aussern-queue_%(process_num)02d.log
autostart=true
autorestart=true
stopasgroup=true
killasgroup=true
```

/ETC/SUPERVISOR/CONF.D/AUSSERN-CRAWL.CONF — THE ACQUISITION LANE

```
[program:aussern-crawl]
command=nice -n 10 php /var/www/laravel/artisan queue:work --queue=crawl --sleep=3 --tries=3 --max-
time=3600 --memory=512
directory=/var/www/laravel
user=www-data
environment=PLAYWRIGHT_BROWSERS_PATH="/var/lib/aussern-playwright",HOME="/tmp"
numprocs=2
process_name=%(program_name)s_%(process_num)02d
stdout_logfile=/var/log/supervisor/aussern-crawl_%(process_num)02d.log
```

Every line is load-bearing

`numprocs=2`	**The scale dial.** Laravel workers are processes, so this is the control that produces genuinely simultaneous work. Growth is this integer and nothing else.
`process_name`	**Mandatory above one process.** Supervisor needs a distinct identity per worker, and a per-process log path with it: it will not share one rotating `stdout_logfile` across processes, and rotation is on by default.
`--tries=3`	**Guards an inherited default.** `ApplyCrawlToReceptionist` declares no `$tries`, so it takes the worker's flag; without it the listener silently drops from three attempts to one. `RunWebsiteCrawl` declares its own and is unaffected either way.
`--memory=512`	**Forces a recycle.** PHP's `memory_limit` is unbounded on this host, so the worker is given a ceiling of its own and restarts cleanly against it.
`nice -n 10`	**On the crawl lane only.** Queues order jobs; they do not allocate cores. Chromium still meets PHP-FPM and the relay at the OS scheduler, and this is where that is settled: the crawl runs at full speed on an idle machine and yields the instant a call needs the core. The default lane is never niced, because it carries the caller's data across the boundary.

Sizing is documented rather than guessed. Chromium costs roughly 500 MB a slot, which puts two workers on a two-core machine and four on a four-core machine, and the ceiling has a reason: past three slots on two cores the browser goes CPU-bound and begins contending with the live-call path. Upgrading the machine changes one integer in one file.

Co-residency is what makes the two systems fast. Segmentation is what makes it safe.

Designed, built, and operated solo.

Two platforms, two hosts, one private segment. Laravel · Inertia · React · TypeScript · Node WebSocket sidecar · Twilio ConversationRelay · MySQL · Redis · Reverb · Playwright · Stripe · supervised worker groups.

Every architectural decision here was reached the same way: build it, run real traffic against it, find where it breaks, and design the mechanism that makes that failure impossible rather than unlikely. Relentless testing, simulation, and data-transport work — carried by one person from the first crawl to the live call.

Code excerpts are taken from the running system. Comments are verbatim; long parameter lists are elided for width. Figures are generated with D3 and printed as vector.