ChatGPT

# Optimizing Next.js 15 Builds on Vercel Under High Memory Usage

Building a **Next.js 15.3.2** project with numerous dependencies (e.g. Vercel's AI SDK providers, OpenAI/Google SDKs, vector stores, UI libraries, etc.) can strain Vercel's build memory and slow down or even fail deployments. This guide breaks down why high memory usage bloats builds and provides practical strategies to optimize build performance. We'll cover reducing bundle size, managing large AI SDKs, adjusting Vercel build settings, and preventing memory-related issues (like type-check bottlenecks or runtime config errors).

## Understanding Memory Bottlenecks in Vercel Builds

Each Vercel build runs in a container with a fixed memory limit (≈8 GB for standard builds) [1]. If your project approaches this limit, builds can slow down (due to garbage collection or swapping) or be terminated (OOM/SIGKILL errors). Common causes of excessive build memory usage include **too many/large dependencies**, **large import trees**, and inefficient code or data processing [2]. In our case, the dependency list is very large (over 1200 packages were installed, including multiple AI providers and UI libraries), which inherently increases memory usage during bundling and type-checking.

**Why memory affects build speed:** The Next.js build involves compiling your code (using Webpack/Turbopack), optimizing it (tree-shaking, minifying, etc.), and running **static generation** or **type checks**. More code and heavier libraries mean more work and memory needed for these steps. For example, transpiling and code splitting scale with application and dependency size [3]. If memory use gets high, the process may slow due to GC, and in worst cases Vercel might kill the build. Ensuring we stay within memory limits will keep builds fast and reliable.

## Factors Contributing to Slow, Bloated Builds

Let's identify architectural factors in our Next.js app that can inflate memory usage or build times:

- **Excessive Dependencies & Large Import Trees:** The project pulls in many libraries (e.g. multiple `@ai-sdk/*` providers, `googleapis`, `@aws-sdk`, Radix UI components, etc.). Each dependency adds to the bundle and must be parsed/optimized. Large libraries that export many modules can especially hurt performance if not tree-shaken [4]. For instance, including **multiple AI SDK providers** means a big chunk of code for each (OpenAI, Google, Anthropic, Groq, etc.) ends up in the build if imported directly. A deep import tree (modules importing other large modules) further increases the work Webpack must do.

- **AI SDKs and AI Libraries:** AI-focused packages can be heavy. Vercel's `ai` **SDK v5** and its `@ai-sdk/*` provider packages bring in complex logic and type definitions. In fact, issues have been noted where AI SDK v5's complex TypeScript types (especially combined with Zod schemas) **bog down**

**compilation** or type-checking [5] [6]. In our dependency list, we see **Zod 4.1.12** and AI SDK – a known combination that can cause "excessively deep" type instantiations, leading to slow or hanging type checks. The OpenAI and Google client libs can also be sizable (e.g. `googleapis` and `@google-cloud/*` bring in many sub-dependencies). All these increase bundle size and memory usage if included indiscriminately.

- **Runtime Configuration and Environment Variables:** Environment or config code can inadvertently execute during build, causing failures or added overhead. For example, our build log shows an error during "Collecting page data" where `DATABASE_URL` was invalid [7]. Likely, some server code (an API route or `getStaticProps`) tried to use an env var at build-time. This not only caused a **build failure** but indicates how runtime config can affect builds. Large config objects or loading environment-specific data during build can also consume memory. It's crucial to separate build-time and runtime concerns for configurations.

- **Vector Store or Data Initialization:** If the project preloads AI models or vector indexes at build (for example, generating embeddings or loading a vector store in `getStaticProps`), it can be extremely memory-intensive. Bulk data processing (reading big files, images, or datasets) during build will spike memory usage [8]. Ensure that heavy AI data tasks (e.g. embedding a large document corpus) are done at runtime or offline, not as part of the Next build step.

- **Type Checking and Linting Overhead:** By default, Next's build runs ESLint and TypeScript checks after compilation. In a large codebase with complex types (e.g. generics from AI SDK or Zod), this phase can consume a lot of memory and time [9]. Our build spent ~45s on "Linting and checking validity of types" – not catastrophic, but it adds up. In worst cases, TS may run out of memory or hang on intricate types. We need to ensure type-checking doesn't become a bottleneck (there are ways to alleviate this, discussed below).

- **Source Maps and Caching:** Generating source maps uses extra memory during build. Webpack/Next will by default keep a cache of modules in memory for speed, which increases memory footprint [10] [11]. If we're hitting memory limits, these otherwise useful features (cache, source-maps) might need tweaking.

- **Edge Functions Constraints:** If any Next.js functions are deployed to the Edge runtime, note that edge environments have stricter memory and execution time limits. While this primarily affects runtime, it has build implications too: Edge-compatible bundles must be smaller (no Node APIs, limited size). In our case, most heavy AI stuff likely runs in serverless Node functions (since OpenAI/Google SDKs require Node), but it's worth noting: don't accidentally push large logic to an Edge function – it may exceed the edge memory or size limits. Vercel's limit for Node functions is up to **2 GB** memory on Hobby (4 GB on Pro) and ~250 MB bundle size [12] [13]. Our dependency list is huge, but splitting into multiple functions (per route) and pruning unused code helps avoid hitting those limits.

With these pain points identified, let's move on to **optimization strategies**.

## 1. Reduce and Tree-Shake Dependencies

**Audit your bundle**: Start by analyzing which packages contribute most to your bundle. Use Next.js' Bundle Analyzer (`@next/bundle-analyzer`) or Webpack Bundle Analyzer to visualize size [3]. Identify large libraries or duplicate inclusions. For example, if `googleapis` or `@aws-sdk` appear heavily in your server bundle and you only use a small part of them, that's a target for optimization.

**Prune unused packages**: Remove any dependency not actively used in your code. It's easy for AI projects to accumulate SDKs for providers that end up not being used. For instance, if you included `@ai-sdk/groq` or `@ai-sdk/xai` but aren't actively using those providers, remove them to shed weight. Every unused provider or library you drop directly frees memory and speeds up builds [14] [15].

**Tree-shaking and import patterns**: Ensure you import only what you need from large modules. Many util libraries (Lodash, date-fns, Radix UI, etc.) are tree-shakable – but only if you use import syntax that the bundler can shake. For example, import specific components (`import { Dialog } from '@radix-ui/react-dialog'`) rather than importing everything. Next.js 15 introduced an `experimental.optimizePackageImports` option which can be a game-changer for huge libraries that export tons of modules. By adding a package to this list, you let Next/Webpack auto-split out only the modules your code actually uses [16]. Next.js already optimizes several libraries by default (like `lodash-es`, `date-fns`, `lucide-react`, etc.) – see if any of your heavy deps are in that list (e.g. `lucide-react` is optimized out of the box) [17]. You can add others manually. For instance, if you find `googleapis` or some AI SDK submodule is pulling in too much, consider:

```
// next.config.mjs
export default {
  experimental: {
    optimizePackageImports: ['googleapis', '@aws-sdk/client-s3'],
  },
}
```

*(Above is an example – verify that these packages are compatible with this feature.)*

**Code-splitting via dynamic import**: For code that's only needed in certain cases, use `import()` to load it on demand. In a Next App Router project, you might have an AI-related route or page – that code doesn't need to bloat other pages. For example, if an admin page uses the Google SDK, import it dynamically in the server action or handler for that route:

```
// Inside your API route or server action:
const { GoogleAI } = await import('@ai-sdk/google');
// use GoogleAI...
```

This way, Webpack knows it can bundle `@ai-sdk/google` into a separate chunk, only loaded when that route is invoked. Dynamic imports won't reduce **build** memory usage dramatically (the code still gets built), but they **reduce each function's bundle size** and ensure you're not initializing heavy modules

unnecessarily. Smaller function bundles mean lower memory overhead at runtime and less risk of hitting Vercel's size limits [13].

**Find lighter alternatives**: Sometimes, large libraries can be replaced with smaller ones. For instance, if `googleapis` (144.0.0) is huge and you only need a few REST calls, consider using fetch calls to Google's API directly or a slim SDK. Vercel's docs suggest using tools like **bundlephobia** to compare package sizes and find lighter alternatives [13]. Similarly, if an AI SDK provider is too heavy and you only call an external API, you might call the REST API directly instead of loading the whole SDK (trade-offs: you lose SDK conveniences but gain performance).

**Deduplicate versions**: Check if multiple versions of the same library are present (e.g. multiple `esbuild` or others were in the log). PNPM usually deduplicates, but sometimes different sub-dependencies bring their own versions. Running `pnpm why <pkg>` or `pnpm ls` can reveal duplicates. Use `pnpm dedupe` to consolidate where possible [18]. Fewer unique packages = less to build and less memory used.

## 2. Optimize Large AI SDK Usage

When using Vercel's **AI SDK (v5)** and related packages, be mindful of how to integrate them efficiently:

- **Update and Patch Known Issues**: Make sure you're using the latest patch versions of the AI SDK and related tools. The **AI SDK v5** had known compilation issues in early versions when combined with certain libraries (like Zod). According to Vercel's AI SDK troubleshooting, upgrading **Zod to 4.1.8+** resolved a TypeScript performance bug where Zod's types were loading twice and causing huge overhead [19]. In your dependencies, Zod is at 4.1.12 (which is ≥4.1.8, so that should include the fix). Additionally, they suggest using `moduleResolution: "nodenext"` in `tsconfig.json` as a workaround if you can't upgrade Zod [20]. Since you already have a compatible Zod, ensure your TS config is also up-to-date (TypeScript 5.9 should be fine, maybe ensure `"moduleResolution": "bundler"` or `"nodenext"` as needed for ESM packages to avoid type duplication).

- **Limit Provider Packages**: Each `@ai-sdk/*` provider you include adds its own code. If you don't need all of them in the final product, remove or disable what you can. For example, if you included both `@ai-sdk/google` and `@google/genai`, consider if both are necessary or if one can be removed in favor of a single approach. Sometimes the AI SDK's **OpenAI-compatible provider** could cover multiple backends via a uniform API, potentially reducing the need for separate heavy SDKs (just an idea if it fits your use-case).

- **Isolate AI code**: If possible, run AI-related code in **serverless functions away from the Next.js build**. For instance, you could deploy a separate API (or use Vercel Edge Functions or Vercel's AI Gateway) for heavy AI tasks. That way, your Next.js app's build doesn't have to bundle all the AI SDK code at all. This is a bigger architectural change, but worth mentioning: offloading AI logic to an external service could drastically cut your build bundle size and memory use. Vercel's AI Gateway, for example, can handle calling model providers, so your app calls a lightweight endpoint instead of bundling each provider's SDK.

- **Streaming and Edge**: If you're using the AI SDK's streaming features (`useChat` or similar hooks from `@ai-sdk/react`), note that these often use **Server-Sent Events (SSE)** or other streaming

under the hood. Ensure that you aren't polyfilling or pulling in Node streams on the client side unnecessarily. Also, streaming responses on Vercel's Edge runtime must start within 25 seconds and can stream up to 300s [21]. This is more of a runtime performance note: if you *are* using Edge functions for streaming, be sure to stay within these limits, or consider using Node runtime where you have more memory/time (especially given AI streams can be long-running). In short, for memory: prefer Node serverless functions for AI tasks over Edge, unless latency demands edge – Node functions have higher memory ceilings (2–4 GB vs. much lower per edge instance).

- **Use Experimental Webpack optimizations**: Next.js 15 includes an experimental flag specifically aimed at **reducing build memory** for projects like yours. You've already enabled `webpackMemoryOptimizations` (it shows as enabled in the build log) – this will tweak Webpack's behavior to use less peak memory (at the cost of slightly longer compilation) [22]. Keep this on – it's low-risk and directly addresses memory bloat.

- **Leverage Partial Bundling**: If certain AI functionality is only used in specific routes, consider marking those routes as **optional or on-demand**. Next 15 App Router doesn't pre-build API routes, but for pages, you can use dynamic routes or `generateStaticParams` carefully so that not too much is pre-rendered. The idea is to avoid doing heavy work for pages that might not need it at build.

## 3. Tune Next.js Build Settings

Next.js and TypeScript offer several configuration tweaks to manage build-time resource usage:

- **Disable Type Checking and Linting in the Build:** For production builds on Vercel, you might skip TS type checking and ESLint, especially since you likely run them in CI or locally. The Next.js docs note that type checking and linting "may require a lot of memory" in large projects and provide settings to disable them during the build [9]. You can add the following to `next.config.js` (or `.mjs`):

```
// next.config.js
module.exports = {
  eslint: {
    ignoreDuringBuilds: true,
  },
  typescript: {
    ignoreBuildErrors: true,
  },
};
```

This will skip ESLint and allow production builds despite type errors [23]. ⚠ **Note:** Use this with caution – it's important to still run linting and type checks elsewhere (e.g. GitHub Actions or a pre-deploy step) to catch issues. But skipping them on Vercel can save both time and memory. In our build, the "Linting and checking

types" step took ~45s; disabling it could cut nearly a minute and avoid any chance of TS OOM during that phase.

- **Use `skipLibCheck` in tsconfig:** In your `tsconfig.json`, set `"skipLibCheck": true` under `compilerOptions`. This tells TypeScript to skip type-checking of declaration files in `node_modules`. It can vastly reduce type-checking work and memory, without impacting your app's correctness (it only skips verifying types of your libraries). This is a common trick for large TS projects to speed up builds and avoid TS memory issues.

- **Source Maps Off in Production:** If you don't need browser-source maps in your deployed app, turn them off to save build work. In Next config, set:

```
productionBrowserSourceMaps: false,
experimental: { serverSourceMaps: false }
```

This prevents generating source maps for both client and server bundles, which **reduces memory and disk usage during build** [11]. (Only do this if you're okay not having stack traces mapping to your original code in production logs, or use an error monitoring service that handles sourcemaps separately.)

- **Consider Webpack Build Cache settings:** By default Next/Webpack uses a filesystem cache to speed up rebuilds, which also uses memory. If you have extremely constrained memory, you could force Webpack to use in-memory cache or none. The Next docs show an example of disabling persistent caching [10]. However, since Vercel builds start fresh (unless you use the remote cache with Turborepo), caching might not be very effective anyway. Tweaking this is usually not needed unless you've identified cache as a problem.

- **Use the Webpack Build Worker:** This is enabled by default since Next 14 for projects without custom webpack config [24]. It runs webpack in a separate thread, reducing main process memory. Ensure you haven't disabled it. If you have a custom webpack config, add `experimental: { webpackBuildWorker: true }` to re-enable it. This can shave off peak memory usage during compilation.

- **Monitor Memory During Build:** Next.js has a flag `--experimental-debug-memory-usage` you can use locally to profile where memory spikes [25]. If you run `NODE_OPTIONS='--max-old-space-size=6144' next build --experimental-debug-memory-usage` on your dev machine, you'll get logs of heap usage throughout the build. This can pinpoint if a particular step (e.g. "collecting page data" or a particular webpack bundle) is using unusual memory. On Vercel, you can also check the **Build Diagnostics** in the project's Observability tab to see memory usage of each build [26]. This might confirm improvements as you implement these optimizations.

- **Increase Node Memory if Needed:** If despite optimizations, you still flirt with the 8 GB limit, you can increase Node's heap during build. Vercel allows setting the `NODE_OPTIONS="--max-old-space-size=..."` env var for builds. For example, in your Project Settings > Environment Variables, add `NODE_OPTIONS` = `--max-old-space-size=6144` (which allocates ~6 GB to the

Node heap) and mark it as **"Preview + Production"**. Next's docs recommend this for build OOMs [27]. Be careful not to set it absurdly high (e.g. 8192) on a 8 GB container – leave some room for overhead. But 6GB is a common tweak. This doesn't reduce usage, but might prevent OOM crashes by giving GC more room to work.

- **Upgrade Your Plan or Use Enhanced Builds:** As a last resort, if builds are still slow, Vercel Pro/Enterprise can double the memory and CPU (16 GB / 8 vCPU) for builds via on-demand "Enhanced Builds" [28]. More memory can reduce build time since the process isn't starved [29]. This is a brute-force solution – ideally try the optimizations first, but it's good to know.

## 4. Handle Runtime Configs and Env Variables Safely

As seen in the latest build log, an **invalid** `DATABASE_URL` **environment variable** caused a build failure. This highlights the importance of managing runtime configuration in a way that doesn't sabotage the build:

- **Provide all required env vars in Vercel**: Double-check that `DATABASE_URL` (and any API keys or config vars) are set in your Vercel project settings for the appropriate environment. An invalid URL suggests it might be blank or malformed. Vercel's official guide recommends using the dashboard's **Environment Variables** feature to manage secrets and configs [30] – ensure these are correctly configured for Preview/Production deployments.

- **Avoid using env-dependent code at build time**: If certain code should only run at runtime (e.g. connecting to a database using `DATABASE_URL`), guard it so it doesn't execute during the static build. In the App Router, most server code won't run until invoked, but if you have something like:

```
// Bad: top-level initialization
const db = connectToDatabase(process.env.DATABASE_URL);
```

placed at module top in a route handler, Next might try to evaluate it when precomputing the route. Instead, initialize inside the handler function or use lazy loading. You can also mark a route as `dynamic = "force-dynamic"` (export const dynamic) to ensure no build-time pre-render. The error occurred during "Collecting page data" for `/api/chat/estimator` – since API routes aren't pre-rendered, perhaps the code was actually running as part of an SSG page. If that's the case, you might be doing something like calling that API route from `getStaticProps` or similar. Re-evaluate if that needs to be done at build – maybe it can be done on demand.

- **Use try/catch for config**: If you have a config validation (the error looks like a thrown exception for invalid env), you might adjust it to not throw during build or provide a dummy fallback in build env. Some projects use dummy env values in Preview builds to satisfy build-time requirements and then rely on real ones at runtime. For instance, you could detect if `process.env.VERCEL` is present (Vercel's env indicator) and if so, and if `DATABASE_URL` is empty, set a placeholder or skip the connection. That way, the build won't error out. The message was clearly custom (likely from your code) – consider making it a warning during build instead of a fatal error.

- **Leverage Next.js Runtime Config if needed**: Next.js (Pages Router) had `publicRuntimeConfig` and `serverRuntimeConfig` in next.config.js for injecting config. In App Router, it's less used – environment variables are the recommended approach. But the principle remains: don't hard-code secrets or environment-specific values into the bundle; use `process.env.X` so that at build time it can be replaced or left for runtime. Next will inline env vars prefixed with `NEXT_PUBLIC_` into client bundle. Server-side `process.env` is accessed at runtime for per-request usage, so generally it's fine. Just ensure those env vars exist when you need them.

- **Avoid large secrets or data in env**: Sometimes people store big blobs (like JSON) in env vars – this can bloat build and memory if Next inlines them. Keep env vars to small config strings/URLs. For large data, use external files or databases.

## 5. Managing Vector Stores and AI Data at Scale

If your project uses a **vector database or embeddings** for AI memory, consider these tips to keep builds smooth:

- **Do not embed large vectors in the build**: If you have a static JSON of embeddings or a local vector store file, don't import it into your Next.js code. That would dramatically increase bundle size and memory usage. Instead, store such data externally (in a database, or even as a static file in public/ and fetch it at runtime, or generate it on the server on first run). Next builds should remain lightweight.

- **Generate heavy data on demand**: For example, if you need to compute embeddings for user queries (like an AI chatbot memory), do it at runtime when needed, rather than precomputing all during build. Build should be mostly compiling code, not running AI computations.

- **Use streaming efficiently**: For AI streaming responses, send data in chunks to avoid buffering huge responses in memory. This is more of a runtime concern, but it complements build optimizations: at runtime on Vercel Functions, streaming I/O doesn't count against CPU time [31], so you can stream large results without timing out – but be mindful of memory if you accumulate too much in memory at once.

- **Monitor production memory**: After deploying, use Vercel's Observability or custom logging to ensure your functions (especially AI-heavy ones) aren't hitting memory limits at runtime. High runtime memory won't directly slow builds, but if a function is too large (in code or data), Vercel might refuse to deploy it (bundle too big) or run it. The bundle size limit is 250 MB uncompressed for functions [12]. Our dependency list is large, but likely each API route's bundle will be under that if code-splitting is working. Just keep an eye on it as you add more.

## 6. Additional Build Pipeline Tips

Finally, a few miscellaneous tips to trim build times:

- **Cache dependencies between builds**: If using a monorepo or custom CI before Vercel, cache the `pnpm store` or `node_modules`. Vercel generally does this automatically for dependencies, but

since we saw “Previous build caches not available”, you might want to ensure cache is enabled (on Vercel, it caches by lockfile – a new lockfile hash invalidates cache). In a CI like GitHub Actions, caching `~/.pnpm-store` can cut install time.

- **Leverage Incremental Static Regeneration (ISR)**: If you have static pages that take long to build due to data fetching, consider using ISR (`revalidate` in Next 13+ App Router) so that you don’t block the build on generating all content. This reduces build memory/time by deferring work to after deployment.

- **Split monolithic tasks**: Instead of one giant Next app, some teams split out heavy functionalities into separate services. For example, an AI vector indexing service separate from the Next frontend. This isn’t always feasible, but microservices can keep each build focused and lighter.

- **Keep Next.js updated**: The Next.js team continuously improves performance. You’re on 15.3.2 which is cutting edge; keep an eye on release notes for any build optimizations. For instance, a memory leak on Edge runtime was fixed in 14.1.3 [32] – always good to have those fixes. Also, watch for **Rspack** or **Turbopack** becoming viable for production builds – these new bundlers aim for faster builds with different performance characteristics. As of 15.x, Webpack is still default for prod, but this could change.

## Conclusion

By trimming unnecessary dependencies, lazy-loading heavy modules, and adjusting Next/Vercel settings, you can significantly reduce memory usage and build times. In summary:

- **Minimize your bundle** by removing dead weight and ensuring only the code you need gets bundled (use tree-shaking, dynamic imports, and Next’s optimizePackageImports [16]).
- **Manage large SDKs** by updating to fix known issues (e.g. Zod with AI SDK [19]), and only include providers/features you truly need. Consider externalizing some AI logic to keep the Next app lean.
- **Tune the build process** with Next config flags – disable costly checks (run them elsewhere), turn on memory optimizations [22], and disable source maps if not needed. Bump Node’s memory or upgrade the plan if absolutely necessary for stability [27].
- **Handle runtime configs smartly** so that env vars or secrets don’t trip up your build. Set them correctly on Vercel and guard their usage so build can succeed with placeholder values [7].
- **Monitor and iterate** – use bundle analysis and Vercel’s diagnostics to see the impact of changes. It may take a few iterations to strike the right balance, but each improvement (no matter how small) can compound to big reductions in build time.

With these strategies, you should see your Vercel deployments become faster and more reliable, even as your Next.js app leverages heavy AI capabilities. Happy optimizing!

**Sources:**


- Vercel Guide – *Troubleshooting Builds Failing with OOM Errors* (Justin Vitale, 2025) [1] [3] [27]
- Next.js Official Docs – *Optimizing Memory Usage* [9] [33], *Optimize Package Imports* [16]
- Vercel Docs – *Function Memory and Bundle Limits* [13] [12]
- Vercel AI SDK Docs – *TypeScript Performance Issues (Zod + AI SDK)* [19] [20]

- Medium – *Optimizing Next.js Build Times* (F.Maria) [30] (environment variables on Vercel)
- GitHub Issue – *Next.js Dev Server Hangs with AI SDK v5* (community investigation) [5]

---


[1] [2] [3] [7] [8] [14] [15] [18] [26] [27] [28] [29] Troubleshooting Builds Failing with SIGKILL or Out of Memory Errors
https://vercel.com/guides/troubleshooting-sigkill-out-of-memory-errors

[4] [16] [17] next.config.js Options: optimizePackageImports | Next.js
https://nextjs.org/docs/pages/api-reference/config/next-config-js/optimizePackageImports

[5] Next.js Dev Server Hangs During Compilation with AI SDK v5 · Issue #9273 · vercel/ai · GitHub
https://github.com/vercel/ai/issues/9273

[6] [19] [20] Troubleshooting: TypeScript performance issues with Zod and AI SDK 5
https://ai-sdk.dev/docs/troubleshooting/typescript-performance-zod

[9] [10] [11] [22] [23] [24] [25] [32] [33] Guides: Memory Usage | Next.js
https://nextjs.org/docs/app/guides/memory-usage

[12] [13] [21] [31] Vercel Functions Limits
https://vercel.com/docs/functions/limitations

[30] Optimizing Build Times and Deployments in Next.js Projects | by Farihatul Maria | Medium
https://medium.com/@farihatulmaria/optimizing-build-times-and-deployments-in-next-js-projects-04183b0a9f3c