ChatGPT

Estimator Assistant MCP – Comprehensive Integration Guide

1. Next.js 15.3.2 on Vercel – Deployment & Runtime Configuration

Current Status (2025): Next.js 15 is a stable, production-ready release focused on stability and performance. It introduced support for React 19 and features like a new async request API and improved caching defaults [1] [2]. By default, Next.js App Router routes run on the Node.js runtime (with full Node API support) unless explicitly configured for Edge runtime [3]. Vercel's platform supports both Node.js (serverless) and Edge runtimes for Next.js functions, each with different constraints. Edge Functions are extremely low-latency but have stricter limits (e.g. ~1–4 MB code size) and no access to Node-specific modules, whereas Node (Serverless) Functions allow up to 50 MB bundles and full Node.js APIs but have higher cold-start latency [4] [5].

Best Practices: For **deployment on Vercel**, ensure your Next.js app is optimized for the platform's constraints and follow a deployment checklist:

- **Memory and Build Optimizations:** Next.js 15 introduced an experimental Webpack memory optimization flag. In your `next.config.js`, set `experimental.webpackMemoryOptimizations: true` to reduce build memory usage (at slight cost of build time) [6]. Regularly analyze your bundle size and dependencies – remove or lazy-load large modules to avoid memory bloat [7]. Use Next.js' **Bundle Analyzer** to identify heavy dependencies [8]. If build OOM issues persist, run `next build --experimental-debug-memory-usage` to get heap snapshots and pinpoint memory leaks [9].

- **Edge vs Node Runtimes:** Use the Node.js runtime for any code that relies on Node APIs (like `fs`, `process`, crypto libraries, etc.). By default, Next.js uses Node runtime for SSR and API routes [3]. Only opt in to the **Edge runtime** for specific routes where ultra-low latency is required and when the code is simple and Node-API-free (e.g. lightweight request handlers, middleware). You can explicitly set the runtime per route or layout with an exported `runtime` variable: `export const runtime = 'edge'` (or `'nodejs'`) [10]. This ensures, for example, that authentication routes using Node libraries run on Node, while trivial personalization routes could run on Edge.

- **App Router Deployment Considerations:** The App Router streams and caches content differently than the old Pages Router. In Next 15, `GET` route handlers are **uncached by default** (changed from Next 14) unless opted in to static caching [11]. Review any Route Handlers and decide if they should be static (`export const dynamic = 'force-static'`) or dynamic. Additionally, consider using Next's **Experiments** (like `authInterrupts`) carefully – for example, `authInterrupts` can simplify protected routes but might have compatibility nuances. Always test these experimental features in staging.

- **Build Performance:** Next.js 15's build uses Turbopack in dev and improved Webpack in production [12]. Keep an eye on warnings during build. Warnings about large chunks or unsupported modules should be addressed (e.g., by dynamic import or polyfills). If your project is very large, consider increasing Vercel's build memory or time limits (e.g., selecting a larger build instance), or splitting the app into multiple smaller apps (micro-frontends) if feasible.

Common Issues & Solutions:

- **Edge runtime errors:** If you see warnings like *"A Node.js API is used (process.platform) which is not supported in the Edge Runtime"* [13], it means some code is being bundled for Edge that shouldn't be. In our case, the Better Auth library triggered this because it uses `process.platform` in a shared util, and the Next.js build attempted to include it in middleware. The solution is to ensure that code only runs on Node runtime. You cannot run Next's **global** `middleware.js` **on Node** (middleware is always Edge), so minimize its use of such libraries. For Better Auth's cookie checking in middleware, the library provides `getCookieCache()` / `getSessionCookie()`, but these currently still rely on Node modules and may return null on Edge [14]. **Solution:** Use a simple presence check on the session cookie string via `request.cookies` in middleware (to avoid Node APIs), or remove the middleware and perform auth checks in Node route handlers where possible. Also, explicitly mark any API routes that use Better Auth or other Node-only libraries with `runtime = 'nodejs'` to force Node execution [10].

- **Serverless function size limits:** Large dependencies (like the AI SDK with multiple providers, or Google APIs) can bloat your serverless function bundles. Vercel's Serverless Functions have a 50 MB compressed size limit [5]. Use Next 15's `outputFileTracing` (enabled by default) to exclude unnecessary files, and consider the `serverExternalPackages` config to keep large packages external (if you use the experimental standalone output) [4]. If hitting size limits, you can also deploy certain heavy tasks as separate serverless functions or edge functions, or switch to Vercel's new **Edge Config/AI Gateway** if applicable for large models.

- **Memory/timeouts on Vercel:** If builds or functions time out, consider optimizing or increasing resources. Use the `memoryUsage()` guide to pinpoint memory hot-spots [15]. For long-running routes, ensure they stream responses (like using incremental streaming for AI responses) so that the 10s execution limit on Vercel Edge or ~60s on Serverless isn't exceeded. In Next 15, you can also leverage the new `unstable_after` API to run non-critical code after streaming the response [16].

Configuration Example: Below is a partial `next.config.js` illustrating some best practices:

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  experimental: {
    webpackMemoryOptimizations: true,  // Reduce memory usage during build [6]
    workerThreads: false,              // If using a custom webpack config,
disable build workers to avoid conflicts
  },
  // Mark certain modules to run on Node.js runtime only
  // (If using App Router, can also set per-route runtime in the file)
  // Example: forcing Node runtime for auth routes (if needed globally)
  // runtime: 'nodejs',  (generally not needed globally – set per route)

  // Optionally, limit which packages are bundled:
  // (e.g., Externalize heavy native deps if not needed in lambda)
  // webpack: (config, { isServer }) => { ... }
};
export default nextConfig;
```

And in an example route file that uses a Node-only library (e.g., `app/api/auth/[...all]/route.ts`):

```
import { auth } from '@/lib/auth';
export const runtime = 'nodejs';  // Ensure this route runs on Node, not Edge [3]

export const { GET, POST } = auth.handlers; // (Better Auth integration example)
```

This configuration ensures that any route under `/api/auth` will use Node.js runtime, preventing Edge-related errors for Better Auth.

Troubleshooting Checklist:

1. **Verify Runtime Settings:** Check which routes (API or dynamic rendering routes) might be inadvertently running on Edge. Look for `runtime` exports or missing ones. Ensure routes needing Node APIs explicitly opt out of Edge.

2. **Monitor Build Warnings/Errors:** Review the Vercel build log for warnings (like unsupported APIs or large bundles). Address each warning:

3. Unsupported Node APIs in Edge → switch runtime or remove usage.

4. Large bundle warnings → enable code-splitting or externalize deps.

5. **Test Locally with Similar Constraints:** Use `next build` locally and run in Node 18/20 to simulate Vercel. If memory issues occur locally, apply the memory optimization flags. For edge functions, use `next dev --turbo` or the Edge runtime simulation if available.

6. **Profile Memory if Needed:** If builds fail due to memory, run `node --max-old-space-size=4096 node_modules/next/dist/bin/next build` to increase memory locally and identify if the limit is hit. Use Next's memory debugging mode for insight [9].

7. **Deployment Testing:** After deploying, use Vercel's function logs and testing tools (like curling API endpoints, visiting pages) to ensure each route responds as expected. Pay attention to any 500 errors – they often indicate runtime mismatches or missing environment variables on Vercel.

By following these practices and checks, your Next.js 15 app should build and run smoothly on Vercel, leveraging the appropriate runtime for each part of the application.

2. Vercel AI SDK v5.0.60 – Multi-Provider AI Integration

Current Status: Vercel's AI SDK (v5) is a unified toolkit for working with multiple AI providers (OpenAI, Anthropic, Google PaLM, xAI Grok, Cohere, OpenRouter, etc.) under a common interface [17] [18]. It standardizes calls through a **Language Model specification**, allowing you to switch providers with minimal code changes [19]. Version 5 introduced improved streaming support, tool usage integration, and a new "Language Model Middleware" concept for intercepting and modifying model interactions [20]. The SDK supports *streaming* responses, *function (tool) calling*, and multi-turn chat out of the box, with a consistent API across providers.

Best Practices: When integrating the AI SDK v5 in a Next.js App Router app:

- **Use the Provided Model Objects:** The SDK exposes helper functions to create provider-specific **Model objects**. For example, use `openai('<model-name>')` from `@ai-sdk/openai` or `anthropic('<model>')` from `@ai-sdk/anthropic` to get a model instance. This is preferred over manually constructing plain JS objects for models. For instance, `model: openai('gpt-4o')` ensures the OpenAI provider is configured (it will automatically use your `OPENAI_API_KEY` env) [21]. You can also specify models by string like `'openai/gpt-4.1'` [22], but using the object/function form gives better type safety and clarity. **Ensure** that the environment variables for each provider are set (e.g. `OPENAI_API_KEY`, `ANTHROPIC_API_KEY`, `GOOGLE_GENERATIVE_AI_API_KEY`, `XAI_API_KEY`, `GROQ_API_KEY`, etc.) – the SDK providers default to these env vars if not explicitly provided [23] [24].

- **Streaming Responses:** Leverage streaming to keep your app responsive. The AI SDK provides a high-level `streamText()` function to initiate a streaming chat/completion [25] [26]. The returned result has helper methods to produce the proper Next.js response:

- Use `result.toTextStreamResponse()` for a **text-only** event stream (SSE) [27]. This is suitable if you just need the raw text streamed to the client (e.g., updating a `<textarea>` or simple UI).
- Use `result.toUIMessageStreamResponse()` for a **UI Message stream**, which includes token-by-token message formatting and tool invocation signals, designed to integrate with the Vercel AI SDK's React hooks or `assistant-ui` components [28]. This wraps the stream in a format that the `@assistant-ui/react` frontend can parse (including any `reasoning` or `tool` event chunks).

When to use which: If you are building a custom UI from scratch, `toTextStreamResponse` (returning `text/event-stream` SSE) is simpler and gives just the content tokens. If you're using the Vercel/assistant-ui chat components or need the structured stream (with markers for model "thinking" or tool usage), use `toUIMessageStreamResponse()` [28]. In our app, since we have integrated `assistant-ui`'s chat (`<Thread />` etc.), we return `toUIMessageStreamResponse()` so the frontend can handle the stream seamlessly [29].

- **Multi-Provider Setup:** The AI SDK allows **dynamic provider selection**. You can configure multiple providers and even runtime-switch them. For example, you might default to OpenAI but fall back to

Anthropic for certain requests. The recommended approach is to instantiate the model at call time based on a parameter. E.g.:

```
import { openai } from '@ai-sdk/openai';
import { anthropic } from '@ai-sdk/anthropic';
// ...
const provider = params.useAnthropic ? anthropic('claude-2') : openai('gpt-4o');
const response = streamText({
  model: provider,
  messages,
  // ...other options
});
return response.toTextStreamResponse();
```

The SDK will automatically route to the correct API using the keys. **Important:** For each provider, ensure any special configuration is handled. For example, Google's PaLM API might require specifying the project or uses an API key; OpenRouter requires setting an `OPENROUTER_API_KEY` and uses the `@openrouter/ai-sdk-provider` package. Check each provider's docs for any setup (the AI SDK docs list default env vars and options for each) [23].

- **Tool Integration (Functions):** The AI SDK v5 supports tool usage natively via function calling-like "tools." You can define **custom tools** using the `tool()` helper on the backend, or connect an **MCP (Model Context Protocol) server** for a library of tools. Best practice is:
- For simple or app-specific functions (like a database lookup or a calculation), define them directly in your code using `tool({ description, inputSchema, execute })` [30] [31]. For example, a weather tool:

  ```
  import { tool } from 'ai';
  import { z } from 'zod';
  const getWeather = tool({
    description: "Get current weather for a city",
    inputSchema: z.object({ city: z.string() }),
    execute: async ({ city }) => {
      // call weather API
      return await fetchWeatherForCity(city);
    }
  });
  ```

  Then include it in the `tools` field when calling `streamText`: `streamText({ model, messages, tools: { get_weather: getWeather } })`. The model can then invoke it by name.

- For **MCP Tools** (tool servers via Model Context Protocol): initialize an MCP client and convert its tools. For instance:

```
import { experimental_createMCPClient } from 'ai';
import { StreamableHTTPClientTransport } from '@modelcontextprotocol/sdk/
client/streamableHttp';
const mcpClient = await experimental_createMCPClient({
  transport: new StreamableHTTPClientTransport(new
URL(process.env.MCP_SERVER_URL!), { /* options */ })
});
const tools = await mcpClient.tools();  // schema discovery mode [32] [33]
const result = streamText({ model, messages, tools });
```

Here, `tools` will include all tools provided by your MCP server, allowing the model to call them. If you prefer type safety and controlling which tools are available, you can specify schemas for each tool you intend to use instead [34] [35]. This maps remote tools to local `tool()` definitions with the correct input types.

- **Frontend Tools:** If using `assistant-ui` React components, you might have *frontend-only tools* (like a client-side text-to-speech or file uploader). The `assistant-ui` library can forward these to the backend via an `AssistantChatTransport`. On the backend, use the provided `frontendTools()` helper to wrap them [36] [37]. In our example, the code merges incoming `tools` from the client with backend tools:

```
import { frontendTools } from '@assistant-ui/assistant-stream/ai-sdk';
// ...
const result = streamText({
  model: openai('gpt-4o'),
  messages: convertToModelMessages(messages),
  system,
  tools: {
    ...frontendTools(tools),       // integrate tools sent by client [36] [37]
    get_current_weather: getWeatherTool  // our backend-defined tool
  }
});
return result.toUIMessageStreamResponse();
```

This ensures the model can call both sets of tools transparently.

- **Streaming Usage Patterns:** Use the `onError` callback of `streamText()` to catch and log errors during streaming (since exceptions won't bubble in a streaming context) [38] [39]. Also utilize `onFinish` or `onCompletion` callbacks to handle any post-processing once a stream ends (for instance, saving the conversation to a database, which you can trigger in the `onFinish` of `toUIMessageStreamResponse` if using that) [40] [41]. The SDK's streaming is robust: it uses backpressure to only generate tokens as the client reads them [42]. This means you should always consume the stream (by returning the Response in Next.js) to let generation proceed.

Common Issues & Solutions:

- **Type Mismatches (LanguageModel vs POJO):** If you attempt to pass a plain object as `model` or use an unsupported format, you may get TypeScript errors. The correct approach is to use either the string format `'provider/model'` or the provider function (as noted above). The SDK's `LanguageModel` type expects certain properties (like `.generate` methods internally), which the provided helpers supply. If you see type errors, ensure you imported the provider correctly and are calling the function (e.g., `openai('gpt-4')` returns a valid LanguageModel object, whereas forgetting the call – using `openai` without `()` – would be wrong).

- **Streaming Response Usage:** Make sure to return the Response from `toTextStreamResponse()` / `toUIMessageStreamResponse()` directly from your Next.js Route Handler. Not doing so (or trying to manually iterate the stream) can break Next's response handling. Also, set the appropriate headers if using `toTextStreamResponse` – at least `Content-Type: text/event-stream` and `Cache-Control: no-cache` are recommended [43]. The `toUIMessageStreamResponse()` sets necessary headers for you, whereas for `toTextStreamResponse()` you can pass headers as an option [27].

- **Provider Capability Differences:** Not all models support the same features. For example, OpenAI GPT-4 supports function calling (tools) natively, while some other providers might not. The AI SDK abstracts tool usage by intercepting model output – if a provider doesn't support function calls, the model might still output a JSON blob that the SDK can intercept as a "tool call." But models vary in how well they follow the spec. *Solution:* Test each target model with your prompts. You might need provider-specific prompt tuning (e.g., instruct Anthropic to follow a certain format for tools). The AI SDK tries to normalize this, but be aware of subtle differences (like token limits per model, or image input support – e.g., if using OpenAI's Vision model via OpenRouter, ensure your UI and request adhere to that provider's input format).

- **API Key Management:** With multiple providers, it's easy to misconfigure keys. Common issues include hitting rate limits or invalid keys. Use the SDK's error messages to distinguish these. For instance, OpenAI errors will surface in `error.message` (e.g. "401 Unauthorized"), Anthropic might throw if the key is missing. A good practice is to **validate keys on startup** – e.g., attempt a small `generateText` with each configured provider when the server starts, or simply check that required env vars are present and log a warning if not. This avoids mystery failures at runtime.

- **Fallback Strategies:** If cost or availability is a concern, implement a graceful fallback. For example, if an OpenAI request fails due to rate limit or other error, you can catch it and retry with another provider (perhaps with a simplified prompt, since models differ). Keep in mind the response format might change – if fallback is used, try to standardize the output (maybe just returning text).

Provider Configuration Example: Here's a snippet configuring multiple providers and using the AI SDK:

```
// Environment: ensure these are set in .env
// OPENAI_API_KEY, ANTHROPIC_API_KEY, OPENROUTER_API_KEY, etc.

import { openai } from '@ai-sdk/openai';
import { anthropic } from '@ai-sdk/anthropic';
import { openRouter } from '@openrouter/ai-sdk-provider';

const modelId = process.env.DEFAULT_MODEL || 'openai/gpt-4.1';
// Decide provider based on prefix
let model;
if (modelId.startsWith('openai/')) {
  model = openai(modelId.split('/')[1]);
} else if (modelId.startsWith('anthropic/')) {
  model = anthropic(modelId.split('/')[1]);
} else if (modelId.startsWith('openrouter/')) {
  model = openRouter(modelId.split('/')[1]);
}

// Stream a chat completion
const result = streamText({
  model,
  messages: [
    { role: 'system', content: 'You are a helpful assistant.' },
    { role: 'user', content: 'Hello! Can you help me?' }
  ],
  // (tools can be added here if needed)
});
return result.toTextStreamResponse({
  headers: { 'Content-Type': 'text/event-stream' }
});
```

This example picks a model based on an environment config and streams a response. In a real app, you'd likely determine the model per request (e.g., based on user input or a parameter).

Troubleshooting Checklist:

1. **Confirm Environment Variables:** Ensure all necessary API keys are loaded. Test by calling a simple `generateText` with each provider in isolation (e.g., in a script or `node` REPL) to verify keys and network access.

2. **Verify Model Names:** Typos in model names can cause silent defaults or errors. Check the SDK docs for exact model IDs (e.g., OpenAI's `gpt-4` might appear as `gpt-4.1` in the SDK for certain versions). Using an invalid model string often yields an error like "Model not found." Cross-reference with provider documentation or use the provider's default if unsure.

3. **Check Streaming Behavior:** When testing streaming endpoints, use a tool like `curl` or your browser console to ensure the SSE stream is well-formed. If the stream terminates early or stalls, there may be an exception in your tool execution or an unhandled promise. Look at server logs for errors from `onError`. Ensure you're not accidentally awaiting the stream result (don't do `await streamText()` without consuming the stream – instead return the Response directly).

4. **Tool Call Debugging:** If tools are not being executed as expected, enable reasoning or debug logs. Many models will include a reasoning step or error if a tool call fails. The AI SDK's `result.toolCalls` and `toolResults` promises can be awaited after the stream finishes to inspect what happened [44] [45]. Use these to verify that the model attempted the tool and that the tool returned output.

5. **Consistency Across Providers:** When switching providers, verify that the response format still meets your UI's needs. For example, OpenAI might return a role "assistant" with certain content, while another might include additional metadata. The AI SDK normalizes message roles and such, but if using provider-specific features (like OpenAI function calling vs. an Anthropic workaround), you may need conditional handling. Test each provider path end-to-end with sample prompts.

By following these guidelines, you can harness Vercel's AI SDK to flexibly integrate multiple AI models, stream outputs to the user in real-time, and incorporate advanced features like tool use, all under a unified framework.

3. Better Auth 1.3.27 – Next.js 15 Authentication & OAuth

Current Status: Better Auth is an authentication framework that has gained popularity as an alternative to Auth.js/NextAuth for Next.js apps. As of version 1.3.x, it supports Next.js 15 (App Router) with features like email/password, social logins (OAuth providers), passkeys, and more [46] [47]. It's designed to work with both **Edge and Node runtimes**, but there are caveats. Better Auth's Next.js integration uses a catch-all Route Handler (`/api/auth/[...all]`) for its server endpoints [48]. Internally, some parts of Better Auth (especially cookie/session handling) assume Node.js APIs, which can conflict with Edge runtime. Overall, it's compatible with Next 15, but developers must carefully configure runtimes and middleware.

Best Practices: Setting up Better Auth in Next.js 15 involves a few key steps:

- **Configure the Auth Instance:** In a server file (e.g. `lib/auth.ts`), create your Better Auth instance with your desired providers and options:

```
import { betterAuth } from 'better-auth';
import { nextCookies } from 'better-auth/next-js';
export const auth = betterAuth({
  // Database adapter, e.g., drizzle or prisma (not shown here)
  // Enable providers:
  socialProviders: {
    google: {
      clientId: process.env.GOOGLE_CLIENT_ID!,
      clientSecret: process.env.GOOGLE_CLIENT_SECRET!,
      scope: ['email', 'profile']  // optional scopes [49] [50]
    },
    github: {
      clientId: process.env.GITHUB_CLIENT_ID!,
      clientSecret: process.env.GITHUB_CLIENT_SECRET!
    }
    // ...other providers
  },
  // Other options: JWT, session length, etc.
  plugins: [nextCookies()]  // ensures cookies set in Server Actions get
properly persisted [51] [52]
});
```

This configuration sets up Google and GitHub OAuth. The `socialProviders` config is straightforward – provide `clientId` and `clientSecret` (which you obtain from Google Cloud Console and GitHub OAuth app respectively), and optionally scopes or a custom redirect URI if needed (by default Better Auth uses `/api/auth/callback/<provider>` as redirect) [53] [54]. The `nextCookies()` plugin is highly recommended in Next 13+; it makes sure that when you call auth actions in Server Actions or RSC, any cookies (like session cookies) in the response are properly set via Next's `cookies()` API [51].

- **Mount the Auth Route:** Next, create the Next.js Route Handler for Better Auth's endpoints. Typically, you add `app/api/auth/[...all]/route.ts`:

```
import { auth } from '@/lib/auth';
import { toNextJsHandler } from 'better-auth/next-js';
export const { GET, POST } = toNextJsHandler(auth.handler);
```

This single file exposes all the necessary `GET`/`POST` routes for login, callbacks, etc., under the `/api/auth/*` path, which Better Auth expects by default [48] [55]. (If using the Pages Router, the setup would differ with `toNodeHandler`, but in Next 15 App Router, the above is correct).

- **Client Integration:** On the client side, create a Better Auth client instance to call auth functions:

```
// e.g., lib/auth-client.ts
import { createAuthClient } from 'better-auth/react';
export const authClient = createAuthClient();
```

This `authClient` provides methods like `authClient.signIn.email(...)`, `authClient.signIn.social({ provider: 'google' })`, etc., which automatically call your API routes and manage internal state (Better Auth uses a nano-store to track auth state and re-render on changes) [56] [57]. Use these methods in your React components for login forms or social login buttons.

- **Protected Routes & Middleware:** For guarding pages, Better Auth suggests using Next.js Middleware to quickly check for a session cookie and redirect if not present [58]. The provided helper `getSessionCookie()` (or `getCookieCache()`) can parse the auth session cookie from the `NextRequest`. **However, as noted earlier, this function currently uses some Node APIs and isn't fully Edge-safe** – it may return `null` unexpectedly on Edge [14]. The recommended approach is to keep the middleware check simple: for example, in `middleware.ts`:

```
import { NextResponse } from 'next/server';
import { getSessionCookie } from 'better-auth/cookies';
import { cookiePrefix } from '@/lib/auth/config';  // wherever you defined
your cookie prefix
export function middleware(request: NextRequest) {
  const session = getSessionCookie(request, { cookiePrefix });
  const { pathname } = request.nextUrl;
  if (
    !session &&
    !pathname.startsWith('/api/auth') &&
    pathname !== '/login'
  ) {
    // If no session cookie, redirect to login (preserve redirect path)
    return NextResponse.redirect(new URL(`/login?callbackUrl=${pathname}`,
request.url));
  }
  // If logged in or on auth pages, proceed
  return NextResponse.next();
}
export const config = { matcher: ['/((?!_next/static|_next/image|
favicon.ico).*)'] };
```

This example uses `getSessionCookie` to simply detect if the session cookie exists (and is unexpired) – it doesn't fully validate it, but enough to decide redirect vs allow. This avoids a DB call in middleware (which is good, as middleware should be fast and non-blocking). If you encounter issues with `getSessionCookie` on Edge (as some have), a fallback is to manually check for a cookie name like `ba-session` (Better Auth's default) in `request.cookies`.

- **OAuth Provider Setup:** To add OAuth (Google, GitHub, etc.), as shown above, configure `socialProviders` in the server config. Additionally, you must set the OAuth app's redirect URI to your site. For Google, add `https://your-domain.com/api/auth/callback/google` in Google Cloud Console for OAuth. Better Auth handles the OAuth flow: calling `authClient.signIn.social({ provider: 'google' })` will redirect the user to Google's consent page, then back to the callback which Better Auth processes. One thing to note: if your app uses custom domains or multi-tenant subdomains, you may need to dynamically configure allowed callback URLs or handle them in the Better Auth config (as per your needs).

Common Issues & Solutions:

- **Edge Runtime Warnings:** The build warning about `process.platform` in Better Auth is benign but indicates that part of Better Auth isn't Edge-compatible [13]. Currently, Better Auth's devs acknowledge that `getCookieCache` / `getSessionCookie` might not fully work in Edge middleware environments [14] [59]. If you rely on those and notice that `session` is always null in middleware, consider the simplified approach above or await improved Edge support in future releases. The good news is that this is just a **warning** and doesn't break the build – your app can still function if the cookie check returns null (Better Auth will then redirect to login, which is fail-safe).

- **Session Not Persisting in RSC/Actions:** If you call `auth.api.signInEmail()` or other server actions to log in and notice the cookie isn't set, ensure you have the `nextCookies()` plugin added [51]. Without it, Next's Server Actions cannot set cookies automatically due to their nature [60]. The plugin intercepts responses and uses Next's cookie store to set the session cookie for you.

- **OAuth Callback Issues:** A common mistake is not including the proper redirect. Better Auth by default uses `/api/auth/callback/[provider]`. If, after OAuth login, users end up at a blank page or `/api/auth` without further action, double-check that:

- The OAuth app's redirect URI matches exactly.
- Your domain is correct (for development, use something like `http://localhost:3000/api/auth/callback/google`).

- In production, if using custom domains, Better Auth might need configuration for the base URL (in many cases it infers from the request).

- **Email Verification & Others:** Better Auth has additional features (email verification, 2FA). If enabled, ensure you configure an email provider for sending emails, etc. Always test the full flow: sign up, verify email (check for email sending issues), sign in, sign out, password reset if applicable.

- **Database Setup:** While the question doesn't focus on it, note that Better Auth needs a database (it doesn't manage its own by default). You likely use Drizzle or Prisma with it. Ensure migrations for the user tables (Better Auth might provide these or you create them according to its schema) are applied. Many authentication issues (unable to log in, etc.) can be due to missing DB tables or columns.

Migration Guides (if applicable): If coming from NextAuth, note that Better Auth has a different API. However, since we started with Better Auth, main migration notes are between versions. v1.3.x is current; if upgrading from earlier v1.2, ensure to update the plugin usage (e.g., `nextCookies()` replaced older cookie handling). Also, Next.js 15's App Router is fully supported by Better Auth, so migrating from Pages Router involves switching to the `toNextJsHandler` and possibly adjusting how you use server actions vs. `getServerSession` (Better Auth uses `auth.api.getSession()` as shown in their docs for RSC/Server Actions [61] [62]).

Configuration Example: Quick reference of key files:

- `lib/auth.ts`:

```
import { betterAuth } from 'better-auth';
import { nextCookies } from 'better-auth/next-js';
import { drizzleAdapter } from 'better-auth/drizzle'; // hypothetical adapter
for Drizzle ORM
import { db } from '@/lib/db'; // your drizzle db instance
export const auth = betterAuth({
  database: drizzleAdapter(db, { /* ...options... */ }),
  session: { cookieName: 'ba-session', domain: undefined }, // domain if needed
for cookies
  socialProviders: {
    google: { clientId: 'XXX', clientSecret: 'YYY', scope:
['email','profile'] },
    github: { clientId: 'AAA', clientSecret: 'BBB' }
  },
  plugins: [nextCookies()]
});
```

- `app/api/auth/[...all]/route.ts`:

```
import { auth } from '@/lib/auth';
import { toNextJsHandler } from 'better-auth/next-js';
export const { GET, POST } = toNextJsHandler(auth.handler);
```

- `lib/auth-client.ts`:

```
import { createAuthClient } from 'better-auth/react';
export const authClient = createAuthClient();
```

- `middleware.ts`: (if using, for protected routes)

```
import { NextRequest, NextResponse } from 'next/server';
import { getSessionCookie } from 'better-auth/cookies';
import { cookiePrefix } from '@/lib/auth/config';
export function middleware(request: NextRequest) {
  const session = getSessionCookie(request, { cookiePrefix: cookiePrefix ??
'ba' });
  const { pathname } = request.nextUrl;
  if (!session && pathname !== '/login' && !pathname.startsWith('/api/auth')) {
    return NextResponse.redirect(new URL('/login', request.url));
  }
  return NextResponse.next();
}
export const config = { matcher: ['/((?!_next/static|_next/image|
favicon.ico).*)'] };
```

Troubleshooting Checklist:

1. **Edge Warning in Build:** As long as the build completes, this is a warning. Confirm if Better Auth functionality works despite it. If the middleware cookie check isn't working, consider simplifying it or removing it (Better Auth will still protect API routes on the server side).

2. **Test End-to-End:** Create a test account via sign-up (if using email/password) or try the Google login in a development environment. Follow the network requests:

3. `/api/auth/signup` or `/api/auth/signin` calls – check for any 500 errors.
4. OAuth login – ensure you get redirected to the provider and back. On return, check that `/api/auth/callback/google` was called and returned a NextResponse (likely a redirect to your app).

5. **Check Cookies:** After login, verify that the session cookie (by default something like `ba-session`) is present in the browser, not HTTP-only (Better Auth typically uses HttpOnly cookies for session), and contains data. If not present, the login didn't complete properly – check server logs for errors during the auth flow.

6. **Session Retrieval in Server Components:** If you need the user session in a Server Component (RSC), use `await auth.api.getSession({ headers: await headers() })` as in Better Auth docs [63] [62]. If this returns `null` unexpectedly, it could be missing cookies (see above) or an issue with how `headers()` are passed. Ensure you include the `headers` from `next/headers` exactly as shown.

7. **Database Checks:** If certain actions fail (e.g., signIn throwing an error), check your database. Better Auth will error if, say, a user already exists (depending on config) or if the DB connection is failing. Make sure your connection pool is properly set up for Better Auth's DB adapter.

8. **Version Updates:** Keep an eye on Better Auth's changelog. They are actively improving Edge support and adding features. Upgrading might fix issues like the Edge cookie parsing in the future. Always test after upgrading.

By carefully configuring Better Auth and addressing the runtime considerations, you'll get a robust auth system with Next.js 15. It will handle all the heavy lifting of user accounts and OAuth, while you focus on your app's core logic.

4. Drizzle ORM 0.41.0 – PostgreSQL & pgvector Usage

Current Status: Drizzle ORM is a type-safe SQL ORM for TypeScript, which in v0.41.0 supports PostgreSQL and even advanced features like the `pgvector` extension for embeddings. It emphasizes full typing of queries. The current version provides various query builder methods (`select`, `where`, `insert`, etc.) that return immutable query objects. TypeScript types are generated or inferred from your schema definitions, giving you compile-time safety. **PGVector** is supported via Drizzle's `vector` column type and custom SQL functions for similarity search [64] [65]. One challenge with Drizzle's type system is conditional query building – naively adding `.where()` clauses in an if/else can confuse TypeScript's inference, resulting in errors about missing properties (like `'where' is missing in type Omit<...>`). We must use recommended patterns to keep types intact.

Best Practices:

- **Type-Safe Conditional Queries:** To avoid the type narrowing issues when conditionally chaining query builders, use either:
- **Single `.where` with combined conditions:** Utilize Drizzle's `and()` / `or()` logical helpers and the fact that `.where()` can accept `undefined` filters. For example:

```
import { and, eq, ilike } from 'drizzle-orm';
const { title, category } = searchFilters;
const posts = await db.select().from(postsTable).where(
  and(
    title ? ilike(postsTable.title, `%${title}%`) : undefined,
    category ? eq(postsTable.category, category) : undefined
  )
);
```

  In this pattern, each condition is wrapped with a ternary that yields either a valid filter or `undefined` – Drizzle will ignore `undefined` filters [66] [67]. This keeps the query object type consistent (you call `.where()` exactly once).
- **Filters Array Pattern:** Alternatively, accumulate `SQL` conditions in an array, then spread them into an `and()` or into `.where()`. For instance:

```
import { sql } from 'drizzle-orm';
const filters: (SQL | undefined)[] = [];
if (filters.year) {
  filters.push(sql`EXTRACT(YEAR FROM ${LogsTable.date}) = ${filters.year}
`);
}
if (filters.userId) {
  filters.push(sql`${LogsTable.userId} = ${filters.userId}`);
}
const query = db.select().from(LogsTable).where(and(...filters));
```

  This approach was suggested by Drizzle team members to fix the `'where' property missing` error [68] [69]. The key is that `.where()` is called once with all conditions combined, rather than conditionally calling `.where()` multiple times on the same query object (which leads to a narrowed type).

Using these patterns, you avoid the TypeScript issue (where the intermediate query type lost the `.where` method). The error we encountered (*Property 'where' is missing in type 'Omit<PgSelectBase<...>...>* [70] was resolved by adopting the above approach: build conditions first, then apply them in one `.where()` call [71].

- **Drizzle + pgvector (Embeddings):** Drizzle's `pg-core` module provides a `vector` column type for the `vector` data type. Define your schema accordingly:

```
import { pgTable, serial, text, vector } from 'drizzle-orm/pg-core';
export const Documents = pgTable('documents', {
  id: serial('id').primaryKey(),
  content: text('content'),
  embedding: vector('embedding', { dimensions: 1536 })  // e.g. 1536-dim
embedding [64]
});
```

Ensure the `vector` extension is enabled in your database. Drizzle won't auto-create it, so run `CREATE EXTENSION vector;` via a migration or manual SQL [72]. For similarity search, Drizzle offers the `cosineDistance` function. To query for nearest vectors:

```
import { cosineDistance, lt, sql } from 'drizzle-orm';
const embedding = [...];  // some embedding array
const similarity = sql<number>`1 - (${cosineDistance(Documents.embedding,
embedding)})`;
const results = await db.select({
    docId: Documents.id,
    score: similarity
  })
  .from(Documents)
  .where(lt(cosineDistance(Documents.embedding, embedding), 0.3));
```

In practice, you might compute `similarity` as shown for ordering, or use `< 0.3` etc. The snippet above calculates a similarity score (1 - cosineDistance) and filters for those above a threshold (equivalent to distance below a threshold) [65] [73]. Drizzle's typing even covers this – `sql<number>` is used to inform TypeScript that the expression is numeric. Remember to also create an index on the vector column for performance (HNSW or IVF index as needed) [74] [75].

- **Connection Management in Next/Vercel:** Use a single `Pool` and a single Drizzle `db` instance for your app, rather than creating new connections per request. In a serverless context, define it globally (e.g. in `lib/db.ts`) so that it can be reused across invocations when possible:

```
import { Pool } from 'pg';
import { drizzle } from 'drizzle-orm/node-postgres';
const pool = new Pool({ connectionString: process.env.DATABASE_URL });
export const db = drizzle(pool);
```

This uses pg's internal pooling. On Vercel, you can further improve reliability by using Vercel's `attachDatabasePool(pool)` as recommended [76] [77] – it will close idle connections when the function is about to freeze, mitigating connection leaks in serverless.

- **Cloud SQL Specifics:** If you are connecting to Google Cloud SQL (Postgres), note that each serverless function might open new connections. Cloud SQL has a connection limit (depending on instance size, often around 100). Using the pool as above helps reuse connections within each server instance. For additional scaling, consider **Cloud SQL's Managed Connection Pooler** or **PgBouncer**. If using Cloud SQL's public IP, ensure you have authorized Vercel's IPs or use the Cloud SQL Auth Proxy (though on Vercel you'd run that externally, since you can't easily run a sidecar in serverless). The key is to avoid exhausting connections: always release connections (`client.release()` in pg Pool is handled automatically when using `pool.query` or drizzle).

- **Migrations & Schema Management:** Drizzle Kit is the companion for migrations. Best practice:

- Define your schema in code (as shown in `pgTable` definitions).
- Use `drizzle-kit push` or `drizzle-kit generate` to create SQL migration files whenever your schema changes. This ensures your database schema stays in sync with your code types.
- Version control your migration files. In deployment, you have a few options:
    - Run migrations externally (e.g., via CI/CD or a script) before deploying the new code.
    - Or run them at runtime (Better Auth's postinstall log indicated skipping migrations on build, likely to apply on startup). If doing runtime, implement a check to run once (for example, a simple check: if a certain table or a schema_version table indicates out-of-date, run `drizzle.migrate()`). **However,** be cautious: running migrations in a serverless environment can conflict if multiple instances start up. It's safer to run them out-of-band if possible.
- Regularly use `drizzle-kit check` to ensure your code schema and DB schema match, especially if making manual DB changes.

Common Issues & Solutions:

- **Type Errors in Queries:** Aside from the conditional issue above, you might encounter errors if you, say, forget to select a required field or the types don't match. Drizzle's error messages can be verbose (because of deep generic types). Narrow down the problem by simplifying the query or checking the types of each part. If a certain helper (like `sql\`` `) gives an`any`type, cast it with`sql<Type>` as shown in the vector example to help TypeScript.

- **Missing `.where` or `.having` in type:** These occur when you assign a query to a variable and then reuse it in a way TS can't follow. The solution is to avoid reassigning a query in multiple steps if possible. Instead, chain it all or use the array/and pattern. In cases where you must conditionally build in steps (perhaps assembling a complex query with many optional joins/filters), you might consider using Drizzle's `dynamic` mode or even raw SQL for the trickiest part to bypass type limitations, but try the recommended approach first.

- **Pgvector usage pitfalls:** Ensure the `dimensions` in your `vector()` column matches exactly the length of arrays you store. If you attempt to insert a vector of wrong length, Postgres will error. Drizzle might not catch that at compile time (dimensions are in the type but not enforced by TS on the array length). Also, when using `cosineDistance` or others, note that Drizzle generates SQL calling the extension functions. If you see a runtime SQL error, check that the extension is enabled and the SQL generated is correct. You can log queries (`drizzle` has a debug mode or you can `console.log(query.toSQL().sql)` in dev) to verify.

- **Connection Pool Exhaustion:** If you see errors like “sorry, too many clients” or timeouts connecting to the DB on Vercel, it means you’re hitting the connection limit. Using the global pool as above, and possibly reducing `max` connections in the Pool config (to, say, 5) can help. Also, consider Vercel’s new “Fluid Compute” which keeps functions warm and thus reuses connections more effectively [78]. If using Cloud SQL, Google’s **Managed Connection Pooling (PGbouncer)** can be enabled on the instance [79], which can allow many serverless connections to share fewer actual DB connections.

Query Pattern Examples:

- *Conditional Select Example:* Fetch logs with optional filters:

```
import { and, sql } from 'drizzle-orm';
import { LogsTable } from '@/db/schema';
type LogFilters = { clientId?: string; userId?: string };
function getLogs(filters: LogFilters) {
  return db.select().from(LogsTable).where(and(
    filters.clientId ? sql`${LogsTable.clientId} = ${filters.clientId}` :
undefined,
    filters.userId ? sql`${LogsTable.userId} = ${filters.userId}` :
undefined
  ));
}
```

This produces a type-safe query object. If both filters are absent, `.where(and(undefined, undefined))` is effectively ignored by Drizzle (or rather, `and()` returns undefined which means no condition) [66].

- *Vector Similarity Example:* Find top 5 similar items:

```
import { desc, cosineDistance, sql } from 'drizzle-orm';
const embedding = await generateEmbedding(queryText);
const similarityExpr = sql<number>`1 - (${cosineDistance(Items.embedding,
embedding)})`;
const similarItems = await db.select({
    itemId: Items.id,
    similarity: similarityExpr
  })
  .from(Items)
  .orderBy((aliases) => desc(aliases.similarity))
  .limit(5);
```

(Ensure you have an index on `Items.embedding` with `vector_cosine_ops` for performance [74].)

Troubleshooting Checklist:

1. **Schema Sync:** If queries fail or return wrong data, verify the database schema vs. Drizzle schema definitions. E.g., did you run the latest migration that added a new column your code expects? Use `drizzle-kit pull` to introspect DB and compare to your code models.

2. **Run Sample Queries in psql:** For complex queries (especially those with raw SQL parts), copy the SQL (use `query.toSQL().sql`) and run it directly in a SQL client. This helps pinpoint if the issue is with Drizzle or the data. For instance, if a cosine distance query returns zero rows, maybe your threshold was too high or embeddings were not stored correctly.

3. **Enable Debug Logging:** Drizzle can log executed SQL. In Node, set `DEBUG="drizzle:query"` or similar (check Drizzle docs for the exact flag) to see queries printed. This confirms what Drizzle is actually sending to Postgres.

4. **Connection Count Monitoring:** On Cloud SQL, use the monitoring tools (or query `pg_stat_activity`) to see how many connections are open. If it spikes with each request and doesn't drop, you might not be releasing connections properly or not using pooling. The `Pool` approach with global instantiation and usage of `.query` or `.connect/release` should generally handle this. Also ensure you're not accidentally creating a new Pool for each request (common mistake: putting `new Pool()` inside your handler function – avoid that).

5. **Vector Data Checks:** If using pgvector, test the full pipeline: insert an item with an embedding, then query for it by a similar embedding and ensure it comes up. If not, debug whether the embeddings are being saved correctly (e.g., are you normalizing them or using the same model for generating them?). Sometimes issues arise if the embeddings aren't comparable (different model or not normalized for cosine if needed).

By following these strategies and checks, you can confidently use Drizzle ORM for complex queries with full type safety, including leveraging Postgres features like pgvector for AI-driven functionalities.

5. Google Cloud Platform Integration – Storage, Database, and APIs

Current Status: The application leverages several Google Cloud services: **Cloud Storage (GCS)** for file storage, **Cloud SQL (PostgreSQL)** for the database (implied by context), Google **Workspace APIs** (Drive/Sheets/Docs) for document handling, and **Google Maps API** for location services. As of 2025, Google Cloud provides robust Node.js client libraries (e.g. `@google-cloud/storage`, `googleapis` for Workspace) and these integrate with service accounts or OAuth 2.0 credentials. Quota management and secure usage are key concerns when using these APIs in a production app.

Best Practices:

- **Google Cloud Storage (GCS):** Use the official `@google-cloud/storage` library (which we have) for interacting with buckets. For most use-cases in a web app, you'll want to generate **Signed URLs** for secure client access to uploads or downloads. A Signed URL allows a client to PUT or GET a specific file without needing direct credentials, and expires after a short time.

- **Signed URL Generation:**

```
import { Storage } from '@google-cloud/storage';
const storage = new Storage();
const bucket = storage.bucket(bucketName);
const file = bucket.file(fileName);
const [url] = await file.getSignedUrl({
  action: 'read',  // or 'write' for upload
  expires: Date.now() + 5 * 60 * 1000 // 5 minutes
});
console.log(`Generated signed URL: ${url}`); 【53†L1-L9】
```

  Ensure the service account used by your app has the **Storage Object Signer** role (or equivalent permissions) to create signed URLs [80] . By default, if your Vercel app is not running on GCP, you might use a service account JSON key. Set `GOOGLE_APPLICATION_CREDENTIALS` to the path or use `storage.fromJSON()` to auth explicitly with the key.
- **Security:** Limit the expiry (typically 5-15 minutes for upload links, maybe longer for download if needed, but generally short-lived). Also, set content type or disposition in the signed URL options to mitigate certain attacks (like for uploads, specify the exact content type allowed).

- **Public vs Private:** If files aren't sensitive, you could make the bucket public or use Firebase Storage, but since the app likely deals with private user documents (estimations, etc.), keep the bucket private and always use signed URLs or authenticated requests.

- **Cloud SQL (Postgres):** We've discussed pooling in section 4. Additionally:

- If on Google Cloud, consider using the **Cloud SQL Auth Proxy** for local development to connect easily. In production (Vercel), since you can't run the proxy easily, use a direct connection string with a public IP and strong password, or use **VPC connectors** if on Google Cloud Run or similar. On Vercel, you're likely using the public IP approach.
- **Connection Pooling:** As mentioned, use PgBouncer or Cloud SQL's built-in (currently GCP offers "Managed Connection Pooling" in preview which uses an intermediary). If you observe connection issues, one quick fix is enabling the **PGBouncer** option in Cloud SQL (which turns on a mode for Postgres that pools at the database level) [79] .

- **Connection Limit and Slow Start:** Cloud SQL connections can be a bit slow to establish. Use `pool.connect()` during the initialization of your function (or globally) rather than per request, to amortize that cost. Also, you might increase Node's socket keep-alive or Cloud SQL's timeout so that connections aren't dropped too eagerly between requests.

- **Google Workspace APIs (Drive/Sheets/Docs):** For these, the **Google APIs Node.js client** (`googleapis` package) is commonly used. Key points:

- Decide between a **Service Account** vs **User OAuth** for access. If the app is analyzing user's Drive/Docs, you need user OAuth consent (and appropriate scopes). If it's accessing a corporate Drive or a single service account's Drive, you can use a service account directly.

- **OAuth Setup:** Use Google's developer console to create OAuth credentials. Better Auth can integrate Google OAuth for sign-in, but for broader API access (like reading a user's Google Sheets), you must request the additional scopes. For example, if you want to read Google Sheets, you'd add scope `"https://www.googleapis.com/auth/spreadsheets.readonly"` to the Google provider config [81] [82]. After the user consents, you can retrieve the access token using `authClient.getAccessToken({ providerId: 'google' })` [83] [84]. This token can then be used with `googleapis`:

```
import { google } from 'googleapis';
const sheets = google.sheets('v4');
sheets.spreadsheets.values.get({
  spreadsheetId,
  range: 'Sheet1!A1:B10',
  auth: oauthClient // an OAuth2 client with credentials set
});
```

  Alternatively, since we have the token via Better Auth, set the credentials:

```
oauth2Client.setCredentials({ access_token: token });
const drive = google.drive({ version: 'v3', auth: oauth2Client });
const files = await drive.files.list();
```

- **Quotas:** Google APIs have daily and per-second quotas. Monitor usage in Google Cloud Console. If you expect heavy use, you may need to request quota increases. Implement **exponential backoff** on 403 rateLimitExceeded errors. The `googleapis` library does not automatically retry by default, so handle errors: if error code is 403 or 429, consider retrying after a delay.
- **Batching:** For Sheets and Docs, batch requests if possible (Google APIs often support batch via HTTP batch endpoints or simply combining data in one call).

- **Caching:** Use caching (even SWR on the frontend or a simple in-memory cache on the backend) to avoid frequent repeated API calls for the same data. For example, if you fetch a Google Doc content to analyze it, cache that content or analysis result for some minutes if the document isn't expected to change often, to avoid hitting the API repeatedly.

- **Google Maps API:** This typically refers to either Google Maps JavaScript API (for embedding maps in frontend) or web services like Geocoding, Places, Distance Matrix, etc.

- For embedding a map on the frontend, use the Maps JS API with an API key. Restrict the key to your domain in Google Cloud Console for security (so it only works on your site). The Maps JS API is loaded via a `<script>` tag with your API key.
- For server-side geocoding or places search, use the `@googlemaps/google-maps-services-js` library or simple HTTP calls. These require an API key or (for certain enterprise accounts) a client ID and signature. Keep the API key secret (on server side) – do not expose it in frontend code (except for strictly necessary ones like the Maps embed which must use a browser key).
- **Rate Limiting:** Google Maps places/geocode APIs have rate limits (e.g., 50 QPS per project by default and daily quotas). Implement a query rate limiter if needed. The Node client library can

queue and retry some requests, but you might need your own throttle (for instance, using a token bucket or simple delay between calls if you approach QPS limit).

- **Error Handling:** Watch for status codes in Google Maps API responses (`ZERO_RESULTS`, `OVER_QUERY_LIMIT`, etc.). Handle them gracefully (e.g., if geocoding fails due to limit, perhaps fallback to a secondary geocoding service or cache the last known good result).

- **Security Best Practices:**

- **Least Privilege for Service Accounts:** If using a GCP service account for Storage or other APIs, limit its roles to only what's needed (e.g., Storage Object Admin on a specific bucket, not Project Editor).
- **Store Credentials Securely:** On Vercel, use Environment Variables for API keys and service account JSON (you can base64-encode the JSON or use individual env vars for key details). Do not commit these to git.
- **Validate Inputs to APIs:** When making API calls based on user input (e.g., searching a Google Sheet by ID provided by user, or geocoding a user-entered address), validate and sanitize inputs. For example, ensure sheet IDs match an expected pattern to prevent injection, and handle address input carefully to avoid abuse (someone could attempt to use your app as a proxy for bulk geocoding – implement usage limits per user).
- **Monitor Usage:** Set up alerting on GCP for API usage (most APIs allow you to set a budget or quota alert). This can detect if your keys are being misused or if a bug causes a flood of requests (saving you from a large bill or quota exhaustion).

Expected Output Examples:

- *GCS Signed URL Example (Download):*

```
import { Storage } from '@google-cloud/storage';
const storage = new Storage();
const bucket = storage.bucket('my-bucket');
const file = bucket.file('reports/estimate123.pdf');
const [downloadUrl] = await file.getSignedUrl({
  action: 'read',
  expires: Date.now() + 15 * 60 * 1000, // 15 minutes
});
// send this URL to client for downloading the file securely
```

This will generate a time-limited URL that anyone with the link can use to download that specific file [80].

- *Drive API Example:*

```
import { google } from 'googleapis';
// after obtaining OAuth2 tokens for Google Drive scope
const auth = new google.auth.OAuth2();
auth.setCredentials({ access_token: userToken });
```

```
const drive = google.drive({ version: 'v3', auth });
const resp = await drive.files.list({ pageSize: 10 });
console.log(resp.data.files);
```

This lists files in the user's Drive. You might use this after the user connects their Google account via Better Auth.

Troubleshooting Checklist:

1. **Cloud Storage Access Issues:** If a signed URL isn't working (e.g., getting authorization errors when using it):
2. Verify the service account has permission to sign URLs. If not, you might get an "Invalid Credentials" on the URL.
3. Ensure the system clock on the server is correct; signed URLs are time-sensitive and an skew can cause immediate expiration or future dating issues.
4. If uploading via a signed URL (HTTP PUT), ensure you use the exact HTTP method and headers the URL expects (content-type, etc.). Misuse can lead to "Signature does not match" errors.
5. **Cloud SQL Connectivity:** If the app cannot connect to the DB:
6. Confirm the connection string (host, port, user, password) is correct. For Cloud SQL, if using a private IP, your server must be in the same VPC or have the Cloud SQL proxy. If using public, check that the IP is authorized.
7. Check Vercel env vars (DATABASE_URL) is set and no whitespace or encoding issues.
8. Try connecting locally using `psql` or a desktop client with the same creds to rule out network issues.
9. If connections sporadically fail, monitor for pattern – it could be max connections. If so, implement pooling as above, and consider raising the Cloud SQL instance tier if needed for more connections or enabling the pooler.
10. **Google API Quota Errors:** If certain functionality stops working and logs show quota errors (403 or 429 status):
11. Log the response from Google; often it includes `"error": {"message": "...quota..."}`.
12. Apply exponential backoff on retries for that request in code.
13. If user-initiated, inform the user to retry after a short wait.
14. Check Google Cloud Console Quotas section for that API – you might need to request an increase or adjust your usage pattern.
15. In the interim, possibly fall back to a secondary method if available (for example, if Google Maps geocoding is at limit, you could fall back to OpenStreetMap Nominatim for non-production-critical use).
16. **OAuth Token Issues:** If calls to Google APIs using user tokens fail with "Unauthenticated":

17. The token might have expired. Ensure you refresh tokens. Better Auth's `getAccessToken` will automatically refresh if needed [83] [85]. Make sure you store the refresh token when the user links their account (Better Auth should handle this under the hood).
18. The scope might not include what you need. Check that the user granted the scope. If not, you may need to use `authClient.linkSocial({ provider: 'google', scopes: [ ... ] })` to get additional consent [81] [82].

19. If using a service account for Google APIs (e.g., a service account for Sheets in a single Google Workspace), ensure the service account email is added to the doc's share (for Drive/Docs access) or use domain-wide delegation if in an enterprise Google Workspace.

20. **Maps API Errors:** If the map isn't showing or geocoding fails:

21. For maps on frontend: open browser console – Google Maps API usually logs useful errors (like "API key not authorized for this domain" or "Invalid key"). Adjust API key restrictions accordingly.
22. For geocoding: handle the `OVER_QUERY_LIMIT` by slowing down. Also note daily quotas – you might need to pay for a higher tier if exceeding free quota regularly.
23. Ensure you're not exposing your server-side API key inadvertently. Always use server-side geocoding with a server key that's kept secret, and restrict that key by IP or not at all (since it's secret). For client-side map, use a separate browser key with HTTP referer restrictions.

By following the above checklist and applying best practices, your integration with Google Cloud services will be secure, efficient, and robust, allowing your application to manage files, data, and external API calls reliably.

6. Multi-Provider AI Model Management – Capability Detection & Cost Optimization

Current Status: The application uses multiple AI model providers (OpenAI, Anthropic, Google's PaLM via `@google-cloud/genai`, xAI Grok, OpenRouter, etc.). Each provider has different capabilities (context length, function-calling ability, multi-modality, etc.) and different cost profiles. In 2025, model selection and routing is a key concern for AI-driven apps to balance performance and cost. The Vercel AI SDK simplifies using multiple providers but doesn't automatically decide *which* model to use – that logic is up to the application.

Best Practices:

- **Centralize Provider Config:** Maintain a configuration or factory function that knows about all your providers and model options. For example, define an enum or constants for provider names (`'openai' | 'anthropic' | 'google' | ...`) and have a function `getModelForRequest(request: AIRequest): LanguageModel` that returns the appropriate model object. This could consider request parameters like desired accuracy, tools needed, etc. By centralizing, you avoid sprinkling provider-specific logic throughout your code.

- **Capability Detection:** Determine the capabilities you care about (e.g., *supports function calling*, *accepts image inputs*, *maximum tokens*). You can maintain a static mapping:

```
const modelCapabilities = {
  'openai/gpt-4': { tools: true, images: false, maxTokens: 8192 },
  'openai/gpt-4-vision': { tools: true, images: true, maxTokens: 8192 },
  'anthropic/claude-2': { tools: limited, images: false, maxTokens:
100k },
  // ...
};
```

If dynamic detection is needed, query the model metadata if available (some providers have endpoints or version info). For example, OpenAI's function calling is known by model name (all GPT-4 and GPT-3.5 support it), Anthropic's Claude can use a simpler JSON-based tool format but not as explicitly function calling – you might implement a prompt-based tool format for it. The AI SDK's provider registry might expose some flags, but mostly you manage this.

Use these capabilities to route requests. For instance, if a request includes an image as input, you'd choose a model that supports vision (e.g., OpenAI's "multimodal" model via OpenRouter or xAI Grok if it supports images). If a request requires tools (e.g., the user asks for something requiring browsing), prefer a model with strong function-calling (GPT-4) or one explicitly integrated with tools (xAI Grok is designed for tool use and reasoning).

- **API Key Management:** Use separate API keys for each provider and load them from secure storage. The AI SDK picks up keys from env vars, but ensure you do not hard-code them. Have fallback logic: e.g., if a user selects a provider for which the key isn't configured, either throw an error or default to a provider that is available. Log an error if keys are missing to catch misconfiguration early.

- **Fallback and Redundancy:** Design a fallback strategy for provider failures. For example:

- If OpenAI API is down or returns an error (500 or a specific error code), automatically retry the query on Anthropic (maybe truncating or adjusting format if needed).
- If a cheaper model (say GPT-3.5) produces unsatisfactory output (you could detect by some heuristic or user feedback), you might then re-run on GPT-4.

- Ensure that fallback usage is communicated or logged, to monitor when it happens (it could indicate one provider hitting rate limits).

- **Cost Optimization Strategies:**

- **Use Cheaper Models When Appropriate:** Not every request needs the most expensive model. For example, for a quick summarization or less complex queries, use an OpenAI GPT-3.5 (which is, say, 1/10th the cost of GPT-4) or a local model via OpenRouter if available. You could implement a simple heuristic: small input or low complexity questions route to cheaper model.
- **Limit Max Tokens:** Set `maxTokens` (or equivalent parameter) to reasonable values for each model. Don't always use the maximum allowed, as that directly impacts cost. If you expect short answers, cap the tokens to, say, 500 instead of 2000.

- **Batch Requests:** If you have any opportunity to batch multiple tasks into one prompt (few-shot or multi-question answering), do it to reduce overhead per request. But be mindful not to degrade quality.
- **Monitor Usage:** Track how many calls and of what type are going to each provider. This can be as simple as logging or as complex as integrating with billing APIs. Knowing usage patterns helps in adjusting strategies (e.g., if 90% of requests are small but all going to GPT-4, you might save a lot by routing some to GPT-3.5).

- **Leverage OpenRouter or Proxy Services:** OpenRouter can route to multiple model APIs (OpenAI, etc.) possibly with unified billing or community models. Using OpenRouter's `@openrouter/ai-sdk-provider` might allow accessing certain open models or reduced-rate models through one API key. Just ensure the models meet your needs and consider reliability.

- **Rate Limiting & Backoff:** Each provider has its own rate limits. Implement a global rate limit for each provider's API usage in your app (to avoid hitting provider limits which can cause errors or bans). For instance, OpenAI might allow 3 requests/sec for certain endpoints by default – you can use a token bucket or queue in your code to ensure you don't exceed that. If a rate limit error (HTTP 429) occurs, catch it and apply an exponential backoff (wait and retry after a short delay). The AI SDK may not automatically retry on 429, so handle it at the application level.

- **User or Feature-based Provider Selection:** Consider allowing power-users or devs to choose provider (maybe internally via an admin setting or user preference). This can be useful for testing and cost control (e.g., user opts for "fast mode" vs "accurate mode" which correspond to different models). Even if not exposed to end user, having a configuration toggle to switch providers for certain features (like toggling default from OpenAI to Anthropic) can be a lifesaver if one API has issues – you can flip the switch without deploying code changes.

Capability/Provider Management Example:

Suppose we support OpenAI, Anthropic, and X.AI's Grok. We have a scenario: the user uploads an image and asks a question about it (needs vision), with a follow-up that requires a web search tool.

Our logic could be:

```
function chooseModel({ needsVision, needsTools, userPref }): LanguageModel {
  if (needsVision) {
    // only certain models support images:
    if (openRouterVisionAvailable) {
      return openRouter('openai/gpt-4-vision');  // via OpenRouter if configured
    } else {
      // fallback to xAI Grok if it supports images (hypothetically)
      return xai('grok-4-vision');
    }
  }
  if (needsTools) {
    // For heavy tool usage, GPT-4 or xAI Grok might be best
    if (userPref === 'grok' || !OPENAI_API_KEY) {
      return xai('grok-4');  // Grok specializes in tools and doesn't require
separate browsing API
    } else {
      return openai('gpt-4o');  // GPT-4 with function calling
    }
  }
  // Default path (no special needs):
  if (userPref === 'anthropic') {
    return anthropic('claude-2');
  }
  // default to cheapest capable (gpt-3.5) for normal queries
  return openai('gpt-3.5-turbo');
}
```

In this pseudocode, we consider capabilities and preferences to route to the appropriate model.

Common Issues & Solutions:

- **Inconsistent Output Formats:** When using multiple providers, the style or format of responses can differ. One model might be more verbose or format lists differently. To ensure consistent UX, you may need to post-process responses. For instance, normalize the format (strip extra prose if not needed, etc.). Another approach is to include format instructions in the prompt ("Answer in at most 3 sentences.") which most models will follow, albeit with varying success. Testing each provider with your typical prompts is key, and if one provider consistently deviates, handle that (either adjust prompt when using that provider or fix output via parsing code if possible).

- **Quality Disparity:** A cheaper model might sometimes give wrong or inferior answers. Solutions include:

- Implement a lightweight verification. For example, if the task is fact-based and you have a way to verify the answer (like cross-checking a database), do so. If the answer seems incorrect, automatically escalate to a better model.
- Collect user feedback (if user says the answer was bad via a thumbs-down UI, you could offer to retry with a better model).

- Use ensemble: in critical cases, you could call two models and compare answers or choose the one that seems more detailed – but this doubles cost, so do it sparingly (maybe for high-stakes queries).

- **Duplicate Charges Due to Retries/Fallback:** If you implement automatic fallback, be mindful that a failure on one provider followed by a call to another means you pay for both attempts (the failed one might not charge if it truly failed without processing tokens, but e.g., OpenAI might charge for partial prompt if it was a valid request that hit a limit after reading input). Use fallback only when necessary. If a certain type of request often fails on a cheaper model and then goes to expensive model, it might be more cost-efficient to route directly to the expensive model for that case (to avoid double-charging). Analyze your logs to identify such patterns.

- **Maintaining Many API Keys:** Keep an organized system for your API keys. Possibly use a key management service or at least a structured env config (`OPENAI_API_KEY`, `ANTHROPIC_API_KEY`, etc.). Ensure keys are valid and not expired (some keys or tokens for new services might expire or need refresh – e.g., if using Azure OpenAI or others, tokens might rotate). Write a small routine to test all keys on startup (like a trivial API call with each) to catch any misconfigurations early.

Cost Monitoring: It's good to implement logging of usage per provider. For example, log the model name, prompt length, response length for each request, and maybe the cost estimate. OpenAI's API returns usage info in the response (tokens used) which you can multiply by known rates [86]. Others like Anthropic also give token counts. By logging these, you can periodically calculate how much each provider is costing you and optimize. You might find, e.g., 80% of cost is going to GPT-4 for tasks that could be done by GPT-3.5 – which signals an opportunity to adjust your routing logic.

Troubleshooting Checklist:

1. **Provider-Specific Errors:** If calls to a provider suddenly fail (e.g., all OpenAI calls 503), have a mechanism to temporarily disable that provider (feature flag or automatic circuit breaker). You might already see exceptions – catch them and switch provider as needed. After the fact, check the provider's status (OpenAI/Anthropic have status pages) to understand outage or if your API key was revoked or out of credit.

2. **Unexpected Model Behavior:** If one model returns something very unexpected (like a one-word answer where a paragraph is expected, or hallucinations), verify that your prompt and parameters (temperature, etc.) are set appropriately for that model. Some models might need temperature tuning or different system prompts. You might maintain different prompt templates per provider if needed (e.g., Anthropic sometimes benefits from a more explicit instruction style due to its constitutional AI approach).

3. **Ensure Tools Compatibility:** If using the AI SDK's tools, verify each provider's ability to handle them:

4. OpenAI: ensure you pass function definitions to OpenAI (AI SDK does that internally for OpenAI models).
5. Anthropic: it doesn't have native function calling – the AI SDK handles tool calls by feeding the model a message and expecting a JSON reply. Check that Claude follows the format (the AI SDK tries to enforce this). If Claude often fails to call tools correctly, you might avoid using tools with it or switch to a model that does.

6. Custom providers: if using OpenRouter or others, ensure they support the needed features (OpenRouter might route function calls to OpenAI under the hood if you have that configured, etc.).

7. **Performance Issues:** If your response times vary greatly by provider (e.g., one provider is much slower), consider using that provider only when necessary. Also, parallelize requests when possible – although each user request likely uses one model at a time, if you had a case to call two models (like to compare answers), fire them in parallel and race which returns first, perhaps.

8. **Stay Updated on Model Changes:** Providers frequently update models (OpenAI may release GPT-4 vNext, etc.). New models might offer better price/performance. Keep an eye on announcements. E.g., if OpenAI releases a cheaper 16k context model and you currently avoid certain tasks due to token limits, that could change your strategy. Similarly, open-source models via OpenRouter might improve to a point where they handle more tasks at near-zero cost (aside from infra). Periodically re-evaluate: maybe run a set of benchmark queries on all available models to see quality vs cost tradeoffs, and adjust your defaults accordingly.

By actively managing your multi-provider setup with these strategies, you can ensure your app always picks the most suitable AI model for each task – optimizing for capability when needed and cost savings where possible – and maintain resilience against outages or changes in any single provider.

7. SWR 2.3.6 + TypeScript – Data Fetching Patterns and Caching

Current Status: SWR (stale-while-revalidate) is a React hook library for data fetching that simplifies caching and updating data in the UI. Version 2.x has improved TypeScript support and features like `useSWRImmutable` and `useSWRMutation`. With TypeScript, SWR can strongly type the data and error objects returned by the hook, but it relies on correct usage of generics. SWR's caching strategy (stale-while-revalidate) means it will serve cached data first and then update in background, making UI snappy. It also supports features like focus revalidation and network recovery revalidation by default.

Best Practices:

- **Type Safety in `useSWR`:** Always provide the generic type for the data (and error, if needed) when calling `useSWR`. For example:

```
interface Project { id: string; name: string; ... }
const { data: projects, error } = useSWR<Project[], Error>('/api/
projects', fetcher);
```

  This way, `projects` is typed as `Project[]` and `error` as `Error` (or a more specific error type if your fetcher throws something custom). If you forget the generic, TypeScript may infer `data` as `any`, losing safety. By specifying it, you get autocompletion and checks on usage of `projects` (e.g., `projects[0].name` will be known as string).

- **Key & Fetcher Patterns:** The SWR key can be a string or an array (or null to skip fetch). The fetcher function's arguments depend on the key. A good pattern:

- If your data is simply keyed by a static string or URL, use that string as key and a generic fetcher that uses `fetch`:

```
const fetcher = (url: string) => fetch(url).then(res => res.json());
const { data } = useSWR<Project[]>('/api/projects', fetcher);
```

Here SWR passes the key `'/api/projects'` to the fetcher, which expects a string.

- If you have parameters, use an array key for clarity:

```
const { data } = useSWR<Project>(
  projectId ? ['/api/projects', projectId] : null,
  ([url, id]) => fetch(`${url}/${id}`).then(res => res.json())
);
```

In this case, the key is `['/api/projects', projectId]`. SWR will pass that array to the fetcher, which destructures it. TypeScript can infer the tuple type, but you may need to help it. You can do:

```
const fetchProject: Fetcher<Project, [string, string]> = ([url, id]) =>
{ ... };
const { data } = useSWR(projectId ? ['/api/projects', projectId] : null,
fetchProject);
```

This explicitly types the fetcher as receiving a `[string, string]` tuple and returning `Project`. This pattern aligns the key and fetcher types.

- If using a function or more complex key, ensure the fetcher signature matches. A common oversight is using a function key and forgetting to handle it in the fetcher signature, leading TS to think the fetcher is the config.

- **Avoiding "any" in Fetcher:** If you see TS complaining about the fetcher type or mistaking the fetcher for config (there was a regression around 2.2.4) [87], explicitly type the fetcher or the `useSWR` call. Sometimes adding the generic parameters to `useSWR<Data, Error>` resolves ambiguity.

- **Caching Strategies:**

- Use **SWR's built-in cache** (which is global) to your advantage: data fetched on one page can be instantly available on another if the key is the same. For example, if you fetch `/api/user` on one component and navigate to another that also calls `useSWR('/api/user')`, the data is reused.
- **Immutable Data:** If data doesn't change often or you don't want revalidation, use `useSWRImmutable` (which is equivalent to `useSWR` with `revalidateIfStale: false, revalidateOnFocus: false, revalidateOnReconnect: false`). For instance, static reference data (like a list of country codes) can be fetched once and never revalidated.
- **Conditional Fetching:** Return `null` for the key (as shown above) to skip fetching when prerequisites aren't met (like `projectId` is not set). This is better than coding an internal check in fetcher – SWR will simply not run the fetcher until key is non-null.
- **Refreshing Data:** For frequently updating data (like a status), you can use `refreshInterval` in options, e.g., `useSWR('/api/status', fetcher, { refreshInterval: 5000 })` to poll every 5 seconds. SWR ensures it only fetches when component is mounted and will keep the interval if you navigate away (unless you pause focus).

- **Focus/Reconnect:** By default, SWR revalidates on window focus and network reconnect. This is usually desirable. If not (say in a heavy data app where refetch on focus is too much), you can disable with options.

- **Error Handling:** The `error` returned by `useSWR` is useful for UI (display error state). However, to avoid unhandled promise rejections in fetcher, ensure your fetcher throws an `Error` on non-OK HTTP responses. For example:

```
const fetcher = async (url: string) => {
  const res = await fetch(url);
  if (!res.ok) {
    const error = new Error('Error ' + res.status);
    // Optionally attach additional info:
    (error as any).info = await res.json();
    (error as any).status = res.status;
    throw error;
  }
  return res.json();
};
```

  This way, `error` in SWR will be set if the response is not OK. You can then use `error.status` or `error.info` in the UI. (TypeScript may need a custom type for error to know those properties exist.)

- **Mutations and Cache Updates:** If you modify data via some action (like a POST submission), use SWR's `mutate` function to update the cache. For example, after adding a new project, you might do:

```
await createProjectAPI(newProject);
mutate('/api/projects'); // revalidate the list of projects
```

  This triggers SWR to refetch `/api/projects`. You can also do optimistic updates: `mutate(key, newData, false)` to update cache without refetch, or use `useSWRMutation` for a structured way to handle writes.

Examples:

- *Type-safe SWR with tuple key:*

```
import useSWR, { Fetcher } from 'swr';
type UserProfile = { id: string; name: string; role: string };
const fetchUser: Fetcher<UserProfile, [string, string]> = ([url, id]) =>
  fetch(`${url}?id=${id}`).then(res => res.json());
```

```
const { data: user, error } = useSWR(userId ? ['/api/user', userId] :
null, fetchUser);
```

Here, if `userId` is not set, the key is `null` (no fetch). If set, the fetcher gets `[url, id]`. `user` will be typed as `UserProfile` with no extra effort because we defined Fetcher<UserProfile,...>.

- *Global configuration:* In `_app` or similar, you can wrap your app in `<SWRConfig value={{ fetcher, onErrorRetry: ... }}>` to set defaults. For example, to globally handle retries:

  ```
  <SWRConfig value={{
    fetcher: myFetcher,
    onErrorRetry: (error, key, config, revalidate, { retryCount }) => {
      if (error.status === 404) return; // don't retry on 404
      if (retryCount >= 3) return; // max 3 retries
      setTimeout(() => revalidate({ retryCount: retryCount + 1 }), 2000);
    }
  }}>
    <App />
  </SWRConfig>
  ```

  This ensures standardized behavior across uses of SWR.

Caching & SWR Specific Issues:

- **Too Aggressive Revalidation:** If you find SWR re-fetching too often (e.g., on every focus when not needed), adjust `revalidateOnFocus: false` for that hook or globally.
- **Cache Keys Colliding:** Make sure your keys are unique. E.g., if you had `useSWR('/api/project/'+id, ...)` ensure no other data uses the same URL format inadvertently. Keys should incorporate all relevant params. If using tuples, you're usually safe, but with string keys, be careful about constructing them (e.g., `'/api/search?query='+term` vs properly encoding term).
- **Memory leaks or large caches:** SWR by default keeps a cache in memory (and possibly in React's context). If you mount/unmount many different keys (like user navigates through 1000 items detail pages quickly), the cache might grow. In such rare cases, you might clear unused keys with `mutate(key, undefined)` when appropriate or configure a cache provider that purges (there are third-party cache providers for SWR that can set TTLs). For most apps, this isn't an issue, but it's something to note if memory usage grows.

Troubleshooting Checklist:

1. **Type Errors with SWR:** If TypeScript is complaining about the fetcher or data type, double-check the generic arguments. Usually adding `<DataType, ErrorType>` to `useSWR` call resolves it [88]. If using a tuple key, ensure your fetcher is typed to accept that tuple. The error messages can be a bit cryptic; refer to SWR's TypeScript docs [89] or discussions if stuck.

2. **Data Not Updating:** If you expect data to refresh but it isn't:

3. Confirm that the key is consistent. A common mistake is changing the key slightly (like using object references that are different on each render). Use stable primitives (string or tuple of serializable values).
4. If you manually mutated, check if you passed `revalidate: false` by accident or something. Use `mutate(key)` to force revalidation.
5. Check `SWRConfig` defaults – perhaps someone disabled `revalidateOnFocus` globally.

6. **Multiple Requests Issue:** If you see duplicate requests, it might be because you have the same `useSWR` in two components at the same time with slightly different keys, or using a function key that returns differently each time. Consolidate them or lift state if needed. SWR will de-dup concurrent requests to the exact same key automatically, so duplicates usually mean keys are not exactly the same.

7. **Error Handling UX:** Ensure your UI considers the three states: `!data && !error` (loading), `error` (show error message), and `data` (show content). If you forget the loading state, you might flash an error before data arrives or vice versa.

8. **Using SWR with Next.js 15 App Router:** If fetching in Server Components, you can't directly use SWR (as it's for client-side). You'd do an initial fetch with Next's `fetch()` in server and pass it as initialData via `useSWR` on client if needed. SWR's context in App Router may need `<SWRConfig>` placed appropriately (perhaps in a Client Component wrapper). In our case, likely we use SWR in client components only (which is fine). Just be mindful of hydration: if you pre-fetched some data via `generateStaticParams` or passed as prop, you might want to seed SWR's cache with it using `SWRConfig` or `mutate` so it doesn't refetch unnecessarily on load.

By following these best practices, you can make efficient use of SWR for client-side data fetching in a type-safe way, resulting in a responsive UI that gracefully handles caching, refetching, and errors.

8. Tailwind CSS v4 + Next.js 15 – Configuration and Performance

Current Status: Tailwind CSS v4.x is in use, which introduced a new engine (powered by Lightning CSS) for faster builds and added features like native CSS variables for theming, container queries, and zero-config mode [90]. Next.js 15 works seamlessly with Tailwind – Next.js automatically detects a `tailwind.config.js` and integrates it via PostCSS (using the built-in `@tailwindcss/postcss7` or now PostCSS 8 plugins). The focus is on ensuring your Tailwind config covers your content (especially in App Router), and leveraging new v4 features for theming and performance.

Best Practices:

- **Configuration Setup:** A basic `tailwind.config.js` for Next 15 App Router should specify the paths to all your content files so Tailwind can tree-shake unused styles. For example:

```
module.exports = {
  content: [
    "./app/**/*.{js,ts,jsx,tsx}",
    "./components/**/*.{js,ts,jsx,tsx}",
    "./pages/**/*.{js,ts,jsx,tsx}",  // if any
  ],
  theme: {
    extend: {
      // custom colors, spacing, etc.
    },
  },
  plugins: [
    require('@tailwindcss/forms'),
    // ... any tailwind plugins used
  ],
}
```

In v4, **zero-config mode** allows Tailwind to run without a config file by using defaults, but since we often have custom theme extensions, we provide one. Ensure **all directories** with styled components are listed in `content` (including any `node_modules` if you're pulling in templates with classes, though that’s rare). If you notice some styles not generated, likely the content globs missed those files.

- **PostCSS Integration:** Tailwind uses PostCSS. In Next, if you have a `postcss.config.js` ensure it includes Tailwind and Autoprefixer:

```
module.exports = {
  plugins: {
    'tailwindcss': {},
    'autoprefixer': {},
  }
}
```

Next.js will pick this up. If you installed Tailwind via `pnpm dlx tailwindcss init -p`, it likely created this. The **Lightning CSS engine** in Tailwind v4 improves build speeds drastically behind the scenes [90], so you might notice faster `next build` times compared to v3.

- **Theming with CSS Variables:** Tailwind v4 introduced native CSS variables for theme values, which facilitates things like dark mode or multiple themes. For example, enabling `darkMode: 'class'` or `'media'` in config is still the way to do dark mode. But with v4, you can also do something like:

```
:root {
  --tw-clr-primary: 17 120 248; /* your primary color in R G B */
}
```

```
.dark {
  --tw-clr-primary: 51 65 85;
}
```

And use it in classes like `text-primary` if you set `primary` color in config to use CSS variable (Tailwind docs show how). Essentially, Tailwind now can output dynamic themeable variables instead of static colors. Use this to implement a light/dark theme or user-customizable themes with minimal fuss. The `next-themes` library (which we have in dependencies) can toggle a `class` on `<html>` for dark mode. Ensure to configure Tailwind's `darkMode` accordingly (class or media).

- **Performance Considerations:** Tailwind v4's build engine is faster, but output CSS size can still be large if not careful:

- Make sure purge (content scanning) is working – in production builds, Tailwind should only include classes actually used in your content files. If you see your CSS containing a lot of utility classes that you don't use, the content paths might be wrong.
- Avoid dynamic class names that Tailwind cannot statically analyze. E.g., `className={"text-" + size}` – Tailwind won't catch that `text-lg` or `text-sm` is used. If you must do dynamic classes, list all possibilities in a `safeList` in config or construct them differently. Tailwind's purge is smart but not AI; it needs literal class names or certain patterns. You can use template literals with complete classes (like `` `text-${size}` `` is not safe, but `${size === 'lg' ? 'text-lg' : 'text-sm'}` is fine because both `'text-lg'` and `'text-sm'` appear in the source).

- Use `@apply` in CSS for repeated combos if needed, but sparingly (since it increases build time slightly). Tailwind is meant to avoid writing much CSS, but sometimes `@apply bg-primary text-white py-2 px-4` in a `.btn` class can be useful for readability.

- **New Features Highlights:**

- *Container Queries:* Tailwind v4 supports container queries via the `container` plugin (if enabled) or built-in utilities. This can be useful for responsive components. E.g., `@container` CSS at-rule usage becomes easier. If your design needs them, look up Tailwind's docs on container queries.
- *Text Shadows, Masks (v4.1):* Tailwind now includes utilities like `text-shadow` and masking utilities [91]. If those fit your design (maybe for fancy headers or image masks), use them instead of writing custom CSS.

- *Aspect Ratio & Typography:* Likely you might use official plugins (like `@tailwindcss/typography` if dealing with rich text). They work with v4 as usual. Always ensure plugin versions are compatible with tailwind v4 (most official ones updated).

- **Development Workflow:** Use `npm run dev` (or `pnpm dev`) and enjoy JIT updates of Tailwind – as you edit classes in markup, you should see styles appear. If something isn't styling, check the class name and config. In dev, Tailwind no longer prints the "purged X classes" as it did in v2 era, but you can still inspect output in browser dev tools (tailwind classes are usually not minified, which is helpful).

Example Tailwind Config for reference:

```
// tailwind.config.js
module.exports = {
  darkMode: 'class', // use 'class' based dark mode toggle
  content: [
    "./app/**/*.{js,ts,jsx,tsx}",
    "./components/**/*.{js,ts,jsx,tsx}",
    // etc.
  ],
  theme: {
    extend: {
      colors: {
        primary: 'rgb(var(--tw-clr-primary) / <alpha-value>)', /* using CSS
variable for primary color */
      },
    },
  },
  plugins: [
    require('@tailwindcss/forms'),   // example plugin for better form styles
    require('@tailwindcss/typography'), // if using prose classes
  ],
};
```

Troubleshooting Checklist:

1. **No Styles / Not Applied:** If you load your app and see unstyled content:
2. Verify that the Tailwind directives (`@tailwind base; @tailwind components; @tailwind utilities;`) are present in your global CSS (e.g., `globals.css` in Next 15). Next.js with Tailwind setup usually has these in a `globals.css` imported in `_app` or in a layout. If missing, add them so Tailwind CSS is actually generated.
3. Check the build output or console for errors. Sometimes a misconfigured `tailwind.config.js` can cause Tailwind to silently not include anything. For example, if `content` is empty or wrong, Tailwind might purge everything (since it thinks no files use any classes).
4. If using CSS Modules or other file types, ensure those are included in content globs or use the `safelist` option for classes that Tailwind might not see.
5. **Build Time Too Slow:** If you find `next build` is slow during “Generating CSS”, it might be due to enormous content globs (like scanning `node_modules` unnecessarily). Ensure `content` only covers necessary files. Tailwind v4’s engine is much faster (5x full, 100x incremental per Tailwind’s notes [90]), so slowness might indicate a misconfig. Also, huge safelist or regex in content can slow things.
6. **Purge Removing Needed Styles:** If you dynamically generate class names, you may find some styles missing in production. Solve by:
7. Using `safeList` in config: e.g.,

```
safelist: [
  { pattern: /^text-(red|green|blue)-\d{3}$/ }  // keep classes matching
this regex
]
```

or an array of specific class strings that you know are used but not statically present.

8. Or refactor to avoid generating classes dynamically.

9. **Compatibility Issues:** Tailwind v4 should be compatible with most things. If you use a third-party Tailwind plugin or UI library, check if they needed any updates for v4 (most pure-CSS ones are fine, but if one relied on internals, just ensure you have the latest). In our deps, things like `tailwind-merge` or `clsx` are fine as they just deal with class strings.

10. **CSS Variables and Theming:** If you attempt to implement theming with CSS variables and find it not working:

11. Remember Tailwind's color functions expect the variable to provide the R G B channels and an `<alpha-value>` placeholder as in config above. If you set a CSS variable differently (like including the alpha), adjust accordingly.
12. Ensure the `:root` or `.dark` class definitions for variables are actually present (either in a global CSS file or via a `<style jsx global>`).

13. If multiple themes: you could add classes like `.theme-light` and `.theme-dark` on the `html` and define variables for each, then toggle that class.

14. **Dark Mode Strategy:** If using `'class'` strategy, ensure you add/remove the `dark` class on `<html>` or `<body>` appropriately (the `next-themes` library does this for you via a script). Also, verify that you don't have conflicting `media` strategy anywhere. Sometimes developers accidentally combine, leading to confusion.

Overall, Tailwind CSS v4 with Next.js 15 should "just work" and provide a smooth styling experience with fast builds and easy theming. By fine-tuning the config and following the above practices, you'll maintain a maintainable, performant styling system for your app.

Sources:

- Next.js 15 Official Blog [1] [2] [11]
- Next.js Docs – Edge vs Node runtimes [4] [5] [10] [3]
- Next.js Docs – Memory Optimization Guide [6] [15]
- Vercel AI SDK Guide (Streaming) [26] [27]
- AI SDK Documentation (Core) [28] [92]
- Assistant UI Integration Example [36] [37] [29]
- AI SDK Providers Docs (xAI, etc.) [23] [24]
- Better Auth Documentation [48] [51] [14]

- Better Auth Q&A (Edge runtime issue) [13] [93]
- Drizzle ORM Q&A (conditional where) [68] [69]
- Drizzle Docs – Conditional Filters [66] [67]
- Drizzle Docs – pgvector Guide [64] [65]
- Vercel Guide – Connection Pooling [76] [77]
- Google Cloud Storage Signed URL Example [80]
- SWR Documentation & Discussions [88] [87]
- Tailwind CSS 4.0 Release Info [90] [94]

---

[1] [2] [11] [12] [16] Next.js 15 | Next.js
https://nextjs.org/blog/next-15

[3] [4] [5] [10] Rendering: Edge and Node.js Runtimes | Next.js
https://nextjs.org/docs/14/app/building-your-application/rendering/edge-and-nodejs-runtimes

[6] [7] [8] [9] [15] Guides: Memory Usage | Next.js
https://nextjs.org/docs/app/guides/memory-usage

[13] [14] [59] [93] A Node.js api is used which is not supported in the edge runtime - Better Auth
https://www.answeroverflow.com/m/1424763131086770367

[17] [18] [19] Foundations: Providers and Models
https://ai-sdk.dev/docs/foundations/providers-and-models

[20] AI SDK 5 - Vercel
https://vercel.com/blog/ai-sdk-5

[21] [25] [26] [27] Streaming responses from LLMs
https://vercel.com/guides/streaming-from-llm

[22] [28] [38] [39] [42] [44] [45] [86] [92] AI SDK Core: Generating Text
https://ai-sdk.dev/docs/ai-sdk-core/generating-text

[23] AI SDK Providers: xAI Grok
https://ai-sdk.dev/providers/ai-sdk-providers/xai

[24] https://console.groq.com/llms-full.txt
https://console.groq.com/llms-full.txt

[29] [36] [37] AI SDK v5 | assistant-ui
https://www.assistant-ui.com/docs/runtimes/ai-sdk/use-chat

[30] [31] AI SDK Core: Tool Calling
https://ai-sdk.dev/docs/ai-sdk-core/tools-and-tool-calling

[32] [33] [34] [35] AI SDK Core: Model Context Protocol (MCP) Tools
https://ai-sdk.dev/docs/ai-sdk-core/mcp-tools

[40] Advanced: Stopping Streams - AI SDK
https://ai-sdk.dev/docs/advanced/stopping-streams

[41] streamText onFinish returns responseMessage with id = "" #8305
https://github.com/vercel/ai/issues/8305

[43] AI-SDK not flushing traces to Langfuse Clouad in Nodjs backend ...
https://github.com/langfuse/langfuse/issues/5416

[46] jabirdev/nextjs-better-auth: A Next.js 15 starter template ... - GitHub
https://github.com/jabirdev/nextjs-better-auth

[47] Better Auth Starter – A Next.js 15 authentication boilerplate ... - GitHub
https://github.com/devAaus/better-auth

[48] [51] [52] [55] [56] [57] [58] [60] [61] [62] [63] Next.js integration | Better Auth
https://www.better-auth.com/docs/integrations/next

[49] [50] [53] [54] [81] [82] [83] [84] [85] OAuth | Better Auth
https://www.better-auth.com/docs/concepts/oauth

[64] [65] [72] [73] [74] [75] Drizzle ORM - Vector similarity search with pgvector extension
https://orm.drizzle.team/docs/guides/vector-similarity-search

[66] [67] Drizzle ORM - Conditional filters in query
https://orm.drizzle.team/docs/guides/conditional-filters-in-query

[68] [69] [70] [71] Running into some type errors with query.where - Drizzle Team
https://www.answeroverflow.com/m/1318073949258518569

[76] [77] [78] Connection Pooling with Vercel Functions
https://vercel.com/guides/connection-pooling-with-functions

[79] Managed Connection Pooling overview | Cloud SQL for PostgreSQL
https://cloud.google.com/sql/docs/postgres/managed-connection-pooling

[80] Create a GET-signed URL for an object using Cloud Storage ...
https://cloud.google.com/storage/docs/samples/storage-generate-signed-url-v4

[87] Regression in 2.2.4: Type issue with useSWR: fetcher mistaken for ...
https://github.com/vercel/swr/issues/2809

[88] Is there a typescript example of useSWR · vercel swr · Discussion #939
https://github.com/vercel/swr/discussions/939

[89] TypeScript - SWR
https://swr.vercel.app/docs/typescript

[90] [94] Tailwind CSS 4.0 Is Here: What's New and Why It Matters - Medium
https://medium.com/@Luna-Rojas/tailwind-css-4-0-is-here-whats-new-and-why-it-matters-c872d1cb77bb

[91] Tailwind CSS v4.1: Text shadows, masks, and tons more
https://tailwindcss.com/blog/tailwindcss-v4-1