Base44 Platform Limitations: A Field Report

Findings from building a multi-tenant SaaS application on Base44, and what we'd have on a standard backend instead.

Context

Telos is a multi-tenant coaching platform built on Base44. Each coach gets a workspace and owns brands, contacts, appointments, and other private data. The platform also has a public-facing surface — anonymous visitors book appointments through public booking pages and submit testimonials. Private data must never leak between coaches, even when they share the same workspace.

Building this on Base44, we've hit a series of platform-level limitations that wouldn't exist with a standard backend setup. This document catalogs what we found and what the same architecture would look like with Claude Code working against a normal stack.

Limitation 1 — `asServiceRole` doesn't actually bypass RLS

Base44's `asServiceRole` is the platform's pattern for service-role operations — public functions, system jobs, anything that runs without a real authenticated user. Documentation and the SDK's framing imply it bypasses row-level security.

It does not. Confirmed by Base44 support: `asServiceRole` impersonates a synthetic admin user with no `workspace_id`. RLS rules are still evaluated against this synthetic identity. Most platforms give service-role calls a true bypass token. The result is that every public function has to be architected around RLS instead of trusted to act on behalf of the system.

Limitation 2 — No way to distinguish service from human

The synthetic service user presents as `role: admin` — indistinguishable in RLS evaluation from a real human admin. There is no service-role flag, no machine identity, no bypass token visible to the rule engine.

This means a rule that grants the `admin` role any read access also grants every real admin user that access. We cannot write a rule like "service can read everything but humans can only read their own data" because the rule engine cannot tell them apart. For a platform where coaches are `role: admin`, this is a serious privacy constraint — it forces parallel "snapshot" entities for any data that public functions need to see.

Limitation 3 — Chained function calls strip auth context

When one Base44 backend function calls another via `asServiceRole.functions.invoke()`, the inner function receives a raw HTTP request with no forwarded auth headers.

`createClientFromRequest(req)` in the inner function has no app context, and all entity operations fail — including `asServiceRole` ones. We hit this for two days before support clarified that the only fix is to inline the logic.

On most platforms, internal function-to-function calls preserve context automatically. Base44 documents no caveat for this; you discover it by hitting it.

Limitation 4 — RLS template variable gotcha

RLS rules use template variables like `{{user.id}}`. The platform supports built-in fields (`user.id`, `user.email`, `user.role`), but custom User fields require `user.data.<field>` syntax. Using `{{user.workspace_id}}` for our custom `workspace_id` field silently resolves to undefined and breaks every workspace check.

The failure mode is invisible to super_admin testing — super_admin matches the first OR branch of the rule unconditionally and never reaches the workspace check. We lost days of debugging across 14+ entities to this single typo before support flagged it.

Limitation 5 — Deploy reliability

Edits to function source files don't always propagate to the runtime. Deploys can time out or silently fail. During our debugging cycles we frequently saw the source code show one version and the live function execute another, and we had to add explicit version markers (`_v: "v5"`) to the response body just to confirm which code was actually running. We've also had functions disappear from the function registry between calls.

How we work around all of this

For any data a public function needs to read, we build a parallel "snapshot" entity. The snapshot has open read RLS by design, mirrors only public-facing fields from the source, and is populated by hooks when the source record changes. Public functions read from the snapshot; the source entity stays locked down.

It works, but:

- Doubles the data layer for any entity touched by a public flow.
- Creates sync drift risk — the snapshot can fall out of sync with the source if a hook fails.
- Doesn't work for private entities like Contact, where exposing any public-readable mirror would be a privacy disaster. For those, we have to validate scope in function code instead of trusting RLS.
- Adds complexity to every new feature that touches the public surface.

What Claude Code with a standard backend gives you

The same application built with Claude Code against a standard stack — Postgres + Supabase / Neon / Convex, or even a plain Express + Prisma setup — does not hit any of these limits:

- **Service-role keys actually bypass RLS.** Public functions can be trusted to act on behalf of the system without architectural workarounds.
- **Internal function calls preserve auth context.** No "inline everything" mandate.
- **Distinct service vs human identities.** Rules can target one or the other explicitly.
- **Standard SQL templating.** No surprise undefined variables. Errors are loud, not silent.
- **Privacy rules written once, in one place.** They behave the way they read. Snapshot entities aren't necessary.
- **Reliable deploys.** Source equals runtime.

> The work we did over a week on Base44 — building the snapshot pattern, debugging chained invocations, hunting the template variable bug, working around the synthetic admin's missing workspace_id — would have been a one-day feature on a standard stack. The platform's friction grew with the privacy model, not with the feature complexity.

The honest trade-off

Base44 is appealing for the same reason it's limiting: it bundles entity definitions, hosting, deploy, RLS, and a builder UI into one product. For simple apps with few users and simple data scoping, that bundling delivers real speed. The "no-code-ish" builder can take you a long way before you hit anything sharp.

But the further your privacy model has to go — multi-tenant, public-facing surface, real privacy guarantees between users — the more you're fighting the platform instead of using it. Every workaround compounds. Documentation lags behind actual platform behavior. Support is responsive and competent (genuinely — we've had good interactions), but the fundamental constraints are baked in.

Bottom line: If you're building a single-user app, a small team tool, or a prototype, Base44's bundled approach is hard to beat. If you're building a multi-tenant SaaS with a real public surface and real privacy requirements, Claude Code working against a standard backend will ship the same feature in a fraction of the time, with rules that actually do what they say.

Prepared by Ed Ferrigan, founder of Telos. Based on direct experience building the Telos coaching platform on Base44 from 2025 through 2026. All findings cross-referenced against Base44 support correspondence on tickets through May 2026.