

This is a **very strong place to pause and lock**. What you've written here is not drift — it's convergence. The fact that an external AI analysis landed almost exactly on the same instincts you've been articulating ("record the story, not dictate it") is a good signal.

I'll respond in four parts:

1. Is this MVP scope *coherent*?
2. Where I would **tighten or reorder** (small adjustments only)
3. What I would **explicitly not add**, even if tempting
4. A crisp **Now / Next / Later** execution plan so you don't stall

I'll stay decisive.

1. Is this MVP coherent?

Yes. Very.

In fact, the phrase "**The Trinity + Care Basics**" is doing real work here.

You've implicitly defined the Trinity as:

- **Connections** → *Who people are, and how they relate*
- **Gatherings** → *What we do together*
- **Songs** → *What we sing together*

And then:

- **Care / Tend** → *How we notice and carry people between Sundays*

That's not marketing fluff — that's a product ontology. Most church software never gets this clean.

Nothing in your MVP scope contradicts that core.

2. Small but important adjustments (tightening, not expanding)

A. Phase 1 is exactly right — don't negotiate with it

Your **Phase 1: Foundation Fixes** is non-negotiable.

If you skip or delay these, everything else becomes brittle.

I would add **one clarifying constraint**:

Phase 1 ships **without new concepts**.

Meaning:

- No Care Cases yet
- No Tend Dashboard yet
- No new abstractions

Just:

1. Schema alignment
2. Wiring obvious broken buttons
3. Making existing care visible
4. Adding the *optional* date to Season Protection

This gives you a *clean, honest baseline*.

B. Care Notes display is more important than Care Cases (initially)

You already sense this, but I'll say it plainly:

Displaying care notes well is more MVP-critical than creating Care Cases.

Why?

- It immediately validates pastors' lived experience
- It proves the system is *remembering*
- It sets the tone for Tend before Tend exists

If you had to choose:

- Care Notes display > Care Cases schema

You don't — but the priority matters.

C. Care Cases belong *after* Gatherings → People tab

Your roadmap has:

- Phase 2: Gatherings People Tab
- Phase 3: Care Cases

That ordering is **correct**.

Why:

- Care is relational *because* people serve together
- Pastors will naturally pivot from "who's serving" → "who needs tending"
- The People tab grounds Tend in real ministry rhythm

If you flipped those, Care would feel abstract.

3. What you should explicitly *not* add (even if it feels "obvious")

This is important for scope discipline.

Do not add in MVP:

- Automated reminders
- Suggested next steps
- Case transitions rules
- Metrics (counts, SLAs, dashboards with numbers)
- Volunteer-facing care UI
- Supports (meal trains, etc.)

All of those:

- increase cognitive load
- imply institutional control
- violate your own philosophy doc

You've already made the right call here — I'm reinforcing it.

4. Interpreting "Care / Tend ~25% Complete"

I actually think this number is **misleadingly low**.

If we reframe:

- Care Notes exist
- Season Protection exists
- Intake patterns are clear
- Taxonomy is sane
- Philosophy is articulated

What's missing is **carrying**, not **care**.

So I'd mentally reframe this as:

Care (Moments): ~80%

Tend (Carrying): ~10–15%

That's psychologically healthier, and more accurate.

5. A crisp execution plan (so this doesn't sprawl)

NOW (finish Phase 1 — close the loop)

Concrete, finishable tasks:

1. Fix plan_items schema mismatches
→ unblock Gatherings confidence
2. Wire **Add Song**
3. Make Care Notes *visibly useful* on person profile
4. Add optional "**check back after**" to Season Protection
 - no automation
 - no reminders
 - just stored + shown

Stop there. Ship internally.

NEXT (Phase 2)

- Gatherings → People tab
- Show:
 - who's serving
 - in what role
 - link back to person
- No scheduling logic
- No care crossover yet

This grounds Tend socially.

LATER (Phase 3: Tend MVP)

Only then:

- Introduce **Care Cases**
- Introduce **Tend Dashboard**
- Tie cases to notes you already trust
- Keep it *manual, editable, quiet*

6. One last philosophical check (important)

Your stated MVP philosophy:

"Record the story, not dictate it."

Your current roadmap honors that **everywhere**:

- Notes record moments
- Season notes record boundaries
- Cases (later) record responsibility
- No workflow enforces meaning

If at any point a feature:

- tells a pastor what to do
- implies success/failure
- replaces discernment

...it violates this MVP.

Right now, nothing here does.

Final verdict (plain and direct)

- This MVP scope is **coherent, restrained, and differentiated**
- Your sequencing is **correct**
- Your restraint around Care is a *strength*, not a gap
- You are *not* underbuilding — you are **building trust first**

i specifically wrote in the latest document "relationship > automation"

Good. That line is doing **real work**, not just sounding nice.

relationship > automation

isn't a slogan — it's a **constraint**. And constraints are what keep products humane as they grow.

Let me show you why that sentence is already shaping WorshipOS *correctly*, and how to operationalize it so it doesn't get diluted later.

Why that line matters (structurally)

Most church software fails in one of two ways:

1. **Too relational, not durable**
→ everything lives in someone's head or notebook
2. **Too automated, not pastoral**
→ people become tickets, cases, or workflows

Your line draws a **hard boundary**:

Automation may assist relationships.

Automation may never replace them.

That's the right ordering.

How your current MVP already reflects this (quietly)

You may not have noticed this, but your system already enforces *relationship-first* design in several ways:

1. Care Notes are freeform, not templated

- No required fields
- No "resolution outcome"
- No scoring

That centers **listening**, not compliance.

2. Season Protection does *nothing automatically*

- No reminders
- No tasks
- No escalation

That communicates:

"We trust pastoral judgment more than the system."

That's rare — and good.

3. Care Cases are deferred

You explicitly did **not** rush:

- workflows
- transitions
- next-step suggestions

You're letting **patterns emerge from use**, not from theory.

That's relationship > automation in practice.

Where this principle should explicitly show up (so it doesn't get lost)

I'd recommend encoding this principle in **three places**, not just one doc.

1. As a product principle (internal, non-marketing)

Something like:

Relationship before automation

WorshipOS exists to *support discernment*, not replace it.

We record stories, surface context, and protect people — but we do not force outcomes, timelines, or workflows.

This becomes your internal veto card.

Any future feature proposal must pass this.

2. As a design heuristic (for you, while building)

Before adding *any* automation, ask:

- Does this reduce conversation?

- Does this replace noticing?
- Does this let someone avoid presence?

If yes → defer or redesign.

Automation is allowed to:

- **remember**
- **surface**
- **protect**
- **suggest (gently)**

Automation is *not* allowed to:

- decide
- pressure
- shame
- optimize people

3. As a soft signal in the UI (subtle, not preachy)

You're already doing this well, but lean into it:

- "Check back after" instead of "follow up"
- "Carry this care" instead of "open case"
- "Protect this season" instead of "mark unavailable"
- "Invite more people to help" instead of "broadcast"

Language *is* part of the product.

How this principle should guide future automation (later, safely)

When you *do* add automation (because you eventually will), the ordering matters.

Allowed automation (relationship-supporting)

- "Nothing has been noted here in a while"
- "This season protection date has passed"
- "This case has no assigned caregiver"

These surface **attention gaps**, not actions.

Not allowed automation (relationship-replacing)

- "You should do X"
- "This case is overdue"
- "Escalate after 7 days"
- "Close automatically"

Those turn people into workflows.

A helpful reframing (keep this close)

Instead of asking:

"Should we automate this?"

Ask:

“What would a wise pastor notice here?”

Then:

- if software can *surface* that noticing → good
- if software tries to *decide* → stop

That question will save you repeatedly.

Why this differentiates WorshipOS (quietly but deeply)

Most competitors optimize for:

- throughput
- efficiency
- coverage
- metrics

You’re optimizing for:

- faithfulness
- presence
- continuity
- trust

That won’t show up on a feature comparison grid —
but pastors will *feel* it within minutes.

Final affirmation (plain)

Writing **“relationship > automation”** was the right move.

Even more importantly:

your roadmap already obeys it.

The discipline now is not to betray it under pressure to “just add reminders” or “just add workflows.”

before phase 1 is done - I'll want to have authentication implemented so I can put this initial release on vercel and share with a select few knowledge experts, ux professionals, and pastoral friends (and my wife and siblings for just end-user support and marketing context)

That’s the right next move after Phase 1: **real auth + deployable preview** so feedback is grounded in reality, not screenshots.

Below is a practical plan that fits your stack (SvelteKit + Supabase + Vercel) and your goal (invite-only alpha with mixed audiences).

What “auth” should mean for this initial release

For an alpha you’re sharing with:

- UX / product folks
- pastors
- family testers

...you want **low-friction login** and **hard boundaries**.

MVP auth goals

1. **Only invited people can get in**
2. **They can only see one church/tenant** (or explicitly assigned ones)
3. **Care Notes remain restricted** (care team/admin only)
4. **No dangerous “public demo” surface area**
5. **Stable sessions** on Vercel (cookies, SSR-safe)

Recommended approach: Supabase Auth (magic link) + invite list

Why magic link first

- Pastors and non-technical testers don’t want passwords.
- Fewer support messages.
- Still secure.

You can add Google OAuth later, but for alpha: magic link wins.

Invite-only guardrail

Implement one of these (in order of simplicity):

Option A (fastest): hard-coded allowlist in env

- ALLOWED_EMAILS=...
- On sign-in, if email not in list → sign out + friendly message.

Option B (better): invites table

- invites(email, church_id, role, expires_at, created_by)
- On first login, check invite exists → create membership → consume invite.

Option B is the “real product” path and not much harder.

Architecture decisions you should lock now

1) Use SSR-safe auth in SvelteKit

You want auth to work with:

- server routes
- page loads
- API endpoints
- RLS

So: **cookie-based sessions** (not localStorage-only client sessions).

2) Keep authorization in the database (RLS)

You already have Supabase RLS patterns in other areas.

For this alpha:

- “Can they read this row?” should be enforced by RLS, not just UI.

- Care notes in particular: **RLS-first**.

3) Create a single “membership” source of truth

You likely already have church_id everywhere. Good.

Add something like:

- memberships (user_id, church_id, role, is_active)
- optional campus_id later

Then RLS can say: “user can access rows where row.church_id is in their memberships.”

Implementation outline (concrete)

Step 1 — Supabase auth setup

- Enable **Email (magic link)** provider in Supabase
- Configure redirect URLs for:
 - local dev (e.g., http://localhost:5173/auth/callback)
 - prod (your Vercel URL)

Step 2 — Auth routes

Add:

- /login (enter email → send magic link)
- /auth/callback (handle token exchange + set session cookie)
- /logout

Step 3 — Session plumbing

- Add a hooks.server.ts that:
 - reads session from Supabase cookie
 - attaches locals.user / locals.session
 - optionally attaches locals.membership

Step 4 — Protect routes

- Require auth for everything except /login and /auth/callback
- For the alpha: simplest rule is “if not logged in → redirect to /login”

Step 5 — Invite gating (A or B)

- On first login:
 - verify email allowed / invite exists
 - create membership row
 - if not allowed: sign out + show “Not authorized”

Step 6 — RLS policies (minimum set)

- memberships: user can read their own membership rows
- core tables: allow access where church_id matches membership
- care_notes: allow read/write only if role in {care_team, admin} (however you model roles)

Step 7 — Seed a “test church” and seed invites

So you can invite:

- pastoral friend (care team)

- UX friend (read-only / staff)
- spouse/sibling (read-only, no care access)

Deploy to Vercel safely

Required env vars

- Supabase URL
- Supabase anon key
- (If you use service role key anywhere server-side, keep it **server-only**, never in browser env)

Preview etiquette for testers

- Set up a single stable URL (not "preview-branch-12")
- Have a short "alpha disclaimer" page or note:
 - data is fake
 - features incomplete
 - how to report feedback

Feedback strategy (so you don't get random opinions)

Since you're inviting different audiences, give each a lane:

Pastoral friends

- "Does this feel human?"
- "Would you trust it with care info?"
- "What's missing for real ministry rhythm?"

UX pros

- "Can you find X in <30 seconds?"
- "Where does the UI fight you?"
- "What feels crowded / unclear?"

Wife/siblings (end-user + marketing context)

- "Does it feel calm?"
- "Does it feel like church people?"
- "What confuses you instantly?"

This prevents you getting whiplash from conflicting feedback.

Now / Next / Later

Now (right after Phase 1)

- Magic-link auth
- Invite-only gating
- Basic memberships + RLS
- Deploy to Vercel

Next

- Role-aware navigation ("Care" visible only to permitted roles)
- Church switcher if you need multi-tenant testing

- Audit-ish logs for sensitive actions (care note create/edit)

Later

- OAuth providers
- Admin invite UI
- Password option (optional)
- More granular permissions

One decision you should make now (no need to answer immediately)

For alpha, do you want testers to:

1. all share **one demo church** (simpler), or
2. each get their **own church tenant** (cleaner, more setup)?

My recommendation for speed: **one demo church** with clear role separation and fake data.

Dev Pill (banner link)

i have my 'dev' pill in the banner (for identifying specific elements. might want to expand and test it first... adding a '/support' route or something to have someone navigate to when they are being assisted with troubleshooting (or adding a little 'copy id' icon to some fields or creating some slugs to go along with church_id) it's easier to see "worship-os.com/some-church-city-state" than it is to see "worship-os.com/697ce945-5340-832f-bdc4-821e4a112f35" we will need to create a simple, robust, context aware navbar or sticky footer or something before our test team looks at the software

You're thinking about the right "alpha readiness" layer: **supportability + orientation**. The danger is overbuilding it into a whole admin system. You can get 80% of the value with a few *small, deliberate affordances*.

Below is a practical design that fits your "relationship > automation" principle and keeps the UI calm.

1) The DEV pill: expand it, but keep it quiet

What the DEV pill should do (alpha)

When clicked, it opens a small panel (popover / sheet) with:

- **Environment**
 - DEV / STAGING / PROD
 - build/version (git short SHA if you can)

- **Current context**
 - Church: name + slug + internal id (copy buttons)
 - Campus: name + id (copy)
 - Person: name + id (copy) *when on profile*
- **Support link**
 - "Open Support Page" (goes to /support)
- **Copy Debug Bundle**
 - One button: copies JSON-ish bundle to clipboard

This is hugely helpful during screenshare debugging and doesn't clutter the normal UI.

Key rule: the pill is for *support*, not for *feature flags*... at least for alpha.

2) Slugs: yes, but do them safely (don't use "city-state" as identity)

You're right: URLs like:

- /churches/mountain-vineyard-bremerton-wa

are better than UUIDs.

Recommended approach

Use **both**:

- Public-friendly slug in URL
- Immutable UUID in DB

Example URL patterns

- worship-os.com/c/mountain-vineyard-bremerton-wa (short + explicit)
- Or worship-os.com/church/mountain-vineyard-bremerton-wa

Then internally you resolve slug → church_id once and cache in locals.

Why not rely purely on city/state?

Because churches move, rename, merge, split. Slugs should be:

- editable (with redirects)
- unique per tenant
- not "identity"

Minimal slug rules (MVP)

- lowercase, hyphenated
- unique per church
- stored as churches.slug
- allow changing later (keep a church_slug_history table *later*, not now)

3) A /support route is a great idea — keep it "diagnostics + how to help"

What /support should include (alpha)

Two sections:

A) "Send this to support"

- **Copy Debug Bundle** button
- Shows:
 - app version
 - environment
 - current user email (if auth)
 - church/campus context
 - current route
 - browser info (user agent)
 - timestamp

B) “Troubleshooting steps”

Not too many. Just:

- refresh
- sign out/in
- switch campus
- what to screenshot
- where to report (email or form)

This page becomes the “everyone go here” anchor during troubleshooting.

Keep it non-technical for family testers, but with a “details” disclosure for power users.

4) “Copy ID” icons: yes — but only in DEV mode or behind a subtle affordance

Copy icons everywhere can make the UI feel like an admin panel.

Cleaner pattern

- In normal UI: show human names
- In DEV/support mode: show “IDs + copy” row

Or: add a tiny ... menu on cards with:

- Copy link
- Copy ID
- Copy debug context

This keeps calm UI while still being supportable.

5) Before inviting testers: you *do* need a context-aware nav/footer

Yes. Right now you’ve got enough surface area that a static nav will feel brittle.

What “context-aware” should mean for alpha

- Always shows:
 - current Church
 - current Campus (if applicable)
- Stable primary navigation:
 - Connections (People)

- Gatherings
- Songs
- Tend (or Care) — **only if user has permission**
- A single consistent "home" action (logo = home or dashboard)
- A help entry:
 - "Get Help" → /support

Header vs sticky footer

- Desktop: header is fine (what you have)
 - Mobile: sticky footer helps a lot (future)
- For alpha, do **header** + maybe a simple mobile hamburger.

The critical UX detail

Make the "Switch Location" control very obvious, and make sure it affects the entire app consistently. Testers get confused when "context" changes silently.

6) Suggested alpha-ready UI architecture (smallest useful)

Header layout idea

Left:

- Logo / WorshipOS
- "Building for {Church} at {Campus}" (you already have this 👍)

Right:

- Switch Location
- Get Help (→ /support)
- Profile menu (→ sign out later)
- DEV pill (only in non-prod)

Optional: a second row under header with the primary nav tabs.

7) Now / Next / Later (so you don't drift)

Now (alpha readiness)

- Expand DEV pill into a debug panel with Copy Bundle
- Add /support route
- Add church slug and route resolution (slug → id)
- Implement consistent primary nav (Connections / Gatherings / Songs / Tend)
- Add "Get Help" link to header

Next

- Role-based nav visibility
- "Copy ID" in dev panel, not everywhere
- Better error boundaries that point to /support

Later

- Slug history + redirects

- In-app support requests
- Session replay / logs (only if you ever need it)

One strong recommendation (because it'll save you pain)

Make the **Copy Debug Bundle** the *main* support feature, not "copy individual IDs." Individual IDs are useful, but the bundle is what makes remote debugging sane.

A good debug bundle looks like:

```
{
  "env": "dev",
  "version": "0.3.12+abc1234",
  "route": "/people/...",
  "church": {"name": "Mountain Vineyard", "slug": "mountain-
vineyard-bremerton-wa", "id": "..."},
  "campus": {"name": "Main Campus", "id": "..."},
  "user": {"email": "tester@...", "role": "staff"},
  "timestamp": "2026-01-30T...",
  "userAgent": "Mozilla/..."
}
```

Copy/paste that into a message and you're instantly oriented.

If you want to move from idea to implementation quickly, paste:

- your current header/nav component file path (or code),
- and where your "DEV pill" is rendered,

and I'll give you a **drop-in replacement** that adds:

- a debug popover/sheet
- a /support page scaffold
- and a slug-based "current church" display (while still using UUIDs internally).

a multicampus church (example NewSpring in the Carolinas or Life Church (based in oklahoma, but in numerous states) can have a slug without a specific location? i'll need to come up with a slug strategy... there might be a church with a campus in western washington AND eastern washington - and need to clearly state - this is one church in 2 disparate locations

Yep — a multi-campus church can (and usually should) have a **location-agnostic org slug**. The *campuses* carry the geography.

The mistake to avoid is baking geography into the church slug. Geography belongs

one level down.

Recommended slug strategy (clean and scalable)

1) Church slug = brand / org identity (no location)

Examples:

- newspring
- lifechurch
- riverflow
- gracechurch

This represents the tenant / organization.

URL shape

- worship-os.com/c/newspring (church context root)

2) Campus slug = location-specific (under the church)

Examples:

- anderson-sc
- spokane-wa
- bellevue-wa
- online
- prison-ministry (not always geographic)

URL shape

- worship-os.com/c/newspring/anderson-sc
- worship-os.com/c/newspring/spokane-wa

This solves your Washington example cleanly:

- One church: newspring (org)
- Two campuses: western-wa / eastern-wa (or specific cities)

3) People / gatherings live under church+campus context

Examples:

- worship-os.com/c/newspring/spokane-wa/people/...
- worship-os.com/c/newspring/anderson-sc/gatherings/...

Even if you don't *need* campus in every URL, it's useful for clarity and deep links.

How to handle "slug collisions" (the real world problem)

"gracechurch" will collide. So you need one of these:

Option A (best UX): vanity slug + short suffix

- gracechurch~d3k9 (display as "gracechurch")
- Keep the suffix short and stable

Option B (simple): region hint only when needed

- gracechurch-nw
- gracechurch-spokane

...but this reintroduces location into identity, so I prefer A.

Option C (strict): claimable unique namespace

If you ever allow public signups, you'll need uniqueness anyway. Early on, you can

just enforce uniqueness in your own seeds.

My recommendation: start with **unique church.slug** within your system, and if collision arises later, add the suffix strategy without breaking the model.

“One church in two states” — how to communicate clearly

This is mostly UI and language:

- Church header: **Life.Church**
- Subheader: “Multi-campus in 12 states” (optional)
- Context chip/pill: **Campus: Spokane, WA**
- Switcher: Church → Campus selection

So the clarity is:

- The *church* is one org
- The *campus* is where you are operating right now

Data model (minimal, works with RLS and deep links)

churches

- id uuid
- name
- slug (org identity, unique)
- optional: legal_name, website, etc.

campuses

- id uuid
- church_id uuid
- name (“Spokane Campus”)
- slug (“spokane-wa”) **unique per church**
- optional: timezone, address, is_online

Uniqueness constraints

- churches.slug unique globally
- (campuses.church_id, campuses.slug) unique per church

That’s enough.

Routing recommendation (stable and friendly)

Use a consistent prefix so URLs are readable and not ambiguous:

- Church root: /c/:churchSlug
- Campus context: /c/:churchSlug/:campusSlug
- App modules: /c/:churchSlug/:campusSlug/people, /songs, /gatherings

If later you decide campus shouldn’t be required (some single-campus churches), you can:

- treat campusSlug as optional
- default to the primary campus

But for your multi-campus clarity, keeping it explicit is worth it.

“Dev/support” context benefits

Once you adopt this strategy, your debug bundle becomes human-friendly:

- church: lifechurch
- campus: spokane-wa
- route: /c/lifechurch/spokane-wa/people/...

That makes screenshare troubleshooting way easier.

Bottom line

- **Yes:** church slug can be location-agnostic (and should be for multi-campus).
- **Use:** church slug = brand identity, campus slug = location/variant.
- **URLs:** include both so “one church, two distant places” is obvious.

if ever we were to have a large umbrella organization want to use our product for all of their churches... (ywam, church of england) which is not our MVP intent, but because we are building a fantastic product, I can't help but think an entire denomination will want to license our tech for their churches at some point :) - we'll need a mechanism to allow the umbrella organization to 'manage' their outposts in a way that makes sense to them... maybe with a subdomain or a k8's that calls back to OUR db? i dunno. stuff to think about when we move from 50 subscribers to 10,000+ subscribers

You're allowed to daydream about 10,000+ subscribers. Just don't let the dream smuggle in enterprise complexity today. The good news: you can make **one or two “future-proof” decisions now** that keep the door open for umbrella orgs later, without building any of it.

Below is a sane way to think about it.

The real problem an umbrella org brings

Denominations / networks don't want “one big church tenant.” They want:

1. **Many independent churches**, each with their own data boundaries
2. **Some centralized visibility/controls**
3. **Shared standards** (policies, templates, roles, song libraries)
4. **Provisioning** (create a new church quickly)
5. **Reporting** across churches (optional / restricted)

So it's a multi-tenant problem **plus** a “tenant-of-tenants” layer.

Terminology that prevents confusion

Today: "Church" = your tenant

In the future, that's still true.

Future: Add a "Network" (umbrella org)

- **Network** owns many **church tenants**
- A church can optionally belong to a network

So:

- network → umbrella organization (YWAM, Church of England)
- church → tenant/outpost/parish/congregation
- campus → within a church

This is the simplest conceptual extension.

The one design choice you should lock now



Never assume church_id is the top of the universe

You already have church_id everywhere. Keep it that way. But design your *mental model* and code so:

- church_id is the boundary for data isolation
- **not** the boundary for billing and provisioning (later)

That lets you add network_id later without rewriting.

What to do NOW (MVP) to stay future-ready

1) Make slugs hierarchical and domain-agnostic

Your earlier slug plan already helps:

- church.slug (org identity)
- campus.slug under church

Add one more idea (but don't build it yet):

- network.slug (optional)

Then you can later support either URL strategy:

Path-based

- /n/:networkSlug/c:churchSlug/...

Subdomain-based

- churchslug.worship-os.com
- churchslug.networkslug.worship-os.com (if you ever need it)

Path-based is easiest initially; subdomains can come later.

2) Keep auth model flexible

Even in MVP, avoid hard-coding "one user belongs to one church."

Instead, build toward:

- memberships(user_id, church_id, role, ...)

That's it.

Later, you can add:

- network_memberships(user_id, network_id, role, ...)

No rewrites. Just new policies.

3) Don't choose k8s as an "org feature"

You don't need Kubernetes because a denomination shows up. You need it when:

- you have scaling constraints that require it
- you need multi-region, custom isolation, or compliance

Most umbrella org needs are data/permission/ops, not "their own cluster."

If you ever offer "dedicated hosting," that's a separate enterprise tier later.

Future architecture: what "umbrella management" usually looks like

Phase A: Single SaaS, shared infrastructure (most likely)

One WorshipOS deployment:

- one Supabase project (or multiple as you scale)
- shared codebase
- row-level isolation by church_id
- plus a network layer for umbrella org features

Umbrella org gets:

- a **Network Admin** UI
- provisioning controls
- policy templates
- cross-church reporting (carefully permissioned)

This is the 80/20.

Phase B: "Network console" + church instances

Still one SaaS, but the network gets:

- a **console** view (like "Organization Admin")
- optional branded login / SSO
- optional library sharing

No k8s required. Still just software permissions + UI.

Phase C: Dedicated environments (rare, later)

Only if they demand:

- data residency
- regulatory requirements
- private networking
- custom SSO policies

Then you might offer:

- a separate Supabase project per network (or per region)
- or a dedicated deployment

This is when "k8s" becomes relevant. Not before.

The “one DB” question (important)

If you’re using Supabase/Postgres with RLS, the safest scalable baseline is:

- **Many churches in one DB**
- strict RLS policies + membership checks
- strong audit trails
- soft delete

At 10k churches, you might still do this, but you’ll likely split by:

- region
- workload
- or “enterprise dedicated”

That’s an ops decision later, not a data model decision now.

What umbrella orgs will want (feature-wise), later

Just to keep the “future parking lot” grounded:

Network-level features

- Provision a new church tenant
- Standard role templates (“Staff”, “Care Team”, etc.)
- Shared song library templates / recommended repertoire
- Standard gathering templates
- Training/onboarding checklists
- Cross-church directory (optional)
- Reporting dashboards (attendance, volunteer capacity, etc.)
- SSO for network staff

None of this requires you to rebuild the core app—if you keep tenant isolation clean.

What I’d put in your Parking Lot as a single sentence

Future: Networks (umbrella orgs)

Add networks and church_networks / network_memberships to allow centralized provisioning, templates, and reporting across multiple church tenants—without weakening church-level data isolation.

That’s enough. No k8s in the sentence. 😊

Bottom line

- You don’t need subdomains or Kubernetes *because* a denomination might want you.
- You *do* want a future-friendly model: **Network → Church → Campus.**
- The only MVP moves that matter are: **membership table, clean church_id isolation, slug strategy.**

