Messaging & Communications Decision Guide

Companion to the Messaging & Communications docs.

Section 1: Use Case & Participants

These questions establish the "who and why" before scoping any features. The answers should shape how you frame the rest of the conversation.

1.1 Who is messaging whom?

- Patient ↔️ provider
- Provider ↔️ provider
- Care team internally (no patient in the thread)
- System/automated → patient (one-way notifications)
- Some combination of the above

Why it matters: different participant types affect access control, thread structure, and whether a patient subject is always required on a thread.

1.2 What is the primary purpose of messaging in their product?

- Clinical care coordination (e.g. async visits, follow-ups)
- Patient engagement / support
- Internal operational communication
- Outbound notifications only (reminders, results)

Why it matters: determines how much of the feature set is actually needed and whether async encounters/billing will come up.

1.3 Is this a new build or replacing something they have today?

- If replacing: what are they replacing, and what are the gaps that drove this?
- If new: is messaging central to the product or additive?

Why it matters: existing systems often carry assumptions about data model or UX that will surface as requirements. Knowing this early prevents surprises.

Section 2: Feature Scoping

Go through each together. For each: Yes / No / Nice-to-have / Not sure.

#	Feature	Yes	No	Nice-to-have	Not sure
1	Live / real-time updates (new messages appear without refresh)				
2	Message response tracking and routing (assigning threads to providers or queues)				
3	Read receipts and unread counts				
4	File and image attachments				
5	Message editing and drafts				
6	Automated messaging (reminders, out-of-office, SLA escalations)				
7	External channel delivery (SMS, email — beyond in-app)				
8	Async encounters / billing				

Section 3: Feature Deep Dives

Cover each feature flagged Yes or Nice-to-have. The goal is to land on a clear recommended approach by the end of each section.

3.1 Thread Structure *(always covered)*

Questions:

- Are threads always tied to a specific patient, or do you need internal/admin threads with no patient context?
- Are threads 1:1 only, or do you need group conversations with more than two participants?
- Do threads need to be tagged or categorized (e.g. by specialty, product line, urgency)? If so, do tags need to be combinable (e.g. a thread tagged both “cardiology” and “urgent”)?

What the answers drive:

Situation	Approach
Threads always tied to a patient	Set `subject` on every thread header — enables filtering by patient across the inbox
Internal/admin threads needed	`subject` is optional — don’t assume it’s always present in queries or UI
1:1 only	Simpler participant model; simplifies read receipt implementation (see 3.4)
Group threads needed	Recipient list on thread header must include all participants including the thread creator

Situation	Approach
Tags/categories needed, single dimension	Use a single `category` entry on both the thread header and child messages
Tags/categories needed, multiple combinable dimensions	Use multiple `category` entries (e.g. one for specialty, one for credential level) — more maintainable but queries may need to match several categories

3.2 Live Updates

Questions:

- Do new messages need to appear in real time without a page refresh?
- Is this needed for the patient-facing app, the provider-facing app, or both?

What the answers drive:

Situation	Approach
Real-time updates needed	WebSocket subscriptions on the thread — requires this to be scoped and enabled for the project
Polling is acceptable	Simpler implementation; periodic re-fetch of thread messages
Notification only (e.g. badge, push) without live message rendering	Can use subscription to trigger a notification without requiring in-place message updates

3.3 Routing & Assignment

Questions:

- Do messages get assigned to specific providers, to a pool/queue by role, or both?
- Can threads be reassigned or escalated after initial assignment?
- After reassignment, does the previous owner need to retain visibility of the thread?
- Do you need a structured audit trail of who owned a thread and why it was rerouted?

What the answers drive:

Situation	Approach
Individual assignment only	Set `Task.owner` to a specific provider; remove `performerType`
Pool/queue routing	Set `Task.performerType` by role type; clear `Task.owner`; providers claim from pool
Both	Use `performerType` for initial pool routing; set `owner` and clear `performerType` when claimed
Reassignment needed, previous owner loses access	Update `Task.owner` in place; update `Communication.recipient` to match
Reassignment needed, previous owner retains visibility	Create a new Task for the new owner; mark original as cancelled with a note pointing to the new Task
Audit trail needed (free text)	Append to `Task.note` on each reroute with author and timestamp

Situation	Approach
Audit trail needed (structured/queryable)	Create a `Provenance` resource on each reroute with a coded reason — queryable via `Provenance?target=Task/{id}`

3.4 Read Receipts & Unread State

Questions:

- Are threads ever group conversations (more than 2 participants)?
- Do you need to query unread counts via API — e.g. a badge count or a list of all unread threads for a user?
- Do unread counts need to be tracked per-recipient in a group thread?

What the answers drive:

Situation	Approach
1:1 threads only	Option A: mark message `status = completed` when read; query unread via status filter. Simplest model.
Group threads, no API unread query needed	Option B: store per-participant read state as an extension on the thread header. Shows unread dot in UI; not searchable via API.
Group threads, API unread query needed	Option C: create a read-receipt Task per recipient per message; complete it when read. Supports badge counts and unread queries. Most complex.

3.5 Attachments

Questions:

- What file types need to be supported?
- Do attachments need to be searchable or referenceable as clinical documents — e.g. queryable by patient or document type — or are they just files to download within a message?

What the answers drive:

Situation	Approach
Files are just downloads within a message	Store as `contentAttachment` on the message payload — simple, no separate resource needed
Attachments need to be clinical documents (searchable, referenceable by patient/type)	Create a `DocumentReference` and attach via `contentReference` on the payload — file becomes a first-class searchable clinical document. Note: this is the default behavior of Medplum's React `ThreadChat` component.
Attachments are references to existing clinical resources (e.g. a lab result)	Use `contentReference` pointing at the relevant FHIR resource (e.g. `DiagnosticReport`)

3.6 Editing & Drafts

Questions:

- Can sent messages be corrected or retracted after sending?
- If so, does the original message content need to be discoverable via search (e.g. "show all edited messages"), or is it sufficient for the history to exist but not be directly queryable?
- Do users need to save a draft and return to it across sessions or devices, or is per-device/browser draft storage acceptable?

What the answers drive:

Situation	Approach
Sent messages cannot be edited	No special handling needed
Sent messages can be corrected; edits need to be searchable	Retract-and-correct: mark original `status = entered-in-error`, create new message with corrected content linked via `inResponseTo` and tagged with a `correction` category. Corrections are discoverable via status and category filters.
Sent messages can be corrected; searchability of edits not required	In-place payload update via `patchResource`. Full version history still exists via `_history` but is not searchable.
Drafts only needed per browser/device	Store in browser `localStorage` keyed by thread ID — no server resources needed, but lost if user clears storage or switches devices
Drafts need to persist across devices/sessions	Store as `Communication` with `status = preparation`; update as user types (debounced); promote to `in-progress` with a `sent` timestamp when sent. Filter drafts out of thread message queries with `status:not=preparation`.

3.7 Automations

Questions:

- Are there events that should trigger automated messages (e.g. new lab result, appointment reminder)?
- Are there SLA or response-time requirements (e.g. all messages responded to within 4 hours)?

- Do providers have out-of-office or availability states that should affect routing?
- Do you need to report on SLA compliance or response time metrics?

What the answers drive:

Situation	Approach
Event-triggered automation (e.g. send message when lab result arrives)	Bot triggered by a Subscription on the relevant resource type
Recurring automation (e.g. flag threads with no response after N days)	Cron-scheduled Bot that scans for stale threads and creates reminder Tasks
Out-of-office / availability-based rerouting	Bot triggered on new message checks provider Schedule; reroutes Task to pool if provider has no available slots
SLA reporting needed	Task data (`authoredOn`, status change timestamps, `owner`, `priority`) should be exported to an analytics platform — FHIR search is not designed for aggregate SLA metrics

3.8 External Channels

Questions:

- Which channels — SMS, email, other?
- Is this outbound-only (notify the patient) or does the patient reply back into the thread?
- If inbound replies are needed, how is the patient identified from an incoming message (e.g. phone number, email address)?

- Does the external provider (e.g. Twilio) have its own conversation ID that can be stored for thread matching?

What the answers drive:

Situation	Approach
Outbound notifications only	App creates the Communication, then calls `executeBot` directly to send via the external channel — not Subscription-triggered, so errors surface synchronously
Inbound replies needed	Inbound webhook Bot receives the payload, identifies the patient, matches or creates a thread, creates a child Communication
External conversation ID available (e.g. Twilio conversation SID)	Upsert thread header by that identifier on first message; use it as conditional `partOf` reference on all inbound messages — most reliable thread matching strategy
No external conversation ID	Match thread by patient phone/email and open thread status; ambiguous cases need a defined fallback strategy
Round-trip (provider replies in-app, patient receives via SMS/email)	Store channel on thread header as `medium`; app creates the Communication then calls `executeBot` to dispatch outbound — Bot reads `medium` to route to the right channel
Webhook retry / duplicate messages likely	Use `createResourceIfNoneExist` with the provider's message ID as an identifier to prevent duplicate inbound Communications

3.9 Async Encounters / Billing

Questions:

- Does the customer intend to bill for messaging-based care?
- How do you define a "session" — one thread per session, grouped by day, or patient-initiated?
- Can a single messaging session involve multiple patients (e.g. a parent asking about two children)?
- Do you need to capture diagnosis codes, service type, or other clinical details per encounter?

What the answers drive:

Situation	Approach
No billing intent	No Encounter resource needed; thread stands alone
Billing needed, one patient per session	Create an Encounter per session; link to the thread header via `Communication.encounter`
One thread = one session	Straightforward 1:1 mapping of thread to Encounter
Rolling interaction model (e.g. continuous text thread, grouped by day)	One Encounter spans multiple threads or a time window; multiple thread headers link to the same Encounter
Multiple patients can be in one session	Create a parent session Encounter + one child Encounter per patient via `Encounter.partOf`; clinical details live on child Encounters
Clinical documentation needed for billing	Populate `Encounter.participant`, `Encounter.reasonCode`, and `Encounter.serviceType` per patient Encounter

Related Integrations

The external messaging channels Medplum integrates with first-party — both deliver through FHIR `Communication` resources, so they slot into the thread model above:

- **Twilio SMS** — send and receive SMS via the `$send-sms-twilio` operation.
- **eFax** — send and receive faxes via the `$send-efax` and `$receive-efax` operations.