PRACTICAL LEGAL TECH

Connecting Zoom to Your Practice Management System

Three field guides for law firms: the architecture, the MyCase build, and the hands-on setup runbook.

practicallegaltech.com

By Staff

Connecting Zoom to Your Practice Management System: A Lawyer's Field Guide

Every practice management vendor sells the same fantasy: a "Zoom integration" button that magically wires your calendar, your matters, and your call logs to Zoom. Then you click it and find out it pastes a meeting link onto an event. That is it. The other ninety percent of what you actually wanted is on you.

So before you go shopping for a connector, get clear on what "connecting Zoom" even means. It is not one thing. It is four separate jobs, and each one uses a different part of Zoom. The whole skill here is picking the smallest part that does your job and ignoring the rest.

I built Zoom Phone into my own practice management system, so this is field experience, not a brochure. Here is how it actually works.

First, pick your job. There are only four.

1. **Schedule meetings from your PMS.** A consult gets booked; a Zoom link should exist and ride along onto the matter. This is the Zoom REST API, or a no-code bridge.
2. **React to what happens in a meeting.** Meeting started, meeting ended, recording is ready. This is webhooks or websockets.
3. **Log phone calls to the matter.** A client calls your firm number; the call, the duration, and the recording should attach to the file. This is Zoom Phone. For a law firm this is the sleeper hit.
4. **Pull recordings and transcripts into the matter.** This is the REST recordings API for predictable pulls, or Zoom's MCP server when you want AI to search across meetings by content.

Most firms think they want Job 1. The real money is usually in Jobs 3 and 4.

Pick yours now. The rest of this guide is organized around them.

Build the Zoom app and provision it (the part nobody shows you)

Every guide tells you to "connect Zoom." None of them show you the actual work, which is building an app inside Zoom and provisioning it. That is where the time goes, and it is where I am going to spend yours, because once the app is built and provisioned right, the connections are the easy part. I built ours; here is exactly what it takes.

Pick the app type first. In the Zoom App Marketplace, go to Develop, then Build App. The choice you make here decides everything downstream:

- **Server-to-Server OAuth (S2S)** automates your own firm's Zoom account from your own backend. No human clicks "allow," no redirect flow, no Marketplace review. This is what a firm wiring up its own office wants. Ours is an S2S app; we named it "NLF PMS."

- **General app (User OAuth)** is for software that acts on *other firms'* Zoom accounts. Building a product other lawyers log into, you need this. Wiring your own office, you do not. Its tokens carry a refresh token, and the rule that bites is that every refresh returns a new refresh token and kills the old one; save the new one or you are locked out.

The old "JWT app" type is dead, retired June 2023. If a tutorial says create a JWT app, the tutorial is stale. The rest of this assumes S2S, because that is the firm case.

Building the app hands you three credentials. On the app's Credentials page you get an Account ID, a Client ID, and a Client Secret. Those three are how your backend gets a token. Put them in your environment, never in code, never in chat.

Provisioning is four steps, in order:

1. **Fill in the basic Information.** Company name, developer contact, the boilerplate. Zoom will not let you activate until the required fields are filled. Annoying, mandatory, two minutes.
2. **Add scopes, and add only what the job needs.** This is the real work, and where time gets lost. You add scopes per capability: read call logs, read recordings, read meetings, and so on. Two hard-won rules: - **The Marketplace picker lies about names.** Zoom split its old coarse scope names (like `phone:read:admin`) into granular ones (like `phone:read:call_log:admin`). The picker still shows the coarse names; the token your app actually receives lists the granular ones. Trust the token, not the picker. This mismatch cost me real time before I caught it. - **Verify the grant from the token itself.** After saving scopes, mint a token and print its `scope` field. If a scope is not in that printed list, your app does not have it, whatever the picker showed. That one check kills a whole category of phantom permission errors.

For the common law-firm jobs, the scopes that actually matter are read-only and look like this: - Phone call logging: `phone:read:call_log:admin` (the critical capture scope), plus `phone:read:call_recording:admin`, `phone:read:recording_transcript:admin`, `phone:read:voicemail:admin`, and `phone:read:user:admin` (gives you each user's phone ID). - Meetings and recordings: `meeting:read:meeting:admin`, `cloud_recording:read:meeting_transcript:admin`, and `meeting:read:summary:admin` for AI Companion summaries. - Users: `user:read:list_users:admin` and `user:read:user:admin`, so you can map Zoom users to your staff.

Note what is missing on purpose: no `phone:write` scope, which means click-to-dial and outbound calling are not authorized. Adding a write scope later forces a re-consent. Least privilege is not just hygiene; every extra scope is a bigger key to your client data.

1. **Set up Event Subscriptions, if you want events.** Need Zoom to tell you when a call finishes or a recording is ready? Add an Event Subscription, pick a delivery method (Webhook to an HTTPS endpoint, or WebSocket), then select the event types. Two things bite people: - **There is a second secret.** Event subscriptions use a Webhook Secret Token that is separate from your Client Secret. You need it for the validation and signature steps in Job 2. Do not confuse

the two. - **Do not subscribe before your receiver is live.** Zoom validates your endpoint the instant you save a webhook subscription, and the save fails if your endpoint cannot answer the validation challenge. Build the receiver first, then subscribe. One app holds up to 20 subscriptions and can mix Phone, Meeting, and Recording events freely.

2. **Activate the app.** S2S apps activate on your own account; there is no Marketplace submission or review for internal use. Once activated, the credentials work.

Getting a token. With the app live, your backend asks for a token using the `account_credentials` grant. The token endpoint wants HTTP Basic auth built from your Client ID and Secret; let your HTTP client build that header rather than hand-rolling an encoding step:

```
curl -s -u "$ZOOM_CLIENT_ID:$ZOOM_CLIENT_SECRET" \
  -X POST "https://zoom.us/oauth/token?
grant_type=account_credentials&account_id=$ZOOM_ACCOUNT_ID"
```

The token lasts one hour. Cache it and refresh about five minutes before it expires; do not fetch a fresh token on every call. There is no refresh-token dance for S2S; when it ages out, you ask again.

The provisioning gotcha that ambushes everyone: host user IDs. An S2S app is not "you"; it is the account. So when you create a meeting or pull a user's call logs, you must pass a real user ID, and you have to go fetch those IDs first. Pull them from `GET /v2/users` and, for Zoom Phone, `GET /v2/phone/users`. Critical trap: a person's meeting user ID and their phone user ID are different values. Pull both explicitly, never assume they are equal, map them to your staff, and store them. Everything downstream keys off these IDs.

That is the build and the provisioning. Now you build the connections.

Job 1: Auto-create Zoom meetings from your PMS

The mechanics are simple. You POST to Zoom and a meeting comes back with a join link:

```
curl -X POST "https://api.zoom.us/v2/users/HOST_USER_ID/meetings" \
  -H "Authorization: Bearer ACCESS_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "topic": "Smith Consult",
    "type": 2,
    "start_time": "2026-07-15T15:00:00Z",
    "duration": 30,
    "settings": { "waiting_room": true, "join_before_host": false }
  }'
```

Take that `join_url` from the response and write it onto the matter. Done.

Three landmines, in order of how often they bite:

- **The `me` trap.** With S2S OAuth you must put a real user ID or email in the path. The shortcut `me` only works for User OAuth apps. Put `me` in an S2S call and you get a confusing "invalid token" error that has nothing to do with your token.
- **The 100-per-day wall.** Each Zoom user can create at most 100 meetings per day. That is a hard per-user limit, separate from rate limits. A high-volume firm spreads meeting creation across several host users.
- **Rate limits scale with your plan.** Free accounts get a trickle; Pro gets more; Business and up gets a real budget. The limits are shared across every app on your account, so build in retry-with-backoff and watch the `X-RateLimit-Remaining` header.

The crutch you will be tempted by. A paid broker like Zapier can watch your PMS for a new event and fire a "Create Meeting" in Zoom without you running anything. Skip it. You are renting a middleman to do badly what a short script does well, and you are capped at whatever fields the broker's trigger happens to expose. That cap is not academic; in the MyCase subguide you will watch it produce Zoom meetings that go out with the client never invited, because the broker never receives the client's email. Call the API yourself and you have the email. Going direct is barely more work, and it actually delivers.

Job 2: React to meeting events (and the law-firm security angle nobody mentions)

When a meeting starts, ends, or finishes recording, you want your PMS to know. Zoom offers two ways to hear about it, and the choice actually matters for a law firm.

Webhooks are push. You stand up a public HTTPS endpoint; Zoom POSTs events to it. Simple to start, but you are now running a public door that the whole internet can knock on, so you have two mandatory mechanics.

First, the **URL validation challenge.** The moment you register the endpoint (and on every config change), Zoom POSTs an `endpoint.url_validation` event with a `plainToken`. You have to echo it back, plus an HMAC-SHA256 of it keyed with your Webhook Secret Token, within three seconds:

```
// On event === 'endpoint.url_validation':
const encrypted = crypto.createHmac('sha256', WEBHOOK_SECRET_TOKEN)
  .update(req.body.payload.plainToken).digest('hex');
res.status(200).json({ plainToken: req.body.payload.plainToken, encryptedToken:
encrypted });
```

Get this wrong and the subscription will not even save. Second, **verify the signature on every real event.** Rebuild the hash from the raw request body and the Webhook Secret Token and

compare; drop anything that does not match, and drop anything whose timestamp is over five minutes old (replay guard):

```
const payload = `v0:${timestamp}:${rawBody}`;
const hash = crypto.createHmac('sha256',
WEBHOOK_SECRET_TOKEN).update(payload).digest('hex');
if (signature !== `v0=${hash}`) return res.status(401).send('Invalid signature');
```

One more rule that catches people: **acknowledge within three seconds, do the real work async.** Answer Zoom with a 200 immediately and hand the event to a worker; if you try to process inline and take too long, Zoom treats it as failed and retries on a punishing schedule (roughly 5 minutes, 30 minutes, 2 hours, 5 hours, 10 hours). Build for idempotency, because you will see duplicates.

Websockets are pull. Your server opens a persistent connection *out* to Zoom and events stream down it. No public endpoint, nothing exposed inbound. For a profession that lives and dies on confidentiality, that is a real advantage; you are not poking a hole in your firewall and praying your signature check is airtight. The tradeoff is you manage a long-lived connection, send heartbeats so Zoom does not hang up on you, and reconnect when it drops. Note one quirk: only one websocket connection per subscription is allowed at a time.

Rule of thumb: if you already run a hardened web app, webhooks are fine. If you are security-paranoid or do not want a public endpoint at all, websockets. Either way, the highest-value event to catch is `recording.completed`, which is your trigger for Job 4.

Job 3: Log Zoom Phone calls to the matter (the sleeper hit)

This is the one most firms overlook, and it is the one I built first. Your firm's phone is a goldmine of unbilled, unlogged client contact. Zoom Phone lets you capture it.

You have three surfaces, and you can mix them:

- **REST API plus webhooks** for the data. Catch the `phone.*` events as calls happen, then pull and store the call records. The identifiers you must persist to correlate everything later are `call_id`, `call_history_uuid`, and `call_element_id`. Map those to your matter and the call now lives on the file.
- **Smart Embed** for the humans. This drops a working softphone pane *inside* your PMS as an iframe and talks to your app through postMessage events. Your staff dials, logs disposition, and writes call notes without leaving the matter.
- **Click-to-dial URIs** for the lazy-but-smart path. A `zoomphonecall://` or `zoomphonesms://` link in your contact table launches the call straight from Zoom. No backend required to get started.

The payoff is concrete: every client call, with timestamp and recording, automatically filed to the right matter, ready to bill. That is hours a month a small firm currently throws away.

Job 4: Pull recordings and transcripts into the matter

Two tools, two purposes; do not confuse them.

REST recordings API is for predictable, deterministic pulls. The `recording.completed` webhook tells you a recording is ready; you then download the files using a Bearer token and follow the redirect to the actual file. Store it on the matter. This is the workhorse.

Zoom's MCP server is for AI. It exposes a small set of tools (`search_meetings`, `recordings_list`, `get_recording_resource`, `get_meeting_assets`) that let an AI agent search across your meetings by *content*, not just by date, and retrieve transcripts and summaries. This is how you ask "what did we agree to with the Smith family back in March" and get an answer instead of scrubbing a video. Two things to know: it runs on User OAuth with its own narrow scopes (not the broad REST scopes), and the good results depend on Zoom's AI Companion features like Smart Recording and Meeting Summary being turned on for your account. MCP does not do meeting create, update, or delete; for that you go back to REST. Use REST to *do* things and MCP to *find* things.

The layer you cannot skip: confidentiality and ethics

You are wiring a video and call platform into your client files. Treat it like the privileged data pipe it is.

- **Encrypt tokens at rest.** Never store a Zoom token in plain text. AES-256 at minimum. A leaked refresh token is a standing key to your firm's Zoom.
- **Least-privilege scopes.** Ask Zoom for only the scopes the job needs. A meeting-scheduler does not need recording-download rights.
- **Waiting rooms on, join-before-host off** for client meetings, so a client never lands in an empty room or, worse, the wrong one.
- **Data residency.** If your clients or rules demand US-only data handling, Zoom publishes regional API base URLs (`api-us.zoom.us`, `api-eu.zoom.us`, and others). The `api_url` field in your token response tells you your account's region.
- **Recording consent.** Catching `recording.completed` is easy; the harder duty is making sure everyone on the call knew they were recorded and that two-party-consent rules in your jurisdiction are satisfied. That is a you problem, not an API problem.

This is general information on building the plumbing, not legal advice on your specific ethics obligations; check your state's rules on cloud vendors, client confidentiality, and recording consent before you ship.

Build, buy, or bridge?

- **Bridge (a paid broker like Zapier):** fastest to click together, no server, and the wrong default. Fragile, monthly rent, and capped at whatever fields the broker's trigger exposes. Fine for a throwaway test; not for anything a client depends on.
- **Buy (a vendor's native connector, if one exists):** zero build, but you get exactly the features the vendor decided to build and nothing more.
- **Build (S2S OAuth plus REST plus webhooks):** full control, especially for Zoom Phone call logging, which is where the real return is and which the no-code paths handle poorly. More work up front; pays for itself in captured billable time.

Troubleshooting the usual suspects

- **401 Unauthorized:** token expired (they only last an hour) or you are missing a scope. Re-mint the token; check the scopes on your Marketplace app.
- **"Invalid token" when you swear the token is fine:** you used `me` in an S2S app. Put a real user ID in the path.
- **429 Too Many Requests:** you hit a rate limit. Back off and read the reset header; do not hammer.
- **Redirect URI mismatch (error 4709):** the number-one OAuth error. Your redirect URI must match *exactly*, down to the trailing slash and http versus https.
- **Refresh token suddenly invalid (4735):** you reused an old refresh token. Save the new one returned on every refresh.
- **"Does not contain scopes" (4711) when the picker clearly shows the scope:** the picker shows coarse names, the token carries granular ones, and the scope you need is not actually in the grant. Mint a token, print its `scope` field, and add whatever is missing in the Marketplace (an owner action), then re-consent.
- **Webhooks not arriving:** your signature check is rejecting them, or you never passed Zoom's initial URL validation challenge. Verify against the raw body, not a re-serialized one, and key the HMAC on the Webhook Secret Token, not the Client Secret.
- **Recording download returns nothing:** you need the Bearer token on the download call and you must follow the redirect to the real file URL.

The 70% solution

You will never have perfect information about every edge case before you start, so do not wait for it. Here is the decisive call for most solo and small firms:

Build and provision one S2S app first, scoped read-only to exactly the jobs you are doing, and verify the grant from the token before you write a line of connection code. Then wire up **Zoom**

Phone call logging to your matters first, because that is the highest-value, most-overlooked win. Add **meeting auto-creation** second. Catch `recording.completed` and pull recordings onto the file. Layer in **MCP transcript search** once you have recordings worth searching. Skip the rest until a real need shows up.

That is seventy percent of the value for twenty percent of the effort. Build that, ship it, and refine from real use.

Next: the MyCase-specific subguide, where this all meets a platform with no native Zoom connector and one very specific gotcha that will waste your afternoon if nobody warns you.

Connecting Zoom to MyCase: Build It Direct, and Let Claude Do the Heavy Lifting

MyCase has no native Zoom button. The honest answer is to wire the two APIs together yourself, and that is less work than it sounds once you let Claude map the API and write the build spec for you. Here is the whole path: set up the Zoom side, map the MyCase side, generate the build spec, and let Claude Code build it.

The Zoom side: follow the runbook

Do not rebuild the Zoom app here. Stand up your Server-to-Server OAuth app, scopes and all, using the companion runbook, "Zoom and Your PMS, Hands On." For the basic job of putting a Zoom link on a booked consult, you only need meeting-create scope. Provision it there, collect your four values, and come back.

The MyCase side: a real API with sharp edges

MyCase has a real REST API. It is gated; you request credentials through MyCase rather than self-serving them, and auth is OAuth 2.0 with a refresh token. Once you are in, you can read contacts (the client's email lives on the contact record), cases, and events, and you can write notes onto a matter.

It also has three edges that shape any build: there are no working webhooks, the documented filter parameters are ignored, and there is no server-side search. That points you at a simple poll-cache-diff design: pull the data on a schedule, cache it, diff to find what is new, then act. (A no-code Zapier path exists, but its trigger does not include the client's email, so the invite never reaches the client; that is the main reason to build direct.)

You do not have to take my word for any of it. Have Claude confirm it against your own account.

Map the API: hand this prompt to Claude

Do not map the MyCase API by hand from its docs; the docs and the live API disagree in ways that will cost you a day. Let Claude probe it. Paste this:

```
My MyCase API credentials are set (OAuth 2.0; token endpoint auth.mycase.com,
API base external-integrations.mycase.com). Map my ACTUAL MyCase API surface.
Treat live calls as truth over the docs. Never print secret values.

1. Authenticate (refresh-token grant) and confirm you can call the API.

2. Enumerate the entities I care about and their read endpoints: contacts
   (I specifically need the field carrying the client email), cases, events,
   and calls. For each, fetch one record and show me the shape.

3. Test the documented filter parameters (for example updated_after, case_id,
   status, date ranges on events). Pass a future-dated updated_after and tell
   me whether it actually filters or returns old records anyway.

4. Probe for webhooks and for any search endpoint. Tell me plainly whether
   either exists or returns 404.

5. Confirm the write paths I need: posting a note onto a matter, and whether
   the note field renders HTML or plain text.

6. Output one reference table: Entity or Action | Endpoint | Works? |
   Filterable? | Notes. Then list the constraints that should drive the design.
```

What comes back is your real map: which endpoints work, which filters are decorative, whether webhooks and search exist, the exact path to the client's email, and how to write a note back. That map is the input to the next step.

Write the PRD: hand this prompt to Claude, then give the PRD to Claude Code

With the map in hand, do not free-hand the build. Have Claude write a build-ready PRD first, then hand that PRD to Claude Code to implement. A good PRD here means zero open questions; every decision is settled before a line is written, so Claude Code never stops to ask. Paste this, and if you have a PRD skill, tell Claude to use it:

```
Using a strict PRD discipline (or your PRD skill if you have one), write a
build-ready PRD for a MyCase-to-Zoom integration, using the API map you just
produced as the authoritative source.

The integration: on a schedule, pull MyCase events and contacts, cache them,
diff to find new events that need a Zoom meeting (by a rule I control, such as
an event type or a title keyword), resolve the linked client and read their
email from the contact record, create the Zoom meeting via the Zoom REST API,
write the join link back onto the matter as a note, and send the client the
invite.

Requirements for the PRD itself:
- Zero open questions and zero stop-gates; pre-resolve every decision with me
  here in chat before you finalize it.
- A recon gate that verifies the live API and schema before any build starts.
- Requirements with acceptance criteria written as given / when / then.
- Build stages, each ending in an oppositional QC pass against its criteria and
  an approval gate; the build agent does not self-certify done.
- Named authoritative sources and a conflict-resolution order (live system wins
  over docs).
- Output as a single markdown document I can hand directly to Claude Code.

Ask me every unresolved question now, before you write it, so the finished PRD
has none left.
```

Then open Claude Code, point it at the PRD, and let it build stage by stage. You review at each gate; it does not certify itself done.

Bottom line

No native button, but a gated API you can read the client's email from and write notes back onto. Map it with the first prompt, spec it with the second, and let Claude Code build it from the PRD. A weekend, fully yours, no monthly broker in the middle.

Coming July 7

The MyCase API map and the PRD writing skill behind this guide both ship with standard membership when Practical Legal Tech goes live July 7. Everything here works on your own with the prompts above; membership just hands you the maintained versions, the map kept current as MyCase shifts the API under you, and the PRD skill that turns a fresh map into a build-ready spec for Claude Code.

Sources and what to verify

MyCase API behavior changes, which is the whole point of the mapping prompt; run it against your own credentials rather than trusting any writeup, including this one. The auth and base

values (auth.mycase.com, external-integrations.mycase.com, posting a note to a matter as HTML) and the constraints (no working webhooks, ignored filter parameters, no server-side search, contact records carrying the client email) reflect direct API testing; reconfirm them live.

Zoom and Your PMS, Hands On: Build the App, Map the Endpoints, Run the PRD

The first two guides gave you the architecture and the MyCase reality. This one is the runbook: what to gather, the exact steps to stand up the app, the prompt that maps your live API surface for you, and a real PRD you can adapt. Quick and direct. Do this, then this, then this.

Step 1: Assemble the materials

Before you click anything, get these in one place.

- **The Zoom plugin, loaded into Claude.** It hands Claude the Zoom mechanics: Server-to-Server OAuth, webhook signature verification, the Smart Embed postMessage contracts. One caveat I learned mid-build: the plugin is right about generic Zoom mechanics and wrong about your specific account's quirks. Treat it as the manual, then verify everything against your own live account, because plans differ and some endpoints that the docs promise simply are not there on yours.
- **A Zoom account with the right license and admin or owner access.** Zoom Phone needs a Phone license; AI Companion summaries need AI Companion; several scopes are owner-only to grant. If you are not the account owner, line that person up now.
- **A home for secrets.** An environment file or a secrets manager. Not in your code, not pasted into chat.
- **The four values you will collect after the app exists (Step 2):**
- Account ID (a key)
- Client ID (a key)
- Client Secret (a secret)
- Webhook Secret Token (a second, separate secret, needed only if you subscribe to events)

The keys identify your app; the two secrets prove it is you. The Webhook Secret Token trips people because it is distinct from the Client Secret and is used only for validating and verifying webhooks.

Step 2: Set up the app

Server-to-Server OAuth, because you are automating your own firm's account. Do these in order.

1. **Create it.** Zoom App Marketplace, Develop, Build App, choose Server-to-Server OAuth. Give it a name.
2. **Grab the credentials.** On the Credentials page, copy the Account ID, Client ID, and Client Secret into your secrets store.

3. **Fill the Information page.** Company name, developer contact, the required boilerplate. Zoom will not let you activate until these are filled. Two minutes.
4. **Add scopes, read-only, only what the job needs.** For a firm the useful set is: `phone:read:call_log:admin` (the critical call-capture scope), `phone:read:call_recording:admin`, `phone:read:recording_transcript:admin`, `phone:read:voicemail:admin`, `phone:read:user:admin`, then `meeting:read:meeting:admin`, `cloud_recording:read:meeting_transcript:admin`, `meeting:read:summary:admin`, and `user:read:list_users:admin` plus `user:read:user:admin`. The picker shows old coarse names; the token returns granular ones. You will confirm the real grant from the token in Step 3, not from this screen.
5. **Set up events, but build the receiver first.** If you want Zoom to push you call and recording events, stand up your webhook endpoint and get it answering the URL validation challenge before you add the subscription. Then add an Event Subscription, pick Webhook or WebSocket, select your events, and copy the Webhook Secret Token. Subscribe before your endpoint can answer and the save fails.
6. **Activate.** Server-to-Server apps activate on your own account. No Marketplace review for internal use.
7. **Collect your host user IDs.** An app is the account, not a person, so you must pass real user IDs. Pull `GET /v2/users` and, for Zoom Phone, `GET /v2/phone/users` for each staff member. The meeting user ID and the phone user ID are different values; store both, mapped to your staff.
8. **Softphone only: it is a separate app.** The in-app softphone (Smart Embed) is not configured inside your app; it is a distinct prebuilt Marketplace app, "Zoom Phone Smart Embed." If you want in-app dialing, install that app account-wide, add your portal's domain to its approved list, and enable the account setting "Automatically Call From Third Party Apps." Call logging and voicemail capture do not need any of this; only in-app dialing does.

That is the whole setup. Most of the value (call logging, recordings, transcripts) is live after step 7.

Step 3: Map your endpoints (hand this prompt to Claude)

Once the app is active, do not hand-copy the API from the docs. Probe the live grant and let Claude build the map against what your account actually has. Paste this:

```
You have the Zoom plugin loaded. My Server-to-Server OAuth app is live.
Map my ACTUAL Zoom API surface. Treat live probes as truth over any doc or
plugin assumption. Never print secret values.

1. Fetch a token and print only its granted scopes, one per line, sorted:
   curl -s -u "$ZOOM_CLIENT_ID:$ZOOM_CLIENT_SECRET" -X POST \
     "https://zoom.us/oauth/token?
grant_type=account_credentials&account_id=$ZOOM_ACCOUNT_ID"
   (read the space-delimited `scope` field from the response)

2. For each capability I want, give the exact REST endpoint (method + path
   against https://api.zoom.us/v2) and the granular ":admin" scope it needs.
   My capabilities: log phone calls; fetch call recordings and transcripts;
   read voicemails; create meetings; pull meeting recordings, transcripts,
   and AI summaries; list users.

3. Cross-check each endpoint: granted or NOT granted, by comparing its
   required scope against the printed grant from step 1. Flag every case
   where the Marketplace picker shows a coarse scope name but the grant
   uses a granular one.

4. Probe the read-only ones live and tell me which actually return data
   versus 404 on my account. Call out any endpoints Zoom has sunset.

5. Output one table: Capability | Endpoint | Required scope | Granted | Notes.
   Then list the gotchas you found for my account.
```

You get back an endpoint-to-scope-to-granted table (your own scope phonebook), the picker-versus-token mismatches called out, and the dead ends specific to your plan. Regenerate it after any scope change; the map and your account cannot be allowed to drift.

Before you build a line: write the PRD

Zoom's docs and any plugin describe the platform, not your account. We learned mid-build that the server-side callout endpoints return 404 on our plan, and that the softphone was a separate prebuilt app rather than config inside ours. A PRD with a recon gate catches that kind of thing before you burn a build stage on an assumption.

Here is ours, sanitized, as a template. Names, hostnames, ports, database identifiers, and staff details are stripped; the engineering substance and the hard-won corrections are intact. Adapt it to your stack.

Appendix: Sanitized PRD (adapt freely)

PRD: Practice Management System + Zoom Phone Integration

Status: Approved; in build. Schema, OAuth client, and webhook receiver shipped and live-verified. Corrected against the live Zoom API after the first stages shipped. **Author:** Staff **Target:** the firm's staff portal (the sole practice-management surface).

Revision note

The integration was corrected against the live Zoom API and account after the first stages shipped. Reason: both the generic Zoom plugin and an internal reference assumed a server-side click-to-call endpoint and an in-app softphone configured inside the main app; neither held on this account. Live probing showed the server callout paths return 404 (no server callout exists; the granted write scopes are config-only and none places a call), and the softphone turned out to be a separate prebuilt Marketplace app, not config inside the main app. The event, click-to-call, and softphone requirements, the build order, and the open questions were rewritten to match what was verified live. Where any reference conflicts with this PRD or the live system, the live system wins.

1. Problem statement

The staff portal already reads Zoom call and SMS tables to drive the home dashboard, the matter call log, and a sidebar phone panel, but nothing writes to those tables. The schema describes a complete call-primacy comms design (inbound webhook inbox, click-to-call intents, calls, SMS, meetings, participants, number classification), yet the entire producer side is missing: no Marketplace app, no credentials, no webhook receiver, no processor, no client library. The consumers return zero rows on every load. Until this is built, the firm has no call logging, no voicemail triage, no click-to-call, and the sidebar panel is a static mockup that misrepresents presence. Phone is the firm's primary client-contact channel, so a logged, matter-linked, dispositionable call record is load-bearing for billing capture and conflicts hygiene, not a convenience.

2. Goals

1. Every inbound and outbound firm call lands in the calls table, matched to a contact and matter where a match exists, within seconds of call end.
2. Voicemails and missed calls surface as actionable items on the home dashboard and clear when a staff member dispositions them.
3. Staff can place a call from a contact or matter record (click-to-call) and have the resulting call reconcile back to the originating intent.
4. Staff can run their phone from inside the portal via an embedded softphone, replacing the static panel with live presence and controls.

5. The ORM schema matches the live database across all comms tables, ending the current drift.

3. Non-goals

1. **SMS send and receive.** Deferred. Carrier registration (A2P/10DLC) is unsettled, so no SMS producer or UI ships here; the read path degrades gracefully to zero.
2. **Replacing auth or session handling.** Zoom auth is a server-side Server-to-Server credential, separate from staff login.
3. **Consolidating other integrations' webhook homes.** A separate cutover; this PRD only opens the Zoom webhook path.
4. **Contact Center, Virtual Agent, or IVR.** Not part of the firm's plan.
5. **Migrating historical call data.** Logging starts forward from go-live.

4. User stories

1. As the attorney, I want every inbound call automatically logged against the right matter so I can bill the time and see contact history without manual entry.
2. As the office manager, I want missed calls and voicemails as a to-do list with transcripts so I can triage who needs a callback.
3. As a staff member, I want to link a call that did not auto-match to the right contact and matter so the record is complete.
4. As a staff member, I want to click a phone number on a record and have my Zoom phone place the call so I do not retype numbers or lose the matter link.
5. As a staff member, I want a softphone inside the portal showing real presence and letting me answer, hold, and transfer so I do not switch apps mid-task.
6. As the attorney, I want a call's recording and transcript filed into the matter so the record is complete.
7. As the office manager, I want an inbound number I classified as spam to auto-dismiss next time so the triage list stays clean. (P1)
8. As the attorney, I want scheduled video consults logged with their recording, transcript, and summary against the matter. (P1)

5. Authoritative sources and conflict resolution

Named sources, in precedence order; where two disagree, the higher wins.

1. **The live database schema** is the source of truth for table and column shape. Reconcile the ORM to the database, never the reverse. Adding a column is a schema-design decision: the build agent stops and asks.
2. **A firm-tested Zoom reference** for auth, token fetch (a client basic-auth option, no hand-rolled base64), webhook signature verification, the validation challenge, REST endpoints, and the granted-scope list. The generic Zoom plugin is useful for mechanics but is not authoritative for

this account: live probing disproved its click-to-call and in-app softphone assumptions. Where any reference conflicts with the live system or this PRD's verified sections, the live system wins.

3. **The consumer code** is the read contract: the home dashboard query, the matter call list, and the sidebar panel to replace. Match what they already expect.
4. **The ingress configuration** is the source of truth for routing; the public webhook carve-out is a deliberate exception to an otherwise locked-down surface.

6. Discovered system facts (recon inputs, verify in the gate)

- The app runs behind a reverse proxy that terminates TLS and enforces an IP allowlist, then denies all. A single public carve-out for the webhook path is the only externally reachable route, confirmed reachable from an external IP while every other path returns 403.
- **Provisioning is done.** The Server-to-Server OAuth app exists; Account ID, Client ID, Client Secret, and Webhook Secret Token are wired into the app's environment. The user-mapping table is seeded for the in-scope staff, each row carrying both the meeting user ID and the phone user ID. Granted capture scopes: call log, call recording, recording transcript, voicemail, phone user, the SMS family, meeting read, cloud recording, and AI Companion archives. The event subscription is registered and validation-passed against the live receiver and proven end to end with a live call. Not granted: the list-call-logs scope needed only for historical backfill. The write scopes present are config-only (call queue, device, site, call handling); none places a call.
- The matcher path is: external number in E.164 to the phone-numbers table to the matter-party link to the matter. Contact numbers live in their own table, not on the contact row.
- Recording and voicemail audio stay on Zoom Cloud (fetch on demand); voicemail transcript text is stored on the call row. Binary document filing into the documents store is deferred pending a storage-layout decision; when filing resumes, key folders by the matter's stable ID, not a display number.

7. Standing constraints

These govern the build; the build agent reads the cited standards and does not re-derive them.

- PRD discipline, a recon gate, a separate reporter step, and per-stage oppositional QC. The ceremony does not scale down for a small change.
- The build agent does not default on schema design, auth and authorization patterns, audit and immutability behavior, or anything creating a new public surface. The webhook carve-out is a new public surface and was authorized explicitly in this PRD.
- Commit and push at session end with explicit paths; no blanket staging.
- House writing style for any generated copy or comments: no em dashes, no double hyphens, semicolons fine.

- Clarifying questions are asked inline in the build log, never as popups.
- Prefer the established tooling over ad-hoc shell where a tool fits.
- No base64 for the token; use a client basic-auth option.

8. Requirements

Must-have (P0)

P0-1. Schema reconciliation. The ORM models every live comms table with columns matching the database exactly; no live columns added or dropped here. Verified by a column diff with zero mismatches. Existing consumer reads still return without error.

P0-2. Server-to-Server OAuth and Phone REST client. A library fetches and caches an account-level access token and exposes typed Phone REST calls. No outbound-dial method (no such endpoint exists; see P0-7). The token call uses a client basic-auth option, not a hand-rolled base64 step. Secrets are read from environment only, never logged, never committed; the token is cached and refreshed before expiry.

P0-3. Inbound webhook receiver. The webhook route handles the validation challenge, verifies the signature over the raw request body, inserts the raw event into the inbox with a verified flag and a pending status, and returns 200 within Zoom's timeout. A failed signature is dismissed and not processed. A replayed or out-of-window timestamp does not pass.

P0-4. Ingress carve-out for the webhook (authorized public surface). A single public location for the webhook proxying to the app; everything else still denied. Verified that only the webhook path reaches the app from an external IP.

P0-5. Webhook processor. A dedicated background worker drains the inbox (claimed with row-level skip-locked), parses the subscribed events into call rows, matches the external number via the phone-numbers to matter-party to matter path, writes the live columns, and advances each inbox row to done or failed with a retry count. Subscribed events, validated live: caller call-log completed, callee call-log completed, voicemail received, recording completed. There is no recording-transcript-completed event; pull the transcript via API when a recording exists. The pre-sunset call-log endpoints are retired; historical backfill must use the call-history endpoint, which needs a scope that is not granted, so do not ship against the sunset endpoint. Ambiguity rule: a single number can map to multiple contacts and multiple matters (verified live). On any ambiguity, do not auto-pick; match what is unambiguous, leave the rest null, and route to manual association. Idempotency: dedupe on event plus event timestamp plus object ID.

P0-6. Voicemail and missed-call disposition UI. The home dashboard list becomes actionable: each item shows caller, matched contact and matter, timestamp, and transcript, with a disposition action that records who dispositioned it and when, and removes it from the pending list.

P0-7. Click-to-call (rides on the softphone; no server endpoint). There is no server-side callout API on this account; the relevant paths all return 404. The write scopes present are config-

only; none places a call, and there is no callout scope to add. Click-to-call therefore rides on the softphone: a call control writes an outbound-intent row for pre-attribution, then dials by posting a make-call message to the softphone iframe. The resulting outbound call-log event reconciles back via the intent ID, marking the call as initiated from the PMS. Requires the account setting that allows calling from third-party apps. Depends on P0-8.

P0-8. Softphone (separate prebuilt app). The softphone is not configured inside the main app; it is a distinct prebuilt Marketplace app, installed account-wide with the portal's domain on its approved list and the third-party-calling account setting enabled. The build embeds the softphone iframe in the sidebar, with the postMessage origin pinned to the Zoom applications origin. Staff sign into the softphone with their own Zoom credentials inside the iframe; no server credential is involved. Handle the init, make-call, ringing, connected, and ended messages, answer the contact-match message by resolving the number against the phone-numbers table, and ignore any postMessage from another origin. Capture (logging, voicemail, recording) does not depend on the softphone; it runs off the server webhook path. The softphone adds in-app dialing, call control, and click-to-call only.

P0-9. User and presence sync. The mapping table is seeded for the in-scope staff. This requirement is the maintenance path: when a staff member is added, they gain a row with both user IDs and email, linked to the staff record, without clobbering existing rows.

P0-10. Manual call association. Any call, auto-matched or not, can be linked or re-linked to a contact and a matter from both the triage list and the matter surface. This resolves the ambiguous-match cases from P0-5. A re-link overwrites the prior association and records the change (actor, call, old and new matter).

P0-11. Matter-detail call log. The matter page renders that matter's calls, newest first, each row showing direction, date and time, the other party's number, the matched contact name, result, duration, links to any recording, transcript, or voicemail, and disposition status. A matter with no calls shows a clean empty state, not an error.

Nice-to-have (P1)

P1-1. Video meeting logging. Process meeting and recording events into the meetings and participants tables: topic, host, scheduled and actual times, duration, type, linked matter and appointment, recording and transcript documents, and a summary note. Participants resolve to staff or contacts where matchable.

P1-2. AI call and meeting summary note. Generate a summary note from the transcript and link it from the call and meeting rows. The post-call summary scope is not exposed for server apps, but the recording transcript carries the text.

P1-3. Inbound number classification triage. A classification table drives triage: a staff member classifies a number (spam, known vendor, client), optionally auto-dismiss, and future inbound from that number applies the classification.

Future (P2)

P2-1. SMS send and receive, blocked on carrier registration. Design the matcher and inbox processor generic across event families so SMS slots in without rework.

9. Build stages

Each stage ends with an oppositional QC review against that stage's acceptance criteria, a QC report, and an owner approval gate. The build agent does not self-certify done.

- Stage 0, recon gate: done.
- Stage 1: schema reconciliation, OAuth and Phone REST client, user sync. Done.
- Stage 2: webhook receiver, ingress carve-out. Done and live-validated.
- Stage 3: the processor (background worker), match and transcript text; binary filing deferred.
- Stage 4: home disposition UI, manual association, matter-detail call log.
- Stage 5: softphone (prerequisites done).
- Stage 6: click-to-call, a thin layer on the softphone. Reordered after the softphone because it depends on it.
- Stage 7 (P1): meetings and participants, number classification, AI summary where the path is ready.

10. Success metrics

Leading (first 30 days): - Call capture rate (share of Zoom calls producing a call row). Target 99 percent. - Auto-match rate for numbers already on file. Target 80 percent. - Voicemail triage latency, median. Target under one business day. - Click-to-call adoption (share of outbound calls started from the PMS). Target 50 percent within 30 days.

Lagging: - Billable time captured from calls trends up versus the pre-integration baseline. - Fewer manual "who called this client" lookups.

11. Open questions

Resolved: the OAuth app and credentials; the capture scopes; which staff are in scope; the event subscription and ingress carve-out; the softphone prerequisites; and the click-to-call write scope (void, because there is no server callout endpoint; click-to-call rides on the softphone).

Open, non-blocking: historical backfill needs one scope that is not granted and can wait; AI summary routing; recording retention versus pull-to-archive and binary filing, both gated on the storage-layout decision; and optional cleanup of the unused config-only write scopes. Note any Marketplace access windows your account is subject to, since adding scopes later may have to wait for one.

Sanitized template. Where a generic reference conflicts with your live account, the live account wins. Probe before you trust.