EcomLabs

STEP-BY-STEP GUIDE

Next.js high memory usage: measure and decide

How to read your Next.js server's memory, crawl your sitemap with and without images, decide whether images.unoptimized suits you, and roll it back if it does not. With the dependency-free crawl script I used.

FOR	Self-hosted Next.js on next start (Railway, VPS, Docker)
TESTED WITH	Node 22.22.3 · PowerShell 5.1 · Windows 11
UPDATED	November 2, 2026
DOCS	nextjs.org/docs/15/app/api-reference/components/image
ARTICLE	ecomlabs.dev/g/ram-en
AUTHOR	Luis Morales · @luismoralesdev

Contents

Step 01	Where to read memory
Step 02	Crawl the sitemap
Step 03	Decide if it suits you
Step 04	The change, and how to roll it back
When you finish	Checklist

STEP 01

Where to read memory

First, the baseline. On ecomlabs.dev (Next.js 15.5 on Railway) I went from a 0.99 GB average to 0.29 GB after crawling all 276 sitemap pages; without a measurement before and after, you will not know whether the change helped.

- **On Railway:** click the service on the project canvas and open the **Metrics** tab. The memory graph is there, with up to 30 days of data. With replicas, the Sum view adds them up and the Replica view splits them out. Write down a one-week average before you change anything.
- **On Linux or macOS:** `ps -o rss= -p <pid>` prints the process's resident memory in KiB.
- **On Windows (PowerShell):** `(Get-Process -Id <pid>).WorkingSet64` gives the working set in bytes. It is not identical to a Linux RSS, so always compare on the same machine.

To read it every 2 seconds while the crawl runs, this PowerShell loop is enough (against my test server it printed 62, 62 and 61 MB). The article includes `sample-rss.mjs`, which does the same with `ps` on Linux and macOS and keeps the peak:

```
$id = 95172
1..3 | ForEach-Object { "{0:N0} MB" -f ((Get-Process -Id $id).WorkingSet64 / 1MB); Start-Sleep 2 }
```

STEP 02

Crawl the sitemap

Save `crawl-sitemap.mjs`. It swaps the host of every `<loc>` for the sitemap's own (a local build's sitemap lists your production domain), and with `--images` it also requests every same-host `src` and `srcset` URL, which is what puts the optimizer to work:

```
// Requests every <loc> in a sitemap, on the sitemap's own host, and prints a summary.
// usage: node crawl-sitemap.mjs <sitemap-url> [--concurrency 1-4] [--images]
import { pathToFileURL } from "node:url";

const decode = (s) => s.replaceAll("&amp;", "&");

export function pageUrls(xml, sitemapUrl) {
  const host = new URL(sitemapUrl);
  // A local build's sitemap still lists the production domain: keep the path, swap the host.
  return [...xml.matchAll(/<loc>\s*([^<]+?)\s*<\/loc>/g)].map((m) => {
    const u = new URL(decode(m[1]), host);
    return new URL(u.pathname + u.search, host.origin).href;
  });
}

export function imageUrls(html, pageUrl) {
  const found = [];
  for (const [tag] of html.matchAll(/<img\b[^>]*>/gi)) {
```

```
    for (const [, name, value] of tag.matchAll(/\s(src|srcset)="([^"]*)"/gi)) {
      const list = name.toLowerCase() === "srcset" ? value.split(",").map((c) => c.trim().split(/\s+/)[0]) :
[value];
      for (const v of list) if (v && !v.startsWith("data:")) found.push(new URL(decode(v), pageUrl).href);
    }
  }
  // Same host only: the crawl measures your server, never someone else's CDN.
  return found.filter((u) => new URL(u).origin === new URL(pageUrl).origin);
}

export async function crawl(sitemapUrl, { concurrency = 1, images = false, log = () => {} } = {}) {
  const started = performance.now();
  const res = await fetch(sitemapUrl);
  if (!res.ok) throw new Error(`sitemap: HTTP ${res.status}`);
  const pages = pageUrls(await res.text(), sitemapUrl);
  if (pages.length === 0) throw new Error("no <loc> found (a sitemap index is not supported)");

  const statuses = {};
  const seen = new Set();
  let bytes = 0;
  const get = async (url) => {
    try {
      const r = await fetch(url, { signal: AbortSignal.timeout(30_000) });
      const body = Buffer.from(await r.arrayBuffer());
      bytes += body.length;
      statuses[r.status] = (statuses[r.status] ?? 0) + 1;
      if (r.status >= 400) log(`${r.status} ${url}`);
      return { r, body };
    } catch (err) {
      statuses.error = (statuses.error ?? 0) + 1;
      log(`error ${url}: ${err.message}`);
      return null;
    }
  };

  let next = 0;
  const worker = async () => {
    while (next < pages.length) {
      const url = pages[next++];
      const got = await get(url);
      if (!images || !got?.r.headers.get("content-type")?.includes("text/html")) continue;
      for (const img of imageUrls(got.body.toString("utf8"), url)) {
        if (seen.has(img)) continue;
        seen.add(img);
        await get(img);
      }
    }
  };
  await Promise.all(Array.from({ length: Math.min(concurrency, pages.length) }, worker));
  return { pages: pages.length, images: seen.size, statuses, bytes, ms: Math.round(performance.now() -
started) };
```

```
}

if (import.meta.url === pathToFileURL(process.argv[1]).href) {
  const args = process.argv.slice(2);
  const url = args.find((a) => /^https?:\/\//.test(a));
  const i = args.indexOf("--concurrency");
  const concurrency = Math.min(4, Math.max(1, Number(i >= 0 ? args[i + 1] : 1) || 1));
  if (!url) {
    console.error("usage: node crawl-sitemap.mjs <sitemap-url> [--concurrency 1-4] [--images]");
    process.exit(2);
  }
  const s = await crawl(url, { concurrency, images: args.includes("--images"), log: console.log });
  const codes = Object.entries(s.statuses).map(([code, n]) => `${code} x ${n}`).join(", ");
  console.log(`pages:     ${s.pages}`);
  console.log(`images:    ${s.images}`);
  console.log(`responses: ${codes}`);
  console.log(`received:  ${(s.bytes / 1024 / 1024).toFixed(1)} MB`);
  console.log(`time:      ${(s.ms / 1000).toFixed(1)} s (concurrency ${concurrency})`);
  process.exitCode = Object.keys(s.statuses).some((c) => c === "error" || Number(c) >= 400) ? 1 : 0;
}
```

With the server in production mode (`next build` and `next start`) and memory in view, run `node crawl-sitemap.mjs http://localhost:3000/sitemap.xml`, write down memory, then repeat with `--concurrency 2 --images`. This is the real output of that second command against my test server (13 URLs, one of them a deliberate 404); it exits with code 1 if there was an error or a 4xx/5xx:

```
404 http://localhost:4321/gone
pages:     13
images:    10
responses: 200 x 22, 404 x 1
received:  0.4 MB
time:      0.1 s (concurrency 2)
```

> **CAUTION**
>
> If memory climbs just the same on the crawl without `--images`, the optimizer is not (only) your problem, and switching it off will not fix it. And a local number does not predict production: your computer does not reproduce the glibc memory allocator Linux uses on Railway.

STEP 03

Decide if it suits you

YOUR CASE	UNOPTIMIZED?
Every image in `/public`, already WebP or AVIF at its final size	Yes
Large photos in `/public`	No, or convert them first
Remote images (CMS, bucket)	No, unless they already arrive at their final size
Images your users upload	No
You need a `srcset` for large photos on phones	No, or write the `srcset` by hand
A separate service already optimizes your images	Use a `next/image` loader instead
Memory grows just the same without `--images`	Not yet: look for another cause

If only some images gain nothing (SVGs, animated GIFs, under 1 KB), use the `unoptimized` prop on those and keep the optimizer for the rest.

STEP 04

The change, and how to roll it back

In `next.config.ts`, set `images: { unoptimized: true }` and delete whatever only served the optimizer (`formats`, `minimumCacheTTL`, `experimental.imgOptConcurrency`). Next to the option, write down the rule and when to undo it; in my repository it says: `shortcut: no srcset, so phones get the desktop-sized file; turn the optimizer back on if /public gains large photos.` Deploy, crawl production, read Metrics and, a week later, compare an average with an average.

To roll back: remove `unoptimized: true` (or revert the commit) and deploy. On Next.js 15.5, sharp ships as an optional dependency of `next`, so there is nothing to install. On Next.js 15 the default `minimumCacheTTL` is 60 seconds and `/public` is served with `max-age=0`, so every variant is re-encoded each minute: set a long value again. On Next.js 16 the default is already 4 hours.

WHEN YOU FINISH

Checklist

- [ ] You wrote down a one-week memory average from Railway Metrics.
- [ ] Every image comes from `/public`: no remote images and no user uploads.
- [ ] Every image is already WebP or AVIF, sized for its largest display.
- [ ] You crawled the sitemap without and with `--images` while measuring memory.
- [ ] After the deploy, you crawled production and read Metrics.
- [ ] You left a comment saying when to turn the optimizer back on.