

Conversico: AI reception with human control

System design case study | Peter McCann Strain | September 2026

I cofounded Conversico and built the product in full as the sole technical member of the team. It was an AI reception platform for dental practices, designed to handle patient enquiries, scheduling, and follow-up across voice and messaging while keeping reception in control.

The business closed before we signed up a practice, while we were still working with synthetic data. I decided the expected return did not justify committing to it full time, and I wanted to work on other things. I resigned my position and retained the product, which I had built and owned.

This case study explains the implemented system and the decisions behind it. The module blueprint proposes stronger boundaries within that system; the scale estimates are retrospective planning scenarios, and the distributed design describes its hypothetical evolution.

01 / Business problem

A dental practice needs to handle patient enquiries, appointments, and follow-up while its reception team is already looking after people. The business question was how much of that administrative work an assistant could complete while leaving staff with a clear account of what happened and control over what happened next.

When a patient calls to book an appointment, sounding helpful is only the beginning. The assistant has to identify the practice, check which services are available, and establish whether an appointment was actually booked. A smooth conversation does not help if the patient expects to arrive on Tuesday and reception has only received a vague request.

I designed Conversico around that gap between conversation and completed work. Patients need a truthful outcome. Reception needs the conversation, the resulting appointment or request, and any action still outstanding. Managers need control over the information and behaviour presented on behalf of their practice; owners need visibility across the practices they are permitted to access.

The scope stays administrative. Clinical decisions remain with people, and escalation must preserve enough context for someone to take over. The measures of product value are completed administrative tasks and the work still left with staff. Those needs determine both what the system does and the conditions under which its results can be trusted.

02 / Functional requirements

The product needs to support a complete reception workflow across patients, staff, managers, and owners:

User	What Conversico needs to support
Patient	Get approved information; request, book, change, or cancel an eligible appointment through voice or messaging.
Receptionist	Review conversations and outcomes, arrange callbacks, confirm assistant proposals, and resolve outstanding work.
Practice manager	Maintain operational facts, configure scheduling capabilities, and review changes to the assistant's behaviour before publication.
Owner	Inspect activity and reporting across authorised practices.

Scheduling needs three explicit modes. A practice using live Dentally booking can write to its external diary when credentials and capability checks permit it. An internal-calendar practice can manage eligible appointments within Conversico. A request-only practice captures work for staff to confirm. These modes make different promises to the patient, so the assistant must know which one applies before offering an outcome.

Staff need a contextual assistant that can read the workspace, propose an action, and wait for confirmation before a protected write. Patient messaging needs contact lookup, identity verification, and confirmation of the intended change. SMS, email, and WhatsApp share the underlying patient workflow while retaining their own reply and delivery rules.

The workflow continues after the conversation. Conversico needs to record the outcome, retain unresolved tasks, and arrange eligible confirmations and reminders. A changed appointment must update pending follow-up. Practice configuration and reporting make those individual interactions manageable as a service operated by reception, rather than a collection of disconnected chats.

03 / Non-functional requirements

A booking confirmation needs evidence that the appointment exists. A repeated message must not execute the same completed protected action twice. When a provider fails after accepting work, the system must retain enough evidence to recover without guessing. These are correctness requirements: they apply even when traffic is low and the interface feels fast.

Access has the same priority. Staff can only use records within an authorised practice, and a matching contact detail alone cannot authorise a patient's appointment change. Staff actions need an audit record. The channel changes how the request arrives, but not the authority required to carry it out. Patient data also needs protection in storage and a defined retention policy.

Reception must be able to keep working while model and provider requests complete. My proposed service targets are p95 local-data reads below 500 ms, useful staff-assistant content within two seconds at p95, and 99.9% monthly availability for core staff APIs. A local-read measurement starts when an authenticated request reaches the API; useful assistant content means an answer or actionable result, not a loading indicator. Availability includes the authentication dependencies needed to use those APIs. These targets define a validation goal; they are separate from configured limits such as page sizes, connection pools, and retention periods.

Recoverability matters when a conversation finishes before its follow-up. Pending work must remain visible, repeated delivery must be handled safely, and staff must be able to distinguish a completed appointment change from a reply still waiting to send. Maintainability matters for the same reason: voice, messaging, and staff assistance need consistent scheduling rules as the product changes. Managed providers reduce the components I need to operate, while their latency, outages, and quotas remain dependencies the design has to accommodate.

04 / Planned scale and capacity considerations

For sizing, I use 10 practices as a small deployment scenario, 100 as the central planning case, and 1,000 to expose the next constraints. Each practice receives 40 calls a day, averaging three minutes across a ten-hour demand window, with a sustained fourfold peak and three active staff sessions. At 100 practices, that means 4,000 calls a day, 20 concurrent calls on average, 80 during the peak period, and 300 staff sessions. The voice providers carry the audio; Conversico handles the control requests, model work, and records those calls create.

The workload also depends on what those conversations do. I use a mix of 50% enquiries, 45% confirmed bookings, and 5% uncertain bookings. The synthetic journeys use three, nine, and nine voice model requests respectively, followed by one classification per completed call. Ten initial assistant turns per staff session per day are split between reads (80%, two model requests) and writes (20%, one). Twenty patient journeys per practice per day add one model request each. Dashboard polling runs every 30 seconds:

Demand dimension	10 practices	100 practices	1,000 practices
Calls per day	400	4,000	40,000
Mean / peak-period mean voice concurrency	2 / 8	20 / 80	200 / 800
Active staff sessions	30	300	3,000
Dashboard polling requests/second	1	10	100
Peak voice-control HTTP requests/second	0.4	4	40
Voice model requests/day	2,400	24,000	240,000
Staff + patient model requests/day	740	7,400	74,000
Post-call model requests/day	400	4,000	40,000
Eventual jobs generated/day	3,420	34,200	342,000

These are workload scenarios, not measured capacity. The booking journeys assume a verified patient, eligible messaging, and appointments far enough ahead for reminders. Eventual jobs include reminders due later; retries, evaluations, purge, and periodic work add demand. A confirmed patient cancellation can continue without another model request, while duplicate-message handling avoids repeating the original model work.

The sensitivity is more useful than the precision of the totals. At 1,000 practices, doubling mean call duration raises peak-period mean voice concurrency from 800 to 1,600. Two extra spoken turns per call add 80,000 daily model requests without adding a practice. Model quotas and cost therefore depend on conversation length and tool use as well as customer count.

As a stress scenario, compressing the eventual jobs into the fourfold peak window gives about 38 jobs per second. At 70% utilisation, a half-second mean service time needs roughly 28 concurrent worker slots; two seconds needs about 109. These are workload allowances, not machine counts. Queue age and measured service time determine the process count.

Database pressure follows transaction occupancy. At 1,000 practices, the model creates about 2.22 booking attempts per peak second. Holding a connection for one second averages 2.22 occupied connections for that work; four seconds raises it to 8.89. A three-minute call does not hold one connection throughout. Pool budgets must include API processes, workers, rollout overlap, and recovery headroom.

Retained data grows even when request rates stay modest. The synthetic encrypted slice averages about 9.8 KB and twelve transcript rows per call. At the configured 395-day horizon, the largest scenario retains roughly 155 GB of that serialized content and about 190 million transcript rows. Redaction can remove text while retaining rows. Indexes, other business records, backups, and provider audio need separate budgets; physical database sizing requires a populated PostgreSQL measurement.

These estimates separate audio concurrency from backend throughput, worker demand, and storage. The high-level design follows that separation: managed voice handles media, a shared backend owns application decisions, and separate processes execute work that can finish after the request.

05 / High-level architecture

The implemented backend is a layered monolith with functional modules, shared PostgreSQL persistence, and separate API, worker, and scheduler processes. It is the foundation for the modular monolith blueprint shown below. Booking, verification, and reception follow-up share practice and appointment state, so local application calls and transactions let me coordinate that work and test changes together.

Opening a practice workspace shows how the layers fit together. The browser requests the current practice context and then its appointments through the same-origin Next.js backend-for-frontend (BFF), which forwards the requests to FastAPI. Clerk supplies the staff identity; local memberships and roles determine practice access. The backend checks that authority when reading PostgreSQL.

The staff interface consists of typed, versioned resource APIs organised according to the practice and its appointments, and the workspace opens with two queries:

Interface	Purpose
Read current practice context	Resolve the practice and staff permissions on the server.
Read practice appointments	Return a filtered, paginated appointment list with a stable sort.

Once the server resolves the practice context, the appointment query has an authorised scope. The browser can request a view of the diary, but cannot establish its own authority over that practice. The query accepts a date range and typed filters, returns at most 100 records per page, and uses an identifier to break sorting ties. Offset pagination keeps reception’s lists simple, although a changing diary can move records between page requests.

The backend combines scoped queries with transaction-local PostgreSQL row-level security. A commit or rollback ends that transaction’s tenant context, so the next transaction must bind it again before

reading protected records. This matters when a workflow records an intention, waits for a provider, and then resumes local work: tenant isolation has to survive the whole operation, not just its first request.

Reception needs a prompt reply while other work continues. A call that has just ended may still need analysis, and an evaluation can take longer than a dashboard read. I gave those tasks separate processes within the same backend. Figure 1 shows where they run, with the providers carrying call audio outside the application's control path.

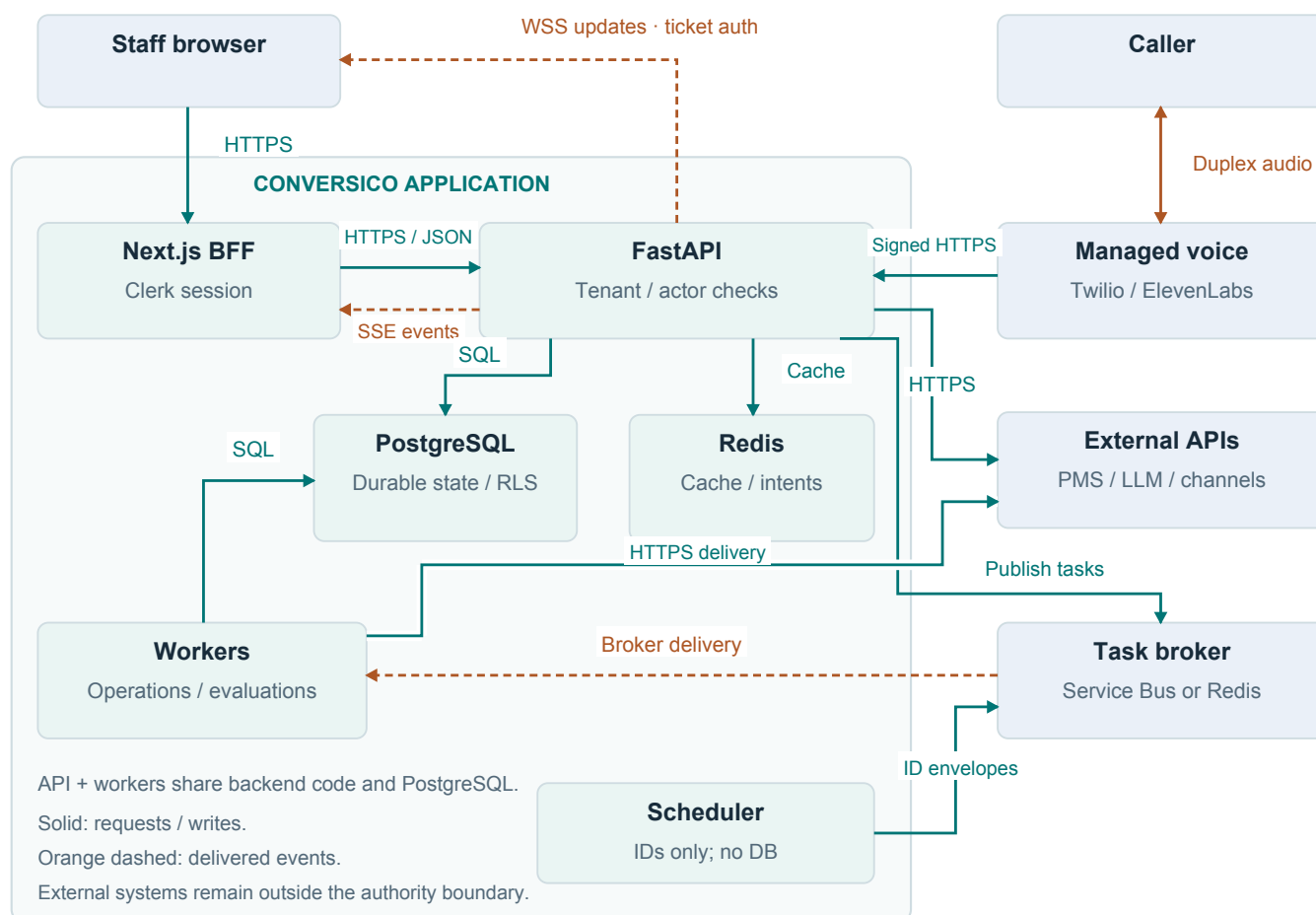


Figure 1. Current runtime. The staff workspace reaches a shared backend; managed providers carry the voice media. API and worker processes share application code and PostgreSQL.

The API and workers share application rules and PostgreSQL state. Redis provides temporary context, coordination, and live updates; queue adapters pass jobs to operational or evaluation workers. The scheduler publishes work identifiers without database or provider credentials. A slow analysis can continue independently of reception's request without distributing ownership of the appointment.

An eligible appointment needs the same definition across voice, staff assistance, and patient messaging. The module blueprint places those rules in shared scheduling and practice capabilities. Splitting the processes does not enforce that ownership by itself; the application boundaries do.

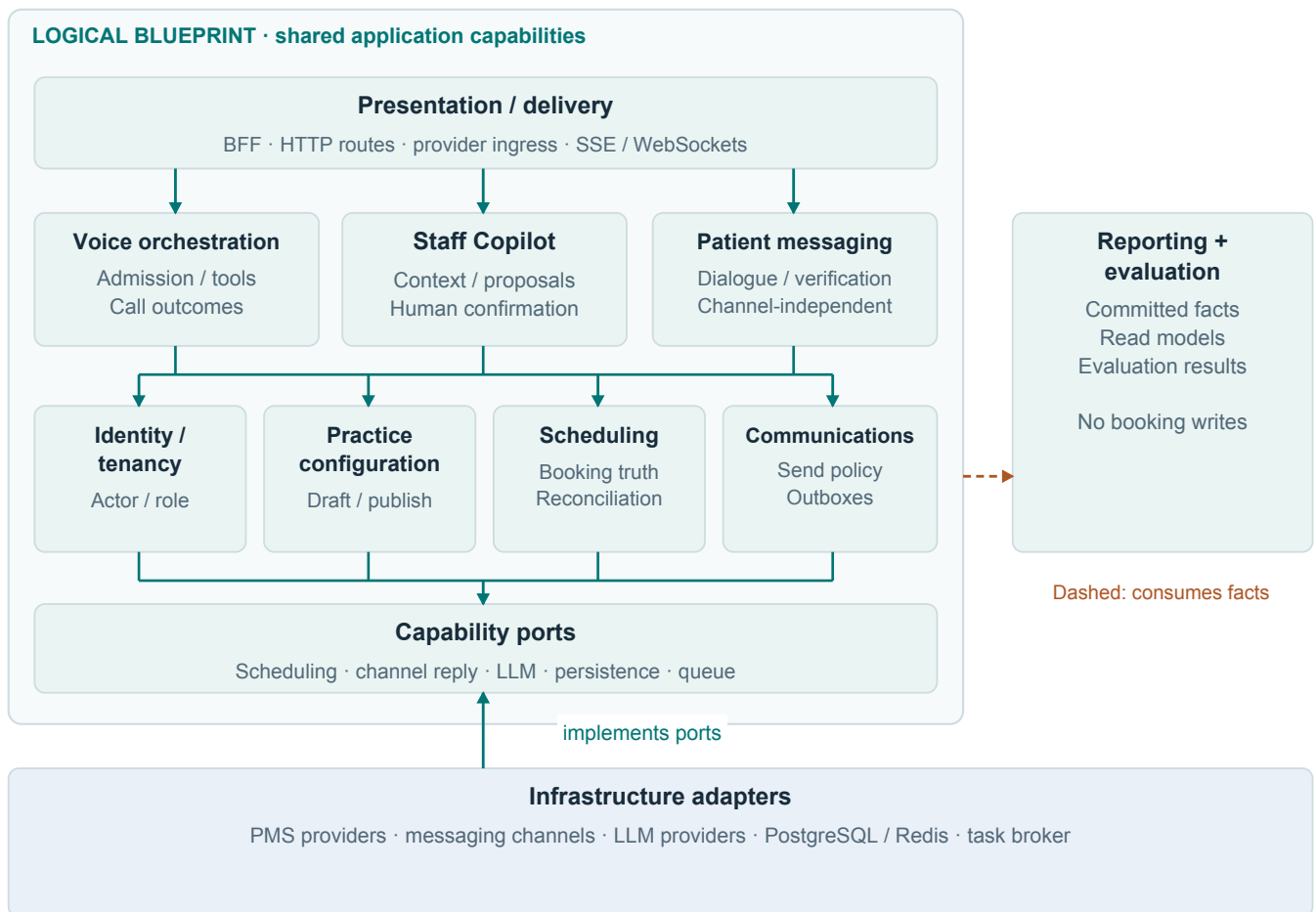


Figure 2. Proposed module boundaries. Conversation modules use shared application capabilities; infrastructure adapters implement the provider and persistence interfaces.

Copilot's route-owned reads expose a concrete boundary to improve. My proposed design moves them into shared application services, keeping tenant checks and result contracts together as more assistants use them. Delivery code accepts the request, application services coordinate the work, and provider adapters translate external contracts. Figure 2 assigns business responsibilities across those layers.

The existing backend import contracts and frontend dependency rules enforce parts of that separation, including browser/server boundaries and cycle checks. Their declared exceptions identify the remaining work. Those checks provide the guardrails for consolidating eligibility and scheduling outcomes below the conversation interfaces.

These are application modules and provider adapters, not a general plugin framework. Their shared database makes local coordination straightforward. An external practice diary sits outside that transaction boundary, which is where the first deep dive begins.

06 / Deep dives into the design

The architecture becomes easier to assess through the work it has to complete. The first journey follows a voice booking whose provider response disappears. A staff callback and a patient cancellation then expose different authority and confirmation boundaries. Their shared records explain how work survives beyond the conversation, and the tests establish which outcomes the application can claim.

Voice, published configuration, and booking recovery

In this synthetic example, a patient calls a practice with live Dentally booking enabled to ask for an appointment. Twilio sends a signed webhook; Conversico uses the called number to identify the practice, checks admission, and registers the conversation with ElevenLabs.

Twilio and ElevenLabs handle the audio, while Conversico supplies the practice's published context, adapts model requests through its LLM proxy, and authorises tools. At registration, the shared voice agent receives practice-specific variables and overrides for that call without changing the shared agent's global defaults. Changes to conversational behaviour need publication evidence; operational details such as hours refresh separately after save. Tools recheck call identity, tenant scope, patient ownership where required, and enabled capabilities. The model chooses a tool; the application decides whether it may execute.

The model proxy translates ElevenLabs requests for Anthropic. It can retry opening a response before any output begins, with a brief fallback if opening fails. Replaying an answer after output has started could repeat speech the patient has already heard. That is why I kept this retry boundary separate from recovery of a booking action.

The patient chooses an available time slot. The application must handle competing requests for that slot and prevent repeated requests from creating duplicate bookings. A local advisory lock and slot hold reduce competition between requests. Local uniqueness checks protect the appointment mirror, while the provider decides whether its diary accepts the booking.

For repeated requests, I used a durable operation ledger. The operation identity includes the patient and selected slot. Before making the provider write, the application records the operation and that an external attempt is about to happen. A stale claim that never reached the attempt can be reclaimed; a recorded attempt cannot be treated as though nothing happened. A unique marker goes with the booking so a later lookup can identify the appointment created by that operation. Known provider success is recorded before the local mirror is completed.

Dentally creates the appointment, but the connection drops before Conversico receives its reply. Repeating the write could book twice. Reporting failure could lead reception to create another appointment. Both "booked" and "failed" would go beyond the evidence Conversico has.

The ledger leaves the operation unresolved. The application queries the provider instead of retrying the mutation. Read-back must find exactly one appointment with the operation's metadata marker, expected start time and practitioner, and a provider appointment identifier. A marker in free-text notes is not enough. Several lookups are scheduled over the following half-hour, with a later check for unresolved work; none permits another write. Figure 3 places that uncertain result alongside the call's registration and completion.

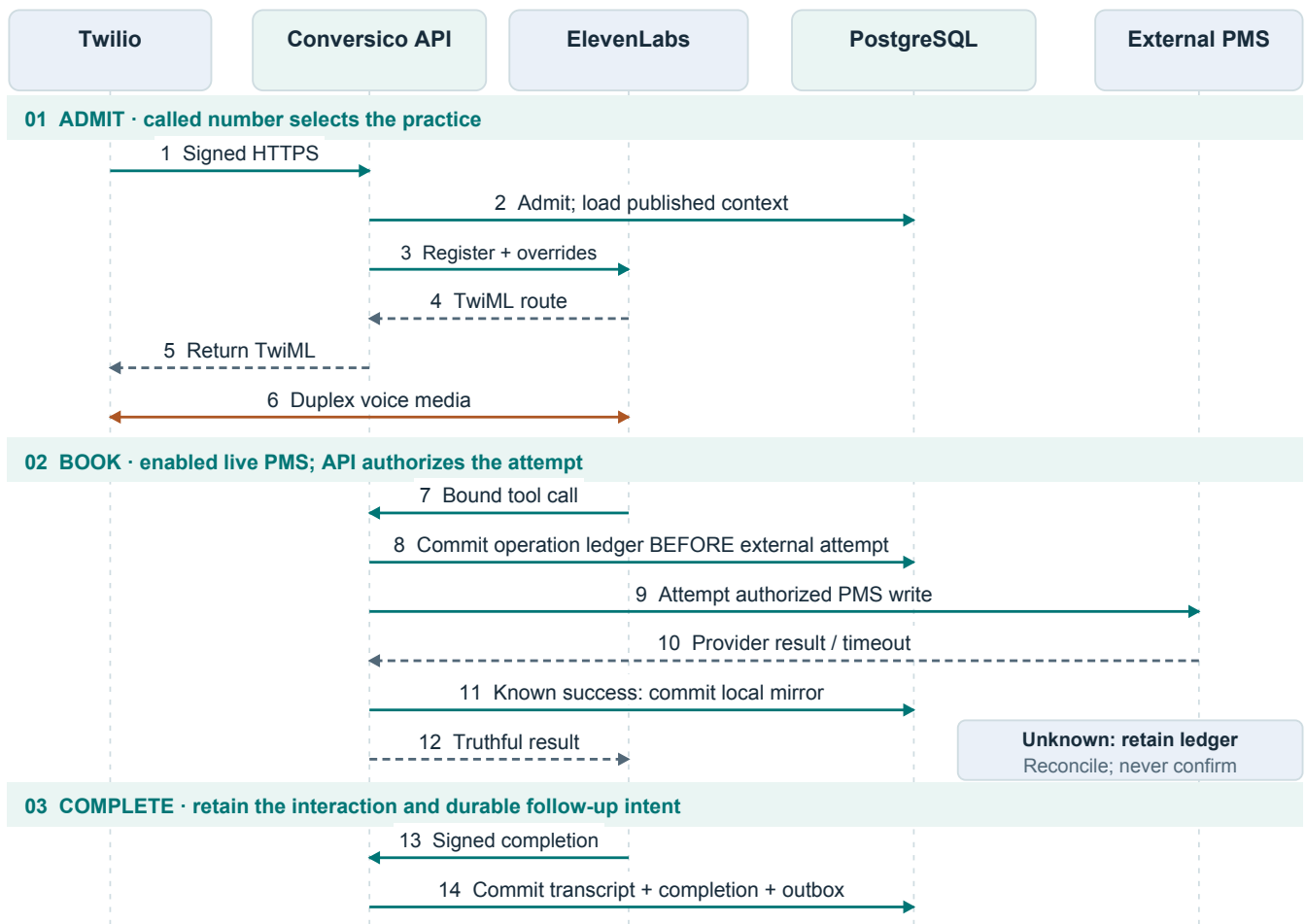


Figure 3. A call with live PMS booking enabled. Registration supplies practice-specific context; providers carry the audio while Conversico checks tools and records booking outcomes.

While lookup is pending, the voice tool returns an unresolved result with no confirmation reference. It instructs the agent to explain that the team is checking and to discourage another submission. The application creates booking-review work immediately, leaving reception the operation evidence after the call ends.

I leave the possibly valid booking in place during the investigation. Automatically cancelling it would involve another external write and could undo the appointment the patient wanted. The better course is to establish what happened before making another change.

In the recovery path shown here, the provider lookup finds the existing booking and local mirror completion succeeds. The worker marks the operation complete and closes its still-open matching review task. The appointment is now recorded both locally and in the provider's diary, and reception's review queue reflects the resolution. If local completion fails, the task stays open even with provider confirmation. That is one of the failures I test later.

If the outcome remains unresolved, reception has a recorded investigation to finish. Staff must explain what they established and who owns the resolution; the backend requires supporting review details and rejects overwriting closed work. Other interruptions use evidence already stored: known provider success allows mirror completion, while a completed local operation can return its recorded result. Neither needs another provider write. Completing reception's task does not, by itself, establish what happened in the provider diary.

The promise also depends on the practice's scheduling mode. Alongside live Dentally, Conversico supports an internal calendar and request capture for staff confirmation. Capture promises less from the outset; it is different from a booking that may already exist. Internal automation needs reviewed configuration, and live Dentally writes need credentials and capability evidence. A request-only practice collects work for its team. Unsupported configurations can raise an error, so capture is a deliberate mode rather than a universal fallback.

If admission or registration fails, an available staffed transfer or logged-call response gives the call a defined exit. A transfer has its own limit: the provider can accept a redirect without anyone answering. Reception therefore needs to distinguish an appointment, a captured request, and a conversation still needing attention.

Contextual staff assistance and confirmation

Reception also needs to initiate work. Consider a callback request for a patient already in the authorised practice, separate from the voice booking-review task. The receptionist asks the staff assistant through the dashboard, sending the conversation, current route, and recent workspace events with the question. Each event retains its kind, time, and label so navigation context reaches the model. That lets the assistant relate the request to the work on screen.

The backend supplies tools permitted for the staff member's role and checks the model's choice before dispatch. Reads and navigation return results; a write produces a proposed change for reception to inspect. Server-sent events (SSE) let the interface show those results as the conversation develops.

For the callback request, the interface displays a proposed diff and a separate proposal ID. Redis holds the proposal for 60 seconds. If the receptionist confirms before expiry, the client returns that ID and the server rechecks the actor, practice, and current role before consuming the proposal once. Consuming the Redis proposal and committing the database action are separate boundaries. The callback request and its audit then commit in PostgreSQL, leaving reception a callback to carry out and a record of who authorised it.

In Figure 4, the staff review is located between the proposed action and the execution.

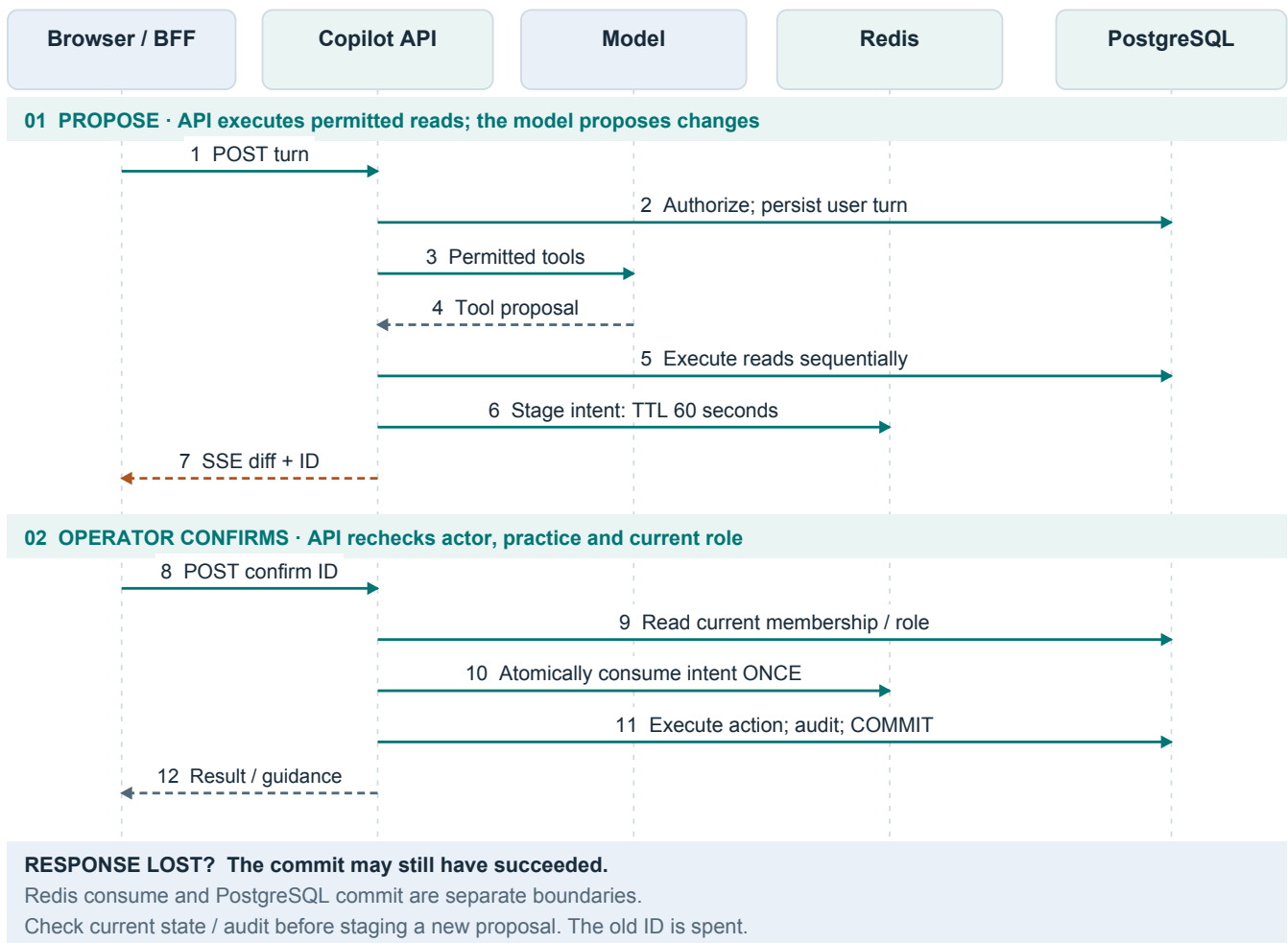


Figure 4. A staff callback proposal. The receptionist reviews the change before confirming; the backend rechecks authority and records the executed action.

The confirmation reply can still disappear after the callback request has been created. Reusing the consumed proposal would not establish whether that request exists. I kept the diff and review note visible, blocked reuse of the confirmation, and directed staff to current state or audit history before asking for another change. If success was already acknowledged, a later disconnection preserves that known result.

An executor error can arrive inside an HTTP 200 stream. The interface therefore uses the result event to decide whether the callback exists; the HTTP status only says streaming began. Missing or expired proposals are rejected before streaming starts, separately from action errors after it begins.

A single answer can contain several tool results. If reception sees a proposal before another result or error from the same turn, it has an incomplete basis for review. I show every dispatched result before a required confirmation pauses the model loop. The tools run sequentially because they share the request's database session, keeping one owner of its transaction state, and the loop has a fixed round limit.

The callback has a clear review boundary. Settings changes need an additional safeguard: an expected-version check to reject a proposal if another staff member changes the value during review. This is a proposed addition. Current settings writes recheck authority but do not compare the live value with the staged diff.

Staff can decide whether to confirm with their signed-in role already established. A patient arriving through a message has no equivalent staff identity.

Patient messaging and verification

An adult patient messages the practice to cancel an appointment. This separate synthetic example uses the internal calendar. The request is simple, but the application still has to establish who is asking before it changes the appointment.

A matching phone number helps locate a patient, but family members may share that number. Keyed indexes over normalised values allow exact matches while configured application encryption protects the contact fields. This supports contact lookup without enabling arbitrary plaintext searches.

A contact match still leaves authority unsettled. Across SMS, email, and WhatsApp, Conversico requires an active verified identity before a protected appointment change. If that identity is absent, it sends a verification link and leaves the appointment unchanged. The patient then asks again on a later turn; completing verification does not automatically execute the earlier request.

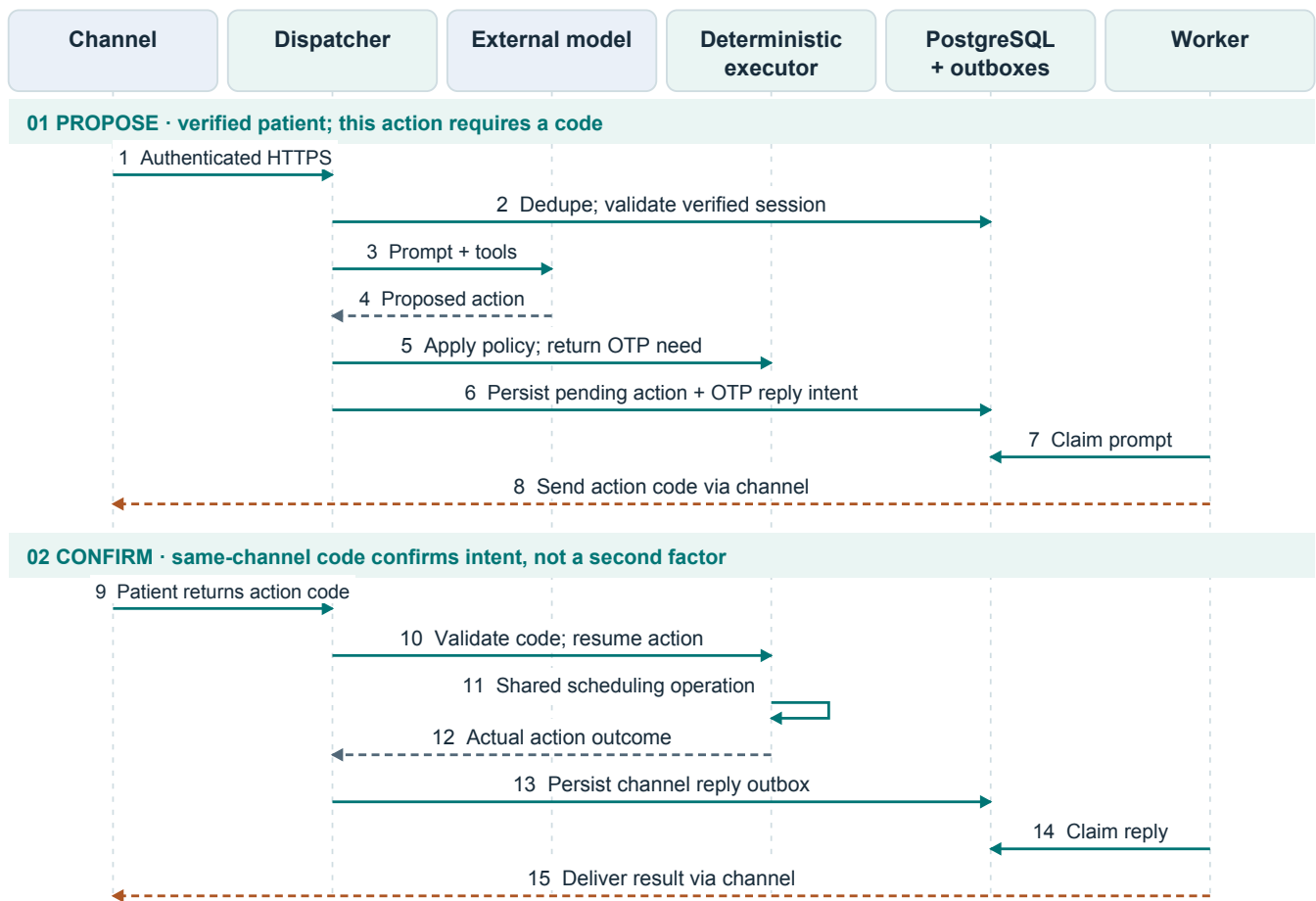
On the renewed request, automated changes must be enabled and the scheduling mode must permit the write. The application stores the proposed cancellation, appointment ID, and an action-code session bound to the active identity session. The appointment remains confirmed while the patient reviews the request.

The patient returns the six-digit code in the same conversation. This confirms their intent to perform the action; it is not an independent second factor. Before changing the calendar, the executor checks ownership, capability, and current appointment state. It also rechecks the identity session that authorised the proposal; an expired or replaced session cannot continue with an old code. Identity verification establishes who is asking, while the action code confirms which change they intend.

The appointment is now cancelled, and the reply is waiting in the outbox. A worker still has to deliver it. If the channel fails after the calendar changes, the stored cancellation remains valid and the undelivered reply remains work to do. Treating both as one success flag would hide which part needs recovery.

I kept verification and scheduling checks in shared patient processing so SMS, email, and WhatsApp could use the same rules. Channel adapters authenticate incoming providers and handle addressing, formatting, and delivery. The dispatcher binds each message to a practice and claims its provider identifier to suppress duplicate handling. Each adapter retains its own transaction boundaries; the shared workflow does not imply one atomic transaction across all channels. A new channel can use that workflow without recreating its authorisation and scheduling logic.

Figure 5 shows how the patient's request was confirmed, the calendar entry was changed, and the reply was queued for delivery.



Solid: application calls / writes. Orange dashed: worker delivery. Scheduling truth: Figure 3.

Figure 5. A verified patient confirms a cancellation. The same-channel code confirms intent; the executor rechecks authority and records a reply for later delivery.

The patient has cancelled, but a reminder for the old appointment may still be scheduled. Recording the cancellation needs to change that pending work too.

Durable records and follow-up

Cancelling an appointment cancels pending reminder intent and creates an acknowledgement. Moving it replaces pending reminders; a confirmed voice booking linked to a patient creates confirmation and reminder intent in the first place. I connected those records to the appointment lifecycle so a reminder still waiting to be dispatched can be stopped when its appointment changes. This cannot recall a message already handed to a delivery outbox.

Before an acknowledgement or reminder is enqueued, the communications orchestrator checks timing, contact suppression, and permitted send windows. Quiet hours delay the communication without switching channels to bypass the decision. Another worker may be preparing a different message for the same patient, so both jobs must share a send allowance. I reserve frequency-cap capacity with the outbox under a patient lock, so the second worker sees the allowance already taken. The reservation and outbox commit together. Queued delivery retains its allowance, and delivered or ambiguous attempts remain counted for a defined window. A slow or uncertain send cannot immediately make room for another message.

The cancellation is complete while its acknowledgement can still be waiting to send. Reception needs to see both facts, which is why I keep appointment state and communication state in separate records. The earlier call has the same need: its transcript explains the conversation, its ledger records the external

attempt, and its matching review task records unfinished investigation. Figure 6 brings those records together without combining their meanings into one “completed” status.

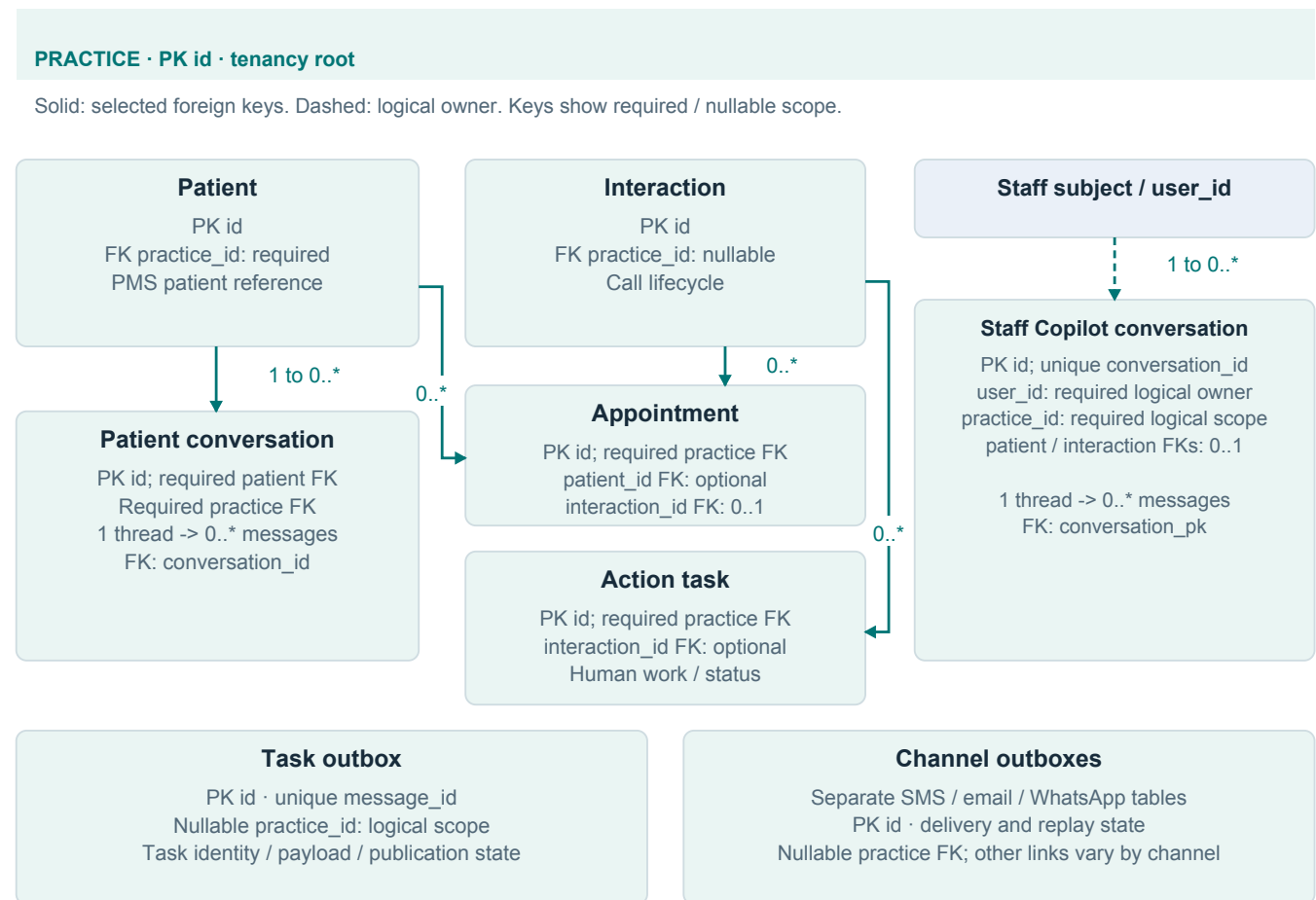


Figure 6. Selected records behind the workflows. Practice scope connects the data; calls, appointments, conversations, tasks, and outboxes retain distinct ownership and state.

The practice is the tenancy root, and each call has an interaction record with its transcript, tool receipts, evaluation results, and recording metadata. Audio recordings stay with the provider; Conversico owns their references and lifecycle policy. Ownership within the practice is explicit. Staff conversations belong to a staff subject; patient conversations require a patient. A callback awaiting reception and an outbox awaiting delivery remain separate from the appointment.

Protected records can also fail to open. If a field cannot be decrypted, the application redacts it in supported staff lists while strict individual-record reads fail. The rest of the queue can remain usable without treating unreadable content as a valid empty value.

The outbox keeps pending work visible when the application cannot immediately hand it to a broker. Saving the intent first means the database still identifies that pending handoff after a connection failure. I use the same pattern when a signed completion webhook arrives: one transaction stores the transcript and call state, marks the webhook complete, and retains analysis intent plus any required recording-purge intent. Publication happens afterward. Call-ended notifications and eligible satisfaction messages belong to later work, so the initial commit makes a narrower durability promise. Analysis can continue without depending on the original request process surviving.

Publishing work does not prove it finished. A worker can complete an operation and lose its acknowledgement, leading to another delivery. Its state transitions must tolerate that repetition. Publication retries are bounded, and failed publication retains its state for investigation. Post-call

analysis jobs carry identifiers so the worker reloads protected data under the right practice scope instead of copying patient details into every queue message.

The receptionist can close the page before this work finishes. On returning to the workspace, reception can inspect the stored appointment and outstanding work. The call feed separately reconstructs active calls from a database snapshot, followed by updates through a ticket-authenticated WebSocket. It buffers updates while reading the snapshot so changes arriving during the read are retained. Redis PubSub has no replayable event history. Call and tool events provide updates; the configured voice integration does not supply a continuous transcript stream.

Unresolved-booking age, pending-outbox age, and queue delay are the operational signals for locating stalled work. A cancellation with an undelivered acknowledgement needs a different recovery from a booking whose outcome is unknown. Keeping those states separate makes both visible and testable.

Testing the outcomes and publishing behaviour

For a missing booking response, a convincing test has to distinguish “the provider write may have happened” from “nothing was attempted.” For a cancellation, it must show when the appointment changes and when the reply is merely queued. I used deterministic backend fixtures to test those transitions without depending on live providers.

One recovery test supplies a confirmed provider result, then makes completion of the local appointment mirror fail. It checks that the operation still requires review and its review task stays pending. A provider confirmation alone would not show that the application had finished its own work; checking the stored operation and task catches that gap.

Another test sends four tools in one assistant turn: two proposed writes, a successful read, and a failed read. It checks that all four results reach the client in order and that the model does not start another round before confirmation. Checking only the final answer could miss the result or error reception never saw.

The patient cancellation test checks the appointment before and after action-code confirmation. It remains confirmed while the request is awaiting confirmation, then becomes cancelled after the patient returns the code. The reply is inspected in the outbox, with delivery outside that assertion. The same checks cover tenant authority, duplicate delivery, session ownership, and the result the interface keeps after disconnection.

The call also depends on a published practice snapshot. Backend tests can verify that a booking tool changes the right state, but cannot show whether a new greeting or instruction causes the agent to use it appropriately. A conversation simulation can answer that behavioural question without making a real booking. I keep its evidence separate from tests of application side effects and use it to review changes to published behaviour.

For a manager’s changed greeting, the publication gate requires evidence matching the current draft, practice, prompt, and agent versions. Stale or mismatched evidence blocks publication, and synthetic demo evidence cannot authorise live publication. Until publication succeeds, the greeting, persona, and red-line wording retain their published values. Checking only that the draft saved would miss whether the call could use it.

Hours and fees take a separate path: saving those operational facts can trigger a best-effort refresh after the save. Checking the saved value alone would miss a failed refresh of the voice snapshot. Separating

routine facts from reviewed behaviour lets the manager maintain practice information while keeping evidence for changes to the conversation.

The staff preview lets a reader inspect call review, contextual assistance, tool cards, and confirmation with synthetic data. Patient-message fixtures use the actual dispatcher with deterministic model and channel adapters. The preview shows the interface; the fixtures establish stored transitions. Instrumented tests also count database and model work within selected workflow boundaries. Those SQLite measurements explain what the path executes, not deployed PostgreSQL capacity. Separate audio trials would assess recognition, interruptions, and the interval from the end of speech to an audible response.

07 / Design tradeoffs

The shared backend gave me local transactions, a single place to change scheduling rules, and one release to test across the conversation interfaces. The cost is shared database capacity and coordinated releases. Separate workers isolate slow execution, but they still depend on the same code and data. For this product stage, that was a useful balance. Stronger module boundaries improve it without introducing network calls between every capability.

Managed voice made the media path a provider responsibility. I could concentrate on practice configuration, authorised tools, and the outcome of the call. The tradeoff is dependence on provider behaviour, availability, quotas, and latency. The proxy's retry boundary and the registration handoff handle specific failures; they do not remove that dependency. Running a speech and telephony stack would exchange it for a substantially larger operating responsibility.

For external bookings, I chose durable evidence and reconciliation over automatically retrying a write whose response disappeared. That leaves some patients waiting for an answer and creates review work for reception. It also avoids creating another appointment merely because the first response was lost. Automatic cancellation would add a further uncertain write. Keeping the operation unresolved until evidence settles it gives both recovery and staff review a clear starting point.

Confirmation adds another interaction for staff and patients. Staff must review a proposed write; patients must establish identity and confirm the intended change. Those steps make authority and intent explicit. A consumed staff proposal cannot safely be reused after a missing reply, so the interface retains its diff and directs reception to current state. Sequential tool execution similarly favours clear ownership of the database session over parallel reads. Separate sessions become worthwhile when traces show that serial reads materially delay reception.

Durable outboxes let appointment changes and their follow-up intent survive a failed handoff to a broker. They also introduce a visible interval between committing a change and delivering its message. Retries need duplicate-safe handlers, and pending work needs recovery. The live call feed makes a different tradeoff: Redis PubSub gives immediate updates without a replayable history, so a database snapshot restores the current view after reconnecting. Neither a queued reply nor a live notification is evidence that its recipient has acted.

These choices keep the critical rules in application code while allowing conversations and delivery to proceed at different speeds. Scaling should preserve those rules and address the resource or ownership boundary under pressure.

08 / Evolution as demand grows

I would scale and tune the existing processes first, using API latency, database pool waits, queue age, provider quotas, and reconnect behaviour to locate contention. Tenant count sets the workload scenario; it does not prescribe a microservice boundary. A process change is sufficient when the problem is execution capacity and the capability still benefits from shared ownership.

The live feed exposes one concrete constraint. It uses one PubSub connection per socket, with a pool of 50 per API process. At 3,000 staff sockets, that implies a theoretical floor of 60 processes before uneven distribution or headroom. Even 300 sockets need at least six pools. My first experiment is a shared subscription relay per practice and process, assessed through connection use, buffer growth, and reconnect latency. Adding a business service alone does not remove that connection pattern.

The native worker also processes batches serially in a fixed queue order. Several slow handlers can delay unrelated work. Fair polling and dedicated consumers are the next bounded experiments. Evaluations already have a dedicated queue and worker. Additional evaluation capacity or quotas, dedicated communications and analytics consumers, and separate realtime delivery become candidates for further isolation when measurements show that their resource use or failure patterns interfere with reception. Query plans and transaction duration need the same scrutiny before adding database connections or replicas.

A business service earns its place when independent scaling, releases, failure containment, or a separate owner justifies the operating cost. Persistent interference after process-level changes is one trigger; a provider integration needing its own release cadence is another. Scheduling is a strong candidate because its consistency rules form a coherent boundary. In the proposed design, it owns appointment mirrors, write operations, slot coordination, and reconciliation. The external PMS retains ownership of its diary.

Figure 7 shows that next stage. Reception keeps conversations and staff work. Scheduling owns booking outcomes, communications owns delivery, and each extracted service owns its writes rather than continuing to update shared tables.

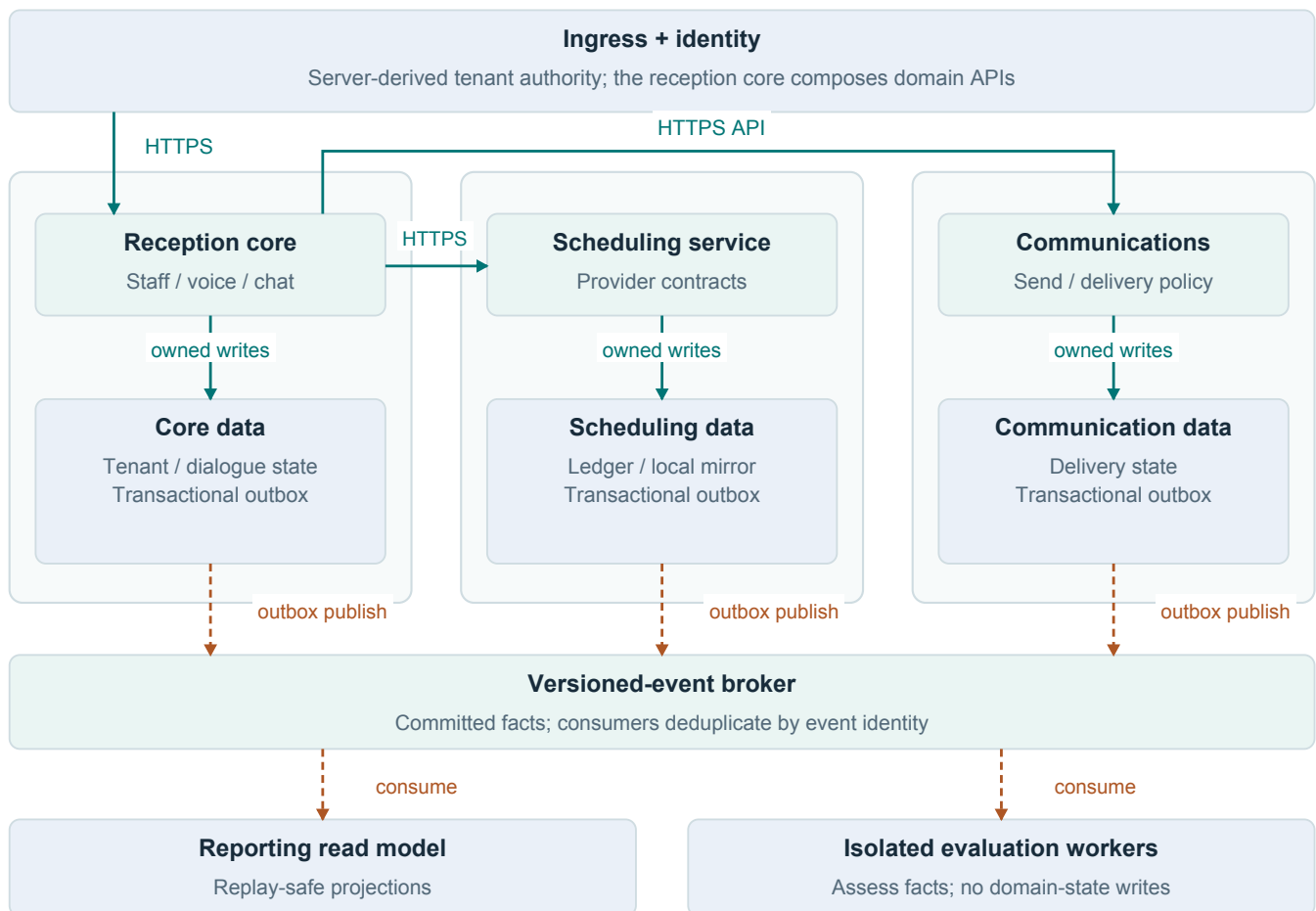


Figure 7. Prospective service ownership. Scheduling and communications own their data and publish versioned facts; the reception core composes their results for staff and conversations.

The current system already uses asynchronous jobs, outboxes, and live events. I would extend that foundation into event-driven integration when separate owners need to react to committed changes. A booking command still needs an explicit confirmed, rejected, or unresolved outcome. After committing its state, scheduling publishes a versioned fact through its outbox; communications and reception can consume it independently. That avoids making message delivery part of the booking's synchronous success path.

This introduces network failures and local views that can lag behind their owner. Scheduling retains a command identifier and request fingerprint, rejects changed content under the same identifier, and returns the recorded outcome for a repeat command. Consumers ignore duplicates and older outcomes. Versioned contracts, lag monitoring, and recovery become part of the operating cost. Event-driven integration is useful when that independence solves a demonstrated problem; adding an event bus alone does not establish ownership or consistency.

The migration starts by establishing the contract inside the monolith, backfilling the proposed service's data, and comparing shadow reads. Cutover requires pausing new commands, draining work, and fencing every old writer, including reconciliation, before transferring final changes and unresolved operations. Moving only the endpoint leaves recovery under the wrong owner. Rollback must also transfer changes made since cutover; without that transfer, repairing the new owner is safer than reactivating stale state.

The same boundaries govern my proposed follow-up assistant. It drafts callbacks through permitted tools; staff confirmation and a current-state check reject work reception has already resolved. The design starts in the existing worker runtime, with a task-and-version identity to suppress duplicate runs.

Dialogue evaluations assess proposals, while application tests challenge changed resolution, cross-practice references, unavailable providers, and earlier booking uncertainty. Volume, quotas, or release cadence must justify separate execution.

09 / Conclusion

The missing booking response makes the difference between a good conversation and a completed task concrete. A patient's appointment can exist before Conversico has enough evidence to confirm it. Keeping operation evidence and a matching review task lets the application recover without asking reception to guess or repeat the booking. I would make that distinction explicit at the start of another assistant project.

I would also define shared application boundaries earlier, including the route-owned reads discussed in the architecture. Scheduling commands need consistent confirmed, rejected, and unresolved outcomes, while request capture keeps its separate promise. Clear definitions make another interface easier to add without changing what an outcome means.

I would choose managed voice and the shared backend again. They let me focus on the point where an appointment request becomes an action someone has to rely on. The test for another agent or service would be whether it helps reception finish that work with a clearer account of what happened.